Магдалина Тодорова

Структури от данни и програмиране на C++

Дигитализация на предоставения скан

Нова номерация и кликаемо съдържание

Пълният предоставен учебник е пренабран като избираем текст и код, със SVG схеми. Използван е и допълващият скан на страници 32–33. Само декоративните корици са запазени като изображения; надписите им са преписани отделно.

Сверени и оформени като текст: 334 оригинални страници. Векторните схеми са прерисувани по съответните оригинални страници. Непроверен OCR не се представя като готов текст.

Номерът в долния десен ъгъл е действителната позиция в този PDF, включително заглавната страница и съдържанието. Означенията „Оригинал“ пазят връзката с печатното издание. Номерата в съдържанието се изчисляват автоматично.

Сверката е визуална спрямо предоставените снимки. Допълнителните сканове на оригинални стр. 16, 71, 225, 241, 261, 263, 313 и 319 са сверени; предишните неясноти по краищата са разрешени. Корекцията на стр. 313 и историята на сверката са описани в придружаващите бележки. Авторски грешки и остарял C++ синтаксис са запазени и описани отделно. Кодът не е проверяван чрез компилация.

Сиела, София, 2011 · локално работно копие · без публикуване

Съдържание

Предговор 7

Глава 1. Структури от данни и алгоритми 9

1.1. Понятие за структури от данни и алгоритми 9

1.2. Величини и данни. Структура на величините 11

1.3. Класификация на структурите от данни 13

1.4. Оценка на алгоритми 17

Глава 2. Линейни структури от данни 27

2.1. Стек 27

2.2. Опашка 51

2.3. Линеен списък 72

2.4. Нехомогенни (хетерогенни) линейни структури от данни 126

Глава 3. Нелинейни структури от данни дърво и граф 151

3.1. Дърво 151

3.2. Двоично дърво 156

3.3. Двоично наредено дърво 177

3.4. Балансирани и идеално балансирани двоично наредени дървета 190

3.5. B-дървета 206

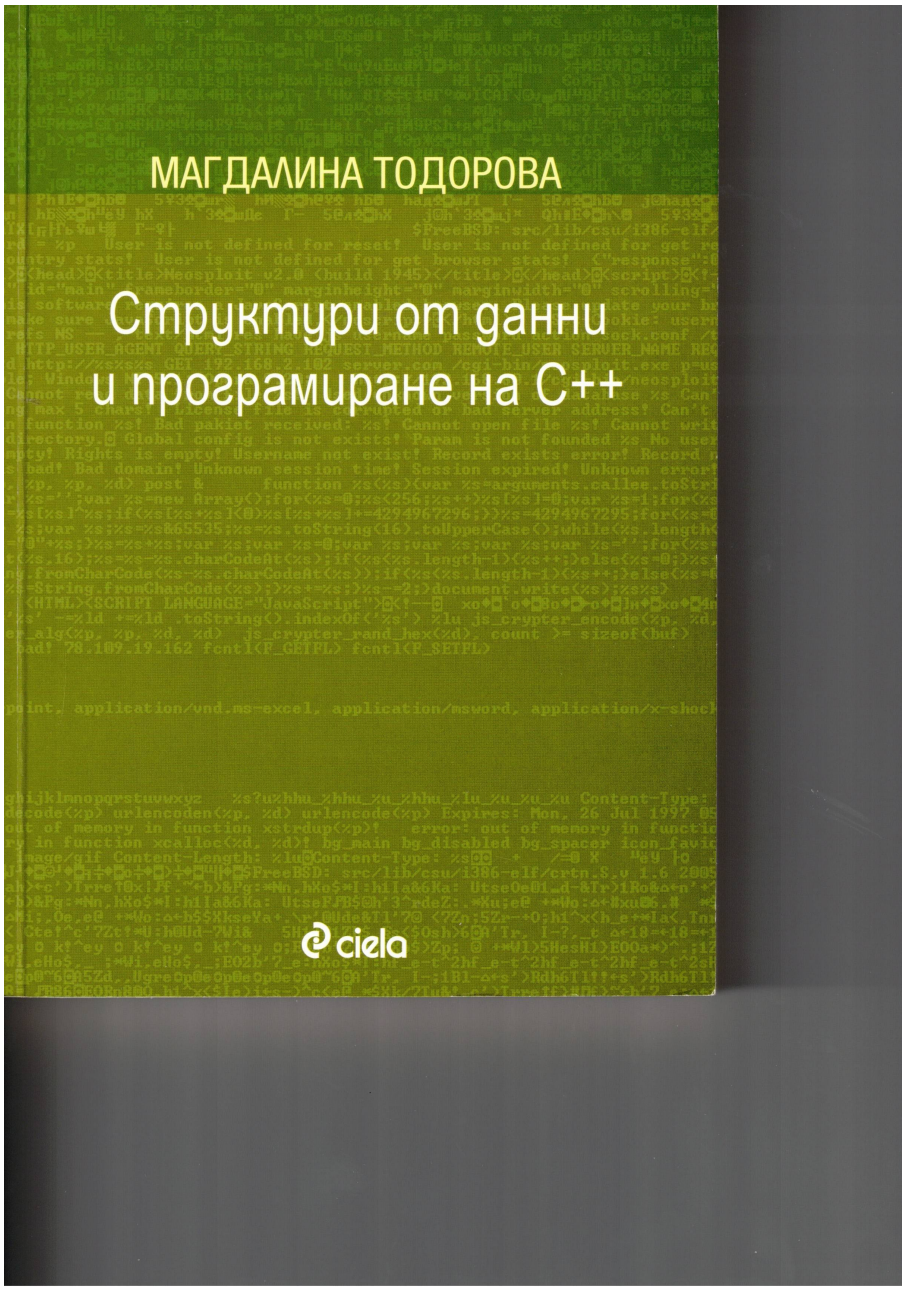
3.6. Граф 214

Глава 4. Алгоритми за сортиране и търсене 253

4.1. Алгоритми за сортиране 253

4.2. Алгоритми за търсене 282

Оригинал: предна корица



МАГДАЛИНА ТОДОРОВА

Структури от данни и програмиране на C++

ciela

Книгата е отпечатана на хартия от устойчиво управлявани гори от мрежата PEFC CHAIN OF CUSTODY.

© Магдалина Тодорова, *автор*

© Дамян Дамянов, *художник*

© Сиела Норма АД

София • 2011

ISBN: 978-954-28-0990-6

МАГДАЛИНА ТОДОРОВА

Структури от данни и програмиране на C++

ciela

Оригинал: стр. 4

Празна страница в оригинала.

Оригинал: стр. 5

Празна страница в оригинала.

Оригинал: стр. 6

СЪДЪРЖАНИЕ

Съдържанието на оригинална стр. 6 е преместено в началото на този PDF. Всички негови 21 позиции са запазени; номерата са преизчислени по новите PDF страници, а заглавията и номерата са кликаеми. Оригиналните страници са означени при самия текст.

ПРЕДГОВОР

Настоящата книга е замислена като уводен курс по структури от данни и програмиране на студентите от направление информатика и компютърни науки. Свързана е с книгата Програмиране на C++, част II, в която като приложение на създаването на шаблони на класове са разработени някои абстрактни типове данни, представящи динамичните структури стек, опашка, линеен списък, дърво и граф. Тя няма за цел да обяснява езика C++, а да разгледа по-детайлно някои от най-често приложимите в програмирането съставни структури от данни, а също и интересни техни приложения. В книгата се предполага, че читателят е запознат с основите на процедурното и обектно-ориентираното програмиране, че владее простите структури от данни: числови, знакови, логически; съставните структури масив и запис, както и основните алгоритми за работа с тях, предмет на обучението в уводните курсове по програмиране.

Съставянето на програмите и структурите от данни са неразривно свързани. Затова книгата започва с кратко въведение в структурите от данни и алгоритмите. Предложена е класификация на структурите от данни, разгледани са основните начини за оценка на алгоритми, както и някои основни правила за изчисление на времето за изпълнение на програма.

Във втора глава са дефинирани и реализирани чрез шаблони на класове на езика C++ линейните структури от данни: стек, опашка и линеен списък. За стека и опашката са предложени реализации на последователно и свързано физически представяния, а за линейния списък - свързани представяния с една и с две връзки, както и циклично свързано представяне с една връзка. Дискутирани са някои приложения на разгледаните линейни структури от данни като: намиране на стойност на аритметичен израз, елиминиране на рекурсията, реализация на някои функции на текстовите редактори чрез използване на стек, сортиране и сливане чрез опашки и линейни списъци, полиномна аритметика, операции над множества чрез използване на линейни списъци и др. Предложени са реализации на някои от често използваните функции от по-висок ред (accumulate, map и filter). Чрез пример е показано създаването на нехомогенни линейни структури от данни.

Някои йерархични и мрежови структури от данни са обект на разглеждане в третата глава на книгата. Дефинирани са структурите от данни: дърво, двоично дърво, двоично наредено дърво, балансирано и идеално балансирано двоично наредено дърво, B-дърво и граф. Направените дефиниции се състоят от логическо описание и физическо представяне на структурите. Разгледа-

ни са голяма част от използваните в практиката физически представяния. За структурите двоично дърво, двоично наредено дърво, балансирано двоично наредено дърво и граф са дефинирани шаблони на класове, които ги реализират. Предложени са интересни техни приложения като: представяне на аритметичен израз чрез двоично дърво и намиране на стойността му, кодиране и декодиране на съобщения, създаване и работа с балансирани и идеално балансирани двоично наредени дървета, обхождане и търсене на пътища в дърво и граф и др.

В последната глава на книгата са разгледани някои от широко използваните методи за сортиране и търсене. За едномерни масиви и файлове с пряка организация на достъпа са описани и реализирани методите за сортиране: пряка селекция, пряка размяна (метод на мехурчето и сортиране чрез клатене), вмъкване, метод на Шел, вариант на бързо сортиране, пирамидално сортиране, сливане и сортиране чрез сливане. За голяма част от тях е предложена и оценка на

ефективността им. За едномерни масиви и файлове с пряк достъп са предложени и реализации на методите за последователно и двоично търсене. В края на главата като техника за ефективно търсене, включване и изключване на елемент се разглежда хеширането. Реализирани са два от най-често използваните начини за разрешаване на колизиите - пряко свързване и отворено адресиране.

Изложението в книгата е неформално и богато илюстрирано с примери и решени задачи. В края на всяка глава е предложен голям набор от задачи за самостоятелна работа и проекти.

Оригинал: стр. 9

Глава 1

СТРУКТУРИ ОТ ДАННИ И АЛГОРИТМИ

1.1. ПОНЯТИЕ ЗА СТРУКТУРИ ОТ ДАННИ И АЛГОРИТМИ

„Алгоритми + структури от данни = програми“ - това е заглавието на книгата на знаменития швейцарски професор по компютърни науки, авторът на езиците Паскал, Модула-2, Оберон, Никлаус Вирт. С името на Вирт е свързан и структурният подход за програмиране. Двете съставки на програмите, определени от Вирт, са еднакво важни. Не само несъвършеният алгоритъм, но и неподходящата организация на работата със структурите от данни може да доведе до забавяне на работата на програмата десетки, а понякога стотици и повече пъти. От друга страна, владеенето на теорията и техниките за програмиране и умението те да се прилагат на практика, позволява бързо да се разработват ефективни и красиви програми.

Освен програмите и компютрите „се състоят“ от структури от данни и алгоритми. Вградените в хардуера структури от данни се съхраняват в двойчен вид, а вградените алгоритми описват правилата, по които запомнените в паметта на компютрите данни се интерпретират като команди, които трябва да бъдат изпълнени.

Задачите, които се решават с помощта на компютър, рядко се изразяват на езика на битовете. Като правило данните са във вид на числа, символи, символни низове и по-сложни структури като масиви, списъци, дървета, графи и др. Освен това за решаването на различни задачи се прилага голямо разнообразие на алгоритми - не по-малко от разнообразието на задачите за решаване.

За точното описание на структурите от данни и алгоритмите се използват езиците за програмиране. Последните са системи за формално означаване, в които смисълът на всяко твърдение се определя точно и еднозначно. Сред средствата, които предлагат почти всички езици за програмиране, е възможността за достъп до данни чрез свързаните с тях имена (идентификатори). Някои от именуваните величини са константи. Те съхраняват постоянни стойности - стойности, които не се променят в частта от програмата, където са дефинирани. Други величини са променливи. Тези величини могат да се променят, като им се присвоят нови стойности. Но докато програмата не започне да се изпълнява, стойностите на величините са неопределени.

Оригинал: стр. 10

Имената на величините, константи и променливи, помагат на програмистите, но за компютъра те не означават нищо. Компилаторът транслира текста на програмата в двоичен код, свързва всеки идентификатор с определен адрес от паметта, но за да може да продължи работата си, той се нуждае от информация за типа на всеки идентификатор.

Типовете данни в езиците за програмиране включват естествените и целите числа, реалните числа, символите, низовете и др. В някои езици за програмиране (тези с неявно или нестрого типизиране) типовете на величините се определят от компилатора по стойностите, с които величините се свързват. Например наличието на десетична точка, определя стойността и съответната величина за реална. В други езици (тези с явно или строго типизиране) се изисква програмистът явно да укаже тип за всяка величина.

Явното типизиране има някои важни предимства. Две от тях са:

- типовете данни подпомагат компилатора, за да разпредели паметта;

- наличието на типовете позволява на компилатора да открива грешки, свързани с несъответствието на типа на аргументите при някой операции.

В зависимост от предназначението на езика за програмиране защитата на типовете, осъществявана по време на компилация на програмата може да е в по-голяма или по-малка степен. Например езикът Паскал, едно от първоначалните предназначения на който беше да илюстрира структурите от данни и алгоритмите, реализира строга защита на типовете. Смесването в израз на данни от различни типове или прилагането към тип данни на несвойствени за типа операции, причиняват съобщаване за фатална грешка от компилаторите на Паскал. Езикът С, предназначен главно за системно програмиране, е със слаба защита на типовете. В горните случаи неговите компилатори издават само предупреждения. Слабата защита на типовете дава допълнителни възможности на програмистите, но в този случай програмистите отговарят за правилността на действията си.

Структура от данни - това е програмна единица, която позволява да се съхранява и обработва множество от еднотипни и/или логически свързани данни чрез компютър. По-точно всяка величина (константа или променлива), определена в програма, се нарича структура от данни.

Неформално всяка структура от данни е конкретен представител на съответен тип данни.

В рамките на тази книга ще използваме интуитивната дефиниция на понятието алгоритъм.

Алгоритъм, от името на Ал-Хорезми, е точен набор от инструкции, описващи реда на действията на изпълнителя за достигане до резултата от решението

Оригинал: стр. 11

на задачата за крайно време. В по-старата трактовка на понятието вместо думата „ред“ се използваше „редица“. Причината за промяната и използването на по-общата дума е, че работата на някои инструкции на алгоритъма може да зависи от други инструкции или от резултата от тяхната работа. Други инструкции може да са независими или да станат независими след завършване на изпълнението им. Тези инструкции могат да се изпълняват в произволен порядък, паралелно или едновременно, ако това се позволява от използваните процесор и операционна система.

Първите алгоритми са били за решаване на числови задачи - за извършване на аритметични действия с числа, за намиране стойностите на тригонометрични функции и др. Сега са важни както алгоритмите за решаване на числови задачи, така и алгоритмите за решаване на нечислови задачи.

Изборът на подходящи алгоритми и структури от данни е от голяма важност за създаването на качествени и високопроизводителни програми.

Тъй като структурите от данни са величини, в следващата секция ще дадем някой дефиниции, определящи величините и тяхната структура.

1.2. ВЕЛИЧИНИ И ДАННИ. СТРУКТУРА НА ВЕЛИЧИНИТЕ

Величини

Величината е абстрактен информационен обект, чрез който се изразяват определени свойства на реални обекти. Характеризира се с име и тип. Името се използва за назоваване на величината и обикновено пояснява същността ѝ, а типът представлява множество от допустими стойности, с които е свързана определена съвкупност от операции, приложими над тези стойности.

Величина, която не променя стойността си, се нарича постоянна или константа. В противен случай тя е променлива.

Величините могат да бъдат прости (елементарни) и съставни.

Проста е такава величина, която не може да се разложи на по-малки части, за които има смисъл да бъдат разглеждани самостоятелно. Такива в езиците от високо ниво са например числата, знаците (символите), логическите стойности „истина“ и „лъжа“ и др.

Съставна е такава величина, която не е проста, т.е. може да се разложи на по-малки части, които могат да бъдат разглеждани самостоятелно. Такива са например низът, масивът, записът и др.

Оригинал: стр. 12

Данни

Информацията, с която работят различните програми, обикновено се нарича данни. Терминът е широко използван. С него се означават различни понятия. Понякога под данни се разбира съвкупността от сведения за определен обект. Под данни също се разбира множество от измерени стойности на определена величина, записани по стандартен начин във вид, готов за някаква обработка. В информатиката понятието данна се разглежда като стойност на величина, представена по определени правила. В този смисъл данните нямат име, но се характеризират с тип и формат.

Всеки *тип на данни T* (ще казваме също *тип данни*) се определя с две множества: D_T - множество от стойности (константи на типа) и O_T - множество от операции на типа. Накратко се записва по следния начин $T = (D_T, O_T)$.

Пример. Нека D_T е множеството на целите числа от интервала $[-2^{31}; 2^{31}-1]$, а O_T = {+, -, *, /, %, <, <=, ==, !=, >, >=}. Полученият тип в езика C++ е известен като *int*.

Пример. Нека D_T = *{false, true}*, а O_T = {!, &&, ||, <, <=, ==, !=, >, >=}. Полученият тип в езика C++ е известен като *bool*.

Езиците за програмиране предлагат голямо разнообразие от предварително определени типове данни, наречени вградени типове данни. Тези типове могат да се използват, без да е необходимо предварително да се определят в програмите. В C++ стандартни типове данни са: bool, short, unsigned short, long int, float, double, char и др.

Форматът на данните задава начина, по който данните се представят в паметта на компютъра. По-точно форматът задава представянето на константите на типа данни в паметта на компютъра.

Пример. Булевите данни {false, true} се представят чрез двоичната азбука {0, 1}, като 0 означава false, а 1 - true. В езика C++ се използва двоичен формат от 1 байт без знак.

Пример. Целите константи на типа данни int в езика C++ за реализацията Microsoft Visual C++ (6.0, 2005 и др.) се представят в двоична позиционна система с 4 байта със знак.

Във всички езици с явно типизиране, в частност в езика C++, всяка променлива величина трябва предварително да се свърже с определен тип. Това се осъществява със съответна декларация.

Пример. Следващите декларации са на езика C++.

Оригинал: стр. 13

C013-01 · Копиране: code.html

```
int i, j;
double x, y;
int a[100];
```

В този пример *i* и *j* са имена на променливи величини от тип *int*. Последното означава, че допустимите стойности, с които те могат да се свържат, са целите числа от някакъв интервал, например [-2^{31}, 2^{31}-1] за реализацията *Microsoft Visual C++* (6.0, 2005 и др.) и над тях могат да се изпълняват операциите +, -, *, /, %, <, <=, ==, !=, >, >=. С *x* и *y* са означени имената на променливи величини от тип *double*. Последното означава, че допустимите стойности, с които те могат да се свържат са реалните числа от множеството от стойности на типа *double* и над тях могат да се изпълняват операциите +, -, *, /, <, <=, ==, !=, >, >=. С *a* е означено името на променлива величина от тип *int[100]*, определящ тип масив със 100 цели компоненти. С *a* може да се свърже всяка редица от 100 цели числа, а над индексираните променливи, свързани с *a*, може да се изпълнят всички операции, характерни за типа *int*.

До момента говорихме за данните като за някаква аморфна маса, лишена от каквато и да е вътрешна организация. На практика в голям брой от случаите данните изграждат една или друга структура.

Структура на величините

Структурата на величините се определя на логическо и физическо ниво. На логическо ниво всяка величина се задава чрез име и тип данни. На физическо ниво структурата се определя по време на компилация. На всяка декларирана величина компилаторът съпоставя определени фрагменти от паметта, наречени структури за съхранение или структури на паметта. Най-общо това са фрагменти от паметта, които физически представят величините на машинно ниво. Достъпът до тези фрагменти се осъществява чрез съответен адрес.

Тези две нива за определяне на структурата на величините задават начина за дефиниране на структурата:

- логическо описание и

- физическо представяне.

Логическото описание определя структурата от данни на базата на декомпозицията на константите на типа данни на структурата на съставящите ги компоненти и на декомпозиция на операциите над константите на типа данни на структурата на съответни операции над съставящите ги компоненти.

Физическото представяне дава методи за представяне на структурата в паметта на компютъра, а също определя начина за реализиране на операциите над структурата в конкретната среда за

програмиране.

Оригинал: стр. 14

1.3. КЛАСИФИКАЦИЯ НА СТРУКТУРИТЕ ОТ ДАННИ

Различават се прости (скаларни, примитивни) и съставни (структурирани, сложни) структури от данни.

1.3.1. Прости структури от данни

Прости са тези структури от данни, които не могат да бъдат разложени на съставни части, за които има смисъл да се разглеждат самостоятелно. Такива са числовите, логическите (булевите), знаковите (символните), изброените, ограничените и други структури от данни.

Числови структури от данни

Най-често се използват два типа числови структури от данни: цели и реални.

Цели структури от данни

Целите константи принадлежат на крайно подмножество на целите числа от математиката, чиито размери зависят от конкретната изчислителна система и реализация на езика за програмиране. Всички операции с цели структури от данни са точни и отговарят на общоприетите правила в аритметиката. Освен това ако резултатът от изпълнението на операция се окаже извън множеството от допустими за типа стойности, се издава съобщение или предупреждение за грешка по време на компилация. Основните операции са четирите аритметични действия: събиране (+), изваждане (-), умножение (*) и деление (/ за езика С++), а също намирането на остатъка от целочисленото деление (% за езика С++). Други операции са присвояването (= за езика С ++) и сравненията за: равно (== за езика С++), различно (!= за езика С++), по-голямо (>), по-голямо или равно (>=), по-малко (<), по-малко или равно (<=).

Реални структури от данни

Реалните константи принадлежат на подмножество на реалните числа от математиката, чиито размери зависят от конкретната изчислителна система и реализация на езика за програмиране. Докато аритметиката с цели числа винаги води до точни резултати, при аритметиката с реални структури от данни може да възникне грешка, която трябва да е в допустими граници. Операциите с реални структури от данни се подчиняват на общоприетите правила в аритметиката. Основните операции са четирите аритметични действия: събиране (+), изваждане (-), умножение (*) и деление (/). Други операции са присвояването (= за езика С++) и сравненията (==, !=, >, >=, <, <= за езика С++).

Оригинал: стр. 15

Логически структури от данни

Логическите константи са true и false. Основните операции са известните от математическата логика: конюнкция (and, && за езика С++), дизюнкция (or, || за езика С++) и отрицание (not, ! за езика С++). Други операции са присвояването (= за езика С++) и сравненията (==, !=, >, >=, <, <= за езика С++).

Ще отбележим, че резултатът от прилагането на операциите за сравнение над числови данни е логическа данна.

Знакови (символни) структури от данни

Знаковите константи принадлежат на набор от знаци, дефиниран от Международната организация за стандарти и обикновено на неговата американска версия ASCII (American Standard Code for Information Interchange - американски стандартен код за обмен на информация). Този набор се състои от 256 различни знака, наредени по определен начин и включва главните и малките латински букви, арабските цифри, известен брой графични знаци (препинателни, за отношения, за аритметични операции и др.), а също съвкупност от управляващи знаци. Стойностите на знаковите структури заемат 1 байт памет. Кодовете от 0 до 255 задават символите от ASCII таблицата. ASCII кодирането не е единствено възможното. Друго достатъчно широко използвано множество е EBCDIC (Extended Binary Coded Decimal Interchange Code). В EBCDIC кодирането всеки символ също заема 1 байт, но с различни кодове спрямо ASCII. В двете кодирания са включени само символите на латинската азбука. За символите от националните азбуки са оставени свободни места. Така тези кодирания поддържат само една национална азбука. Този недостатък е преодолян от кодирането UNICODE. В UNICODE всеки символ се кодира с цяло число, заемащо два байта. Това кодиране осигурява повече от 64 хиляди възможни кодови комбинации и дава възможност за единна таблица за кодиране, включваща всички национални азбуки.

Специфични операции за знаковите структури от данни са присвояването (= за езика C++) и сравненията (==, !=, >, >=, <, <= за езика C++).

Изброени структури от данни

Константите на изброените структури от данни са идентификатори, например (rectangle, square, ellipse, circle) или (in, out, app, ate, trunc, binary). Допустими операции са присвояването (= за езика C++), сравненията (==, !=, >, >=, <, <= за езика C++) и др.

Оригинал: стр. 16

Указатели

Константите на типа указател са адреси на клетки на паметта (в голяма част от съвременните изчислителни системи минималната адресируема единица памет е 1 байт). При езиците за програмиране на ниско ниво (Асемблери, C и др.) работата с адреси е значителната част от програмния код. При решение на приложни задачи с използване на езици от високо ниво, най-често срещаните случаи, когато на програмистите се налага да използват указатели са:

1) При необходимост едни и същи физически данни да се представят като данни с различна логическа структура. За целта се дефинират два или повече указателя, които адресират една и съща област от паметта, но са от различни типове.

2) При работа с динамични структури от данни (т. 1.3.2). Паметта за динамичните структури от данни се отделя в хода на изпълнение на програмата. Стандартните процедури (функции или оператори) за отделяне на памет връщат адреса на отделената памет (указател към нея). Към съдържанието на динамично отделената памет, програмистът може да се обръща само чрез указатели.

1.3.2. Съставни структури от данни

Структурите от данни, които не са прости, са съставни. Те се състоят от компоненти, които са други структури от данни (прости или съставни) и релация, която определя отношенията между компонентите. Например структурите от данни масив, запис, таблица, множество, низ, стек, опашка, дек, списък, дърво, граф и др. са съставни структури от данни.

Хомогенни и нехомогенни структури от данни

Ако всички компоненти на съставна структура са от един и същ тип, се казва, че структурата е хомогенна, а типът на компонентите й определя нейния базов тип данни. В противен случай, т.е. ако компонентите на структурата са от различни типове, структурата е нехомогенна (хетерогенна). Например структурата от данни масив е хомогенна, а структурата от данни запис е хетерогенна.

Статични и динамични структури от данни

Важна характеристика на съставните структури от данни е променливостта им, т.е. промяната на броя на компонентите и (или) връзките между компонентите на структурите по време на изпълнението на програмата. По тази характеристика се различават статични и динамични структури от данни. Статичните структури от данни не променят броя на компонентите и (или)

Оригинал: стр. 17

връзките между тях по време на изпълнение на програмата, в която се използват. Динамичните структури ги променят. Характерно за последните е, че те се създават по време на изпълнение на програмата. Например масивът, записът, низът са статични структури, а линейният списък, стекът, опашката, декът, графът, дървото и др. са динамични структури от данни.

Ще направим кратко описание на добре известните от уводните курсове по програмиране структури от данни масив и запис. В следващата дефиниция на масив нямаме предвид динамичен масив.

Масив

Логическо описание

Масивът е хомогенна, статична съставна структура от данни. Състои се от фиксиран брой компоненти (елементи), достъпът до които е пряк и се осъществява чрез указване на индекс, задаващ позицията на съответната компонента. Компонентите на масив се наричат още негови елементи.

Броят на елементите на масива не може да се променя по време на изпълнението на програмата, в която масивът е дефиниран. Тъй като масивът е хомогенна и статична структура от данни количеството памет, необходима за него се определя и заделя още по време на компилация на програмата.

Физическо представяне

Елементите на масива се съхраняват в последователни фрагменти на паметта. Това позволява да се пресметне адресът на всеки елемент на масива. Ако a е масив, деклариран чрез

C017-01 · Копиране: code.html
```
T a[size];
```

адресът на елемента *a[i]* ($0 \le i < size$) е равен на *address(a[0]) + i*sizeof(T)*.

Запис

Логическо описание

Записът е нехомогенна, статична съставна структура от данни. Състои се фиксиран брой компоненти (елементи), достъпът до които е пряк и се осъществява чрез указване на името на съответната компонента. Компонентите на записите се наричат техни членове или полета. В езика C++ записите се наричат структури, тъй като се реализират като такива.

Физическо представяне

Елементите на запис се съхраняват в последователни фрагменти на паметта. Ако p е структура от данни от тип point

Оригинал: стр. 18

C018-01 · Копиране: code.html
```
struct point
{ double x, y;
};
point p;
```

p притежава две реални полета: *p.x* и *p.y*. За *p* се отделя памет с размер *2*sizeof(double)*, която памет се именува с *p.x* и *p.y*.

Линейни и нелинейни структури от данни

Важна характеристика на съставните структури от данни е характерът на наредеността на техните елементи. Според тази характеристика те могат да се разделят на линейни и нелинейни.

Една съставна структура от данни е линейна, ако е или празна, или не е празна, но в този случай за нея са в сила:

- структурата притежава точно един елемент, който е първи и точно един елемент, който е последен;

- за всеки елемент, без първия, съществува точно един елемент, който го предхожда;

- за всеки елемент, без последния, съществува точно един елемент, който го следва.

В противен случай структурата е нелинейна.

Основни операции за линейните структури са: създаване; достъп до първия, до последния, до следващ или предходен елемент; актуализиране на елемент; включване и изключване на елемент; обхождане на структурата; проверка на свойства.

Линейни структури от данни са: масивът, линейният списък, стекът, опашката, декът и др.

Нелинейни съставни структури от данни са: дърветата, графите, множествата и др.

Дървета

Дърветата се наричат още йерархични структури от данни. Елементите на структурата от данни дърво се наричат върхове или възли.

Дървото е хомогенна съставна структура от данни, която или е празна, или не е празна, но в този случай за нея са в сила:

- в структурата има един връх, наречен корен;

- всеки връх без корена има единствен предшественик;

- всеки връх може да има нула, един или повече върхове, които го следват. Наричат се негови наследници.

Оригинал: стр. 19

Основни операции за дървета са: създаване; достъп до корен, до предшественик и до наследник; актуализиране на връх, включване и изключване на връх; обхождане на структурата; проверка на свойства.

Графи

Графът е хомогенна съставна структура от данни, задаваща се чрез множество от върхове и множество от дъги, определящи връзките между елементите на структурата. Структурата е или празна, или не е празна, но в този случай за нея са в сила:

- всеки връх може да има нула, един или повече предшественика;

- всеки връх може да има нула, един или повече наследника.

Основни операции за граф са: създаване, достъп до връх, актуализиране на връх, включване и изключване на връх, включване и изключване на дъга; проверка на свойства; търсене на път между върхове.

Множества

Множеството е хомогенна съставна структура от данни, за елементите на която е дефинирана операция, определяща дали елемент принадлежи на множеството.

Основни операции за работа с множества са: създаване, достъп до елемент (проверка дали елемент принадлежи на множество), включване и изключване на елемент, обединение, сечение и разлика на множества, проверка на свойства.

1.4. ОЦЕНКА НА АЛГОРИТМИ

За решаването на повечето проблеми има много различни алгоритми. Изборът на алгоритъм е от изключителна важност, тъй като определя ефективността на програмата.

Ефективността на програма има две съставящи: памет (пространство) и време.

Пространствената ефективност се измерва с количеството памет, необходима за изпълнение на програмата. Нарича се още пространствена сложност. Компютрите имат ограничено количество памет. Ако две програми реализират идентични функции, програмата, която използва по-малко количество памет, е с по-голяма пространствена ефективност.

Времевата ефективност определя времето, което е необходимо за изпълнение на програмата. Нарича се още времева сложност.

Оригинал: стр. 20

Най-добрият начин за сравнение на ефективността на алгоритмите се състои в съпоставяне на порядъците на техните сложности, както по отношение на необходимото време, така и по отношение на необходимата памет.

Понякога паметта е доминиращ фактор в оценката на ефективността. През последните години заради бързото поевтиняването на паметта значението на пространствената ефективност постепенно намалява.

В настоящото изложение за оценяване на алгоритмите ще използваме само техните времеви сложности (или само сложности). Най-често се използват следните видове (класове) сложности:

Име	Математическо означение	Пример за алгоритъм със съответната сложност
факториелна	n!	алгоритми от комбинаториката
експоненциална	k^n	алгоритми на грубата сила
полиномиална	n^k	алгоритми за проста сортировка (bubble sort, select sort, insert sort и др. имат сложност n^2)
линейно логаритмична	n.log(n)	алгоритъм за бързо сортиране
логаритмична	k.log(n)	алгоритъм за двоично търсене
линейна	k.n	алгоритъм за линейно търсене
константна	k	обръщение към елементите на масив по указан индекс

където n е размерът на входа (количеството на обработваните данни), а k е константа.

При алгоритми, обработващи редици от елементи, дължината на редицата n е размерът на входа. При алгоритми, обработващи матрици с размерност nxm, размерът на входа е двойката (n, m).

Нека размерът на входа на алгоритъм е n. Формално сложността на алгоритъма е функция, която изразява зависимостта между размера на входа на алгоритъма n и времето, необходимо за неговото изпълнение. В литературата тази функция се означава с T. За измерване на сложността се използват някои нотации, дефинирани по следния начин:

Дефиниция 1. С $O(f(n))$ се означава множеството от функции $g(n)$, за които съществуват константите c $(c > 0)$ и $n_0(c)$ - естествено число, такива че $0 \leq g(n) \leq c.f(n)$ е в сила, за всяко $n \geq n_0$.

Оригинал: стр. 21

Дефиниция 2. Ако времето за изпълнение на програма има стойност $O(f(n))$, се казва, че програмата има порядък или степен на растене или сложност $f(n)$.

Пример. Функцията $T(n) = 2n^3 + 3n^2$ има порядък n^3, т.е. принадлежи на $O(n^3)$. За да докажем това, ще изберем $n_0 = 0$ и $c = 5$. Тогава лесно следва, че $0 \leq 2n^3 + 3n^2 \leq 5n^3$ е в сила за всяко $n \geq 0$.

Дефиниция 3. С $\Omega(f(n))$ се означава множеството от функции $g(n)$, за които съществуват константите c $(c > 0)$ и $n_0(c)$ - естествено число, такива че $g(n) \geq c.f(n) \geq 0$ е в сила, за всяко $n \geq n_0$.

Пример. За да проверим, че функцията $T(n) = 2n^3 + 3n^2$ принадлежи на $\Omega(n^3)$, ще изберем $n_0 = 0$ и $c = 1$. Тогава $2n^3 + 3n^2 \geq n^3 \geq 0$ е в сила, за всяко $n \geq 0$.

Примери.
$5n \in O(n)$; $5n \in \Omega(n)$;
$5n \in O(n^2)$; $5n \in \Omega(1)$;
$5n^2+3n+1 \in O(n^2)$; $5n^2+3n+1 \in \Omega(n^2)$;
$\sqrt{n} \in O(n)$; $\sqrt{n} \in \Omega(\log n)$;
$n^2 \in O(n^3)$; $n^3 \in \Omega(n^2)$.

Дефиниция 4. С $\Theta(f(n))$ се означава множеството от функции $g(n)$, за които съществуват константите c_1 $(c_1 > 0)$, c_2 $(c_2 > 0)$ и естествено число $n_0(c_1, c_2)$, такива че $0 \leq c_1.f(n) \leq g(n) \leq c_2.f(n)$ е в сила, за всяко $n \geq n_0$, т.е. $\Theta(f) = O(f) \cap \Omega(f)$.

Пример. $5n \in \Theta(n)$; $5n^2+3n+1 \in \Theta(n^2)$.

Освен нотациите O, Ω и Θ макар и по-рядко се използват и нотациите o и ω. В книгата за оценка на сложността ще използваме нотацията O, макар че напоследък се предпочита по-точната оценка Θ.

Някой свойства на нотацията O

• *Рефлексивност:* $f \in O(f)$
• *Транзитивност:* $f \in O(g)$ *и* $g \in O(h) \Rightarrow f \in O(h)$
• *Константите могат да бъдат игнорирани: ако* k $(k > 0)$ *е константа, то* $k.f \in O(f)$
• *Ако* $f \in O(h)$ *и* $g \in O(h)$, *тогава* $f + g \in O(h)$
• *Ако* $f \in O(h)$ *и* $g \in O(p)$, *тогава* $f + g \in max\{O(h), O(p)\}$
• *Ако* $f \in O(h)$ *и* $g \in O(p)$, *тогава* $f.g \in O(h.p)$
• *Ако* $f(n)$ *е полином от степен* m *или по-ниска, тогава* $f \in O(n^m)$
• *Ако* $f(n) = k.g(n)$, *то* $f \in O(g)$.

Оригинал: стр. 22

За да се намери сложността O-голямо, трябва да се определи колко пъти размерът на входа n се „разглежда“. Разглеждането може да е сравняване, създаване или изтриване на стойност и др. От дефиницията и свойствата на нотацията O следва, че анализът O-голямо дава асимптотичното време за изпълнение на алгоритъма, т.е. границата на времето за изпълнение, когато n расте неограничено. Така, ако алгоритъмът се изпълнява за време $n+5$ или $5.n$, анализът O-голямо определя времето за изпълнение като $O(n)$, т.е. времето за изпълнение е от порядъка на n. Ако времето за изпълнение е n^2+n, приема се, че алгоритъмът е $O(n^2)$, тъй като разликата между n^2 и n^2+n е пренебрежима при големи n.

Операции за събиране и умножение с използване на анализа О-голямо

1) операция събиране

Нека $T_1(n)$ и $T_2(n)$ са времената за изпълнение на програмните фрагменти P_1 и P_2 и $T_1(n) \in O(f(n))$, а $T_2(n) \in O(g(n))$. Тогава сумата $T_1(n)+T_2(n)$, т.е. времето за последователното изпълнение на фрагментите P_1 и P_2 е равно на $O(max(f(n), g(n)))$. Доказателството следва от дефиниция 1.

Пример. Нека три последователни програмни фрагмента P_1, P_2 и P_3 имат времена за изпълнение съответно $O(n^2)$, $O(n^3)$ и $O(n \log n)$. Тогава времето за последователното изпълнение на първите два фрагмента е $O(max(n^2, n^3))$, т.е. $O(n^3)$, а времето за изпълнението на трите фрагмента е $O(max(n^3, n \log n))$, т.е. $O(n^3)$.

Пример. Времето за изпълнението на фрагмента

C022-01 · Копиране: code.html
```
for (int i = 1; i <= n; i++)
  a[i] = 5;                    // T1(n) = O(n)
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
    for (int k = 1; k <= n; k++)
      a[j] = a[i]*a[j] + 5*a[k]; // T2(n) = O(n*n*n)
```

е равно на $max\ \{ O(n), O(n^3) \} = O(n^3)$.

2) операция умножение

Нека $T_1(n)$ и $T_2(n)$ са времената за изпълнение на програмните фрагменти P_1 и P_2 и $T_1(n) \in O(f(n))$, а $T_2(n) \in O(g(n))$. Тогава произведението $T_1(n).T_2(n) \in O(f(n).g(n))$.

Доказателството също следва от дефиниция 1.

Оригинал: стр. 23

Пример. Нека да намерим времето за изпълнение на функцията, която сортира във възходящ ред елементите на едномерен масив от цели числа по метода на мехурчето.

C023-01 · Копиране: code.html
```
void bubbleSort(int* a, int n)
{ int i, j, x;
  for(i = 0; i <= n-2; i++)       // (1)
    for(j = n-1; j >= i+1; j--)   // (2)
      if(a[j-1] > a[j])          // (3)
        { x = a[j];              // (4)
          a[j] = a[j-1];         // (5)
          a[j-1] = x;            // (6)
        }
}
```

За мярка за обема на входните данни ще служи броят n на елементите, които подлежат на сортиране. Всички оператори за присвояване имат някакво постоянно време за изпълнение, което не зависи от размера на входните данни (операторите (4), (5) и (6) имат време за изпълнение O(1), т.е. от порядъка на константа). Съгласно правилото за събиране, времето за изпълнение на фрагмента (4)-(6) е равно на O(max(1, 1, 1)) = O(1). Сега трябва да пресметнем времето за изпълнение на операторите за цикъл и условния оператор. Тези оператори са вложени един в друг. Затова, започвайки от най-вътрешния към най-външния последователно, ще определим времето за изпълнението им. Проверката на условието в оператора if се осъществява за време O(1). Не е известно дали ще се изпълнят операторите в тялото на условния оператор, но тъй като се търси най-лошото време за изпълнение, то предполагаме, че ще се изпълнят. Така времето за изпълнение на фрагмента (3)-(6) е O(1). Намирането на времето за изпълнение на оператор за

цикъл става, като се сумират времената за изпълнение на всяка итерация на цикъла. За операторите от (2)-(6) времето за изпълнение на всяка итерация е O(1). Вложеният оператор за цикъл се изпълнява n-i-1 пъти. По правилото за умножение общото време за изпълнени на този оператор за цикъл е O((n- i-1).1), т.е. O(n-i-1). Външният оператор за цикъл, означен с (1), се изпълнява n-1 пъти. Сумарното време за изпълнение на функцията bubbleSort е:

$\sum_{i=0}^{n-2} (n-i-1) = n(n-1)/2$, т.е. $O(n^2)$.

Оригинал: стр. 24

Правила за изчисление на времето за изпълнение на програма

От теоретична гледна точка намирането на времето за изпълнение на програма е сложна математическа задача. На практика намирането на времето за изпълнение се осъществява, като се използват няколко основни правила.

1) Времето за изпълнение на операторите за присвояване, въвеждане и извеждане обикновено е O(1). Изключения от това правило са свързани с оператора за присвояване, когато се присвояват обръщения към функции и масиви.

2) Времето за изпълнение на редица (последователност) от оператори се определя чрез правилото за сумиране и е равно на най-голямото време за изпълнение на оператор от последователността.

3) Времето за изпълнение на условен оператор if (B) S1; else S2; е равно на сумата на времето за проверка на условието и по-голямото от времената за изпълнение на операторите S1 и S2.

4) Времето за изпълнение на оператор за цикъл е равно на сумата от времената за изпълнение на всички изпълними итерации на цикъла и времето за проверка на условието за прекъсване на изпълнението на цикъла (обикновено последното е от порядъка на константа). Най-често времето за изпълнение на оператор за цикъл се изчислява, като се пренебрегват константите за пропорционалност.

Пример. Времето за изпълнение на оператора

C024-01 · Копиране: code.html
```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++) p++;
```

е $O(n^2)$.

Пример. Времето за изпълнение на програмния фрагмент

C024-02 · Копиране: code.html
```
int s = 0;
for (int i = 1; i <= n; i *= 2) s++;
```

е $O(\log_2 n)$. Причината е, че променливата *i* получава стойностите 1, 2, 4, 8, ..., докато достигне *n*.

5) Времето за изпълнение на програми, които съдържат обръщения към процедури и функции, сред които няма рекурсивни, може да се пресметне по следния начин: последователно се намират времената за изпълнение на процедурите и функциите. Започва се от тази, която не извършва обръщения към други процедури или функции. След това може да се определи времето за изпълнение на процедурата (функ-

Оригинал: стр. 25

цията), която извиква тази процедура, като за целта се използва вече изчисленото време за изпълнение на извиканата процедура. Този процес продължава до намиране на времето за изпълнение на всички процедури (функции). Най-накрая се намира времето за изпълнение на програмата.

6) Времето за изпълнение на програми, които съдържат обръщения към процедури и функции, сред които има рекурсивни, може да се пресметне по аналогичен начин на описания на по-горната стъпка. Разликата е в намирането на времето за изпълнение на рекурсивните процедури (функции). С всяка рекурсивна процедура (функция) се свързва функция *T(n)*, където *n* определя обема на аргументите на процедурата (функцията). След това за *T(n)* се намира рекурентно съотношение, т.е. уравнение или неравенство за *T(n)*, в което участват стойности на *T(k)* за различни значения за *k*. Техниката за решение на рекурентните съотношения зависят от вида им.

Пример. Ще намерим времето за изпълнение на функцията за намиране на факториела на естественото число *n*, дефинирана рекурсивно по следния начин:

C025-01 · Копиране: code.html
```
int fact (int n)
{ if (n <= 1)          // (1)
    return 1;         // (2)
  else
    return n*fact(n-1); // (3)
}
```

Нека времето за изпълнение на функцията *fact* е *T(n)*. Времето за изпълнение на операторите от (1) и (2) е *O(1)*, а времето за изпълнение на (3) е *O(1) + T(n-1)*. Следователно

$$T(n) = \begin{cases} c + T(n-1), & \text{ако } n > 1 \\ d, & \text{ако } n \leq 1 \end{cases}$$

е в сила за някакви константи *c* и *d*. От тази рекурентна зависимост може да се получат следните уравнения: *T(n) = 2.c + T(n-2)*, при *n > 2*; *T(n) = 3.c + T(n-3)*, при *n > 3*; ...; *T(n) = i.c + T(n-i)*, при *n > i*. Ако в последния израз положим *i=n-1*, получаваме: *T(n) = c.(n-1) + T(1) = c.(n-1) + d*. Следователно *T(n)* е *O(n)*.

Още примери за намиране на времето за изпълнение на функции са дадени в глава 4.

Оригинал: стр. 26

ЗАДАЧИ

1. Докажете, че са в сила следните твърдения:
а) 5 принадлежи на O(1);
б) n(n-1)/3 принадлежи на $O(n^2)$;
в) $\sum_{i=1}^{n} i^k$ принадлежи на $O(n^{k+1})$ и $\Omega(n^{k+1})$, за положителни цели стойности на k;
г) ако p(x) е полином от степен k с положителен старши коефициент, то p(x) принадлежи на $O(n^k)$ и $\Omega(n^k)$.

2. Разгледайте следните дефиниции на функции:

$f_1(n) = n^2$;
$f_2(n) = n^2 + 500n + 150$;

$$f_3(n) = \begin{cases} n, & \text{ако n е нечетно} \\ n^3, & \text{ако n е четно} \end{cases}$$

$$f_4(n) = \begin{cases} n + 3, & \text{ако } n \leq 300 \\ n^3 + n^2 + 7, & \text{ако } n > 300 \end{cases}$$

Укажете за кои двойки функции $f_i(n)$ и $f_j(n)$ е в сила:
$f_i(n)$ принадлежи на $O(f_j(n))$;
$f_i(n)$ принадлежи на $\Omega(f_j(n))$;
$f_i(n)$ принадлежи на $\Theta(f_j(n))$.

3. Разгледайте следните дефиниции на функции:

$$g_1(n) = \begin{cases} n^2, & \text{ако n е четно и } n \geq 0 \\ n^3, & \text{ако n е нечетно и } n \geq 1 \end{cases}$$

$$g_2(n) = \begin{cases} n + 3, & \text{ако } 0 \leq n \leq 100 \\ n^3 + 7, & \text{ако } n > 100 \end{cases}$$

$g_3(n) = n^{2.5}$.

Укажете за кои двойки функции $g_i(n)$ и $g_j(n)$ е в сила:
- $g_i(n)$ принадлежи на $O(g_j(n))$;
- $g_i(n)$ принадлежи на $\Omega(g_j(n))$;
- $g_i(n)$ принадлежи на $\Theta(g_j(n))$.

Оригинал: стр. 27

4. За всеки от следващите програмни фрагменти извършете следните действия:
- като използвате нотацията О-голямо, намерете времето за изпълнението му;
- включете го в програма на C++ и определете времето ѝ за изпълнение при различни стойности на n;
- сравнете направения анализ с реалните времена за изпълнение.

а)

C027-01 · Копиране: code.html
```
p = 1;
for (int i = 1; i <= n; i++)
  p *= i;
```

б)

C027-02 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
    s++;
```

в)

C027-03 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    s++;
```

г)

C027-04 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n*n; j++)
    s++;
```

д)

C027-05 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
    for (int k = 1; k <= j; k++)
      s++;
```

е)

C027-06 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i*i; j++)
    for (int k = 1; k <= j; k++)
      s++;
```

ж)

C027-07 · Копиране: code.html
```
s = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i*i; j++)
    if (j%i == 0)
      for (int k = 1; k <= j; k++)
        s++;
```

Оригинал: стр. 28

5. Като използвате нотацията О-голямо, намерете времето за изпълнение на следните функции:

а)

C028-01 · Копиране: code.html
```
void example1(int n, int a[][30], int b[][30],
                                   int c[][30])
{ int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      { c[i][j] = 0;
        for (k = 0; k < n; k++)
          c[i][j] += a[i][k]*b[k][j];
      }
}
```

б)

C028-02 · Копиране: code.html
```
int example2(int n)
{ int i, j, sum = 0;
  for (i = 1; i < n; i++)
    for (j = i+1; j <= i*i; j++)
      sum++;
  return sum;
}
```

в)

C028-03 · Копиране: code.html
```
void example3(int n)
{ int i, j, k;
  for (i = 1; i < n; i++)
    for (j = i+1; j <= n; j++)
      for (k = 1; k <= j*j; k++)
        { // последователност от оператори
          // с време за изпълнение O(1)
        }
}
```

г)

C028-04 · Копиране: code.html
```
void example4(int n)
{ int i, j, x = 0, y = 0;
  for (i = 1; i <= n; i++)
    if (i%2 == 1)
      { for (j = 1; j <= n; j++) x++;
        for (j = 1; j <= i; j++) y++;
      }
}
```

Оригинал: стр. 29

д)

C029-01 · Копиране: code.html
```
void recA(int n)
{ for (int i = 1; i <= n; i++)
    { // последователност от оператори
      // с време за изпълнение O(1)
    }
  if (n > 1) recA(n-1);
}
```

е)

C029-02 · Копиране: code.html
```
void recB(int n)
{ for (int i = 1; i <= n; i++)
    { // последователност от оператори
      // с време за изпълнение O(1)
    }
  if (n > 1) recB(n/2);
}
```

ж)

C029-03 · Копиране: code.html
```
void recC(int n)
{ // последователност от оператори
  // с време за изпълнение O(1)
  for (int i = 1; i <= n; i++)
    if (n > 1) recC(n-1);
}
```

з)

C029-04 · Копиране: code.html
```
int example5(int n)
{ if (n <= 1) return 1;
  else return example5(n-1) + example5(n-1);
}
```

и)

C029-05 · Копиране: code.html
```
int fib(int n)
{ if (n == 0) return 0; else
    if (n == 1) return 1; else
      return fib(n-1) + fib(n-2);
}
```

Оригинал: стр. 30

Глава 2

ЛИНЕЙНИ СТРУКТУРИ ОТ ДАННИ

В тази глава ще разгледаме дефинирането, реализирането и някои приложения на линейните динамични структури от данни стек, опашка и линеен списък.

2.1. СТЕК

В информатиката стековете (*stack* - куп, купчина) се използват за редица цели, някои от които са свързани с реализацията на механизма на обръщенията към подпрограми, с представянето и намирането на стойности на изрази, със синтактичния анализ и др.

2.1.1. Дефиниране на стек

Логическо описание

Нека *T* е даден тип данни. *Стек с елементи от тип T* е линейна динамична структура. Определя се като крайна (възможно празна) редица от елементи от тип *T*. Единият край на редицата, в случай че тя не е празна, се нарича *връх* или *начало на стека*. Над стек могат да се изпълняват следните операции:
- създаване на празен стек;
- определяне дали стек е празен;
- добавяне (включване) на елемент във върха на стек;
- отстраняване (изключване) на елемент от върха на стек, ако той не е празен;
- достъп до елемента от върха на стек, ако той не е празен.

Забележка. Операциите добавяне, отстраняване и достъп до елемент се извършват само във върха на стека.

От дефиницията следва, че последният елемент, включен в стек, се изключва първи. Затова стекът се определя още като структура „последен влязъл - пръв излязъл“ (*Last In First Out - LIFO*).

Физическо представяне

Широко се използват две физически представяния на стек: последователно и свързано. Нека редицата от елементи от тип *T*

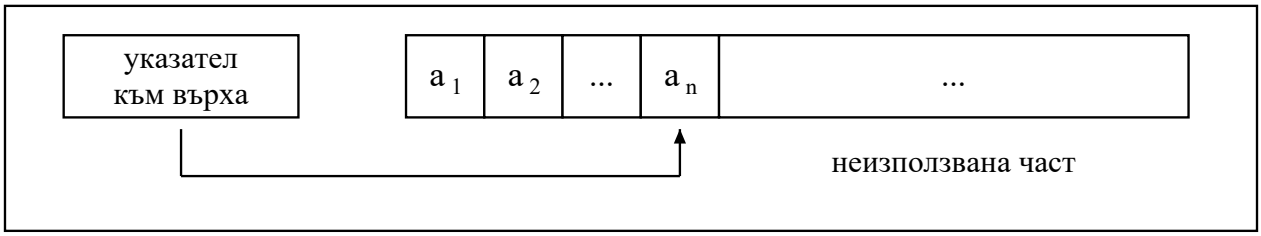
$$a_1, \quad a_2, \quad \ldots, \quad a_n \uparrow$$

е стек с връх a_n. Ще покажем последователното и свързаното представяне на този стек.

Оригинал: стр. 31

Последователно представяне

При това представяне (фиг. 2.1) се запазва блок от паметта, вътре в който стекът да се променя: нараства при добавяне на елементи или намалява при отстраняване на елементи.

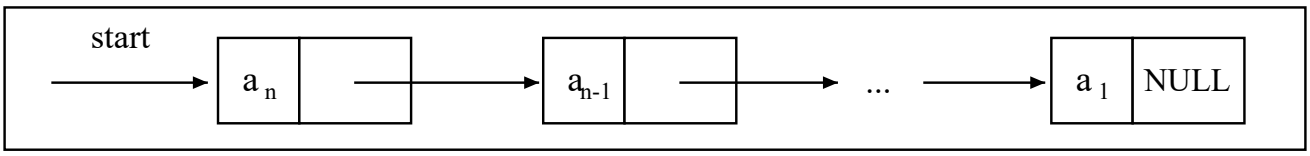


Фигура 2.1. Последователно представяне на стек

При добавяне на елементи те се поместват в последователни адреси в неизползваната част веднага след върха на стека. Тази операция е възможна до пълното изчерпване на неизползваната част. При отстраняване на елементи се променя указателят към върха на стека. Блокът от паметта, в който се съхранява стекът, се реализира чрез едномерен масив, а указателят към върха на стека - чрез индекс.

Свързано представяне

При това представяне (фиг. 2.2) елементите на стека не се разполагат в последователни адреси в паметта, а „пръснато“ - там където има място. Реализира се като всеки елемент на стека се представя чрез блок с две полета (двойна кутия): *информационно*, съдържащо стойността на елемента и *адресно*, съдържащо адреса на предишния елемент на стека. Адресното поле на последния елемент в схематичното изображение по-долу е означено чрез константата *NULL*. Указателят към върха на стека е означен със *start*.



Фигура 2.2. Свързано представяне на стек

Добавянето на елемент се осъществява като се създаде нова двойна кутия. В информационното ѝ поле се записва добавяният елемент, а в адресното поле - *start* (адресът на върха на стека). Тъй като новата двойна кутия представя върха на стека, на указателя *start* се присвоява адресът ѝ.

Отстраняването на елемент се реализира като в помощна променлива се записва информационното поле на двойната кутия, представяща върха на стека, указателят *start* се свързва с адреса от адресното поле, след което се разрушава двойната кутия от върха на стека.

2.1.2. Реализация на стек

2.1.2.1. Последователно представяне

В шаблона на класа, който реализира последователното представяне, стекът е представен чрез динамичния масив *arr* с максимален размер *stack_size* и указателя *stack_top* към върха на стека. Тъй като стекът е разположен в динамичната памет, в шаблона е включено каноничното представяне, състоящо се от: конструктор *stack(int)*, създаващ стек, разположен в динамичната памет с указания чрез параметъра размер; деструктор *~stack()*, разрушаващ (изтриващ) неявния стек; конструктор за присвояване *stack(const stack<T>&)*, реализиращ копирането на указания чрез параметъра стек в неявния стек и операторна функция за присвояване *operator=*. За реализирането на каноничното представяне класът е допълнен с две помощни член-функции: *copyStack*, създаваща копие на стек на друго място в паметта и *deleteStack*, разрушаваща неявния стек. Член-функциите *empty* и *full* определят дали неявният стек е празен или е пълен. Чрез *push* указаният чрез формалния параметър елемент се добавя в неявния стек. Член-функцията *pop* отстранява елемент от неявния стек, ако последният не е празен и връща отстранения елемент в указания формален параметър. Член-функцията *top*, без да отстранява елемент, намира върха на неявния стек, ако последният не е празен и го записва в указания формален параметър. Член-функцията *print* извежда на екрана елементите на неявния стек. При реализацията ѝ предполагаме, че операторът << може да извежда данни от *тип T*.

C032-01 · Копиране: code.html

```
const int MAXSIZE = 20;

template <typename T = int>
class stack
{ public:
    // канонично представяне
    stack(int = MAXSIZE);
    ~stack();
    stack(const stack<T>&);
    stack<T>& operator=(const stack<T>&);
    // основни операции
    bool empty() const;
    bool full() const;
    void push(const T&);
    void pop(T&);
```

Оригинал: стр. 33

C033-01 · Копиране: code.html

```
    void top(T&) const;
    void print();
  private:
    int stack_size;   // максимален размер на стека
    int stack_top;    // връх на стека
    T* arr;           // носител на стека
    void copyStack(const stack<T>&);
    void deleteStack();
};

// параметричен конструктор
template <typename T>
stack<T>::stack(int size)
{ if (size <= 0) stack_size = MAXSIZE;
  else stack_size = size;
  arr = new T[stack_size];
  assert(arr != NULL);
  stack_top = -1;
}

// деструктор
template <typename T>
stack<T>::~stack()
{ deleteStack();
}

// конструктор за присвояване
template <typename T>
stack<T>::stack(const stack<T>& s)
{ copyStack(s);
}

// операторна функция за присвояване
template <typename T>
stack<T>& stack<T>::operator=(const stack<T>& st)
{ if (this != &st)
    { deleteStack();
      copyStack(st);
    }
  return *this;
```

Оригинал: стр. 34

C034-01 · Копиране: code.html

```
}

// помощна член-функция, която копира st в неявния стек
template <typename T>
void stack<T>::copyStack(const stack<T>& st)
{ stack_size = st.stack_size;
  stack_top = st.stack_top;
  arr = new T[stack_size];
  assert(arr != NULL);
  for(int i = 0; i <= stack_top; i++)
    arr[i] = st.arr[i];
}

// помощна член-функция, която изтрива неявния стек
template <typename T>
void stack<T>::deleteStack()
{ delete [] arr;
}

// проверява дали неявният стек е празен
template <typename T>
bool stack<T>::empty() const
{ return stack_top == -1;
}

// проверява дали неявният стек е пълен
template <typename T>
bool stack<T>::full() const
{ return stack_top == stack_size-1;
}

// намира върха на неявния стек, ако последният не е
// празен, и го записва във формалния параметър x
template <typename T>
void stack<T>::top(T& x) const
{ if (!empty())
    x = arr[stack_top];
  else
    { cout << "The stach is empty.\n";
      exit(1);
```

Оригинал: стр. 35

C035-01 · Копиране: code.html

```
    }
}

// включва x в неявния стек, ако последният не е пълен
template <typename T>
void stack<T>::push(const T& x)
{ if (!full())
    { stack_top++;
      arr[stack_top] = x;
    }
  else
    { cout << "The stack is full.\n";
      exit(1);
    }
}

// изключва елемент от неявния стек, ако
// последният не е празен и го записва в x
template <typename T>
void stack<T>::pop(T& x)
{ if(!empty())
    { x = arr[stack_top];
      stack_top--;
    }
  else
    { cout << "The stack is empty.\n";
      exit(1);
    }
}

// извежда на екрана елементите на
// неявния стек като го разрушава
template <typename T>
void stack<T>::print()
{ T x;
  while (!empty())
    { pop(x);
      cout << x << " ";
    }
  cout << endl;
}
```

Оригинал: стр. 36

Направената реализация записваме във файла *stack_s.cpp*. Включена е в следващата програма, в която са създадени три стека: *st*, *st1* и *st2*.

C036-01 · Копиране: code.html
```
#include <iostream>
#include <cassert>
#include "stack_s.cpp"
using namespace std;

int main()
{ stack<> st(3);
  st.push(10); st.push(20); st.push(30);
  stack<int> st1 = st, st2;
  st2 = st1;
  st2.print();
  return 0;
}
```

Дефиницията

C036-02 · Копиране: code.html
```
stack<> st(3);
```

създава стека *st* от 3 цели числа чрез прилагане на еднопараметричния конструктор. Чрез

C036-03 · Копиране: code.html
```
st.push(10); st.push(20); st.push(30);
```

се включват числата 10, 20 и 30 в стека *st*, като числото 30 е на върха на стека.

Дефиницията

C036-04 · Копиране: code.html
```
stack<int> st1 = st, st2;
```

създава стековете *st1* и *st2*. Стекът st1 е създаден от конструктора за присвояване като копие на стека *st*, а стекът *st2* – от еднопараметричния конструктор, като се използва подразбиращата се стойност *MAXSIZE*. След това чрез оператора за присвояване *st2* се разрушава и отново се създава като дословно копие на стека *st1*. Накрая се извеждат на екрана елементите на стека *st2*.

2.1.2.2. Свързано представяне

Ще дефинираме шаблон на клас *stack*, който реализира стек с елементи от тип *T*. При това представяне (фиг. 2.2) стекът е редица от „двойни кутии“ с две части: информационна част, задаваща текущ елемент, и адресна част, задаваща адреса на двойната кутия, представяща елемента, предшестващ текущия. Двойните кутии ще реализираме чрез шаблона на структурата *elem*

C036-05 · Копиране: code.html
```
template <typename T>
struct elem
```

Оригинал: стр. 37

C037-01 · Копиране: code.html
```
{ T inf;        // информационна част
  elem* link;   // адресна част
};
```

Изборът на структура, вместо клас, е с цел да се опрости реализацията на представянето на елементите на стека, тъй като ще осигури пряк достъп до компонентите на двойната кутия.

Достъпът до върха на стека се осъществява чрез указателя *start*, деклариран в *private* секцията на шаблона на класа *stack*. В тази секция са декларирани и две помощни член-функции: *copyStack*, създаваща копие на стек на друго място в динамичната памет и *delStack*, изтриваща неявния стек. Тези член-функции се използват за дефиниране на каноничното представяне. Конструкторът по подразбиране създава празен стек (*start е равен на NULL*).

Член-функцията *empty* определят дали неявният стек е празен. Чрез *push* указаният чрез формалния параметър елемент се добавя в неявния стек. Член-функцията *pop* отстранява елемент от неявния стек, в случай че последният не е празен и връща отстранения елемент в указания формален параметър. Член-функцията *top*, без да отстранява елемент от неявния стек, в случай че последният не е празен, намира върха му и го записва в указания формален параметър. Член-функцията *print* извежда на екрана елементите на неявния стек, а *length* намира броя на елементите на неявния стек.

C037-02 · Копиране: code.html
```
template <typename T = int>
class stack
{ public:
    // канонично представяне
    stack();
    ~stack();
    stack(const stack&);
    stack& operator=(const stack&);
    // основни операции
    bool empty() const;
    void push(const T&);
    void pop(T&);
    void top(T&) const;
    void print();
    int length();
  private:
    elem<T> *start; // указател към върха на стека
    void copyStack(const stack&);
    void delStack();
```

Оригинал: стр. 38

C038-01 · Копиране: code.html

```
};

// канонично представяне на шаблона на класа
template <typename T>
stack<T>::stack()
{ start = NULL;
}

template <typename T>
stack<T>::~stack()
{ delStack();
}

template <typename T>
stack<T>::stack(const stack<T>& r)
{ copyStack(r);
}

template <typename T>
stack<T>& stack<T>::operator=(const stack<T>& r)
{ if (this != &r)
    { delStack();
      copyStack(r);
    }
  return *this;
}

// помощна член-функция, която копира r в неявния стек
template <typename T>
void stack<T>::copyStack(const stack<T>& r)
{ if (r.start)
    { elem<T> *p = r.start, *q = NULL, *s = NULL;
      start = new elem<T>;
      assert(start != NULL);
      start->inf = p->inf;
      start->link = q;
      q = start;
      p = p->link;
      while (p)
```

Оригинал: стр. 39

C039-01 · Копиране: code.html

```
        { s = new elem<T>;
          assert(s != NULL);
          s->inf = p->inf;
          q->link = s;
          q = s;
          p = p->link;
        }
      q->link = NULL;
    }
  else start = NULL;
}

// помощна член-функция, която изтрива неявния стек
template <typename T>
void stack<T>::delStack()
{ elem<T> *p;
  while (start)
    { p = start;
      start = start->link;
      delete p;
    }
}

// проверява дали неявният стек е празен
template <typename T>
bool stack<T>::empty() const
{ return start == NULL;
}

// включва s в неявния стек
template <typename T>
void stack<T>::push(const T& s)
{ elem<T>* p = start;
  start = new elem<T>;
  assert(start != NULL);
  start->inf = s;
  start->link = p;
}
```

Оригинал: стр. 40

C040-01 · Копиране: code.html

```
// изключва елемент от неявния стек и го записва
// в параметъра s
// операцията е възможна, ако неявният стек не е празен
template <typename T>
void stack<T>::pop(T& s)
{ if (start)
    { s = start->inf;
      elem<T>* p = start;
      start = start->link;
      delete p;
    }
  else
    { cout << "The stack is empty.\n";
      exit(1);
    }
}

// намира върха на неявния стек и го записва в параметъра s
// операцията е възможна, ако неявният стек не е празен
template <typename T>
void stack<T>::top(T& s)const
{ if (start)
    s = start->inf;
  else
  { cout << "The stack is empty.\n";
    exit(1);
  }
}

// извежда на екрана елементите на неявния стек
// предполага, че за елементите от тип T е
// дефиниран операторът <<
// след завършване на изпълнението на функцията,
// неявният стек става празен
template <typename T>
void stack<T>::print()
{ T x;
  while (!empty())
    { pop(x);
      cout << x << " ";
```

Оригинал: стр. 41

C041-01 · Копиране: code.html

```
    }
  cout << endl;
}

// намира броя на елементите на неявния стек
// след завършване на изпълнението на функцията,
// неявният стек става празен
template <typename T>
int stack<T>::length()
{ int n = 0;
  T x;
  while (!empty())
    { pop(x);
      n++;
    }
  return n;
}
```

Направената реализация записваме във файла *stack.cpp* и ще използваме в някои от следващите приложения.

2.1.3. Приложения на стек

2.1.3.1. Използване на обратен полски запис за пресмятане на стойност на аритметичен израз

През 1920 г. полският математик Ян Лукашевич е предложил начин за записване на аритметични изрази, без да се използват скоби. При общоприетия начин за записване на аритметични изрази знакът на двуаргументните операции се поставя между аргументите. Този запис се нарича още *инфиксен*. Ян Лукашевич е предложил две други форми за записване на аритметични изрази: *постфиксна*, при която знакът за операция се записва след аргументите си, и *префиксна*, при която знакът за операция се записва пред аргументите си. Например изразът *(2+3)*5* се представя в постфиксен формат във вида *2 3 + 5 **, а в префиксен формат – във вида *** + *2 3 5*. В чест на Ян Лукашевич постфиксната форма се нарича *обратен полски запис*, а префиксната форма – *прав полски запис*.

Правият полски запис не се прилага често в програмистката практика, но обратният е изключително полезен. Предимството на обратния пред правия полски запис неформално може да се обясни с това, че изпълнението на някакво действие е по-удобно, когато обектите, над които то ще се извърши, са вече дадени.

Оригинал: стр. 42

В тази част ще илюстрираме приложение на обратен полски запис за пресмятане на стойността на аритметичен израз. Ще разглеждаме аритметични изрази от вида:

C042-01 · Копиране: code.html

```
<израз> ::= <терм>|
            <израз> + <терм>|
            <израз> - <терм>
<терм> ::= <множител>|
           <терм> * <множител>|
           <терм> / <множител>
<множител> ::= <цифра>|<променлива>|(<израз>)|
               <множител> ^ <цифра>
<цифра> ::= 0|1| ... |9
<променлива> ::= <буква>
<буква> ::= a| b| c| ... |x| y| z
```

чрез ^ е означена операцията степенуване.

Пресмятането на стойността на аритметичен израз се реализира по-просто, ако изразът е записан не в обичайния му инфиксен вид, а в обратен полски запис.

Примери.

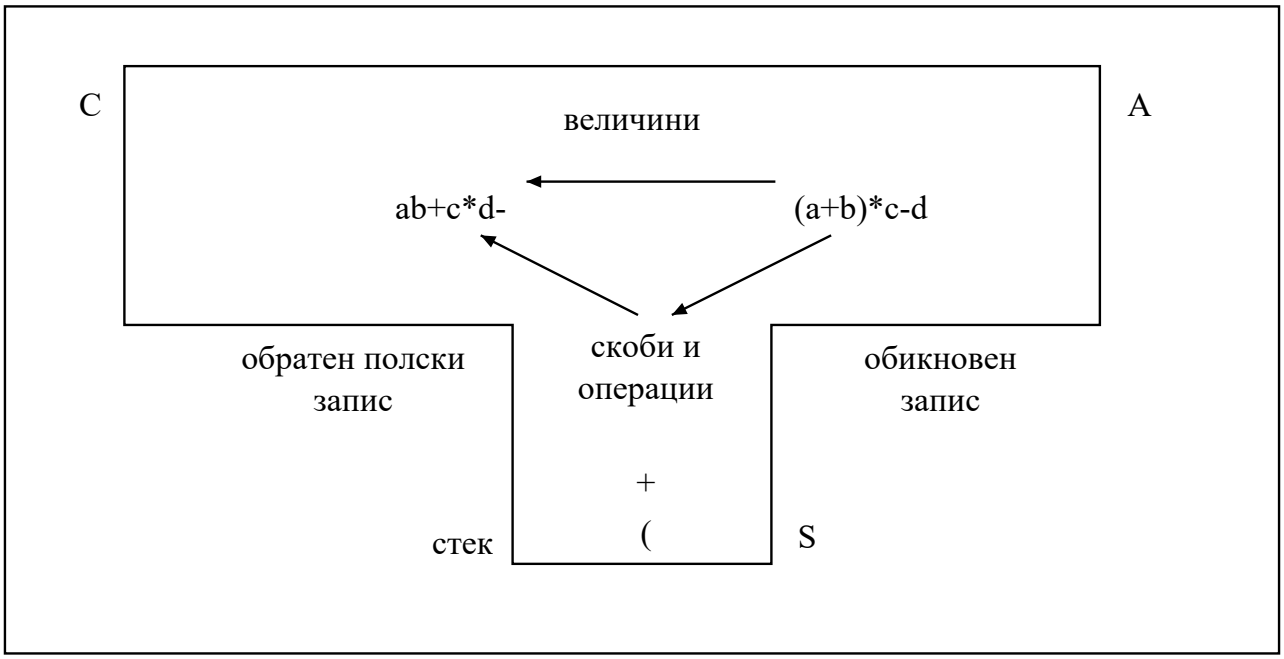
обикновен запис	обратен полски запис
(a-b/c)*m	abc/-m*
a^n*b^m	an^bm^*
(a-b)*(a+b)	ab-ab+*
(a+b)*c-d*f+m^p	ab+c*df*-mp^+

Ако аритметичен израз е представен чрез обратен полски запис, пресмятането на стойността му се осъществява по следния начин. Сканира се изразът, представен чрез обратен полски запис отляво надясно до намиране на операция. Пресмята се стойността на терма с аргументи – първите два, намиращи се непосредствено пред операцията. Операцията и операндите ѝ се заменят с резултата от пресмятането, след което продължава търсенето на следваща операция. Сканирането продължава до достигане края на израза. За целта е удобно да се използва стек като запомнящо устройство на операнди и междинни резултати.

Преобразуване на израз от обикновен в обратен полски запис

Фигура 2.3 илюстрира преобразуването. На нея стекът е означен с *S*. Символите, участващи в обикновения запис, се прехвърлят от частта *A* в частта *C*, като някои от тях временно престояват в стека *S*. В частта *C* се получава изразът, записан в обратен полски запис.

Оригинал: стр. 43



Фигура 2.3. Преобразуване на аритметичен израз в обратен полски запис

Правилата за движение са следните:

Величините (цифри и променливи) се преместват директно от частта *A* в частта *C*.

Скобата ‘(’ се включва в стека *S*.

Аритметичните операции +, -, *, / и ^ също се включват в стека *S*. Всяка операция има определен приоритет, който се представя чрез цяло число, наречено тегло или приоритетен номер на операцията. Приемаме, че най-тежки са операциите с най-нисък приоритет (+ и -), по-леки от тях са операциите с по-висок от тях приоритет * и / и най-лека е операцията с най-висок приоритет – за степенуване. Ако при включване на операция в стека *S* под нея има операция с по-малко или с равно тегло, по-леката или с равно тегло операция се премества от *S* в частта *C*. Това се повтаря, докато се достигне до по-тежка операция, до ‘(’ или до изпразването на стека.

Скобата ‘)’ изключва от стека *S* всички операции до достигане до ‘(’. Операциите се записват в частта *C* в реда на изключването им, а скобата ‘(’ се унищожава от ‘)’.

Ако всички символи от частта *A* са обработени, елементите на стека *S*, до изпразването му или до достигане до ‘(’, се прехвърлят в областта *C*.

Задача 1. Да се дефинира функция, която преобразува аритметичен израз от вида, определен по-горе, в обратен полски запис.

Оригинал: стр. 44

Процедурата

C044-01 · Копиране: code.html
```
void translate(char *s, char *ns);
```

преобразува аритметичния израз, представен в обикновен запис чрез низа *s*, в обратен полски запис – низа *ns*. За да се избегне проверката за празен стек, в помощния стек още в началото

включваме ‘(’.

C044-02 · Копиране: code.html

```
void translate(char* s, char* ns)
{ stack<char> st;   //помощен стек
  st.push('(');
  char x;
  int i = 0, j = -1, n = strlen(s);
  while (i < n)
    { if (s[i] >= '0' && s[i] <= '9' ||
          s[i] >= 'a' && s[i] <= 'z')
        { j++;
          ns[j] = s[i];
        }
      else
        if (s[i] == '(') st.push(s[i]); else
          if (s[i] == ')')
            { st.pop(x);
              while (x != '(')
                { j++;
                  ns[j] = x;
                  st.pop(x);
                }
            }
          else
            if (s[i] == '+' || s[i] == '-' ||
                s[i] == '*' || s[i] == '/' ||
                s[i] == '^')
              { st.pop(x);
                while (x != '(' && t(x) <= t(s[i]))
                  { j++;
                    ns[j] = x;
                    st.pop(x);
                  }
                st.push(x);
                st.push(s[i]);
```

Оригинал: стр. 45

C045-01 · Копиране: code.html

```
                }
      i++;
    }
  st.pop(x);
  while (x != '(')
    { j++; ns[j] = x;
      st.pop(x);
    }
  j++;
  ns[j] = '\0';
}
```

Използваната в *translate* функция *t* намира теглото на операциите, използвани в аритметичния израз, определен по-горе и има вида:

C045-02 · Копиране: code.html
```
int t(char c)
{ int p;
  switch (c)
  { case '+':
    case '-': p = 2; break;
    case '*':
    case '/': p = 1; break;
    case '^': p = 0; break;
    default:  p = -1;
  }
  return p;
}
```

Намиране на стойността на израз, представен в обратен полски запис

Задача 2. Да се дефинира функция, която намира стойността на аритметичния израз, представен в обратен полски запис.

Функцията

C045-03 · Копиране: code.html
```
int value(char* s);
```

намира стойността на израза, като използва обратния полски запис (представен чрез низа *s*), който му съответства. Предполага се, че променливите (буквите) в израза предварително са свързани с конкретни цифри. За пресмятането на стойността се създава помощен стек от цели числа, който отначало е празен. Сканира се низът *s*. При срещане на цифра тя се включва в стека. При срещане на знак за операция от стека се изключват операндите на операцията, изпълнява се операцията и резултатът от изпълнението ѝ се включва в стека. След изчерпването на низа, стекът съдържа стойността на израза.

Оригинал: стр. 46

C046-01 · Копиране: code.html

```
int value(char* s)
{ stack<int> st;  // създава празен стек st от цели числа
  int x, y, z;
  unsigned int i = 0, n = strlen(s);
  while (i < n)
    { if (s[i] >= '0' && s[i] <= '9')
        st.push((int)s[i]-(int)'0');
      else
        if (s[i] == '+' || s[i] == '-' ||
            s[i] == '*' || s[i] == '/' ||
            s[i] == '^')
          { st.pop(y);
            st.pop(x);
            switch (s[i])
            { case '+': z = x+y; break;
              case '-': z = x-y; break;
              case '*': z = x*y; break;
              case '/': z = x/y; break;
              case '^': z = (int)pow(x,y);
            }
            st.push(z);
          }
      i++;
    }
  st.pop(z);
  return z;
}
```

Намирането на стойността на аритметичен израз може да се реализира чрез изпълнението на следната главна функция.

C046-02 · Копиране: code.html

```
int main()
{ char s[200];
  cout << "s: "; cin >> s;    // въвеждане на
                              // аритметичния израз
                              // в инфиксен формат
  char s1[200];
  translate(s,s1);            // преобразуване на низа s
                              // в обратен полски запис
  cout << value(s1) << endl; // пресмятане и извеждане
                              // на стойността на израза
  return 0;
}
```

Оригинал: стр. 47

Подобна идея е реализирана при стековите калкулатори. За пръв път стеков калкулатор е реализиран от фирмата Hewlett Packard. Обикновените калкулатори не позволяват да се изчисляват сложни формули без използване на хартия и молив за записване на междинни резултати. В някои модели на обикновени калкулатори има скоби, които могат да се влагат до две нива, но по-сложни формули тези калкулатори не могат да изчисляват. Обикновените калкулатори използват регистри. Стековите калкулатори имат памет във вид на стек. Чрез тях могат да се намират стойности на сложни формули, без да се налага да се записват междинни резултати.

2.1.3.2. Други приложения на стек, чрез използване на обратен полски запис

Обратният полски запис е изключително удобен не само при намиране на стойност на израз. Много компилатори го използват като вътрешна форма за представяне на програми. Например компилаторът на *Java* преобразува *Java*-програмата не в машинен код, а в междинен език, наречен байткод. Опростено казано, байткодът е обратен полски запис на *Java*-програмата, а *Java*-машината е стеков изчислител.

Друг пример за използване на обратен полски запис и в частност на структурата стек е графичният език *PostScript*. Той е предназначен за печат на текстове на висококачествени лазерни принтери. Явява се стандарт за представяне на текстове с типографско качество, които не зависят от конкретния модел на принтера. Програмата на езика *PostScript* е редица от оператори, зададени в обратен полски запис.

Например

Език C	Език PostScript
-A	A neg
A + B	A B add
A * (B + C)	A B C add mul
sqrt(sqr(A) + sqr(B))	A dup mul B dup mul add sqrt

Четейки файл на програма на езика *PostScript*, интерпретаторът на езика записва в стек операндите, след което изпълнява оператора като изключва операндите от стека. Освен стек на операндите интерпретаторът на езика *PostScript* използва още три стека: стек за речниците, стек за изпълнение и стек за графичните състояния.

Оригинал: стр. 48

2.1.3.3. Пресмятане на стойност на израз с използване на стек и без преобразуване в обратен полски запис

Чрез пример ще покажем прилагането на структурата от данни стек за намиране на стойност на израз без преобразуване в обратен полски запис.

Задача 3. В масив е записан без грешка булев израз от вида

C048-01 · Копиране: code.html
```
<булев_израз> ::= t | f | (~<булев_израз>)|
                  (<булев_израз>*<булев_израз>)|
                  (<булев_израз>+<булев_израз>)
```

където *t* означава *истина*, *f – лъжа*, а ~, * и + означават съответно логическо отрицание, конюнкция и дизюнкция. Да се напише програма, която намира стойността на правилно записан в символен низ булев израз.

Функцията *boolFormula* предлага решение на задачата. Тя използва два помощни стека *s1* и *s2*. Ако сканираният символ е знак за операция, се включва в стека *s1*, а ако е *t* или *f* – в стека *s2*. Затварящата скоба изключва знак за операция от стека *s1*, анализира го и в зависимост от вида на операцията – унарна или бинарна изключва един или два елемента от *s2*. Изпълнява се операцията над съответните аргументи и резултатът се включва в стека *s2*. Останалите символи, в т.ч. ‘(’, се пропускат.

C048-02 · Копиране: code.html
```
// Program Zad3.cpp
#include <iostream>
#include <cstring>
#include <cassert>
#include "stack.cpp"

using namespace std;

bool boolFormula(char* s)
{ stack<char> s1, s2;    //помощни стекове
  char c, x, y;
  int i = 0, n = strlen(s);
  while (i < n)
    { if (s[i] == '~' || s[i] == '*' || s[i] == '+')
        s1.push(s[i]);
      else
        if (s[i] == 't' || s[i] == 'f') s2.push(s[i]);
        else
```

Оригинал: стр. 49

C049-01 · Копиране: code.html

```
          if (s[i] == ')')
            { s1.pop(c);
              switch(c)
              { case '~': s2.pop(x);
                          if (x == 't') c = 'f';
                          else c = 't';
                          s2.push(c); break;
                case '*': s2.pop(y); s2.pop(x);
                          if (x == 't' && y == 't') c = 't';
                          else c = 'f';
                          s2.push(c); break;
                case '+': s2.pop(y); s2.pop(x);
                          if (x == 'f' && y == 'f') c = 'f';
                          else c = 't';
                          s2.push(c);
              }
            }
      i++;
    }
  s2.pop(c);
  return c == 't';
}

int main()
{ char s[50];
  cout << "s: "; cin >> s;
  cout << boolalpha << boolFormula(s) << endl;
  return 0;
}
```

2.1.3.4. Използване на стек за елиминиране на рекурсията

Това приложение ще покажем чрез два примера.
Нека да елиминираме рекурсията в рекурсивно дефинираната функция за намиране факториела на естествено число.

C049-02 · Копиране: code.html

```
int fact(int n)
{ if (n == 0) return 1;
  return n*fact(n-1);
}
```

Оригинал: стр. 50

За целта ще използваме стек от цели числа. Включването на елемента *k* в стека ще интерпретираме като командата „*намери k!*“. Новата версия на програмата има вида

C050-01 · Копиране: code.html
```
int fact(int n)
{ stack<int> st; st.push(n);
  int x, f = 1;
  while(!st.empty())
  { st.pop(x);
    if(x == 0) return f;
    f = f*x;
    st.push(x-1);
  }
  return 1;  // за да се избегне предупреждението
             // "not all control paths return a value"
}
```

Реализацията на метода за бързо сортиране в задача 57 (Глава 4) е с използване на рекурсия. В нея всяко от рекурсивните обръщения във фрагмента

C050-02 · Копиране: code.html
```
if (j > 0) qsort(a, j+1);
if (i < n-1) qsort(a+i, n-i);
```

може да се разглежда като команда. В първия случай командата е „сортирай частта от масива *a*, започваща от адреса *a* с дължина *j+1*“, а във втория случай – „сортирай частта от масива *a*, започваща от адреса *a+i* с дължина *n-i*“. Известно е, че рекурсивните обръщения се реализират, като се използва стек.

За да изключим рекурсията от реализацията на шаблона на процедурата *qsort*, ще дефинираме помощен стек, чрез който ще имитираме рекурсията. Елементите на стека ще са команди от горния вид. Командите са дефинирани чрез шаблона на структурата *rec*, притежаваща две полета: *arr* от тип *T** и указващо адреса на началото на масива и *len* от тип *int*, указващо дължината на масива, чиито елементи ще бъдат сортирани.

Отначало в стека записваме командата „*сортирай целия масив a*“. След това, докато има команди в стека, от него се изключва команда и се изпълнява съгласно алгоритъма на бързото сортиране. Следва програма, която реализира метода на бързото сортиране (задача 57, Глава 4), без да използва рекурсия.

C050-03 · Копиране: code.html
```
#include <iostream>
#include <cassert>
#include "stack.cpp"
```

Оригинал: стр. 51

C051-01 · Копиране: code.html

```
using namespace std;

// шаблон на структура, задаваща фрагмент за сортиране
template <typename T>
struct rec
{ T* arr;    // адрес на началото на масива
  int len;   // дължина на масива, който ще се сортира
};

template <typename T>
void qsort(T* a, int n)
{ // създаване на празен стек
  stack< rec<T> > st;
  // създаване на командата "сортирай цялата редица"
  rec<T> r1 = {a, n};
  // включване на командата в стека st
  st.push(r1);
  // докато стекът с команди не е празен,
  // се изпълняват командите, записани в него
  while(!st.empty())
  { // изключване на команда от стека st
    st.pop(r1);
    // намиране на началото и дължината на масива, който
    // ще се сортира по метода на бързото сортиране
    a = r1.arr, n = r1.len;
    // намиране на оста x
    int i = 0, j = n-1;
    T x = a[(i+j)/2], y;
    // разделяне на редицата на две подредици спрямо x
    do
    { while (a[i] < x) i++;
      while (a[j] > x) j--;
      if (i < j)
        { y = a[i]; a[i] = a[j]; a[j] = y;
          i++; j--;
        }
      else
        if (i==j)
          { i++;
            j--;
          }
```

Оригинал: стр. 52

C052-01 · Копиране: code.html

```
    } while (i <= j);
    // конструиране и включване на команда
    // за сортиране в стека
    if (j > 0)
      { rec<T> r1 = {a, j+1};
        st.push(r1);
      }
    // конструиране и включване на команда за сортиране в стека
    if (i < n-1)
      { rec<T> r1 = {a+i, n-i};
        st.push(r1);
      }
  } // край на while
}

int main()
{ int a[100];
  int n; cout << "n= "; cin >> n;
  for (int i = 0; i < n; i++)
    cin >> a[i];
  qsort(a, n);
  for (i = 0; i < n; i++)
    cout << a[i] << " ";
  cout << endl;
  return 0;
}
```

Съществуват и други реализации на елиминирането на рекурсията при метода за бързо сортиране. Следващата използва стек, елементите на който са записи, задаващи индексите на началото и края на частта от масива, която трябва да се сортира. Например:

C052-02 · Копиране: code.html

```
struct rec
{ int left,     // индекс, задаващ началото на масива
      right;   // индекс, задаващ края на масива,
               // който ще се сортира
};

template <typename T>
void qsort(T* a, int n)
{ // създаване на празен стек
  stack<rec> st;
```

Оригинал: стр. 53

C053-01 · Копиране: code.html

```
  int l, r;        // помощни променливи
  // създаване на командата "сортирай цялата редица"
  rec r1 = {0, n-1};
  // включване на командата в стека st
  st.push(r1);
  // докато стекът с команди не е празен,
  // се изпълняват командите от стека
  while(!st.empty())
    { st.pop(r1);
      int i = r1.left, j = r1.right;
      T x = a[(i+j)/2], y;
      // запазване на границите на частта от масива,
      // която ще се сортира
      l = i;
      r = j;
      do
      { while (a[i] < x) i++;
        while (a[j] > x) j--;
        if (i < j)
          { y = a[i]; a[i] = a[j]; a[j] = y;
            i++; j--;
          }
        else
          if (i==j)
            { i++;
              j--;
            }
      } while (i <= j);
      // конструиране и добавяне на команда за сортиране в стека
      if (j > l)
        { rec r1 = {l, j}; st.push(r1);
        }
      if (i < r)
        { rec r1 = {i, r}; st.push(r1);
        }
    }
}
```

Оригинал: стр. 54

2.1.3.5. Използване на стек при реализиране на някои функции на текстовите редактори

Всички текстови редактори имат определени символи, които изпълняват специални функции като: изтриване на символ, изтриване на цял ред и др. В следващото приложение като символ за изтриване на един символ ще използваме ‘!’, а като символ за изтриване на ред – ‘#’. Например, ако е въведен ред от символи от вида abcd!efg!!!!hij!klm, то в действителност въведеният ред е abhiklm, а ако е въведен ред от вида abcdefgh#, то в действителност редът е празен. За реализиране на тези функции е удобно да се използва стек. За целта текстовият редактор последователно чете символи. Ако символът не е ‘!’ и ‘#’, то той се включва в стек от символи. Ако символът е ‘!’, то от стека се изключва символ, а ако символът е ‘#’ – стекът се изпразва. Процедурата *textEdit* реализира описаните действия при предположение, че символът ‘!’ за изтриване на символ е използван коректно, т.е. че съществуват символи за изтриване. Входният низ се въвежда от клавиатурата.

C054-01 · Копиране: code.html

```
void textEdit()
{ stack<char> s, t; // помощни стекове
  char c, x;
  cin.get(c);
  if (c != '!' && c != '#')
    { while (c != '\n')
        { if (c == '!') s.pop(x);
          else if (c == '#') s = t;
               else s.push(c);
          cin.get(c);
        }
      while (!s.empty())
        { s.pop(x);
          t.push(x);
        }
      while (!t.empty())
        { t.pop(x);
          cout << x;
        }
    }
  cout << endl;
}
```

Оригинал: стр. 55

Стекът *s* във функцията по-горе съдържа резултата от редакцията, но записан в обратен ред. Обръщането се осъществява чрез използването на помощния стек *t*.

2.2. ОПАШКА

В информатиката опашките (*queue* – опашка, редица) се използват за редица цели, но класическото им приложение е за организация на обработка на паралелни процеси. Прилагат се и за симулиране поведението на реални системи.

2.2.1. Дефиниране на опашка

Логическо описание

Нека *T* е даден тип данни. *Опашка с елементи от тип T* е линейна динамична структура от данни. Определя се като крайна (евентуално празна) редица от елементи от тип *T*. Единият край на редицата, в случай че тя не е празна, се нарича *начало* или *глава на опашката*, а другият край се нарича *край* или *тил* (задна част) на опашката.

Над опашка могат да се изпълняват следните операции:
– създаване на празна опашка;
– определяне дали опашка е празна;
– добавяне (включване) на елемент само в края на опашката;
– отстраняване (изключване) на елемента само в началото на опашката, ако тя не е празна;
– достъп до елемента в началото на опашката, ако тя не е празна.

Резултатът от отстраняването на елемент от непразна опашка е опашка от същия тип и се нарича *тяло* на първоначалната опашка. Достъпът до началото на опашка е пряк. Реализира се чрез указател. За удобство при реализиране на операцията добавяне на елемент е предвиден и пряк достъп до последния (или след последния) елемент на опашка. На фиг. 2.4 е илюстрирана непразна опашка.

От дефиницията следва, че елементът, добавен първи в опашката, се отстранява първи. Затова опашката се определя още като структура „първи влязъл – първи излязъл“ (*First In First Out – FIFO*).

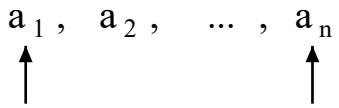
Физическо представяне

Широко се използват две физически представяния на опашка: последователно и свързано.

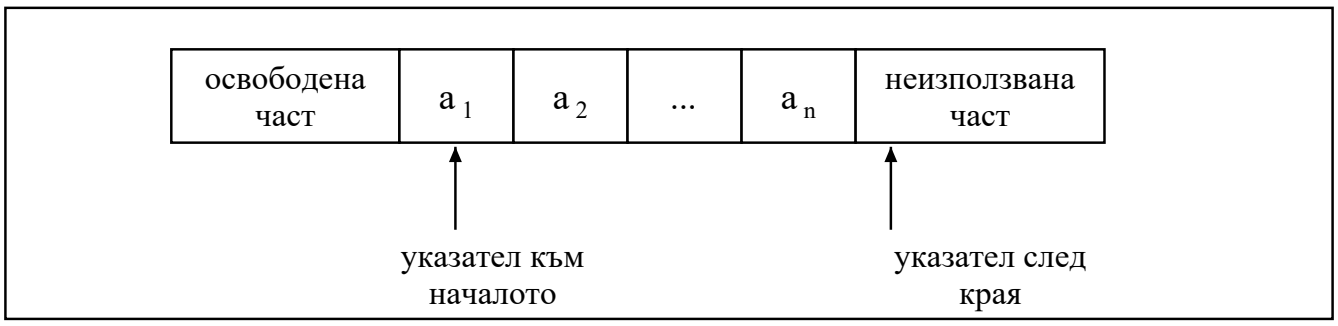
Оригинал: стр. 56

Последователно представяне

При това представяне се запазва блок от паметта, вътре в който опашката да расте и да се съкращава. Ако редицата с елементи от тип *T*



е опашка с начало a_1 и край a_n, последователното представяне има следния вид:



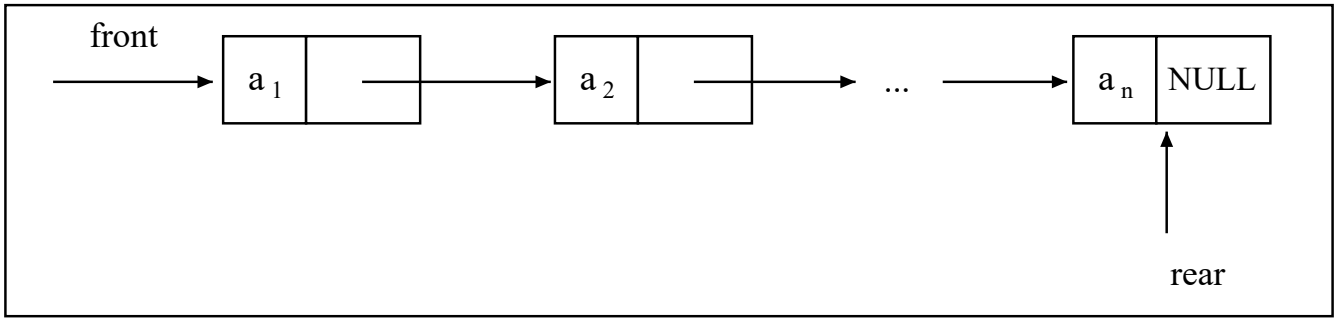
Фигура 2.4. Последователно представяне на опашка

Включването на елемент в опашката се осъществява чрез поместването му в последователни адреси в неизползваната част веднага след края на опашката. Съществуват разновидности на това представяне. Една от тях, която най-често се използва, е *цикличното* представяне. При него при изчерпване на масива, ако има свободна памет в началото му, включването се извършва там. При отстраняване на елементи се променя указателят към началото на опашката и се освобождава паметта, заета от отстранения елемент. Блокът от паметта, в който се съхранява опашката, се реализира чрез едномерен масив, а указателите към началото и след края на опашката – чрез индекси.

Свързано представяне

Това представяне е аналогично на свързаното представяне на стек. Елементите на опашката (фиг. 2.5) не се разполагат в последователни адреси в паметта, а „пръснато“ – там където има място. Всеки елемент на опашката се представя чрез блок с две полета (двойна кутия): *информационно*, съдържащо стойността на елемента и адресно, съдържащо адреса на следващия елемент на опашката. Адресното поле на последния елемент е означено чрез константата *NULL* (нулев указател), указателят към началото на опашката е означен с *front*, а указателят към края ѝ – с *rear*.

Оригинал: стр. 57



Фигура 2.5. Свързано представяне на опашка

2.2.2. Реализация на опашка

2.2.2.1. Последователно представяне

Ще реализираме цикличното последователно представяне. В шаблона на класа, който дефинира това представяне, опашката е представена чрез динамичния масив *arr* с максимален размер *queue_size* и текущ размер на опашката *queue_len*, с указател към началото на опашката *front* и указател след края – *rear*. Указателите *front* и *rear* се реализират като индекси на масива *arr*.

Тъй като опашката е разположена в динамичната памет, в шаблона е включено каноничното представяне, състоящо се от: конструктор *queue(int)*, създаващ опашка, разположена в динамичната памет с указания чрез параметъра размер, в случай че параметърът задава коректен размер и с размер *MAXSIZE*, в случай че параметърът не е коректен; деструктор *~queue()*, разрушаващ опашката; конструктор за присвояване *queue(const queue<T>&)*, реализиращ копирането на указаната чрез параметъра опашка в неявната опашка и операторна функция за присвояване *operator=*. За реализирането на каноничното представяне класът е допълнен с две помощни член-функции: *copyQueue*, създаваща копие на зададена опашка на друго място в паметта и *deleteQueue*, изтриваща неявната опашка. Член-функциите *empty* и *full* определят дали неявната опашка е празна и съответно дали е пълна. При цикличното последователно представяне опашката е празна, ако текущият ѝ размер е 0 и опашката е пълна, ако текущият ѝ размер е равен

на максималния размер *queue_size*. Чрез *push* указаният чрез формалния параметър елемент се включва в неявната опашка. Член-функцията *pop* изключва елемент от неявната опашка и връща отстранения елемент в указания формален параметър. Член-функцията *head*, без да отстранява елемент от неявната опашка, намира началото ѝ, в случай че опашката не е празна и го записва в указания формален параметър. Член-функцията *print* извежда на екрана елементите на неявната опашка. За реализирането ѝ предполагаме, че операторът << е предефиниран за типа *T*.

Оригинал: стр. 58

Включването на елемент в неявната опашка се осъществява само ако опашката не е пълна. За целта в соченото от *rear* празно място се записва елементът. Текущият размер на опашката се увеличава с 1, след което указателят *rear* се премества в следващата позиция. Тъй като след изпълнението на *rear++* е възможно *rear* да излезе от границите на масива *arr* (т.е. *rear == queue_size*), чрез изпълнение на оператора *rear = rear % queue_size; rear* се премества в началото на масива (т.е. *rear == 0*).

Изключването на елемент се осъществява, ако неявната опашка не е празна. Елементът, сочен от указателя *front*, се запомня във формалния параметър на процедурата *pop*. Текущият размер на опашката се намалява с 1, след което указателят *front* се премества в следващата позиция. Тъй като след изпълнението на оператора *front++* също е възможно *front* да излезе от границите на масива *arr* (т.е. *rear == queue_size*), чрез изпълнение на оператора *front = front % queue_size; front* се премества в началото на масива (т.е. *front == 0*).

C058-01 · Копиране: code.html

```
const int MAXSIZE = 20;
template <typename T = int>
class queue
{ public:
    // канонично представяне
    queue(int = MAXSIZE);
    ~queue();
    queue(const queue&);
    queue& operator=(const queue&);
    // основни функции
    bool empty() const;
    bool full() const;
    void push(const T&);
    void pop(T&);
    void head(T&) const;
    void print();
  private:
    int front,      // указател към началото на опашка
        rear,       // указател след края на опашка
        queue_size, // максимален размер на опашка
        queue_len;  // текущ размер на опашка
    T *arr;         // носител на опашката
    // помощни функции
```

Оригинал: стр. 59

C059-01 · Копиране: code.html

```
    void copyQueue(const queue&);
    void deleteQueue();
};

// канонично представяне
template <typename T>
queue<T>::queue(int size)
{ if (size <= 0) queue_size = MAXSIZE;
  else queue_size = size;
  arr = new T[queue_size];
  assert(arr != NULL);
  front = 0;
  rear = 0;
  queue_len = 0;
}

template <typename T>
queue<T>::~queue()
{ deleteQueue();
}

template <typename T>
queue<T>::queue(const queue<T>& r)
{ copyQueue(r);
}

template <typename T>
queue<T>& queue<T>::operator=(const queue<T>& r)
{ if (this != &r)
    { deleteQueue();
      copyQueue(r);
    }
  return *this;
}

// помощна член-функция, която копира r в неявната опашка
template <typename T>
void queue<T>::copyQueue(const queue<T>& r)
{ queue_size = r.queue_size;
  arr = new T[queue_size];
```

Оригинал: стр. 60

C060-01 · Копиране: code.html

```
  assert(arr != NULL);
  for (int i = 0; i < queue_size; i++)
    arr[i] = r.arr[i];
  queue_len = r.queue_len;
  front = r.front;
  rear = r.rear;
}

// помощна член-функция, която изтрива неявната опашка
template <typename T>
void queue<T>::deleteQueue()
{ delete [] arr;
}

// проверява дали неявната опашка е празна
template <typename T>
bool queue<T>::empty() const
{ return queue_len == 0;
}

// проверява дали неявната опашка е пълна
template <typename T>
bool queue<T>::full() const
{ return queue_len == queue_size;
}

// включва x в неявната опашка, ако последната не е пълна
template <typename T>
void queue<T>::push(const T& x)
{ if (!full())
    { arr[rear] = x;
      queue_len++;
      rear++;
      rear = rear % queue_size;
    }
  else
    { cout << "The queue is full.\n";
      exit(1);
    }
}
```

Оригинал: стр. 61

C061-01 · Копиране: code.html

```
// изключва елемент от неявната опашка, ако
// последната не е празна и го записва в x
template <typename T>
void queue<T>::pop(T& x)
{ if (!empty())
    { x = arr[front];
      queue_len--;
      front++;
      front = front % queue_size;
    }
  else
    { cout << "The queue is empty.\n";
      exit(1);
    }
}

// намира главата на неявната опашка, ако последната не
// е празна, и я записва във формалния параметър x
template <typename T>
void queue<T>::head(T& x) const
{ if (!empty())
    x = arr[front];
  else
    { cout << "The queue is empty.\n";
      exit(1);
    }
}

// извежда на екрана елементите на
// неявната опашка като я разрушава
template <typename T>
void queue<T>::print()
{ T x;
  while (!empty())
    { pop(x);
      cout << x << " ";
    }
  cout << endl;
}
```

Оригинал: стр. 62

Записваме този шаблон във файл с име *queue_s.cpp*.
Следва програма, която използва дефинирания шаблон.

C062-01 · Копиране: code.html

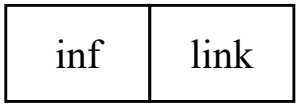
```
#include <iostream>
#include <cassert>
using namespace std;
#include "queue_s.cpp"
```

C062-01 · продължение · Копиране: code.html

```
int main()
{ queue<> q;                    // създава празна опашка q
                                // от тип int с максимален
                                // размер MAXSIZE
  for (int i = 1; i <= 10; i++) // включва в q целите числа
    q.push(i);                  // от 1 до 10
  q.print();                    // извежда опашката q
  queue<char> p(26);             // създава празна опашка от
                                // символи p с максимален
                                // размер 26
  for (char c = 'a'; c <= 'z'; c++) // запълва p с малките
    p.push(c);                    // латински букви
  p.print();                    // извежда опашката p
  return 0;                     // извиква деструктора на
                                // класа за разрушаване на
                                // опашките p и q
}
```

2.2.2.2. Свързано представяне

Ще дефинираме шаблон на клас *queue*, реализиращ свързаното представяне на опашка. Двойната кутия



ще реализираме чрез шаблона на структурата *elem_q*

C062-02 · Копиране: code.html

```
template <typename T = int>
struct elem_q
{ T inf;             // информационна част
  elem_q<T>* link;   // адресна част
};
```

Оригинал: стр. 63

а опашката – чрез шаблона на класа *queue*. Указателят към началото на опашката ще означим с *front*, а този към края – с *rear*. Тъй като опашката е разположена в динамичната памет, в шаблона е реализирано каноничното представяне. Конструкторът по подразбиране създава празна опашка. За реализирането на останалите функции от каноничното представяне се използват помощните функции *copyQueue*, която създава копие на указана опашка и *deleteQueue*, която изтрива неявната опашка.

C063-01 · Копиране: code.html

```
template <typename T = int>
class queue
{ public:
    // канонично представяне
```

C063-01 · продължение · Копиране: code.html

```
    queue();
    ~queue();
    queue(const queue&);
    queue& operator=(const queue&);
    // основни функции
    bool empty() const;
    void push(const T&);
    void pop(T&);
    void head(T&) const;
    void print();
    int length();
  private:
    elem_q<T> *front,   // указател към началото
              *rear;   // указател към края
    // помощни функции
    void copyQueue(const queue&);
    void deleteQueue();
};

// канонично представяне
template <typename T>
queue<T>::queue()
{ front = rear = NULL;
}

template <typename T>
queue<T>::~queue()
{ deleteQueue();
```

Оригинал: стр. 64

C064-01 · Копиране: code.html

```
}

template <typename T>
queue<T>::queue(const queue<T>& r)
{ copyQueue(r);
}

template <typename T>
queue<T>& queue<T>::operator=(const queue<T>& r)
{ if (this != &r)
    { deleteQueue();
      copyQueue(r);
    }
  return *this;
}

// помощна член-функция, която копира r в неявната опашка
template <typename T>
void queue<T>::copyQueue(const queue<T>& r)
{ front = rear = NULL;
  elem_q<T> *p = r.front;
  while (p)
    { push(p->inf);
      p = p->link;
    }
}

// помощна член-функция, която изтрива неявната опашка
template <typename T>
void queue<T>::deleteQueue()
{ T x;
  while (!empty())
    pop(x);
}

// проверява дали неявната опашка е празна
template <typename T>
bool queue<T>::empty() const
{ return rear == NULL;
}
```

Оригинал: стр. 65

C065-01 · Копиране: code.html

```
// включва x в неявната опашка
template <typename T>
void queue<T>::push(const T& x)
{ elem_q<T> *p = new elem_q<T>;
  assert(p != NULL);
  p->inf = x;
  p->link = NULL;
  if (rear) rear->link = p;
  else front = p;
  rear = p;
}

// изключва елемент от неявната опашка, ако
// последната не е празна и го записва в x
template <typename T>
void queue<T>::pop(T& x)
{ if (empty())
    { cout << "The queue is empty.\n";
      exit(1);
    }
  else
    { elem_q<T> *p = front;
      x = p->inf;
      if (p == rear)
        { rear = NULL;
          front = NULL;
        }
      else front = p->link;
      delete p;
    }
}

// намира главата на неявната опашка, ако последната не
// е празна, и я записва във формалния параметър x
template <typename T>
void queue<T>::head(T& x) const
{ if (empty())
    { cout << "The queue is empty.\n";
      exit(1);
```

Оригинал: стр. 66

C066-01 · Копиране: code.html
```
    }
  else
    x = front->inf;
}

// извежда на екрана елементите на
// неявната опашка като я разрушава
template <typename T>
void queue<T>::print()
{ T x;
  while (!empty())
    { pop(x);
      cout << x << " ";
    }
  cout << endl;
}

// намира броя на елементите на
// неявната опашка като я разрушава
template <typename T>
int queue<T>::length()
{ T x; int n = 0;
  while (!empty())
    { pop(x);
      n++;
    }
  return n;
}
```

Записваме този шаблон във файл с име *queue_link.cpp* и ще го използваме в някои следващи задачи.

Задача 4. Да се напише програма, която създава и извежда опашка от опашки от цели числа.

Програма *Zad4.cpp* решава задачата като създава опашка от 5 опашки от вида:

C066-02 · Копиране: code.html
```
1 2              // първа опашка
2 3 4            ...
3 4 5 6
4 5 6 7 8
5 6 7 8 9 10     // пета опашка
```

Оригинал: стр. 67

В нея член-функцията *print* на шаблона на класа *queue* е специализирана, за да може да извежда опашка от опашки от цели числа. Причината е, че тя извежда елементите на опашка чрез използване на *оператора* <<, а в шаблона *queue<T>* не е предефиниран *operator<<*.

C067-01 · Копиране: code.html

```
// Program Zad4.cpp
#include <iostream>
#include <cassert>
#include "queue_link.cpp"
using namespace std;

// дефиниране на клас intQueue, определящ опашка от цели числа
typedef queue<int> intQueue;

// дефиниране на клас queueQueue, определящ опашка
// от опашки от цели числа
typedef queue<intQueue> queueQueue;

// специализиране на шаблонната член-функция print
// за извеждане на опашка от опашки от цели числа
void queue<intQueue>::print()
{ intQueue x;
  while (!empty())
    { pop(x);
      x.print();
    }
  cout << endl;
}

int main()
{ // qq е обект, представящ опашка от опашки от цели числа
  queueQueue qq;
  for (int i = 1; i <= 5; i++)
    { // конструиране на опашка q от цели числа
      intQueue q;
      for (int j = i; j <= 2*i; j++)
        q.push(j);
      // включване на опашката q в опашката qq
      qq.push(q);
    }
```

Оригинал: стр. 68

C068-01 · Копиране: code.html

```
  qq.print();
  return 0;
}
```

Задача 5. Да се напише програма, която само с едно преминаване през елементите на масив от числа (без използване на допълнителни масиви) извежда на екрана елементите на масива в следния ред: отначало всички числа, които са по-малки от *a*, след това всички числа в интервала *[a, b]* и накрая всички останали числа, запазвайки техния първоначален ред (*a* и *b* са дадени числа, $a < b$).

Програма *Zad5.cpp* решава задачата. Тя използва две опашки *q1* и *q2*. В опашката *q1* се записват числата от интервала *[a, b]*, а в опашката *q2* – тези, които са по-големи от *b*, в реда на тяхното

срещане в масива.

C068-02 · Копиране: code.html
```
// Program Zad5.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "queue_link.cpp"

typedef queue<int> intQueue;
// въвеждане на масив
void read(int n, int* a)
{ for (int i = 0; i < n; i++)
    { cout << "a[" << i << "]= ";
      cin >> a[i];
    }
}

int main()
{ // създаване на масив arr от числа
  int arr[100], n;
  do
  { cout << "n= ";
    cin >> n;
  } while (n < 1 || n > 100);
  read(n, arr);
  // въвеждане на границите на интервала
  cout << "a < b = ";
```

Оригинал: стр. 69

C069-01 · Копиране: code.html
```
  int a, b;
  cin >> a >> b;
  // създаване на две празни опашки от цели числа
  intQueue q1, q2;
  // обхождане на масива arr и извеждане на екрана на
  // всички числа, които са по-малки от a; включване в
  // опашката q1 на числата, които са в интервала [a, b]
  // и включване на числата по-големи от b в опашката q2
  for (int i = 0; i < n; i++)
    if (arr[i] < a) cout << arr[i] << " ";
    else if (arr[i] <= b) q1.push(arr[i]);
         else q2.push(arr[i]);
  // извеждане на елементите натрупани в опашките q1 и q2
  cout << endl;
  q1.print();
  q2.print();
  return 0;
}
```

Задача 6. Да се напише програма, която извежда на екрана в растящ ред първите *n* естествени числа, като при разлагането на прости множители участват само числата 2, 3 или 5.

За решението на задачата ще използваме три опашки *q2*, *q3* и *q5*, в които ще включваме естествени числа, които са съответно 2, 3 и 5 пъти по-големи от последното изведено естествено число. Например ако последното изведено число е *x*, в опашката *q2* включваме *2.x*, в опашката *q3* включваме *3.x*, а в опашката *q5* включваме *5.x*. Поредният елемент, който ще извеждаме, ще е най-малкият елемент от главите на опашките *q2*, *q3* и *q5*. След като го изведем, го изключваме и включваме неговите кратни в опашките. Броят на елементите в опашките не е по-голям от изведените елементи.

C069-02 · Копиране: code.html

```
// Program Zad6.cpp
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
#include "queue_link.cpp"

typedef queue<int> intQueue;
```

Оригинал: стр. 70

C070-01 · Копиране: code.html

```
// намиране на минимума на три числа
int min(int a, int b, int c)
{ int m = a;
  if (b < m) m = b;
  if (c < m) m = c;
  return m;
}

intQueue q2, q3, q5; // създаване на три празни опашки

void displayAndPush(int x)
{ if (x != 1) cout << setw(5) << x;
  q2.push(x*2);
  q3.push(x*3);
  q5.push(x*5);
}

int main()
{ int x, i, n;
  cout << "Брой на елементите, които да се изведат ";
  cin >> n;
  displayAndPush(1);
  for(i = 1; i <= n; i++)
    { int a, b, c;
      q2.head(a);
      q3.head(b);
      q5.head(c);
      x = min(a, b, c);
      displayAndPush(x);
      if (x == a) q2.pop(a);
      if (x == b) q3.pop(b);
      if (x == c) q5.pop(c);
    }
  cout << endl;
  return 0;
}
```

Оригинал: стр. 71

2.2.3. Сортиране и сливане на опашки

Сортиране

Ще реализираме метода на пряката селекция за сортиране на елементите на опашка. За целта ще използваме помощната процедура

C071-01 · Копиране: code.html

```
void minqueue(queue<T> q, T& min, queue<T>& newq)
```

която за непразната опашка *q* намира минималния ѝ елемент *min* и опашка *newq*, състояща се от елементите на опашката *q* без минималния елемент. Преди обръщение към *minqueue*, опашката *newq* трябва да е празна.

Резултатът от сортирането ще намерим в нова опашка (опашката *newq*, която отначало трябва да е празна). За сортиране на елементите ще извършим следните действия: докато дадената опашка не е празна, чрез процедурата *minqueue* намираме минималния ѝ елемент и опашката, без минималния елемент; включваме минималния елемент в опашката, която ще съдържа резултата от сортирането и присвояваме на първоначалната опашка опашката, получена след отстраняване на минималния елемент.

C071-02 · Копиране: code.html
```
template <typename T>
void minqueue(queue<T> q, T& min, queue<T>& newq)
{ T x;
  q.pop(min);
  while (!q.empty())
    { q.pop(x);
      if (x < min)
        { newq.push(min);
          min = x;
        }
      else newq.push(x);
    }
}

template <typename T>
void sortQueue(queue<T> q, queue<T>& newq)
{ while (!q.empty())
    { T min;
      queue<T> q1;
      minqueue(q, min, q1); // опашката q не е празна,
                            // а опашката q1 е празна
      newq.push(min);
      q = q1;
```

Оригинал: стр. 72

C072-01 · Копиране: code.html
```
    }
}
```

Реализациите на шаблоните на тези две функции предполагат, че за типа *T* е дефиниран операторът за сравнение <. Записваме ги във файла *sortQueue.cpp*.

Сливане

Ще реализираме алгоритъм, аналогичен на описания в т. 4.1.7 (глава 4). Функцията *merge* слива сортираните във възходящ ред непразни опашки *p* и *q*. За улесняване на реализацията в края на всяка сортирана опашка добавяме фиктивен елемент *dummy*, който се нарича *сентинел*. Този подход често се използва при работа с линейни динамични структури.

C072-02 · Копиране: code.html

```
template <typename T>
queue<T> merge(queue<T> p, queue<T> q, const T& dummy)
{ p.push(dummy);    // включване на dummy в опашката p
  q.push(dummy);    // включване на dummy в опашката q
  queue<T> r;       // r ще съдържа резултата
  T x, y;
  p.pop(x);
  q.pop(y);
  while (!p.empty() && !q.empty())
    if (x < y)
      { r.push(x);
        p.pop(x);
      }
    else
      { r.push(y);
        q.pop(y);
      }
  if (!p.empty())   // опашката q е празна
    do
      { r.push(x);
        p.pop(x);
      } while (x != dummy);
  else              // опашката p е празна
    do
      { r.push(y);
        q.pop(y);
      } while (y != dummy);
```

Оригинал: стр. 73

C073-01 · Копиране: code.html

```
  return r;
}
```

Реализацията на този шаблон на функция предполага, че за типа *T* са дефинирани (предефинирани) операторите за сравнение < и !=. Предефинирането на оператора *!=* ще е излишно, ако условието *x != dummy* се замени с *!p.empty()*, а условието *y != dummy* се замени с *!q.empty()*. Записваме шаблона на тази функция във файла *mergeQueues.cpp*.

Задача 7. Да се напише програма, която създава две опашки, елементите на които са структури, съдържащи име и възраст на човек. Сортира във възходящ ред по полето възраст елементите на опашките, след което ги слива и извежда получената опашка.

В следващата програма е дефинирана структурата *people* с две информационни полета *name* и *age*, задаващи името и възрастта на човек и две член-функции, предефиниращи операторите за сравняване за „<“ и „!=“ на две лица. Освен това член-функцията *print* на шаблона на класа *queue* е специализирана, за да може да извежда обект от класа *people*.

C073-02 · Копиране: code.html

```
// Program Zad7.cpp
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;

#include "queue_link.cpp"
#include "sortQueue.cpp"
#include "mergeQueues.cpp"

struct people
{ char name[32];
  int age;
  bool operator<(const people& p) const
  { return age < p.age;
  }
  bool operator!=(const people& p) const
  { return age != p.age;
  }
};
```

Оригинал: стр. 74

C074-01 · Копиране: code.html

```
typedef queue<people> pqueue;

void queue<people>::print()
{ people x;
  while (!empty())
    { pop(x);
      cout << setw(35) << x.name << setw(15) << x.age << endl;
    }
}

int main()
{ people s, dummy = {"xxx", -1};
  pqueue q1;
  int i, n;
  // създаване на първата опашка
  cout << "First Queue:\n n= ";
  cin >> n;
  for (i = 0; i < n; i++)
    { cout << "Name: ";
      cin.ignore();
      cin.getline(s.name, 32);
      cout << "Age: ";
      cin >> s.age;
      q1.push(s);
    }
  // създаване на втората опашка
  pqueue q2;
  cout << "Second Queue \n n= ";
  cin >> n;
  for (i = 0; i < n; i++)
    { cout << "Name: ";
      cin.ignore();
      cin.getline(s.name, 32);
      cout << "Age: ";
      cin >> s.age;
      q2.push(s);
    }
  pqueue p, q, r;
  sortQueue(q1, p); // сортиране на първата опашка
```

Оригинал: стр. 75

C075-01 · Копиране: code.html

```
  sortQueue(q2, q);        // сортиране на втората опашка
  r = merge(p, q, dummy);   // сливане на сортираните опашки
  r.print();               // извеждане на резултата
  return 0;
}
```

Задача 8. Да се напише програма, която създава опашка от опашки, елементите на които са структури, съдържащи име и възраст на човек. Да се сортират във възходящ ред по възраст опашките на опашката, след което да се слеят.

Следва само главната функция, която решава задачата. Останалата част е като в предходната задача.

C075-02 · Копиране: code.html

```
int main()
{ queue<pqueue> qq;    // празна опашка от опашки от хора
  int m;              // брой на опашките на qq
  cout << "m= "; cin >> m;
  // създаване на опашката от опашки от хора qq
  for (int j = 1; j <= m; j++)
    { int i, n;
      pqueue q;
      cout << j << " Queue:\n n= ";
      // брой на елементите на j-тата опашка
      cin >> n;
      // създаване на j-тата опашка
      for (i = 0; i < n; i++)
        { people s;
          cout << "Name: ";
          cin.ignore();
          cin.getline(s.name, 32);
          cout << "Age: ";
          cin >> s.age;
          q.push(s);
        }
      // включване на j-тата опашка в опашката qq
      qq.push(q);
    }
  // създаване на помощна опашка от опашки от хора qq1,
  // която ще съдържа сортираните опашки
```

Оригинал: стр. 76

C076-01 · Копиране: code.html

```
  queue<pqueue> qq1;
  while (!qq.empty())
    { pqueue r;
      qq.pop(r);
      pqueue t;
      sortQueue(r, t);
      qq1.push(t);
    }
  // сливане на сортираните по възраст опашки
  pqueue merge_queue, help_queue; // помощни опашки от хора
  qq1.pop(merge_queue);
  people dummy = {"xxx", -1};
  while (!qq1.empty())
  { qq1.pop(help_queue);
    merge_queue = merge(merge_queue, help_queue, dummy);
  }
  merge_queue.print();    // извежда опашката,
                          // съдържаща резултата
                          // от сливането
  return 0;
}
```

2.2.4. Приложения на структурата от данни опашка

В операционните системи структурата от данни опашка се използва за съхраняване на чакащи за работа задания, за съхраняване на задания, за извеждане на данни на принтер и др.
Опашки се използват за симулиране поведението на реални опашки. Компютърната симулация е средство, което се използва от програмисти, учени и инженери за описване на различни ситуации от работата на технически системи. Една от най-старите и известни симулации е на движение на самолети. Пример за реализация на такава симулация с използване на опашки са дадени например в книгата на Udi Manber [24], както и в [2]. Други доста разпространени симулации са на телефонни мрежи, на обслужване на клиенти в банки и институции, в които могат да се създадат реални опашки от хора.

Някои проблеми от теорията на графите дават добро изпълнение, като се използват опашки.

Оригинал: стр. 77

2.3. ЛИНЕЕН СПИСЪК

Стекът и опашката са редици от краен брой елементи от един и същ тип с дисциплина (с ред) на добавяне и отстраняване на елементите. Линейният списък (ще го наричаме понякога само списък) е обобщение на тези структури от данни. При него операциите добавяне и отстраняване на елемент могат да се реализират в произволно място на редицата. Възможен е достъп до всеки елемент на списъка.

2.3.1. Дефиниране на линеен списък

Логическо описание

Нека *T* е даден тип данни. *Линеен списък с елементи от тип T* е линейна динамична структура от данни. Определя се като крайна (възможно празна) редица от елементи от тип *T*. Един от елементите, например най-левият, се нарича *първи, глава* или *начален елемент*. Останалите елементи на редицата също образуват линеен списък, който се нарича *подсписък* или *опашка на списъка*. Над линеен списък могат да се извършват следните операции:

- създаване на празен линеен списък;
- определяне дали линеен списък е празен;
- добавяне (включване) на елемент към линеен списък;
- отстраняване (изключване) на елемент от линеен списък;
- достъп до елемент на линеен списък.

Операциите добавяне, отстраняване и достъп се извършват в произволно място на списъка. Операциите отстраняване и достъп са възможни само ако списъкът не е празен. Достъпът е пряк само до първия елемент и е последователен до елементите на опашката на списъка. С цел улесняване на изпълнението на операциите повечето реализации допускат пряк достъп до последния (най-десния) елемент на списъка, както и до произволен елемент, ако е известен адресът му.

За разлика от стека и опашката, обхождането на елементите на линейния списък се осъществява без разрушаването му.

Физическо представяне

Има два основни начина за представяне на линеен списък в паметта на компютъра: последователно представяне (чрез едномерен масив) и свързано представяне (чрез свързани списъци с една и с две връзки фиг. 2.6, 2.7 и 2.8). Последователното представяне не се използва напоследък заради добре развитите средства за динамично разпределение на паметта. Затова понякога

Оригинал: стр. 78

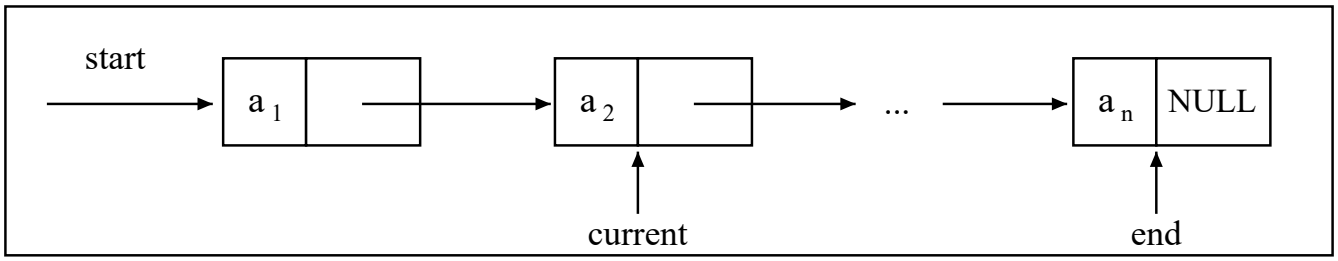
линейният списък се нарича също и *свързан списък*. За някои приложения са полезни цикличните представяния с една и с две връзки.

Свързано представяне с една връзка

Аналогично е на свързаното представяне на стек и на опашка. Свързаното представяне с една връзка на списъка

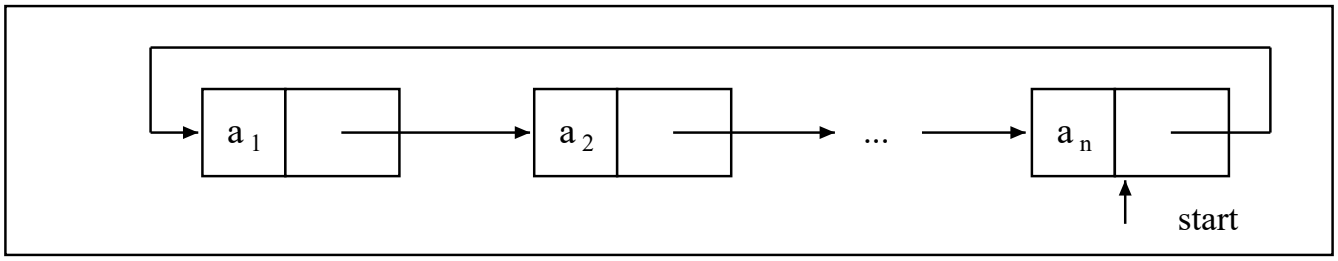
$a_1, a_2, ..., a_n$

с първи елемент a_1, е показано на фиг. 2.6. За удобство при реализирането на операциите включване, изключване и достъп, освен указател към първия елемент (*start*), се въвеждат и указатели към последния елемент (*end*) и към текущ елемент (*current*) на списъка. Това представяне се нарича още *едносвързано представяне на списък*.



Фигура 2.6. Представяне на линеен списък с една връзка

Циклично едносвързано представяне на линеен списък е дадено на фиг. 2.7.



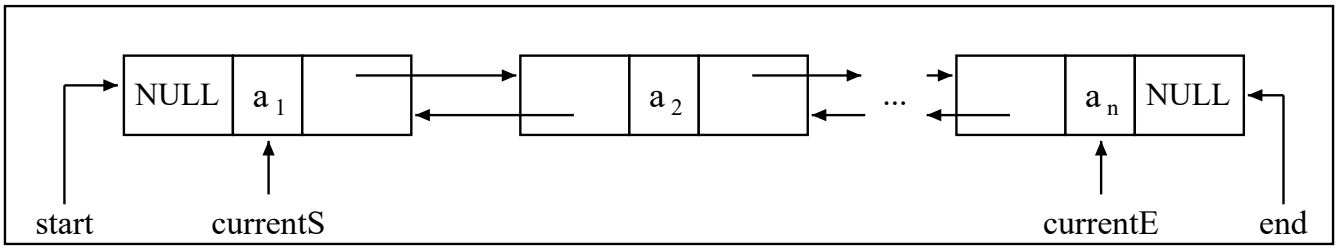
Фигура 2.7. Циклично представяне на линеен списък с една връзка

Свързано представяне с две връзки

За удобство при реализирането на операциите включване, изключване, достъп и обхождане се използват „тройни кутии“ с едно информационно и две свързващи полета, съдържащи съответно текущия елемент и адресите на предшестващия и следващия го елементи на линейния списък (фиг. 2.8). Освен указатели *start* и *end* към първия и към последния елемент се въвеждат още

Оригинал: стр. 79

два указателя *currentS*, придвижващ се от началото към края и *currentE*, придвижващ се от края към началото на линейния списък.



Фигура 2.8. Представяне на линеен списък с две връзка

В книгата ще разгледаме свързаното представяне с една и с две връзки, както и цикличното представяне с една връзка.

Предимства на свързаното представяне

- лесно се добавят и отстраняват елементи;
- размерът на списъка е ограничен само от обема на паметта на компютъра;
- динамично се добавят и отстраняват елементите.

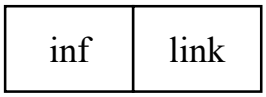
Недостатъци на свързаното представяне

- сложно се определя адресът на елемент на списъка по неговия пореден номер в списъка;
- изразходва се допълнителна памет за адресните полета;
- работата с линеен списък е по-бавна, отколкото с масив, тъй като достъпът до елемент на списъка се осъществява, като се обходят всички предшестващи го елементи;
- над свързаните списъци трудно се извършват паралелни операции.

Реализира се като абстрактен тип от данни (АТД), формализиращ понятието наредена съвкупност от данни.

2.3.2. Реализация на линеен списък, представен чрез една връзка

Ще дефинираме шаблон на клас *LList*, реализиращ линеен списък, представен чрез една връзка. Двойната кутия



ще реализираме чрез шаблона на структурата *elem_link1* с параметър *T*

C079-01 · Копиране: code.html
```
template <typename T = int>
struct elem_link1
```

Оригинал: стр. 80

C080-01 · Копиране: code.html
```
{ T inf;                  // информационна част
  elem_link1<T>* link;    // адресна част
};
```

а линейния списък - чрез шаблона на класа *LList*. Указателя към началото на линейния списък ще означим с *start*, указателя към края - чрез *end*, а указателя към текущ елемент - чрез *current*.

C080-02 · Копиране: code.html
```
template <typename T = int>
class LList
{ public:
    // канонично представяне
    LList();
    ~LList();
    LList(const LList&);
    LList& operator=(const LList&);
    // основни функции
    bool empty() const;
    void iterStart(elem_link1<T>* = NULL);
    elem_link1<T>* iter();
    void toEnd(const T&);
    void insertAfter(elem_link1<T>*, const T&);
    void insertBefore(elem_link1<T>*, const T&);
    void deleteAfter(elem_link1<T>*, T&);
```

C080-02 · продължение · Копиране: code.html
```
    void deleteBefore(elem_link1<T>*, T&);
    void deleteElem(elem_link1<T>*, T&);
    void print() const;
    int length() const;
    void concat(const LList&);
    void reverse();
  private:
    elem_link1<T> *start,   // указател към началото
                  *end,    // указател към края
                  *current;// указател към текущ елемент
    // помощни функции
    void copyList(const LList&);
    void deleteList();
};
```

Член-функциите на каноничното представяне реализират аналогични идеи на тези при стека и опашката. Конструкторът по подразбиране създава празен линеен списък.

Оригинал: стр. 81

C081-01 · Копиране: code.html
```
template <typename T>
LList<T>::LList()
{ start = end = NULL;
}
```

Деструкторът изтрива линеен списък от паметта, ако списъкът не е празен.

C081-02 · Копиране: code.html
```
template <typename T>
LList<T>::~LList()
{ deleteList();
}
```

Конструкторът за присвояване присвоява указания чрез формалния параметър *r* линеен списък на неявния.

C081-03 · Копиране: code.html
```
template <typename T>
LList<T>::LList(const LList<T>& r)
{ copyList(r);
}
```

Операторната функция за присвояване присвоява указания чрез формалния параметър *r* линеен списък на сочения от указателя *this* и създаден преди това линеен списък.

C081-04 · Копиране: code.html
```
template <typename T>
LList<T>& LList<T>::operator=(const LList<T>& r)
{ if (this != &r)
    { deleteList();
      copyList(r);
    }
  return *this;
}
```

За реализирането на каноничното представяне се използват двете помощни член-функции, декларирани в *private* частта на шаблона на класа: *copyList*, която копира указания чрез формалния параметър линеен списък на друго място в паметта, сочено от указателя *this* и *deleteList*, която изтрива неявния линеен списък.

Копиране на списък

C081-05 · Копиране: code.html
```
template <typename T>
void LList<T>::copyList(const LList<T>& r)
{ elem_link1<T> *p = r.start, *q;
  if (r.start)
```

Оригинал: стр. 82

C082-01 · Копиране: code.html
```
    { start = new elem_link1<T>;
      assert(start != NULL);
      end = start;
      while (p)
        { end->inf = p->inf;
          end->link = NULL;
          p = p->link;
          if (p)
            { q = end;
              end = new elem_link1<T>;
              assert(end != NULL);
              q->link = end;
            }
        }
    }
  else
    start = end = NULL;
}
```

Друга, по-лесна реализация на тази член-функция на шаблона е следната

C082-02 · Копиране: code.html
```
template <typename T>
void LList<T>::copyList(const LList<T>& r)
{ start = end = current = NULL;
```

C082-02 · продължение · Копиране: code.html
```
  elem_link1<T> *p = r.start;
  while (p)
    { toEnd(p->inf);
      p = p->link;
    }
}
```

Тя създава празен линеен списък и в него чрез член-функцията *toEnd* (за включване на елемент в края на линеен списък) се включват елементите на указания чрез формалния параметър *r* списък. Обхождането на *r* се осъществява чрез помощния указател *p*, тъй като *r* е от тип *const LList<T>&*.

Изтриване на линеен списък

C082-03 · Копиране: code.html
```
template <typename T>
void LList<T>::deleteList()
{ elem_link1<T> *p;
  while (start)
    { p = start;
```

Оригинал: стр. 83

C083-01 · Копиране: code.html
```
      start = start->link;
      delete p;
    }
  end = NULL;
}
```

Член-функции за работа с итератора

Член-функциите *iterStart* и *iter* реализират операциите с т.нар. *итератор*. Итераторът е абстракция на означението указател към елемент на редица или може да се смята за указател към елемент на контейнер (стекът, опашката, линейният списък са контейнери). В някои случаи итераторите са почти обикновени указатели към обекти, в други - са указател, снабден с индекс и т.н. В реализациите на линеен списък, които ще направим, итераторът ще е указател към двойна или тройна кутия. Общото на всички итератори е тяхната семантика и имената на техните операции. Обикновено операциите са:

++, която приложена към итератор, намира итератор, който сочи следващия елемент;
--, която приложена към итератор, намира итератор, който сочи към предшестващия елемент;
*, която намира елемента, към който сочи итераторът.

В шаблона на класа *LList*, реализиращ линеен списък, представен с една връзка, итераторът е означен с *current*. Операцията ++ е реализирана чрез член-функцията *iter*. Операцията -- не е реализирана. Процедурата *iterStart* се използва за инициализиране на итератора на линеен списък. Тя е с един подразбиращ се параметър с подразбираща се стойност е *NULL*. Обръщението към *iterStart* установява *current* в началото на линейния списък, ако параметърът има нулева стойност и в указан адрес, в случай че параметърът е ненулев. В шаблона операцията * се реализира от

обръщението *iter()->inf*.

Инициализиране на итератора

C083-02 · Копиране: code.html
```
template <typename T>
void LList<T>::iterStart(elem_link1<T> *p)
{ if (p) current = p;
  else current = start;
}
```

Преместването на указателя *current* в следващата позиция се осъществява чрез член-функцията *iter*.

Оригинал: стр. 84

Установяване на итератора в следващата позиция

Член-функцията *iter* връща текущата позиция на итератора и го премества в следващата позиция.

C084-01 · Копиране: code.html
```
template <typename T>
elem_link1<T>* LList<T>::iter()
{ elem_link1<T> *p = current;
  if (current) current = current->link;
  return p;
}
```

Включване на елемент в линеен списък

Тази операция ще реализираме чрез следните три член-функции:

Включване на елемент в края на списъка

C084-02 · Копиране: code.html
```
template <typename T>
void LList<T>::toEnd(const T& x)
{ elem_link1<T>* p = end;
  end = new elem_link1<T>;
  assert(end != NULL);
  end->inf = x;
  end->link = NULL;
  if (p) p->link = end;
  else start = end;
}
```

Включване на елемент след указан елемент

Член-функцията *insertAfter* включва елемента *x* след елемента, сочен от указателя *p*. Предполагаме, че *p* е ненулев и сочи елемент на неявния линеен списък.

C084-03 · Копиране: code.html
```
template <typename T>
void LList<T>::insertAfter(elem_link1<T>* p, const T& x)
{ elem_link1<T> *q = new elem_link1<T>;
  assert(q != NULL);
  q->inf = x;
  q->link = p->link;
  if (p == end) end = q;
  p->link = q;
}
```

Оригинал: стр. 85

Включване на елемент пред указан елемент

Член-функцията *insertBefore* включва елемента *x* пред елемента, сочен от указателя *p*. Предполагаме, че *p* е ненулев и сочи елемент на неявния линеен списък.

C085-01 · Копиране: code.html
```
template <typename T>
void LList<T>::insertBefore(elem_link1<T>* p, const T& x)
{ elem_link1<T>* q = new elem_link1<T>;
  assert(q != NULL);
  *q = *p;
  if (p == end) end = q;
  p->inf = x;
  p->link = q;
}
```

Изключване на елемент от линеен списък

Тази операция ще реализираме чрез следните три член-функции:

Изключване на елемент след указан елемент

Член-функцията *deleteAfter* изключва елемента, намиращ се след елемента, сочен от указателя *p*, и го запомня в параметъра *x*. Предполагаме, че *p* е ненулев и сочи елемент на неявния линеен списък.

C085-02 · Копиране: code.html
```
template <typename T>
void LList<T>::deleteAfter(elem_link1<T>* p, T& x)
{ if (p->link)
    { elem_link1<T>* q = p->link;
      x = q->inf;
      p->link = q->link;
      if (q == end) end = p;
      delete q;
    }
  else
    { cout << "There is not an element after p.\n";
      exit(1);
```

C085-02 · продължение · Копиране: code.html
```
    }
}
```

Изключване на указан елемент

Член-функцията *deleteElem* изключва елемента, сочен от указателя *p* и го запомня в параметъра *x*. Предполагаме, че *p* е ненулев и сочи елемент на неявния списък.

Оригинал: стр. 86

C086-01 · Копиране: code.html
```
template <typename T>
void LList<T>::deleteElem(elem_link1<T>* p, T& x)
{ if (p == start)
    { x = p->inf;
      if (start == end)
        { start = end = NULL;
        }
      else start = start->link;
      delete p;
    }
  else
    { elem_link1<T>* q = start;
      while (q->link != p) q = q->link;
      deleteAfter(q, x);
    }
}
```

Изключване на елемент пред указан елемент

Член-функцията *deleteBefore* изключва елемента пред елемента, сочен от указателя *p*, и го запомня в параметъра *x*. Предполагаме, че *p* е ненулев и сочи елемент на неявния списък.

C086-02 · Копиране: code.html
```
template <typename T>
void LList<T>::deleteBefore(elem_link1<T>* p, T& x)
{ if (p != start)
    { elem_link1<T>* q = start;
      while (q->link != p) q = q->link;
      deleteElem(q, x);
    }
  else
    { cout << "There is not an element before p.\n";
      exit(1);
    }
}
```

Член-функции за достъп

Проверка дали линеен списък е празен

C086-03 · Копиране: code.html
```
template <typename T>
bool LList<T>::empty() const
{ return start == NULL;
}
```

Оригинал: стр. 87

Извеждане на елементите на линеен списък

Извеждането на елементите на линеен списък се реализира чрез последователното им обхождане без разрушаване на списъка.

C087-01 · Копиране: code.html
```
template <typename T>
void LList<T>::print() const
{ elem_link1<T> *p = start;
  while (p)
    { cout << p->inf << " ";
      p = p->link;
    }
  cout << endl;
}
```

Намиране на дължината на линеен списък

Член-функцията *length* обхожда и преброява елементите на линеен списък, без да го разрушава.

C087-02 · Копиране: code.html
```
template <typename T>
int LList<T>::length() const
{ int n = 0;
  elem_link1<T> *p = start;
  while (p)
    { n++;
      p = p->link;
    }
  return n;
}
```

Конкатенация на линейни списъци

Следващата член-функция на шаблона на класа *LList<T>* реализира конкатенация на неявния с указания като формален параметър линеен списък. В резултат в края на неявния линеен списък са включени елементите на дадения линеен списък *r*. Така неявният линеен списък е променен (носи резултата).

C087-03 · Копиране: code.html
```
template <typename T>
void LList<T>::concat(const LList<T>& r)
{ elem_link1<T> *p = r.start;
  while (p)
    { toEnd(p->inf);
      p = p->link;
```

Оригинал: стр. 88

C088-01 · Копиране: code.html
```
    }
}
```

Забележка. Следващата дефиниция на член-функцията *concat*

C088-02 · Копиране: code.html
```
template <typename T>
void LList<T>::concat(const LList& r)
{ end->link = r.start;
  end = r.end;
}
```

е неправилна, тъй като линейният списък *r* е поделен.

Обръщане на елементите на линеен списък

Следващата член-функция на шаблона на класа *LList* обръща елементите на неявния линеен списък. За целта използва помощен линеен списък *r*, с който оперира като със стек.

C088-03 · Копиране: code.html
```
template <typename T>
void LList<T>::reverse()
{ LList<T> r;
  iterStart();
  elem_link1<T>* p = iter();
  if (p)
    { r.toEnd(p->inf);
      p = p->link;
      while (p)
        { r.insertBefore(r.start, p->inf);
          p = p->link;
        }
    }
  *this = r;
}
```

Реализацията на *reverse*, дадена по-долу, обръща елементите на непразен линеен списък, зададен чрез неявния параметър, без да прави негово копие в паметта. Тя поправя връзките в него, така че първите елементи да станат последни и обратно.

C088-04 · Копиране: code.html
```
template <typename T>
void LList<T>::reverse()
{ elem_link1<T> *p, *q, *temp;
  p = start;
```

Оригинал: стр. 89

C089-01 · Копиране: code.html
```
  if (p)
    { q = NULL;
      temp = start;
      start = end;
      end = temp;
      while (p != start)
        { temp = p->link;
          p->link = q;
          q = p;
          p = temp;
        }
      p->link = q;
    }
}
```

Шаблона на класа *LList* записваме във файла *LList1.cpp* и ще използваме в следващите задачи.

Задача 9. Да се напише програма, която:
- създава линеен списък *L1*, съдържащ целите числа 1, 2, 3, 4 и 5;
- създава линеен списък *L2*, съдържащ целите числа 6, 7 и 8;
- създава линеен списък *L3*, равен на списъка *L1*;
- извежда елементите на списъците *L1*, *L2* и *L3*;
- обръща елементите на списъците *L1*, *L2* и *L3* и ги извежда;
- създава нов списък *L*, който е равен на конкатенацията на елементите на обърнатите списъци *L1*, *L2* и *L3*, след което извежда *L*.

C089-02 · Копиране: code.html
```
// Program Zad9.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "LList1.cpp"

int main()
{ // създаване на празните списъци L1 и L2
  LList<int> L1, L2;
  // включване на елементите 1, 2, 3, 4, 5 в списъка L1
  int i;
  for (i = 1; i <= 5; i++)
    L1.toEnd(i);
```

Оригинал: стр. 90

C090-01 · Копиране: code.html

```
  // включване на елементите 6, 7, 8 в списъка L2
  for (i = 6; i <= 8; i++)
    L2.toEnd(i);
  // създаване на списъка L3, равен на L1
  LList<int> L3 = L1;
  // извеждане на елементите на L1, L2 и L3
  cout << "L1= "; L1.print();
  cout << "L2= "; L2.print();
  cout << "L3= "; L3.print();
  // обръщане на елементите на L1, L2 и L3
  L1.reverse();
  L2.reverse();
  L3.reverse();
  // извеждане на екрана на обърнатите списъци
  cout << "L1-rev: "; L1.print();
  cout << "L2-rev: "; L2.print();
  cout << "L3-rev: "; L3.print();
  // създаване на списъка L, равен на конкатенацията
  // на списъците L1, L2 и L3
  LList<int> L;
  L.concat(L1);
  L.concat(L2);
  L.concat(L3);
  // извеждане на списъка L
  cout << "L= "; L.print();
  return 0;
}
```

2.3.3. Приложения на свързаните списъци, представени с една връзка

2.3.3.1. Реализация на шаблон на стек чрез шаблона LList<T>

Структурата от данни стек може да се разглежда като частен случай на структурата от данни линеен списък. Следващият шаблон на клас реализира стек чрез наследяване на шаблона на класа *LList<T>*. Макар че е излишно, за пълнота ще дефинираме и каноничното представяне на стека.

C090-02 · Копиране: code.html

```
template <typename T = int>
class stack : public LList<T>
{ public:
```

Оригинал: стр. 91

C091-01 · Копиране: code.html

```
    // канонично представяне
    stack();
    ~stack();
    stack(const stack&);
    stack& operator=(const stack&);
    // основни операции
    bool empty() const;
    void push(const T&);
    void pop(T&);
    void top(T&);
    void print();
    int length();
};

template <typename T>
stack<T>::stack() : LList<T>()
{}

template <typename T>
stack<T>::~stack()
{}

template <typename T>
stack<T>::stack(const stack<T>& r) : LList<T>(r)
{}

template <typename T>
stack<T>& stack<T>::operator=(const stack<T>& r)
{ if (this != &r)
    { LList<T>::operator=(r);
    }
  return *this;
}

template <typename T>
bool stack<T>::empty() const
{ return LList<T>::empty();
}

template <typename T>
```

Оригинал: стр. 92

C092-01 · Копиране: code.html

```
void stack<T>::push(const T& x)
{ if (!empty())
    { iterStart();
      elem_link1<T>* p = iter();
      insertBefore(p, x);
    }
  else
    toEnd(x);
}

template <typename T>
void stack<T>::pop(T& x)
{ if (!empty())
    { iterStart();
      elem_link1<T>* p = iter();
      deleteElem(p, x);
    }
  else
    { cout << "The stack is empty.\n";
      exit(1);
    }
}

template <typename T>
void stack<T>::top(T& x)
{ if (!empty())
    { iterStart();
      elem_link1<T>* p = iter();
      x = p->inf;
    }
  else
    { cout << "The stack is empty.\n";
      exit(1);
    }
}

template <typename T>
void stack<T>::print()
{ LList<T>::print();
}
```

Оригинал: стр. 93

C093-01 · Копиране: code.html

```
template <typename T>
int stack<T>::length()
{ return LList<T>::length();
}
```

2.3.3.2. Реализация на шаблон на опашка чрез шаблона LList<T>

Структурата от данни опашка може да се разглежда като частен случай на структурата от данни линеен списък. Следващият шаблон на клас реализира опашка чрез наследяване на шаблона на класа *LList<T>*. Макар че е излишно, за пълнота ще дефинираме и каноничното представяне на опашката.

C093-02 · Копиране: code.html

```
template <typename T = int>
class queue : public LList<T>
{ public:
    // канонично представяне
    queue();
    ~queue();
    queue(const queue&);
    queue& operator=(const queue&);
    // основни функции
    bool empty() const;
    void push(T const&);
    void pop(T&);
    void head(T&);
    void print();
};
template <typename T>
queue<T>::queue() : LList<T>()
{}

template <typename T>
queue<T>::~queue()
{}

template <typename T>
queue<T>::queue(const queue<T>& r) : LList<T>(r)
{}

template <typename T>
queue<T>& queue<T>::operator=(const queue<T>& r)
```

Оригинал: стр. 94

C094-01 · Копиране: code.html

```
{ if (this != &r)
    { LList<T>::operator=(r);
    }
  return *this;
}

template <typename T>
bool queue<T>::empty() const
{ return LList<T>::empty();
}

template <typename T>
void queue<T>::push(T const& x)
{ LList<T>::toEnd(x);
}

template <typename T>
void queue<T>::pop(T& x)
{ if (!empty())
    { iterStart();
      elem_link1<T>* p = iter();
      deleteElem(p, x);
    }
  else
    { cout << "The queue is empty.\n";
      exit(1);
    }
}

template <typename T>
void queue<T>::head(T& x)
{ if (!empty())
    { iterStart();
      elem_link1<T>* p = iter();
      x = p->inf;
    }
  else
    { cout << "The queue is empty.\n";
      exit(1);
    }
```

Оригинал: стр. 95

C095-01 · Копиране: code.html

```
}

template <typename T>
void queue<T>::print()
{ LList<T>::print();
}
```

2.3.3.3. Сортиране и сливане на линейни списъци

В тази част ще реализираме метода на пряката селекция, описан и реализиран за структурите от данни масив и файл в т. 4.1.1 (глава 4). Освен него ще реализираме сливане на сортирани линейни списъци, както и метода сортиране чрез сливане, описани и реализирани в т. 4.1.7 (глава 4). И при трите метода елементите на линейния списък са от тип *T*, който тип допуска сравнение на елементите му за <.

Сортиране по метода на пряката селекция

Задача 10. Да се дефинира шаблон на функция, който реализира метода на пряката селекция за сортиране на линейни списъци с елементи от тип *T*.

C095-02 · Копиране: code.html
```
template <typename T>
void sortList(LList<T>& L)
{ if (L.empty()) return;
  // линейният списък не е празен
  elem_link1<T> *mp,     // указател към минимален елемент
                *p;     // указател към текущ елемент
  L.iterStart(); p = L.iter();
  while (p->link)
    { T min = p->inf;
      mp = p;
      elem_link1<T> *q = p->link;
      while (q)
        { if (q->inf < min)
            { mp = q;
              min = q->inf;
            }
          q = q->link;
        }
      min = mp->inf;
      mp->inf = p->inf;
```

Оригинал: стр. 96

C096-01 · Копиране: code.html
```
      p->inf = min;
      p = p->link;
    }
}
```

Записваме този шаблон на функция във файл с име *sortList.cpp*.

Сливане на сортирани линейни списъци

Задача 11. Дадени са сортираните във възходящ ред линейни списъци *L1* и *L2* с елементи от тип *T*. Да се дефинира шаблон на функция *mergeLists*, която слива *L1* и *L2* и връща резултата от сливането.

C096-02 · Копиране: code.html

```
template <typename T>
LList<T> mergeLists(LList<T> L1, LList<T> L2)
{ LList<T> L;        // празен списък, който ще съдържа
                    // резултата от сливането
  // установяване на указател p към първия елемент на списъка L1
  // установяване на указател q към първия елемент на списъка L2
  L1.iterStart();
  L2.iterStart();
  elem_link1<T> *p = L1.iter(),
                *q = L2.iter();
  // докато и двата списъка L1 и L2 не са празни
  // се обхождат елементите им, сравняват се сочените
  // от указателите p и q елементи и по-малкият елемент
  // се записва в списъка L
  while (p && q)
    if (p->inf < q->inf)
      { L.toEnd(p->inf);
        p = L1.iter();
      }
    else
      { L.toEnd(q->inf);
        q = L2.iter();
      }
  // след изчерпване на елементите на единия списък,
  // елементите на другия се прекопират в списъка L
  if (q)
    while (q)
      { L.toEnd(q->inf);
```

Оригинал: стр. 97

C097-01 · Копиране: code.html

```
        q = L2.iter();
      }
  else
    while (p)
      { L.toEnd(p->inf);
        p = L1.iter();
      }
  return L;
}
```

Записваме този шаблон на функция във файл с име *mergeLists.cpp*.

Сортиране чрез сливане

Този вид сортиране се осъществява по начина, описан в т. 4.1.7 (глава 4). Линейният списък се разделя на две половини. Всяка половинка се сортира и накрая сортираните половинки се сливат.

Задача 12. Да се напише шаблон на функция *mergeSort*, който реализира метода сортиране чрез сливане за линейни списъци.

C097-02 · Копиране: code.html
```
template <typename T>
void mergeSort(LList<T>& L)
{ LList<T> L1, L2;
  L.iterStart();
  elem_link1<T> *p = L.iter();
  if (!p || p->link == NULL) return;
  while (p)
    { L1.toEnd(p->inf);
      p = p->link;
      if (p)
        { L2.toEnd(p->inf);
          p = p->link;
        }
    }
  mergeSort(L1);
  mergeSort(L2);
  L = mergeLists(L1, L2);
}
```

Записваме този шаблон на функция във файл с име *mergeSort.cpp*.

Оригинал: стр. 98

Задача 13. Да се напише програма, която създава линеен списък от линейни списъци от имена на хора, сортира лексикографски списъците на списъка, след което ги слива и извежда получения линеен списък.

В програмата са използвани шаблоните на функциите *sortList* и *mergeLists*. Дефинирана е структурата *people*, която има за член-данна символен низ, означаващ име на човек, и в която операторът за сравнение < е предефиниран за символни низове. Реализирани са две специализации на член-функцията *print* на шаблона на класа *LList*: едната за извеждане на списък от обекти на структурата *people* и другата за извеждане на списък от списъци от обекти на класа *people*.

C098-01 · Копиране: code.html
```
// Program Zad13.cpp
#include <iostream>
#include <iomanip>
#include <cassert>
#include <cstring>
using namespace std;

#include "LList1.cpp"
#include "sortList.cpp"
#include "mergeLists.cpp"

struct people
{ char name[32];    // име на човек
  bool operator<(const people& p) const
  { return strcmp(name, p.name) < 0;
  }
```

C098-01 · продължение · Копиране: code.html

```
};

typedef LList<people> peopleList;
typedef LList<peopleList> list_of_lists;

void LList<people>::print() const
{ elem_link1<people> *p = start;
  while (p)
  { cout << p->inf.name << endl;
    p = p->link;
  }
```

Оригинал: стр. 99

C099-01 · Копиране: code.html

```
  cout << endl;
}

void LList<peopleList>::print() const
{ elem_link1<peopleList> *p = start;
  while (p)
  { p->inf.print();
    p = p->link;
  }
}

int main()
{ // създаване на празен списък от
  // списъци от имена на хора
  list_of_lists LL;
  cout << "Number of the sublists: ";
  int n; cin >> n;
  for (int k = 1; k <= n; k++)
    { // създаване на празен списък от имена на хора
      peopleList L;
      people s;
      // въвеждане на броя на елементите на списъка
      cout << "Number of the elements of the list: ";
      int p; cin >> p;
      // игнориране на символа за край на ред
      cin.ignore();
      // включване на имена в списъка L
      for (int i = 1; i <= p; i++)
        { cout << "s= "; cin.getline(s.name, 32);
          L.toEnd(s);
        }
      // включване на списъка L в списъка от списъци LL
      LL.toEnd(L);
    }
  // извеждане на списъка от списъци от имена на хора
  LL.print();
  // установяване на указател p към началото на
  // списъка от списъци от имена на хора LL
  LL.iterStart();
```

Оригинал: стр. 100

C100-01 · Копиране: code.html

```
  elem_link1<peopleList> *p = LL.iter();
  // обхождане на LL и сортиране на всеки списък от имена
  while (p)
    { sortList(p->inf);
      p = LL.iter();
    }
  // извеждане на LL след сортирането на
  // всеки от подсписъците му
  LL.print();
  // сливане на подсписъците на LL в списъка L
  LL.iterStart();
  p = LL.iter();
  peopleList L = p->inf;
  p = LL.iter();
  while (p)
    { L = mergeLists(L, p->inf);
      p = LL.iter();
    }
  // извеждане на списъка L
  L.print();
  return 0;
}
```

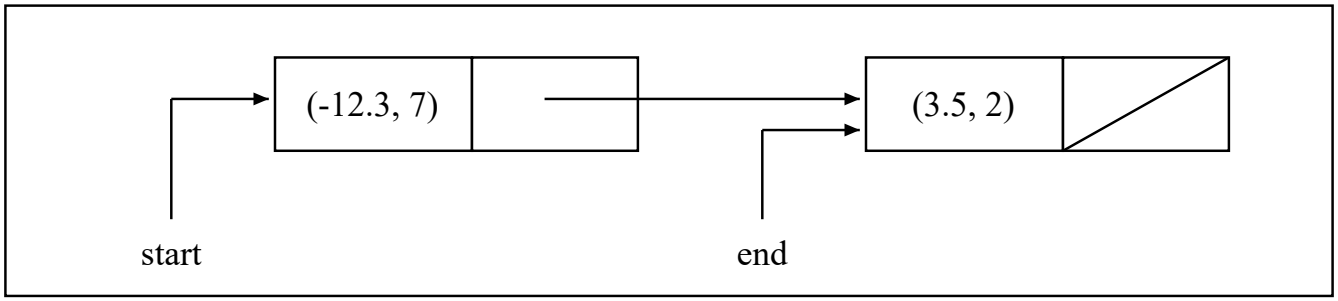
2.3.3.4. Представяне на полином чрез линеен списък. Работа с полиноми

Полиномът $a_nx^n + a_{n-1}x^{n-1} +... + a_1x + a_0$ най-често се представя по два начина: чрез едномерен масив и чрез линеен списък. Първото представяне се избира, когато е известно, че голям брой коефициенти на полинома са различни от 0. В противен случай е по-добре да се използва представяне чрез линеен списък.

За представяне на полином чрез линеен списък избираме елементите на списъка да са обекти на структура с две полета: *coefficient*, реално число означаващо коефициента и *power*, цяло неотрицателно число, означаващо степента на съответния едночлен. При това, ако коефициентът a_k $(0 \le k \le n)$ е равен на 0, съответният едночлен не се включва в списъка.

Например полиномът $-12.3x^7 + 3.5x^2$ представяме по следния начин:

Оригинал: стр. 101



Структурата *monomial* представя едночлен на полинома.

C101-01 · Копиране: code.html
```
struct monomial
{ double coefficient;    // коефициент
  int power;             // степен
};
```

Чрез дефиницията

C101-02 · Копиране: code.html
```
typedef LList<monomial> polynomial;
```

е определен тип *polynomial*, представящ полином.

Създаване и извеждане на полиноми

Процедурата *create* създава полинома *p*. При създаването изискваме едночлените да се въвеждат в намаляващ ред на степените си.

C101-03 · Копиране: code.html
```
void create(polynomial& p)
{ char flag;
  do
  { monomial mono;
    cout << "coefficient: "; cin >> mono.coefficient;
    cout << "power: "; cin >> mono.power;
    p.toEnd(mono);
    cout << "Input y/n: ";
    cin >> flag;
  } while (flag == 'y');
}
```

Извеждането на полином ще реализираме чрез следната специализация на член-функцията *print* на шаблона на класа *LList*

C101-04 · Копиране: code.html
```
void LList<monomial>::print() const
{ elem_link1<monomial> *p = start;
  while (p)
    { if (p->inf.coefficient > 0) cout << " +";
      else cout << " ";
```

Оригинал: стр. 102

C102-01 · Копиране: code.html
```
      cout << p->inf.coefficient << "x^" << p->inf.power;
      p = p->link;
    }
  cout << endl;
}
```

Събиране на полиноми

За реализиране на операцията събиране на два полинома се сканира всеки от полиномите. Ако текущите едночлени са от една и съща степен, новият едночлен е от същата степен и с коефициент, равен на сбора от коефициентите на едночлените. Ако това не е така, новият едночлен съвпада с едночлена от по-висока степен. Конструираният едночлен се включва в полинома, който ще съдържа сумата. При достигане на края на линейния списък, представящ единия полином, останалата част на другия се копира в полинома, съдържащ сумата. Процедурата *addPolynomials* реализира описаните действия. Параметърът *r* в нея съдържа сумата на полиномите *p* и *q* и началната му стойност трябва да е празен списък. Освен това тя следи да не се включи едночлен с нулев коефициент.

C102-02 · Копиране: code.html

```
void addPolynomials(polynomial p, polynomial q,
                    polynomial& r)
{ p.iterStart();
  q.iterStart();
  elem_link1<monomial> *pp = p.iter(),
                       *qq = q.iter();
  while (pp && qq)
    if (pp->inf.power > qq->inf.power)
      { r.toEnd(pp->inf);
        pp = pp->link;
      }
    else
      if (qq->inf.power > pp->inf.power)
        { r.toEnd(qq->inf);
          qq = qq->link;
        }
      else
        { monomial mono;
          mono.power = pp->inf.power;
          mono.coefficient = pp->inf.coefficient +
                             qq->inf.coefficient;
          pp = pp->link;
          qq = qq->link;
```

Оригинал: стр. 103

C103-01 · Копиране: code.html

```
          if (mono.coefficient != 0) r.toEnd(mono);
        }
  while (qq)
    { r.toEnd(qq->inf);
      qq = qq->link;
    }
  while (pp)
    { r.toEnd(pp->inf);
      pp = pp->link;
    }
}
```

Произведение на полиноми

Процедурата *multPolynomials* намира полинома *r*, равен на произведението на полиномите *p* и *q*. За реализирането на тази операция над полиноми процедурата използва помощната функция *multPolynomialMonomial*, която намира произведението на полином с едночлен.

C103-02 · Копиране: code.html

```
void multPolynomialMonomial(polynomial p, monomial q,
                            polynomial& r)
{ p.iterStart();
  elem_link1<monomial> *pp = p.iter();
  while (pp)
    { monomial mono;
      mono.power = pp->inf.power + q.power;
      mono.coefficient = pp->inf.coefficient *
                         q.coefficient;
      pp = pp->link;
      if (mono.coefficient != 0) r.toEnd(mono);
    }
}
```

Процедурата *multPolynomials* умножава всеки едночлен на полинома с по-малък брой едночлени с другия полином и събира получените полиноми. Параметърът *r* съдържа произведението на полиномите *p* и *q* и началната му стойност трябва да е празен списък.

C103-03 · Копиране: code.html

```
void multPolynomials(polynomial p, polynomial q,
                     polynomial& r)
{ int n = p.length(),
      m = q.length();
  if (n <= m)
```

C104-01 · Копиране: code.html
```
    { p.iterStart();
      elem_link1<monomial> *pp = p.iter();
      monomial mono = pp->inf;
      pp = pp->link;
      multPolynomialMonomial(q, mono, r);
      while (pp)
        { mono = pp->inf;
          polynomial s, r1;
          multPolynomialMonomial(q, mono, s);
          addPolynomials(r, s, r1);
          r = r1;
          pp = pp->link;
        }
    }
  else
    { q.iterStart();
      elem_link1<monomial> *pp = q.iter();
      monomial mono = pp->inf;
      pp = pp->link;
      multPolynomialMonomial(p, mono, r);
      while (pp)
        { mono = pp->inf;
          polynomial s, r1;
          multPolynomialMonomial(p, mono, s);
          addPolynomials(r, s, r1);
          r = r1;
          pp = pp->link;
        }
    }
}
```

2.3.3.5. Операции над множества, представени чрез линейни списъци

В тази част ще реализираме операциите обединение, сечение и разлика на множества, представени чрез линейни списъци. За целта ще използваме помощния шаблон на булева функция *member*, който проверява дали елементът *x* от тип *T* принадлежи на множество, представено чрез линейния списък *L*. Предполагаме, че за типа *T* е дефиниран операторът !=. Записваме шаблона във файла *member.cpp*.

C104-02 · Копиране: code.html
```
template <typename T>
bool member(const T& x, LList<T>& l)
```

C105-01 · Копиране: code.html
```
{ l.iterStart();
  elem_link1<T> *p = l.iter();
  while (p && p->inf != x) p = p->link;
  return p != NULL;
}
```

Въпреки че дефиницията на множество предполага, че елементите на множеството са различни, в следващите реализации на операциите обединение, сечение и разлика на множества се извършва проверка за това.

Обединение на множества

Обединението на две множества е множество, съдържащо елементите, които принадлежат на поне едно от множествата. Шаблонът на функцията *unionSets*, дефиниран по-долу, намира обединението на множествата *p* и *q*, представени чрез свързани списъци.

C105-02 · Копиране: code.html
```
template <typename T>
LList<T> unionSets(LList<T> p, LList<T> q)
{ LList<T> r; // ще съдържа резултата от обединението на p и q
  p.iterStart();
  q.iterStart();
  elem_link1<T> *pp = p.iter(),
                *qq = q.iter();
  while (pp)
    { if (!member(pp->inf, r)) r.toEnd(pp->inf);
      pp = pp->link;
    }
  while (qq)
    { if (!member(qq->inf, r)) r.toEnd(qq->inf);
      qq = qq->link;
    }
  return r;
}
```

Записваме шаблона във файла *unionSets.cpp*.

Сечение на множества

Сечението на две множества е множество, съдържащо елементите, които принадлежат и на двете множества. Шаблонът на функцията *intersectionSets*, дефиниран по-долу, намира сечението на множествата *p* и *q*, представени чрез свързани списъци.

Оригинал: стр. 106

C106-01 · Копиране: code.html
```
template <typename T>
LList<T> intersectionSets(LList<T> p, LList<T> q)
{ LList<T> r; // ще съдържа резултата от сечението на p и q
  p.iterStart();
  q.iterStart();
  elem_link1<T> *pp = p.iter(),
                *qq = q.iter();
  while (pp)
    { if (!member(pp->inf, r) && member(pp->inf, q))
        r.toEnd(pp->inf);
      pp = pp->link;
    }
  return r;
}
```

Записваме шаблона във файла *intersectionSets.cpp*.

Разлика на множества

Разликата $A \setminus B$ на множествата A и B е множество, съдържащо елементите, които принадлежат на A, но не принадлежат на B. Шаблонът на функцията *differenceSets*, дефиниран по-долу, намира разликата на множествата p и q, представени чрез свързани списъци.

C106-02 · Копиране: code.html
```
template <typename T>
LList<T> differenceSets(LList<T> p, LList<T> q)
{ LList<T> r; // ще съдържа резултата от разликата на p и q
  p.iterStart();
  q.iterStart();
  elem_link1<T> *pp = p.iter(),
                *qq = q.iter();
  while (pp)
    { if (!member(pp->inf, r) && !member(pp->inf, q))
        r.toEnd(pp->inf);
      pp = pp->link;
    }
  return r;
}
```

Записваме шаблона във файла *differenceSets.cpp*.

Оригинал: стр. 107

Задача 14. Дадени са множествата от цели числа A, B, C и D. Да се намери множеството $(A \setminus B) \cup (C \cap D)$.

C107-01 · Копиране: code.html
```
// Program Zad14.cpp
#include <iostream>
#include <cassert>
```

C107-01 · продължение · Копиране: code.html

```
using namespace std;

#include "LList1.cpp"
#include "member.cpp"
#include "unionSets.cpp"
#include "intersectionSets.cpp"
#include "differenceSets.cpp"

typedef LList<int> intSet;

// създаване на множество от цели числа
void createSet(intSet& L)
{ int s;
  cout << "number= ";
  int p; cin >> p;
  for (int i = 1; i <= p; i++)
    { cout << "elem= "; cin >> s;
      L.toEnd(s);
    }
}

int main()
{ intSet A, B, C, D, E, F, R;
  cout << "Set A\n";
  createSet(A);
  cout << "Set B\n";
  createSet(B);
  cout << "Set C\n";
  createSet(C);
  cout << "Set D\n";
  createSet(D);
  cout << "A: "; A.print();
  cout << "B: "; B.print();
  cout << "C: "; C.print();
```

Оригинал: стр. 108

C108-01 · Копиране: code.html

```
  cout << "D: "; D.print();
  E = differenceSets(A, B);
  F = intersectionSets(C, D);
  R = unionSets(E, F);
  R.print();
  return 0;
}
```

2.3.4. Функции от по-висок ред за работа с линейни списъци, представени чрез една връзка

Функция accumulate

Нека L е линеен списък с елементи от тип T, а *op* е лявоасоциативна операция от вида:

op: T × T → T

с начална стойност *null_val*. Ще дефинираме шаблон на функция от по-висок ред *accumulate*, чрез който да се реализира натрупване на елементите на списъка L чрез операцията *op*. Например, ако L се състои от елементите $(a_1, a_2, ..., a_n)$, резултатът от изпълнението на *accumulate(op, null_val, L)* ще е $(... ((null_val\ op\ a_1)\ op\ a_2) ...\ op\ a_n)$.

C108-02 · Копиране: code.html
```
template <typename T>
T accumulate(T (*op)(T, T), T null_val, LList<T>& L)
{ T s = null_val;
  L.iterStart();
  elem_link1<T> *p = L.iter();
  while (p)
    { s = op(s, p->inf);
      p = p->link;
    }
  return s;
}
```

Записваме шаблона във файла *accumulate.cpp*.

Задача 15. Като се използва функцията от по-висок ред *accumulate*, да се напише програма, която намира сумата от елементите на списък от цели числа.

C108-03 · Копиране: code.html
```
// Program Zad15.cpp
#include <iostream>
#include <cassert>
using namespace std;
```

C109-01 · Копиране: code.html
```
#include "LList1.cpp"
#include "accumulate.cpp"

typedef LList<int> intList;

void createList(intList& L)
{ int s;
  cout << "number= ";
  int p; cin >> p;
  for (int i = 1; i <= p; i++)
    { cout << "elem= "; cin >> s;
      L.toEnd(s);
    }
}

int sum(int x, int y)
{ return x+y;
}

int main()
{ intList L;
  cout << "List\n";
  createList(L);
  cout << "L: "; L.print();
  cout << accumulate(sum, 0, L) << endl;
  return 0;
}
```

Функция map

Нека L е линеен списък с елементи от тип T, а f е едноаргументна функция от вида:

$f: T \rightarrow T$

Ще дефинираме шаблон на функция от по-висок ред *map* за намиране на списък, който е резултатът от прилагането на функцията f над всеки от елементите на L. Например, ако L се състои от елементите $(a_1, a_2, ..., a_n)$, резултатът от изпълнението на $map(f, L)$ ще е списъкът $(f(a_1), f(a_2), ..., f(a_n))$.

C109-02 · Копиране: code.html
```
template <typename T>
LList<T> map(T (*f)(T), LList<T>& L)
{ L.iterStart();
```

C110-01 · Копиране: code.html
```
  elem_link1<T> *p = L.iter();
  LList<T> L1;
  while (p)
    { L1.toEnd(f(p->inf));
      p = p->link;
    }
  return L1;
}
```

Записваме шаблона във файла *map.cpp*. Ако е даден линеен списък от реални числа *L*, в резултат от изпълнението на обръщението *map(sin, L)* се получава линеен списък от синусите на елементите на *L*.

Функция filter

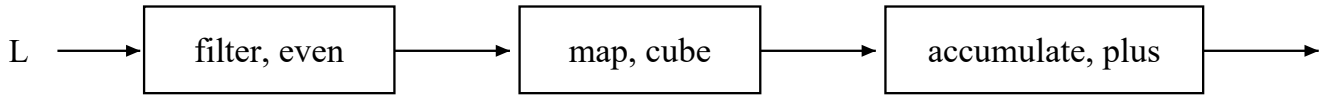
Нека *L* е списък с елементи от тип *T*, а *pred* е предикат. Ще дефинираме шаблон на функция от по-висок ред *filter*, чрез който се намира линеен списък, състоящ се от онези елементи на *L*, за които е в сила *pred*.

C110-02 · Копиране: code.html
```
template <typename T>
LList<T> filter(bool (*pred)(T), LList<T>& L)
{ LList<T> L1;
  L.iterStart();
  elem_link1<T> *p = L.iter();
  while (p)
    { if (pred(p->inf)) L1.toEnd(p->inf);
      p = p->link;
    }
  return L1;
}
```

Ако *L* е линеен списък от цели числа, чрез обръщението *filter(odd, L)*, където *odd* е предикат, установяващ дали аргументът му е нечетно число, се получава линеен списък от нечетните елементи на списъка *L*. Записваме шаблона във файла *filter.cpp*.

Задача 16. Да се напише програма, която намира сумата от кубовете на четните числа на даден линеен списък *L* от цели числа.

За решаването на тази задача ще реализираме следните стъпки:



Оригинал: стр. 111

C111-01 · Копиране: code.html

```
// Program Zad16.cpp
#include <iostream>
#include <cassert>
using namespace std;

#include "LList1.cpp"
#include "accumulate.cpp"
#include "map.cpp"
#include "filter.cpp"

int plus(int a, int b)
{ return a + b;
}

bool even(int x)
{ return x%2 == 0;
}

int cube(int x)
{ return x*x*x;
}

int main()
{ LList<int> L, L1;
  // създаване на линеен списък L от цели числа
  int k;
  cout << "Length of the list: "; cin >> k;
  for (int j = 1; j <= k; j++)
    { cout << "Element: ";
      int a; cin >> a;
      L.toEnd(a);
    }
  L.print(); // извеждане на списъка L
  // намиране на списъка L1 от кубовете
  // на четните числа на списъка L
  L1 = map(cube, filter(even, L));
  L1.print();
  // намиране и извеждане на сумата от елементите на L1
  cout << accumulate(plus, 0, L1) << endl;
  return 0;
}
```

Оригинал: стр. 112

Задача 17. Да се напише програма, която създава линеен списък от линейни списъци от реални числа, след което конкатенира списъците, съставящи линейния списък.

C112-01 · Копиране: code.html

```
// Program Zad17.cpp
#include <iostream>
#include <cassert>
using namespace std;

#include "LList1.cpp"
#include "accumulate.cpp"

typedef LList<double> doubleList;

doubleList concat(doubleList L1, doubleList L2)
{ doubleList L;
  L.concat(L1);
  L.concat(L2);
  return L;
}

int main()
{ // създаване на линеен списък от
  // линейни списъци от реални числа
  LList<doubleList> L;
  int n;
  cout << "n= "; cin >> n; // брой на елементите на L
  for (int i = 1; i <= n; i++)
  { doubleList m;
    cout << "k= "; int k; // брой на елементите на i-тия
                          // линеен списък от реални числа
    cin >> k;
    // създаване на i-тия линеен списък от реални числа
    for (int j = 1; j <= k; j++)
      { cout << "elem: ";
        double elem; cin >> elem;
        m.toEnd(elem);
      }
    // включване на i-тия линеен списък от реални числа
    // в списъка от списъци от реални числа L
```

Оригинал: стр. 113

C113-01 · Копиране: code.html

```
    L.toEnd(m);
  }
  // конкатенация на списъците, съставящи L
  doubleList L2;
  doubleList L1 = accumulate(concat, L2, L);
  L1.print();
  return 0;
}
```

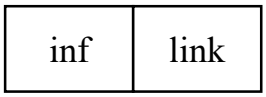
2.3.5. Циклични линейни списъци

В редица случаи се налага да се използват разновидности на линейните списъци. Например, ако искаме операцията включване да се осъществява само в началото или само в края на линейния списък, не е нужно да поддържаме два указателя *start* и *end* към началото и края съответно. В този и в някои други случаи се използват т.нар. *циклични* или *кръгови линейни списъци*.

В тази част ще реализираме цикличен линеен списък, представен с една връзка. За целта ще използваме само един указател *start*, указващ последния елемент на списъка, както е показано на фиг. 2.7.

2.3.5.1. Реализация на цикличен линеен списък

Ще дефинираме шаблон на клас *cyclicList*, реализиращ линеен списък, представен циклично чрез една връзка. Двойната кутия



ще реализираме чрез шаблона на структурата *elem_cyclic* с параметър *T*

C113-02 · Копиране: code.html
```
template <typename T = int>
struct elem_cyclic
{ T inf;                  // информационно поле
  elem_cyclic<T> *link;    // адресно поле
};
```

а цикличния линеен списък - чрез шаблона на класа *cyclicList*. Указателя към последния елемент на цикличния линеен списък ще означим с *start*, а указателя към текущ елемент - чрез *current*.

C113-03 · Копиране: code.html
```
template <typename T = int>
class cyclicList
{ public:
    // канонично представяне
```

Оригинал: стр. 114

C114-01 · Копиране: code.html
```
    cyclicList();
    ~cyclicList();
    cyclicList(const cyclicList&);
    cyclicList& operator=(const cyclicList&);
    // основни функции
    bool empty() const;
    void iterStart(elem_cyclic<T>* = NULL);
    elem_cyclic<T>* iter();
    void insert(const T&);
    void toEnd(const T&);
    void deleteElem(elem_cyclic<T>*, T&);
    void print() const;
    int length() const;
  private:
    elem_cyclic<T> *start,
                   *current;
    void copyList(const cyclicList&);
    void deleteList();
};
```

Член-функциите на каноничното представяне реализират идеи аналогични на тези при линейния списък, представен с една връзка. Конструкторът по подразбиране създава празен цикличен линеен списък, представен чрез една връзка.

C114-02 · Копиране: code.html
```
template <typename T>
cyclicList<T>::cyclicList()
{ start = NULL;
}

template <typename T>
cyclicList<T>::~cyclicList()
{ deleteList();
}

template <typename T>
cyclicList<T>::cyclicList(const cyclicList<T>& r)
{ copyList(r);
}
template <typename T>
cyclicList<T>& cyclicList<T>::operator=(const cyclicList<T>& r)
{ if (this != &r)
```

C115-01 · Копиране: code.html
```
    { deleteList();
      copyList(r);
    }
  return *this;
}
```

За реализирането на каноничното представяне се използват двете помощни член-функции, декларирани в *private* частта на шаблона на класа: *copyList*, която копира указания чрез формалния параметър цикличен линеен списък на друго място в паметта, сочено от указателя *this* и *deleteList*, която изтрива неявния цикличен линеен списък.

Копиране на списък

За реализиране на член-функцията *copyList* ще използваме член-функцията *toEnd*, която включва даден елемент като последен елемент на цикличния линеен списък.

C115-02 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::copyList(const cyclicList& r)
{ start = NULL;
  elem_cyclic<T> *p = r.start;
  if (p)
    do
      { p = p->link;
        toEnd(p->inf);
      } while (p != r.start);
}
```

Изтриване на списък

За реализиране на *deleteList* ще използваме член-функциите за работа с итератора.

C115-03 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::deleteList()
{ iterStart();
  elem_cyclic<T> *p = iter();
  while (p)
    { delete p;
      p = iter();
    }
  start = NULL;
}
```

Проверка дали списък е празен

C116-01 · Копиране: code.html
```
template <typename T>
bool cyclicList<T>::empty() const
{ return start == NULL;
}
```

Член-функции за работа с итератора

За работа с итератора *current* са предвидени две член-функции - *iterStart* и *iter*. Първата инициализира итератора, а втората връща текущата му стойност и го премества в следващата позиция.

Установяване на итератора в началото на цикличен списък

В случай че линейният цикличен списък не е празен, указателят *start* сочи последния елемент на цикличния списък. Елементът, намиращ се след указания от *start*, е началото на списъка.

Член-функцията *iterStart* установява *current* в указан като формален параметър адрес *p*, в *NULL* - ако списъкът е празен и в началото на списъка в противен случай.

C116-02 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::iterStart(elem_cyclic<T>* p)
{ if (p) current = p;
  else
    if (!start) current = NULL;
    else current = start->link;
}
```

Преместване на итератора в следваща позиция

Член-функцията *iter* връща текущата позиция на *current*, след което го премества в следващата позиция. При достигане до *start*, *current* става *NULL*.

C116-03 · Копиране: code.html
```
template <typename T>
elem_cyclic<T>* cyclicList<T>::iter()
{ if (!current) return NULL;
  elem_cyclic<T> *p = current;
  if (current == start) current = NULL;
  else current = current->link;
  return p;
}
```

Оригинал: стр. 117

Включване на елемент

Ще реализираме две член-функции за включване на елемент в цикличен списък. Член-функцията *insert* включва указан като формален параметър елемент след последния (като първи) елемент на списъка, а член-функцията *toEnd* включва указания елемент като последен елемент на списъка.

C117-01 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::insert(const T& x)
{ elem_cyclic<T>* p = new elem_cyclic<T>;
  assert(p != NULL);
  p->inf = x;
  if (start) p->link = start->link;
  else start = p;
  start->link = p;
}
template <typename T>
void cyclicList<T>::toEnd(const T& x)
{ insert(x);
  start = start->link;
}
```

Изключване на елемент

Член-функцията *deleteElem* изключва указания от указателя *p* елемент и го запомня в параметъра *x*. Предполагаме, че неявният цикличен списък не е празен и че *p* сочи негов елемент.

C117-02 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::deleteElem(elem_cyclic<T>* p, T& x)
{ x = p->inf;
  if (start != start->link)
    { elem_cyclic<T> *q = start;
      while (q->link != p) q = q->link;
      q->link = p->link;
      if (p == start) start = q;
    }
  else start = NULL;
  delete p;
}
```

Оригинал: стр. 118

Извеждане на цикличен списък

При реализирането на тази член-функция предполагаме, че за елементите от тип *T* е дефиниран операторът <<.

C118-01 · Копиране: code.html
```
template <typename T>
void cyclicList<T>::print() const
{ elem_cyclic<T> *p = start;
  if(p)
    do
    { p = p->link;
      cout << p->inf << " ";
    } while (p != start);
  cout << endl;
}
```

Намиране на дължината на цикличен списък

C118-02 · Копиране: code.html
```
template <typename T>
int cyclicList<T>::length() const
{ int n = 0;
  elem_cyclic<T> *p = start;
  if(p)
    do
    { p = p->link;
      n++;
    } while (p != start);
  return n;
}
```

Записваме създадения шаблон във файла *cyclicList.cpp* и ще го приложим в някои от следващите задачи.

Задача 18. Дадени са естествените числа *n* и *m*. Предполага се, че *m* човека са наредени в кръг и всеки от тях е получил пореден номер от 1 до *m* (бройки в посока обратна на часовниковата стрелка). Започвайки от първия елемент в посока обратна на часовниковата стрелка, се отброява *n*-тият човек и се отстранява от кръга. Отново започвайки от следващия човек (*n+1-вия*), се отброява отново *n*-тият човек и се отстранява. Този процес продължава до отстраняване на всички хора от кръга. Да се напише програма, която извежда номерата на отстранените хора в реда на отстраняването им.

Оригинал: стр. 119

C119-01 · Копиране: code.html

```
// Program Zad18.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "cyclicList.cpp"

typedef cyclicList<int> intCycl;

// създаване на цикличен списък,
// съдържащ целите числа 1, 2, ..., m,
// подредени последователно
void create(int m, intCycl& L)
{ for (int i = 1; i <= m; i++)
    L.toEnd(i);
}

// докато има елементи в списъка L отброява n-тия
// елемент и го изключва от цикличния списък
void josiff(int n, intCycl L)
{ L.iterStart();
  elem_cyclic<int> *p = L.iter(), *q;
  while (p != p->link)
    { q = p;
      for (int i = 1; i <= n-1; i++)
        q = q->link;
      p = q->link;
      int x; L.deleteElem(q, x);
      cout << x << " ";
    }
  cout << p->inf << endl;
}

int main()
{ intCycl L;
  int m, n;
  cout << "m: "; cin >> m;
  create(m, L);
  L.print();
  cout << "n: "; cin >> n;
  josiff(n, L);
```

Оригинал: стр. 120

C120-01 · Копиране: code.html

```
  return 0;
}
```

2.3.5.2. Функции от по-висок ред за работа с циклични линейни списъци

Задача 19. Да се дефинира шаблон на функцията от по-висок ред *accumulate* за циклични линейни списъци.

Обхождането на списъка ще реализираме чрез итератора и член-функциите за работа с него.

C120-02 · Копиране: code.html
```
template <typename T>
T accumulate(T (*op)(T, T), T null_val, cyclicList<T>& L)
{ T s = null_val;
  L.iterStart();
  elem_cyclic<T> *p = L.iter();
  while (p)
    { s = (*op)(s, p->inf);    // или s = op(s, p->inf);
      p = L.iter(); // използването на p = p->link ще
                    // предизвика зацикляне
    }
  return s;
}
```

Задача 20. Да се дефинира шаблон на функцията от по-висок ред *map* за циклични линейни списъци.

C120-03 · Копиране: code.html
```
template <typename T>
cyclicList<T> map(T (*f)(T), cyclicList<T>& L)
{ L.iterStart();
  elem_cyclic<T>* p = L.iter();
  cyclicList<T> L1;
  while (p)
    { L1.toEnd((*f)(p->inf));    // или L1.toEnd(f(p->inf));
      p = L.iter();
    }
  return L1;
}
```

Оригинал: стр. 121

Задача 21. Да се дефинира шаблон на функцията от по-висок ред *filter* за циклични линейни списъци.

C121-01 · Копиране: code.html
```
template <typename T>
cyclicList<T> filter(bool (*pred)(T), cyclicList<T>& L)
{ L.iterStart();
  elem_cyclic<T> *p = L.iter();
  cyclicList<T> L1;
  while (p)
    { if ((*pred)(p->inf)) L1.toEnd(p->inf);
      p = L.iter();
    }
```

C121-01 · продължение · Копиране: code.html

```
  return L1;
}
```

Задача 22. Да се напише програма, която създава цикличен линеен списък от циклични линейни списъци от числа. Програмата да конкатенира цикличните линейни списъци, съставящи дадения и да изведе получения цикличен линеен списък.

Програма *Zad22.cpp* използва функцията от по-висок ред *accumulate*.

C121-02 · Копиране: code.html

```
// Program Zad22.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "cyclicList.cpp"

template <typename T>
T accumulate(T (*op)(T, T), T null_val, cyclicList<T>& L)
{ T s = null_val;
  L.iterStart();
  elem_cyclic<T> *p = L.iter();
  while (p)
    { s = op(s, p->inf);
      p = L.iter();
    }
  return s;
}

typedef cyclicList<int> intList;
```

Оригинал: стр. 122

C122-01 · Копиране: code.html

```
typedef cyclicList<intList> intListList;

// конкатениране на циклични списъци
intList concat(intList L1, intList L2)
{ intList L;
  L1.iterStart();
  L2.iterStart();
  elem_cyclic<int> *p = L1.iter(),
                   *q = L2.iter();
  while (p)
    { L.toEnd(p->inf);
      p = L1.iter();
    }
  while (q)
    { L.toEnd(q->inf);
      q = L2.iter();
    }
  return L;
}
// създаване на цикличен списък L от n
// циклични списъка от цели числа
void create(int n, intListList& L)
{ for (int i = 1; i <= n; i++)
    { intList m;
      cout << "k= "; int k; cin >> k;
      for (int j = 1; j <= k; j++)
        { cout << "x: "; int x; cin >> x;
          m.toEnd(x);
        }
      L.toEnd(m);
    }
}

int main()
{ int n; cout << "n= "; cin >> n; // брой на подсписъците
  intListList L;
  // създаване на цикличен списък L от n циклични списъка
  create(n, L);
  intList L2;                     // нулева стойност на concat
  // конкатенация на цикличните списъци от цели
```

Оригинал: стр. 123

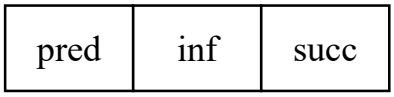
C123-01 · Копиране: code.html

```
  // числа, съставящи цикличния списък L
  intList L1 = accumulate(concat, L2, L);
  L1.print();
  return 0;
}
```

2.3.6. Реализация на линеен списък, представен с две връзки

Шаблонът на клас *DLList* реализира това представяне на линеен списък. Чрез *start* и *end* са означени указателите към началото и края на списъка. Използват се два итератора: *currentS*, преместващ се от началото към края и *currentE*, преместващ се от края към началото на линейния списък (фиг. 2.8).

Тройната кутия



ще реализираме чрез шаблона на структурата *elem_link2* с параметър *T*

C123-02 · Копиране: code.html
```
template <typename T = int>
struct elem_link2
{ T inf;
  elem_link2<T> *pred,
                *succ;
};
```

а линейния списък - чрез шаблона на класа *DLList*.

C123-03 · Копиране: code.html
```
template <typename T = int>
class DLList
{ public:
    // канонично представяне
    DLList();
    ~DLList();
    DLList(const DLList&);
    DLList& operator=(const DLList&);
    // основни функции
    bool empty() const;
    void iterStart(elem_link2<T>* = NULL);
    void iterEnd(elem_link2<T>* = NULL);
    elem_link2<T>* iterSucc();
    elem_link2<T>* iterPred();
```

Оригинал: стр. 124

C124-01 · Копиране: code.html
```
    void toEnd(const T&);
    void toStart(const T&);
    void deleteElem(elem_link2<T>*, T&);
    void print() const;
    void print_reverse() const;
    int length() const;
  private:
    elem_link2<T> *start,     // указател към началото
                  *end,     // указател към края
                  *currentS,// итератор към началото
                  *currentE;// итератор към края
    // помощни член-функции
    void copyList(const DLList&);
    void deleteList();
};
```

Член-функциите на каноничното представяне няма да коментираме. Реализират идеи, аналогични на тези при свързаното представяне с една връзка и при цикличното представяне с една връзка.

C124-02 · Копиране: code.html
```
template <typename T>
DLList<T>::DLList()
{ start = end = NULL;
}

template <typename T>
DLList<T>::~DLList()
{ deleteList();
}

template <typename T>
DLList<T>::DLList(const DLList<T>& r)
{ copyList(r);
}

template <typename T>
DLList<T>& DLList<T>::operator=(const DLList<T>& r)
{ if (this != &r)
    { deleteList();
      copyList(r);
```

Оригинал: стр. 125

C125-01 · Копиране: code.html
```
    }
  return *this;
}
```

Следващите две функции са помощни и реализират копиране на линеен списък, представен чрез две връзки на друго място в паметта и изтриване на неявния линеен списък, представен чрез две връзки. Използват се за реализиране на каноничното представяне.

Копиране на списък

C125-02 · Копиране: code.html

```
template <typename T>
void DLList<T>::copyList(const DLList<T>& r)
{ start = end = NULL;
  elem_link2<T>* p = r.start;
  while (p)
    { toEnd(p->inf);
      p = p->succ;
    }
}
```

Изтриване на списък

C125-03 · Копиране: code.html

```
template <typename T>
void DLList<T>::deleteList()
{ elem_link2<T>* p = start;
  while (p)
  { start = start->succ;
    delete p;
    p = start;
  }
  end = NULL;
}
```

Проверка дали списък е празен

C125-04 · Копиране: code.html

```
template <typename T>
bool DLList<T>::empty() const
{ return start == NULL;
}
```

Член-функции за работа с итераторите

Следващите четири член-функции реализират операциите с итераторите *currentS* и *currentE*.

Оригинал: стр. 126

Установяване на итератора currentS в началото на списъка

C126-01 · Копиране: code.html
```
template <typename T>
void DLList<T>::iterStart(elem_link2<T>* p)
{ if (p) currentS = p;
  else currentS = start;
}
```

Установяване на итератора currentE в края на списъка

C126-02 · Копиране: code.html
```
template <typename T>
void DLList<T>::iterEnd(elem_link2<T>* p)
{ if (p) currentE = p;
  else currentE = end;
}
```

Преместване на итератора currentS в следващата позиция

C126-03 · Копиране: code.html
```
template <typename T>
elem_link2<T>* DLList<T>::iterSucc()
{ elem_link2<T>* p = currentS;
  if (currentS) currentS = currentS->succ;
  return p;
}
```

Преместване на итератора currentE в предходната позиция

C126-04 · Копиране: code.html
```
template <typename T>
elem_link2<T>* DLList<T>::iterPred()
{ elem_link2<T>* p = currentE;
  if (currentE) currentE = currentE->pred;
  return p;
}
```

Включване на елемент

Включването на елемент в края и в началото на линеен списък, представен с две връзки, ще реализираме чрез член-функциите *toEnd* и *toStart*. Първата включва елемент пред първия елемент, а втората - след последния елемент на неявния линеен списък.

C126-05 · Копиране: code.html
```
template <typename T>
void DLList<T>::toEnd(const T& x)
{ elem_link2<T>* p = end;
```

C126-05 · продължение · Копиране: code.html
```
  end = new elem_link2<T>;
  assert(end != NULL);
  end->inf = x;
```

Оригинал: стр. 127

C127-01 · Копиране: code.html
```
  end->succ = NULL;
  if (p) p->succ = end;
  else start = end;
  end->pred = p;
}

template <typename T>
void DLList<T>::toStart(const T& x)
{ elem_link2<T>* p = start;
  start = new elem_link2<T>;
  assert(start != NULL);
  start->inf = x;
  start->pred = NULL;
  if (p) p->pred = start;
  else end = start;
  start->succ = p;
}
```

Изключване на указан елемент

Изключването на указан елемент в линеен списък ще реализираме чрез член-функцията *deleteElem*. Предполагаме, че неявният линеен списък, представен с две връзки, не е празен и че указателят *p* сочи негов елемент. Изключеният елемент се записва в параметъра *x*.

C127-02 · Копиране: code.html
```
template <typename T>
void DLList<T>::deleteElem(elem_link2<T>* p, T& x)
{ x = p->inf;
  if (start == end)
    start = end = NULL;
  else
    if (p == start)
      { start = start->succ;
        start->pred = NULL;
      }
    else
      if (p == end)
        { end = p->pred;
          end->succ = NULL;
        }
      else
```

C128-01 · Копиране: code.html
```
        { p->pred->succ = p->succ;
          p->succ->pred = p->pred;
        }
  delete p;
}
```

Извеждане на елементите

Извеждането на елементите на линеен списък се реализира чрез последователното им обхождане без разрушаване на списъка. При обхождане, започващо от *start*, списъкът се извежда от началото му, а при обхождане, започващо от *end*, извеждането е в обратен ред.

- извеждане от началото към края

C128-02 · Копиране: code.html
```
template <typename T>
void DLList<T>::print() const
{ elem_link2<T>* p = start;
  while (p)
    { cout << p->inf << " ";
      p = p->succ;
    }
  cout << endl;
}
```

- извеждане от края към началото

C128-03 · Копиране: code.html
```
template <typename T>
void DLList<T>::print_reverse() const
{ elem_link2<T>* p = end;
  while (p)
    { cout << p->inf << " ";
      p = p->pred;
    }
  cout << endl;
}
```

И при двете член-функции предполагаме, че за типа *T* е предефиниран операторът <<.

Дължина на линеен списък

Намирането дължината на линеен списък, представен с две връзки, може да се осъществи по същия начин, като при линейни списъци, представени с една връзка.

C129-01 · Копиране: code.html
```
template <typename T>
int DLList<T>::length() const
{ int n = 0;
  elem_link2<T>* p = start;
  while (p)
    { n++;
      p = p->succ;
    }
  return n;
}
```

Шаблонът на класа *DLList* записваме във файла *DLList.cpp*.

Задачи върху линеен списък, представен чрез две връзки

Задача 23. Линеен списък, съдържащ *2n* цели числа $a_1, a_2,..., a_{2n-1}, a_{2n}$, е представен чрез две връзки (*n* е дадено естествено число). Да се дефинира функция, която намира:

а) $S = a_1.a_{2n} + a_2.a_{2n-1} + a_n.a_{n+1}$
б) $M = \max \{ \min \{ a_1, a_{2n} \}, \min \{ a_2, a_{2n-1} \}, ..., \min \{ a_n\ a_{n+1} \} \}$
в) $M = \max \{ \min \{ a_1, a_2, ..., a_n \}, \min \{ a_{n+1}, a_{n+2}, a_{2n} \} \}$.

C129-02 · Копиране: code.html
```
// Program Zad23.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "DLList.cpp"

typedef DLList<int> intList;

// създаване на линеен списък с две връзки
// списъкът от цели числа L трябва да е празен
// при започване изпълнението на процедурата
void createList(int n, intList& L)
{ for (int i = 1; i <= 2*n; i++)
    { cout << "> ";
      int x; cin >> x;
      L.toEnd(x);
    }
}
```

Оригинал: стр. 130

C130-01 · Копиране: code.html

```
// a)
int sum(intList& L)
{ L.iterStart();
  elem_link2<int>* p = L.iterSucc();
  L.iterEnd();
  elem_link2<int>* q = L.iterPred();
  int n = L.length()/2;
  int s = 0;
  for (int i = 1; i <= n; i++)
    { s = s + p->inf * q->inf;
      p = L.iterSucc();
      q = L.iterPred();
    }
  return s;
}
// 6)
int maximin(intList& L)
{ int max;
  L.iterStart();
  elem_link2<int>* p = L.iterSucc();
  L.iterEnd();
  elem_link2<int>* q = L.iterPred();
  int n = L.length()/2;
  // инициализиране на променливата max с
  // минимума на първия и последния елементи
  if (p->inf <= q->inf) max = p->inf;
  else max = q->inf;
  for (int i = 1; i <= n-1; i++)
    { p = L.iterSucc();
      q = L.iterPred();
      int min;
      if (p->inf <= q->inf) min = p->inf;
      else min = q->inf;
      if (min > max) max = min;
    }
  return max;
}

int main()
{ intList L;
```

Оригинал: стр. 131

C131-01 · Копиране: code.html
```
  cout << "n= "; int n; cin >> n;
  createList(n, L);
  L.print();
  cout << "sum: " << sum(L) << endl;
  cout << "maximin: " << maximin(L) << endl;
  return 0;
}
```

Задача 24. Даден е линеен списък, съдържащ $2n$ цели числа $a_1, a_2, \ldots, a_{2n-1}, a_{2n}$, представен чрез две връзки така, че първите му n елемента са сортирани във възходящ, а вторите n елемента - в низходящ ред. Да се дефинира функция, която сортира във възходящ ред елементите на списъка.

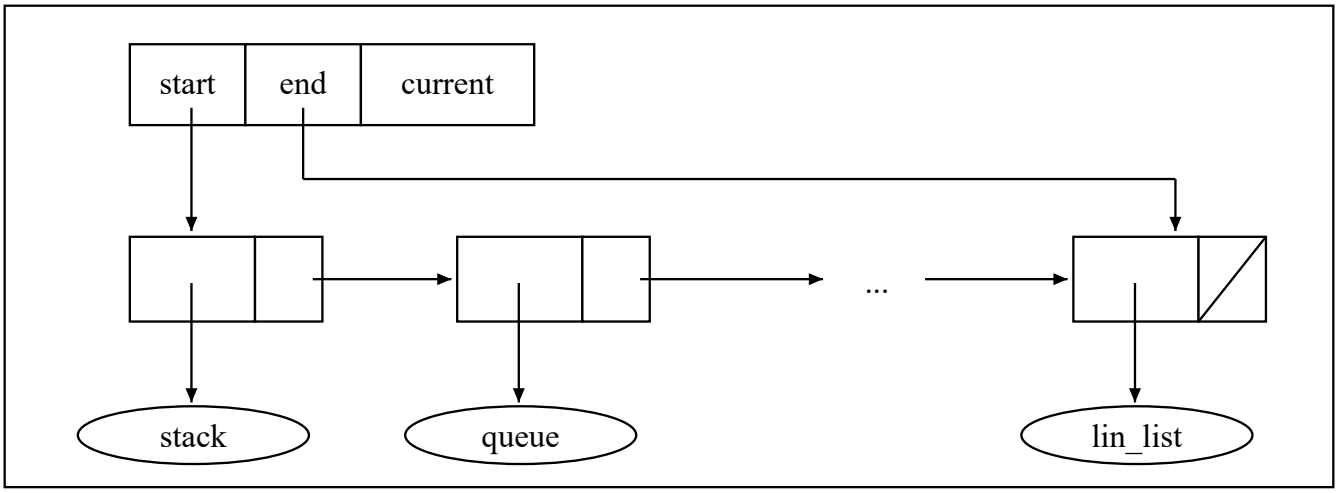
Функцията *sort* решава задачата. Тя реализира следната идея. Установяват се указателите p и q съответно в началото и в края на линейния списък. Докато p и q са различни, по-малкият от сочените от p и q елементи се включва в нов линеен списък, след което се прескача.

C131-02 · Копиране: code.html
```
typedef DLList<int> intList;
intList sort(intList& L)
{ L.iterStart();
  elem_link2<int>* p = L.iterSucc();
  L.iterEnd();
  elem_link2<int>* q = L.iterPred();
  intList L1;
  while (p != q)
    if (p->inf <= q->inf)
      { L1.toEnd(p->inf);
        p = L.iterSucc();
      }
    else
      { L1.toEnd(q->inf);
        q = L.iterPred();
      }
  L1.toEnd(p->inf);
  return L1;
}
```

Оригинал: стр. 132

2.4. НЕХОМОГЕННИ (ХЕТЕРОГЕННИ) ЛИНЕЙНИ СТРУКТУРИ ОТ ДАННИ

Линейна структура от данни с компоненти от различни типове се нарича *линейна нехомогенна* или *линейна хетерогенна структура от данни*. Нарича се също *линейна полиморфна структура от данни* тъй като най-често операциите над нея се реализират като полиморфни действия. Например линеен списък, представен с една връзка с компоненти линейни списъци, стекове, опашки, декове и други структури от данни (линейни и/или нелинейни) е хетерогенен линеен списък (фиг. 2.9).



Фигура 2.9. Хетерогенен линеен списък от стекове, опашки и линейни списъци

Полиморфизмът, с помощта на механизма на абстрактните класове, дава възможност за създаване на линейни хетерогенни структури от данни. От примерната хетерогенна структура на фигура 2.9. се вижда, че елементите на хетерогенния линеен списък са указатели към обекти на класове, реализиращи стекове, опашки и линейни списъци.

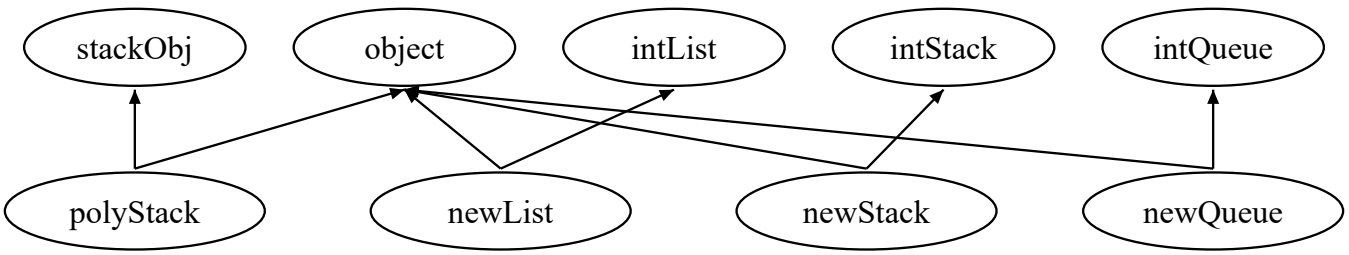
Линейният списък, елементите на който са от различни типове, се нарича *носител* на линейната хетерогенна структура. Носител на линейна хетерогенна структура може да е която и да е линейна структура от данни (масив, стек, опашка, списък, дек и др.).

В следващата задача ще реализираме хетерогенен стек, компонентите на който са линейни списъци от цели числа, представени с една връзка, стекове от цели числа и опашки от цели числа. Наготово ще използваме шаблоните на класовете: *LList*, реализиращ свързан списък, представен с една връзка и с елементи от тип *T*; *stack*, реализиращ стек с елементи от тип *T* и *queue*, реализиращ опашка с елементи от тип *T*.

Оригинал: стр. 133

Задача 25. Да се напише програма, която създава хетерогенен стек, елементите на който могат да съхраняват обекти от класовете: *intList* (линеен списък от цели числа), *intStack* (стек от цели числа) и *intQueue* (опашка от цели числа), както и хетерогенни стекове, съдържащи линейни списъци, стекове и опашки от цели числа. Програмата да извежда контейнерите на хетерогенния стек, а също и броя на елементите, както на хетерогенния стек, така и на всеки негов контейнер.

Програма *Zad25.cpp* реализира две полиморфни действия: извеждане и намиране на броя на компонентите на всеки контейнер на хетерогенния стек. Тя осигурява, при сканирането на елементите на полиморфния стек, контейнерите му да не се разрушават. Реализира следната йерархия от класове



C133-01 · Копиране: code.html

```
// Program Zad25.cpp
#include <iostream>
#include <cassert>
using namespace std;

#include "LList1.cpp"
#include "stack_link.cpp"
#include "queue_link.cpp"

class object
{ public:
    // полиморфни действия
    virtual void print() = 0;
    virtual int length() = 0;
};

typedef stack<object*> stackObj;
typedef LList<int> intList;
typedef stack<int> intStack;
typedef queue<int> intQueue;
```

Оригинал: стр. 134

C134-01 · Копиране: code.html

```
class polyStack: public stackObj, public object
{ public:
    void print();
    int length();
};

void polyStack::print()
{ polyStack ps(*this);    // копира хетерогенния стек в
                          // стека ps
  object *x;
  while (!ps.empty())
    { ps.pop(x);
      x->print();         // разрешава се динамично
    }
  cout << endl;
}

int polyStack::length()
{ polyStack ps(*this);    // копира хетерогенния стек в
                          // стека ps
  int num = 0;
  object *x;
  while (!ps.empty())
    { ps.pop(x);
      cout << x->length() << " "; // разрешава се динамично
      num++;
    }
  cout << endl;
  return num;
}

class newList : public intList, public object
{ public:
    void print();
    int length();
};

void newList::print()
{ intList::print();
```

Оригинал: стр. 135

C135-01 · Копиране: code.html

```
}

int newList::length()
{ return intList::length();
}

class newStack : public intStack, public object
{ public:
    void print();
    int length();
};

void newStack::print()
{ intStack temp(*this);   // копира текущия стек
                          // в стека temp
  temp.print();
}

int newStack::length()
{ intStack temp(*this);   // копира текущия стек
                          // в стека temp
  return temp.length();
}

class newQue: public intQueue, public object
{ public:
    void print();
    int length();
};

void newQue::print()
{ intQueue temp(*this);   // копира текущата опашка
                          // в опашката temp
  temp.print();
}

int newQue::length()
{ intQueue temp(*this);   // копира текущата опашка
                          // в опашката temp
  return temp.length();
}
```

Оригинал: стр. 136

C136-01 · Копиране: code.html

```
int main()
{ // създаване на празен хетерогенен стек lh1
  polyStack lh1;
  // създаване на контейнерите на lh1:
  // първи контейнер - линеен списък от цели числа ln1
  newList ln1;
  int i;
  for (i = 15; i <= 65; i = i+10)
    ln1.toEnd(i);
  // включване на първия контейнер в хетерогенния стек lh1
  lh1.push(&ln1);
  // втори контейнер - стек от цели числа ls1
  newStack ls1;
  for (i = 10; i <= 40; i = i+10)
    ls1.push(i);
  // включване на втория контейнер в хетерогенния стек lh1
  lh1.push(&ls1);
  // трети контейнер - опашка от цели числа lq1
  newQue lq1;
  for (i = 17; i <= 87; i = i+10)
    lq1.push(i);
  // включване на третия контейнер в хетерогенния стек lh1
  lh1.push(&lq1);
  // извеждане на контейнерите на хетерогенния стек lh1
  lh1.print();
  // намиране и извеждане на броя на елементите на
  // хетерогенния стек lh1 и на неговите контейнери
  cout << lh1.length() << "\n\n";
  // създаване на нов празен хетерогенен стек lh2
  polyStack lh2;
  // създаване на контейнерите на lh2:
  // първи контейнер - опашка от цели числа lq2
  newQue lq2;
  for (i = 11; i <= 51; i = i+10)
    lq2.push(i);
  // включване на първия контейнер в хетерогенния стек lh2
  lh2.push(&lq2);
  // втори контейнер - стек от цели числа ls2
  newStack ls2;
  for (i = 18; i <= 58; i = i+10)
```

Оригинал: стр. 137

C137-01 · Копиране: code.html

```
    ls2.push(i);
  // включване на втория контейнер в хетерогенния стек lh2
  lh2.push(&ls2);
  // извеждане на контейнерите на хетерогенния стек lh2
  lh2.print();
  // намиране и извеждане на броя на елементите на
  // хетерогенния стек lh2 и на неговите контейнери
  cout << lh2.length() << "\n\n";
  // създаване на нов празен хетерогенен стек lh
  polyStack lh;
  // създаване на контейнерите на lh:
  // първи контейнер - хетерогенният стек lh1
  lh.push(&lh1);
  // втори контейнер - хетерогенният стек lh2
  lh.push(&lh2);
  // трети контейнер - стекът от цели числа ls1
  lh.push(&ls1);
  // четвърти контейнер - опашката от цели числа lq2
  lh.push(&lq2);
  // извеждане на контейнерите на хетерогенния стек lh
  lh.print();
  // намиране и извеждане на броя на елементите на
  // хетерогенния стек lh и на неговите контейнери
  cout << lh.length() << endl;
  return 0;
}
```

Оригинал: стр. 138

ЗАДАЧИ

1. Дадени са декларациите:

C138-01 · Копиране: code.html

```
typedef char string20[21];
struct studentType
{ struct20 couseName;
  char courseGrade;
  studentType* next;
};
```

Нарисувайте диаграмата на паметта след изпълнението на следните програмни фрагменти:

а)

C138-02 · Копиране: code.html
```
studentType* temp = new studentType;
assert(temp != NULL);
temp->next = new studentType;
assert(temp->next != NULL);
strcpy(temp->couseName, "101 Math-1");
temp->courseGrade = 'B';
strcpy(temp->next->couseName, "103 CS-1");
temp->next->courseGrade = 'A';
temp->next->next = NULL;
```

б)

C138-03 · Копиране: code.html
```
studentType* temp = new studentType;
assert(temp != NULL);
temp->next = new studentType;
assert(temp->next != NULL);
temp->next->next = new studentType;
assert(temp->next->next != NULL);
delete temp->next;
```

в)

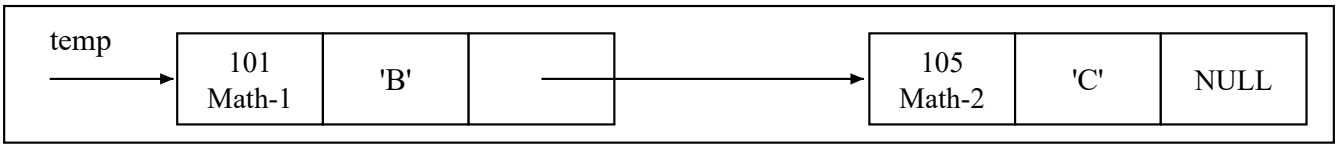
C138-04 · Копиране: code.html
```
studentType* temp1 = new studentType;
assert(temp1 != NULL);
strcpy(temp1->couseName, "305 Eng");
temp1->courseGrade = 'F';
studentType* temp2 = new studentType;
assert(temp2 != NULL);
strcpy(temp2->couseName, "105 Math-2");
temp2->courseGrade = 'C';
temp2->next = NULL;
studentType* temp = new studentType;
assert(temp != NULL);
```

Оригинал: стр. 139

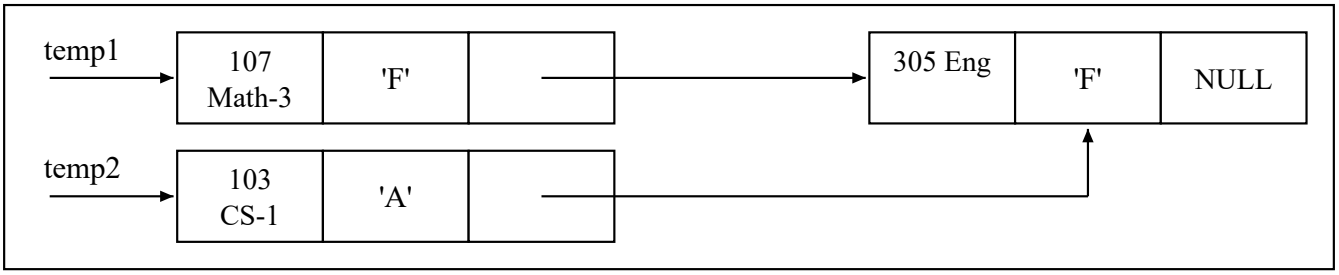
C139-01 · Копиране: code.html
```
temp1->next = temp;
strcpy(temp1->next->couseName, "107 Math-3");
temp1->next->courseGrade = 'F';
temp = temp2;
temp2 = temp1->next;
temp2->next = NULL;
```

2. За всяка от диаграмите по-долу, напишете програмен фрагмент, който да я реализира:

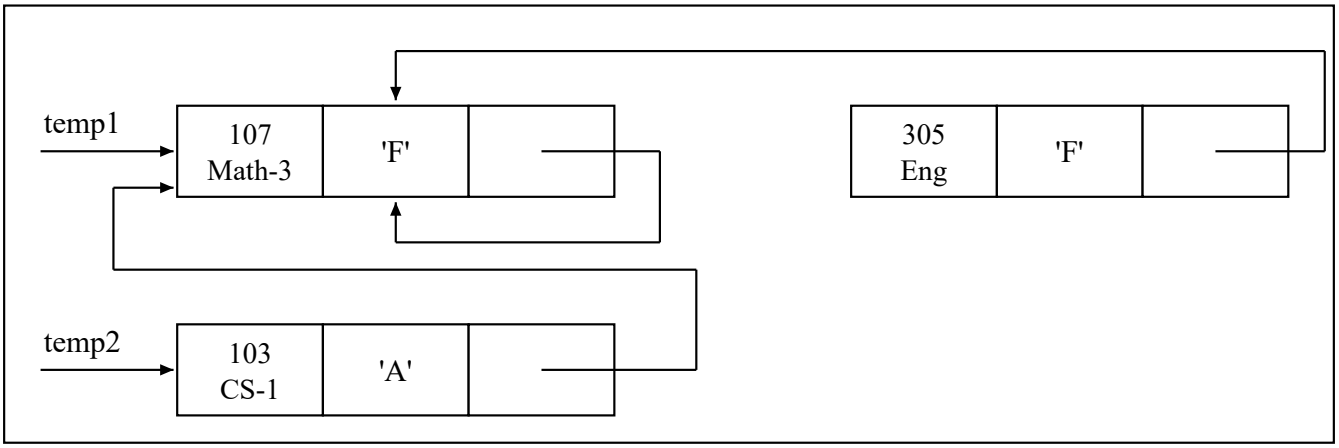
а)



б)



в)



3. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на стек.

4. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа (думи) са наредени във възходящ (лексикографски) ред.

Оригинал: стр. 140

5. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елементите на стек от числа са различни.

6. Даден е стек от цели числа. Да се дефинира булева функция, която установява дали елементите на стека могат да се пренаредят така, че разликата на всеки два съседни елемента да е наредена по намаляване. Ако за дадения стек това е възможно, да се пренаредят елементите на стека, след което се изведат.

7. Да се дефинира АТД, който представя два стека, реализирани последователно, използвайки само един едномерен масив. Препълване да се получава само когато всички елементи на масива са заети.

8. Опишете алгоритъм, чрез който може да се реализират три стека в един едномерен масив.

9. Дадени са два стека от числа, елементите на които са сортирани в низходящ ред. Да се дефинира функция, която слива двата стека. Намерете времето за изпълнение на дефинираната

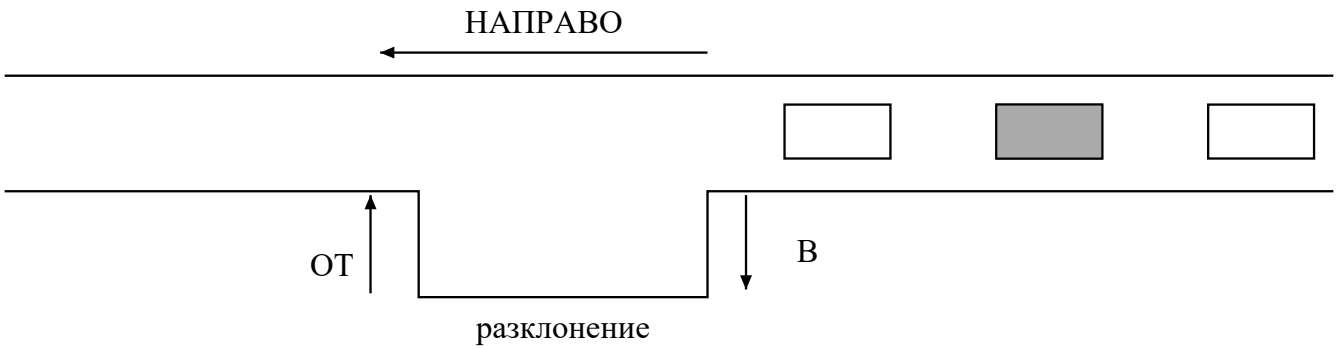
функция.

10. В стек е записан без грешка израз от вида:

<израз> ::= <цифра>|
s(<израз>)|
p(<израз>)

където *s(x)* и *p(x)* намират *x+1* и *x-1*, съответно (*s(9)* се приема за 0, а *p(0)* – за 9). Като се използва стек да се намери стойността на израза.

11. Железопътен сортировъчен възел е построен по следния начин:



В дясната страна са събрани *n* черни и *n* бели вагона. В разклонението могат да се поместят всички вагони. Като се използват трите операции В, ОТ и НАПРАВО, да се съберат вагоните в лявата страна така, че да се редуват вагоните от двата вида. За целта да се използват не повече от *3n-1* операции В, ОТ и НАПРАВО.

12. Железопътен сортировъчен възел е построен по начина от предната задача. В дясната част са разположени *n* различни вагона, номерирани с 1, 2, ..., *n*. Да се напише програма, която като използва операциите В, ОТ и НАПРАВО, намира всички възможни начини за подреждане на вагоните.

Оригинал: стр. 141

13. Да се напише програма, която въвежда програма на C++ и проверява дали на всеки символ '(', '{' и '[' в програмата съответства символ ')', '}' и ']', съответно. За целта да се използва стек.

14. Граматиката *G = (Vn, Vt, S, P)*, където *Vn* е множеството от нетерминалните символи, *Vt* е множеството от терминалните символи ($Vn \cap Vt = \varnothing$), $S \in Vn$ е началният символ, *P* е крайно множество от правила е дефинирана по следния начин:

Vn = {S},
Vt = {a, b, c, d, e, f},
P = {S -> aSa, S -> bSb, S -> cSc, S -> dSd, S -> eSe, S -> f}.

Тази граматика генерира низове от вида {f, afa, bfb, cfc, dfd, efe, abfba, acfca, adfda, aefea, aafaa, bbfbb, ccfcc, ddfdd, bdcfcdb, …}. Да се дефинира функция, която определя дали даден символен низ принадлежи на езика, генериран от граматиката *G*, или не принадлежи. За целта да се използва стек.

15. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елемент принадлежи на опашка.

16. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа (думи) са наредени във възходящ (лексикографски) ред.

17. Да се дефинира нерекурсивна (рекурсивна) функция, която проверява дали елементите на опашка от числа са различни.

18. Дадени са две опашки от цели числа *p* и *q*. Да се дефинира функция, която намира опашка от цели числа, съдържаща елементите, които:

а) принадлежат на поне една от опашките *p* или *q*;
б) принадлежат едновременно и на *p*, и на *q*;
в) принадлежат на *p*, но не принадлежат на *q*;
г) принадлежат на една от опашките, но не принадлежат на другата.

19. Да се дефинира функцията от по-висок ред *accumulate* за стек (опашка).

20. Да се дефинира функцията от по-висок ред *map* за стек (опашка).

21. Да се дефинира функцията от по-висок ред *filter* за стек (опашка).

22. Да се дефинира функция, която сортира във възходящ ред елементите на опашка от числа.

23. Символен низ съдържа правилен текст от вида:

<текст> ::= <интервал> | <елемент><текст>
<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се дефинира функция, която за всяка двойка съответстващи си отваряща и затваряща скоба извежда позициите им в текста, подредени по нарастване на позицията на отварящите скоби. (Например за текста A+(TP-F(X)*(B-C)) резултатът е 3 17; 8 10; 12 16.)

Оригинал: стр. 142

24. Символен низ съдържа правилно записан текст от вида:

<текст> ::= <интервал> | <елемент><текст>
<елемент> ::= <буква> | (<текст>).

Като се използват опашка и/или стек, да се дефинира функция, която за всяка двойка съответстващи си отваряща и затваряща скоба извежда позициите им в текста, подредени по нарастване на позицията на затварящите скоби. (Например за текста A+(TP-F(X)*(B-C)) резултатът е 8 10; 12 16; 3 17).

25. В опашка е записан без грешка израз от вида:

<израз> ::= <цифра> | f(<израз>) | g(<израз>) |
h(<израз>, <израз>) |
p(<израз>, <израз>)

където *f(x) = x+1* и *g(x) = x-1*, като *f(9) = 0*, а *g(0) = 9*, *h(x, y) = max{ f(x), g(y) }*, *p(x, y) = min{ g(x), f(y) }*. Като се използва стек да се намери стойността на израза.

26. *Опашка от „познати“* е разновидност на структурата от данни опашка. Когато елемент се включва в опашка от „познати“, се сканират елементите на опашката с цел откриване на „познати“. Ако елементът намери „познати“, се включва след тях. Ако не намери „познати“, се включва накрая на опашката. Изключването на елемент се осъществява по общоприетия начин –

от началото на опашката. Да се напише програма, която по дадени групи от елементи, определящи „познати“, симулира опашка от „познати“.

27. *Приоритетната опашка* е разновидност на структурата от данни опашка. Всеки нейн елемент съдържа и цяло число, определящо приоритета на елемента. Нов елемент се включва в приоритетната опашка не в края й, а след последния елемент с приоритет, по-голям от неговия. Равните елементи са с еднакви приоритети и наредбата им в опашката е по реда на включването им. Да се дефинира АТД, реализиращ приоритетна опашка.

28. *Декът* (*DEQue*, съкращение от англ. език *Double-Ended-Queue* – опашка с два края) е структура, при която елементи могат да се включват и изключват в двата й края. Да се дефинира АТД, реализиращ дек.

29. Дек, операцията включване на който не е допустима в началото му, се нарича *дек с ограничен вход*. Да се дефинира АТД, реализиращ дек с ограничен вход.

30. Дек, операцията изключване на който не е допустима в края му, се нарича дек с ограничен изход. Да се дефинира АТД, реализиращ дек с ограничен изход.

31. Даден е линеен списък, представен с една връзка от четен брой символи. Да се дефинира булева функция, която определя дали елементите от първата половина съвпадат с елементите от втората половина на списъка. Елементите на списъка да се обхождат от началото към края.

Оригинал: стр. 143

32. Да се дефинира функция, която изтрива от линейния списък L, представен с една връзка:

а) първия отрицателен елемент, ако такъв съществува;
б) всички отрицателни елементи;
в) всички повтарящи се елементи.

Да се намери времето за изпълнение на дефинираните функции.

33. Да се напише нерекурсивен и рекурсивен вариант на функция, която намира позицията на даден елемент в линеен списък, представен с една връзка. Изследвайте колко голям може да бъде списъкът, за да има оперативна памет за представянето му при двата варианта на решение.

34. Нека са дадени линеен списък L_1 от реални числа и линеен списък L_2 от цели числа. Елементите на L_2 са сортирани във възходящ ред. Да се дефинира функция *printElem(L_1, L_2)*, която извежда на екрана елементите на L_1, които се намират в позициите, указани в списъка L_2. Да се намери времето за изпълнение на дефинираната функция.

35. Нека са дадени линеен списък L_1 от реални числа и линеен списък L_2 от цели числа. Елементите на L_2 са сортирани във възходящ ред. Да се дефинира функция *delElem(L_1, L_2)*, която изтрива елементите на L_1, които се намират в позициите, указани в списъка L_2. Да се намери времето за изпълнение на дефинираната функция.

36. Да се дефинира нерекурсивна функция за обръщане на елементите на списък с n елемента, представен с една връзка, за време $O(n)$.

37. Да се дефинира шаблон на функция, която сортира във възходящ ред по метода на мехурчето (т. 4.1.2, глава 4) елементите на линеен списък от тип T, представен с една връзка.

38. Да се реализира линеен списък от рационални числа, представен с една връзка, и да се извършат следните действия:

– да се намери минималното рационално число на списъка;
– да се сортира във възходящ ред по метода на пряката селекция (или мехурчето) списъкът;
– да се слеят два сортирани линейни списъка от рационални числа.

За целта да се дефинира АТД *rat*, реализиращ рационално число.

39. Да се реализира линеен списък от комплексни числа, представен с една връзка и да се извършат следните действия:

– да се провери дали всички елементи на списъка са по-малки по модул от *R (R > 0)*;
– да се сортира във възходящ ред по метода на пряката селекция (или мехурчето) списъкът;
– да се слеят два сортирани линейни списъка от комплексни числа.

За целта да се дефинира АТД *complex*, реализиращ рационално число.

Оригинал: стр. 144

40. Да се създаде линеен списък, представен с една връзка, елементите на който са стекове от числа. Да се промени свързаният списък, като се филтрират елементите на стековете, така че от всеки стек се отстранят отрицателните елементи. За получения списък да се намери сумата от елементите на всеки от стековете. За целта да се използват функции от по-висок ред.

41. Да се създаде линеен списък представен с една връзка, елементите на който са опашки от естествени числа. Да се промени линейният списък като се филтрират елементите на опашките, така че от всяка опашка се отстранят елементите, които са числа на Фибоначи. За получения списък да се намери произведението от *k*-тите степени на елементите на всяка от опашките (*k* е дадено естествено число). За целта да се използват функции от по-висок ред.

42. Полиномът $a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$ е представен чрез линеен списък (т. 2.3.3.4.). Дефинирайте функция, която реализира деление на два полинома. Намерете времето за изпълнение на дефинираната функция.

43. Полином на три променливи (например *x, y* и *z*) е представен чрез линеен списък с една връзка. Елементите на списъка са от тип *coef4*:

C144-01 · Копиране: code.html
```
struct coef4
{ double coef;    // коефициент
  int powX;       // степенен показател на променливата x
  int powY;       // степенен показател на променливата y
  int powZ;       // степенен показател на променливата z
};
```

Степенните показатели на полинома са подредени лексикографски в низходящ ред. Пример за такъв полином и подредбата на едночлените му е $P(x, y, z) = 1.5x^4y^3z - 3.7x^4z^2 + 8.2x^2y^5z^6 + 13.4y^5z^6 - 4.1y^3z^6 + 6.9z^7$. За степенните показатели на едночлените му са в сила: (4, 3, 1) > (4, 0, 2) > (2, 5, 6) > (0, 5, 6) > (0, 3, 6) > (0, 0, 7). Да се дефинира функция, която намира сумата на два полинома, представени по описания начин.

Забележка. Наредбата $(a, b, c) > (p, q, r)$ е в сила, ако $a > p$, или $a = p$ и $b > q$, или $a = p$ и $b = q$ и $c > r$.

Да се намери времето за изпълнение на дефинираната функция.

44. Полином на три променливи (например *x, y* и *z*) е представен, както е показано в предходната задача. Да се дефинира функция, която намира произведението на два полинома от този вид. Да се намери времето за изпълнение на дефинираната функция.

45. Полином на *n* променливи (например $x_1, x_2, ..., x_n$) е представен чрез линеен списък с една връзка. Степенните показатели на полинома са подредени лексикографски в низходящ ред. Да се дефинира функция, която намира сумата на два полинома, представени по описания начин.

Оригинал: стр. 145

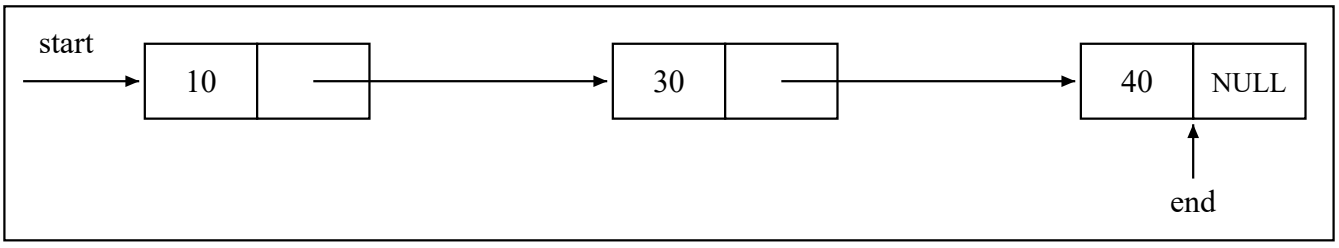
46. Полином на *n* променливи (например $x_1, x_2, ..., x_n$) е представен чрез линеен списък с една връзка. Степенните показатели на полинома са подредени лексикографски в низходящ ред. Да се дефинира функция, която намира произведението на два полинома, представени по описания начин.

47. Да се напише програма, която създава линеен списък, елементите, на който са линейни списъци от данни за студенти (име и среден успех). Програмата да сортира всеки от линейните списъци по успеха на студентите и да извежда получения линейния списък от линейни списъци.

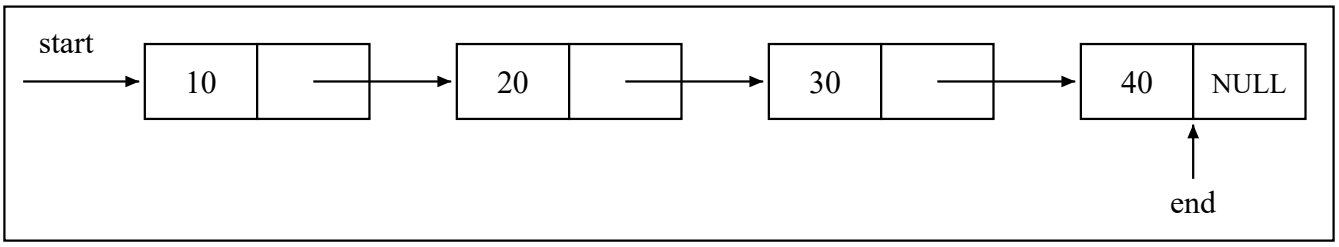
48. Да се напише програма, която създава линеен списък, съдържащ информация за студентите от една група. Програмата да може да:

а) добавя данни за нов студент;
б) изтрива данни за студент;
в) търси данни за студент;
г) сортира по някаква данна за студент елементите на списъка.

49. (нареден линеен списък) Нека е дефинирана релация на наредба ≤ за елементите от тип *T*. *Линеен списък от тип T е нареден*, ако $a \leq b$ е в сила за всяка двойка елементи (*a, b*) на списъка, за които *a* предшества *b*. Да се дефинира АТД *ordList*, който реализира нареден линеен списък, представен свързано с една връзка. Основните операции над списъка трябва да запазват наредеността му. Например в резултат от вмъкването на елемента 20 в наредения линеен списък от цели числа, показан по-долу



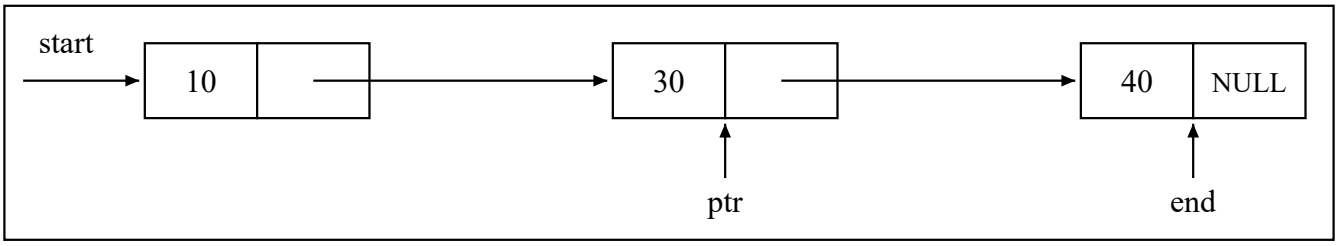
трябва да се получи списъкът



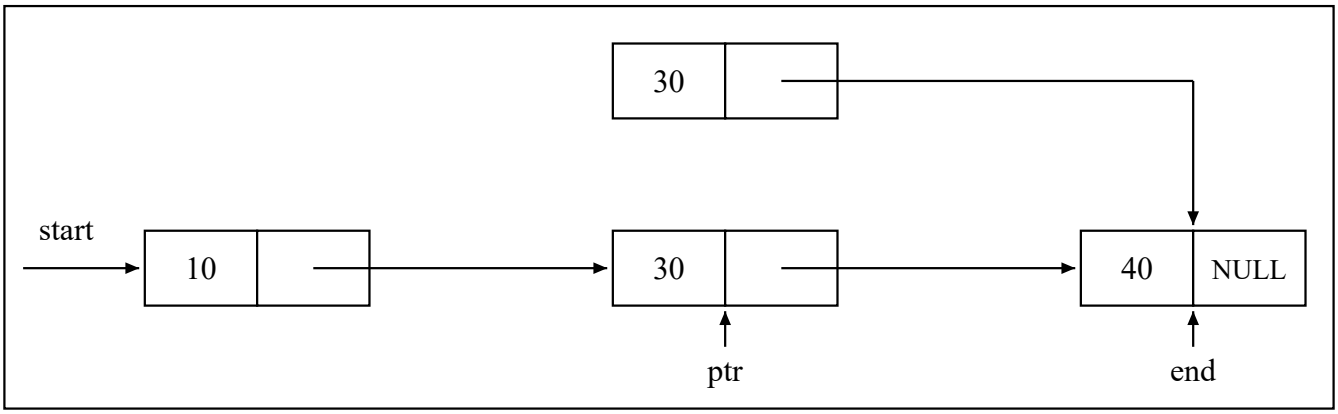
Реализирането на вмъкването може да се осъществи по следния начин:

– обхожда се първоначалният списък до намирането на връх със стойност, не по-малка от 20 (по-долу е посочена чрез указателя *prt*), т.е.

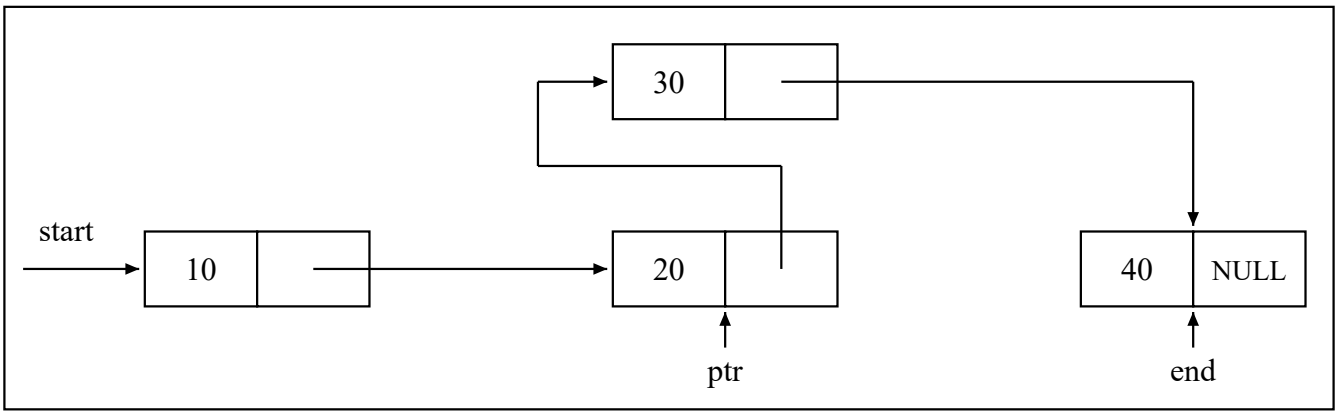
Оригинал: стр. 146



– прави се копие на **ptr*, т.е.



– променят се *ptr->inf* на 20 и *ptr->link* да сочи новата двойна кутия, т.е.



50. Да се дефинира шаблон на функция, която сортира във възходящ ред по метода на мехурчето (т. 4.1.2, глава 4) елементите на линеен списък от тип *T*, представен:

а) с две връзки;
б) циклично.

За елементите на типа *T* е дефиниран операторът за сравнение >.

51. Дадени са два линейни списъка от числа, елементите на които са сортирани във възходящ ред. Да се дефинира функция, която слива двата списъка. Задачата да се реши в случаите, когато списъкът е представен:

Оригинал: стр. 147

а) с две връзки;
б) циклично.

Да се намери времето за изпълнение на дефинираната функция.

52. Даден е линеен списък с *2n* елемента, представен с две връзки. Да се дефинира функция, която проверява дали елементите на списъка са симетрични относно средата му. Да се намери времето за изпълнение на дефинираната функция.

53. Линеен списък от *2n* цели числа е представен чрез две връзки. Да се дефинира булева функция, която проверява дали първите *n* елемента на списъка са сортирани във възходящ ред, следващите *n* елемента на списъка са сортирани в низходящ ред и списъкът е симетричен (списъкът 1 3 5 5 3 1 удовлетворява исканото свойство). Да се дефинира функция, която включва елемент в симетричен списък от горния вид, като запазва монотонността и симетричността (например в резултат от включването на елемента 4 в списъка 1 3 5 5 3 1 трябва да се получи списъкът: 1 3 4 5 5 4 3 1).

54. Да се дефинира функция, която конкатенира два линейни списъка, представени:

а) с две връзки;
б) циклично.

Да се намери времето за изпълнение на дефинираната функция.

55. Да се дефинира функция, която проверява дали два линейни списъка, представени:

а) с две връзки;
б) циклично

са равни. Да се намери времето за изпълнение на дефинираната функция.

56. Да се напише програма, която създава цикличен линеен списък от циклични линейни списъци от символи. Да се конструира цикличен линеен списък от думите, получени след конкатенирането на символите на всеки от цикличните линейни списъци. Да се конструира изречение, получено след конкатенирането на всяка от думите на цикличния линеен списък. Да се намери времето за изпълнение на дефинираната функция.

57. Намерете времето за изпълнение на програмата от задача 18 (глава 2).

58. Нека са дадени линеен списък L_1 от реални числа и линеен списък L_2 от цели числа. И двата списъка са представени циклично с една връзка. Елементите на L_2 са сортирани във възходящ ред. Да се дефинира функция *printElem(L_1, L_2)*, която извежда на екрана елементите на L_1, които се намират в позициите, указани в списъка L_2. Да се намери времето за изпълнение на дефинираната функция.

Оригинал: стр. 148

59. Нека са дадени линеен списък L_1 от реални числа и линеен списък L_2 от цели числа. Елементите на L_2 са сортирани във възходящ ред. И двата списъка са представени циклично с една връзка. Да се дефинира функция *delElem(L_1, L_2)*, която изтрива елементите на L_1, които се намират в позициите, указани в списъка L_2. Да се намери времето за изпълнение на дефинираната функция.

60. Даден е линеен списък от цифри, представен циклично с една връзка. Да се провери дали дадено естествено число може да се прочете в списъка след последователно преминаване през елементите му. Ако това е така, да се намери позицията на началото му.

61. Да се дефинира АТД, който реализира линеен списък, представен циклично с две връзки.

62. Да се дефинира функция, която проверява дали линеен списък, представен циклично с две връзки е палиндром (симетричен относно средата си). Да се намери времето за изпълнение на дефинираната функция.

63. Да се напише програма, която:

а) създава хетерогенен линеен списък, представен с една връзка, елементите на който са стекове, опашки и линейни списъци от цели числа;
б) намира поредния номер на контейнера и позицията в контейнера на първото срещане на дадено цяло число.

64. Да се напише програма, която:

а) създава хетерогенен стек, елементите на който са стекове, опашки и линейни списъци от цели числа;
б) сортира елементите на всеки от контейнерите на хетерогенния стек;
в) извежда получения хетерогенен стек.

65. Да се напише програма, която:

а) създава хетерогенен линеен списък, представен с една връзка, елементите на който са стекове, опашки и линейни списъци от реални числа;
б) изтрива от хетерогенния списък всички контейнери, които съдържат отрицателен елемент, ако такива съществуват;
в) извежда получения хетерогенен линеен списък.

66. Да се напише програма, която създава цикличен хетерогенен списък, елементите на който са линейни списъци и стекове от символи. Да се намерят думите, които се получават при конкатенирането на символите на всеки от контейнерите на цикличния хетерогенен списък. Да се намери изречението, получено след конкатенирането на всяка от думите на цикличния хетерогенен списък.

67. Да се напише програма, която създава хетерогенен линеен списък с две връзки, съдържащ $2n$ елемента (n е дадено естествено число), който са линейни списъци, представени с една, с две връзки и циклично, стекове и опашки от

Оригинал: стр. 149

символи. Да се намерят две изречения, първото от които се получава при конкатенирането на символите на всеки от първите n контейнера, а второто – при конкатениране на символите на вторите n контейнера. При първото изречение обхождането на хетерогенния линеен списък да е от началото към края, а при второто – от края към началото.

68. Да се създаде хетерогенен цикличен списък, елементите на който са стекове и опашки от цифри. Да се дефинира функция, която изключва първия контейнер, в който може да се прочете зададено цяло число след конкатениране на последователни цифри на контейнера.

ПРОЕКТИ

1. Стеков калкулатор

Да се реализира стеков калкулатор, който да може да изпълнява поне следния набор от математически операции, функции и команди:

– аритметичните операции +, -, *, /;
– стандартните функции *sin, cos, exp, log* (натурален логаритъм), *sqrt, asin, acos, tan, atan, ctan, actan*;
– командите

pop	изтрива елемента от върха на стек;
clear	изтрива стек;
=	извежда върха на стек;
show	извежда стек;
help	извежда подсещане за помощ
quit	завършва изпълнението на програмата.

2. Симулация на полетите в летище

Малко летище има само една писта. Във всяка единица от време само един самолет може да кацне или само един самолет може да излети. В даден момент от време пистата може да е или празна, или самолет да каца, или самолет да излита. Възможно е няколко самолета да чакат за кацане или за излитане. Да се напише програма, която симулира поведението на летището. За целта да се използват две опашки от самолети *landing* (за кацане) и *takeoff* (за излитане), в които самолетите да чакат. След получаване на заявка от самолет за кацане или за излитане, симулацията да обслужи самолета от главата на опашката

Оригинал: стр. 150

за кацане *landing* и само ако опашката *landing* е празна да позволи да излети самолет. Симулацията да протече за периода от време *endtime*. Програмата да направи и изведе някои статистики за процеса на симулация като:

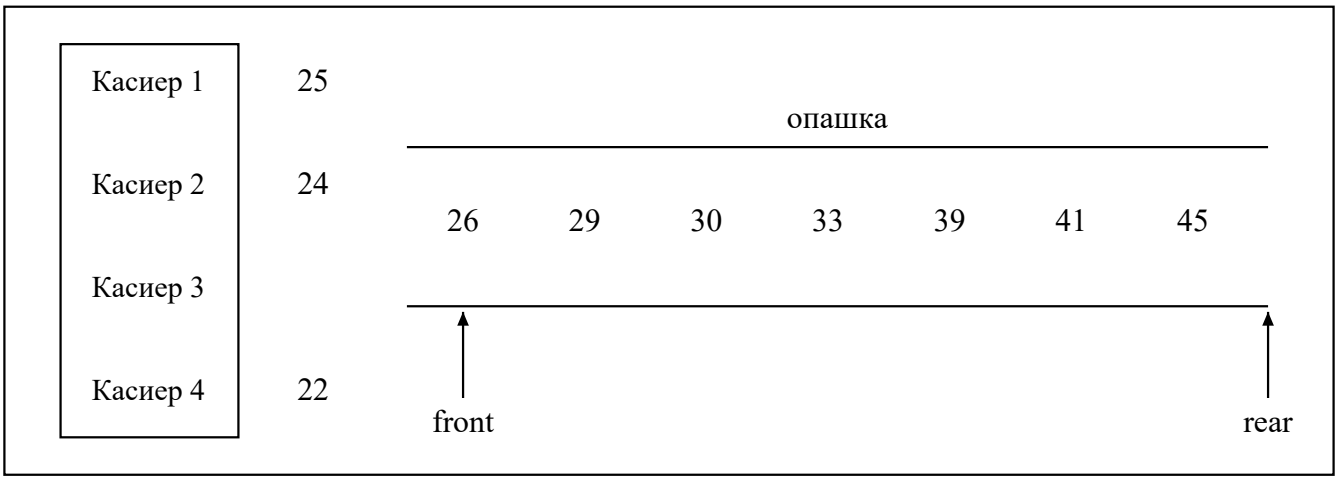
– брой на кацналите самолети;
– брой на излетелите самолети;
– брой на самолетите, на които е направен отказ за кацане или за излитане;
– брой на чакащите за кацане и за излитане самолети;
– процент на времето, когато пистата е свободна;
– средно време за чакане за кацане.

Генерирането на самолети да се осъществява по случаен начин. За целта да се използва функцията *rand*, в резултат от изпълнението на която се получава цяло число от 0 до *RAND_MAX* (*RAND_MAX* е константа, различна за отделните реализации).

3. Симулация на опашка в банка

Целта на тази симулация е да изгради представа у управителя на банка за ефективността на обслужването на случайно пристигащи в банката клиенти от определен брой касиери, както и за броя и времето за обслужване на клиенти от всеки касиер.

Когато клиент пристига в банката и всички касиери са заети с обслужване на други клиенти, новият клиент се нарежда на опашка. Когато лице се нареди на опашката, то получава уникален номер, съвпадащ с минутата на пристигането му в банката (от 1 до *MAX*). За простота се приема, че в дадена минута в банката може да пристигне не повече от един клиент. На фиг. 2.10 е показана графично опашка в банка със 7 клиента и 4 касиера.



Фигура 2.10. Симулация на опашка от клиенти в банка

Оригинал: стр. 151

В показаната на фигурата симулация единствено касиер 3 може да обслужи намиращия се в началото на опашката клиент (останалите касиери са заети и обслужват клиенти: касиер 1 обслужва клиента, пристигнал в 25-тата минута, касиер 2 обслужва клиента, пристигнал в 24-тата минута, а касиер 4 обслужва клиента, пристигнал в 22-рата минута).

Да се напише програма, която симулира обслужването на опашка от клиенти в банка за интервал от време с дължина 480 минути. Да се реализира лицата да постъпват в опашката в случайно време. Освен това времето за обслужване на клиент от касиер също да се определя по случаен начин (чрез генератор на случайни числа).

Програмата да направи и изведе някои статистики за процеса на симулация като:

– общ брой на обслужените клиенти;
– брой на обслужените клиенти от всеки касиер;
– средно количество време за чакане;
– средна дължина на опашката;
– максимална дължина на опашката;
– процент на времето, когато касите са свободни.

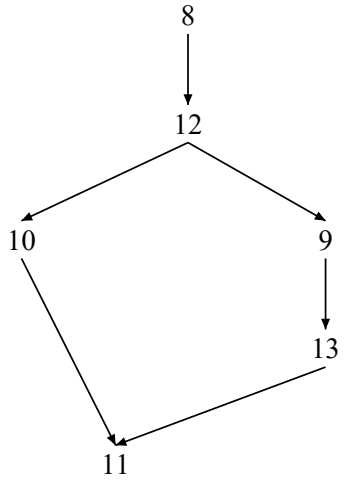
4. Топологично сортиране на частично наредени множества

Множеството от елементи *A* е частично наредено, ако над елементите му е определено отношение *предшества*, означавано с < и притежаващо следните свойства:

(а) транзитивност
Ако x, y, z ∈ A и x < y и y < z, то x < z;
(б) асиметричност
Ако x, y ∈ A и x < y, то не е в сила y < x;
(в) нерефлексивност
Ако x ∈ A, то не е в сила x < x.

Ако отношението $x < y$ означим със стрелка от x към y, връзката между елементите на частично наредено множество може да се изобрази чрез диаграма. Например ако A е частично наредено множество, съдържащо елементите {8, 12, 10, 9, 11, 13} и 8 < 12, 12 < 10, 12 < 9, 10 < 11, 9 < 13 и 13 < 11, връзките между елементите на A могат да се изобразят чрез следната диаграма

Оригинал: стр. 152



Задачата за топологичното сортиране се състои в намирането на алгоритъм за преобразуване на частично наредено множество в линейно. В случая това се изразява в подреждане на елементите на множеството в линейна последователност. Последното не е възможно ако в диаграмата има поне един цикъл. Съществуването на цикли обаче нарушава свойството частична нареденост. За елементите от примерното множество резултатът от топологичното сортиране е 8, 12, 10, 9, 13, 11. За реализиране на топологично сортиране на частично наредено множество може да се използва следният алгоритъм:

C152-01 · Копиране: code.html
```
while (множеството A не е празно)
  { 1. Избира се елемент на A, който не се предшества
    от друг и се премества в края на редицата,
    натрупваща резултата от топологичното сортиране.
    2. Изключва се избраният елемент от множеството A.
  }
```

Да се дефинира програма, която реализира топологично сортиране на частично наредено множество. За целта да се използват линейни списъци.

5. Топологично сортиране на функции в програма

Да се напише програма, която въвежда програма на C++, разпознава всички дефиниции на функции (процедури) и обръщенията към тях и прави опит за установяване на топологична наредба на функциите. За две функции *A* и *B* приемаме, че е в сила $A < B$ винаги когато *A* се извиква от *B*.

6. Мултисписък

Списък, на който всеки елемент може едновременно да участва в повече от един линеен списък, се нарича *мултисписък*.

В университет се обучават *n* студенти ($100 \le n \le 25000$) и се преподават *m* избираеми учебни дисциплини ($20 \le m \le 1000$). Да се напише програма, която дава възможност да се правят справки за:

– студентите, които посещават дадена учебна дисциплина;
– учебните дисциплини, записани от даден студент.

За представяне на данните да се реализира мултисписък от вида, представен на фиг. 2.11.

Оригинал: стр. 153

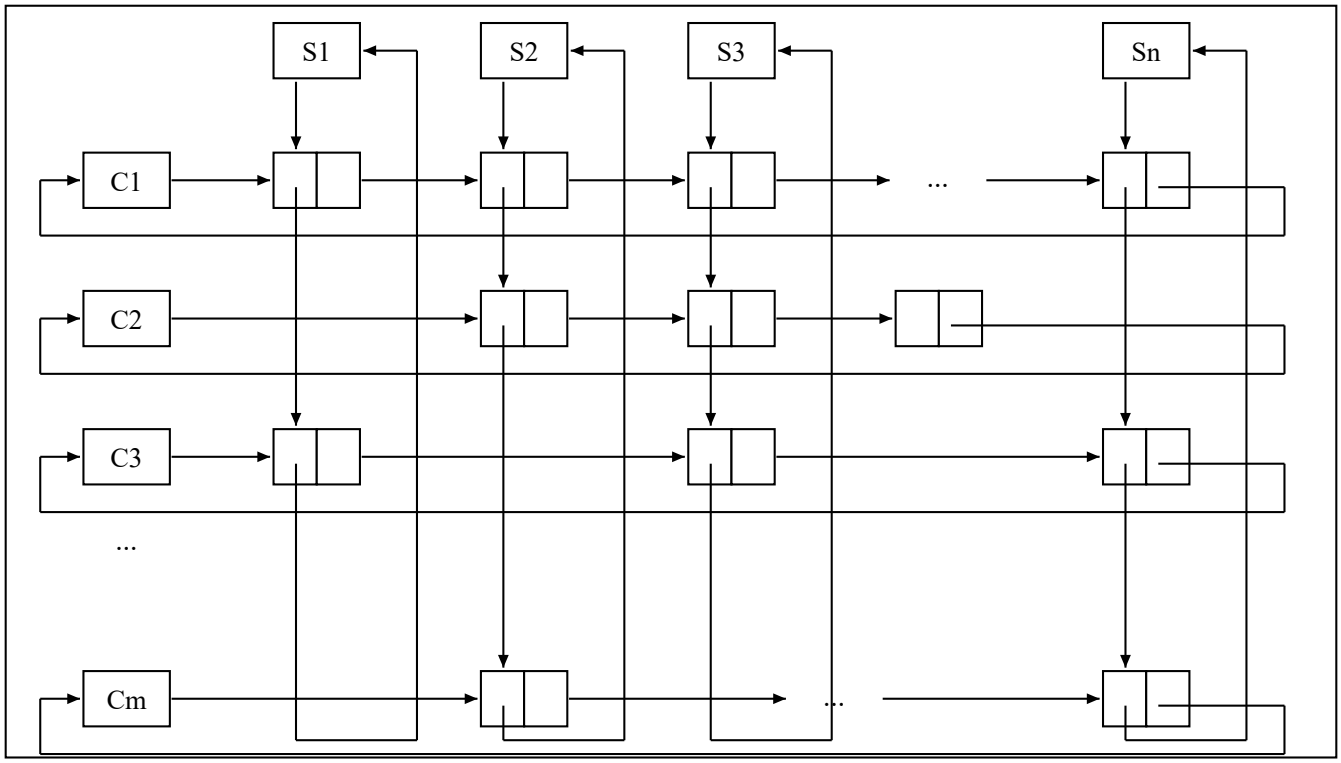
Мултисписъкът е изграден от линейни списъци, представени циклично с една връзка и съдържащи:

а) за всеки студент ($Si,\ 1 \le i \le n$) – информация за посещаваните от него избираеми дисциплини;
б) за всяка учебна дисциплина ($Cj,\ 1 \le j \le m$) – студентите, които са я записала.

Първият елемент на всеки от списъците означава име на студент или име на учебна дисциплина. Останалите елементи на списъците са адреси, както е показано на фиг. 2.11. При това представяне, ако трябва да се намерят студентите, които са записали учебна дисциплина *C2*, се обхожда списъкът *C2*. Първият студент, записал *C2* принадлежи на списъка *S2*. За да се намери името му, се обхожда цикличният списък *S2*, започвайки от пресечния елемент на списъците *S2* и *C2* до достигане до първия елемент на списъка *S2*. Следващият елемент на *C2* принадлежи на списъка *S3*. Отново, за да се намери името на студента, се обхожда списъкът *S3* от пресечния елемент на списъците *S3* и *C2* до достигане до първия елемент на списъка *S3*. Тези действия продължават до изчерпване на списъка *C2*.

По аналогичен начин се реализира търсенето курсовете, които е записал студент.

Ще отбележим, че този начин за представяне на информацията за студентите и избираемите дисциплини е икономичен по отношение на използваната памет, но това е за сметка на времето за търсене. Най-лошият случай е, когато студент е записал всички избираеми дисциплини. Тогава ще бъдат обхождани всички списъци за курсове, за да се направи справката за учебните дисциплини, които посещава този студент.



Фигура 2.11. Мултисписък, представящ курсовете, които студентите посещават

Оригинал: стр. 154

7. Редактор на линии

Да се напише програма, която създава редактор на линии *LINEDIT*. Като основни структури от данни да се използват текстов файл и линеен списък, представен свързано с една връзка. *LINEDIT* да може да обработва линии, дълги до 80 знака чрез следните команди:

IN	INsert (следвана от текст, организиран като линии и завършващ с "//")
DE	DElete
MO	MOve (следвано от цяло число)
LA	List All
EX	EXit

Редакторът трябва да поддържа указател към текуща линия (цяло число). Командата *IN* въвежда нови входни линии във файла точно пред текущата линия. Командата *DE* изтрива текущата линия. Линията след изтритата става текуща. Командата *MO*, следвана от цялото число *n*, премества указателя към текущата линия на *n* линии след текущата, ако *n* е положително цяло число и на $-n$ линии пред текущата, ако *n* е отрицателно цяло число. След преместването на указателя се извежда текущата линия. Командите *MO* и *DE* могат да причинят указателят към текущ ред да е в края на текстовия файл. Тази ситуация е разрешена и позволява вмъкване след последната линия. Също когато програмата започва работата си не съществуват линии за редактиране. Тогава указателят към текущата линия трябва да е позициониран след несъществуваща крайна линия. Командата *LA* извежда всички линии от текущата до края на

текста и текущата линия не се променя. Командата *EX* завършва работата на програмата.

Програмата да може да обработва следните видове грешки:

– Отрицателна стойност на цялото число в командата *MO*, водеща до репозициониране на указателя към текуща линия преди първата линия, да установява първата линия за текуща.

– Положителна стойност на цялото число в командата *MO*, водеща до репозициониране на указателя към текуща линия след последната линия или изтриване чрез *DE* на последната линия, да причинят указателят към текуща линия да е след последната линия.

– Когато указателят към текуща линия е след последната линия, командата *DE* да не извършва нищо.

– Невалидните команди (командите, различни от *IN, DE, MO, LA* и *EX*) да се игнорират.

Оригинал: стр. 155

Пример за използване на редактора LINEDIT. В него командите са удебелени.

>LA
<<END>>
>IN
aaaa aaaa aaaa aaaa aaaa aaaa
bbbbbbbb bbbbbbbb bbbbbbbb
//
>LA
<<END>>
>MO -2
aaaa aaaa aaaa aaaa aaaa aaaa
>LA
aaaa aaaa aaaa aaaa aaaa aaaa
bbbbbbbb bbbbbbbb bbbbbbbb
<<END>>
>MO 1
bbbbbbbb bbbbbbbb bbbbbbbb
>LA
bbbbbbbb bbbbbbbb bbbbbbbb
<<END>>
>IN
cccc cccc cccc cccc cccc cccc
ddddd ddddd ddddd ddddd ddddd
//
>LA
bbbbbbbb bbbbbbbb bbbbbbbb
<<END>>
>MO -10
aaaa aaaa aaaa aaaa aaaa aaaa
>LA
aaaa aaaa aaaa aaaa aaaa aaaa
cccc cccc cccc cccc cccc cccc

ddddd ddddd ddddd ddddd ddddd
bbbbbbbb bbbbbbbb bbbbbbbb
<<END>>
>LA
aaaa aaaa aaaa aaaa aaaa aaaa
cccc cccc cccc cccc cccc cccc
ddddd ddddd ddddd ddddd ddddd
bbbbbbbb bbbbbbbb bbbbbbbb
<<END>>
>MO 10
<<END>>
>IN
eeeeeeeeeeeeeeeeeeeeeeeeee
//

>MO -20
aaaa aaaa aaaa aaaa aaaa aaaa
>LA
aaaa aaaa aaaa aaaa aaaa aaaa
cccc cccc cccc cccc cccc cccc
ddddd ddddd ddddd ddddd ddddd
bbbbbbbb bbbbbbbb bbbbbbbb
eeeeeeeeeeeeeeeeeeeeeeeeee
<<END>>
>DE
>LA
cccc cccc cccc cccc cccc cccc
ddddd ddddd ddddd ddddd ddddd
bbbbbbbb bbbbbbbb bbbbbbbb
eeeeeeeeeeeeeeeeeeeeeeeeee
<<END>>
>MO 1
ddddd ddddd ddddd ddddd ddddd
>DE
>LA
bbbbbbbb bbbbbbbb bbbbbbbb
eeeeeeeeeeeeeeeeeeeeeeeeee
<<END>>
>MO -10
cccc cccc cccc cccc cccc cccc
>LA
cccc cccc cccc cccc cccc cccc
bbbbbbbb bbbbbbbb bbbbbbbb
eeeeeeeeeeeeeeeeeeeeeeeeee
<<END>>
>EX

Оригинал: стр. 157

Глава 3

НЕЛИНЕЙНИ СТРУКТУРИ ОТ ДАННИ ДЪРВО И ГРАФ

За редица приложения се налага да се използват обекти, които са с йерархична или с мрежова структура. За създаването на такива обекти се използват структурите от данни дърво и граф.

3.1. ДЪРВО

Използването на линейни списъци с голям брой елементи е неефективно при задачи, при които операцията за достъп до елемент се изпълнява по-често от операциите, които модифицират списъка. Причината е, че операцията за достъп е от порядъка на *n*, а операциите за включване и изключване на елемент при динамичните реализации са от порядъка на константа. Структурата от данни дърво преодолява този недостатък, тъй като операциите достъп, включване и изключване на елемент за дърво имат логаритмична сложност.

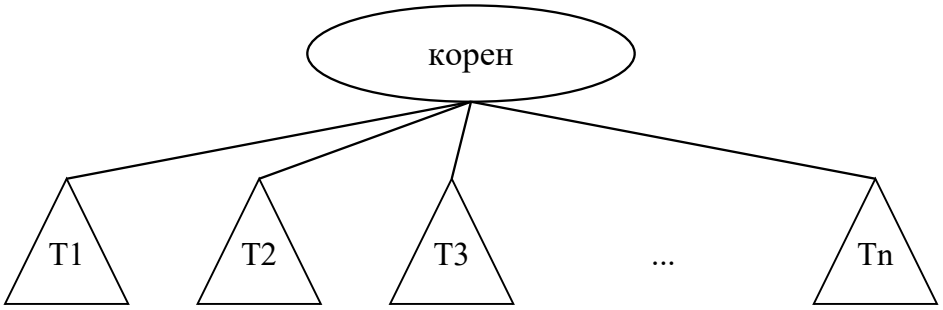
3.1.1. Дефиниране на дърво

Дърво от тип T е структура от данни, която е или празна, или е образувана от:

– данна от тип *T*, наречена *корен (връх, възел, ключ)* на двоичното дърво от тип *T* и

– крайно, възможно празно множество от променлив брой несвързани помежду си елементи T_1, T_2, T_3, ..., T_n, които са дървета от тип *T*, наречени *поддървета на дървото от тип T*.

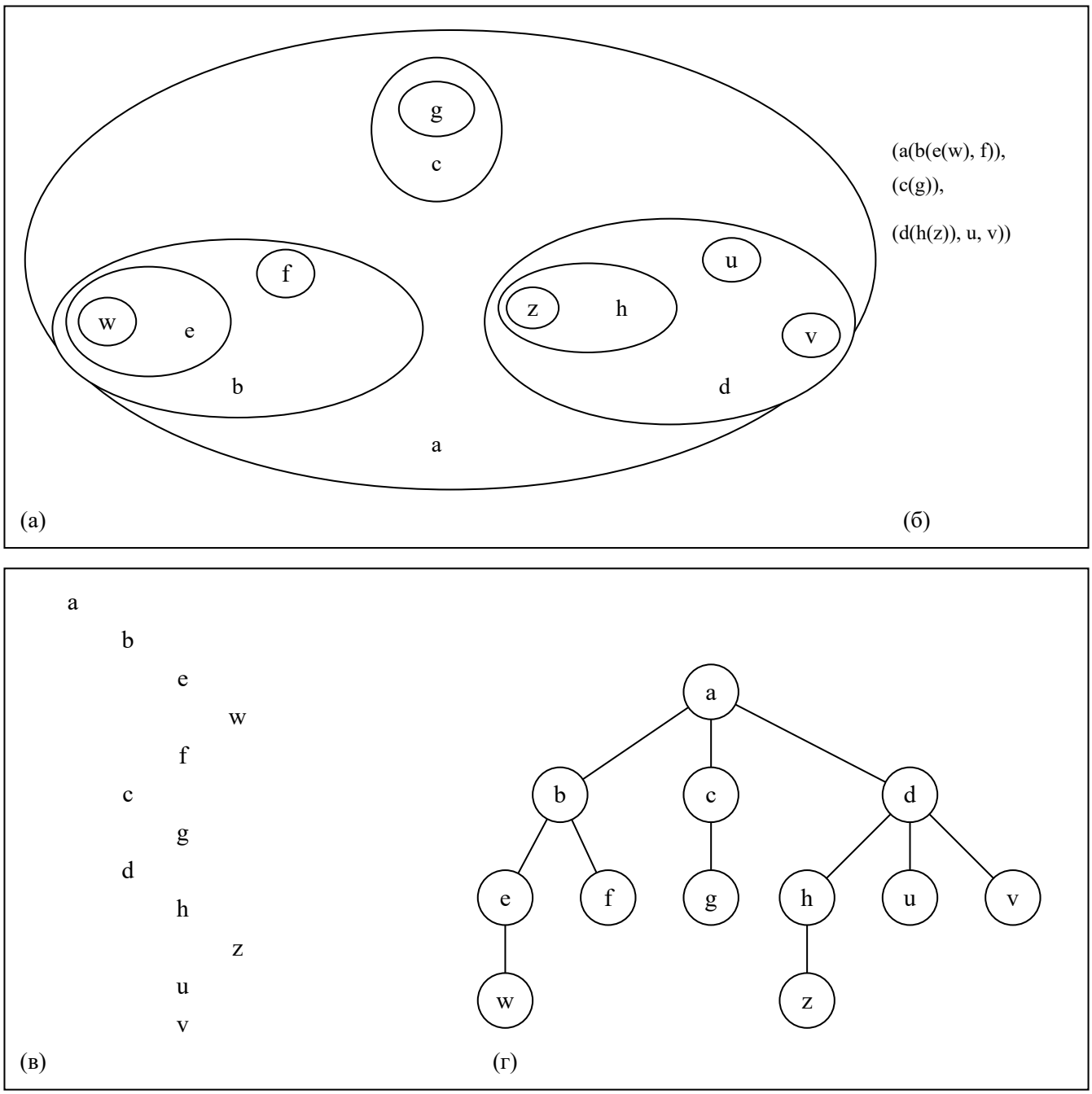
Най-често непразното дърво се изобразява графично като граф, както е показано на фиг. 3.1 и фиг. 3.2 (г). Други представяния са:
– чрез вложени множества, фиг. 3.2 (а);
– чрез вложени скоби, фиг. 3.2 (б);
– чрез стъпаловидно отместване, фиг. 3.2 (в).



Фигура 3.1. Дърво от тип T

Оригинал: стр. 158

Нека *a, b, c, …, x, y, z* са елементи от тип *T*. На фиг. 3.2 са дадени примери за четирите основни начини за графично изобразяване на дърво от тип *T*.



Фигура 3.2. Примери за представяне на дърво от тип T: (а) вложени множества; (б) вложени скоби; (в) стъпаловидно отместване; (г) граф

Оригинал: стр. 159

От тази дефиниция се вижда, че линейният списък може да се разглежда като дърво, всеки възел на което има най-много едно поддърво. Затова понякога линейният списък се нарича *изродено дърво*.

Някои определения

Всяко поддърво се нарича *наследник (син)* по отношение на своето дърво и *родител (баща)* по отношение на своите поддървета. Връх *y*, който е непосредствено под връх *x* в графичното изобразяване на дърво, се нарича *пряк потомък (пряк наследник)* на *x*. Върхът *x* се нарича *пряк предшественик (родител)* на върха *y*.

Пример. Върхът *d* на дървото от фиг. 3.2 е пряк предшественик на върховете *h, u* и *v*, а върховете *h, u* и *v* са преки потомци на върха *d*.

Връзките между връх и преките му потомци (ако такива съществуват) се наричат *клони* или *ребра*.

Броят на преките наследници на връх на дърво от тип *T* се нарича негова *степен*.

Пример. Степента на върховете *a* и *d* на дървото от фиг. 3.2 е 3, степента на върха *b* е 2, степента на върховете *c, e* и *h* е 1, а степента на върховете *w, f, g, z, u* и *v* е 0.

Максималната степен измежду степените на всички възли се нарича *степен на дървото*.

Пример. Степента на дървото от фиг. 3.2 е 3.

Връх на дърво от тип *T* със степен 0 се нарича *листо* или *терминален елемент*.

Пример. Върховете *w, f, g, z, u* и *v* на дървото от фиг. 3.2 са листа.

Върховете, които не съвпадат с корена и листата, се наричат *вътрешни върхове*.

Пример. Върховете *b, c, d, e* и *h* на дървото от фигура 3.2 са негови вътрешни върхове.

На всеки връх на непразно дърво от тип *T* може да се съпостави „*ниво*“. Приемаме, че коренът има ниво 1 и ако един връх има ниво *i* и има преки потомци, то те имат ниво *i+1*. Така нивото на връх е всъщност броя на върховете, които трябва да бъдат обходени, като се започне от корена и се стигне до върха.

Оригинал: стр. 160

Най-голямото ниво измежду всички нива на върхове на двоично дърво, се нарича *височина (дълбочина)* на дървото.

Пример. Ако коренът *a* на дървото от фиг. 3.2 има ниво 1, нивото на върховете *b, c* и *d* е 2, нивото на върховете *e, f, g, h, u* и *v* е 3, а нивото на върховете *w* и *z* е 4. Височината на дървото е 4.

Път в дърво от тип T е редица от различни върхове $v_1, v_2, ..., v_k$, като за всеки два последователни върха v_{i-1} и v_i ($2 \leq i \leq k$) е в сила или че v_i е пряк потомък на v_{i-1}, или че v_{i-1} е пряк потомък на v_i. Числото *k* се нарича *дължина на пътя*.

Пример. Редицата от върхове *a, b, e, w* на дървото от пример 3.2 е път от връх *a* до връх *w*, а редицата от върхове *b, a, d, h, z* е път от връх *b* до връх *z*.

Дължина на път в дърво от тип T, започващ от корена

Броят на върховете, през които трябва да се премине, за да се достигне от корена до връх *x* на дърво от тип *T*, е дължината на пътя от корена до *x*. Понякога съкратено се казва *дължина на пътя до x*. Коренът има дължина *1*, преките му потомци имат дължина на пътя *2* и т.н.

Дължината на пътя на едно дърво се дефинира като сума на дължините на пътищата за всички негови върхове. Нарича се още *дължина на вътрешния път*.

Пример. Дължината на вътрешния път на дървото от пример 3.2 е 33 (1+3.2+6.3+2.4).

Ако степента на всеки връх на дърво от тип *T* е по-малка или равна на 2, дървото се нарича *двоично*.

Над структурата от данни дърво от тип T са допустими следните операции:

• достъп до връх

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на дървото.

• включване и изключване на поддърво

Тези операции са възможни в произволно място на дърво, но в резултат трябва да се получи дърво от тип *T*.

• включване и изключване на връх от тип T

Оригинал: стр. 161

Тези операции са възможни в произволно място на дървото, но в резултат трябва да се получи дърво от тип *T*.

• проверка дали дърво е празно

Тъй като дърво може да е празно, за редица приложения се налага да се използва тази операция.

• обхождане на дърво от тип T

Операцията дава метод за достъп до всеки връх на дървото един единствен път.

Осъществява се чрез изпълнение на следните действия в някаква последователност:

– обработка на корена;
– обхождане на наследниците на дървото от тип *T*.

Забелязваме, че процесът на обхождане е рекурсивен.

В практиката се използват:

– *предхождащо обхождане* (коренът се обработва преди наследниците);
– *последващо обхождане* (коренът се обработва след наследниците).

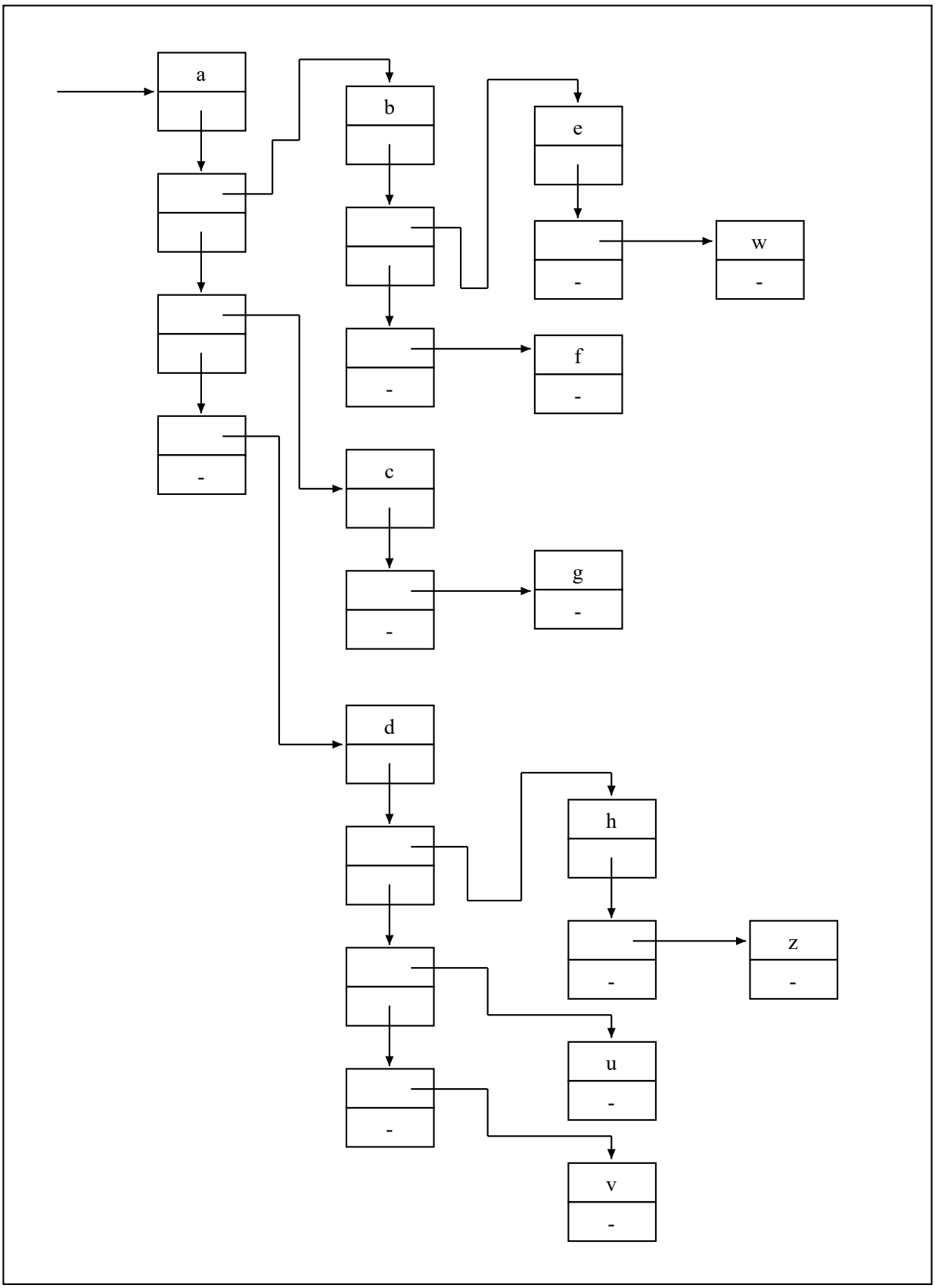
Пример. За дървото от фиг. 3.2 предхождащото обхождане има вида *a, b, e, w, f, c, g, d, h, z, u, v*, а последващото обхождане – *w, e, f, b, g, c, z, h, u, v, d, a*.

Физическо представяне

Най-често се използва свързано представяне на дърво от тип T. За целта се използва линеен списък, представен с една връзка. При това физическо представяне в един свързан списък се реализират коренът и указателите към поддърветата на дървото от тип *T*.

Пример. Дървото от фиг. 3.2 физически се представя както е показано на фиг. 3.3.

Оригинал: стр. 162



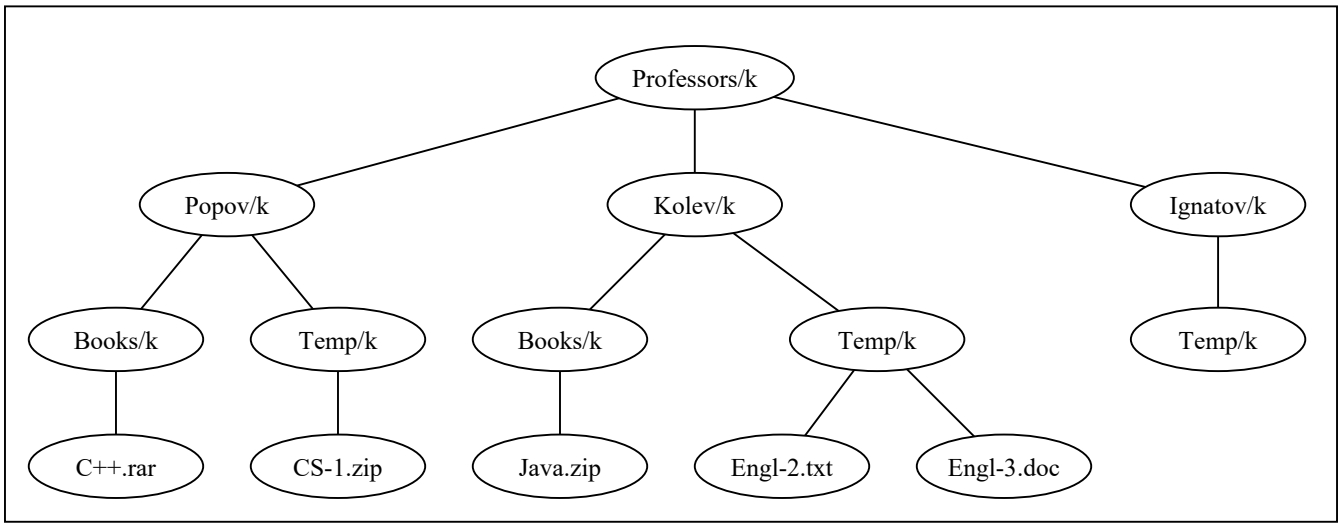
Фигура 3.3. Пример за физическо представяне на дърво от тип T

Оригинал: стр. 163

3.1.2. Приложения на структурата от данни дърво

Едно от приложенията на структурата от данни дърво е изграждането и поддържането на дървовидни каталози на файлове в операционните системи.

Пример. Каталозите на файлове на фиг. 3.4 са маркирани с /k. Достъпът до файла *CS-1.zip* се осъществява чрез цитиране на пътя *Professors/Popov/Temp/CS-1.zip*.



Фигура 3.4. Примерен каталог на файлове

Предхождащото обхождане на каталога от примера може да се използва при извеждане на екрана на структурата на каталога на файлове. Последващото обхождане може да се използва за намиране на обема памет, заемана от всеки каталог и файл.

Чрез структурата от данни дърво могат да се изграждат и поддържат информационни системи, описващи родови връзки, биологични класификационни системи, йерархични системи за управление на фирми и др.

В следващите два параграфа ще разгледаме структурите от данни двоично дърво и двоично наредено дърво.

Оригинал: стр. 164

3.2. ДВОИЧНО ДЪРВО

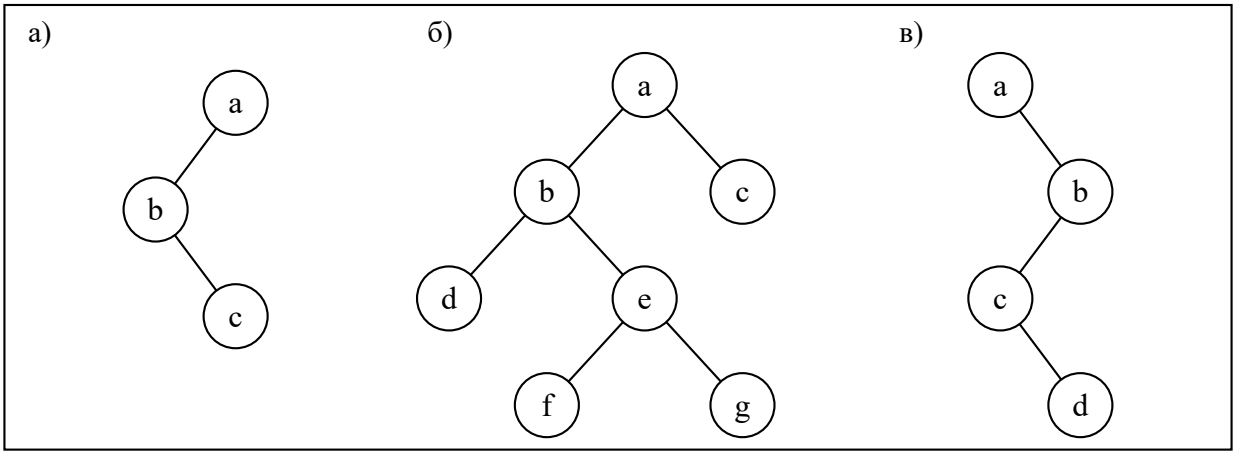
3.2.1. Дефиниране на двоично дърво

Логическо описание

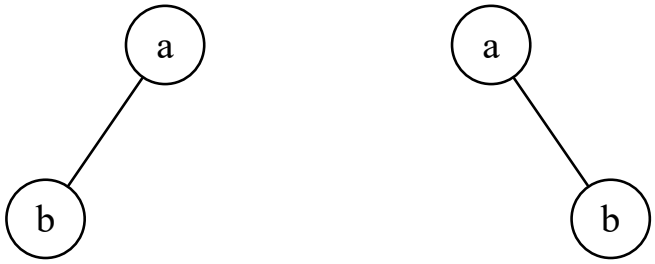
Двоично дърво от тип T е структура от данни, която е или празна, или е образувана от:

– данна от тип T, наречена *корен (връх, възел, ключ)* на дървото от тип *T*;
– двоично дърво от тип *T*, наречено *ляво поддърво* (ЛПД) на двоичното дърво от тип *T*;
– двоично дърво от тип *T*, наречено дясно *поддърво* (ДПД) на двоичното дърво от тип *T*.

Примери. Нека *a, b, c, d, e, f* и *g* са данни от тип *T*. Тогава следните графични изображения определят двоични дървета от тип *T*.



Посоката на линиите, свързващи върховете с поддървета, позволява да се различи ляво от дясно поддърво. Двоичните дървета



са различни. В единия случай дясното поддърво е празно, а в другия – дясното дърво не е празно.

Ще отбележим, че след въведената наредба на поддърветата в структурно отношение двоичните дървета от тип *T* съществено се различават от дърветата от тип *T* и не могат да бъдат разглеждани като тяхно подмножество.

Оригинал: стр. 165

Над структурата от данни двоично дърво са допустими следните операции:

• достъп до връх

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на двоичното дърво.

• включване и изключване на връх

Тези операции са възможни в произволно място на двоичното дърво, но в резултат трябва да се получи двоично дърво от тип *T*.

• проверка дали двоично дърво е празно

Тъй като двоично дърво може да е празно, за редица приложения се налага да се използва тази операция.

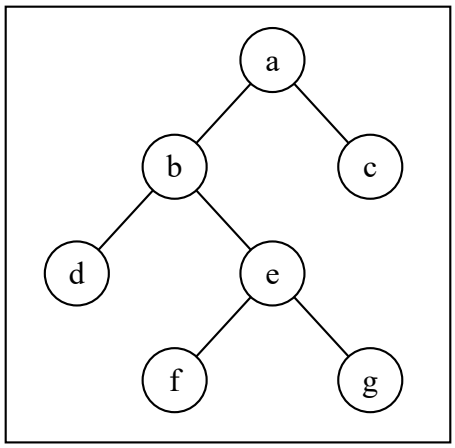
• обхождане на двоично дърво

Операцията дава метод за достъп до всеки връх на двоичното дърво един единствен път.

Осъществява се чрез изпълнение на следните три действия в някаква последователност:
– обработка на корена;
– обхождане на лявото поддърво;
– обхождане на дясното поддърво.

Процесът на обхождане е рекурсивен. Съществуват шест различни начина за обхождане на двоично дърво: ляво-корен-дясно (ЛКД или *смесено обхождане*), корен-ляво-дясно (КЛД или *низходящо обхождане*), ляво-дясно-корен (ЛДК или *възходящо обхождане*), дясно-корен-ляво (ДКЛ), корен-дясно-ляво (КДЛ) и дясно-ляво-корен (ДЛК). Обхождането ЛКД например означава, че първо се обхожда лявото поддърво, след него се обработва коренът и накрая се обхожда дясното поддърво.

Пример. За двоичното дърво



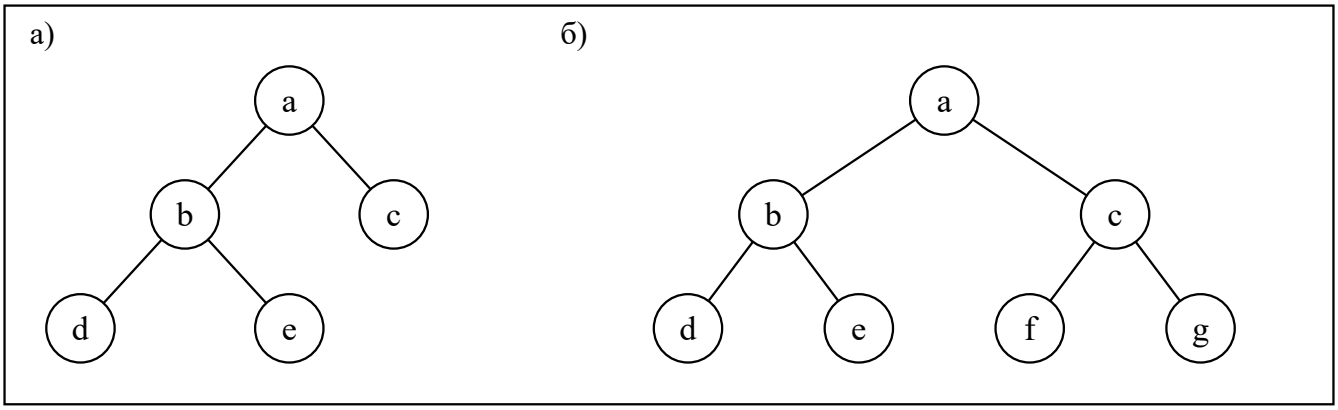
обхожданията имат вида:

Оригинал: стр. 166

ЛКД: dbfegac	ДКЛ: cagefbd
КЛД: abdefgc	КДЛ: acbegfd
ЛДК: dfgebca	ДЛК: cgfedba

Двоично дърво, в което всеки възел, който не е листо, има точно два наследника, се нарича *строго*. Ако всички листа на строго двоично дърво са на едно и също ниво, двоичното дърво се нарича *пълно*.

Пример. На фиг. 3.5 (а) и (б) са илюстрирани две строги двоични дървета. Двоичното дърво от фиг. 3.5. (б) е *пълно*.



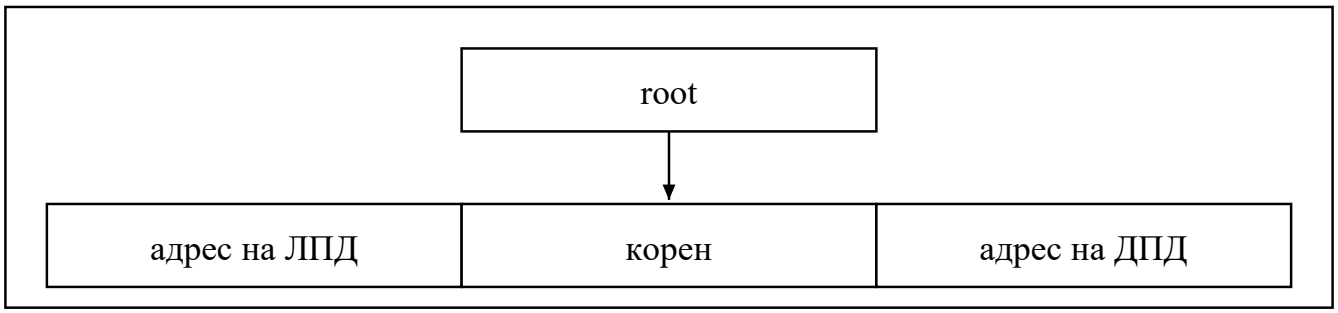
Фигура 3.5. Строги двоични дървета (а) и (б) и пълно двоично дърво (б)

Физическо представяне

Използват се главно два начина за физическо представяне на двоично дърво от тип *Т – свързано* и *верижно*.

Свързано представяне

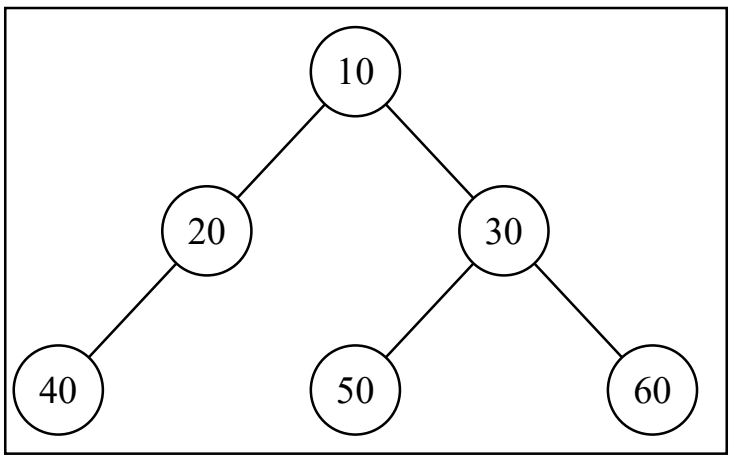
Реализира се чрез указател към кутия с три полета: информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на ЛПД и ДПД съответно (Фиг. 3.6).



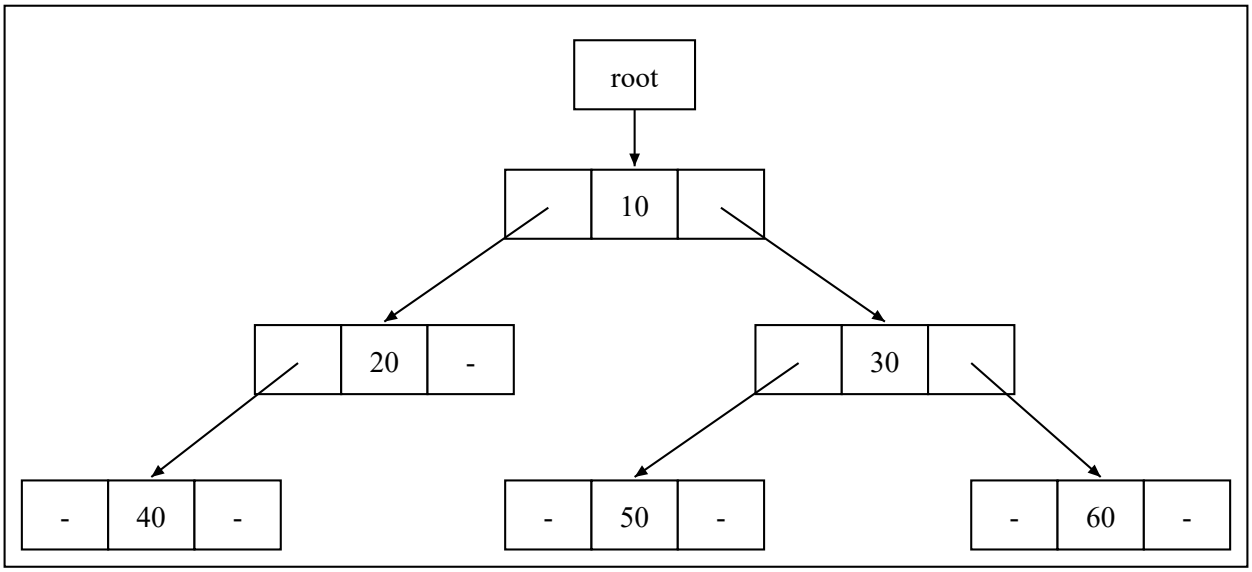
Фигура 3.6. Свързано представяне на двоично дърво

Оригинал: стр. 167

Пример: Двоичното дърво



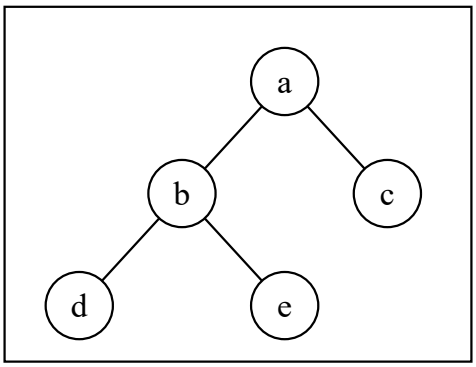
се представя по следния начин:



Верижно представяне

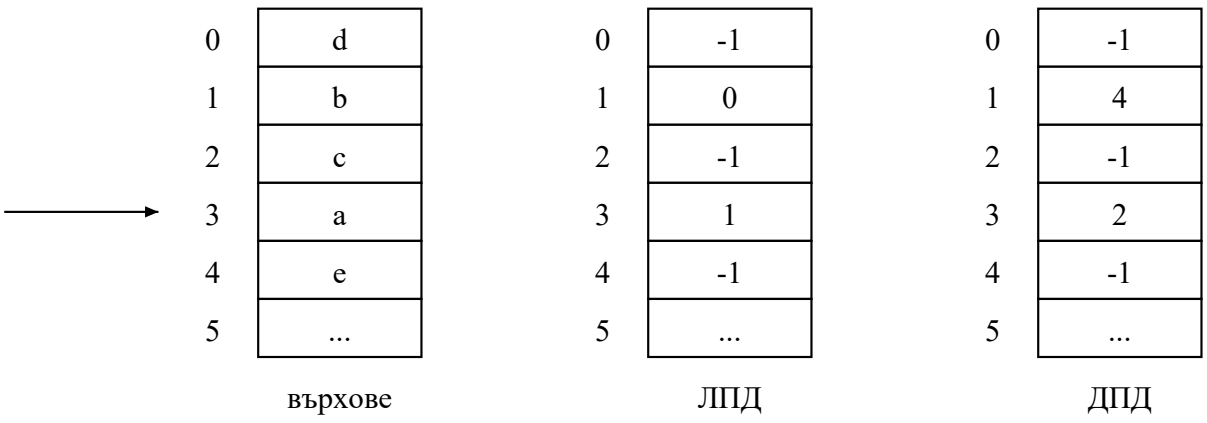
Това представяне се нарича още *последователно*. Използват се три масива – за представяне на върховете на дървото, за адресите на лявото и за адресите на дясното поддърво. Ролята на адреси се изпълнява от индекси. *i*-ят елемент на масива „върхове“ съдържа стойността на връх на двоичното дърво. *i*-ят елемент на масива „ЛПД“ съдържа адреса на лявото поддърво на поддървото с корен *i*-я елемент, а *i*-ят елемент на масива „ДПД“ – адреса на дясното му поддърво. Поддържа се указател, който съдържа адреса на корена.

Пример. Верижното представяне на двоичното дърво от тип *T*



Оригинал: стр. 168

може да има вида:



Указателят към корена е 3. Празното дърво е представено чрез -1.

В книгата ще реализираме и използваме само свързаното представяне на двоично дърво.

3.2.2. Реализация на свързаното представяне

Тройната кутия, съдържаща корена и адресите на лявото и дясното поддървета, представяме чрез следния шаблон на структурата *node*.

C168-01 · Копиране: code.html
```
template <typename T = int>
struct node
{ T inf;
  node *left,
       *right;
};
```

Двоичното дърво ще представим чрез указател към тройна кутия от вида, описан по-горе. Ще го реализираме чрез шаблона на класа *tree*.

C168-02 · Копиране: code.html
```
template <typename T = int>
class tree
{ public:
    // канонично представяне
    tree();
    ~tree();
    tree(const tree&);
    tree& operator=(const tree&);
    // основни операции
    bool empty() const;          // проверява дали двоично
                                 // дърво е празно
    node<T>* getRoot() const;    // достъп до корена
    T rootTree() const;          // намира корена
                                 // на двоично дърво
```

Оригинал: стр. 169

C169-01 · Копиране: code.html

```
    tree leftTree() const;  // намира лявото поддърво
    tree rightTree() const; // намира дясното поддърво
    // създава непразно двоично дърво по дадени
    // корен, ляво и дясно поддърво
    void create3(const T&, const tree<T>&, const tree<T>&);
    // извежда двоично дърво
    void print() const
    { pr(root);
      cout << endl;
    }
    // създава непразно двоично дърво
    // чрез въвеждане на върховете му
    void create()
    { createTree(root);
    }
  private:
    node<T>* root;          // указател към корена
    // помощни член-функции
    // създава копие на дадено двоично дърво на друго
    // място в паметта, сочено от указателя this
    void copyTree(const tree<T>&);
    // помощна за copyTree член-функция
    void copy(node<T>*&, node<T>* const &) const;
    // изтрива даденото чрез указател
    // към корена двоично дърво
    void deleteTree(node<T>*&) const;
    // помощна за print член-функция
    void pr(const node<T>*) const;
    // помощна за create член-функция
    void createTree(node<T>*&) const;
};
```

Член-функциите на каноничното представяне реализират познати идеи. Конструкторът по подразбиране създава празно двоично дърво.

C169-02 · Копиране: code.html

```
template <typename T>
tree<T>::tree()
{ root = NULL;
}

template <typename T>
```

C170-01 · Копиране: code.html
```
tree<T>::~tree()
{ deleteTree(root);
}

template <typename T>
tree<T>::tree(const tree<T>& r)
{ copyTree(r);
}

template <typename T>
tree<T>& tree<T>::operator=(const tree<T>& r)
{ if (this != &r)
    { deleteTree(root);
      copyTree(r);
    }
  return *this;
}
```

За реализирането им използваме член-функциите *deleteTree, copyTree* и *copy* на шаблона на класа *tree*.

C170-02 · Копиране: code.html
```
template <typename T>
void tree<T>::copyTree(const tree<T>& r)
{ copy(root, r.root);
}

template <typename T>
void tree<T>::copy(node<T>*& pos, node<T>* const& r) const
{ pos = NULL;
  if (r)
    { pos = new node<T>;
      assert(pos != NULL);
      pos->inf = r->inf;
      copy(pos->left, r->left);
      copy(pos->right, r->right);
    }
}

template <typename T>
void tree<T>::deleteTree(node<T>*& p) const
{ if (p)
```

C171-01 · Копиране: code.html
```
    { deleteTree(p->left);
      deleteTree(p->right);
      delete p;
      p = NULL;
    }
}
```

Използването на допълнителния параметър от тип *node<T>*&* в *deleteTree* и *copy* е заради рекурсията. Проверката дали двоично дърво е празно се осъществява чрез булевата член-функция *empty* на шаблона.

C171-02 · Копиране: code.html
```
template <typename T>
bool tree<T>::empty() const
{ return root == NULL;
}
```

За бързо обхождане на елементите на двоично дърво е предвидена член-функция за достъп до корена на двоично дърво.

C171-03 · Копиране: code.html
```
template <typename T>
node<T>* tree<T>::getRoot() const
{ return root;
}
```

Достъпът до корена, до лявото и до дясното поддърво на непразно двоично дърво се осъществява съответно чрез член-функциите *rootTree, leftTree* и *rightTree*.

C171-04 · Копиране: code.html
```
template <typename T>
T tree<T>::rootTree() const
{ return root->inf;
}

template <typename T>
tree<T> tree<T>::leftTree() const
{ tree<T> t;
  copy(t.root, root->left);
  return t;
}

template <typename T>
tree<T> tree<T>::rightTree() const
```

Оригинал: стр. 172

C172-01 · Копиране: code.html
```
{ tree<T> t;
  copy(t.root, root->right);
  return t;
}
```

Извеждането на елементите на двоично дърво се реализира чрез член-функцията *print*. Тъй като реализацията й е рекурсивна, *print* използва помощната член-функция *pr*. Направената реализация обхожда двоичното дърво по метода ЛКД. В нея се предполага, че за елементите на типа *T* е дефиниран операторът <<.

C172-02 · Копиране: code.html
```
template <typename T>
void tree<T>::pr(const node<T>* p) const
{ if (p)
    { pr(p->left);
      cout << p->inf << " ";
      pr(p->right);
    }
}
```

Следващите две член-функции създават непразно двоично дърво. Член-функцията *create3* създава непразно двоично дърво по дадени корен, ляво и дясно поддървета.

C172-03 · Копиране: code.html
```
template <typename T>
void tree<T>::create3(const T& x, const tree<T>& l,
                                  const tree<T>& r)
{ if (root) deleteTree(root);
  root = new node<T>;
  assert(root != NULL);
  root->inf = x;
  copy(root->left, l.root);
  copy(root->right, r.root);
}
```

Член-функцията *create* създава непразно двоично дърво чрез въвеждане на върховете му в последователността КЛД. Тя използва член-функцията *createTree*, дефинирана рекурсивно по следния начин:

C172-04 · Копиране: code.html
```
template <typename T>
void tree<T>::createTree(node<T>*& pos) const
{ T x; char c;
  cout << "root: "; cin >> x;
```

Оригинал: стр. 173

C173-01 · Копиране: code.html
```
  pos = new node<T>;
  assert(pos != NULL);
  pos->inf = x;
  pos->left = NULL;
  pos->right = NULL;
  cout << "Left Tree of: " << x << "  y/n? ";
  cin >> c;
  if (c == 'y') createTree(pos->left);
  cout << "Right Tree of: " << x << " y/n? ";
  cin >> c;
  if (c == 'y') createTree(pos->right);
}
```

Дефинираният шаблон записваме във файла *tree.cpp* и ще го използваме в следващите задачи.

Задача 26. Да се напише програма, която създава непразно двоично дърво от цели числа. Намира и извежда корена, лявото и дясното му поддърво. Конструира и извежда двоично дърво с корен, съвпадащ с корена на първоначалното двоично дърво, с ляво поддърво – дясното поддърво на първоначалното и дясно поддърво – лявото поддърво на първоначалното двоично дърво.

C173-02 · Копиране: code.html
```
// Program Zad26.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "tree.cpp"

typedef tree<int> intTree;
int main()
{ intTree t;
  t.create();       // създаване на непразно двоично дърво
  t.print();        // извеждане на двоичното дърво
  int x = t.rootTree();        // намиране на корена
  cout << "root: " << x << endl; // извеждане на корена
  intTree t1 = t.leftTree(),   // намиране на ЛПД
          t2 = t.rightTree();  // намиране на ДПД
  cout << "leftTree: \n";
  t1.print();                  // извеждане на ЛПД
  cout << "rightTree: \n";
```

Оригинал: стр. 174

C174-01 · Копиране: code.html
```
  t2.print();                // извеждане на ДПД
  intTree t3;
  t3.create3(x, t2, t1); // генериране на новото двоично дърво
  t3.print();           // извеждане на полученото дърво
  return 0;
}
```

Задача 27. Да се дефинира функция, която увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

C174-02 · Копиране: code.html
```
// Program Zad27.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "tree.cpp"

typedef tree<int> intTree;

intTree addElem(int a, const intTree& t)
{ intTree t1;       // създаване на празно двоично дърво
  if (!t.empty())
    // създаване на търсеното двоично дърво
    t1.create3(t.rootTree() + a,
               addElem(a, t.leftTree()),
               addElem(a, t.rightTree()));
  return t1;
}

int main()
{ intTree t;
  t.create();
  t.print();
  cout << "Number: "; int a; cin >> a;
  addElem(a, t).print();
  return 0;
}
```

Задача 28. Да се дефинира шаблон на булева функция *equal*, който проверява дали двоичните дървета *t1* и *t2* от тип *T* са равни.

C174-03 · Копиране: code.html
```
template <typename T>
```

Оригинал: стр. 175

C175-01 · Копиране: code.html
```
bool equal(const tree<T>& t1, const tree<T>& t2)
{ if (t1.empty() && t2.empty()) return true;
  if (t1.empty() && !t2.empty() ||
      !t1.empty() && t2.empty()) return false;
  if (t1.rootTree() != t2.rootTree()) return false;
  return equal(t1.leftTree(), t2.leftTree()) &&
         equal(t1.rightTree(), t2.rightTree());
}
```

Задача 29. Да се дефинира шаблон на цяла функция *depth*, която намира дълбочината на двоично дърво от тип *T*, т.е. броя на върховете в най-дългия път от корена до листо.

C175-02 · Копиране: code.html
```
template <typename T>
int depth(const tree<T>& t)
{ if (t.empty()) return 0;
  int n, m;
  n = depth(t.leftTree());
  m = depth(t.rightTree());
  if (n > m) return n + 1;
  return m + 1;
}
```

3.2.3. Функции от по-висок ред за работа с двоични дървета

Има смисъл да бъдат реализирани и използвани шаблони на функциите от по-висок ред *accumulate* и *map*.

Задача 30. Да се дефинира функцията от по-висок ред *accumulate*, прилагаща бинарната лявоасоциативна операция *op* с нулева стойност *null_val* над елементите на дадено двоично дърво в реда ЛКД.

C175-03 · Копиране: code.html
```
template <typename T>
T accumulate(T (*op)(T, T), T null_val, const tree<T>& t)
{ if (!t.empty())
    return op(op(accumulate(op, null_val, t.leftTree()),
                 t.rootTree()),
              accumulate(op, null_val, t.rightTree()));
  return null_val;
}
```

Оригинал: стр. 176

Задача 31. Като се използва функцията *accumulate*, да се намери сумата и произведението на върховете на дадено двоично дърво от тип *int*.

Ако

C176-01 · Копиране: code.html
```
typedef tree<int> intTree;
int sum(int x, int y)
{ return x + y;
}

int po(int x, int y)
{ return x * y;
}
```

следващата главна функция решава задачата

C176-02 · Копиране: code.html
```
int main()
{ intTree t;
  t.create();
  t.print();
  cout << accumulate(sum, 0, t) << endl;
  cout << accumulate(po, 1, t) << endl;
  return 0;
}
```

Задача 32. Да се дефинира функцията от по-висок ред *map*, прилагаща едноаргументната функция *f* към всеки от елементите на дадено двоично дърво.

C176-03 · Копиране: code.html
```
template <typename T>
tree<T> map(T (*f)(T), const tree<T>& t)
{ tree<T> t1;
  if (!t.empty())
    t1.create3(f(t.rootTree()), map(f, t.leftTree()),
                                map(f, t.rightTree()));
  return t1;
}
```

Задача 33. Дадено е двоично дърво с естествени числа във върховете. Като се използва функцията *map*, да се създаде ново двоично дърво, което има структурата на даденото и върховете му са с 1 по-големи от съответните върхове на даденото двоично дърво. Да се създаде също двоично дърво, което има структурата на даденото, като върховете му са факториелите на съответните върхове на даденото двоично дърво.

Оригинал: стр. 177

Ако

C177-01 · Копиране: code.html
```
typedef tree<int> intTree;
int f(int x)
{ return x + 1;
}
int g(int x)
{ int f = 1;
  for (int i = 2; i <= x; i++)
    f = f*i;
  return f;
}
```

следващата функция решава задачата

C177-02 · Копиране: code.html
```
int main()
{ intTree t;
  t.create();
```

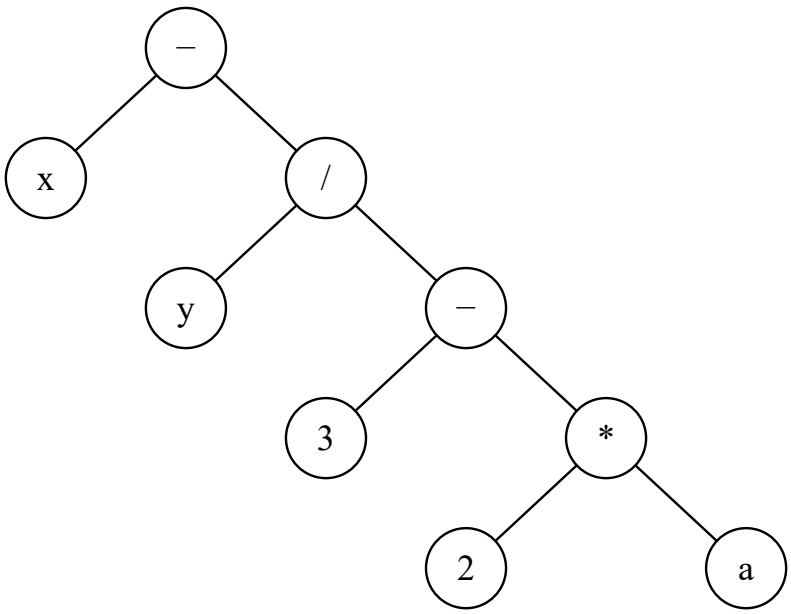
C177-02 · продължение · Копиране: code.html

```
  t.print();
  map(f, t).print();    // увеличаване с 1 и извеждане
  map(g, t).print();    // намиране на дървото
                       // с факториелите и извеждане
  return 0;
}
```

3.2.3. Прилагане на двоични дървета за представяне на аритметични изрази

Всеки аритметичен израз може да се представи чрез двоично дърво, в листата на което са операндите, а във вътрешните върхове и корена – операциите.

Пример. Изразът *x-y/(3-2*a)* може да се представи чрез двоичното дърво



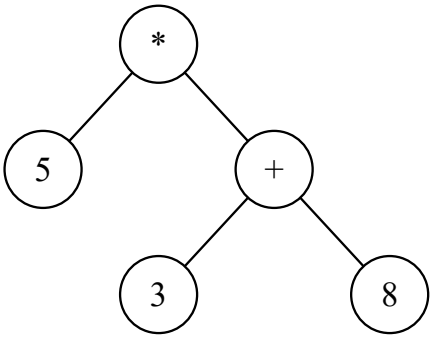
Оригинал: стр. 178

Възходящият обход на двоично дърво, представящо аритметичен израз, получава обратния полски запис на аритметичния израз. Смесеният обход на двоично дърво, представящо аритметичен израз, дава общоприетия (инфиксен) запис на аритметичния израз, но без скобите, а низходящият обход на двоично дърво, представящо аритметичен израз, получава правия полски запис на аритметичния израз.

Задача 34. Формулата

<формула> ::= <терминал>|
(<формула><знак><формула>)
<знак> ::= +|-|*|/
<терминал> ::= 0|1| ...|9

може да се представи във вид на двоично дърво от тип *char* съгласно следното правило: формула състояща се от един терминал се представя чрез двоично дърво с един връх – цифрата; формула от вида *(f1 s f2)* се представя чрез двоично дърво, коренът на което е знакът *s*, ЛПД съответства на формулата *f1*, ДПД – на формулата *f2*. Например двоичното дърво



представя формулата (5 * (3 + 8))

Да се напише рекурсивна функция, която:
а) създава двоично дърво, представляващо правилна формула от горния вид;
б) проверява, явява ли се двоично дърво, дърво на формула;
в) намира стойността на формула, представена с двоично дърво;
г) извежда двоично дърво във вид, съответстващ на формула.

Дървото може да се създаде чрез член-функцията *create* на шаблона на класа *tree*. В програмата е предложено създаване на двоично дърво, представящо въведена от клавиатурата *правилно записана* формула от горния вид.

C178-01 · Копиране: code.html
```
// Program Zad34.cpp
#include <iostream>
#include <cassert>
#include "tree.cpp"
using namespace std;
```

Оригинал: стр. 179

C179-01 · Копиране: code.html

```
typedef tree<char> charTree;
// а
charTree create_tree_form()
{ char ch; charTree t;
  cin >> ch;
  if (ch >= '0' && ch <= '9')
    { charTree t1;
      t.create3(ch, t1, t1);
    }
  else
    { charTree t1, t2;
      t1 = create_tree_form();
      cin >> ch;
      t2 = create_tree_form();
      t.create3(ch, t1, t2);
      cin >> ch;
    }
  return t;
}
// б)
bool isForm(const charTree& t)
{ if (t.empty()) return false;
  char c = t.rootTree();
  if (c >= '0' && c <= '9')
    return t.leftTree().empty() && t.rightTree().empty();
  if (c != '+' && c != '-' &&
      c != '*' && c != '/') return false;
  return isForm(t.leftTree()) && isForm(t.rightTree());
}
// в)
int arrExpr(const charTree& t)
{ char c = t.rootTree();
  if (c >= '0' && c <= '9') return (int)c - (int)'0';
  switch (c)
  { case '+': return arrExpr(t.leftTree()) +
                     arrExpr(t.rightTree());
    case '-': return arrExpr(t.leftTree()) -
                     arrExpr(t.rightTree());
    case '*': return arrExpr(t.leftTree()) *
                     arrExpr(t.rightTree());
```

Оригинал: стр. 180

C180-01 · Копиране: code.html
```
    case '/': int denom = arrExpr(t.rightTree());
              if (denom != 0)
                return arrExpr(t.leftTree())/denom;
              else
                { cout << "Error! Division by 0!\n";
                  return 0;
                }
  }
  return INT_MAX;   // добавено е за коректност на кода
}
// г)
void print_tree(const charTree& t)
{ char c = t.rootTree();
  if (c >= '0' && c <= '9') cout << c;
  else
    { cout << '(';
      print_tree(t.leftTree());
      cout << c;
      print_tree(t.rightTree());
      cout << ')';
    }
}

int main()
{ charTree t;
  t.create(); // или t=create_tree_form();
  if (isForm(t))
    { print_tree(t);
      cout << endl << arrExpr(t) << endl;
    }
  else cout << "It is not a formula.\n";
  return 0;
}
```

Задача 35. Нека в двоично дърво от тип *char* е записана формула съгласно синтаксиса от предишната задача, но в качество на терминали се използват не само цифри, а и букви, играещи ролята на променливи. Да се дефинира функция, която:
а) опростява двоично дърво, представящо формула, като заменя в него всички поддървета, съответстващи на формулите *(f+0), (0+f), (f-0), (f*1), (1*f)*

Оригинал: стр. 181

и *(f/1)* с поддърво, съответстващо на формулата *f*, а поддърветата, съответстващи на формулите *(f*0), (0*f)* и *(0/f)* – с върха 0;
б) преобразува двоично дърво, представляващо формула, като заменя в него всички поддървета, съответстващи на формулите *((f1+f2)*f3), ((f1-f2)*f3), (f1*(f2+f3)), (f1*(f2-f3)), ((f1+f2)/f3)* и *((f1-f2)/f3)* – с поддървета, съответстващи на формулите *((f1*f3)+(f2*f3)), ((f1*f3)-(f2*f3)), ((f1*f2)+(f1*f3)), ((f1*f2)-(f1*f3)), ((f1/f3)+(f2/f3))* и *((f1/f3)-(f2/f3))* съответно.

Програма *Zad35.cpp* решава условие а) на задачата. Условие б) оставяме за самостоятелна работа.

C181-01 · Копиране: code.html

```
// Program Zad35.cpp
#include <iostream>
#include <cassert>
using namespace std;
#include "tree.cpp"

typedef tree<char> charTree;
void reduce(charTree& t)
{ char c = t.rootTree();
  if (c == '+' || c == '-' || c == '*' || c == '/')
    { charTree t1, t2;
      t1 = t.leftTree();
      t2 = t.rightTree();
      reduce(t1);
      reduce(t2);
      t.create3(c, t1, t2);
      if (c == '+' && t1.rootTree() == '0' ||
          c == '*' && t1.rootTree() == '1')
        t = t2;
      else
        if ((c == '*' || c == '/') && t2.rootTree() == '1' ||
            (c == '+' || c == '-') && t2.rootTree() == '0')
          t = t1;
        else
          if (c == '*' && (t1.rootTree() == '0' ||
                           t2.rootTree() == '0') ||
              c == '/' && t1.rootTree() == '0')
            { charTree t3;
              t.create3('0', t3, t3);
```

Оригинал: стр. 182

C182-01 · Копиране: code.html

```
            }
    }
}

int main()
{ charTree t;
  t.create();
  reduce(t);
  t.print();
  return 0;
}
```

Задача 36. Да се дефинира шаблон на булева функция, който реализира проверка за принадлежност на елемент в двоично дърво.

C182-02 · Копиране: code.html

```
template <typename T>
bool member(const T& a, const tree<T>& t)
{ if (t.empty()) return false;
  if (a == t.rootTree()) return true;
  return member(a, t.leftTree()) || member(a, t.rightTree());
}
```

Това решение е неефективно, тъй като, ако елементът не се съдържа в дървото, се налага обхождане на дървото с пълно изчерпване. Операциите включване и изключване на връх също не се реализират добре за обикновените дървета. За редица цели е по-удобно използването на т. нар. *двоично наредени дървета* или *двоични справочници*. За тези двоични дървета сложността на операциите включване, изключване и търсене на елемент (ключ) е логаритмична.

3.2.4. Прилагане на двоични дървета за декодиране на Морзови съобщения

Морзовата азбука е начин за предаване на буквите чрез манипулатор (ключ) от азбуката с помощта на дълги и къси сигнали, условно наречени „тире“ и „точка“. Тя е първият метод за пренос на информация посредством използването на предавател и приемник. Използвала се е в летищата, телеграфията, радиотелеграфията и др. Доказано е, че при съотношение *сигнал : шум* равно на 3:1 човешкото ухо може да възприеме предавания сигнал. За единица време се приема времетраенето на късия сигнал – точ-

Оригинал: стр. 183

ката, дължината на тирето е равна на 3 единици време. Паузата между знаците в буквата е една единица време, между буквите в думата – 3 единици време, между думите – 7 единици време. В таблица 3.1 е дадено Морзовото кодиране за главните латински букви и някои препинателни знаци. Морзовият код ще разглеждаме като редица от точки, тирета и интервали. За краткост в следващите разглеждания между буквите ще поставяме 1 интервал, а между думите – 2 интервала.

Пример. Изречението „ALL RIGHTS RESERVED“ се кодира по следния начин:

C183-01 · Копиране: code.html

```
.- .-.. .-..  .-. .. --. .... - ...  .-. . ... . .-. ...- . -..
```

Знак	Код	Знак	Код	Знак	Код	Знак	Код	Знак	Код	Знак	Код
A	.-	J	.---	S	...	1	.----	.	.-.-.-	:	---...
B	-...	K	-.-	T	-	2	..---	,	--..--	;	-.-.-.
C	-.-.	L	.-..	U	..-	3	...--	?	..--..	=	-...-
D	-..	M	--	V	...-	4	....-	'	.----.	@	.--.-.
E	.	N	-.	W	.--	5	.....	!	-.-.--	-	-....-
F	..-.	O	---	X	-..-	6	-....	/	-..-.	_	..--.-

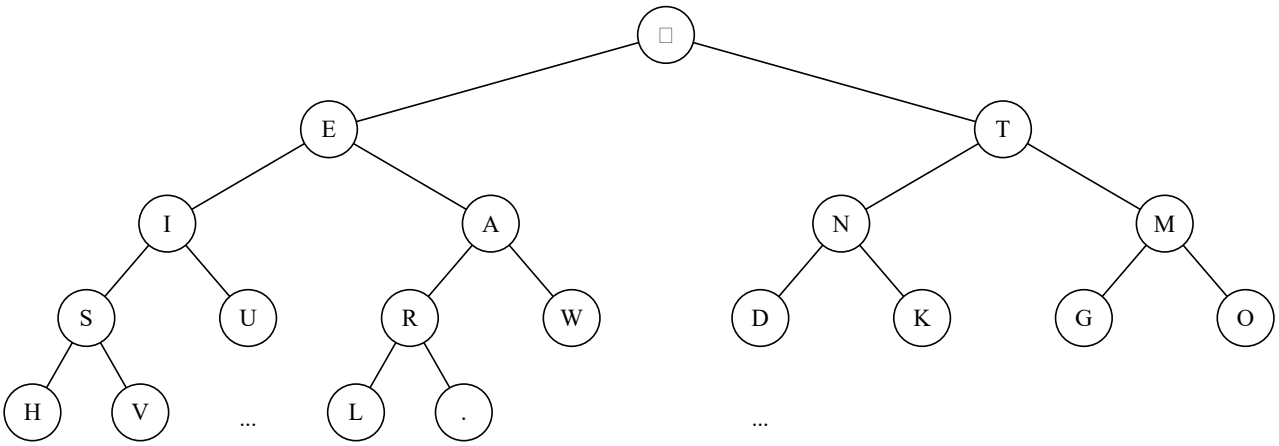
Знак	Код	Знак	Код	Знак	Код	Знак	Код	Знак	Код	Знак	Код
G	--.	P	.--.	Y	-.--	7	--...	(	-.--.	"	.-..-.
H	....	Q	--.-	Z	--..	8	---..	)	-.--.-	$	...-..-
I	..	R	.-.	0	-----	9	----.	&	. ...		

Таблица 3.1. Морзова азбука

Декодирането на Морзово писмо може да стане, като се използва двоично дърво, част от което е представено на фиг. 3.7. В корена на дървото е поставен празният низ. Интерпретирането на текст, кодиран чрез Морзовата азбука, става, като се започва от корена. При срещане на символа ‘.’ се тръгва към

Оригинал: стр. 184

корена на лявото поддърво, при срещане на символа ‘-‘ се тръгва към корена на дясното поддърво, а при срещане на интервал се чете коренът, с което е интерпретирана съответната буква. Следващата интерпретация започва отново от корена.



Фигура 3.7. Част от двоично дърво за интерпретиране на Морзовата азбука

Пример. Низът .- .-.. .-.. .-. .. --. .... - ... .-. . ... . .-. ...- . -.. се обработва по следния начин: започва се от корена. Първият символ е ‘.’, което води до достигане до корена ‘E’ на ЛПД. Следващият символ е ‘-’. Това води до достигане до корена ‘A’ на ДПД на дървото с корен ‘E’. Следващият символ е ‘ ’. Това води до извличане на върха ‘A’. Следващата интерпретация започва от корена на двоичното дърво. Поредният символ е ‘.’. Това води до достигане до корена ‘E’ на ЛПД. Следва символът ‘-’. Това води до достигане до корена ‘A’ на ДПД на дървото с корен ‘E’. Следват две точки, което води до върха ‘L’. Следващият символ е ‘ ’, което води до извличане на върха ‘L’. Следващата интерпретация започва от корена на двоичното дърво и т.н.

Задача 37. Да се напише програма, която:
– създава двоично дърво, представящо Морзовата азбука;
– реализира описания по-горе алгоритъм за разшифриране на текст, кодиран правилно чрез азбуката на Морз.

Оригинал: стр. 185

C185-01 · Копиране: code.html

```
// Program Zad37.cpp
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
#include "tree.cpp"

typedef tree<char> morseTree;

int main()
{ morseTree tr;
  tr.create(); tr.print(); // създава и извежда двоичното
                           // дърво с Морзовата азбука
  node<char>* ptr = tr.getRoot();
  cout << "Enter a sentence!\n";
  char ch, ch1 = '.'; cin.get(ch);
  while(cin)
  { if (ch == '.') { ptr = ptr->left; ch1 = '.'; } else
      if (ch == '-') { ptr = ptr->right; ch1 = '-'; }
      else
        if (ch == ' ')
          { if (ch1 != ch)
              { cout << ptr->inf;
                ptr = tr.getRoot();
                ch1 = ch;
              }
            else
              { cout << ' ';
                ptr = tr.getRoot();
                ch1 = '.';
              }
          }
    cin.get(ch);
  }
  return 0;
}
```

Оригинал: стр. 186

3.3. ДВОИЧНО НАРЕДЕНО ДЪРВО

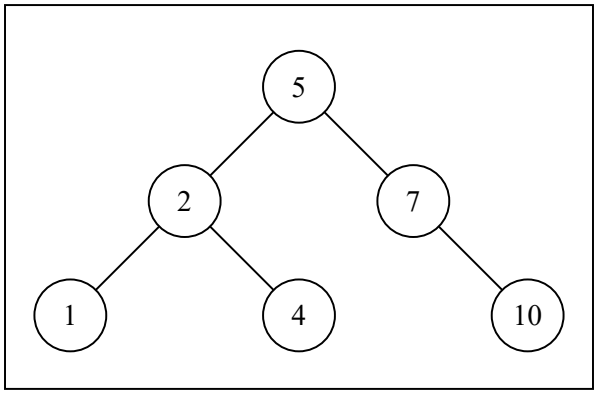
3.3.1. Дефиниране на двоично наредено дърво (ДНД)

Логическо описание

Предполагаме, че за елементите от тип *T* е установена наредба. Празното двоично дърво е *двоично наредено дърво*. Непразно двоично дърво с върхове от тип *T*, за което са в сила:
– върховете на лявото му поддърво са по-малки от корена,
– върховете на дясното му поддърво са по-големи от корена,
– лявото и дясното поддървета са двоично наредени дървета от тип *T*

се нарича *непразно двоично наредено дърво от тип T*.

Пример. Двоичното дърво от тип *int*



е наредено.

Над структурата от данни двоично наредено дърво са допустими следните операции:

• *достъп до връх*

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на двоичното дърво.

• *включване и изключване на връх*

Тези операции са възможни в произволно място на двоичното дърво, но в резултат трябва да се получи двоично наредено дърво от тип *T*.

Включване

Нека *t* е двоично наредено дърво от тип *T*. Включването на елемента *a* от тип *T* в *t* се осъществява по следния начин:

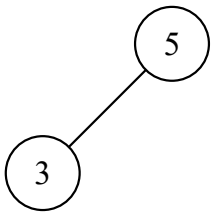
• ако *t* е празното двоично дърво, новото двоично наредено дърво е с корен елемента *a* и с празни ляво и дясно поддървета,

• ако *t* не е празно и *a* е по-малко от корена му, елементът *a* се включва в лявото поддърво на *t*,

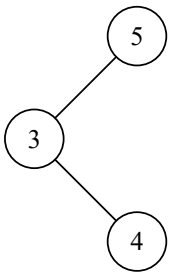
Оригинал: стр. 187

• ако *t* не е празно и *a* е по-голямо от корена му, елементът *a* се включва в дясното поддърво на *t*.

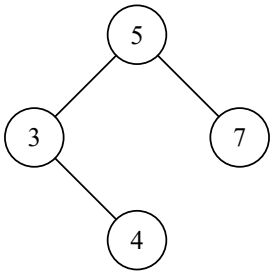
Пример. Ако в двоично нареденото дърво от тип *int* с корен 5 и празни поддървета се включи 3, ще се получи двоично наредено дърво



Ако към полученото двоично наредено дърво се включи 4, ще се получи



Ако към полученото двоично наредено дърво се включи 7, ще се получи



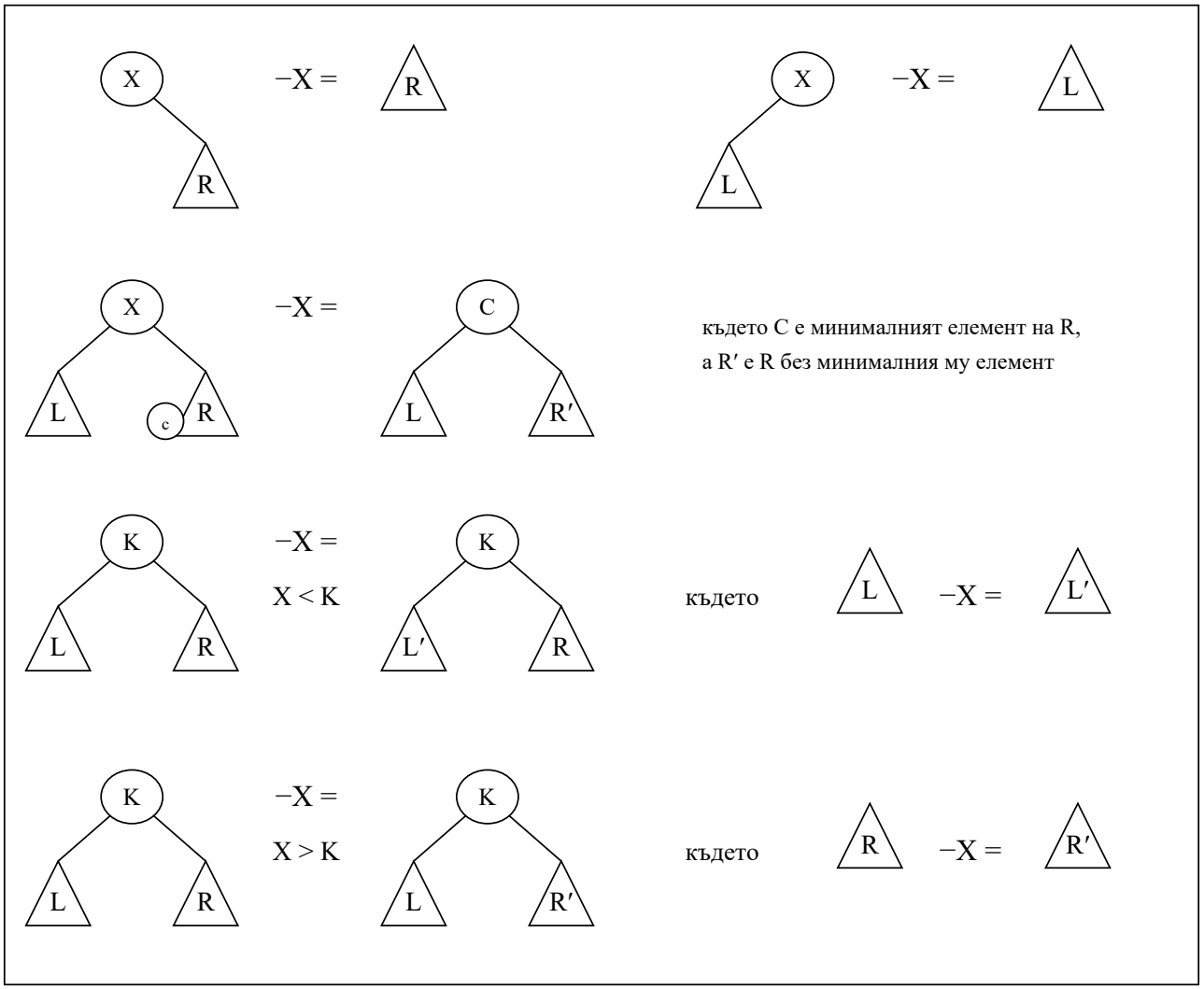
Включваният елемент „се спуска“ по „клоните“ на двоичното дърво, започвайки от корена, и се насочва към лявото или дясното поддърво в зависимост от стойността си, докато намери празно място, което да заеме. Този начин на включване на елемент в двоично наредено дърво ще използваме за създаване на такива дървета.

За някои приложение се допуска двоично наредено дърво да има повтарящи се върхове. В този случай повтарящите се елементи ще включваме в дясното поддърво.

Изключване

Изключването на елемент от двоично наредено дърво графично може да се изобрази, както е показано на фиг. 3.8.

Оригинал: стр. 188



Фигура 3.8. Изключване на елемента X от двоично наредено дърво

• *проверка дали двоично наредено дърво е празно*

Тъй като двоично наредено дърво може да е празно, за редица приложения се налага да се използва тази операция.

• *обхождане на двоично наредено дърво*
Операцията дава метод за достъп до всеки връх на двоично наредената дърво един-единствен път. Осъществява се чрез изпълнение на следните три действия в някаква последователност:
– обработка на корена;
– обхождане на лявото поддърво;
– обхождане на дясното поддърво.

Двоично наредената дърво притежава следното важно свойство: обхождането му по метода ЛКД сортира във възходящ ред елементите от върховете на дървото, а обхождането му по метода ДКЛ сортира в низходящ ред елементите от върховете на дървото.

Оригинал: стр. 189

Физическо представяне

Двоично наредено дърво се представя последователно (верижно) или свързано по същия начин както при обикновените двоични дървета.

Шаблонът на класа *binOrdTree* реализира свързано представяне на двоично наредено дърво от тип *T*.

Реализация

Тройната кутия, съдържаща корена и адресите на лявото и дясното поддърво, представяме чрез следния шаблон на структурата *node_bin*.

C189-01 · Копиране: code.html
```
template <typename T = int>
struct node_bin
{ T inf;
  node_bin<T> *left,
              *right;
};
```

Двоично нареденото дърво ще представим чрез указател към тройна кутия от вида, описан по-горе. Ще го реализираме чрез шаблона на класа *binOrdTree*

C189-02 · Копиране: code.html
```
template <typename T = int>
class binOrdTree
{ public:
    // канонично представяне
    binOrdTree();
    ~binOrdTree();
    binOrdTree(const binOrdTree&);
    binOrdTree& operator=(const binOrdTree&);
    // основни операции
    bool empty() const;    // проверява дали дървото е празно
    T rootTree() const;    // намира на корена
    binOrdTree leftTree() const;   // намира на ЛПД
    binOrdTree rightTree() const;  // намира на ДПД
    // създава непразно двоично наредено
    // дърво по указани върхове
    void create();
    // създава непразно двоично наредено
    // дърво по указани корен, ЛПД и ДПД
    void create3(const T&, const binOrdTree&,
                           const binOrdTree&);
    // извежда двоично наредено дърво
    void print() const
    { pr(root);
```

Оригинал: стр. 190

C190-01 · Копиране: code.html

```
      cout << endl;
    }
    // включва връх в двоично наредено дърво
    // със запазване на наредеността
    void addNode(const T& x)
    { add(root, x);
    }
    // изтрива връх от двоично наредено дърво
    // със запазване на наредеността
    void deleteNode(const T&);
    // помощна член-функция, която намира минималния
    // елемент на дадено двоично наредено дърво и
    // двоично наредено дърво, получено от даденото
    // като от него се изключи минималният елемент
    void minTree(T&, binOrdTree&) const;
  private:
    node_bin<T>* root; // указател към корена
    // създава копие на указаното двоично наредено дърво
    // на друго място в паметта, сочено от указателя this
    void copyTree(const binOrdTree&);
    // помощна член-функция за копиране
    void copy(node_bin<T>*&, node_bin<T>* const&) const;
    // изтрива двоично наредено дърво
    void deleteTree(node_bin<T>*&) const;
    // извежда двоично наредено дърво, сочено
    // от указателя, зададен чрез параметъра
    void pr(const node_bin<T>*) const;
    // включва указания елемент в двоично наредената
    // дърво, зададено чрез първия параметър
    void add(node_bin<T>*&, const T&) const;
};
```

Каноничното представяне е реализирано като при обикновените двоични дървета. Конструкторът по подразбиране създава празно двоично наредено дърво.

C190-02 · Копиране: code.html

```
template <typename T>
binOrdTree<T>::binOrdTree()
{ root = NULL;
}

template <typename T>
```

C191-01 · Копиране: code.html
```
binOrdTree<T>::~binOrdTree()
{ deleteTree(root);
}

template <typename T>
binOrdTree<T>::binOrdTree(const binOrdTree<T>& r)
{ copyTree(r);
}

template <typename T>
binOrdTree<T>& binOrdTree<T>::operator=(const binOrdTree<T>& r)
{ if (this != &r)
    { deleteTree(root);
      copyTree(r);
    }
  return *this;
}
```

Следващите член-функции реализират създаване на копие на двоично наредено дърво и изтриване на двоично наредено дърво. С цел улесняване на реализацията, *copyTree* използва член-функцията *copy*, която копира двоично нареденото дърво, сочено от указателя *r* на друго място в паметта, сочено от указателя *pos*.

C191-02 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::copyTree(const binOrdTree<T>& r)
{ copy(root, r.root);
}

template <typename T>
void binOrdTree<T>::copy(node_bin<T>*& pos,
                         node_bin<T>* const& r) const
{ pos = NULL;
  if (r)
    { pos = new node_bin<T>;
      assert(pos != NULL);
      pos->inf = r->inf;
      copy(pos->left, r->left);
      copy(pos->right, r->right);
    }
}
```

C192-01 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::deleteTree(node_bin<T>*& p) const
{ if (p)
    { deleteTree(p->left);
      deleteTree(p->right);
      delete p;
      p = NULL;
    }
}
```

Член-функциите *empty, rootTree, leftTree, rightTree, pr* и *print* също са като тези на обикновените двоични дървета. Член-функциите *rootTree, leftTree* и *rightTree* намират корена, лявото и дясното поддърво на непразно двоично наредено дърво. Член-функцията *print* извежда в линеен формат двоично наредено дърво, като го обхожда в последователността ЛКД.

C192-02 · Копиране: code.html
```
template <typename T>
bool binOrdTree<T>::empty() const
{ return root == NULL;
}

template <typename T>
T binOrdTree<T>::rootTree() const
{ return root->inf;
}

template <typename T>
binOrdTree<T> binOrdTree<T>::leftTree() const
{ binOrdTree<T> t;
  copy(t.root, root->left);
  return t;
}

template <typename T>
binOrdTree<T> binOrdTree<T>::rightTree() const
{ binOrdTree<T> t;
  copy(t.root, root->right);
  return t;
}
template <typename T>
void binOrdTree<T>::pr(const node_bin<T>* p) const
```

Оригинал: стр. 193

C193-01 · Копиране: code.html
```
{ if (p)
    { pr(p->left);
      cout << p->inf << " ";
      pr(p->right);
    }
}
```

Член-функцията *add* е помощна. Използва се за реализиране на член-функцията *addNode* на шаблона, чрез която се включва елемент в двоично наредено дърво по описания по-горе начин.

C193-02 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::add(node_bin<T>*& p, const T& x) const
{ if (!p)
    { p = new node_bin<T>;
      assert(p != NULL);
      p->inf = x;
      p->left = NULL;
      p->right = NULL;
    }
  else
    if (x < p->inf) add(p->left, x);
    else add(p->right, x);
}
```

Създаването на непразно двоично наредено дърво се осъществява чрез член-функцията *create*. Тя реализира въвеждане на елементи и включването им чрез член-функцията *addNode* в двоично наредено дърво.

C193-03 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::create()
{ root = NULL;
  T x; char c;
  do
    { cout << "> "; cin >> x;
      addNode(x);
      cout << "next elem  y/n? "; cin >> c;
    } while (c == 'y');
}
```

Оригинал: стр. 194

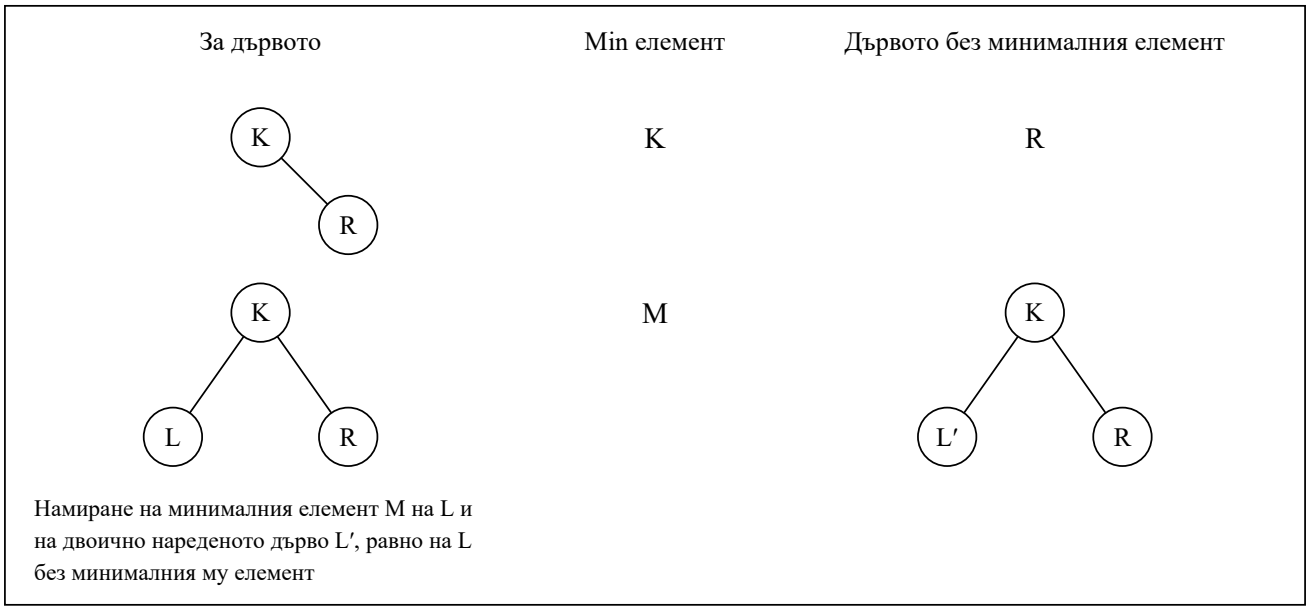
Нека *l* и *r* са двоично наредени дървета, а *x* е по-голямо от всички върхове на *l* и е по-малко от всички върхове на *r*. Реализацията на член-функцията *create3* е аналогична на тази при ненаредените двоични дървета.

C194-01 · Копиране: code.html

```
template <typename T>
void binOrdTree<T>::create3(const T& x, const binOrdTree<T>& l,
                                        const binOrdTree<T>& r)
{ if (root)deleteTree(root);
  root = new node_bin<T>;
  assert(root != NULL);
  root->inf = x;
  copy(root->left, l.root);
  copy(root->right, r.root);
}
```

Член-функцията *minTree* намира минималния елемент на подразбиращото се (соченото от указателя *this*) двоично наредено дърво, в случай че то не е празно и двоично наредено дърво, равно на подразбиращото се двоично наредено дърво без минималния му елемент. Реализира следния алгоритъм:

• Ако лявото поддърво на подразбиращото се двоично наредената дърво е празно, минималният му елемент е коренът, а дървото без минималния елемент е дясното му поддърво.
• Ако лявото му поддърво е непразно, на това двоично наредено дърво се намира минималния елемент и дървото без минималния елемент. След това се конструира двоично наредено дърво от корена, лявото поддърво без минималния му елемент и дясното поддърво на подразбиращото се дърво.
Тези действия могат да се илюстрират графично по следния начин:



Намиране на минималния елемент M на L и на двоично наредената дърво L′, равно на L без минималния му елемент

Оригинал: стр. 195

C195-01 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::minTree(T& x, binOrdTree<T>& mint) const
{ T a = rootTree();
  if (leftTree().empty())
    { x = a;
      mint = rightTree();
    }
  else
    { binOrdTree<T> t, t1 = rightTree();
      leftTree().minTree(x, t);
      mint.create3(a, t, t1);
    }
}
```

Член-функцията *deleteNode* изтрива връх на подразбиращото се двоично наредено дърво. Реализира описания на фиг. 3.8 алгоритъм.

C195-02 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::deleteNode(const T& x)
{ if (root)
    { // двоично наредената дърво не е празно
      T a = rootTree();
      binOrdTree<T> t1, t2;
      if (a == x && leftTree().empty()) *this = rightTree();
      else
        if (a == x && rightTree().empty()) *this = leftTree();
        else
          if (a == x)
            { T c; t1 = leftTree();
              rightTree().minTree(c, t2);
              create3(c, t1, t2);
            }
          else
            if (x < a)
              { t2 = rightTree();
                *this = leftTree();
                deleteNode(x);
                t1 = *this;
                create3(a, t1, t2);
              }
            else
```

Оригинал: стр. 196

C196-01 · Копиране: code.html
```
               { t1 = leftTree();
                 *this = rightTree();
                 deleteNode(x);
                 t2 = *this;
                 create3(a, t1, t2);
               }
    }
}
```

Записваме този шаблон във файла *binOrdTree.cpp* и ще го използваме в следващите задачи.

Задача 38. Да се напише програма, която въвежда редица от цели числа, сортира ги във възходящ ред, след което изключва указани върхове, като запазва наредеността на елементите.

C196-02 · Копиране: code.html
```
// Program Zad38.cpp
#include <iostream>
#include <cassert>
#include "binOrdTree.cpp"
using namespace std;

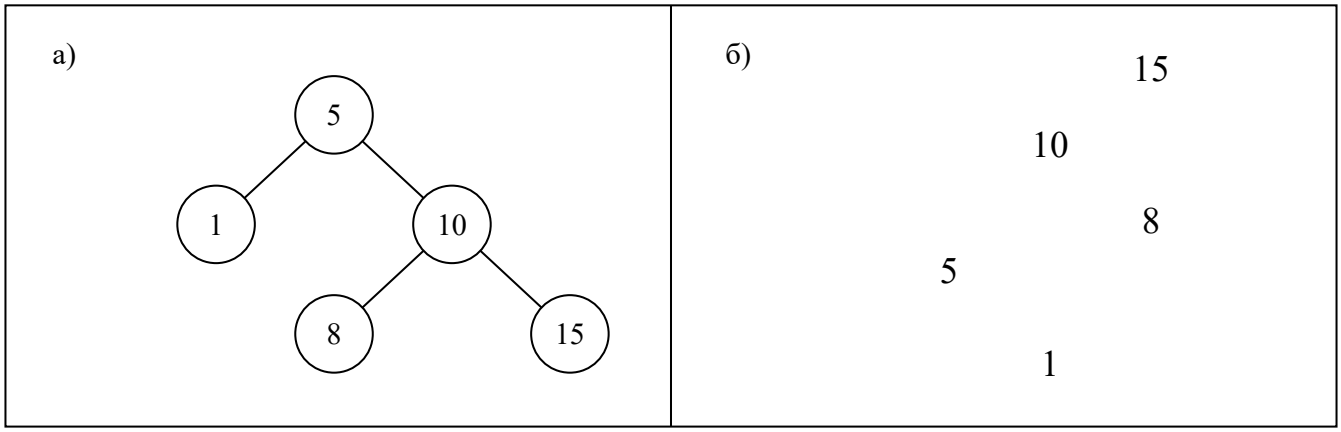
typedef binOrdTree<int> intTree;

int main()
{ intTree t;
  t.create(); // създаване на двоично наредено дърво,
              // съдържащо елементите на въведената редица
  t.print();  // извежда въведените елементи във възходящ ред
  int x;
  char c;
  // изключване на указани върхове
  do
    { cout << "x: ";
      cin >> x;
      t.deleteNode(x);
      t.print();
      cout << "next: y/n: ";
      cin >> c;
    } while (c == 'y');
  t.print();
  return 0;
}
```

Оригинал: стр. 197

В този и в предходния шаблон извеждането на двоично дърво е линейно. За предходната задача този вид е подходящ, но има случаи, при които това не е така. Ще добавим в *private* секцията на шаблона на класа *binOrdTree* процедурата за извеждане *pr1*, а в *public* секцията му – член-функцията *print_tree*, които извеждат двоично наредено дърво, завъртяно на 90 градуса по посока обратна на часовниковата стрелка.

Пример. Двоично наредената дърво от а) ще бъде изведено във вида от б).



C197-01 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::print_tree() const
{ pr1(root, 0);
  cout << endl;
}
```

и

C197-02 · Копиране: code.html
```
template <typename T>
void binOrdTree<T>::pr1(const node_bin<T>* p, int n) const
{ if (p)
    { n = n + 5;
      pr1(p->right, n);
      cout << setw(n) << p->inf << endl;
      pr1(p->left, n);
    }
}
```

Забележка. Използването на манипулатора *setw* налага включването на заглавния файл *iomanip*.

Оригинал: стр. 198

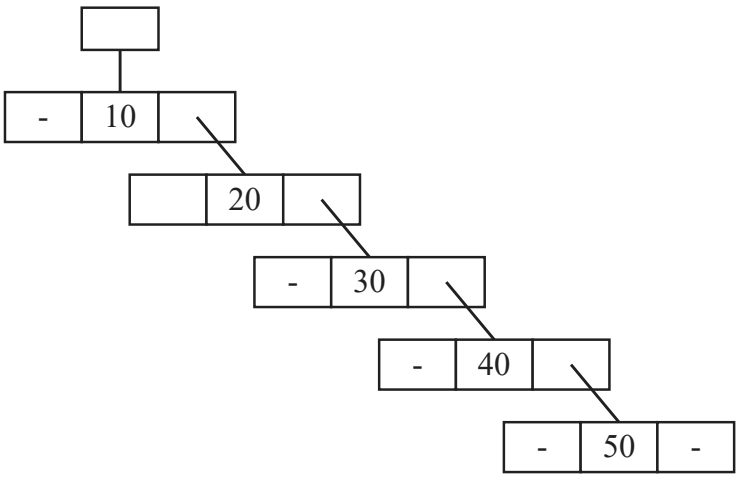
Задача 39. Да се напише шаблон на булева функция, чрез който се проверява дали елемент се съдържа в двоично наредено дърво.

C198-01 · Копиране: code.html
```
template <typename T>
bool member(const T& a, const binOrdTree<T>& t)
{ if (t.empty()) return false;
  if (a == t.rootTree()) return true;
  if (a < t.rootTree()) return member(a, t.leftTree());
  return member(a, t.rightTree());
}
```

Сложността на тази функция е $O(log_2 n)$, ако двоично наредената дърво е балансирано (т. 3.4).

3.4. БАЛАНСИРАНИ И ИДЕАЛНО БАЛАНСИРАНИ ДВОИЧНО НАРЕДЕНИ ДЪРВЕТА

Ако елементите, които се включват в двоично наредено дърво, са наредени в нарастващ или намаляващ ред, се получава двоично наредено дърво с линейна структура. Например, ако към празното двоично наредено дърво последователно се включат елементите: 10, 20, 30, 40, 50, се получава:



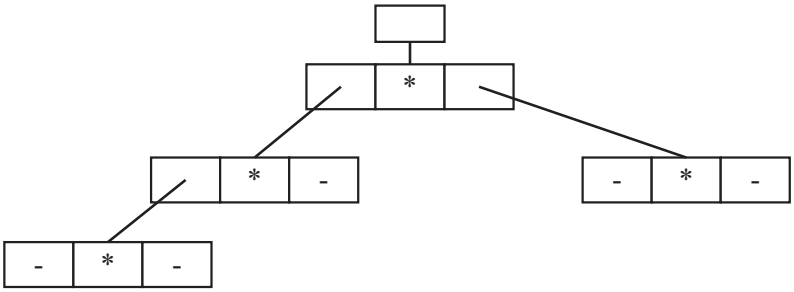
Търсенето на елемент в такава „изродено“ двоично наредено дърво не е по-ефективно от това в линейните списъци. Двоични дървета от този вид се наричат *силно небалансирани*.

Потвърдено по хартиения оригинал: лявото поле при 20 е празно.

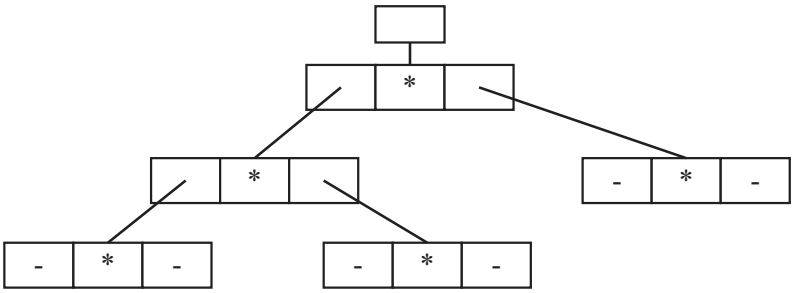
Оригинал: стр. 199

Дефиниция. Двоично дърво се нарича *идеално (перфектно) балансирано*, ако всеки негов връх има ляво и дясно поддърво, в които броят на възлите се различава най-много с 1.

Пример. Двоично дърво със следната структура



е идеално балансирано, но двоично дърво със структура от вида



не е идеално балансирано, тъй като лявото му поддърво има три върха, а дясното му – 1 (разликата е по-голяма от 1). Това двоично дърво е двоично дърво с балансирана височина.

Дефиниция. Двоично дърво се нарича *двоично дърво с балансирана височина* или само *балансирано дърво*, ако за всеки негов връх височините (дълбочините) на лявото и дясното му поддървета се различават най-много с 1. Този вид двоични дървета се наричат също *AVL дървета* (в чест на *Adelson-Velskii u Landis*).

От дефинициите и примера се вижда, че всяко идеално балансирано двоично дърво е двоично дърво с балансирана височина, но обратното не е вярно. Обикновено се изисква тези два вида двоични дървета да бъдат и наредени.

В останалата част на книгата за балансираните и идеално балансираните двоични дървета ще е в сила, че са наредени. Алгоритмът за включване на елемент в двоично наредено дърво, не създава идеално балансирано двоично наредено дърво, нито балансирано двоично наредено дърво. Алгоритмът за

Оригинал: стр. 200

изключване на елемент също не запазва идеалната балансираност или само балансираността на дървото.

Има ефективен алгоритъм за създаване на балансирани двоично наредени дървета, който при включване и изключване на елемент запазва балансираността на двоично нареденото дърво. За идеално балансираното двоично наредено дърво такъв не е известен.

Идеално балансирано двоично наредено дърво

Съществува прост алгоритъм за създаване на идеално балансирано двоично наредено дърво при следните ограничения:

– елементите, които ще се включват към празното двоично наредено дърво, са подредени в нарастващ ред;
– предварително е известен броят на върховете на дървото.

Към шаблона на класа *binOrdTree* ще добавим още един конструктор, който създава идеално балансирано двоично наредено дърво с *n* елемента.

C200-01 · Копиране: code.html

```
template <typename T>
binOrdTree<T>::binOrdTree(int n)
{ root = NULL;
  if (n > 0)
    { int nLeft = (n-1)/2,
          nRight = n-nLeft-1;
      binOrdTree<T> t1(nLeft);
      T x; cin >> x;
      binOrdTree<T> t2(nRight);
      create3(x, t1, t2);
    }
}
```

Като използва *n*, алгоритмът намира броя на върховете *nLeft* и *nRight* на поддърветата на дървото, където

`n = nLeft + nRight + 1 и |nLeft-nRight| ≤ 1.`

Създава тройната кутия и я запълва в последователността: конструиране на лявото поддърво с *nLeft* върха, въвеждане на корена и конструиране на дясното поддърво с *nRight* върха. Конструирането на лявото и дясното поддърво се осъществява чрез рекурсивно обръщение към конструктора на идеално балансираното двоично наредено дърво.

Оригинал: стр. 201

Балансирано двоично наредено дърво

Ще предложим алгоритъм за създаване на балансирано двоично наредено дърво, който при включване и изключване на елемент запазва балансираността на двоично нареденото дърво. За реализацията на алгоритъма отначало ще разгледаме две помощни операции: *завъртане наляво* и *завъртане надясно*. След това ще реализираме операциите включване и изключване на елемент.

За удобство при реализирането с всеки връх на дървото ще свържем още един параметър *balance*, означаващ коефициент на балансиране. Той е от тип *int* и е равен на разликата на височините на дясното и лявото поддърво на съответния връх.

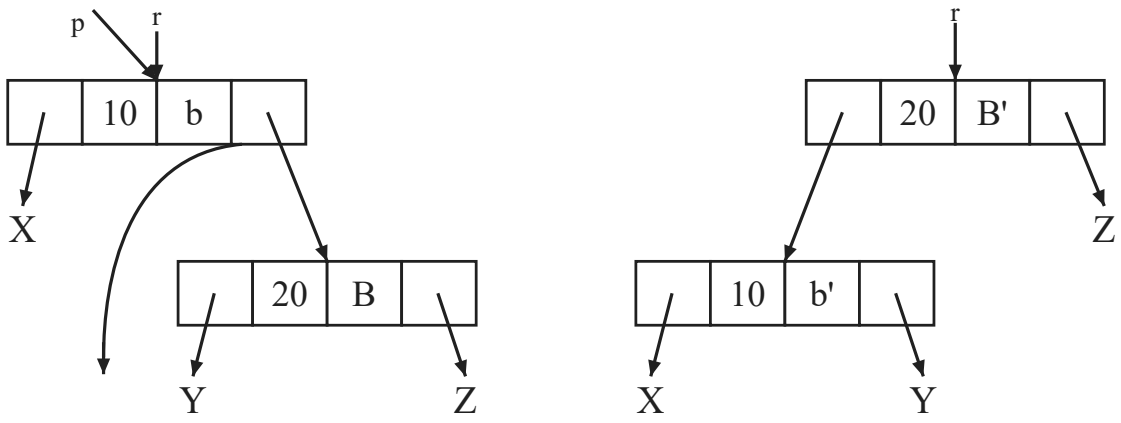
Ако *r* е указател към връх на балансирано двоично наредено дърво,

`r->balance = h(r->right) - h(r->left),`

където *h(X)* е височината на двоичното дърво *X*. Тъй като дървото е балансирано, *r->balance* може да приеме стойностите -1, 0 и 1. В процеса на вмъкване и изключване на елемент в/от балансираното двоично наредено дърво с връх, сочен от *r*, стойността на *r->balance* може да получи и стойностите -2 и 2. Когато се получат тези стойности, двоичното наредено дърво е загубило балансираността си. За да възстанови двоичното наредено дърво свойството си балансираност, са необходими едно или две завъртания.

Завъртане наляво

Операцията завъртане наляво е показана по-долу.



преди завъртането след завъртането

Въпреки че след операцията завъртане двоичното наредено дърво се променя, поддърветата му *X, Y и Z* остават непроменени. Последното означава, че и височините им не се променят.
С *b, B, b′ и B′* са означени коефициентите на балансиране в означените върхове. За да се реализира завъртането, е необходимо по зададени *B* и *b* да се намерят *B′* и *b′*. От илюстрацията по-горе се вижда, че:

$$b = \max\{ h(Y), h(Z) \} + 1 - h(X) = \begin{cases} h(Y) + 1 - h(X), & h(Y) \geq h(Z) \\ h(Z) + 1 - h(X), & h(Y) < h(Z) \end{cases}$$

$B = h(Z) - h(Y)$
$b' = h(Y) - h(X)$

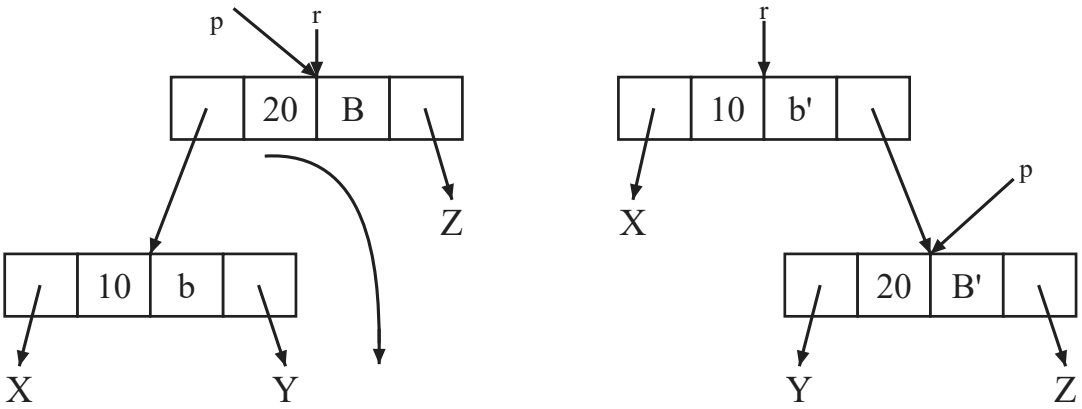
$$B' = h(Z) - \max\{ h(Y), h(X) \} - 1 = \begin{cases} h(Z) - h(Y) - 1, & h(Y) \geq h(X) \\ h(Z) - h(X) - 1, & h(Y) < h(X) \end{cases}$$

От горните зависимости следват:

$$b' = \begin{cases} b - 1, & h(Y) \geq h(Z) \\ b - B - 1, & h(Y) < h(Z) \end{cases} \qquad B' = \begin{cases} B - 1, & h(Y) \geq h(X) \\ B + b' - 1, & h(Y) < h(X) \end{cases}$$

Завъртане надясно

Завъртането надясно е показано по-долу.



С *b, B, b′ и B′* са означени коефициентите на балансиране в означените върхове. За да се реализира завъртането, е необходимо по зададени *B* и *b* да се намерят коефициентите *B′* и *b′*.

От илюстрираното по-горе се вижда, че:

$b = h(Y) - h(X)$

$$B = h(Z) - \max\{ h(Y), h(X) \} - 1 = \begin{cases} h(Z) - h(Y) - 1, & h(Y) \geq h(X) \\ h(Z) - h(X) - 1, & h(Y) < h(X) \end{cases}$$

$B' = h(Z) - h(Y)$

$$b' = \max\{ h(Y), h(Z) \} + 1 - h(X) = \begin{cases} h(Y) + 1 - h(X), & h(Y) \geq h(Z) \\ h(Z) + 1 - h(X), & h(Y) < h(Z) \end{cases}$$

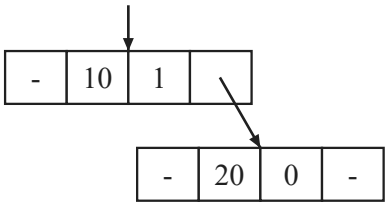
От горните зависимости следват:

$$B' = \begin{cases} B + 1, & h(Y) \geq h(X) \\ B - b + 1, & h(Y) < h(X) \end{cases} \qquad b' = \begin{cases} b + 1, & h(Y) \geq h(Z) \\ B' + b + 1, & h(Y) < h(Z) \end{cases}$$

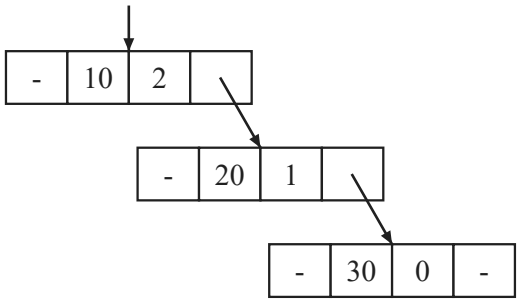
Включване на елемент

Операцията включване на елемент в балансирано двоично наредено дърво се осъществява по аналогичен начин на включването на елемент в двоично наредено дърво. Разликата е, че при всяко включване, което променя балансираността на дървото трябва да се изпълнят едно или две завъртания с цел възстановяване на балансираността. Чрез примери ще илюстрираме необходимостта от завъртане.

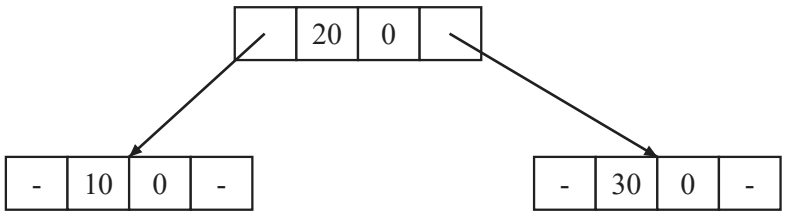
Нека в балансираното двоично наредено дърво



се включи елементът 30. Получава се небалансираното двоично наредено дърво

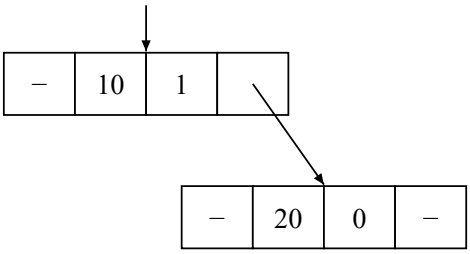


За да се възстанови балансираността му, е необходимо да се извърши завъртане наляво. Ще се получи балансираното двоично наредено дърво

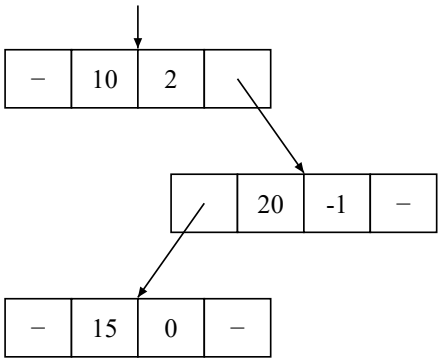


Забелязваме, че след включването на елемента 30, коефициентите на балансиране на корена и десният му наследник са 2 и 1.

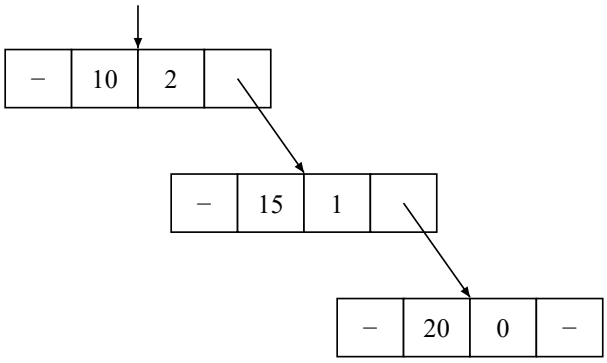
Аналогична ситуация възниква, ако вместо 2 и 1, коефициентите на балансиране са -2 и -1. В този случай ще е нужно завъртане надясно. Ако в балансираното двоично наредено дърво



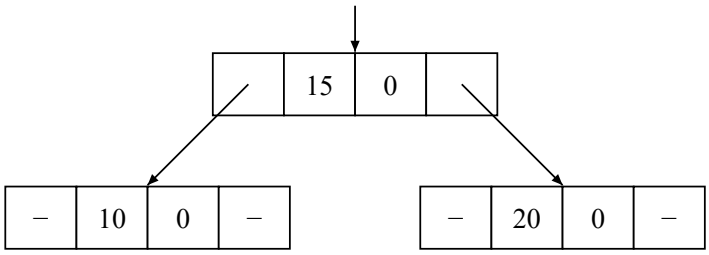
се включи елементът 15, се получава небалансираното двоично наредено дърво



За да се възстанови балансираността му, е необходимо да се извърши отначало завъртане надясно при върха 20. Ще се получи небалансираното двоично наредено дърво



След още едно завъртане, но наляво, се получава



Изключване на елемент

Операцията изключване на елемент в балансирано двоично наредено дърво се осъществява по аналогичен начин на изключването на елемент от

обикновено двоично наредено дърво. Разликата е, че при всяко изключване, което променя балансираността на дървото, трябва да се изпълнят едно или две завъртания с цел възстановяване на балансираността. Освен това изключването на елемент, който се намира в корена на балансирано двоично наредено дърво с две непразни поддървета, ще реализираме, като се разменят стойностите на максималния елемент на ЛПД и корена, след което се изключи елементът, който вече е корен на дърво с празно дясно поддърво.

Създаване на балансирано двоично наредено дърво

Осъществява се, като в празното балансирано двоично наредено дърво последователно се включват елементи чрез операцията включване, описана по-горе. Тъй като последната запазва нареденост та и балансираността на дървото, полученото двоично дърво е балансирано двоично наредено.

Реализация на балансирано двоично наредено дърво

В шаблона на класа *AVL* са реализирани: каноничното представяне; булевата член-функция *empty*, установяваща дали балансирано двоично наредено дърво е празно; селекторите:

C205-01 · Копиране: code.html
```
T rootAVL() const;       // намира корена
int balanceAVL() const;  // намира коефициента
                         // на балансираност
AVL leftAVL() const;     // намира лявото поддърво
AVL rightAVL() const;    // намира дясното поддърво
```

мутаторите:

C205-02 · Копиране: code.html
```
void insNode(const T&x);  // включва елемента x
void delNode(const T&x);  // изключва елемента x
```

член-функцията *print_tree*, която извежда в графичен вид, завъртяно на 90 градуса в посока обратна на часовниковата стрелка, подразбиращото се балансирано двоично наредено дърво и член-функцията *createAVL*, която създава непразно балансирано двоично наредено дърво. Член-функцията *void insNode(const T& x)* на шаблона на класа реализира операцията включване на елемента *x* в подразбиращото се балансирано двоично наредено дърво, а член-функцията *void delNode(const T& x)* на шаблона на класа реализира операцията изключване на елемента *x* от подразбиращото се балансирано двоично наредено дърво. Двете член-функции използват съответно помощните член-функции:

C205-03 · Копиране: code.html
```
int insertElem(node<T>*& r, const T&x);
int deleteElem(node<T>*& r, const T&x);
```

които са включени в *private* секцията на шаблона на класа *AVL*. Първата член-функция включва, а втората изключва елемента *x* в/от балансираното двоично наредено дърво, зададено чрез указателя към корена му *r*. Член-функцията *insertElem* връща увеличението на височината (0 или 1) след вмъкването на елемента *x* в дървото с корен, сочен от указателя *r*. Член-функцията *deleteElem* връща намалението на височината след изтриването на елемента *x* от дървото с корен, сочен от указателя *r*. Ако включване или изключване не е реализирано, член-функциите връщат 0. Операциите завъртане наляво и завъртане надясно са реализирани чрез член-функциите:

C206-01 · Копиране: code.html
```
void leftRotate(node<T>*&);   // завъртане наляво
void rightRotate(node<T>*&);  // завъртане надясно
```

от *private* секцията на шаблона на класа.

Следва реализацията на шаблона на класа *AVL*.

C206-02 · Копиране: code.html
```
template <class T>
struct node
{ T inf;         // връх
  int balance;   // коефициент на балансираност
  node *left,    // указател към ЛПД
       *right;   // указател към ДПД
};

template <class T>
class AVL
{ public:
    // канонично представяне
    AVL();
    ~AVL();
    AVL(const AVL&);
    AVL& operator=(const AVL&);
    // проверява дали дървото е празно
    bool empty() const;
    // селектори
    T rootAVL() const;
    int balanceAVL() const;
    AVL leftAVL() const;
    AVL rightAVL() const;
    // включва елемента x в подразбиращото се балансирано ДНД
```

Оригинал: стр. 207

C207-01 · Копиране: code.html

```
    void insNode(const T&x)
    { insertElem(root, x);
    }
    // изключва елемента x от подразбиращото се балансирано ДНД
    void delNode(const T&x)
    { deleteElem(root, x);
    }
    // извежда подразбиращото се балансирано ДНД, завъртяно на
    // 90 градуса в посока обратна на часовниковата стрелка
    void print_tree() const
    { pr1(root, 0);
      cout << endl;
    }
    // създава непразно балансирано двоично
    // наредено дърво чрез вмъкване
    void createAVL();
  private:
    node<T>* root;             // указател към корена
    // помощни член-функции
    void deleteAVL(node<T>* &) const;
    void copy(node<T>*&, node<T>* const&) const;
    void copyAVL(AVL<T> const&);
    void pr1(const node<T>*, int) const;
    // реализира завъртане наляво
    void leftRotate(node<T>*&);
    // реализира завъртане надясно
    void rightRotate(node<T>*&);
    // включва даден елемент в AVL-дърво с указан корен
    int insertElem(node<T>*&, const T&);
    // изключва даден елемент от AVL-дърво с указан корен
    int deleteElem(node<T>*&, const T&);
};

template <class T>
AVL<T>::AVL()
{ root = NULL;
}

template <class T>
AVL<T>::~AVL()
```

C208-01 · Копиране: code.html

```
{ deleteAVL(root);
}
template <class T>
AVL<T>::AVL(const AVL<T>& r)
{ copyAVL(r);
}

template <class T>
AVL<T>& AVL<T>::operator=(const AVL<T>& r)
{ if (this != &r)
    { deleteAVL(root);
      copyAVL(r);
    }
  return *this;
}

template <class T>
void AVL<T>::copyAVL(const AVL<T>& r)
{ copy(root, r.root);
}

template <class T>
void AVL<T>::copy(node<T>*& pos, node<T>* const& r) const
{ pos = NULL;
  if (r)
    { pos = new node<T>;
      assert(pos != NULL);
      pos->inf = r->inf;
      copy(pos->left, r->left);
      copy(pos->right, r->right);
    }
}

template <class T>
void AVL<T>::deleteAVL(node<T>*& p) const
{ if (p)
    { deleteAVL(p->left);
      deleteAVL(p->right);
      delete p;
      p = NULL;
```

C209-01 · Копиране: code.html

```
    }
}
template <class T>
bool AVL<T>::empty() const
{ return root == NULL;
}

template <class T>
T AVL<T>::rootAVL() const
{ return root->inf;
}

template <class T>
int AVL<T>::balanceAVL() const
{ return root->balance;
}

template <class T>
AVL<T> AVL<T>::leftAVL() const
{ AVL<T> t;
  copy(t.root, root->left);
  return t;
}

template <class T>
AVL<T> AVL<T>::rightAVL() const
{ AVL<T> t;
  copy(t.root, root->right);
  return t;
}

template <class T>
void AVL<T>::createAVL()
{ char ch = 'y';
  while(ch == 'y')
    { T x; cout << "x= "; cin >> x;
      insNode(x);
      cout << "ch = y/n? ";
      cin >> ch;
    }
```

Оригинал: стр. 210

C210-01 · Копиране: code.html

```
}

template <class T>
void AVL<T>::leftRotate(node<T>*& r)
{ // промяна на структурата на дървото
  node<T>* p = r;
  r = r->right;
  p->right = r->left;
  r->left = p;
  // намиране и промяна на коефициентите на балансиране
  p->balance = p->balance - 1;
  if (r->balance > 0) p->balance = p->balance -
                                   r->balance;
  r->balance = r->balance - 1;
  if (p->balance < 0) r->balance = r->balance +
                                   p->balance;
}

template <class T>
void AVL<T>::rightRotate(node<T>*& r)
{ // промяна на структурата на дървото
  node<T>* p = r;
  r = r->left;
  p->left = r->right;
  r->right = p;
  // намиране и промяна на коефициентите на балансиране
  p->balance = p->balance + 1;
  if (r->balance < 0) p->balance = p->balance - r->balance;
  r->balance = r->balance + 1;
  if (p->balance > 0) r->balance = r->balance + p->balance;
}

template <class T>
int AVL<T>::insertElem(node<T>*& r, const T&x)
{ int h = 0; // ще съдържа стойността, която функцията ще върне
             // равна е на 0 или 1 и означава увеличението на
             // височината след вмъкването на x в дървото
             // с корен, сочен от r
  if (!r)
    { // AVL дървото е празно
```

Оригинал: стр. 211

C211-01 · Копиране: code.html

```
      r = new node<T>;
      assert(r != NULL);
      r->inf = x;
      r->balance = 0;
      r->left = NULL;
      r->right = NULL;
      h = 1;
    }
  else
  if (x > r->inf)
    { // включване на x в ДПД на AVL дървото, сочено от r
      if (insertElem(r->right, x))
        { r->balance++;      // защото височината на
                             // ДПД се е увеличила с 1
          if (r->balance == 1) h = 1; else
            if (r->balance == 2)
              { if (r->right->balance == -1)
                  rightRotate(r->right);
                leftRotate(r);
              }
        }
    }
  else  // включване на x в ЛПД на AVL дървото, сочено от r
  if (x < r->inf)
    { if (insertElem(r->left, x))
        { r->balance--;      // защото височината на
                             // ЛПД се е увеличила с 1
          if (r->balance == -1) h = 1; else
            if (r->balance == -2)
              { if (r->left->balance == 1)
                  leftRotate(r->left);
                rightRotate(r);
              }
        }
    }
  return h;
}

template <class T>
int AVL<T>::deleteElem(node<T>*& r, const T&x)
```

Оригинал: стр. 212

C212-01 · Копиране: code.html

```
{ node<T>** p, *temp;
  int h = 0; // ще съдържа стойността, която функцията ще върне
  if (!r) return 0;       // AVL дървото е празно
  // AVL дървото не е празно
  if (x < r->inf)
    { // изключване на x от ЛПД на AVL дървото
      if (deleteElem(r->left, x))
        { r->balance++;
          if (r->balance == 0) h = 1; else
            if (r->balance == 2)
              { if (r->right->balance == -1)
                  rightRotate(r->right);
                leftRotate(r);
                if (r->balance == 0) h = 1;
              }
        }
    }
  else
    if (x > r->inf)
      { // изключване на x от ДПД на AVL дървото
        if (deleteElem(r->right, x))
          { r->balance--;
            if (r->balance == 0) h = 1; else
              if (r->balance == -2)
                { if (r->left->balance == 1)
                    leftRotate(r->left);
                  rightRotate(r);
                  if (r->balance == 0) h = 1;
                }
          }
      }
    else  // x съвпада с корена на дървото
      { if (!r->right)       // ДПД е празно
          { temp = r; r = r->left;
            delete temp;
            return 1;
          }
        else
        if (!r->left)        // ЛПД е празно
          { temp = r; r = r->right;
```

Оригинал: стр. 213

C213-01 · Копиране: code.html

```
            delete temp;
            return 1;
          }
        else  // x съвпада с корена и лявото, и дясното
              // поддървета не са празни
          { // размяна на стойностите на корена и най-големия
            // елемент на ЛПД на AVL дървото
            p = &r->left;
            while((*p)->right) p = &(*p)->right;
            r->inf = (*p)->inf;
            (*p)->inf = x;
            // изключване на x от ЛПД
            if (deleteElem(r->left, x))
              { r->balance++;
                if (r->balance == 0) h = 1; else
                  if (r->balance == 2)
                    { if (r->right->balance == -1)
                        rightRotate(r->right);
                      leftRotate(r);
                      if (r->balance == 0) h = 1;
                    }
              }
          }
      }
  return h;
}

template <class T>
void AVL<T>::pr1(const node<T>* p, int n) const
{ if (p)
    { n = n+7;
      pr1(p->right, n);
      cout << setw(n) << p->inf << " " << p->balance << endl;
      pr1(p->left, n);
    }
}
```

Записваме реализацията на шаблона на класа *AVL* във файл с име *AVL.cpp* и ще я използваме в следващата задача.

Оригинал: стр. 214

Задача 40. Да се напише програма, която създава *AVL* дърво от цели числа чрез изпълнение на операцията включване. След това от полученото *AVL* дърво изключва указани върхове като запазва балансираността на дървото. След всяко изпълнение на операцията изключване, да се изведе на екрана полученото *AVL* дърво.

C214-01 · Копиране: code.html

```
// Program Zad40.cpp
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
#include "AVL.cpp"

typedef AVL<int> intAVL;

int main()
{ // създаване на празно AVL дърво t
  intAVL t;
  // включване на елементи в дървото
  t.createAVL();
  // извеждане на създаденото AVL дърво
  t.print_tree();
  int x;
  char c;
  do
    { cout << "x: ";
      cin >> x;
      t.delNode(x);       // изключва върха x
      t.print_tree();     // извежда AVL дървото
      cout << "next: y/n: ";
      cin >> c;
    } while (c == 'y');
  t.print_tree();
  return 0;
}
```

Примерно изпълнение:

След въвеждане на елементите: 10, 5, 1, 8, 15, 20, 13 се получава *AVL* дървото

Оригинал: стр. 215

C215-01 · Копиране: code.html

```
               20 0
       15 0
              13 0
10 0
               8 0
        5 0
               1 0

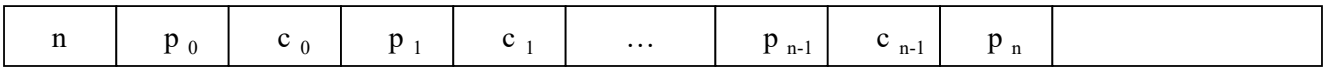
x: 10                    // изключване на 10
               20 0
       15 0
              13 0
8 0
        5 -1
               1 0
```

Дърветата, които използвахме досега, се реализират в оперативната памет. За редица приложения се налага да се съхраняват дървета във външната памет. За тази цел могат да се използват *B*-дървета.

3.5. B-ДЪРВЕТА

3.5.1. Дефиниция

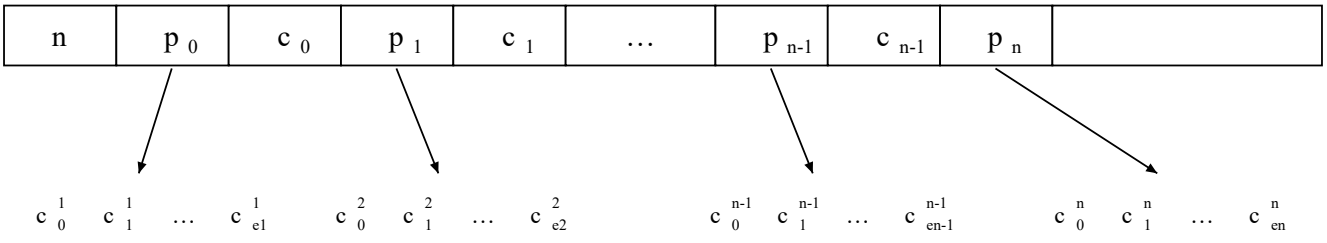
При обработване на данни на линейни структури, основната част от времето се отделя за търсене на (достъп до) данните. Четенето и обработването на данните почти не зависи от дължината им. *B*-дърветата са наредени дървета, които не са двоични. Във възлите им може да има повече от един елемент (ключ, данна).
Нека *m* е положително цяло число. *B-дърво от ред m* е наредено дърво, притежаващо следните свойства:
– Ако възел, който не е листо, съдържа *n* данни, то той има точно *n+1* наследника.
– Коренът е или листо, или има между 2 и *m* наследника.
– Всеки вътрешен възел (възел, различен от корена и листо) има между цялата стойност на *(m+1)/2* и *m* наследника или между цялата стойност на *(m-1)/2* и *m-1* елемента.
– Всяко листо има между цялата стойност на *(m-1)/2* и *m-1* елемента и 0 наследника.
– Всички листа имат една и съща дълбочина.
Структурата на един възел с *n* данни на *B*-дърво от ред *m* е следната:



Оригинал: стр. 216

където $c_0, c_1, ..., c_{n-1}$ са данните във възела, а $p_0, p_1, ..., p_{n-1}, p_n$ са указатели към поддърветата, които са наследници на възела. Наричат се *връзки*. За данните и връзките са в сила:

1. Ключовете $c_0, c_1, ..., c_{n-1}$ са наредени в нарастващ ред, т.е. в сила е
$c_0 < c_1 < \ldots < c_{n-1}$
2. Ако възелът не е листо, всеки от неговите *n+1* указателя p_i *(i = 0, 1, ..., n)* сочи към поддърво. Всички данни, в наследника, сочен от указателя p_i *(i = 1, 2,..., n)* са по-големи от c_{i-1}. Също всички данни в наследника, сочен от указателя p_i *(i = 0, 1, ..., n-1)* са по-малки от c_i, т.е. ако

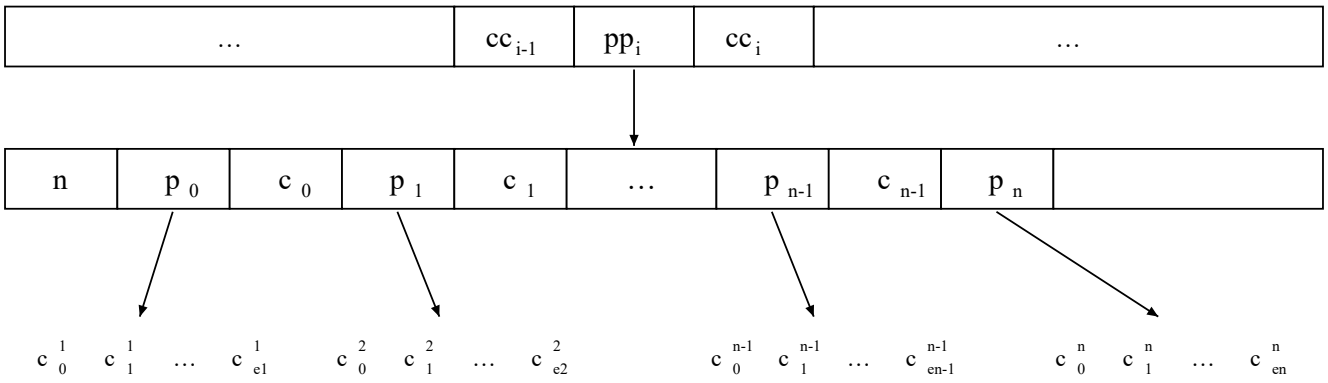


то

$c_0^1 < c_1^1 < \ldots < c_{e1}^1 < c_0 < c_0^2 < c_1^2 < \ldots < c_{e2}^2 < c_1 < \ldots <$

$c_0^{n-1} < c_1^{n-1} < \ldots < c_{en-1}^{n-1} < c_{n-1} < c_0^n < c_1^n < \ldots < c_{en}^n$

Ако възелът е вътрешен и



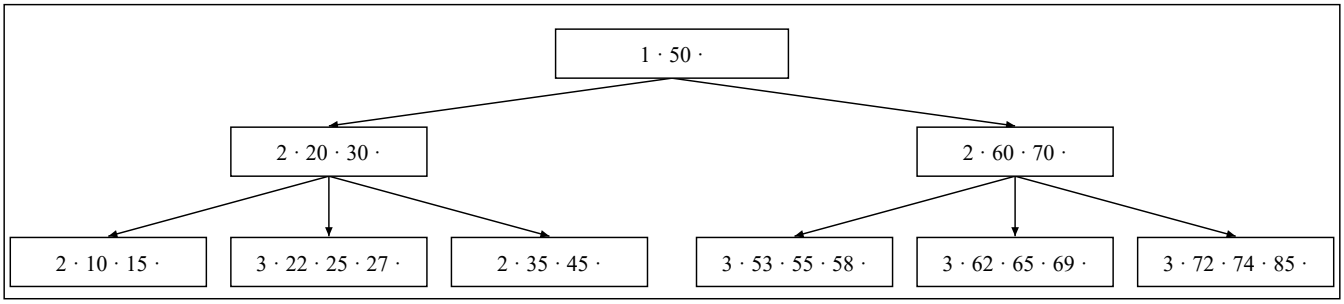
то освен горните релации са в сила и

$cc_{i-1} < c_0^1 < c_{en}^n < cc_i$

3. Ако възелът е листо, всичките му връзки $p_0, p_1, ..., p_{n-1}, p_n$ са равни на *NULL*.

Оригинал: стр. 217

Пример. Дървото от фиг. 3.9. е *B*-дърво от ред 4



Фигура 3.9. Пример за B-дърво от ред 4

B-дърветата от ред 4 се наричат *2–3–4 дървета*, тъй като всички техни възли, които не са листа, имат 2, 3 или 4 указателя към наследници. *B*-дърветата от ред 3 се наричат *2–3 дървета*.

Основни операции за работа с *B*-дървета са включването, изключването и търсенето на елемент, които са с логаритмична сложност. Ще покажем реализирането им чрез примери.

Търсене на елемент

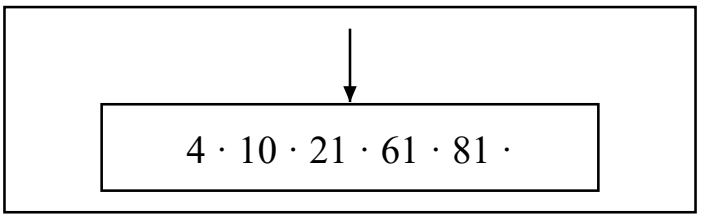
Започва се от корена. Сравнява се търсеният елемент с ключовете в корена. Ако елементът се намери там, търсенето завършва. В противен случай в зависимост от съотношението между елемента, който се търси, и стойностите на елементите, които се намират във възела, се избира едно от поддърветата.

Пример. Ако се търси елементът 62 в *B*-дървото от фиг. 3.9 62 се сравнява с елемента 50 от корена. Тъй като 62 > 50, търсенето продължава във второто поддърво. Сравнява се 62 с елементите 60 и 70 от възела – корен на второто поддърво. Тъй като е в сила 60 < 62 < 70, търсенето продължава във второто поддърво на дървото с възел с елементи 60 и 70. Тъй като 62 се намира във възела с елементи 62, 65 и 69, търсенето завършва успешно. Ако се търси елементът 63, търсенето преминава по същия път. Достига се отново до възела с елементи 62, 65 и 69 и тъй като там не се съдържа 63, търсенето е неуспешно.

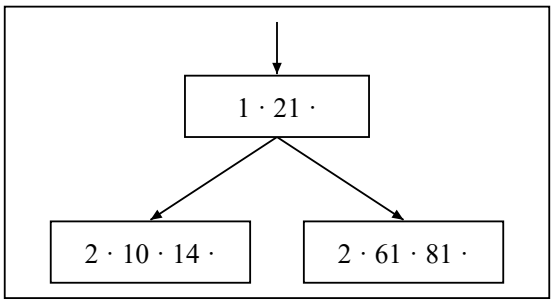
Включване на елемент

Включването ще реализираме за *B*-дърво от ред 5. Ще започнем с празното *B*-дърво. В него последователно ще включим елементите: 21, 81, 61, 10, 14, 34, 72, 23, 11, 17, 18, 4, 5, 6, 3, 2, 1.

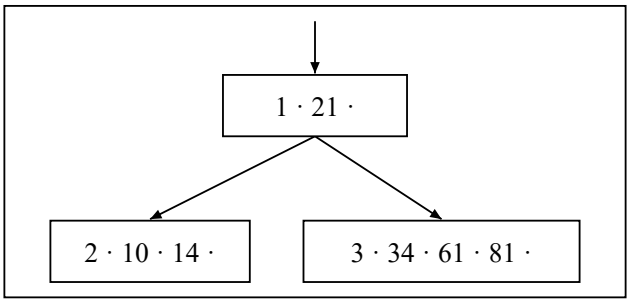
След включването на елементите 21, 81, 61 и 10 в празното *B*-дърво се получава



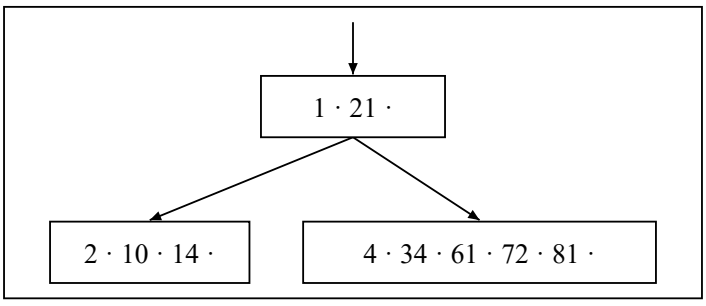
Включването на елемента 14 се осъществява по следния начин. Тъй като *n* на последното *B*-дърво е 4 (максималната стойност), в корена му не може да се включи 14. Ако можеше да се включи, елементите на корена щяха да имат вида: 10, 14, 21, 61, 81. Избира се средата 21 на последната редица, прави се корен на ново *B*-дърво, елементите 10 и 14, по-малки от 21, се разполагат в първото му поддърво, а 61 и 81, по-големи от 21, се разполагат във второто му поддърво. Получава се *B*-дървото



Включването на 34 в последното *B*-дърво не може да стане в корена му, тъй като тогава коренът ще има две информационни (даннови) и две свързващи полета. Включваме го във второто поддърво на *B*-дървото. Получава се

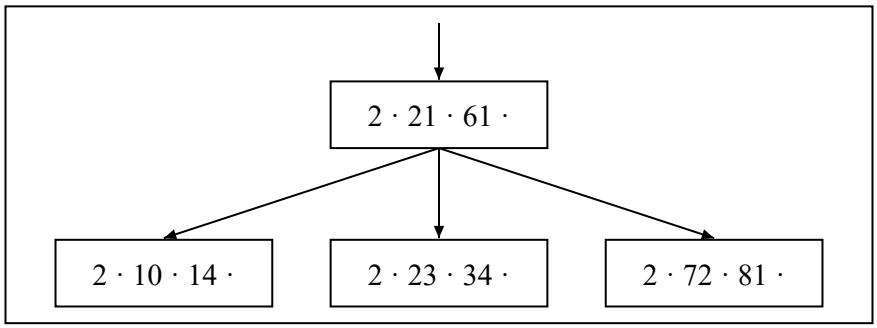


Включването на 72 в последното дърво получава *B*-дървото

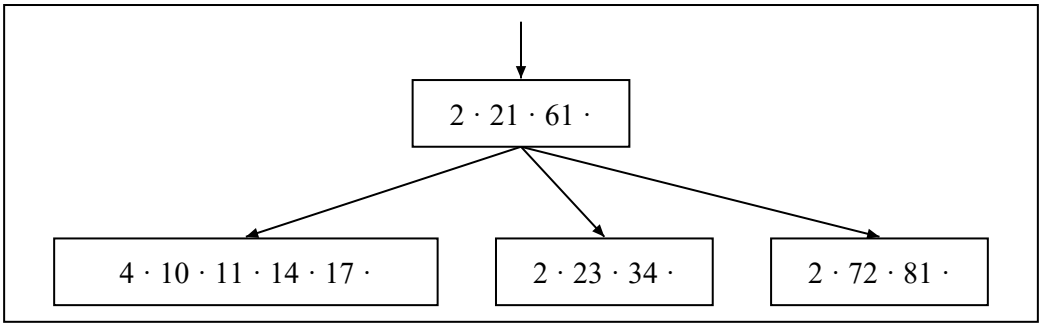


Оригинал: стр. 219

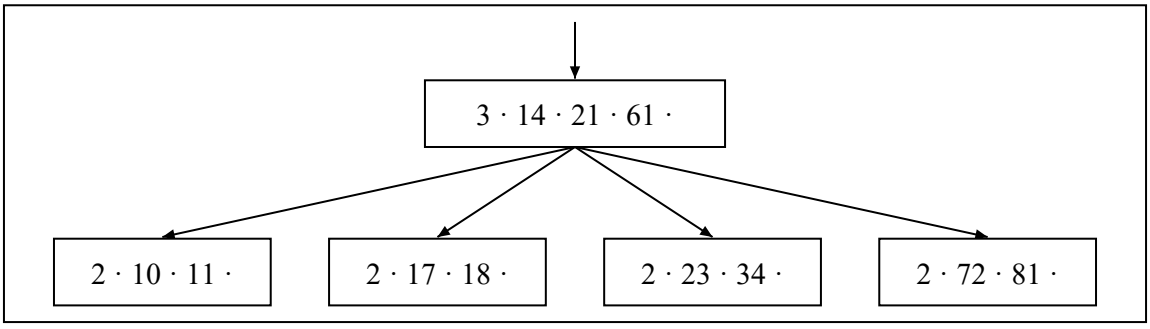
Включването на 23 в последното дърво се осъществява по следния начин: Тъй като 23 е по-голямо от 21, намиращ се в корена, 23 трябва да се включи във второто поддърво. То се състои от корен, който е листо, съдържащ максималния брой елементи. Ако можеше да се включи, елементите на този възел щяха да са 23, 34, 61, 72, 81. Отново се избира средният елемент 61, включва се в корена на *B*-дървото, увеличава се броят на наследниците от 2 на 3 и се разцепва второто поддърво на две поддървета на изходното *B*-дърво. Получава се



В резултат от включването на 11 и 17 в това дърво се получава

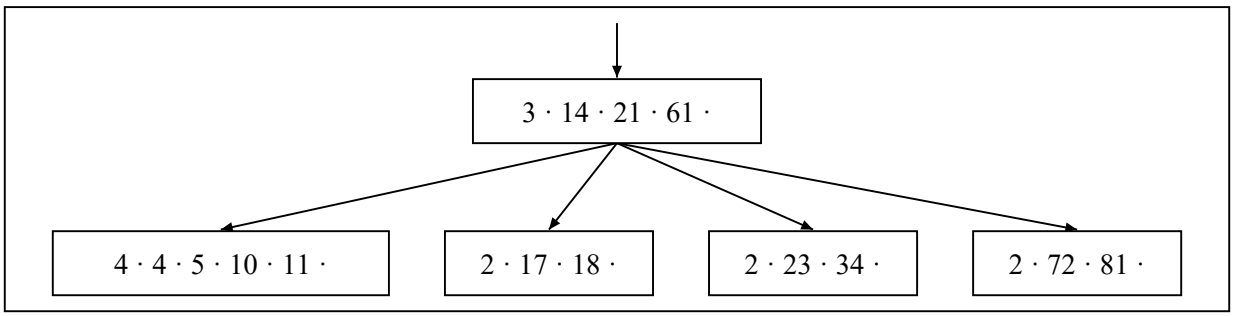


Понеже 18 е по-малко от 21, включването му в последното дърво трябва да стане в първото му поддърво. Тъй като коренът му съдържа 4 елемента, включването на 18 не може да стане като листо. Ако можеше да се включи, листото на първото поддърво щеше да има вида 10, 11, 14, 17, 18. Средният елемент е 14. Вмъква се в корена на *B*-дървото, броят на наследниците на дървото се увеличава с 1, а възелът, съдържащ 10, 11, 14, 17 се разцепва на два. Получава се

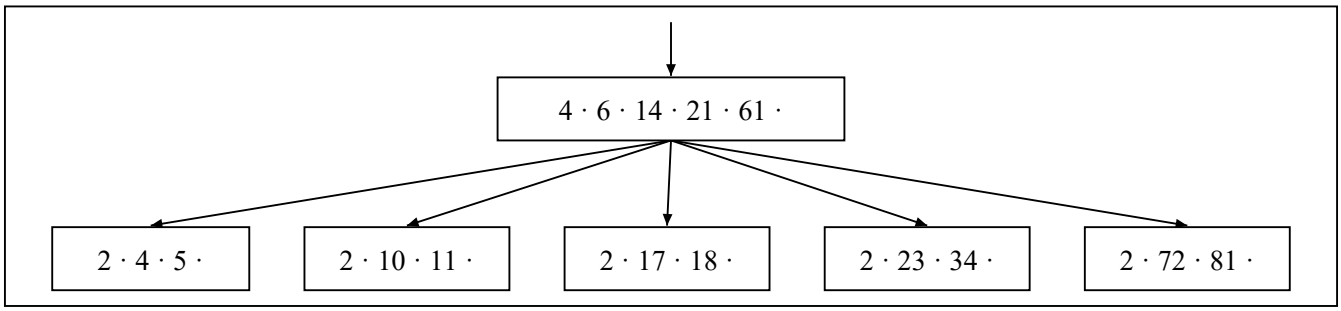


Оригинал: стр. 220

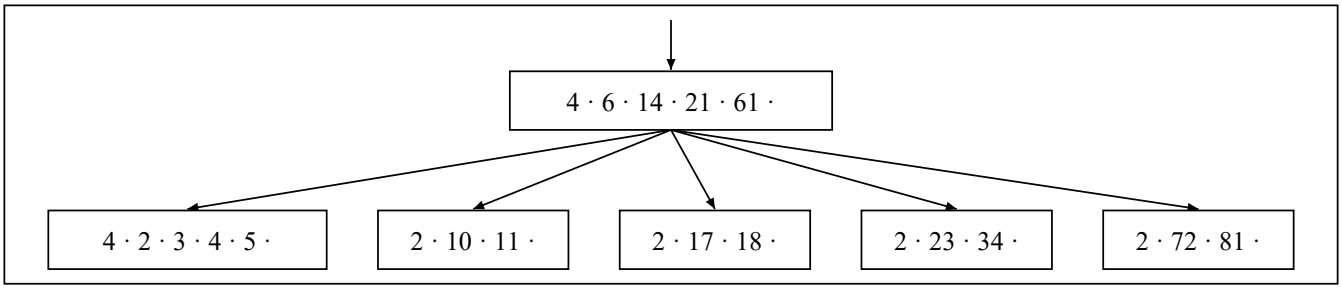
Резултатът от включването на елементите 4 и 5 в последното дърво е



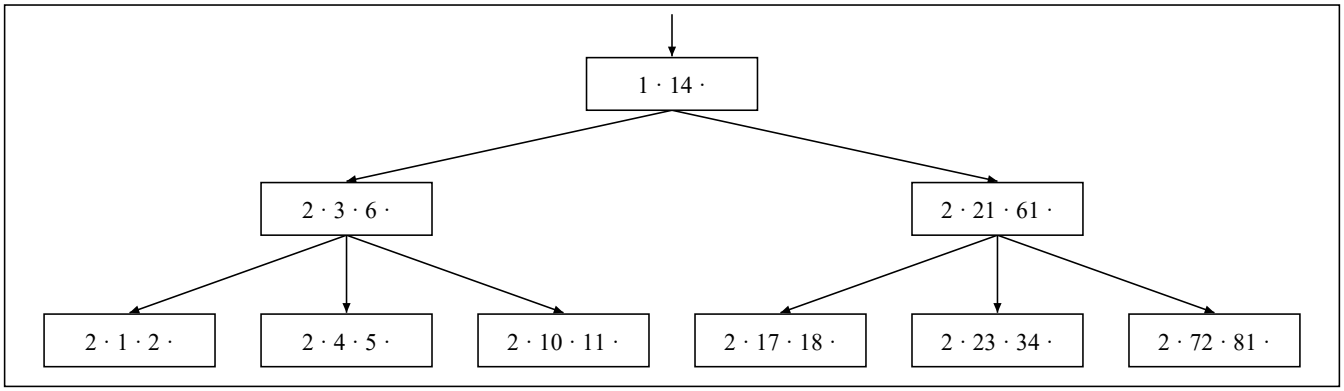
Включването на 6 разцепва първото (най-лявото) листо на две листа. В резултат се получава *B*-дървото



Включването на 3 и 2 в последното *B*-дърво води до получаване на

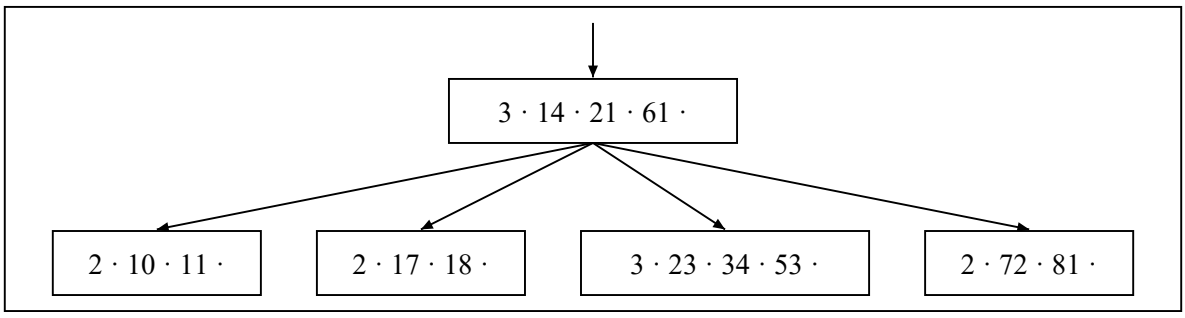


Остава да се включи 1. Тъй като 1 е по-малко от 6, трябва да се включи в първото поддърво с корен, съдържащ елементите 2, 3, 4, 5. Понеже този корен съдържа максималния брой елементи, 1 не може да се включи в него. Ако можеше да се включи, елементите на този възел щяха да са 1, 2, 3, 4, 5. Отново се избира средният елемент 3. Прави се опит за включване на 3 в по-горното дърво – в случая в корена на *B*-дървото. Останалите 4 елемента се разделят на две групи с елементи 1, 2 и 4, 5. Елементът 3 не може да се включи в корена на *B*-дървото, тъй като последният съдържа максималния възможен брой елемента. Ако можеше да се включи, коренът щеше да съдържа 3, 6, 14, 21, 61. Избира се средният елемент 14, прави се елемент на нов корен на *B*-дървото с две поддървета съдържащи 3, 6 и 21, 61. Новото *B*-дърво има вида

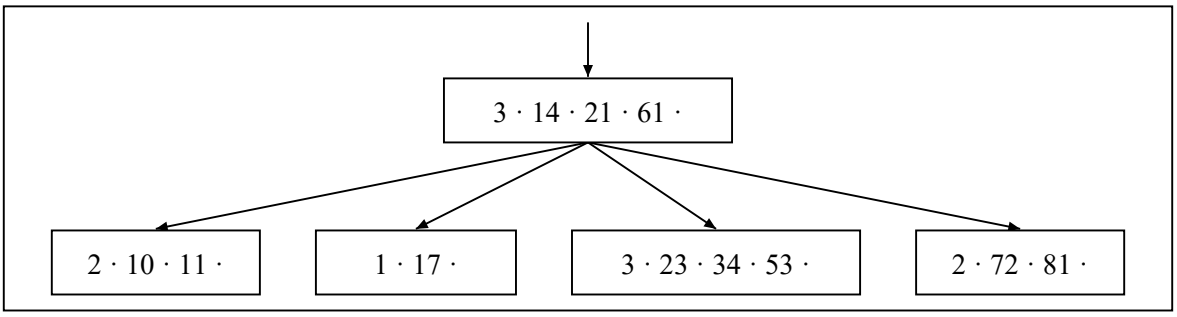


Изключване на елемент

Изключването на елемент е сложна операция. За да се изключи елемент от *B*-дърво, дървото се претърсва и ако се намери елементът, се отстранява. В процеса на всяко изключване основното изискване, отнасящо се до броя на данните (елементите) във всеки връх, трябва да се запази. Ще илюстрираме някои от случаите за изключване чрез пример. Нека да се изключи елемент 18 от *B*-дървото от ред 5 *(m = 5)*



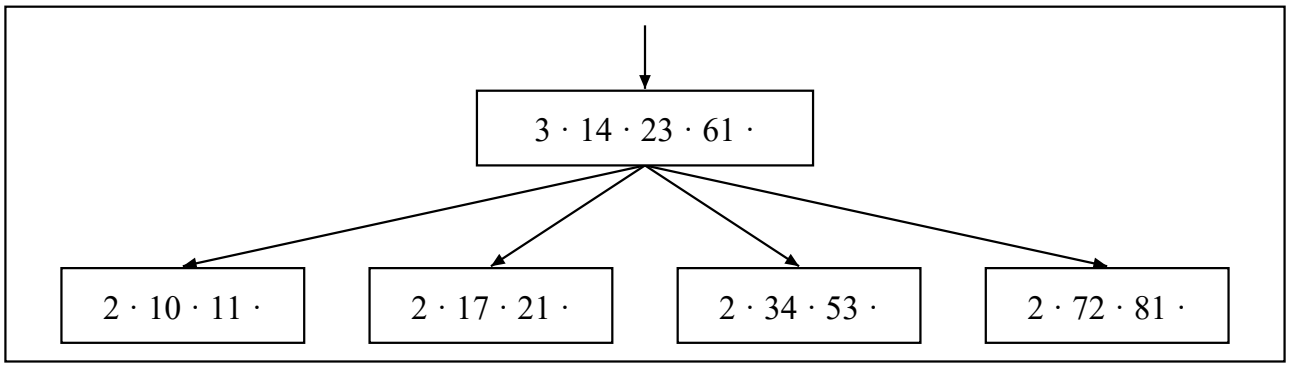
След претърсване, елемент 18 е намерен във второто листо. В резултат от отстраняването му се получава дървото



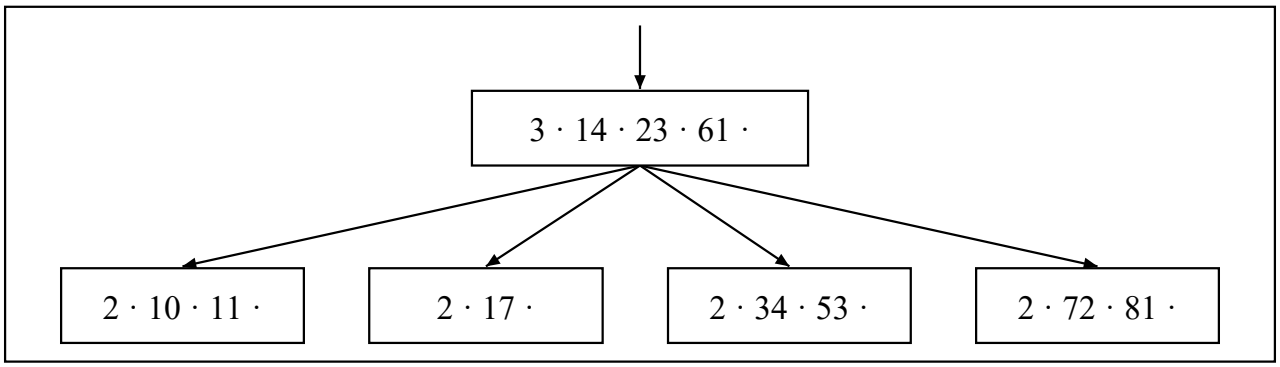
В него не е спазено условието за броя на данните във второто листо. Броят *n* трябва да е между цялата стойност на *(m-1)/2* и *m-1*, т.е. *(2 ≤ n ≤ 4)*. В този случай след елиминирането на елемента от листото, се прави опит *да се заеме елемент*. Заемането на елемент от корена (14 или 21) не е добра идея, тъй като ще намали данните на корена от 3 на 2, а броят на указателите ще е 4. Остава да се заеме елемент от съседен „брат“. Заемането от ляв „брат“ ще е въз-

Оригинал: стр. 222

можно, ако такъв съществува и броят на данните в него е по-голям от цялата стойност на *(m-1)/2*. В случая това не е така. Остава заемане от десния „брат“ на второто листо, броят на данните на който е по-голям от цялата стойност на *(m-1)/2* – в случая е 3. Заемането не може да стане с просто преместване на най-малкия елемент на десния „брат“ в мястото на недостига, тъй като така ще се наруши релацията между елементите в корена и някои от поддърветата. Затова в случая елементът 21 се премества от корена във второто листо, а на мястото на 21 в корена се включва елементът 23, който се заема от десния „брат“ на второто листо. В резултат се получава *B*-дървото

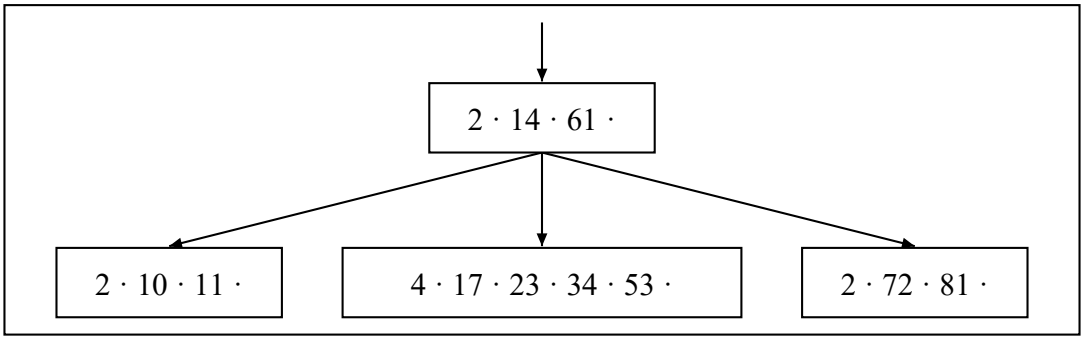


Ако десният брат на второто листо на *B*-дървото от примера имаше 2, а не 3 данни, заемането на елемент от него също щеше да е невъзможно, както не беше възможно заемането от левия му „брат“. Тази ситуация възниква, ако искаме да изключим елемента 21 от последното *B*-дърво. В резултат от отстраняването му се получава дървото

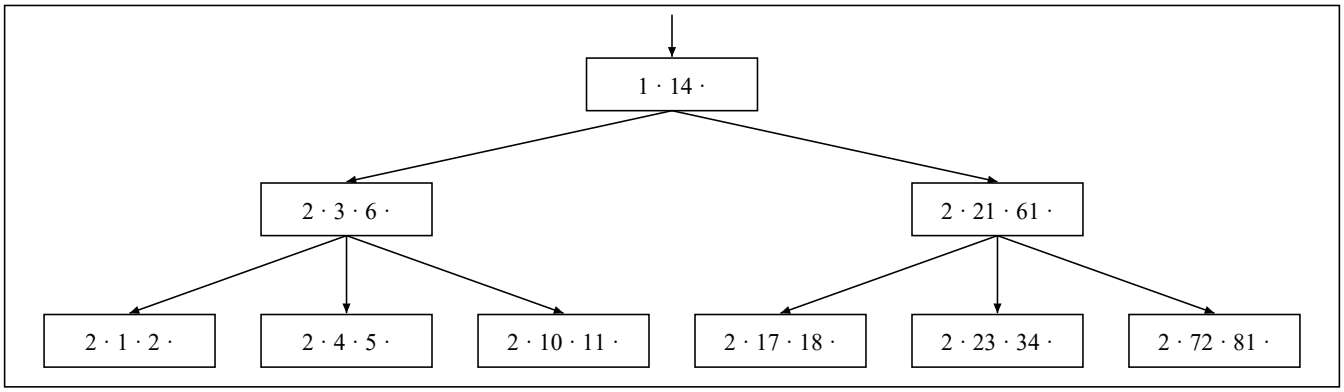


В него не е спазено условието за броя на данните във второто листо. Тъй като разрешаването на проблема с недостига не може да се реши чрез заемане от ляв или десен „брат“, се извършва *сливане на върхове* (в случая на второто и третото листо). Сливането на тези два върха чрез просто сливане на данновите полета не е възможно, тъй като тогава коренът ще е с 4 указателя, а наследниците му ще са 3 на брой. За целта елементът 23 от корена и всички елементи от десния „брат“ се добавят към второто листо, а третото се изключва, както и връзката към него от корена. Резултатът от изключването е *B*-дървото

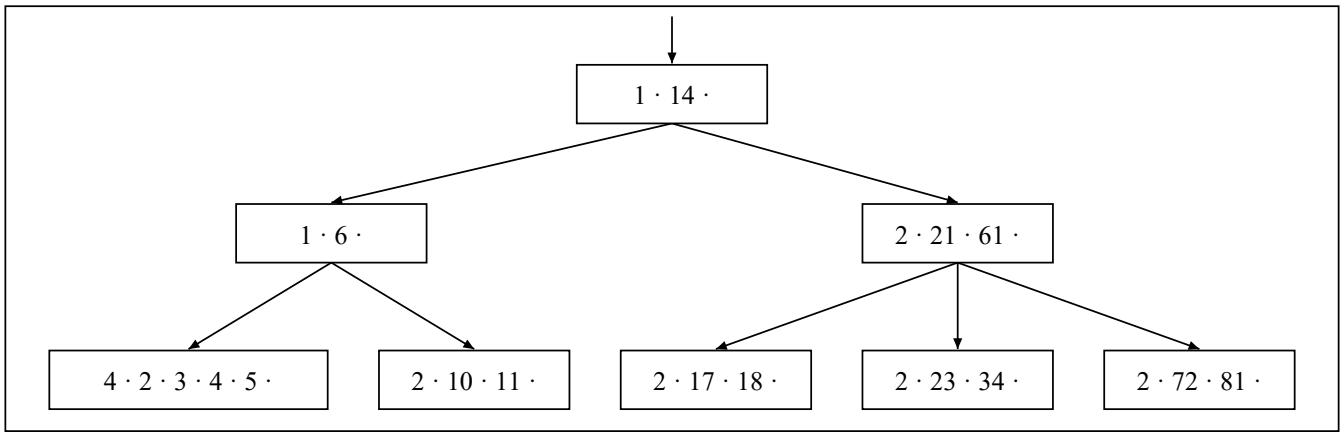
Оригинал: стр. 223



В предложения пример изключването е сравнително проста операция, тъй като заемането на елемент и сливането на възли се осъществява на ниво листа. Операцията се реализира по-сложно, когато заемането на елемент и сливането на възли се осъществява за вътрешни възли. Например нека да изключим елемента 1 от *B*-дървото от ред 5 *(m=5)*

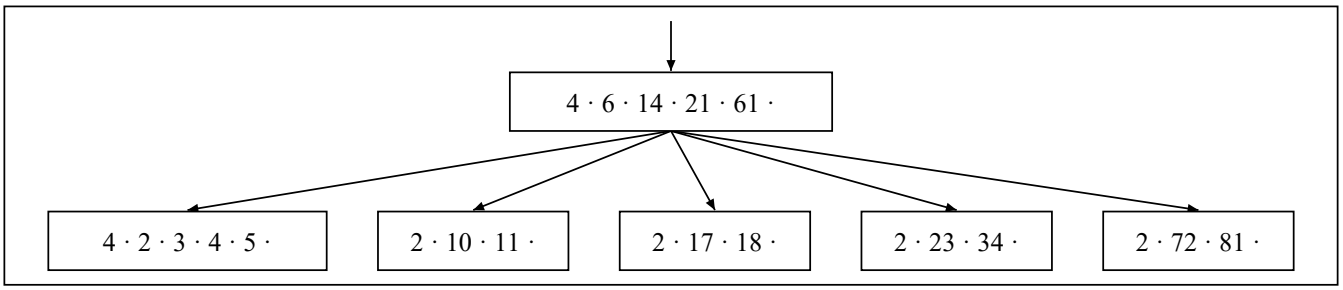


Дословното елиминиране на 1 от най-лявото листо нарушава условието за броя на данните в това листо. Получава се недостиг, който се разрешава със сливане на това листо с десния му „брат“. След сливането на двете листа се получава дървото



Оригинал: стр. 224

което не е *B*-дърво, тъй като възниква недостиг в корена на първото поддърво. Този недостиг трябва да се разреши със заемане и сливане до достигане на корена. В случая се получава *B*-дървото



B-дърветата са ефективни за много приложения. Съществуват добри алгоритми за работа с тях, но не всички от тях са прости за реализиране.
Реализацията на АТД *B*-дърво е оставена за самостоятелна работа.

Приложения на B-дървета

При работа с файлове има смисъл след като данните се достигнат, да не се чете само една данна, а по-големи количества, които да се запишат в ОП, след това да се обработят и да се запишат обратно във файла. Подходяща структура за организиране на тези действия са *B*-дърветата.

3.6. ГРАФ

Теорията на графите е интересен раздел на математиката. На базата на нея са реализирани редица полезни приложения. Началото на изучаването ѝ от математиците започва със статия, публикувана от Ойлер през 1736 година. Изследванията върху алгоритми за работа с графи като област от компютърната наука започва в края на XX век и продължават и до наши дни. В настоящата книга целите ни са много скромни. Ще определим структурата от данни граф и ще разгледаме някои приложения на графи. Голямо разнообразие от описания на алгоритми за работа с графи и техни приложения може да се намери в книгата по алгоритми на Пр. Наков и П. Добриков [13].

3.6.1. Дефиниране на граф

Логическо описание

Графът се определя като наредена двойка *(V, R)*, където $V = \{v_1, v_2, ..., v_n\}$ е крайно множество от върхове, а $R = \{r_1, r_2, ..., r_m\}$ е крайно множество от реб-

Оригинал: стр. 225

ра. Всяко ребро се задава чрез двойка върхове. Ребра, зададени като наредени двойки $<v_i, v_j>$ от върхове $(v_i, v_j \in V \text{ и } 1 \le i, j \le n)$ се наричат *ориентирани* или *дъги*. Ако ребрата на граф са ориентирани графът се нарича ориентиран. Ако ребрата на графа са зададени чрез ненаредени двойки (v_i, v_j) от върхове $(v_i, v_j \in V \text{ и } 1 \le i, j \le n)$, графът се нарича *неориентиран*.

Примери.

Транспортната карта на България може да се разглежда като неориентиран граф. Селищата са върховете на графа, а преките пътища между тях – ребрата му.
Страниците и връзките между тях в интернет могат да бъдат представени чрез ориентиран граф. Молекулярните структури, електронните схеми, маршрутите за магистрални системи и много други ситуации могат да се представят чрез графи.
Над граф са възможни следните основни операции:
– създаване на празен граф;
– достъп до връх;

– включване/изключване на връх;
– включване/изключване на ребро;
– проверка дали граф е празен;
– проверка дали елемент е връх на граф;
– проверка дали съществува ребро между два върха на граф;
– намиране на наследниците на даден връх.
Допълнителни операции над графи са:
– обхождане;
– търсене на различни видове пътища, на цикли, на оптимални пътища и др.

Някои дефиниции

Ребрата (дъгите) на граф могат да се свързват с етикети от произволен вид (имена, числа), в зависимост от конкретната задача. Етикетите се наричат още *тегла*, а граф, за ребрата на който са дефинирани тегла, се нарича *претеглен*. Теглата ще задаваме чрез функция $w: R \rightarrow W$, където R е множеството от ребрата, а W е множеството от теглата.

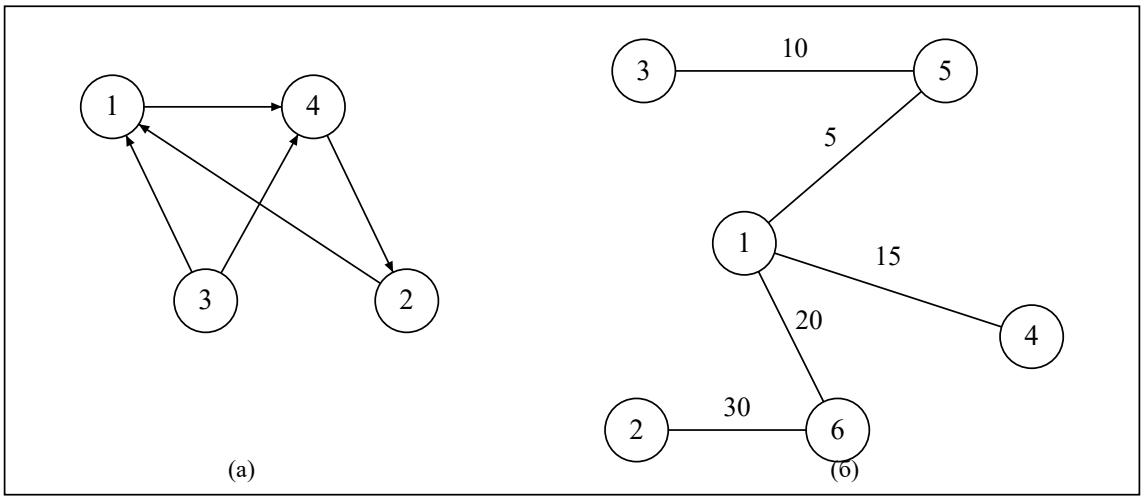
Примери. На фиг. 3.10 (а) е даден ориентиран граф с естествени числа за върхове, а на фиг. 3.10 (б) – неориентиран претеглен граф с естествени числа за върхове и естествени числа за тегла. Всеки връх от графите на двата примера е изобразен чрез кръгче, в което е поставено уникално естествено число. Всяко ребро в графа от фиг. 3.10 (а) е означено чрез стрелка, насочена от един към друг връх, а в графа от фиг. 3.10 (б) – като отсечка.

Ориентираният граф от фиг. 3.10 (а) може да се представи формално по следния начин: *G1 = (V1, R1)*, където *V1 = {1, 2, 3, 4}*, а *R1 = {<1, 4>, <4, 2>,*

Оригинал: стр. 226

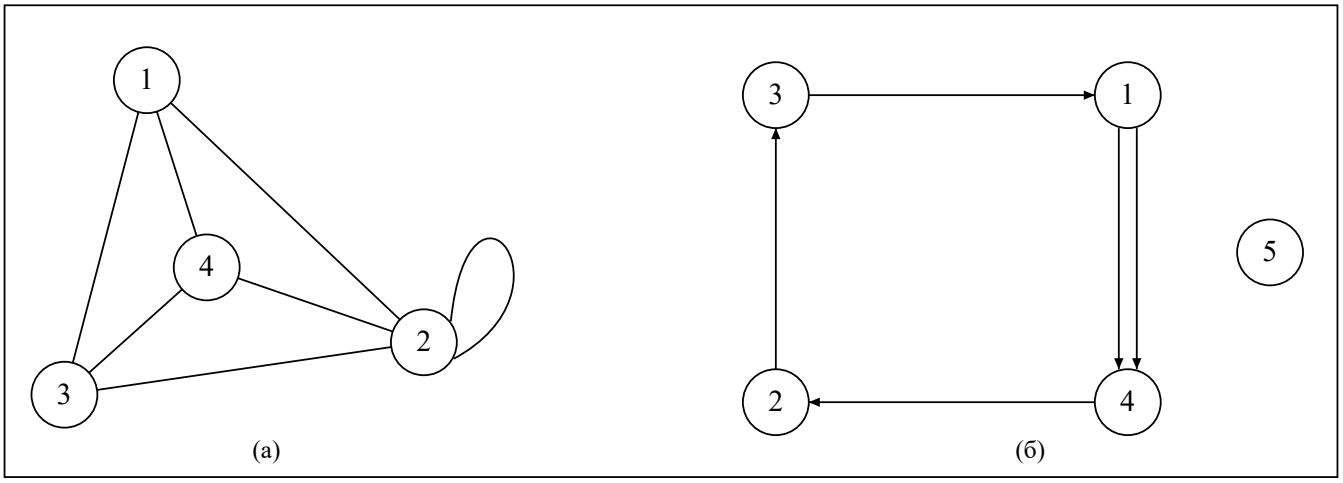
<2, 1>, <3, 1>, <3, 4>}. Неориентираният граф от фиг. 3.10 (б) може да се представи формално чрез *G2 = (V2, R2)*, където *V2 = {1, 2, 3, 4, 5, 6}, R2 = {(3, 5), (1, 5), (1, 4), (6, 1), (2, 6)}*, а функцията *w* за него се дефинира по следния начин:
w((3,5)) = 10, w((1, 5)) = 5, w((1, 4)) = 15, w((6, 1)) = 20, w((2, 6)) = 30.

Неориентираните графи могат да се представят като ориентирани.
Пример. Графът *G2* (от фиг. 3.10 б) може да се представи като ориентиран със следното множество от ребра: *R2 = {<3, 5>, <5, 3>, <5, 1>, <1, 5>, <1, 4>, <4, 1>, <1, 6>, <6, 1>, <2, 6>, <6, 2>}*.



Фигура 3.10. Примери за ориентиран и неориентиран претеглен граф

Възможно е ребро да излиза и влиза в един и същ връх. Такова ребро се нарича примка (фиг. 3.11 а). Ако в множеството на ребрата на граф се допуска повторение, графът се нарича мултиграф (фиг. 3.11 б).

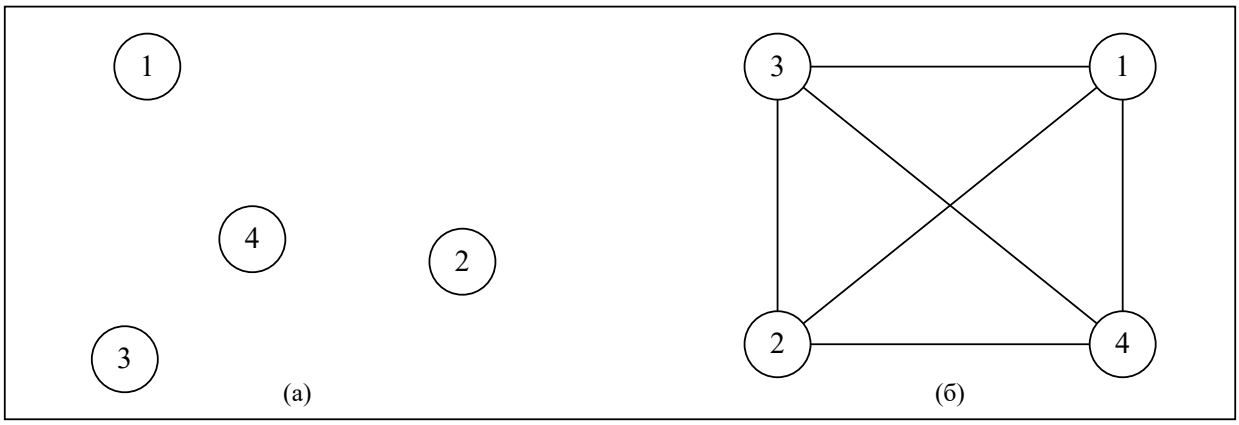


Фигура 3.11. Примери за граф с примка и за мултиграф

В следващата част от тази глава ще разглеждаме ориентирани графи. Неориентираните графи ще представяме като ориентирани.

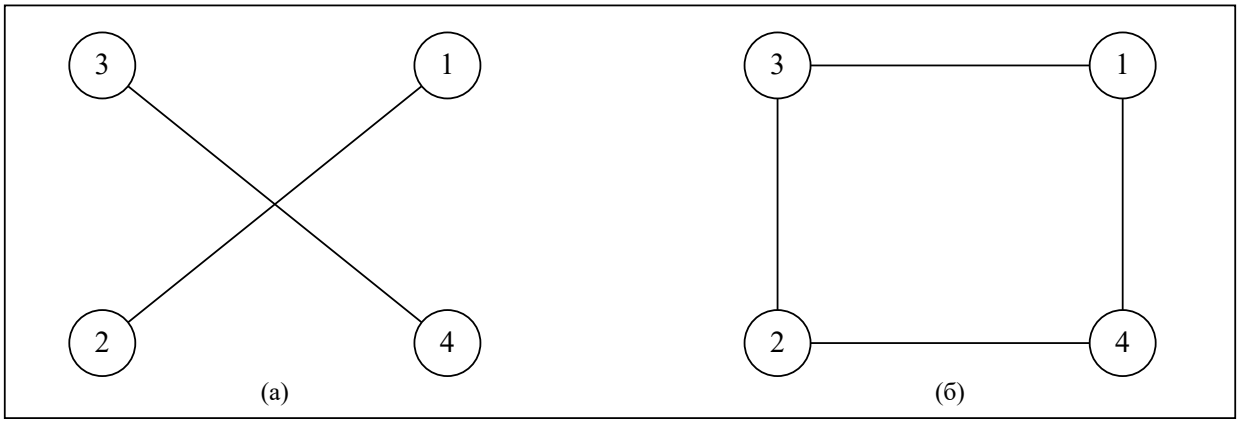
Оригинал: стр. 227

Нека *G = (V, R)* е ориентиран граф. Върховете *a* и *b* се наричат *съседни*, ако или <*a, b*>, или <*b, a*> принадлежи на *R*. Казва се още, че *a* и *b* са *краища на дъгата <a, b>, или <b, a>*. Ако <*a, b*> е дъга в графа *G*, върхът *a* се нарича *предшественик* на върха *b*, а върхът *b* се нарича *наследник* на *a*. Всеки от върховете *a* и *b* е инцидентен на дъгата <*a, b*>.
Полустепен на изхода на връх a се нарича броят на дъгите <*a, b*>, където *b* е връх на *G*. *Полустепен на входа на връх a* се нарича броят на дъгите <*b, a*>, където *b* е връх на *G*. Сумата от полустепените на входа и изхода на връх *a* се нарича *степен на върха a*. Връх със степен 0 се нарича *изолиран*. Например степента на връх 1 от фиг. 3.10 (а) е 3, а степента на връх 2 е 2. Връх 5 от фиг. 3.11 (б) е изолиран.
Граф, който не съдържа ребра, се нарича *празен* (фиг. 3.12. а), а ако всеки два различни върха на граф са свързани с дъга, графът се нарича *пълен* (фиг. 3.12. б).



Фигура 3.12. Примери за празен и за пълен граф

Ако за графите *G1 = (V, R1)* и *G2 = (V, R2)* е в сила, че за всеки два различни върха *a* и *b (a, b ∈ V)* дъгата *<a, b>* принадлежи или на *R1*, или на *R2*, то графът *G2* се нарича *допълнение на графа G1*, както и обратно – графът *G1* се нарича *допълнение на графа G2*. Примери са дадени на фиг. 3.13 и фиг. 3.12.



Фигура 3.13. Примери за допълващи се графи

Оригинал: стр. 228

Път в ориентиран граф *G = (V, R)* се нарича редица от върхове $v_1, v_2, \ldots, v_n$ така, че за всяко *i, i = 1, 2, …, n-1* е в сила $<v_i, v_{i+1}> \in R$. Върховете v_1 и v_n се наричат *краища на пътя*. Ако v_1 съвпада с v_n, пътят се нарича *цикъл*. Ако всички върхове на пътя са различни, пътят се нарича *прост* (или *ацикличен*). Ако върховете v_1 и v_n съвпадат, а останалите върхове на пътя $v_1, v_2, \ldots, v_n$ са различни, цикълът се нарича *прост*.

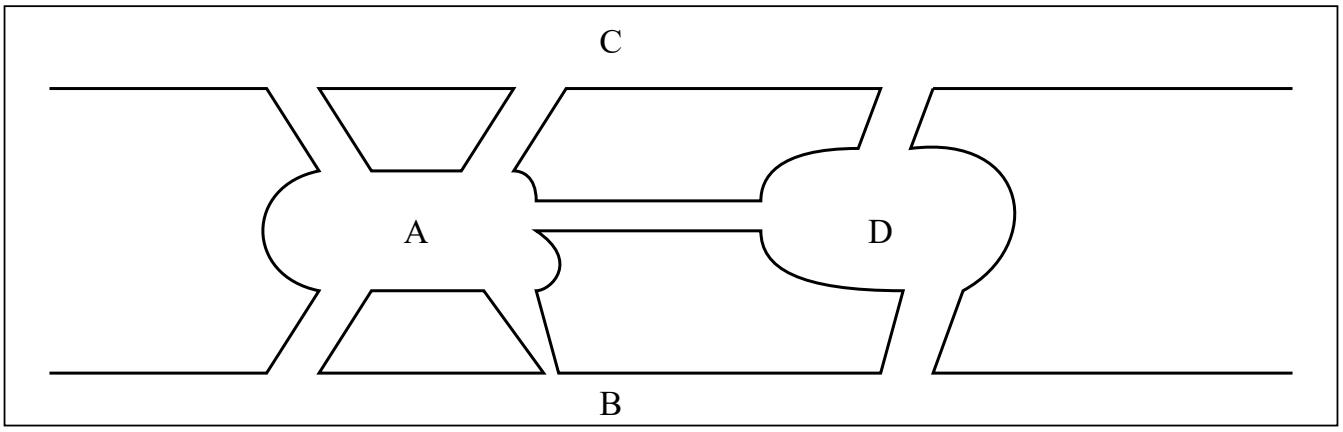
Ако граф съдържа поне един цикъл, той се нарича *цикличен*. В противен случай графът е *ацикличен*.

Ориентиран граф е *слабо свързан*, ако всеки два негови върха *a* и *b* са краища на поне един път, т.е. съществува поне един път от *a* до *b* или от *b* до *a*. Ориентиран граф е *силно свързан* (или само *свързан*), ако всеки два негови върха *a* и *b* са краища както на път от *a* до *b*, така и на път от *b* до *a*.

Неориентиран граф е *свързан*, ако съществува път между всяка двойка негови върхове.
Неориентиран свързан граф без цикли се нарича *дърво*.
Прост път в граф, който съдържа всички върхове на графа, се нарича *Хамилтонов път*.

Ойлерови цикли

Цикъл в свързан мултиграф, в който всяко ребро участва точно веднъж, се нарича *Ойлеров цикъл*. Свързан мултиграф, който съдържа Ойлеров цикъл се нарича *Ойлеров мултиграф*.
Този вид графи и изследванията върху тях водят началото си от Л. Ойлер, който търси решение на задачата за Кьонигсбергските мостове.
Градът Кьонигсберг (сега Калининград, Русия) имал 7 моста на река Прегел, които схематично могат да се изобразят по следния начин



Често жителите на града си задавали въпроса дали е възможно тържествена процесия да премине точно по веднъж по всеки от мостовете и разходката да завърши в изходната позиция. Ойлер представил мостовете като ребра на неориентиран мултиграф с върхове *A, B, C* и *D* и доказал, че това не е възможно.

Оригинал: стр. 229

Тази задача е пример за търсене на Ойлеров цикъл. Известна е следната теорема на Ойлер.

Теорема. Свързан неориентиран мултиграф съдържа Ойлеров цикъл тогава и само тогава, когато всички върхове на мултиграфа са от четна степен.

Физическо представяне

Съществуват два основни начина за представяне на ориентиран граф:
– последователно и
– свързано.

Последователно представяне

Най-често се използват следните два начина за последователно представяне на ориентиран граф:
– чрез *матрица на съседство* и
– чрез *матрица на инцидентност между върховете и ребрата*.

а) чрез матрица на съседство

За ориентиран граф с *n* върха, номерирани с *1, 2, ..., n*, матрицата на съседство *A* има *n* реда и *n* стълба. Ако *<i, j>* е дъга, ориентирана от връх *i* до връх *j* в графа, елементът в *i*-тия ред и *j*-тия стълб на матрицата на съседство е равен на 1 (*A[i][j] == 1*), в противен случай той е 0 (*A[i][j] == 0*).

Пример. Матрицата на съседство за графа от фиг. 3.10 (а) има вида

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Два върха *i* и *j* са свързани с път с дължина 2, ако съществува връх *k*, така че *A[i][k]* и *A[k][j]* са равни на 1. Последното е в сила тогава и само тогава, когато $A^2[i][j] = 1$. Аналогично два върха *i* и *j* са свързани с път с дължина 3, тогава и само тогава, когато $A^3[i][j] = 1$. Съществува път с произволна дължина от връх *i* до връх *j*, тогава и само тогава, когато $A^+[i][j] = 1$, където

$A^+ = A + A^2 + A^3 + \ldots$

Безкрайната сума има само краен брой значещи събираеми. В сила е, че $A^p = A^n$, за всяко $p \geq n$.

За някои приложения, ако не съществува дъга $<i, j>$, но съществува дъга $<j, i>$, елементът *A[i][j]* се свързва с -1, вместо с 0, а *A[j][i]* е равен на 1.
Матрици на съседство, с голям брой нулеви елементи, се наричат *разредени*.

Оригинал: стр. 230

Ако графът е неориентиран, матрицата му на съседство е симетрична.
Ако графът е непритеглен мултиграф, като стойност на *A[i][j]* се указва броят на ребрата между върховете *i* и *j*, а не 1. Ако графът е претеглен, в *A[i][j]* се указва теглото $w(<i, j>)$. В този случай матрицата на съседство се нарича *матрица на теглата*.

Най-важното предимство на този вид представяне е, че проверката дали съществува ребро между два върха се осъществява само с една операция ($O(1)$ – пряк достъп до елемент на двумерен масив), а намирането на наследниците на даден връх е с линейна сложност. Това предимство, както и по-доброто познаване на структурата от данни масив прави представянето сред най-често използваните. При приложения, в които се изисква изграждане на големи графи, матриците на съседство заемат голямо количество памет и доста алгоритми, използващи това представяне водят до квадратично време за изпълнение. В тези случаи се препоръчва да се използва вариант на свързано представяне на граф.

б) чрез матрица на инцидентност между върховете и ребрата

Ако $G = (V, R)$ и броят на върховете на *G* е *n*, а броят на ребрата му е *m*, матрицата на инцидентност между върховете и ребрата на *G* е матрица с размерност *n x m*. Редовете на матрицата представят върховете, а колоните ѝ – ребрата. При ориентираните графи за всяко ребро $r_k = <i, j>$ стълбът с пореден номер *k* ще съдържа 1 в *i*-тата и -1 в *j*-тата си позиции. Останалите му позиции ще съдържат 0.

Пример. Матрицата на инцидентност между върховете и ребрата за графа $G_1 = (V_1, R_1)$ от фиг. 3.10 (а) има вида

$$I = \begin{pmatrix} 1 & 0 & -1 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ -1 & 1 & 0 & 0 & -1 \end{pmatrix}$$

Ако графът е неориентиран, стълбът *k* ще съдържа две единици в позициите *i* и *j*, а в останалите позиции стойностите ще са 0. Ако графът има примка <*i, i*>, тогава в *i*-тата позиция на *k*-тия стълб стойността ще е 2, а в останалите – 0.

Това представяне се използва най-рядко, защото матрицата на инцидентност между върховете и ребрата е силно разредена, а необходимата за представянето ѝ памет е от порядъка на *n.m*.

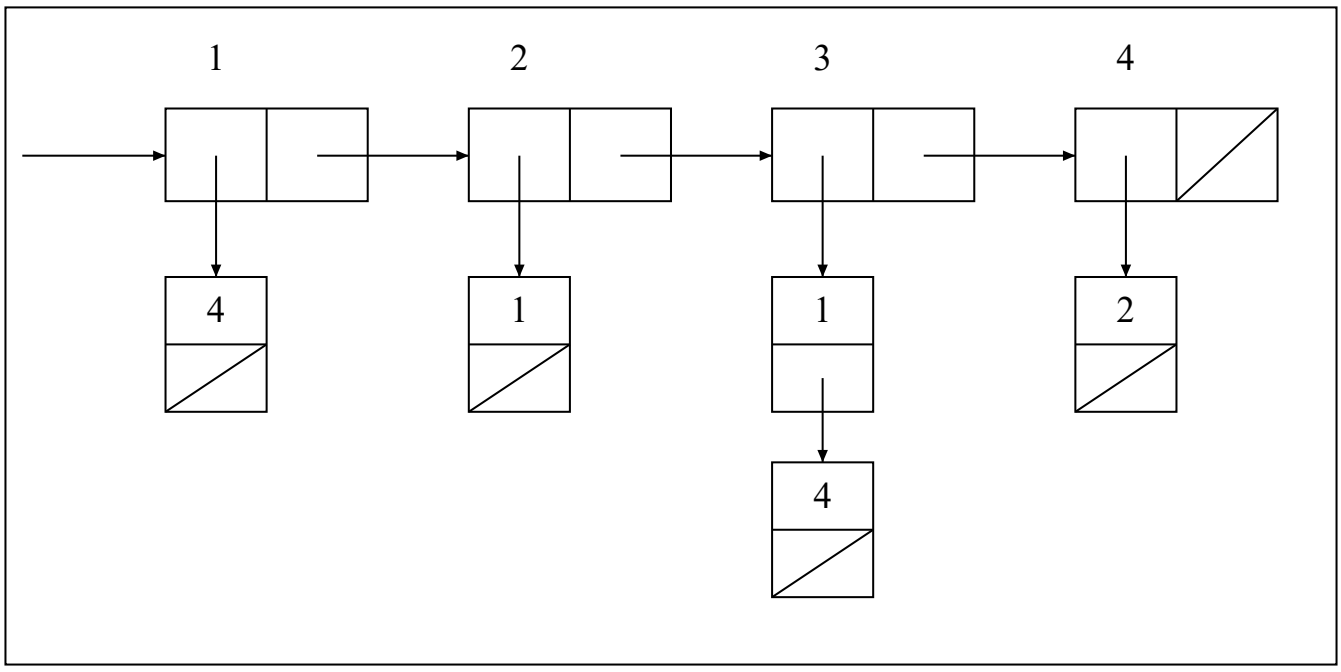
Оригинал: стр. 231

Свързано представяне

Съществуват различни начини за реализиране на това представяне. Сред най-често използваните са представянията чрез списък на инцидентностите и чрез списък на ребрата.

а) чрез списък на инцидентностите

При това представяне за всеки връх на ориентирания граф се съхранява списък от наследниците му. Например за графа от фиг. 3.10 (а) то има вида, показан на фиг. 3.14



Фигура 3.14. Пример за представяне на ориентиран граф
чрез списък на инцидентностите

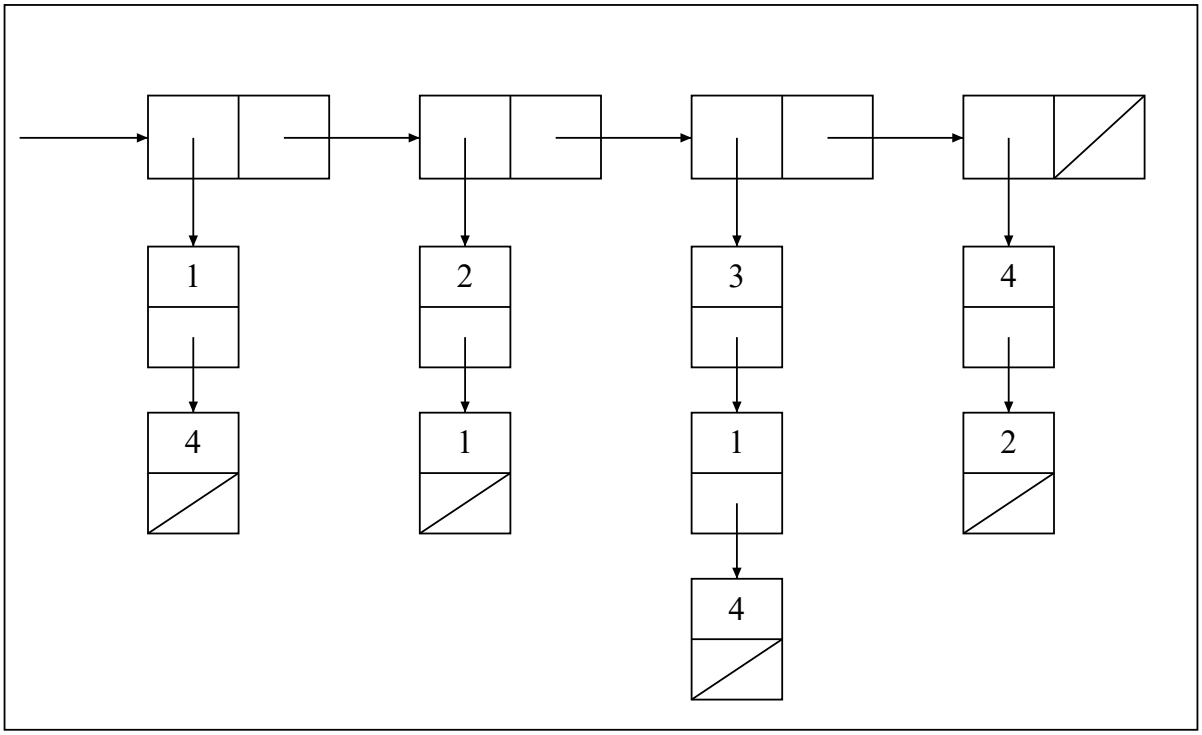
Особеното е, че върховете на графа са представени като поредни номера на звената на линейния списък, представен свързано с една връзка.

В книгата ще използваме вариант на това представяне, който не предполага върховете на графа да са естествени числа, явяващи се поредни номера на звената на линейния списък. При него се използва линеен списък от толкова елемента, колкото са върховете на графа. Елементите на

списъка са списъци, като всеки подсписък започва с връх *i* на графа и съдържа всички върхове *j*, така че има дъга от върха *i* до върха *j*.

Пример. Това представяне за графа от фиг. 3.10 (а) е показано на фиг. 3.15. То позволява типът на върховете на графа да е произволен, а не само цял.

Оригинал: стр. 232

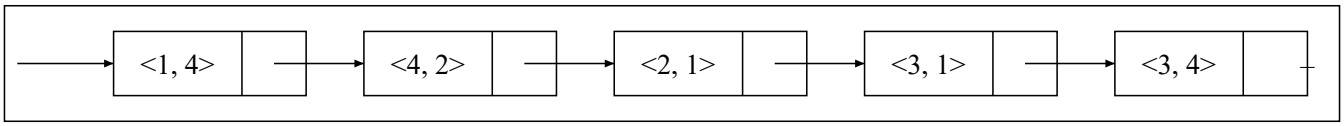


Фигура 3.15. Пример за модифицирано представяне на ориентиран граф
чрез списък на инцидентностите

При това представяне сложността за намиране на всички наследници на даден връх е линейна по отношение на броя на наследниците, но намирането на връх е със сложност от порядъка на броя на върховете. Проверката дали съществува ребро между два върха е също със сложност от порядъка на броя на наследниците.

б) чрез списък на ребрата

Това физическо представяне е най-близко до дефиницията на граф. Ориентиран граф се представя като списък от наредени двойки *<i, j>*.
Например за графа от фиг. 3.10 (а) то има вида



Ако графът е неориентиран, двойките могат да са неориентирани. Ако графът е ориентиран и претеглен, се представя чрез наредени тройки *<i, j, w((i, j))>*. За реализиране на това представяне е нужна памет от порядъка на броя на ребрата. Проверката дали два върха са съседни, както и дали елемент е връх на

графа, е също от порядъка на броя на ребрата. Ако ребрата са лексикографски наредени, може да се приложи двоично, а не линейно търсене, и проверката за връх или ребро е от порядъка $\log_2$брой_на_ребрата. Това представяне е подходящо за графи с относително малък брой на ребрата, т.е. за разредени графи.

Изборът на най-подходящото представяне последователно или свързано зависи от конкретната задача. В книгата ще използваме свързано представяне чрез списък на инцидентностите.

3.6.2. Реализация на граф чрез списък на инцидентностите

Свързаното представяне чрез списък на инцидентностите, описано по-горе, ще реализираме като шаблон на клас *graph*. За целта ще използваме наготово шаблона на класа *LList<T>*, дефиниращ линеен списък с една връзка и записан във файла *LList.cpp*. Шаблонът *graph<T>* включва член-функции за:
– включване на връх (*addTop*);
– изключване на връх (*deleteTop*);
– включване на ребро (*addRib*);
– изключване на ребро (*deleteRib*);
– проверка дали елемент е връх (*top*);
– проверка дали съществува ребро от един до друг връх (*rib*);
– проверка дали граф е празен (*empty*);
– достъп до указан връх на графа (*point*);
– намиране на списък от върховете на графа (*vertexes*);
– извеждане на граф на екрана (*print*).

C233-01 · Копиране: code.html

```
template <typename T>
class graph
{ public:
    void addTop(const T&);
    void deleteTop(const T&);
    void addRib(const T&, const T&);
    void deleteRib(const T&, const T&);
    bool top(const T&);
    bool rib(const T&, const T&);
    bool empty() const;
    elem_link1<T>* point(const T&);
    LList<T> vertexes();
    void print();
  private:
```

Оригинал: стр. 234

C234-01 · Копиране: code.html

```
    LList< LList<T> > g;
};

// включва a като връх на неявния граф
template <typename T>
void graph<T>::addTop(const T& a)
{ // създаване на линеен списък, съдържащ елемента a
  LList<T> l;
  l.toEnd(a);
  // включване на върха a към графа
  g.toEnd(l);
}

// изключва върха a на неявния граф
template <typename T>
void graph<T>::deleteTop(const T& a)
{ // обхожда графа с цел изтриване на всички
  // ребра от произволен връх до върха a
  g.iterStart();
  elem_link1<LList<T> >* p = g.iter(), *r;
  while(p)
    { p->inf.iterStart();
      elem_link1<T>* q = p->inf.iter();
      if (rib(q->inf, a)) deleteRib(q->inf, a);
      p = p->link;
    }
  // изтриване на линейния списък, представящ
  // върха a и неговите наследници
  g.iterStart();
  elem_link1<T>* q;
  do
  { r = g.iter();
    r->inf.iterStart();
    q = r->inf.iter();
  } while(q->inf != a);        // a е връх на графа
  LList<T> x;
  g.deleteElem(r, x);
}

// включва ребро от върха a до върха b на неявния граф
```

Оригинал: стр. 235

C235-01 · Копиране: code.html

```
template <typename T>
void graph<T>::addRib(const T& a, const T& b)
{ // намиране на указател към върха a на неявния граф
  elem_link1<T>* q = point(a),// намира указател към върха a
                 * p;
  // включване на върха b в списъка от наследниците на върха a
  p = new elem_link1<T>;
  assert(p != NULL);
  p->inf = b;
  p->link = q->link;
  q->link = p;
}

// изключва реброто от върха a до върха b на неявния граф
template <typename T>
void graph<T>::deleteRib(const T& a, const T& b)
{ g.iterStart();
  elem_link1<LList<T> > *p;
  elem_link1<T> *q;
  do
    { p = g.iter();
      p->inf.iterStart();
      q = p->inf.iter();
    } while (q->inf != a);
  q = q->link;
  // намиране на указател към наследника b на върха a
  while(q->inf != b) q = q->link;
  T x;
  p->inf.deleteElem(q, x);
}

// проверява дали a е връх на неявния граф
template <typename T>
bool graph<T>::top(const T& a)
{ if(g.empty()) return false;
  g.iterStart();
  elem_link1<LList<T> > *p = g.iter();
  elem_link1<T> *q;
  do
    { p->inf.iterStart();
```

Оригинал: стр. 236

C236-01 · Копиране: code.html

```
      q = p->inf.iter();
      p = p->link;
    } while(q->inf != a && p);
  return q->inf == a;
}

// проверява дали има ребро от върха a до върха b
template <typename T>
bool graph<T>::rib(const T& a, const T& b)
{ elem_link1<T>* p = point(a); // намира указател към върха a
  p = p->link;
  while (p && p->inf != b) p = p->link;
  return p != NULL;
}

// проверява дали неявният граф е празен
template <typename T>
bool graph<T>::empty() const
{ return g.empty();
}

// намира указател към върха a на графа
template <typename T>
elem_link1<T>* graph<T>::point(const T& a)
{ g.iterStart();
  elem_link1<LList<T> > *p;
  elem_link1<T> *q;
  do
    { p = g.iter();
      p->inf.iterStart();
      q = p->inf.iter();
    } while(q->inf != a);
  return q;
}

// връща списък от върховете на неявния граф
template <typename T>
LList<T> graph<T>::vertexes()
{ LList<T> l;
  g.iterStart();
```

Оригинал: стр. 237

C237-01 · Копиране: code.html
```
  elem_link1<LList<T> > *p = g.iter();
  while(p)
    { p->inf.iterStart();
      elem_link1<T>* q = p->inf.iter();
      l.toEnd(q->inf);
      p = p->link;
    }
  return l;
}

// извежда неявния граф
template <typename T>
void graph<T>::print()
{ g.iterStart();
  elem_link1<LList<T> > *p = g.iter();
  while (p)
    { p->inf.print();
      p = p->link;
    }
  cout << endl;
}
```

Записваме шаблона на класа във файла *graph.cpp*.

Задача 41. Да се дефинира шаблон на процедура, която създава непразен граф. Да се включи процедурата в програма, с която да се създаде непразен граф с цели върхове, след което да се изведе.

Дефиницията

C237-02 · Копиране: code.html
```
typedef graph<int> intGraph;
```

определя класа *intGraph*, задаващ граф с цели числа във върховете.

За създаването на графа ще реализираме следния алгоритъм. Създаваме празен граф. Последователно въвеждаме цели числа, означаващи върховете на графа, след което ги включваме в графа чрез член-функцията *addTop*. След това въвеждаме наредени двойки от върхове, означаващи дъги и ги включваме в графа чрез член-функцията *addRib*.

Оригинал: стр. 238

C238-01 · Копиране: code.html

```
// Program Zad41.cpp
#include <iostream>
#include <cassert>
#include "LList.cpp"
#include "graph.cpp"
using namespace std;

template <typename T>
void create_graph(graph<T> &g)
{ // включване на върховете
  char c;
  do
    { cout << "top_of_graph: ";
      T x; cin >> x;
      g.addTop(x);
      cout << "Top y/n? "; cin >> c;
    } while (c == 'y');
  // включване на ребрата
  cout << "ribs:\n";
  do
    { cout << "start top: ";
      T x; cin >> x;
      cout << "end top: ";
      T y; cin >> y;
      g.addRib(x, y);
      cout << "next: y/n? "; cin >> c;
    } while (c == 'y');
}

typedef graph<int> intGraph;
int main()
{ intGraph g;      // създаване на празен граф g
  create_graph(g); // включване на върхове и ребра в g
  g.print();       // извеждане на екрана на графа g
  return 0;
}
```

Друга реализация на шаблона на процедурата *create_graph*, която в някои случаи е по-удобна, е

Оригинал: стр. 239

C239-01 · Копиране: code.html

```
template <typename T>
void create_graph(graph<T>& g)
{ cout << "number of tops: ";
  int n; cin >> n;
  for (int i = 1; i <= n; i++)
    { cout << "top_of_graph: ";
      T x; cin >> x;
      g.addTop(x);
      cout << "number of ribs from " << x << ": ";
      int k; cin >> k;
      for (int j = 1; j <= k; j++)
        { cout << "top: ";
          T y; cin >> y;
          g.addRib(x, y);
        }
    }
}
```

Тя с въвеждането на всеки връх на графа въвежда и наследниците му.

Обхождане на граф

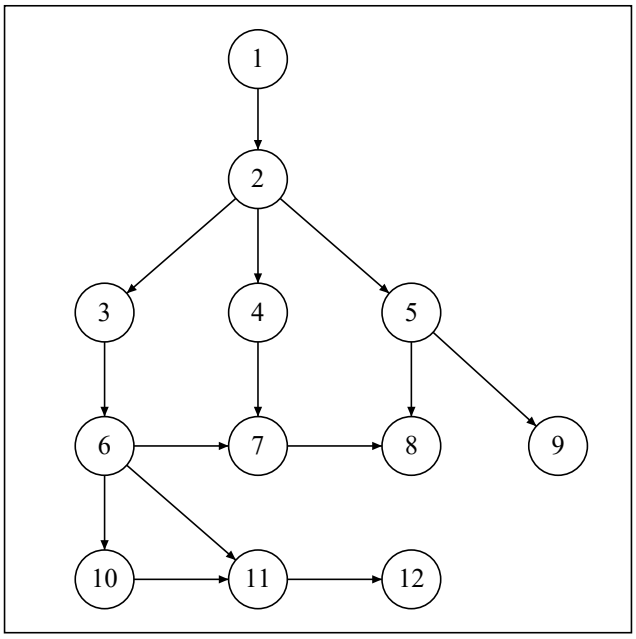
Обхождането на граф е операция, при която последователно и точно по веднъж се посещава всеки връх на графа, като се започва от даден връх и се преминава през ребро.
Най-често се използват:
– обхождане в ширина, съкратено *BFS* (от *Breadth-First-Search*);
– обхождане в дълбочина, съкратено *DFS* (от *Depth-First-Search*).

а) обхождане в ширина

Обхождането в ширина, започващо от даден връх *a* на графа, се осъществява по следния начин: започва се от върха *a*; след това се разглеждат всички негови преки наследници, които още не са обходени и се завършва обхождането с обхождане в ширина, започващо от всеки от преките наследници на *a*.

Оригинал: стр. 240

Пример: Обхождането в ширина на графа



започващо от върха 1 има вида:
стъпка 1: 1
стъпка 2: 2
стъпка 3: 5, 4, 3
стъпка 4: 9, 8, 7, 6
стъпка 5: 11, 10
стъпка 6: 12,

а обхождането в ширина на същия граф, започващо от върха 3, има вида:
стъпка 1: 3
стъпка 2: 6
стъпка 3: 11, 10, 7
стъпка 4: 12, 8.

Резултатите във всяка стъпка са с точност до наредба на преките наследници на даден връх.
От примера се вижда, че от върха 1 на графа се обхождат в ширина всички негови върхове, но от върха 3 – само част от върховете.
Следва реализация на този вид обхождане за направения избор на свързано представяне на ориентиран граф. Параметърът *a* на шаблона на процедурата *BFS*, дефиниран по-долу, означава върха на графа, от който започва обхождането, параметърът *g* – графа, който се обхожда, а *l* е помощен линеен списък, който отначало е празен, а след завършване на изпълнението на *BFS* съдържа върховете на *g*, които са обходени.

Оригинал: стр. 241

Реализацията използва помощна опашка с елементи от тип *T*. Първоначално в опашката се включва върхът, от който започва обхождането. След това, докато опашката не е празна, се изключва връх от нея, извършва се желаното действие над този връх (в случая се извежда на екрана) и в опашката се включват всички наследници на изключения връх, които не са посетени до момента. Проверката дали връх е посетен, е извършена, като се използва помощният списък *l*, в който се записват всички посетени до дадения момент върхове на графа.

C241-01 · Копиране: code.html
```
template <typename T>
void BFS(const T& a, graph<T> g, LList<T>& l)
{ queue<T> q; q.push(a);
  l.toEnd(a);
  while(!q.empty())
  { T x; q.pop(x);
    cout << x << " ";
    elem_link1<T>* p = g.point(x);
    p = p->link;
    while (p)
      { if (!member(p->inf, l))
          { q.push(p->inf);
            l.toEnd(p->inf);
          }
        p = p->link;
      }
  }
}
```

Шаблонът на функцията *BFS* използва шаблона

C241-02 · Копиране: code.html
```
template <typename T>
bool member(const T& x, LList<T>& l)
{ l.iterStart();
  elem_link1<T> *p = l.iter();
  while (p && p->inf != x) p = p->link;;
  return p != NULL;
}
```

който проверява дали елементът *x* принадлежи на списъка *l* с елементи от тип *T*.
Следващата процедура реализира пълно обхождане (обхождане на всички върхове) в ширина на върховете на граф. За целта се намира разликата на

Оригинал: стр. 242

списъка, съдържащ всички върхове и списъка, съдържащ вече обходените върхове. Докато този списък е различен от празния, се взема елемент от него и се извършва обхождане в ширина, започващо от избрания връх.

C242-01 · Копиране: code.html
```
template <typename T>
void full_BFS(graph<T>& g)
{ LList<T> v = g.vertexes(),// списък с върховете на графа
           l,              // празен списък
           h = v;          // или differenceSets(v, l)
  while (!h.empty())
  { h.iterStart();
    elem_link1<T> *p = h.iter();
    BFS(p->inf, g, l);
    h = differenceSets(v, l);
```

C242-01 · продължение · Копиране: code.html
```
  }
}
```

Обхождането в ширина, започващо от даден връх, може да се приложи за проверка дали съществува път между два върха на граф, за намиране на минималния по брой върхове път от един до друг връх на граф, дължината на минималния по брой върхове път и др.

б) обхождане в дълбочина

Обхождането в дълбочина започващо от даден връх *a* на граф, означаваме го с *DFS(a)*, се реализира по следния начин:
– обработва се върхът *a*;
– маркира се, че е обработен;
– за всяка дъга <*a, b*> на графа, такава че върхът *b* е необходен се изпълнява *DFS(b)*.

Пример. Обхождането в дълбочина на графа от предходния пример, започващо от връх 1, има вида:
1, 2, 5, 9, 8, 4, 7, 3, 6, 11, 12, 10,

а обхождането в дълбочина на същия граф, започващо от връх 3, има вида:
3, 6, 11, 12, 10, 7, 8.

Резултатите са с точност до наредба на дъгите, излизащи от даден връх. В случая са дадени резултатите от изпълнението на рекурсивния шаблон на процедура, реализиращ обхождането.

Оригинал: стр. 243

Следва рекурсивна реализация на този вид обхождане за направения избор на свързано представяне на ориентиран граф. В случая обработката на върха *a* се състои в неговото извеждане. Ще отбележим също, че списъкът *l* съдържа върховете, които са обходени в последователността на обхождането им.

C243-01 · Копиране: code.html
```
template <typename T>
void DFS(const T& a, graph<T>& g, LList<T>& l)
{ l.toEnd(a);
  cout << a << " ";
  elem_link1<T>* p = g.point(a);
  p = p->link;
  while(p)
    { if(!member(p->inf, l)) DFS(p->inf, g, l);
      p = p->link;
    }
}
```

Следващата процедура извършва пълно обхождане (обхождане на всички върхове) в дълбочина на върховете на ориентирания граф *g*.

C243-02 · Копиране: code.html
```
template <typename T>
void full_DFS(graph<T>& g)
{ LList<T> v = g.vertexes(),// списък с върховете на графа
           l,              // празен списък
           h = v;          // differenceSets(v, l)
  while (!h.empty())
    { h.iterStart();
      elem_link1<T> *p = h.iter();
      DFS(p->inf, g, l);
      h = differenceSets(v, l);
    }
}
```

Ако граф е свързан, обхождането в ширина или дълбочина, започващо от произволен връх, ще обходи всички върхове на графа.

Оригинал: стр. 244

Търсене на пътища

Намиране на ацикличен (прост) път в граф

Задача 42. Да се дефинира шаблон на булева функция, която установява дали съществува ацикличен път от един до друг връх на ориентиран граф. Ако път съществува, функцията да го намира.

Обикновено се реализира следният алгоритъм за проверка дали съществува път от един връх до друг връх на граф. Съществува път от върха *a* до върха *b*, ако:
– *a* == *b* или
– съществува връх *c*, така че от *a* до *c* има дъга и има път от върха *c* до върха *b*.

Ако *a* не съвпада с *b* и е намерен връх *c*, така че от *a* до *c* има дъга, възможно е задачата за търсене дали има път от *c* до *b* да се сведе до търсене дали съществува път от *a* до *b* и да се стигне до зацикляне. Съществуват различни начини за преодоляване на зациклянето. Тъй като условието на решаваната задача изисква да се провери дали съществува ацикличен път от един до друг връх на ориентиран граф, за да избегнем зациклянето, ще използваме помощен списък *l*, в който ще записваме върховете, които са преминати в процеса на проверка дали съществува път. За следващ връх от пътя ще избираме само такъв, който не се съдържа в списъка *l*. Отначало *l* ще бъде празният списък, а ако ацикличен път съществува, в *l* ще се намери един такъв път от върха *a* до върха *b* на графа. Булевата функция *way* решава задачата като използва метода търсене с връщане назад.

При него всяко от решенията се строи стъпка по стъпка. Частта от едно решение, построена до даден момент, се нарича *частично решение*.

Методът се състои в следното:
– Конструира се едно частично решение.
– Проверява се дали частичното решение е общо (търсеното). Ако е така, решението се запомня или извежда и процесът на търсене или завършва, или продължава по същата схема, докато бъдат генерирани всички възможни решения.

В противен случай се прави опит текущото частично решение да се продължи (според условието на задачата).
Ако на някоя стъпка се окаже невъзможно ново разширяване, извършва се връщане назад към предишното частично решение и се прави нов опит то да се разшири (продължи) по друг, различен от предишния начин.
Този метод е модификация на алгоритъма за обхождане в дълбочина на ориентиран граф.

Оригинал: стр. 245

C245-01 · Копиране: code.html
```
template <typename T>
bool way(const T & a, const T & b, graph<T>& g, LList<T>& l)
{ l.toEnd(a);
  if (a == b) return true;
  elem_link1<T>* q = g.point(a);
  q = q->link;
  while (q)
    { if (!member(q->inf, l))
        { if (way(q->inf, b, g, l)) return true;
          deleteLast(l);  // връщане назад
        }
      q = q->link;
    }
  return false;
}
```

Шаблонът на функцията *way* използва шаблона на функцията *member*, описан по-горе, както и шаблона на процедурата *deleteLast*, който изключва последния елемент на непразния списък *l* с елементи от тип *T*.

C245-02 · Копиране: code.html
```
template <typename T>
void deleteLast(LList<T>& l)
{ // списъкът l не е празен
  l.iterStart(); T x;
  elem_link1<T>* p = l.iter();
  while (p->link) p = l.iter();
  l.deleteElem(p, x);
}
```

Намиране на всички ациклични (прости) пътища в граф
Когато се търсят всички ациклични пътища от един връх до друг връх на граф, трябва да се реализира пълно изчерпване. Сложността е експоненциална. Ограничението, че пътят е ацикличен е важно. В противен случай, ако е намерен път, който съдържа повтарящ се връх, пътят ще е цикличен. Тогава съществуват безкрайно много пътища, които могат да се получат от намерения път с цикъл.

Задача 43. Да се дефинира шаблон на функция, която намира и извежда на екрана всички ациклични пътища от един връх до друг връх на ориентиран граф.

Шаблонът на процедурата

C246-01 · Копиране: code.html
```
void allways(int a, int b, intGraph& g, intList& l)
```

намира всички пътища от върха *a* до върха *b* на графа *g*. Реализира търсене с връщане назад. Всеки път от върха *a* до върха *b* на графа *g* се конструира в списъка *l*, след което се извежда. Глобалната булева променлива *flag* получава стойност *true*, ако съществува път от *a* до *b* в *g*. Инициализирана е с *false*.

C246-02 · Копиране: code.html
```
bool flag = false;
template <typename T>
void allways(const T& a, const T& b, graph<T>& g,
             LList<T>& l)
{ l.toEnd(a);
  if (a == b)
    { flag = true;
      l.print();
    }
  else
    { elem_link1<T>* q = g.point(a);
      q = q->link;
      while (q)
        { if (!member(q->inf, l))
            { allways(q->inf, b, g, l);
              deleteLast(l);       // връщане назад
            }
          q = q->link;
        }
    }
}
```

Намиране на ацикличен път с минимална дължина по брой върхове

а) алгоритъм с пълно изчерпване, използващ обхождане в дълбочина

Задача 44. Дадени са ориентиран граф *G = (V, R)* и два негови върха *a* и *b*. Като се модифицира шаблонът на функцията *DFS* за обхождане в дълбочина на ориентиран граф, да се напише шаблон на функция, който намира ацикличен път с минимална по брой върхове дължина от върха *a* до върха *b*, в случай че съществува път между зададените върхове.

C246-03 · Копиране: code.html
```
bool flag = false;
int minlen = INT_MAX;
```

Оригинал: стр. 247

C247-01 · Копиране: code.html

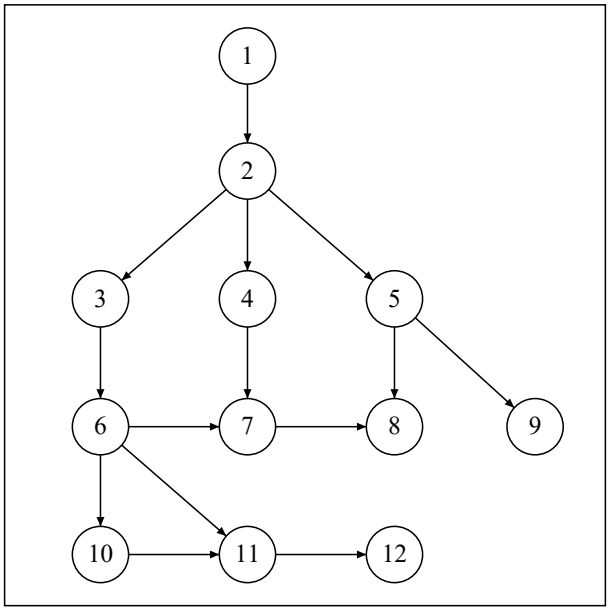
```
template <typename T>
void minway(const T& a, const T& b, graph<T>& g,
            LList<T>& l, LList<T>& minPath)
{ l.toEnd(a);
  if (a == b)
    { flag = true;
      int len = l.length();
      if (len < minlen)
        { minlen = len;
          minPath = l;
        }
    }
  else
    { elem_link1<T>* q = g.point(a);
      q = q->link;
      while (q)
        { if (!member(q->inf, l))
            { minway(q->inf, b, g, l, minPath);
              deleteLast(l);       // връщане назад
            }
          q = q->link;
        }
    }
}
```

Преди всяко изпълнение на горната програма глобалната променлива *minlen* трябва да се инициализира с константата *INT_MAX*, а списъците *l* и *minPath*, формални параметри на процедурата *minway*, трябва да са инициализирани с празния списък.

б) алгоритъм с пълно изчерпване, използващ обхождане в ширина

Задача 45. Дадени са ориентиран граф *G = (V, R)* и два негови върха *a* и *b*. Като се модифицира шаблонът на функцията *BFS* за обхождане в ширина на ориентиран граф, да се напише шаблон на функция, който намира ацикличен път с минимална по брой върхове дължина от върха *a* до върха *b*, в случай че съществува път между зададените върхове.

Ако на някоя от стъпките при изпълнение на *BFS(a)* се достигне върхът *b*, то съществува път от върха *a* до върха *b*. Този път ще е с минимална по брой върхове дължина. Например нека търсим минимален път по брой върхове от връх 2 до връх 8 на графа



В резултат от *BFS(2)* се получава:
стъпка 1: 2
стъпка 2: 5, 4, 3
стъпка 3: 9, 8, ...

и е достигнат върхът 8. Дължината на минималният по брой върхове път е 3. Намирането на минималния път може да се реализира лесно, ако на всяка стъпка заедно с обхождания връх се знае предшественикът му. С цел по-лесна реализация ще използваме структурата от данни линеен списък, представен чрез две връзки както за намиране на минималния път, така и за записване на преките предшественици на всички обходени до даден момент върхове.
Шаблонът на процедурата *minway_bfs*, дефиниран по-долу, намира минималния по брой върхове път от върха *a* до върха *b* на графа *g* в линейния списък *l*, представен с две връзки. Той реализира подобен алгоритъм на този на *BFS* за върха *a*, като го допълва с проверки за достигане на върха *b*. Поддържа помощен линеен списък *p*, представен с две връзки, в който записва непосредствените предшественици на всички обходени върхове.
Шаблонът на функцията *restore* по зададени два линейни списъка *l* и *p*, съдържащи съответно обходените до даден момент върхове и непосредствените им предшественици възстановява минималния ацикличен път от върха *a* до върха *b* на ориентирания граф *g*. Той реализира обхождане отзад напред на списъците *l* и *p*. Последният елемент на *l* е върхът *b*, а последният елемент на списъка *p* е предпоследният елемент на минималния път (означен е с *x* в шаблона). След това *x* се търси в списъка *l*. В съответната позиция в списъка *p* е предшественикът на *x*. Търсеният път се конструира в линейния списък, представен с две връзки *temp*, с който се работи като със стек.

Оригинал: стр. 249

В примера за търсене на минимален път от върха 2 до върха 8, изпълнението на *minway_bfs* преди обръщението към *restore (l, p)* намира *l = (2, 5, 4, 3, 9, 8)* и *p = (2, 2, 2, 2, 5, 5)*. Изпълнението на *restore (l, p)* намира минималния път (2, 5, 8). Последният елемент на списъка *l* е 8 – върха *b*, а последният елемент на списъка *p* е 5 – предшественика на 8. Обхожда се *l* отзад напред до намиране на 5. Съответните движения напред се извършват и в списъка *p*. 5 е намерено във втора позиция на списъка *l*. Във втора позиция на списъка *p* е 2 – предшественикът на 5. Продължава се обхождането на списъка *l* от текущия елемент до намиране на 2. Тъй като 2 е първи елемент на *l*, обхождането завършва, като *temp* съдържа резултата.

C249-01 · Копиране: code.html

```
template <typename T>
DLList<T> restore(DLList<T>& l, DLList<T>& p)
{ DLList<T> temp;
  l.iterEnd();
  elem_link2<T> *ptr1 = l.iterPred();
  p.iterEnd();
  elem_link2<T> *ptr2 = p.iterPred();
  temp.toEnd(ptr1->inf);
  while(ptr1 != NULL)
  { T x = ptr2->inf;
    do
      { ptr1 = ptr1->pred;
        ptr2 = ptr2->pred;
      } while (ptr1 && ptr1->inf != x);
    if (ptr1) temp.toStart(ptr1->inf);
  }
  return temp;
}

bool flag = false;
template <typename T>
void minway_bfs(const T& a, const T& b, graph<T>& g,
                DLList<T>& l)
{ queue<T> q; q.push(a);
  l.toEnd(a);
  DLList<T> p;
  p.toEnd(a);
  if (a == b)
    { flag = true;
```

Оригинал: стр. 250

C250-01 · Копиране: code.html
```
      return;
    }
  while (!q.empty())
    { T x; q.pop(x);
      elem_link1<T>* ptr = g.point(x);
      ptr = ptr->link;
      while (ptr)
        { if (!member(ptr->inf, l))
            { q.push(ptr->inf);
              l.toEnd(ptr->inf);
              p.toEnd(x);
              if (ptr->inf == b)
                { flag = true;
                  l = restore(l, p);
                  return;
                }
            }
          ptr = ptr->link;
        }
    }
}
```

Шаблонът на булевата функция *member*, използван в *minway_bfs*, проверява дали елемент принадлежи на линеен списък, представен с две връзки и може да се дефинира по следния начин

C250-02 · Копиране: code.html
```
template <typename T>
bool member(const T& x, DLList<T>& l)
{ l.iterStart();
  elem_link2<T> *p = l.iterSucc();
  while (p && p->inf != x) p = p->succ;;
  return p != NULL;
}
```

ЗАДАЧИ

1. Дадено е двоично дърво от тип *int*. Да се дефинира функция, която извежда отначало всички положителни, а след това всички отрицателни върхове на двоичното дърво.

Оригинал: стр. 251

2. Дадено е двоично дърво от тип *int*. Да се дефинира функция, която намира сумата на четните елементите от върховете на двоичното дърво.

3. Дадено е двоично дърво от тип *char[16]*. Низовете във върховете означават думи. Да се намери изречението, което се получава след обхождането на двоичното дърво:
а) низходящо;
б) възходящо;
в) смесено.

4. Дадено е двоично дърво от тип *char[24]*. Низовете във върховете означават имена на хора. Всяко име се състои от фамилно име, интервал и собствено име. Да се дефинира функция, която определя има ли в двоичното дърво два върха с еднакви фамилни имена.

5. Да се дефинира функция, която намира броя на върховете на *n*-тото ниво на непразното двоично дърво *d* (коренът се счита за връх от 1-во ниво).

6. Да се дефинира функция, която извежда листата на двоично дърво от тип *char*.

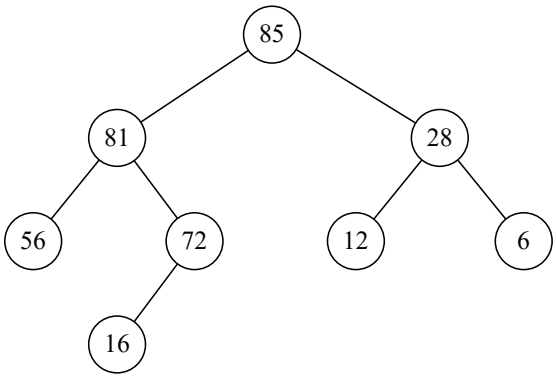
7. Да се дефинира функция, която извежда всички върхове на двоично дърво, които имат за наследници само листа.

8. Да се дефинира функция, която извежда всички върхове на дадено двоично дърво заедно с нивото на всеки връх.

9. Да се дефинира функция, която за дадено двоично дърво намира указан елемент и изпълнява операцията *f* над него.

10. Даден е резултатът от две обхождания на двоично дърво сред възходящото, низходящото и смесеното. Да се намери резултатът от третото от тези обхождания.

11. Двоично дърво е пирамида, ако за всеки негов връх е в сила, че стойността на върха е по-голяма от стойностите на всички негови наследници. Например двоичното дърво от тип *int*



е пирамида. Да се дефинира булева функция, която проверява дали дадено двоично дърво от тип *int* е пирамида.

Оригинал: стр. 252

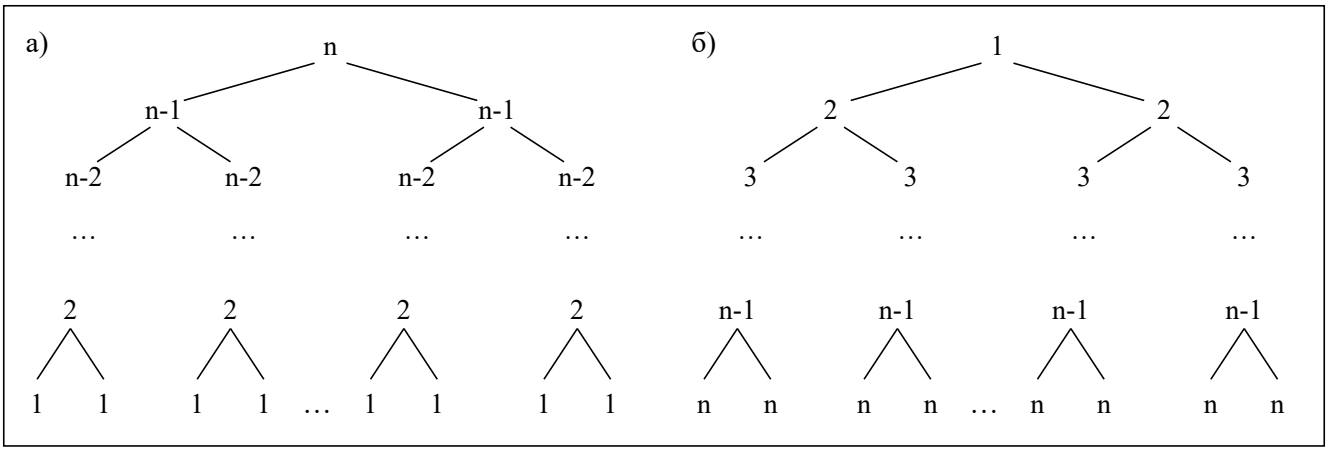
12. Да се дефинира итеративен вариант на някоя от функциите за обхождане на двоично дърво.

13. Да се дефинира шаблон на булева функция, който проверява дали двоично дърво от тип *T* е строго.

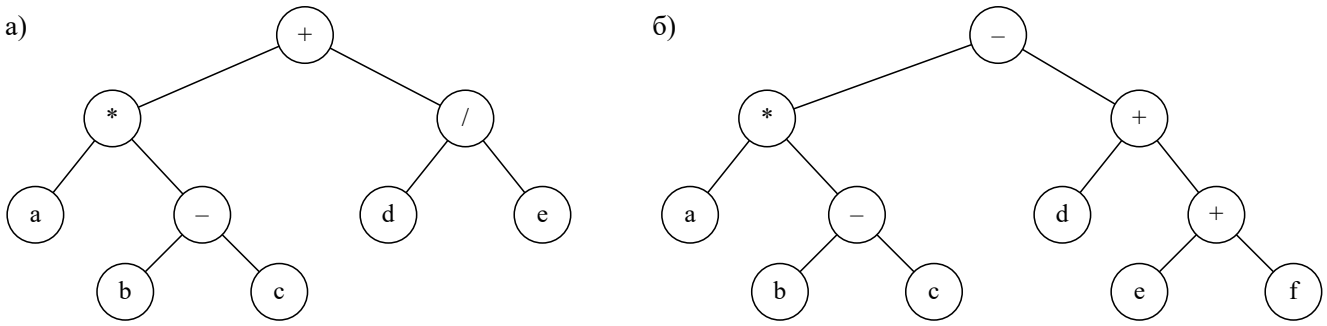
14. Да се дефинира шаблон на булева функция, който проверява дали двоично дърво от тип *T* е пълно.

15. Да се намери дължината на вътрешния път на дадено двоично дърво от тип *T*.

16. Нека *n* е положително цяло число. Да се дефинира функция *createBinTree(d, n)*, която създава следното двоично дърво от тип *int*:



17. Нека *a, b, c, d, e, f* са символи, означаващи имена на променливи от тип *int*. Посочете префиксния, инфиксния и постфиксия запис на аритметичния израз, който съответства на следното двоично дърво от тип *char*



18. Да се напише програма, която за даден аритметичен израз от вида:
<АИ> ::= <терминал> | (<АИ> <знак> <АИ>);
<знак> ::= + | – | * | /;
<терминал> ::= 0 | 1 | ... | 9

намира и извежда обратния полски запис, който съответства на израза.

19. Да се напише програма, която за аритметичен израз от вида:
<АИ> ::= <терминал> | (<АИ> <знак> <АИ>);

Оригинал: стр. 253

<знак> ::= +|–|*|/;
<терминал> ::= 0|1|...|9,

намира и извежда правия полски запис, който съответства на израза.

20. Аритметичен израз от вида
<АИ> ::= <терминал> | (<АИ> <знак> <АИ>);
<знак> ::= +|–|*|/;
<терминал> ::= 0|1|...|9

е записан без грешка в текстовия файл *f*. Да се напише програма, която създава текстов файл, съдържащ обратния полски запис, който съответства на дадения аритметичен израз.

21. Да се дефинира функция, която установява дали съществува път между два върха на дадено двоично дърво, който не минава през корена.

22. Да се дефинира функция, която установява дали съществува път между два върха на дадено двоично дърво. В случай че път съществува функцията го намира и записва в линеен списък.

23. Да се нарисува двоично наредената дърво, което се получава след последователно добавяне на следните върхове към празното двоично наредено дърво:
а) 5, 15, 30, 60, 6, 12, 18, 1, 3, 7, 2, 54, 19;
б) 10, 5, 3, 1, 2, 9, 25, 30, 20, 27, 7, 13, 99, 63;
в) 27, 63, 12, 41, 6, 80, 60, 3, 11, 64, 32, 17, 12.

24. Да се намерят двоично наредените дървета, които могат да се получат като:
а) от двоично наредената дърво от п.т. а) на предходната задача се изтрие елементът 30;
б) от двоично наредената дърво от п.т. а) на предходната задача се изтрие коренът му;
в) от двоично наредената дърво от п.т. б) на предходната задача се изтрие елементът 25;
г) от двоично наредената дърво от п.т. б) на предходната задача се изтрие коренът му;
д) от двоично наредената дърво от п.т. в) на предходната задача се изтрие елементът 63;
е) от двоично наредената дърво от п.т. в) на предходната задача се изтрие коренът му.

25. Даден е линеен списък, съдържащ реални числа. Да се напише програма, която от елементите на списъка генерира двоично наредено дърво.

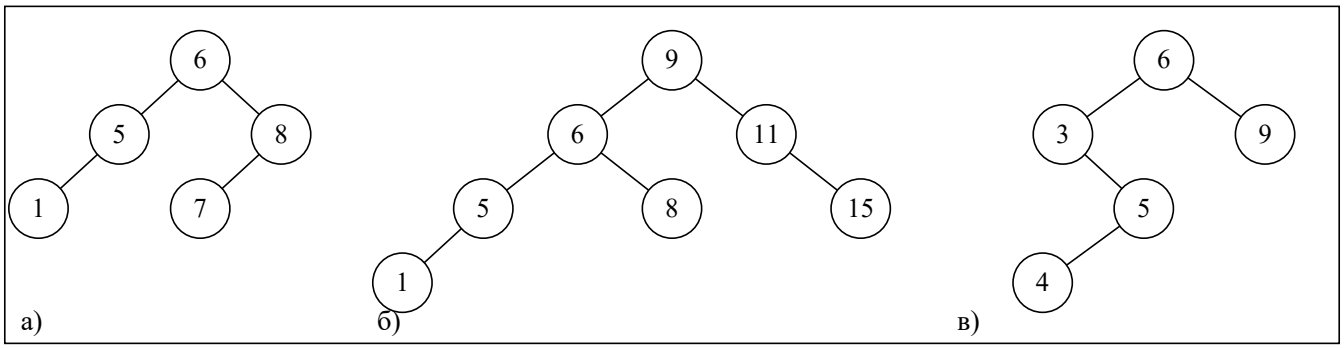
Оригинал: стр. 254

26. Даден е стек от реални числа. Да се напише програма, която като използва двоично наредено дърво сортира елементите на стека.

27. В една система от файлове е организиран каталог на всички файлове като двоично наредено дърво. Всеки възел на дървото указва името на файла и датата на последния достъп до него. Да се напише програма, която сканира каталога и изтрива от него всички файлове, последният достъп до които е бил преди определена дата.

28. Да се дефинира функция, която за дадено двоично наредено дърво от тип *int* и две цели числа *a* и *b (a < b)* извежда върховете, намиращи се в интервала *[a, b]*.

29. Кои от следните двоично наредени дървета са балансирани?



30. Кои от двоично наредените дървета от предходната задача са идеално балансирани?

31. Да се начертае балансирано двоично дърво с 12 възела, което има най-голяма височина измежду всички балансирани двоични дървета с 12 възела.

32. Като се използва алгоритъмът за създаване на двоично наредено *AVL* дърво от тип *T*, описан в т. 3.4 на настоящата глава, да се построи двоично наредено *AVL* дърво, което се получава като в празното *AVL* дърво последователно се включат елементите:
а) 20, 10, 40, 50, 90, 30, 70, 60, 80;

б) всички цели числа от интервала [1, 30];
в) всички цели числа от интервала [1, 30], но в обратен ред.

33. Като се използва алгоритъмът за изключване на елемент от двоично наредено *AVL* дърво от тип *T*, реализиран в шаблона на клас *AVL* в т. 3.4 на настоящата глава, да се изключат елементите, намиращи се в корените на *AVL* дърветата, построени в предходната задача.

34. Да се изведат на екрана елементите, с височина *k* на дадено *AVL* дърво. Височината на корена да се смята за 0, а *k* е дадено цяло число от 0 до дълбочината на *AVL* дървото.

35. Да се изведат на екрана листата на дадено *AVL* дърво.

Оригинал: стр. 255

36. Да се дефинира функция, която проверява дали дадено двоично наредено дърво е балансирано.

37. Да се дефинира функция, която проверява дали дадено двоично наредено дърво е идеално балансирано.

38. Да се построи идеално балансирано двоично наредено дърво за следната редица от върхове: 5, 8, 10, 13, 17, 20, 24, 29, 31, 36, 39, 42, 47, 58, 74, 87, 92, 99, 100.

39. Линеен списък от тип *int* е сортиран във възходящ ред. Да се дефинира функция, която конструира идеално балансирано двоично наредено дърво, състоящо се от елементите на списъка.

40. Да се дефинира функция, която по дадено двоично дърво от тип *int* построява идеално балансирано двоично наредено дърво, съдържащо същите елементи във върховете. За целта може да се построи помощен линеен списък, съдържащ върховете на двоичното дърво сортирани във възходящ ред.

41. Да се дефинира АТД, който реализира верижното представяне на двоично дърво, показано в т. 3.2.1.

42. Да се дефинира АТД, който реализира дърво от тип *T*, представено както е показано на фиг. 3.3. АТД да позволява:
• създаване на дърво от тип *T* по зададени върхове;
• създаване на дърво от тип *T* по зададени корен и поддървета;
• достъп до корена;
• достъп до поддърво, с указан пореден номер в свързания списък, реализиращ наследниците;
• включване на поддърво;
• изключване на поддърво;
• извеждане на дърво от тип *T*.

43. Да се дефинира шаблон на процедура, която реализира предхождащо обхождане на дадено дърво от тип *T*.

44. Да се дефинира шаблон на процедура, която реализира последващо обхождане на дадено дърво от тип *T*.

45. Дадено е дърво от тип *int*. Да се дефинира функция, която намира стойността на минималния връх на дървото.

46. Дадено е дърво от тип *int*. Да се дефинира функция, която установява дали дадено цяло число съвпада със стойността на връх на даденото дърво.

47. Дадено е дърво от тип *int*. Да се дефинира функция, която проверява дали всеки връх на дървото има не повече от два наследника.

48. Дадено е дърво от тип *int*. Да се дефинира функция, която проверява дали дървото е двоично наредено.

Оригинал: стр. 256

49. Дадено е дърво от тип *int*. Да се дефинира функция, която проверява дали дървото е пълно двоично.

50. Дадено е дърво от тип *char*. Да се дефинира функция, която намира броя на върховете на всяко от неговите поддървета.

51. Дадено е дърво от тип *char*. Да се дефинира функция, която намира броя на:
а) вътрешните върхове на дървото;
б) листата на дървото.

52. Дадено е дърво от тип *char*. Да се дефинира функция, която извежда:
– вътрешните върхове на дървото;
– листата на дървото;
– всички върхове и нивата им.

53. Дадено е дърво от тип *char* и цяло число $k > -1$. Да се дефинира функция, която записва в линеен списък върховете от ниво k на дървото. Да се приеме нивото на корена за 0.

54. Родословно дърво съдържа всички деца на дадено лице, техните деца и т.н. Дадено е родословно дърво. Да се дефинира функция, която:
– добавя нов потомък в дървото;
– намира средната възраст на всички лица, включени в дървото;
– намира най-младото лице в родословното дърво.

55. Родословно дърво съдържа всички деца на дадено лице, техните деца и т.н. Дадени са две родословни дървета. Да се дефинира функция, която проверява дали дърветата са равни.

56. Родословно дърво съдържа всички деца на дадено лице, техните деца и т.н. Дадени са две родословни дървета. Да се дефинира функция, която проверява дали съществува лице, което се съдържа и в двете родословни дървета.

57. Да се дефинира функция, която установява дали съществува път между два върха на дадено дърво от тип *T*, който не минава през корена.

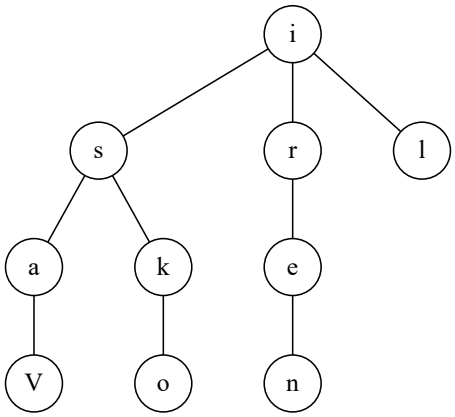
58. Да се дефинира функция, която установява дали съществува път между два върха на дадено дърво от тип *T* с дължина в интервала *[m, n]*. В случай че път съществува го намира и записва в линеен списък.

59. Да се намери дължината на вътрешния път на дадено дърво от тип *T*.

60. Дадени са дърво от тип *char* и символен низ, представящ дума. Да се дефинира функция, която проверява дали дадената дума може да се прочете в дървото след последователно преминаване през върховете му.

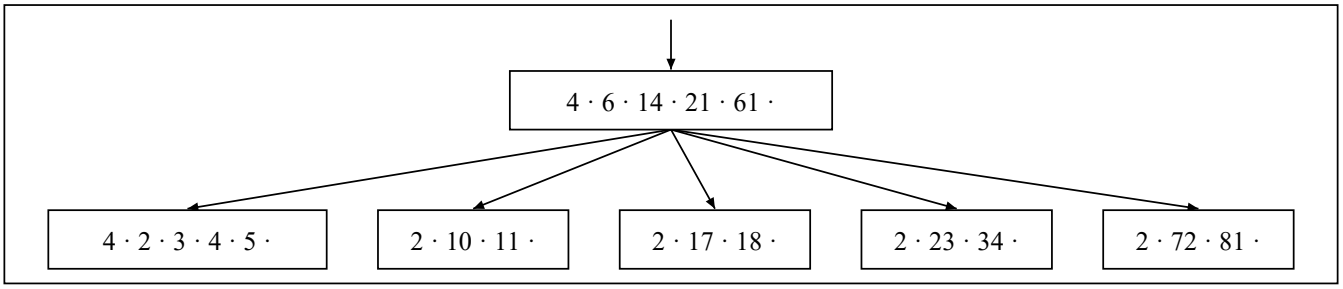
Пример. Името „*Vasko*“ може да се прочете в дървото по-долу. Думата „*sila*“ не може да се прочете.

Оригинал: стр. 257



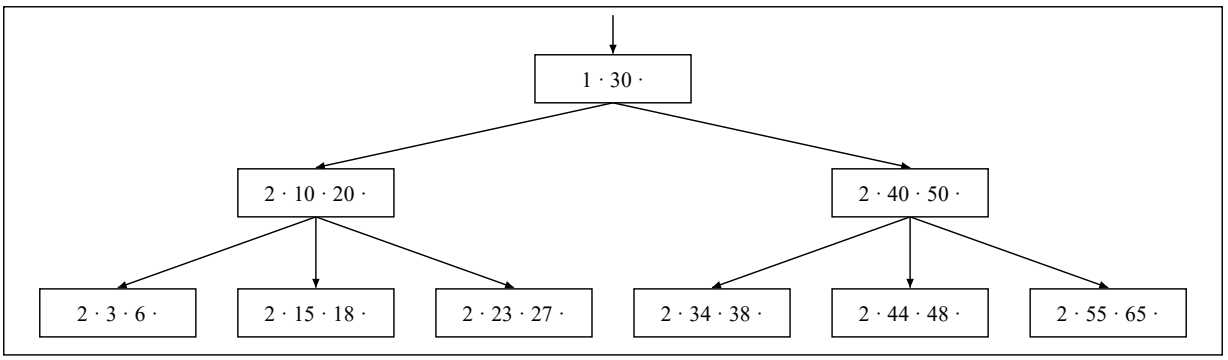
61. В празното *B*-дърво последователно да се включат следните елементи:
а) 25, 10, 60, 90, 70, 15, 5, 1, 9, 80, 75, 16, 18, 24, 12, 34, 66, 48, 59;
б) целите числа от интервала [1, 30];
в) целите числа от интервала [1, 30], но в обратен ред.

62. От *B*-дървото



последователно да се изключат елементите: 21, 81, 61, 10, 14, 34, 72, 23, 11, 17, 18, 4, 5, 6, 3, 2.

63. В *B*-дървото



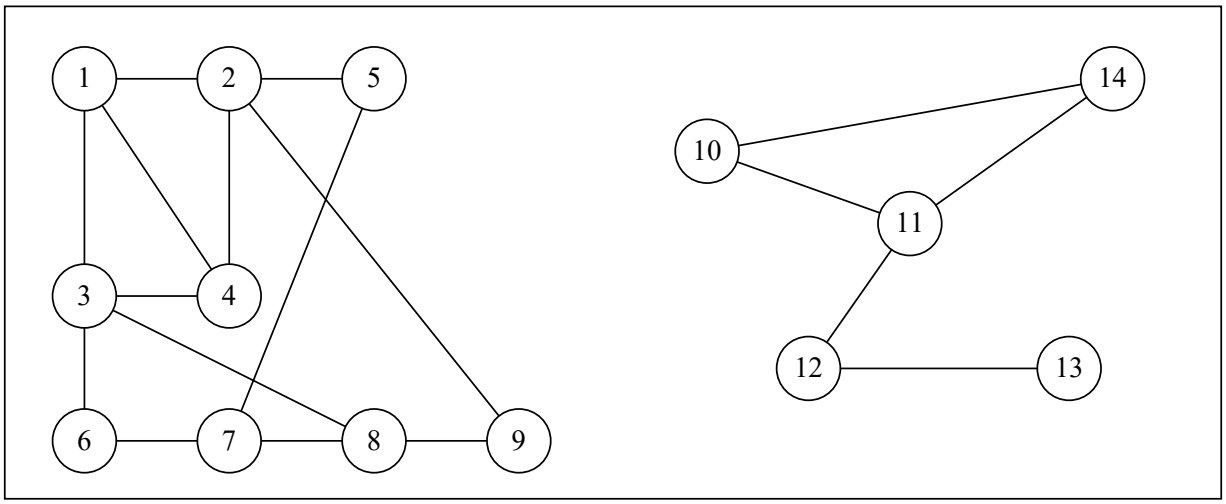
да се включат последователно елементите: 28, 19, 7, 29, 68, 70, 72, 49, 51, 53, 1, 4, 8, 12, 14, 13, 46, 47, 25, 21.

64. Дадени са ориентираните графи *g* и *h* от тип *int*. Да се напише програма, която установява дали графът *g* е равен на графа *h* (Два графа са равни, ако са равни множествата от върховете и множествата от ребрата им).

65. Да се напише програма, която:
а) създава ориентираните графи *g1, g2* и *g3*;
б) проверява дали графът *g3* е обединение на графите *g1* и *g2*;
в) проверява дали графът *g3* е сечение на графите *g1* и *g2*.

66. Да се напише програма, която създава ориентираните графи *g1, g2, g3* и g4 и намира и извежда:
а) g1 ∪ g2 ∪ g3 ∪ g4;
б) g1 ∩ g2 ∩ g3 ∩ g4;
в) (g1 ∪ g2) ∩ (g3 ∪ g4);
г) (g1 ∩ g2) ∪ (g3 ∩ g4).

67. Да се обходи в ширина неориентираният граф



като се започне от връх:
а) 1;
б) 4;
в) 9;
г) 11.

68. Да се обходи в дълбочина неориентирания граф от по-горната задача като се започне от връх:
а) 1;
б) 9;
в) 6;
г) 14.

69. Да се напише програма, която определя всички върхове на даден ориентиран граф, които са недостижими от даден връх.

70. Да се провери дали ориентиран граф съдържа цикъл.

71. Да се напише програма, която намира всички цикли на даден връх на ориентиран граф.

72. Да се напише програма, която намира най-късия по брой върхове цикъл в даден ориентиран граф.

73. Даден е ориентиран граф, във върховете на който са разположени букви. Да се определи дали дадена дума може да се прочете след последователно преминаване през върховете на графа (т.е.

дали думата е път в графа с начало първата буква и край – последната буква на думата).

Оригинал: стр. 259

74. Да се напише програма, която намира всички прости пътища между два върха на даден ориентиран граф. За целта да се използва алгоритъмът за обхождане в ширина.

75. Да се напише програма, която намира всички прости пътища между два върха на даден ориентиран граф с дължина, измервана в брой на преминатите върхове, равна на k $(k \geq 3)$.

76. Дадени са ориентиран претеглен граф $G = (V, R)$ и два негови върха a и b. Теглата на графа са реални числа. Като се използва алгоритъмът за обхождане в дълбочина, да се напише шаблон на функция, който намира ацикличен път с минимална дължина по теглата на ребрата от връх a до връх b, както и дължината му, в случай че съществува път между зададените върхове.

77. Дадени са ориентиран претеглен граф $G = (V, R)$ и два негови върха a и b. Теглата на графа са реални числа. Като се използва алгоритъмът за обхождане в ширина да се напише шаблон на функция, който намира ацикличен път с минимална дължина по теглата на ребрата от връх a до връх b, както и дължината му, в случай че съществува път между зададените върхове.

78. Известно е, че неориентираният граф g не е дърво. Да се провери, може ли да се изключи един връх на графа (заедно със съответните ребра), така че да се получи дърво.

79. Да се напише програма, която намира броя на изолираните точки на даден ориентиран граф.

80. Дадени са ориентиран граф g и негов връх a. От графа да се изключат всички негови върхове, от които е недостижим върхът a.

81. Дадени са ориентиран граф g и два негови върха a и b. Да се напише програма, която проверява дали съществува връх на g, който е различен от върховете a и b и принадлежи на всеки път от a до b.

82. Да се напише програма, която проверява дали даден ориентиран граф е Хамилтонов.

83. Дадени са ориентиран граф g, множество M, съдържащо някои върхове на g и два върха a и b на g, които не принадлежат на M. Да се напише програма, която намира прост път, свързващ върха a с върха b и несъдържащ върхове от множеството M, ако такъв съществува.

84. Дадени са ориентиран граф $G=(V, R)$ с n върха и множество от m различни двойки от върхове на графа $\{(a_1, b_1), (a_2, b_2), \ldots, (a_m, b_m)\}$. Да се напише програма, която проверява дали съществуват m непресичащи се пътища с начало a_i и край b_i $(1 \leq i \leq m)$ в графа G. Два пътя са непресичащи се, ако нямат общи върхове.

85. Диаметър на граф – това е максималното число n, такова че най-краткият прост път между всевъзможните двойки негови върхове съдържа поне n върха. Да се напише програма, която намира диаметъра на даден граф.

Оригинал: стр. 260

ПРОЕКТИ

1. АТД B-дърво

Да се дефинира шаблон на АТД, реализиращ B-дърво от ред m с елементи от тип T. B-дървото да се представи чрез указател към възел, дефиниран чрез следния шаблон:

C260-01 · Копиране: code.html

```
template <typename T, typename m>
struct elem
{ int n;       // текущ брой на елементите във възела
               // ((m-1)/2 <= n <= m-1)
  T c[m-1];    // масив, съдържащ елементите на възела
               // (използват се само първите n от тях)
  elem* p[m];  // масив от указателите към поддърветата
               // на възела (използват се n+1 от тях)
};
```

Основни операции на АТД да са: включване на елемент, изключване на елемент, търсене на елемент, създаване на *B*-дърво от ред *m*, извеждане на *B*-дърво от ред *m*.

2. Алгоритъм на Хъфман за компресия на данни

Идеята да се предава информация по възможно най-икономичен начин е древна. За оптимално кодиране са разработени специални системи, каквито са например стенографската, Морзовата азбука, азбуката за хора с нарушен слух и др. Днес кодирането е силно развит дял на дискретната математика и теоретичната информатика. С навлизането на компютрите се появява възможност автоматично сравнително бързо да се „превежда“ даден поток от информация на по-икономична азбука и обратно. Бързо намират приложение алгоритмите за компресиране на информация, а те от своя страна се доразработват и оптимизират, за да навлязат във всекидневна употреба. Появиха се компресиращи програми (*ARJ, ZIP, PKZIP, WinZIP, RAR, ACE* и др.), които позволяват данните да се преобразуват, така че да се намали необходимото дисково пространство. Появиха се и технологии за компресиране на звук (*MP3, OGG*), графика (*GIF, JPEG*), видео (*MPEG, AVI*). Съществуват 6 основни типа компресиране: кодиране на последователности, статистически методи, речникови методи, вълнови методи, фрактални методи и адаптивно компресиране.

Оригинал: стр. 261

Алгоритъм на Хъфман за компресиране

Алгоритъмът на Хъфман, разгледан тук е сравнително прост алгоритъм за компресия, принадлежащ към статистическите методи. При него се предполага, че е даден краен поток от числа в някакъв предварително фиксиран интервал. Ще считаме, че се използват символи, кодирани с *ASCII* код. Алгоритъмът се базира на простата идея, че най-често срещаните символи в поредицата трябва да се записват с най-малък брой битове. Алгоритъмът построява нова азбука и след това превежда информацията в новата азбука. Кодирането е обратимо, т.е. по кодираната последователност може да се декомпресира – да се намери първоначалната редица.

Построяване на кодиращо дърво на Хъфман

Нека трябва да се компресира даден низ от символи. Искаме да построим двоично дърво, от което ще определим азбука за компресиране.

Алгоритъмът за построяване на кодиращо дърво преминава през следните стъпки:

1. Създава се честотна таблица на низа – за всеки символ се записва броят на срещанията му.

2. Нека различните символи в низа са *n* на брой. Създават се *n* дървета от по един елемент, където всяко дърво съдържа символ и броя на срещанията му.

3. Намират се две дървета, които в корените си имат най-малко число. Обединяват се дърветата в ново дърво, като в корена се записва сумата от стойностите в двете намерени дървета.

4. Повтаря се стъпка 3, докато не се получи само едно дърво – кодиращото дърво на Хъфман за дадения низ.

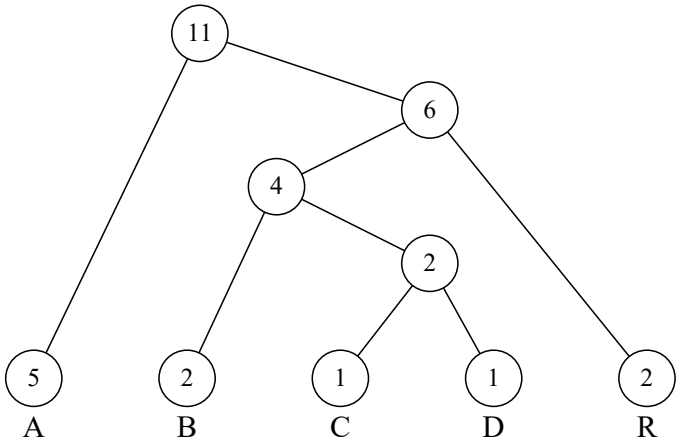
Така построено дървото е двоично и има точно *n* на брой листа, като на всяко листо отговаря един символ от честотната таблица. По начина на построение се вижда, че по-често срещаните символи се намират по-близо до корена от по-рядко срещаните. Това се вижда и в примера, даден по-долу.

Пример. Честотната таблица за низа „ABRACADABRA“ е

Символ	Брой срещания
A	5
B	2
C	1
D	1
R	2

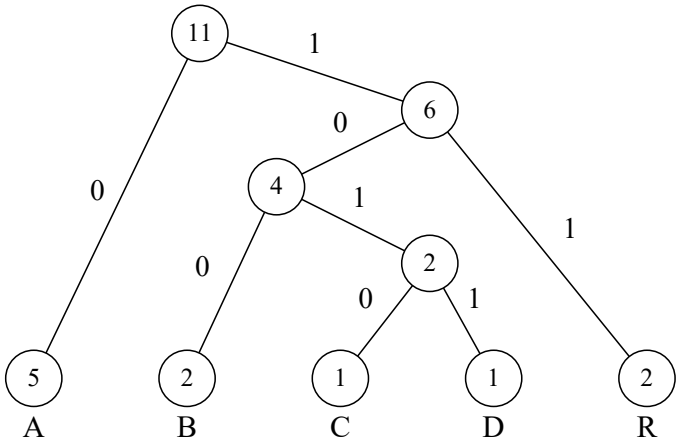
Оригинал: стр. 262

Кодиращото дърво се строи по следния начин



Построяване на азбуката по кодиращото дърво на Хъфман
На всяко ребро на дървото се съпоставя двоична цифра 0 или 1: 0 за ляв клон, 1 за десен клон. Така на всеки път от корена до някое листо отговаря двоичен низ. Тъй като всяко листо е символ от низа, на всеки символ може да се съпостави двоичната последователност, която съответства на пътя от корена до листото на символа. Тъй като най-често срещаните символи са най-близко до корена, на тях ще отговарят най-къси последователности. Обратно – на рядко срещаните символи ще съответстват дълги последователности.

Пример. Продължаваме примера от по-горе. Дървото, отбелязано с 0 и 1 изглежда по следния начин



Таблицата за кодиране е

Символ	Код
A	0
B	100
C	1010
D	1011
R	11

Оригинал: стр. 263

След като се получи таблицата за кодиране, се извършва кодиране на низа – всеки символ се замества с неговия код. Така се получава редица от 0 и 1. Ако се разбие тя на блокове по 8 бита, може да се получи и резултат от байтове.

Пример.

C263-01 · Копиране: code.html
```
ABRACADABRA ->
0 100 11 0 1010 0 1011 0 100 11 0 ->
01001101010010110100110 ->
01001101 01001011 0100110 ->
77 75 38
```

От 11 символа (байта) = 8*11 бита = 88 бита се получиха 23 бита компресирана информация, т.е. около 26% от първоначалния обем.

Декомпресиране на компресирана информация

Декомпресирането на данните се осъществява лесно, при условие че е дадено кодиращото дърво на Хъфман. Обхождат се едновременно двоичният низ (кодът) и кодиращото дърво. Всеки път при срещане на 0 се тръгва наляво, а при срещане на 1 – надясно. Когато се достигне листо, записва се съответният символ от листото и отново се обхожда кодиращото дърво, започвайки от корена. Така стъпка след стъпка се получава първоначалният низ.

Пример.

C263-02 · Копиране: code.html

```
0100110101001011010011 0 ->
0 100 11 0 1010 0 1011 0 100 11 0 ->
A B R A C A D A B R A
```

Да се напише програма, която въвежда низ от *ASCII* символи от клавиатурата (или от файл), построява честотната таблица и кодиращото дърво на Хъфман за въведения низ и извежда двоичната последователност, която кодира дадения символен низ. Да се пресметне степента на компресия (отношението на броя на битовете на компресираната и декомпресираната последователност). По кодиращото дърво на Хъфман да се възстанови оригиналният низ.

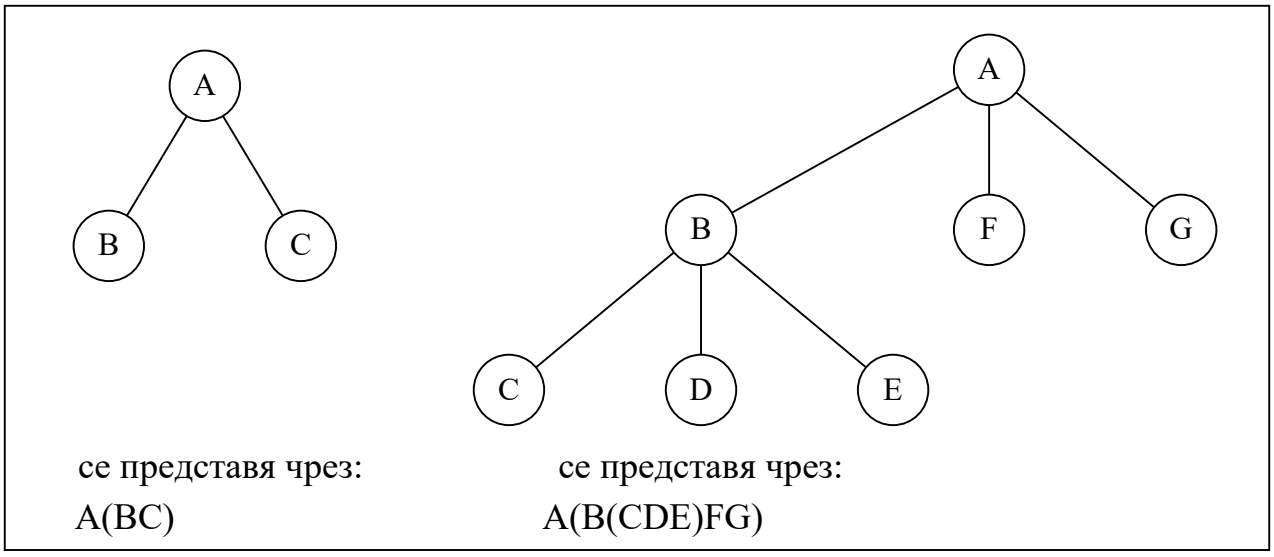
Допълнителни задачи:
• След получаване на кода, да се изведат числа от 0 до 255, получени от разбиване на кода на блокове по 8 бита.
• След като създаде кодиращото дърво на Хъфман, програмата да пита за компресиран низ от нули и единици и да го декомпресира.

Да се сравни реализираната архивираща програма със *ZIP* и *RAR* програмите – при едни и същи входни файлове, какъв е резултатът от работата на трите програми?

Оригинал: стр. 264

3. Представяне на дърво чрез скоби

Структурата от данни „дърво“ може да се представи чрез скоби по следния начин. След името на върха, оградени в скоби, се записват имената на всички негови наследници. Това описание се прилага рекурсивно към всеки от върховете на дървото. Например



Да се напише програма, която въвежда от текстов файл „дървета“, представени чрез скоби. Всяко дърво е записано на отделна линия във файла. Програмата да извежда на екрана всяко дърво, като го обхожда по следния начин:

а) *възходящо обхождане* (1-во, 2-ро, ..., n-то поддърво, корен);
б) *низходящо обхождане* (корен, 1-во, 2-ро, ..., n-то поддърво).
Пример за обхождането от случай а)

Дърво	Възходящо обхождане
a(b(c)d)	cbda
a(b(c))	cba
a(b(cde)fg)	cdebfga
a(b(cd)e(f(g)h(ijk(lm))))	cdbgfijlmkhea

4. Информационна система, базирана на идеално балансирано двоично наредено дърво

Да се напише програма, която реализира информационна система, извършваща описаните по-долу действия над база от данни (БД). БД се състои от записи, съдържащи: *име* (символен низ с дължина до 31 знака), *среден успех* (реално число) и *телефонен номер* (цяло число). Имената на студентите в базата трябва да са уникални, тъй като ще се използват като ключове.

Оригинал: стр. 265

Програмата да позволява:

(1) Прехвърляне на записите на текстов файл в идеално балансирано двоично наредено дърво (записите във върховете на дървото са сортирани по името на студент).

(2) Търсене на средния успех на студент с указано име.

(3) Търсене на телефонния номер на студент с указано име.

(4) Включване на нови записи в двоично наредената дърво.

(5) Актуализиране на средния успех на студент с указано име.

(6) Актуализиране на телефонния номер на студент с указано име.

(7) Изтриване на запис от двоично наредената дърво.

(8) Прехвърляне на записите на двоично наредената дърво във текстов файл.

(9) Извеждане на екрана на всички записи на БД, сортирани по азбучен ред на имената на студентите.

Записите на БД се съхраняват в текстов файл, подредени по азбучен ред на имената на студентите. Отначало файлът може да бъде празен. Като се използват функциите (4) и (8), се създава непразен текстов файл, съдържащ БД, подредена по азбучен ред на имената на студентите. По време на добавянето и изтриването на записи е възможно двоично наредената дърво да загуби идеалната си балансираност, но след функциите (8) и (1) двоично наредената дърво трябва да стане идеално балансирано.

5. Ойлерови цикли

Даден е неориентиран мултиграф, представен чрез списък на инцидентност. Да се напише програма, която:
– проверява дали мултиграфът е Ойлеров;
– намира Ойлеров цикъл, в случай че мултиграфът е Ойлеров;
– намира Ойлеров път.

6. Задача за търговския пътник

Търговски пътник трябва да посети всички селища на дадена област, като направи минимални разходи. За целта той трябва да тръгне от даден град и да се върне в същия град, като посети всеки от останалите селища точно по веднъж.
Да се напише програма, която намира маршрута, по който търговският пътник трябва да премине.
Да се извършат експерименти с направената реализация за „големи“ графи (графи, съдържащи повече от 40 върха).

Оригинал: стр. 266

Глава 4

АЛГОРИТМИ ЗА СОРТИРАНЕ И ТЪРСЕНЕ

В тази глава ще разгледаме някои методи за сортиране и търсене. Ще ги реализираме за едномерни масиви и файлове с пряка организация на достъпа.
За оценка на ефективността на методите ще използваме анализа *O*-голямо.

4.1. АЛГОРИТМИ ЗА СОРТИРАНЕ

Нека за типа *T* на еднотипни данни е въведена някаква наредба. Процесът на пренареждане на тези данни в определен ред се нарича *сортиране*. Използва се при организиране на справочници, речници, телефонни указатели и в други структури, в които се налага бързо търсене на различни обекти.

Формално процесът на сортиране може да се определи по следния начин. Дадени са множество с елементи от тип *T*
$a_0, a_1, ..., a_{n-1}$
и функция *s*, дефинирана над това множество. Под сортиране на елементите $a_0, a_1, \ldots, a_{n-1}$ се разбира пренареждането им в ред
$a_{i_0}, a_{i_1}, ..., a_{i_{n-1}}$
така, че да е в сила
$s(a_{i_0}) < s(a_{i_1}) < \ldots < s(a_{i_{n-1}})$.

Елементите на множеството, което ще се сортира най-често са записи, реализирани като обекти на клас, а функцията *s* е зададена чрез операторна функция на класа.

4.1.1. Метод на пряката селекция

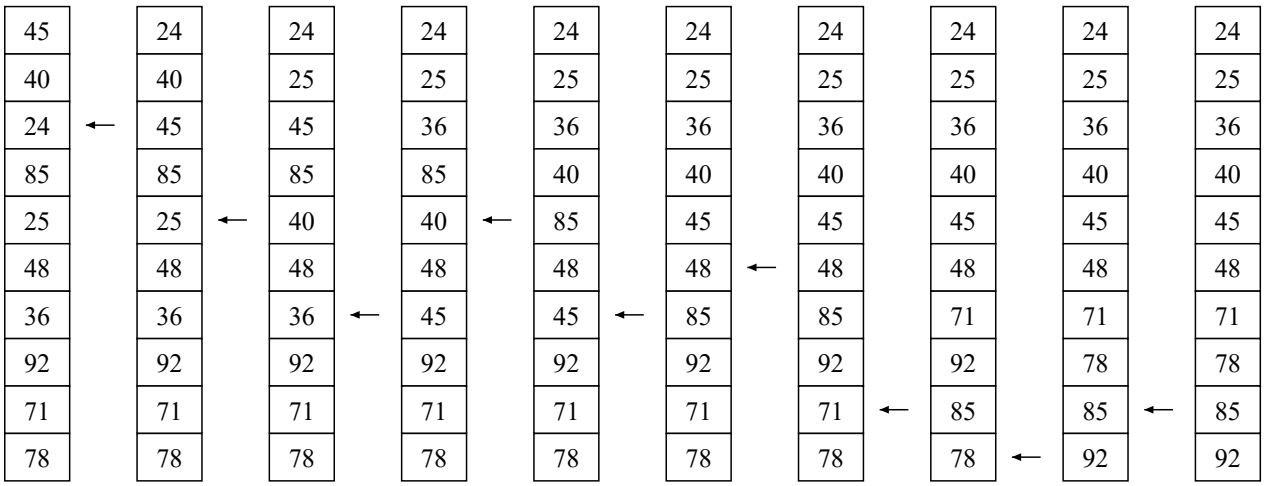
Нарича се още *сортиране чрез размяна*. Ще сортираме във възходящ ред редицата
$a_0, a_1, \ldots, a_{n-1}$

Методът може да се опише по следния начин:
За всяко *i, i = 0, 1, ..., n-2* се разглежда редицата
$a_i, a_{i+1}, \ldots, a_{n-1}$
и се извършват следните действия:

Оригинал: стр. 267

– намира се *k*, така че $a_k = min\{a_i, a_{i+1}, ..., a_{n-1}\}$;
– разменят се стойностите на a_k и a_i.

Пример. Методът е илюстриран при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78 на фиг. 4.1. Стрелката сочи минималния елемент на неоцветената част на редицата. Оцветената част е сортираната подредица.



Фигура 4.1. Пример за сортиране на редица по метода на пряката селекция

Реализация на метода за масиви

Задача 46. Да се дефинира шаблон на процедура, реализиращ метода на пряката селекция за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

C267-01 · Копиране: code.html
```
template <typename T>
void selectSort(T* a, int n)
{ int i, j, k;
  T min;
  for (i = 0; i < n-1; i++)
    { k = i; min = a[k];
      for (j = i+1; j < n; j++)
        if (a[j] < min)
          { k = j;
            min = a[k];
          }
      a[k] = a[i]; a[i] = min;
    }
}
```

Оригинал: стр. 268

Реализация на метода за файлове

Задача 47. Да се дефинира шаблон на процедура, реализиращ метода на пряката селекция за сортиране във възходящ ред на компонентите на файл с пряка организация. Компонентите на файла са от тип *T*.

Шаблонът на процедурата *void selectSort(fstream& f, T& dummy)* решава задачата. Параметърът *dummy* служи за определяне на типа на елементите на файла *f*.

C268-01 · Копиране: code.html
```
template <typename T>
void selectSort(fstream& f, T& dummy)
{ int i, j, k;
  T min, a, b;
```

C268-01 · продължение · Копиране: code.html

```
  // намиране на броя на елементите на f
  f.seekg(0, ios::end);
  long n = f.tellg()/sizeof(T);
  for (i = 0; i < n-1; i++)
    { k = i; f.seekg(k*sizeof(T));
      f.read((char*)&min,sizeof(T));
      for (j = i+1; j < n; j++)
        { f.seekg(j*sizeof(T));
          f.read((char*)&a, sizeof(T));
          if (a < min)
            { k = j;
              min = a;
            }
        }
      f.seekg(i*sizeof(T)); f.read((char*)&a, sizeof(T));
      f.seekg(k*sizeof(T)); f.read((char*)&b, sizeof(T));
      f.seekp(k*sizeof(T)); f.write((const char*)&a, sizeof(T));
      f.seekp(i*sizeof(T)); f.write((const char*)&b, sizeof(T));
    }
  f.close();
}
```

Оценка на метода на пряката селекция

За да намерим сложността на метода, ще използваме реализацията му за масиви. В тялото на процедурата има оператор за цикъл *for* с управляваща променлива *i*, променяща се от 0 до *n-2*, където *n* е броят на елементите на масива, който

Оригинал: стр. 269

сортираме. В тялото на този оператор за цикъл има друг оператор за цикъл *for* с управляваща променлива *j* и няколко оператора за присвояване. Вложеният оператор за цикъл се изпълнява *n-i-1* пъти. Следователно времето за изпълнение на процедурата е от порядъка на

$$\sum_{i=0}^{n-2}(n-i-1) = n-1 + (n-2) + (n-3) + \ldots + 1 = n(n-1)/2,$$

т.е. пропорционално е на квадрата на броя на елементите *n* на редицата. Заради това методът е удобен за сортиране на редици с относително малък брой елементи. Ще отбележим също, че при този метод за сортиране броят на сравненията не зависи от първоначалната наредба на елементите на редицата.

Задача 48. Да се напише програма, която създава файл от цели числа с пряка организация на достъпна. Сортира във възходящ ред по метода на пряката селекция елементите му, след което извежда файла.

Програма *zad48.cpp* решава задачата. Тя дефинира два шаблона на процедури за създаване и за извеждане на елементите на файл с пряка организация на достъпа. За определяне на типа на елементите на файла се използва параметърът *dummy*. Шаблонът на процедурата *selectSort* е записан във файла *selectSort.cpp*.

C269-01 · Копиране: code.html

```
// Program zad48.cpp
#include <iostream>
#include <fstream>
using namespace std;
#include "selectSort.cpp"

template <typename T>
void readT(T& x)
{ cin >> x;
}

template <typename T>
void createFile(fstream& f, T& dummy)
{ T x;
  readT(x);
  while (cin)
```

Оригинал: стр. 270

C270-01 · Копиране: code.html

```
    { f.write((const char*)&x, sizeof(T));
      readT(x);
    }
  f.close();
}
template <typename T>
void writeT(T x)
{ cout << x << " ";
}
template <typename T>
void writeFile(fstream& f, T& dummy)
{ T x;
  cout << "Sequence\n";
  while (f.read((char*)&x, sizeof(T)))
    writeT(x);
  cout << endl;
  f.close();
}

int main()
{ fstream f("pom.dat", ios::out|ios::binary);
  int dummy;
  createFile(f, dummy);
  f.open("pom.dat", ios::in|ios::out|ios::binary);
  selectSort(f, dummy);
  f.open("pom.dat", ios::in|ios::out|ios::binary);
  writeFile(f, dummy);
  return 0;
}
```

4.1.2. Методи за сортиране чрез пряка размяна

Методите за сортиране на мехурчето и чрез клатене се наричат още методи за *сортиране чрез пряка размяна*.

Метод на мехурчето

Този метод е сред най-често използваните от програмистите заради лесното му реализиране. Има няколко варианта. В тази глава ще предложим вариант, който използва флаг.

Оригинал: стр. 271

Нека да сортираме във възходящ ред редицата
$a_0, a_1, \ldots, a_{n-1}$
Методът може да се опише по следния начин:

1. Нека *right = n-1*

2. За редицата
$a_0, a_1, \ldots, a_{right}$
последователно се сравняват всеки два съседни елемента a_i и a_{i+1}.

Ако $a_i > a_{i+1}$, двата елемента си разменят местата и позицията *i* на размяната се запомня в променливата *k*;
Ако $a_i <= a_{i+1}$, двата елемента не си разменят местата.

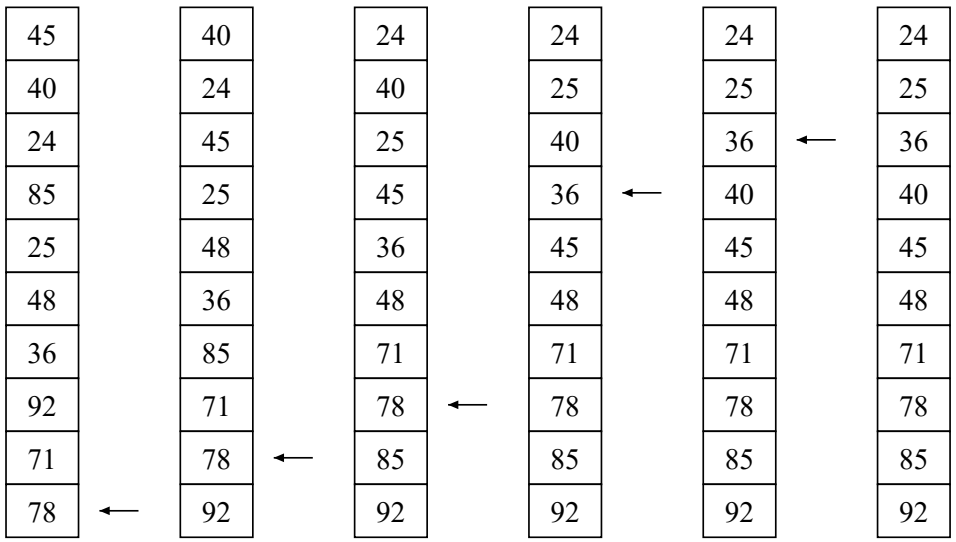
Процесът продължава до изчерпване на редицата. Ако при това сканиране след размяната на елементите от позиции *i* и *i+1* не е извършена друга размяна, елементите от позициите: *k+1, k+2, ..., n-1* са правилно подредени.

3. *right = k*

4. Ако *right > 0*, действията от т. 2 и т. 3 се повтарят.

5. Ако *right = 0*, редицата е сортирана.

Пример. Методът е илюстриран графично на фиг. 4.2. при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. Стрелката сочи елемента a_{right} на неоцветената част на редицата. Оцветената част представя сортираната част от редицата.



Фигура 4.2. Пример за сортиране на редица по метода на мехурчето

Реализация на метода за масиви

Задача 49. Да се дефинира шаблон на процедура, реализиращ метода на мехурчето за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

C272-01 · Копиране: code.html

```
template <typename T>
void swap(T& a, T& b)
{ T x = a; a = b; b = x;
}

template <typename T>
void bubbleSort(T* a, int n)
{ int i, k, right = n-1;
  while (right > 0)
    { k = 0;
      for (i = 0; i < right; i++)
        if (a[i+1] < a[i])
          { swap(a[i], a[i+1]);
            k = i;
          }
      right = k;
    }
}
```

Реализация на метода за файлове

Задача 50. Да се дефинира шаблон на процедура, реализиращ метода на мехурчето за сортиране във възходящ ред на компонентите на файл с пряка организация. Компонентите на файла са от тип *T*.

За реализиране на метода е дефиниран помощен шаблон на функция *swap*, реализиращ размяна на *i*-я и *j*-я елементи на файла *f*.

C272-02 · Копиране: code.html

```
template <typename T>
void swap(fstream& f, int i, int j, T& dummy)
{ T x, y;
  f.seekg(i*sizeof(T));
  f.read((char*)&x, sizeof(T));
  f.seekg(j*sizeof(T));
  f.read((char*)&y, sizeof(T));
```

Оригинал: стр. 273

C273-01 · Копиране: code.html

```
  f.seekp(i*sizeof(T));
  f.write((const char*)&y, sizeof(T));
  f.seekp(j*sizeof(T));
  f.write((const char*)&x, sizeof(T));
}

template <typename T>
void bubbleSort(fstream &f, T& dummy)
{ // намиране на броя на елементите на файла
  f.seekg(0, ios::end);
  int n = f.tellg()/sizeof(T);
  int i, k, right = n-1;
  T a, b;
  while (right > 0)
    { k = 0;
      for (i = 0; i < right; i++)
        { f.seekg(i*sizeof(T));
          f.read((char*)&a, sizeof(T));
          f.read((char*)&b, sizeof(T));
          if (a > b)
            { swap(f, i, i+1, dummy);
              k = i;
            }
        }
      right = k;
    }
  f.close();
}
```

Оценка на метода на мехурчето

За да оценим сложността на метода ще използваме реализацията му за масиви. В тялото на процедурата има оператор за цикъл *while*, изпълняващ се в най-лошия случай *n-1* пъти (за *right* променящ се от *n-1* до 1, където *n* е броят на елементите на масива, който се сортира. В тялото на този оператор за цикъл има друг оператор за цикъл *for* с управляваща променлива *i* и няколко оператора за присвояване. Операторът за цикъл *for* се изпълнява *right* пъти. Следователно времето за изпълнение на *bubbleSort* е от порядъка на

Оригинал: стр. 274

$$\sum_{right=1}^{n-1} right = 1 + 2 + \ldots + (n-1) = n(n-1)/2,$$

т.е. пропорционално е на квадрата на броя на елементите *n* на редицата. Заради това методът е удобен за сортиране на редици с относително малък брой елементи. Освен това при този метод за сортиране броят на сравненията зависи от първоначалната наредба на елементите на редицата. Методът е бърз, когато малко на брой елементи на редицата не са на местата си. Например в редицата

1 2 9 3 5 6 8 10 11

само елементът 9 не е на мястото си. Ако, започвайки отляво, сравняваме всеки два съседни елемента и ги разменяме, когато левият е по-голям от десния, 9 ще се размени с 3, после с 5, с 6 и с 8 и ще достигне своята правилна позиция.

Сортиране чрез клатене

Методът се състои в следното:
1. Нека *left = 0; right = n-1;*
2. Подредицата

$a_{left}, \ldots, a_{right}$

се обхожда два пъти, като се сравняват всеки два съседни елемента. Едното обхождане е отляво надясно, а другото — отдясно наляво.

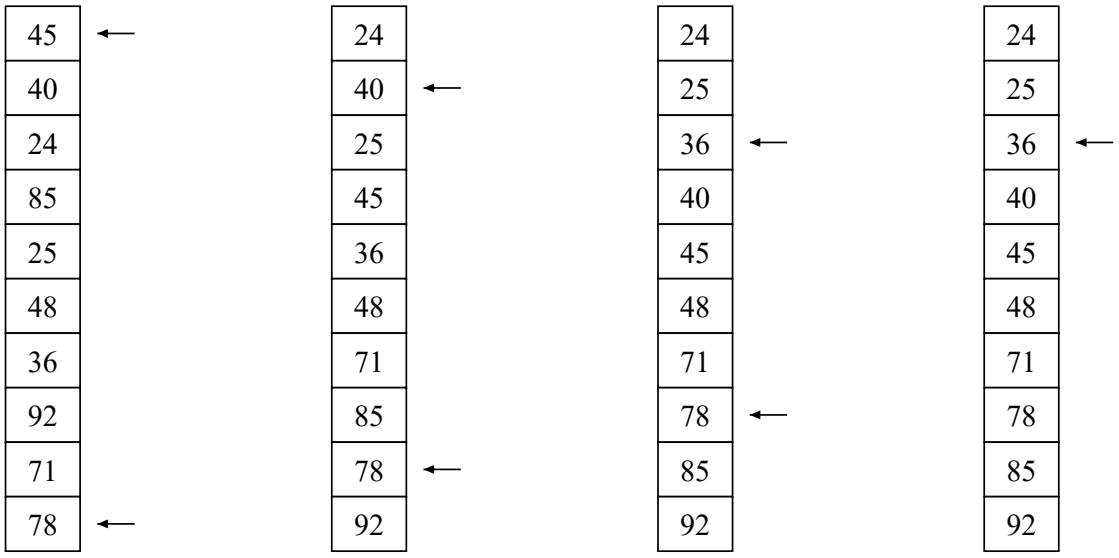
При първото обхождане, аналогично на метода на мехурчето, се намира нова стойност на *right*, чрез която се намалява дължината на редицата, а елементите от позиции *right+1, right+2, ..., n-1* са правилно подредени.

При второто обхождане се извършват симетрични действия, като се намира нова стойност на *left*, а елементите от позиции *0, 1, ..., left-1* са правилно подредени. Ще забележим, че при това обхождане се използва новата стойност на *right*.

3. Обхожданията продължават, докато е в сила *right-left > 0*, т.е. докато съществува подредицата.

Пример. Методът е илюстриран графично на фиг. 4.3. при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. Стрелките вдясно на всеки масив сочат елементите a_{left} и a_{right} на неоцветената част на редицата. Вдясно на последната колонка е показана само една стрелка, тъй като *left == right*. Оцветените части представят сортираните части на редицата.

Оригинал: стр. 275



Фигура 4.3. Пример за сортиране на редица по метода сортиране чрез клатене

Реализация на метода за масиви

Задача 51. Да се дефинира шаблон на процедура, реализиращ метода сортиране чрез клатене за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

C275-01 · Копиране: code.html
```
template <typename T>
void shakerSort(T* a, int n)
{ int i, k;
  int left = 0, right = n-1;
  while (right-left > 0)
    { k = 0;
      for (i = left; i < right; i++)
        if (a[i] > a[i+1])
          { swap(a[i], a[i+1]);
            k = i;
          }
      right = k;
      k = n-1;
      for (i = right; i > left; i--)
        if (a[i-1] > a[i])
          { swap(a[i-1], a[i]);
            k = i;
          }
```

Оригинал: стр. 276

C276-01 · Копиране: code.html
```
      left = k;
    }
}
```

Реализация на метода за файлове

Задача 52. Да се дефинира шаблон на процедура, реализиращ метода сортиране чрез клатене за сортиране във възходящ ред на компонентите на файл с пряка организация. Компонентите на файла са от тип *T*.

C276-02 · Копиране: code.html
```
template <typename T>
void shakerSort(fstream& f, T& dummy)
{ int i, k;
  // намиране на броя на елементите на файла f
  f.seekg(0, ios::end);
  int n = f.tellg()/sizeof(T);
  int left = 0, right = n-1;
  T a, b;
  while (right-left > 0)
    { k = 0;
      for (i = left; i < right; i++)
        { f.seekg(i*sizeof(T));
          f.read((char*)&a, sizeof(T));
          f.read((char*)&b, sizeof(T));
          if (b < a)
            { swap(f, i, i+1, dummy);
              k = i;
            }
```

C276-02 · продължение · Копиране: code.html

```
        }
      right = k;
      k = n-1;
      for (i = right; i > left; i--)
        { f.seekg((i-1)*sizeof(T));
          f.read((char*)&a, sizeof(T));
          f.read((char*)&b, sizeof(T));
          if (b < a)
            { swap(f, i-1, i, dummy);
              k = i;
            }
        }
```

Оригинал: стр. 277

C277-01 · Копиране: code.html

```
      left = k;
    }
  f.close();
}
```

Оценка на метода сортиране чрез клатене

Въпреки че се явява подобрение на метода на мехурчето сортирането чрез клатене също е със сложност $O(n^2)$. Оценката се осъществява по аналогичен начин на направените по-горе оценки и я оставяме на читателя. Аналогично на метода на мехурчето ефективността на метода зависи от първоначалната наредба на елементите на редицата. Методът е бърз (със сложност $O(n)$), когато малко на брой елементи на редицата не са на местата си. Например в редицата

1 2 9 3 5 6 8 4 10 11

само елементите 9 и 4 не са на местата си. Ако, започвайки отляво, сравняваме всеки два съседни елемента и ги разменяме, когато левият е по-голям от десния, 9 ще се размени с 3, после с 5, с 6, с 8 и с 4 и ще достигне своята правилна позиция. Получава се

1 2 3 5 6 8 4 9 10 11

След това, започвайки от елемента 9, сравняваме всеки два съседни елемента и ги разменяме, когато левият е по-голям от десния. При това 4 ще се размени с 8, после с 6 и с 5 и ще достигне своята правилна позиция.

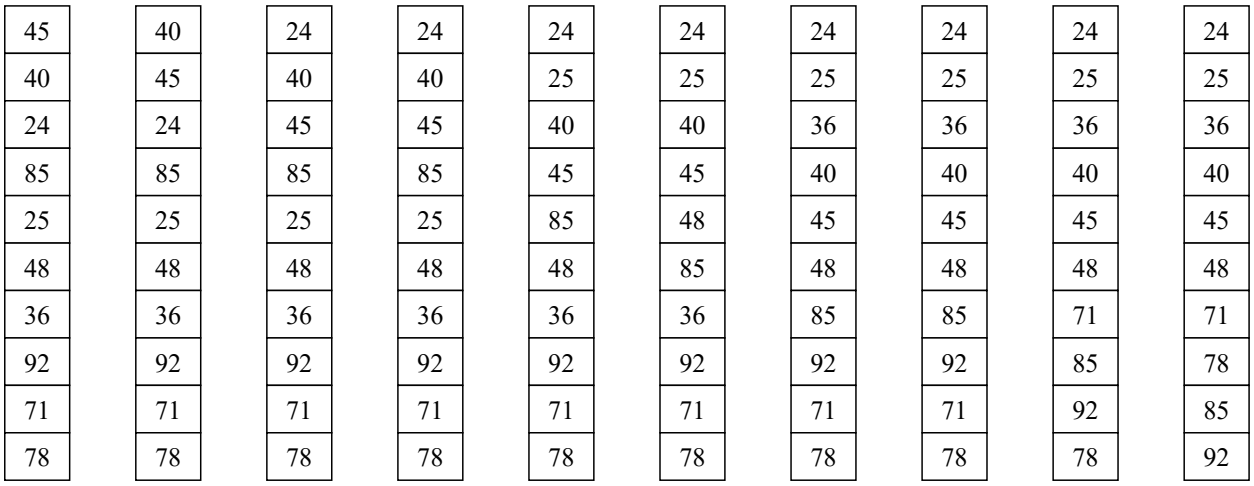
4.1.3. Сортиране чрез вмъкване

Методът реализира следната идея.

Ако редицата
$a_0, a_1, \ldots, a_{i-1}$
е сортирана във възходящ ред, елементът a_i се включва в редицата така, че да заеме „правилна“ позиция, т.е. новата редица
$a_0, a_1, \ldots, a_i, \ldots, a_{i-1}$
да е сортирана във възходящ ред. Тези действия се повтарят за $i = 1, 2, \ldots, n-1$.

Пример. Методът е илюстриран графично на фиг. 4.4. при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. Оцветената част е сортираната подредица, в която се вмъква следващият елемент.

Оригинал: стр. 278



Фигура 4.4. Пример за сортиране на редица по метода сортиране чрез вмъкване

Реализация на метода за масиви

Задача 53. Да се дефинира шаблон на процедура, реализиращ метода сортиране чрез вмъкване за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

Шаблонът на процедурата *insertSort* решава задачата. Тя използва помощния шаблон на процедура *void insert(T* a, int i)*, която вмъква елемента *a[i]* в сортираната вече във възходящ ред редица $a_0, a_1, \ldots, a_{i-1}$.

C278-01 · Копиране: code.html

```
template <typename T>
void insert(T* a, int i)
{ int j = i-1;
  T x = a[i];
  while(j >= 0 && a[j] > x)
  { a[j+1] = a[j];
    j--;
  }
  a[j+1] = x;
}

template <typename T>
void insertSort(T* a, int n)
{ for (int i = 1; i < n; i++)
    insert(a, i);
}
```

Оригинал: стр. 279

Реализация на метода за файлове

Задача 54. Да се дефинира шаблон на процедура, реализиращ метода сортиране чрез вмъкване за сортиране във възходящ ред на компонентите на файл с пряка организация на достъпа до компонентите. Компонентите на файла са от тип *T*.

C279-01 · Копиране: code.html
```
template <typename T>
void insert(fstream& f, int i, T& dummy)
{ int j = i-1;
  T x, a;
  f.seekg(i*sizeof(T));
  f.read((char*)&x, sizeof(T));
  f.seekg(j*sizeof(T));
  f.read((char*)&a, sizeof(T));
  while(j >= 0 && a > x)
    { f.seekp((j+1)*sizeof(T)); // за яснота на алгоритъма
      f.write((const char*)&a, sizeof(T));
      j--;
      f.seekg(j*sizeof(T));
      f.read((char*)&a, sizeof(T));
    }
  f.seekp((j+1)*sizeof(T));
  f.write((const char*)&x, sizeof(T));
}

template <typename T>
void insertSort(fstream& f, T& dummy)
{ //намиране на броя на елементите на f
  f.seekg(0, ios::end);
  long n = f.tellg()/sizeof(T);
  for (int i = 1; i < n; i++)
    insert(f, i, dummy);
  f.close();
}
```

Оценка на сортирането чрез вмъкване

Този метод за сортиране също е ниско ефективен. Сложността му е $O(n^2)$ и може да се намери, като се използват правила 5 и 4 за изчисляване на времето за изпълнение на програма (глава 1). Времето за изпълнение на

Оригинал: стр. 280

шаблона на процедурата *insert* е $O(i)$, а на шаблона *insertSort* е $\sum_{i=1}^{n-1} i = n(n-1)/2$. Полезността му се състои в това, че на базата на него е реализиран методът на Шел.

4.1.4. Сортировка на Шел

Този начин за сортиране е предложен от D.L.Shell през 1959 г. и реализира следната идея.

В зависимост от дължината *n* на редицата се намира „подходяща стойност“ на стъпката *h*. Сортират се елементите на редицата, които са на разстояние *h* един от друг. След това *h* се намалява чрез целочислено деление на дадено естествено число и отново се сортират всички елементи на редицата, които са на разстояние *h* един от друг. Тези стъпки се изпълняват докато *h* > *0*.

По предложение на Кнут и в съответствие с общата практика, стойностите на *h* се избират да принадлежат на редицата 1, 4, 13, 40, 121, 364, … и да са по-малки от някаква гранична стойност, зависеща от *n*. Редицата се получава съгласно следната зависимост:

$h_1 = 1;$
$h_i = 3h_{i-1} + 1;\ i = 2, 3, \ldots$

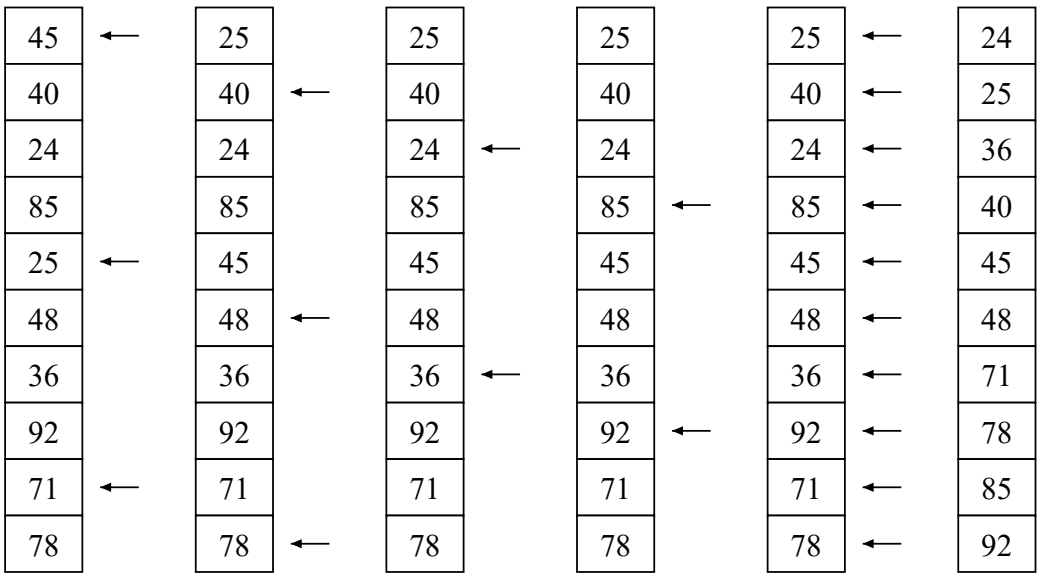
Началната стойност на *h* се избира да удовлетворява условията:
$2h < n$
$2(3h + 1) > n.$

Тази стойност може да се определи чрез следния програмен фрагмент:

C280-01 · Копиране: code.html
```
h = 0;
while (2*(3*h+1) <= n) h = 3*h + 1;
```

Пример. Методът е илюстриран графично на фиг. 4.5 при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. При тази стойност на *n* за *h* се получават стойностите 4 (начална стойност) и 1. Със стрелки са означени елементите от подредиците, които се сортират чрез вмъкване. В първите 4 стъпки се сортират чрез вмъкване елементите на всички подредици с разстояние 4 между елементите си, а в последната стъпка се сортират чрез вмъкване елементите на разстояние 1.

Оригинал: стр. 281



Фигура 4.5. Пример за сортиране на редица по метода на Шел

Реализация на метода за масиви

Задача 55. Да се дефинира шаблон на процедура, реализиращ метода на Шел за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

C281-01 · Копиране: code.html
```
template <typename T>
void ShellSort(T* a, int n)
{ int h = 0, i, j;
  T x;
  while (2*(3*h +1) <= n)
    h = 3*h+1;
  while (h > 0)
    { for (i = h; i < n; i++)
        { x = a[i];
          for (j = i-h; j >= 0; j = j-h)
            if (x < a[j]) a[j+h] = a[j];
            else break;
          a[j+h] = x;
        }
      h = h/3;
    }
}
```

Оригинал: стр. 282

Реализация на метода за файлове

Задача 56. Да се дефинира шаблон на процедура, реализиращ метода на Шел за сортиране във възходящ ред на компонентите на файл с пряка организация на достъпа. Компонентите на файла са от тип *T*.

C282-01 · Копиране: code.html
```
template <typename T>
void ShellSort(fstream& f, T& dummy)
{ int h = 0, i, j;
  T x, a;
  // намиране на броя на елементите на файла f
  f.seekg(0, ios::end);
  long n = f.tellg()/sizeof(T);
  // намиране на стъпката h
  while (2*(3*h +1) <= n)
    h = 3*h+1;
  while (h > 0)
    { for (i = h; i < n; i++)
        { f.seekg(i*sizeof(T));
          f.read((char*)&x, sizeof(T));
          for (j = i-h; j >= 0; j = j-h)
            { f.seekg(j*sizeof(T));
              f.read((char*)&a, sizeof(T));
              if (x < a)
                { f.seekp((j+h)*sizeof(T));
                  f.write((const char*)&a, sizeof(T));
                }
              else break;
            }
          f.seekp((j+h)*sizeof(T));
```

C282-01 · продължение · Копиране: code.html

```
          f.write((const char*)&x, sizeof(T));
        }
      h = h/3;
    }
  f.close();
}
```

Оригинал: стр. 283

Оценка на метода на Шел

Анализът на този метод за сортиране предизвиква трудни математически проблеми, някои от които не са решени. Например не се знае при какъв избор на стъпката h ще се получат най-добри резултати. Освен това средният брой на сравненията за този метод алгебрично не е намерен. При избраната редица ефективността на метода на Шел е $O(n^k)$, където $1 < k \leq 2$. Трудностите при намирането на оценката могат да се видят на страници 98–107 на том 3 на книгата на Доналд Кнут.

Освен горната редица, за получаване на стъпката h, Кнут предлага да се използва и редицата: 1, 3, 7, 15, 31, ...
където
$h_1 = 1$;
$h_i = 2h_{i-1} + 1$; $i = 2, 3, ...$

Емпиричните изследвания за нея показват, че сложността на метода на Шел е $O(n^{1.25})$, което е по-добро от $O(n^2)$ на разгледаните преди него методи.

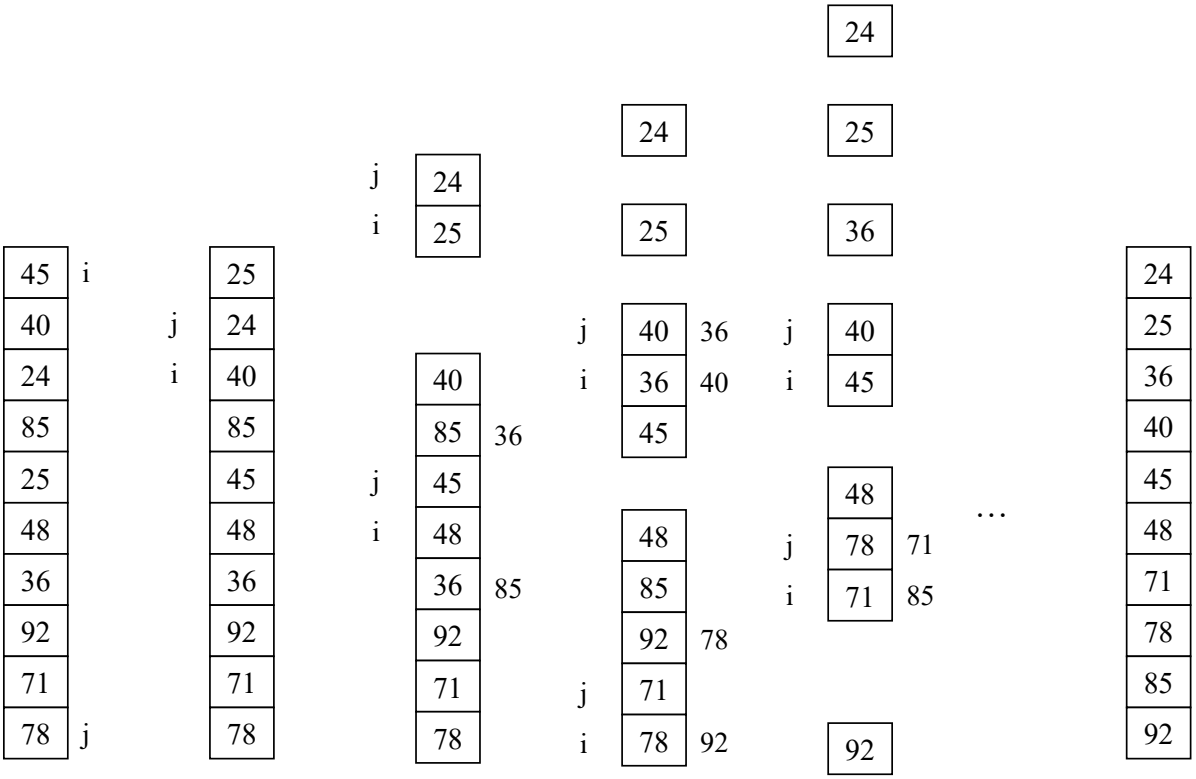
4.1.5. Бързо сортиране

Методът е предложен от C. A. Hoare през 1962 г. и е сред най-ефективните алгоритми за сортиране. Основава се на факта, че на колкото по-големи разстояния се правят разместванията, толкова по-ефективно е сортирането.

Редицата се разделя на две подредици спрямо един до голяма степен произволно избран елемент x, наречен *ос*. Елементите, които са по-малки от x, се записват в първата подредица, а всички, които са по-големи от x — във втората подредица. Същото действие се повтаря за двете подредици.

Ще дадем един опростен вариант на реализация на този метод за масив и за файл. При тях за оси се избират елементите, намиращи се в средата на подредиците.

Пример. Методът е илюстриран графично на фиг. 4.6 при $n = 10$ за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. Осите на съответните подредици са оцветени. Някои от стъпките в илюстрацията са пропуснати заради ограниченото място за фигурата.



Фигура 4.6. Пример за сортиране на редица по метода „бързо сортиране“

Реализация на метода за масиви

Задача 57. Да се дефинира шаблон на процедура, реализиращ метода за бързо сортиране за сортиране във възходящ ред на елементите на едномерен масив от тип *T*.

C284-01 · Копиране: code.html

```
template <typename T>
void qSort(T* a, int n)
{ int i = 0, j = n-1;
  T x = a[j/2], y;
  do
  { while (a[i] < x) i++;
    while (a[j] > x) j--;
    if (i < j)
      { y = a[i]; a[i] = a[j]; a[j] = y;
        i++; j--;
      }
```

C285-01 · Копиране: code.html
```
    else
      if (i==j)
        { i++;
          j--;
        }
  } while (i <= j);
  if (j > 0) qSort(a, j+1);
  if (i < n-1) qSort(a+i, n-i);
}
```

Тъй като функцията *qSort* е рекурсивна, за някои приложения се препоръчва масивът да не се включва като формален параметър, а да се разглежда като глобална променлива. Така ще се спести място в стека и програмата ще стане по-бърза.

Реализация на метода за файлове

Задача 58. Да се дефинира шаблон на процедура, реализиращ метода на бързото сортиране за сортиране във възходящ ред на компонентите на файл с пряка организация на достъп. Компонентите на файла са от тип *T*.

Шаблонът *qSort* реализира сортирането. Той използва помощния шаблон на функция *qSort1*, който сортира по описания начин частта от *m*-тата до *n*-тата компонента на файла *f*.

C285-02 · Копиране: code.html
```
template <typename T>
void qSort1(fstream& f, int m, int n, T& dummy)
{ T x, a, b;
  int i = m, j = n;
  // намиране на оста x
  f.seekg((i+j)/2*sizeof(T));
  f.read((char*)&x, sizeof(T));
  do
  { f.seekg(i*sizeof(T));
    f.read((char*)&a, sizeof(T));
    while (a < x)
      { i++;
        f.read((char*)&a, sizeof(T));
      }
```

Оригинал: стр. 286

C286-01 · Копиране: code.html

```
    f.seekg(j*sizeof(T));
    f.read((char*)&b, sizeof(T));
    while (b > x)
      { j--;
        f.seekg(j*sizeof(T));
        f.read((char*)&b, sizeof(T));
      }
    if (i < j)
      { swap(f, i, j, dummy);
        i++; j--;
      }
    else
      if (i == j)
        { i++;
          j--;
        }
  } while (i <= j);
  if (j > m) qSort1(f, m, j, dummy);
  if (i < n) qSort1(f, i, n, dummy);
}

template <typename T>
void qSort(fstream& f, T& dummy)
{ f.seekg(0, ios::end);
  int n = f.tellg()/sizeof(T);
  qSort1(f, 0, n-1, dummy);
  f.close();
}
```

Оценка на метода на Хоар за бързо сортиране

Ще направим кратък анализ само на най-добрия и на най-лошия случай на метода. Анализ на „средния“ случай може да се намери например в книгата на Tucker Allen (с. 197–199). Сложността на метода зависи от началната наредба на елементите на редицата, както и от избора на оста. Ясно е, че колкото по-равномерно се разпределят елементите в лявата и дясната подредица, толкова по-малко стъпки ще бъдат необходими. При предположение, че за всяко ниво на рекурсията изборът на оста x е такъв, че приблизително половината от елементите на разглежданата подредица са по-малки от x и половината са по-големи от x, както и че n е степен на 2, таблица 4.1 дава размерът на подре-

Оригинал: стр. 287

диците, броя на подредиците, както и броя на обръщенията към *qSort* за всяко ниво на рекурсията.

Ниво на рекурсията	Размер на подредицата	Брой на подредиците	Обръщения към qSort
0	n	1	2
1	n/2	2	4

Ниво на рекурсията	Размер на подредицата	Брой на подредиците	Обръщения към qSort
2	n/4	4	8
k	$n/2^k$	2^k	2^{k+1}
…	…	…	…
$\log_2 n - 1$	2	n/2	n
$\log_2 n$	1	n	0

Таблица 4.1. Броене на обръщенията към qSort

Операторът *do-while* на *k*-тото рекурсивно обръщение (с размер на подредицата $n/2^k$) ще изпълни поне $n/2^{k+1}$ итерации (ако за всяка двойка елементи се осъществява размяна). Освен това за всяка размяна са нужни 3 сравнения. Така сложността на метода в най-добрия случай е

$$\sum_{k=0}^{\log_2 n} 2^k(3n/2^{k+1}) = \sum_{k=0}^{\log_2 n}(3n/2) = (3n/2)(\log_2 n + 1)$$

т.е. $O(n.log_2 n)$.

Този метод за сортиране крие и изненади. Той е умерено ефективен при малки стойности на *n*. Последното е в сила за всички модерни методи за сортиране. Освен това ефективността му зависи от избора на *x*.

Най-лошият случай е този, при който всеки път за *x* се избира най-голямата стойност в сортирания фрагмент от редицата. Тогава всяка стъпка разделя фрагмент с *k* елемента на ляв фрагмент с *k-1* елемента и на десен фрагмент с 1 елемент. Тогава ефективността е $O(n^2)$.

В примерната реализация за *x* е избран средният елемент на фрагмента. Ще отбележим, че еднакво добре е *x* да е най-левият или най-десният елемент на фрагмента. В тези два случая най-лош вариант е редицата да е сортирана.

Методът е ефективен за редици с голям брой елементи.

Оригинал: стр. 288

4.1.6. Пирамидално сортиране

Нарича се още *сортиране чрез двоично дърво*. Автор му е J.Williams.

Редицата
$a_0, a_1, \ldots, a_{n-1}$
удовлетворява условието за пирамида, ако неравенствата $a_i >= a_{2i+1}$ *и* $a_i >= a_{2i+2}$ са в сила стига тези елементи да принадлежат на редицата.

Пример. Редицата

0	1	2	3	4	5	6	7	8	9
8	5	7	3	4	6	2	1	2	1

е пирамида.

Забележка. Условието за пирамида може да се изрази и чрез неравенствата $a_i <= a_{2i+1}$ *и* $a_i <= a_{2i+2}$ стига тези елементи да принадлежат на редицата.

Основна операция за работа с пирамида е *пресяването*. Ще я илюстрираме чрез следния пример. Редицата

0	1	2	3	4	5	6	7	8	9
4	5	7	3	4	6	2	1	2	1

не е пирамида. Единствено първият елемент ѝ пречи. Ако разменим елементът a_0 с максималния от a_1 и a_2, получаваме

0	1	2	3	4	5	6	7	8	9
7	5	4	3	4	6	2	1	2	1

Тази редица не е пирамида, тъй като $a_2 = 4$ не си е на мястото. Отново разменяме a_2 с максималното от a_5 и a_6. Получава се редицата

0	1	2	3	4	5	6	7	8	9
7	5	6	3	4	4	2	1	2	1

която вече е пирамида.

Процесът на получаване на пирамида се реализира чрез няколкократно прилагане на операцията пресяване.

Чрез пример ще покажем как редица може да се преобразува в пирамида. Да разгледаме редицата

0	1	2	3	4	5	6	7	8	9
5	4	1	2	9	3	6	8	7	10

Оригинал: стр. 289

Подредицата ѝ a_5, a_6, ..., a_9 е пирамида. Отначало ще изпълним операцията пресяване на a_4 в a_5, a_6, ..., a_9. Тъй като a_9 е по-голямо от a_4, разменяме им местата. Получава се редицата

0	1	2	3	4	5	6	7	8	9
5	4	1	2	10	3	6	8	7	9

която съдържа пирамидата a_4, a_5, ..., a_9. Ще извършим пресяване на елемента a_3 в подредицата a_4, a_5, ..., a_9. Разменяме a_3 с 8 — максималното от a_7 и a_8. Получава се редицата

0	1	2	3	4	5	6	7	8	9
5	4	1	8	10	3	6	2	7	9

която съдържа пирамидата a_3, a_4, ..., a_9. Ще извършим пресяване и на елемента a_2 в пирамидата a_3, a_4, ..., a_9. Разменяме a_2 с 6 — максималното от a_5 и a_6. Получава се редицата

0	1	2	3	4	5	6	7	8	9
5	4	6	8	10	3	1	2	7	9

която съдържа пирамидата a_2, a_3, ..., a_9. Продължаваме с пресяване на a_1 в пирамидата a_2, a_3, ..., a_9. Разменяме a_1 с 10 — максималното от a_3 и a_4, а също a_4 с a_9. Получава се редицата

0	1	2	3	4	5	6	7	8	9
5	10	6	8	9	3	1	2	7	4

която съдържа пирамидата a_1, a_2, ..., a_9. Накрая пресяваме a_0 в пирамидата a_1, a_2, ..., a_9. Получава се пирамидата

0	1	2	3	4	5	6	7	8	9
10	9	6	8	5	3	1	2	7	4

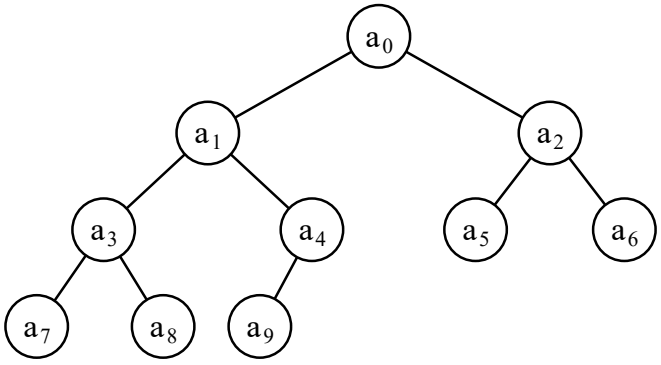
Забелязваме, че максималният елемент на пирамидата е a_0.

Методът за пирамидално сортиране се състои в следното:
1. Чрез прилагане на операцията пресяване, елементите на редицата се пренареждат в пирамида. Максималният елемент на редицата е в позиция 0.
2. Разменят се a_0 и a_{n-1}. Максималният елемент на редицата заема правилната си позиция.
3. Прилага се операцията пресяване за елемента a_0 на редицата a_0, a_1, …, a_{n-2}. В резултат се получава пирамида. Разменят се a_0 и a_{n-2}. Максималният елемент на редицата a_0, a_1, …, a_{n-2} заема правилната си позиция.

Оригинал: стр. 290

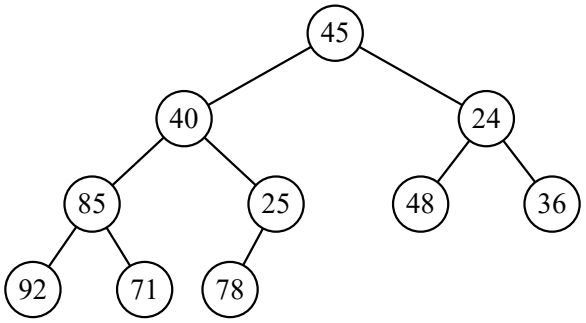
4. Описаните действия се повтарят до изчерпване на редицата, като на всяка стъпка максималният елемент заема правилна позиция, а дължината на редицата намалява с 1.

Следващият пример илюстрира метода като използва дървовидно представяне на редица. Например редицата a_0, a_1, …, a_8, a_9 се представя по следния начин

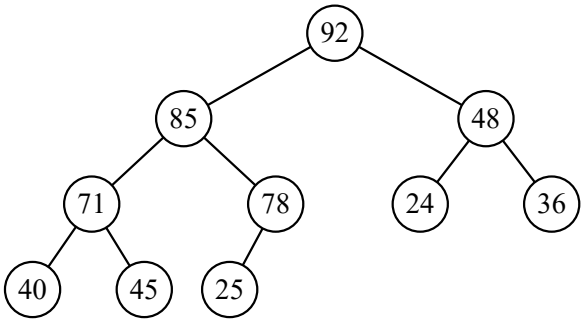


Пример. Методът е илюстриран на фиг. 4.7 при *n = 10* за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78. В част (а) са дадени двоично дърво, представящо редицата и пирамидата, която му

съответства. В останалите (б) – (к) части са представени стъпките, през които преминава сортирането на редицата, съответстващи на т. 2 и 3 от описания по-горе метод. С пунктир са означени елементите на дървото, които са заели правилна позиция, и които не се разглеждат на следващата стъпка.

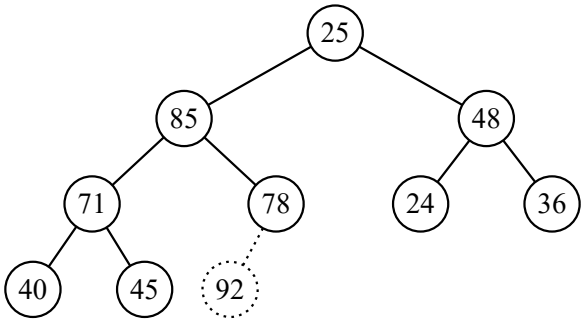


началната редица, представена чрез двоично дърво

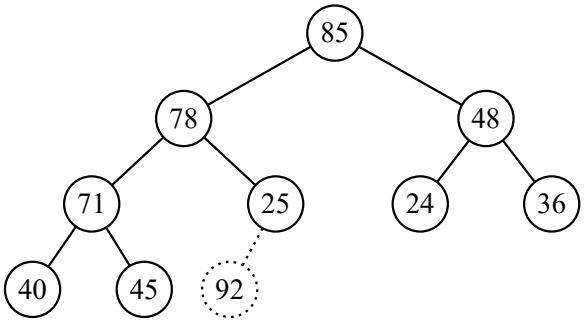


пирамидата, която ѝ съответства

а)

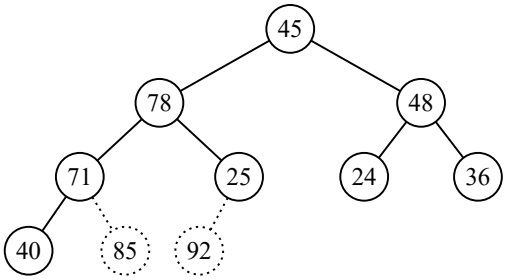


най-големият елемент заема правилната си позиция

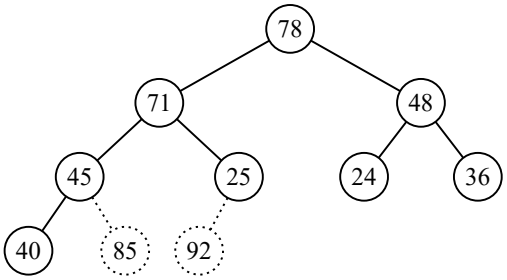


след пресяване на 25

(б) стъпка 1

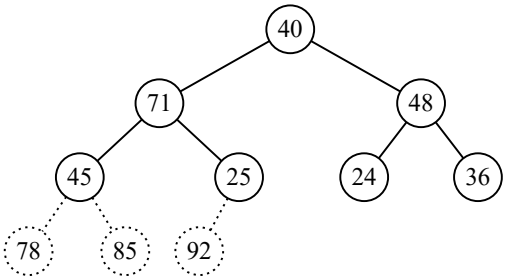


най-големият елемент заема правилната си позиция

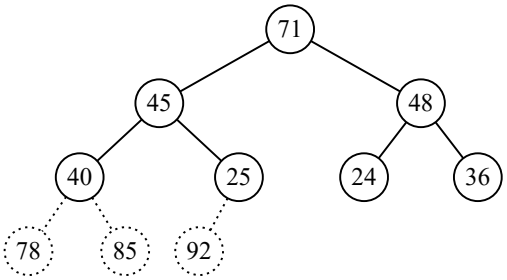


след пресяване на 45

(в) стъпка 2

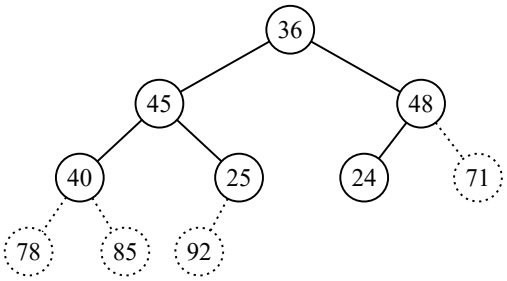


най-големият елемент заема правилната си позиция

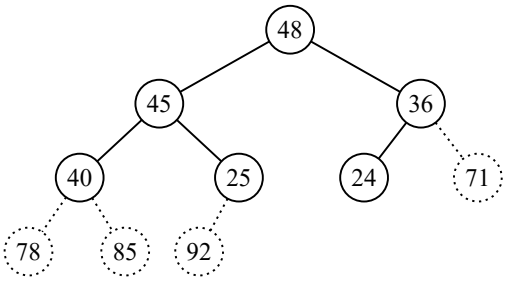


след пресяване на 40

(г) стъпка 3

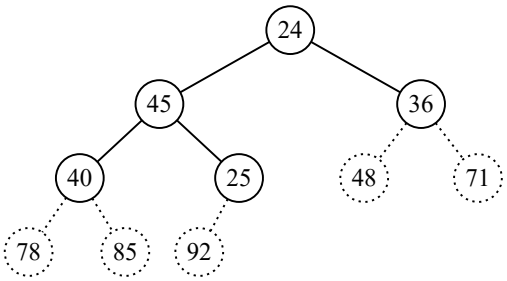


най-големият елемент заема правилната си позиция

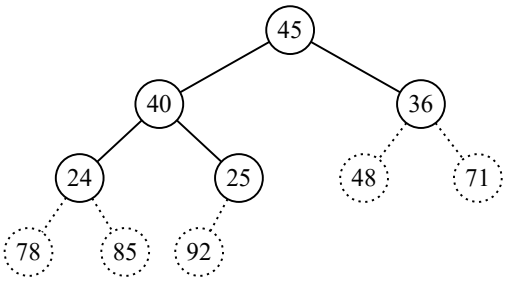


след пресяване на 36

(д) стъпка 4

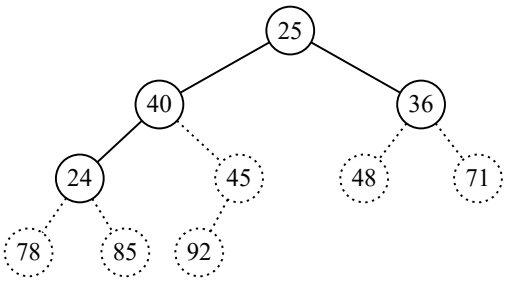


най-големият елемент заема правилната си позиция

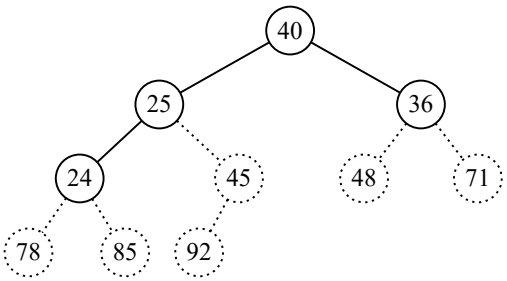


след пресяване на 24

(е) стъпка 5

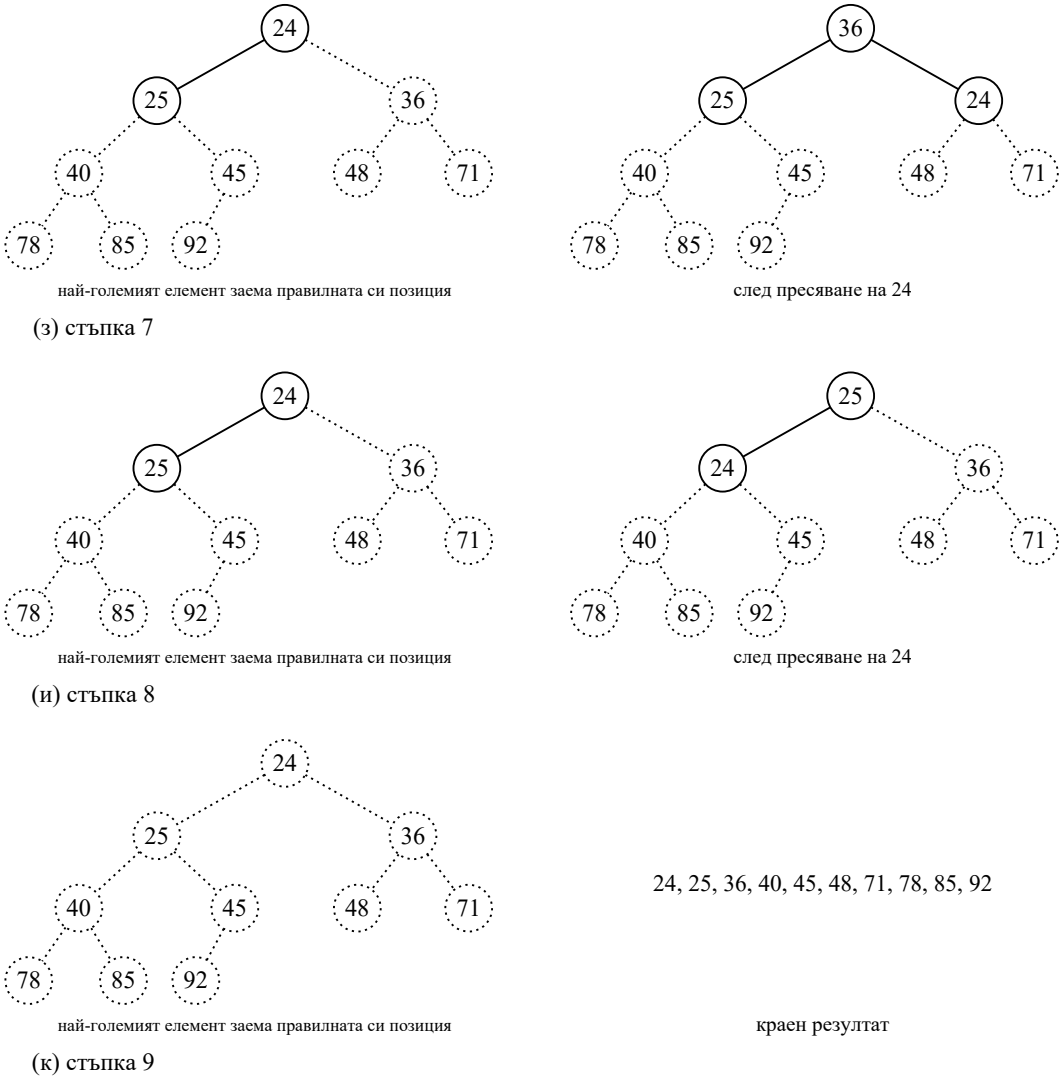


най-големият елемент заема правилната си позиция



след пресяване на 25

(ж) стъпка 6



Фигура 4.7. Пример за пирамидално сортиране на редица

Реализация на метода за масиви

Задача 59. Да се дефинира шаблон на процедура, реализиращ метода за пирамидално сортиране за сортиране във възходящ ред на елементите на едномерен масив с елементи от тип *T*.

Шаблонът на процедурата *void heapInsert(T* a, int k, int n)* вмъква елемента a_k в пирамидата a_{k+1}, a_{k+2}, …, a_{n-1}, така че получената редица да стане пирамида.

C292-01 · Копиране: code.html

```
template <typename T>
void heapInsert(T* a, int k, int n)
{ int i, j = 2*k + 1;
  while (j < n)
    { i = j;
```

Оригинал: стр. 293

C293-01 · Копиране: code.html

```
      if (j < n-1 && a[j] < a[j+1]) i = j+1;
      if (a[k] < a[i])
        { swap(a[k], a[i]);
          k = i; j = 2*k + 1;
        }
      else j = n;
    }
}

template <typename T>
void heapSort(T* a, int n)
{ int i = n/2-1;
  for (int j=i; j>=0; j--)
    heapInsert(a, j, n);
  n--;
  while (n > 0)
    { swap(a[0], a[n]);
      heapInsert(a, 0, n);
      n--;
    }
}
```

Реализация на метода за файлове

Задача 60. Да се дефинира шаблон на процедура, реализиращ метода пирамидално сортиране за сортиране във възходящ ред на компонентите на файл с пряка организация на достъп. Компонентите на файла са от тип T.

C293-02 · Копиране: code.html

```
template <typename T>
void heapInsert(fstream& f, int k, int n, T& dummy)
{ int i, j = 2*k + 1;
  T a, b;
  while (j < n)
    { i = j;
      f.seekg(j*sizeof(T));
      f.read((char*)&a, sizeof(T));
      if (j < n-1)
        { f.read((char*)&b, sizeof(T));
          if (a < b) i = j+1;
        }
      f.seekg(i*sizeof(T));
```

Оригинал: стр. 294

C294-01 · Копиране: code.html

```
      f.read((char*)&b, sizeof(T));
      f.seekg(k*sizeof(T));
      f.read((char*)&a, sizeof(T));
      if (a < b)
        { swap(f, k, i, dummy);
          k = i;
          j = 2*k+1;
        }
      else j = n;
    }
}

template <typename T>
void heapSort(fstream& f, T dummy)
{ f.seekg(0, ios::end);
  int n = f.tellg()/sizeof(T);
  int i = n/2-1;
  for (int j = i; j >= 0; j--)
    heapInsert(f, j, n, dummy);
  n--;
  while (n > 0)
    { swap(f, 0, n, dummy);
      heapInsert(f, 0, n, dummy);
      n--;
    }
  f.close();
}
```

Оценка на метода за пирамидално сортиране

Сложността на метода за пирамидално сортиране зависи от сложността на процедурата *heapInsert*. За подредица с дължина *p* тялото на цикъла на *heapInsert* изпълнява приблизително $log_2 p$ итерации. В първия цикъл на процедурата *heapSort* се изпълняват *n/2* обръщения към процедурата *heapInsert* с подредици с размер, започващ от *n/2+1* и растящ на всяка стъпка с 1. Във втория цикъл на процедурата *heapSort* се изпълняват *n-1* обръщения към процедурата *heapInsert* с подредици с размер, започващ от *n-1* и намаляващ на всяка стъпка с 1. Комбинирайки тези резултати за оценка на сложността на метода се получава

$(n/2 + 1 + (n-1)) \log_2 n = O(n. \log_2 n)$.

Оригинал: стр. 295

4.1.7. Сливане и сортиране чрез сливане

Сливане

Нека са дадени сортираните във възходящ ред редици с елементи от тип *T*:
$a_0, a_1, \ldots, a_{n-1}$
$b_0, b_1, \ldots, b_{m-1}$

Алгоритъмът за сливане конструира сортирана във възходящ ред редица:

$c_0, c_1, \ldots, c_{n+m-1}$

от елементите на дадените редици. Изразява се в следното. Установяват се указатели *i* и *j* към първите елементи на редиците $\{a_n\}$ и $\{b_m\}$ съответно. Докато има елементи и в двете редици, сравняват се елементите, сочени от указателите, по-малкият се записва в новата редица, след което се прескача. След изчерпване на елементите на едната редица останалите елементи на другата редица се преписват в новата редица $\{c_{n+m}\}$.

Реализация на метода за масиви

Задача 61. Да се дефинира шаблон на процедура, реализиращ сливане на сортираните във възходящ ред едномерни масиви $\{a_n\}$ и $\{b_m\}$ с елементи от тип *T*.

C295-01 · Копиране: code.html

```
template <typename T>
void merge(T* a, int n, T* b, int m, T *c)
{ int i = 0, j = 0, k = -1;
  while (i < n && j < m)
    if (a[i] < b[j])
      { k++;
        c[k] = a[i];
        i++;
      }
    else
      { k++;
        c[k] = b[j];
        j++;
      }
  if (i == n)
    while (j < m)
      { k++;
```

Оригинал: стр. 296

C296-01 · Копиране: code.html

```
        c[k] = b[j];
        j++;
      }
  else
    while (i < n)
      { k++;
        c[k] = a[i];
        i++;
      }
}
```

Реализация на метода за файлове

Задача 62. Да се дефинира шаблон на процедура, реализиращ сливане на сортираните във възходящ ред файлове с пряка организация и с компоненти от тип *T*.

C296-02 · Копиране: code.html

```
template <typename T>
void merge(fstream& f, fstream& g, fstream& h, T& dummy)
{ int n, m; T a, b;
  f.seekg(0, ios::end);
  n = f.tellg()/sizeof(T);
  g.seekg(0, ios::end);
  m = g.tellg()/sizeof(T);
  int i = 0, j = 0;
  while (i < n && j < m)
    { f.seekg(i*sizeof(T));
      f.read((char*)&a, sizeof(T));
      g.seekg(j*sizeof(T));
      g.read((char*)&b, sizeof(T));
      if (a < b)
        { h.write((const char*)&a, sizeof(T));
          i++;
        }
      else
        { h.write((const char*)&b, sizeof(T));
          j++;
        }
    }
  while (j < m)
```

Оригинал: стр. 297

C297-01 · Копиране: code.html

```
    { g.seekg(j*sizeof(T));
      g.read((char*)&b, sizeof(T));
      h.write((const char*)&b, sizeof(T));
      j++;
    }
  while (i < n)
    { f.seekg(i*sizeof(T));
      f.read((char*)&a, sizeof(T));
      h.write((const char*)&a, sizeof(T));
      i++;
    }
  f.close();
  g.close();
  h.close();
}
```

Процедурата *merge* реализира прост и бърз алгоритъм за сортиране, който обаче допълнително използва памет.

Оценка на метода за сливане

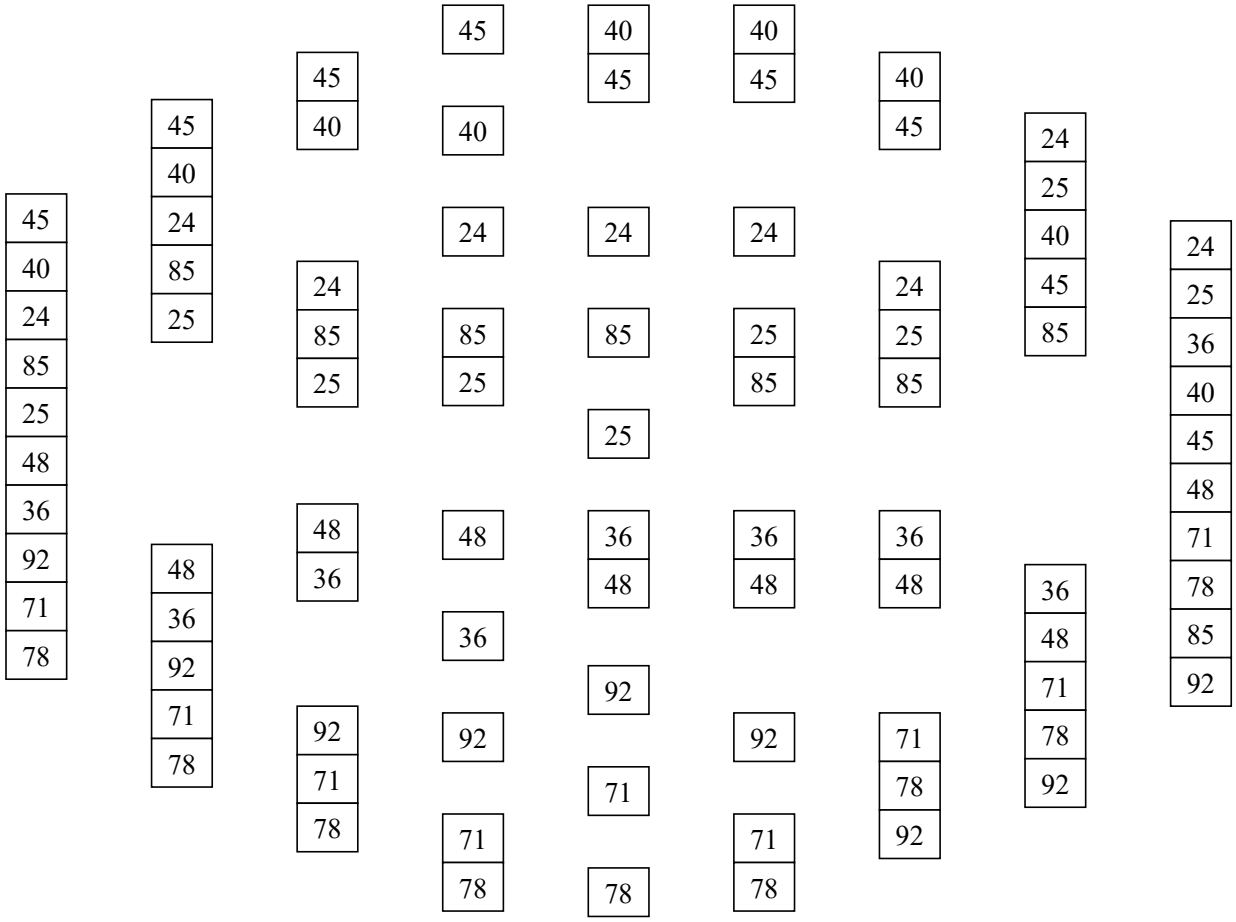
Времето за изпълнение на метода за сливане е от порядъка на $n+m$, където n и m са дължините на редиците, които се сливат.

Сортиране чрез сливане

Алгоритъмът реализира следната идея. Разделя се редицата на две половини. Всяка една от тях се сортира, като използва същия алгоритъм. След това чрез процедурата *merge* се сливат сортираните половинки.

Пример. Методът е илюстриран графично при $n = 10$ за редицата: 45, 40, 24, 85, 25, 48, 36, 92, 71, 78 на фиг. 4.8.

Оригинал: стр. 298



Фигура 4.8. Пример за сортиране на редица по метода сортиране чрез сливане

Реализация на метода за масиви

Задача 63. Да се дефинира шаблон на процедура, реализиращ сортиране чрез сливане на едномерен масив с елементи от тип *T*.

C298-01 · Копиране: code.html

```
template <typename T>
void mergeSort(T* a, int n)
{ if (n < 2) return;
  int nLeft = n/2,
      nRight = n - nLeft;
  mergeSort(a, nLeft);
  mergeSort(a + nLeft, nRight);
```

Оригинал: стр. 299

C299-01 · Копиране: code.html

```
  T* p = new T[n];
  merge(a, nLeft, a + nLeft, nRight, p);
  for (int i = 0; i < n; i++)
    a[i] = p[i];
  delete [] p;
}
```

Оценка на метода сортиране чрез сливане

Тъй като процедурата *mergeSort* е рекурсивна, времето за изпълнението ѝ *T(n)* в най-лошия случай удовлетворява следната рекурентна релация:

$T(n) = 2T(n/2) + O(n), n > 1$
$T(1) = 1.$

Решението на последната е $T(n) = O(n\log_2 n)$. Въпреки че за големи стойности на *n* методът за сортиране чрез сливане е по-добър от методите за сортиране чрез пряка селекция, пряка размяна и чрез вмъкване, той има някои недостатъци. Реализацията на метода за файлове не е ефективна. Стъпката за сливане изисква допълнителна памет за съхраняване на слятата подредица.

Този метод е ефективен за сортиране на линейни списъци (глава 2, т. 2.3.3.3, задача 12).

4.1.8. Балансирано многоходово сливане

Алгоритмите за сортиране, описани в предишните части на главата, са неприложими за сортиране на файлове, ако данните, които ще се сортират не се събират в оперативната памет на компютъра. Балансираното многоходово сливане се прилага за сортиране на файлове с големи размери. Използват се 4, 6, 8 или повече помощни файлове. Наричат се още ленти. Половината от файловете се използват като входни, а другата половина — като изходни. За описанието ще изберем 4 ленти. Четат се блокове от по *n* елемента от входния файл. Елементите на всеки блок се записват в масив, сортират се и се записват последователно на една от двете входни ленти. Процесът продължава до изчерпване на елементите на входния файл. Следва сливане на елементите от блоковете на двете входни ленти. Резултатът от всяко сливане на два блока се записва алтернативно на един от двата изходни файла. След това изходните файлове стават входни. Сливат се поредните блокове от по *2n* елемента. Резултатът от всяко сливане на два блока се записва алтернативно на един от двата изходни файла. После отново изходните файлове стават входни и т.н.

Оригинал: стр. 300

4.2. АЛГОРИТМИ ЗА ТЪРСЕНЕ

В тази част ще разгледаме методите за последователно и двоично търсене, както и техниките за хеширане, пряко свързване и отворено адресиране.

4.2.1. Последователно търсене

Този начин на търсене е най-популярен. Многократно сме го използвали за търсене на елемент с определено свойство в едномерен масив. Методът се състои в последователно сканиране на елементите на редицата до срещане на елемент с указаното свойство или до изчерпване на редицата, без да е намерен елемент.

Задача 64. Да се намери позицията на първото срещане на елемента x в редицата $\{a_n\}$ от елементи от тип T и с дължина n. Броенето на елементите да започва от 0. Ако елементът x не се среща в редицата, позицията му да е равна на n.

C300-01 · Копиране: code.html
```
template <typename T>
int linSearch(T x, T* a, int n)
{ int i = 0;
  while (i < n-1 && x != a[i]) i++;
  if (x == a[i]) return i;
  return n;
}
```

Ефективността на реализацията може да се повиши, като се въведе сентинел. За целта включваме елемента x като следващ последния елемент на редицата. Това позволява да се опрости условието за завършване на цикъла.

C300-02 · Копиране: code.html
```
template <typename T>
int linSearch(T x, T* a, int n)
{ int i = 0; a[n] = x;
  while (x != a[i]) i++;
  return i;
}
```

Друго решение е

C300-03 · Копиране: code.html
```
template <typename T>
int linSearch(T x, T* a, int n)
{ a[n] = x;
```

Оригинал: стр. 301

C301-01 · Копиране: code.html
```
  for (int i = 0; i < n; i++)
    if (x == a[i]) break;
  return i;
}
```

Реализация на метода за файлове

Задача 65. Да се дефинира шаблон на функция, реализиращ последователно търсене на елемент във файл с пряка организация на достъп и с компоненти от тип T.

Компонентите на файла са с поредни номера, започващи от 0. Ако търсеният елемент не се съдържа във файла, приемаме, че поредният му номер е равен на дължината на файла.

C301-02 · Копиране: code.html

```
template <typename T>
int linSearch(fstream& f, T x)
{// намиране на броя на компонентите на f
  f.seekg(0, ios::end);
  long n = f.tellg()/sizeof(T);
  long i = -1; T a;
  do
  { i++;
    f.seekg(i*sizeof(T));
    f.read((char*)&a, sizeof(T));
  } while (i < n-1 && x != a);
  if (x == a) return i;
  return n;
}
```

Сложността на метода е *O(n)*.

4.2.2. Двоично търсене

При този начин за търсене редицата е наредена и се състои от различни елементи. За простота ще разгледаме сортирана във възходящ ред редица от числа. В практиката може да имаме записи с уникални ключове.

Тъй като редицата е растяща, търсенето започва от средата ѝ. Ако средният елемент е търсеният, сканирането се преустановява. В противен случай търсенето продължава в лявата или в дясната половина на редицата в зависимост от това дали *x* е по-малък или по-голям от средния елемент. Тези дейс-

Оригинал: стр. 302

твия се повтарят за съответната подредица. Това описание на метода е неточно, тъй като *x* може да не се съдържа в редицата. Затова, ако *x* не е намерено, ще дадем информация на кое място в редицата *x* би трябвало да се намира.

Реализация на метода за едномерни масиви

Задача 66. Да се дефинира шаблон на функция *binSearch*, реализиращ двоично търсене на елемент в едномерен масив с компоненти от тип *T*. Функцията *binSearch* да връща целочислена стойност *i*, дефинирана по следния начин:

$$i = \begin{cases} -1, & \text{ако } x < a_0 \\ 0, & \text{ако } x = a_0 \\ j, & \text{ако } a_{j-1} < x \le a_j \\ n, & \text{ако } a_{n-1} < x \end{cases}$$

C302-01 · Копиране: code.html
```
template <typename T>
int binSearch(T x, T *a, int n)
{ int middle, left = 0, right = n-1;
  if (x < a[left]) return -1;
  if (x == a[left]) return 0;
  if (x > a[right]) return n;
  while (right-left > 1)
    { middle = (left + right)/2;
      if (x <= a[middle]) right = middle;
      else left = middle;
    }
  return right;
}
```

Реализация на метода за файлове

Задача 67. Да се дефинира шаблон на функция *binSearch*, реализиращ двоично търсене на елемент във файл с пряка организация на достъп и с компоненти от тип *T*. Функцията *binSearch* да връща целочислена стойност *i*, дефинирана по начин, аналогичен на този от предходната задача.

C302-02 · Копиране: code.html
```
template <typename T>
int binSearch(fstream& f, T x)
{ f.seekg(0, ios::end);
```

Оригинал: стр. 303

C303-01 · Копиране: code.html
```
  long n = f.tellg()/sizeof(T);
  T al, ar, am;
  int middle, left = 0, right = n-1;
  f.seekg(left*sizeof(T));
  f.read((char*)&al, sizeof(T));
  f.seekg(right*sizeof(T));
  f.read((char*)&ar, sizeof(T));
  if (x < al) return -1;
  if (x == al)return 0;
  if (x > ar) return n;
  while (right - left > 1)
    { middle = (left + right)/2;
      f.seekg(middle*sizeof(T));
      f.read((char*)&am, sizeof(T));
      if (x <= am) right = middle;
      else left = middle;
    }
  return right;
}
```

Сложността на метода е $O(log_2 n)$.

4.2.3. Хеширане

Това е техника за организация на редица от записи, позволяваща ефективно вмъкване, изтриване и търсене. Записите са организирани така, че да съдържат уникален ключ, чрез който се определя позицията им в редицата. Обикновено ключовете са естествени числа или низове. Ако ключовете са естествени числа, разположени в достатъчно малка област, всеки запис може да се съхранява на позицията, зададена чрез неговия ключ. Най-често обаче това не е така, ключовете са естествени числа, в интервал по-голям от наличното количество памет. Ако ключовете са низове, може да им бъдат съпоставени естествени числа. Изискването на различни низове да бъдат съпоставени различни естествени числа обикновено води до получаването на големи числа, така че съдържащата ги област ще е твърде голяма.

Целта е като се използва подходящо дефинирана функция, наречена *хеш-функция*, дадените записи да се запишат в *хеш-таблица*. Добре е последната да е така организирана, че броят на основните операции над нея (вмъкване, изтриване и търсене) да са от порядъка на константа.

Оригинал: стр. 304

Нека наличното количество памет да е достатъчно за разполагане на не повече от n записа, номерирани от 0 до $n-1$. Хеш-функцията трябва да е такава, че в резултат от прилагането ѝ към ключовете на записите ѝ, да се получи цяло число от 0 до $n-1$, наречено *първичен индекс*. Въпреки че е добре да е в сила еднозначно съответствие, такова не се изисква, т.е. не се иска на два различни ключа $k1$ и $k2$ хеш-функцията $hash$ да съпоставя различни цели числа. Състояние, при което за $k1 \neq k2$ е в сила $hash(k1) = hash(k2)$, се нарича *колизия*.

Два често използвани начина за разрешаване на колизиите са *прякото свързване* и *отвореното адресиране*.

4.2.3.1. Пряко свързване

Построява се едномерен масив от k свързани списъка, представени с една връзка (k може да е по-голямо, равно или по-малко от n). При прилагане на хеш-функцията към ключа се получава индекс, по който в масива се намира линейният списък, над който ще се приложи желаната операция (търсене, изтриване или вмъкване). Търсенето в тези списъци е възможно, тъй като ключовете са част от записите. Ако хеш-функцията е добра, свързаните списъци ще бъдат къси и методът ще е със сложност от порядъка на константа.

За илюстрация на метода на прякото свързване ще използваме следната таблица с n $(n = 7)$ записа

Фамилия на студент	Факултетен номер
Petrov	34560
Todorov	43930
Kolev	21210
Penev	92980
Lekov	76360
Dineva	46520

Фамилия на студент	Факултетен номер
Tonova	92220

Оригинал: стр. 305

Всеки запис означава фамилно име и факултетен номер на студент. За ключ ще използваме символния низ, означаващ фамилията на студент. Тази таблица ще е вход на програмата, дефинирана по-долу, и ще се задава чрез текстовия файл *Student_Number.txt*. Програмата прехвърля записите на файла в хеш-таблица с k елемента. Ако $k<7$, непременно ще има колизии. Ако $k \geq 7$, тогава отново може да има колизии. Хеш-функцията може да се определи по различни начини. Ще я дефинираме като *public* член-функция на класа *hashTable* по следния начин

C305-01 · Копиране: code.html
```
unsigned hashTable::hash(const char* str) const
{ unsigned sum = 0;
  for(unsigned i = 0; str[i]; i++)
    sum += (i+1)*str[i];
  return sum % k;
}
```

При стойност 5 за k за ключа „Petrov“ се получава първичен индекс

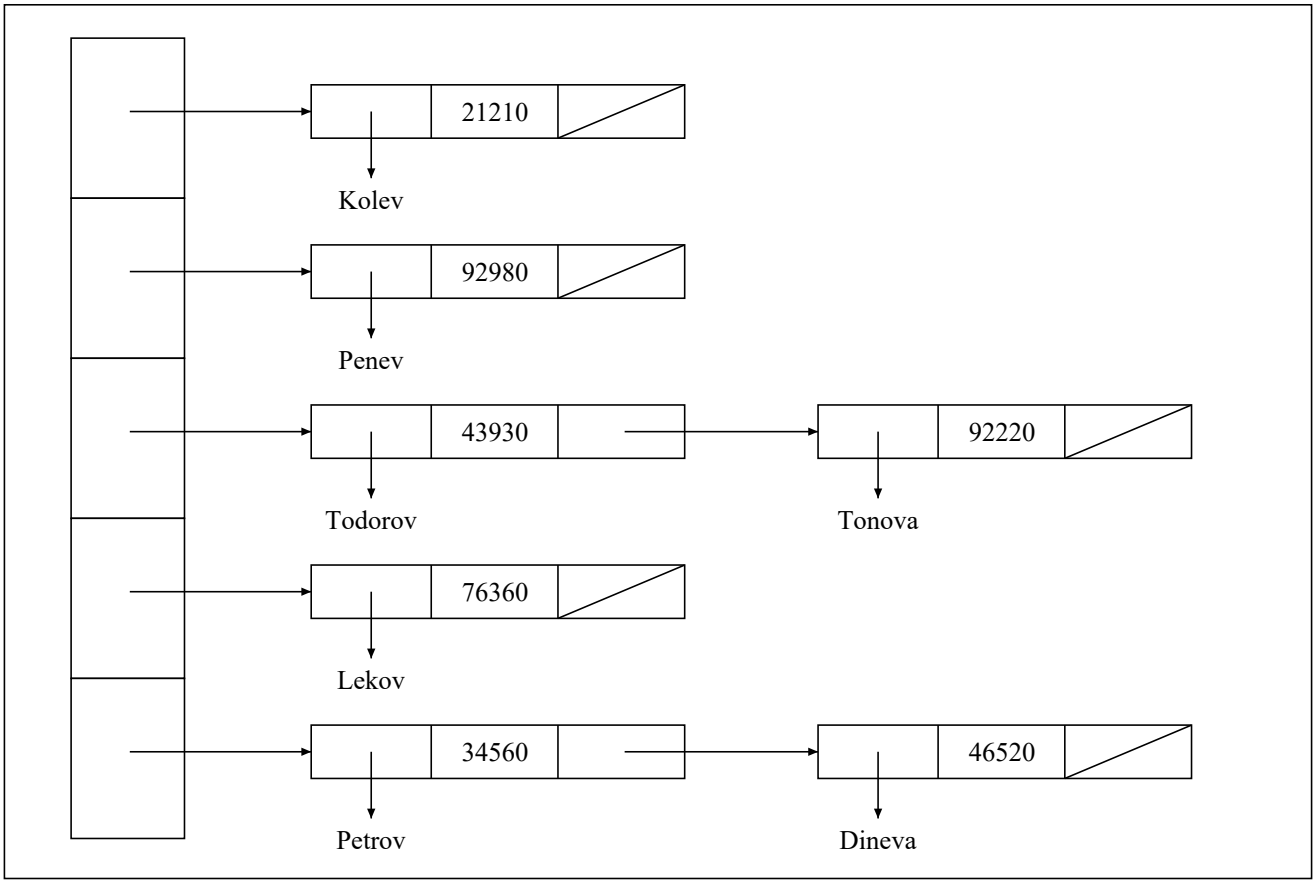
C305-02 · Копиране: code.html
```
hash("Petrov") = (1*'P' + 2*'e' + 3*'t' +
                  4*'r' + 5*'o' + 6*'v') % 5
               = (1*80 + 2*101 + 3*116 +
                  4*114 + 5*111 + 6*118) % 5 = 4
```

и показва, че записът с ключ „Petrov“ ще бъде включен в линейния списък с индекс 4 на хеш-таблицата. За записите от файла *Student_Number.txt* първичните индекси са

Ключ	Първичен индекс
Petrov	4
Todorov	2
Kolev	0
Penev	1
Lekov	3
Dineva	4
Tonova	2

а хеш-таблицата има вида

Оригинал: стр. 306



Следващата програма реализира метода за хеширане с пряко свързване.

C306-01 · Копиране: code.html

```
#include <iostream>
#include <cassert>
#include <iomanip>
#include <fstream>
#include <cstring>

#include "LList.cpp"
using namespace std;

struct rec
{ char* name;             // фамилия на студент
  unsigned num;           // факултетен номер на студент
  // канонично представяне на структурата
  rec(const char* s = "", unsigned n = 0)
  { name = new char[strlen(s)+1];
    assert(name != NULL);
    strcpy(name, s);
```

Оригинал: стр. 307

C307-01 · Копиране: code.html

```
    num = n;
  }
  ~rec()
  { delete [] name;
  }
  rec(const rec& r)
  { name = new char[strlen(r.name)+1];
    assert(name != NULL);
    strcpy(name, r.name);
    num = r.num;
  }
  rec& operator=(const rec& r)
  { if (this != &r)
      { delete [] name;
        name = new char[strlen(r.name)+1];
        assert(name != NULL);
        strcpy(name, r.name);
        num = r.num;
      }
    return *this;
  }
};

typedef LList<rec> list_Rec;// тип свързан списък от записи

void LList<rec>::print() const
{ elem_link1<rec>* p = start;
  while (p)
    { cout << p->inf.name << " " << p->inf.num << " ";
      p = p->link;
    }
  cout << endl;
}

class hashTable
{ public:
    hashTable(unsigned = 1021);
    ~hashTable();
    // хеш-функция
    unsigned hash(const char*) const;
```

Оригинал: стр. 308

C308-01 · Копиране: code.html

```
    // включване на нов запис по зададени компоненти
    void insert(const char*, unsigned);
    // търсене на запис по указан ключ
    elem_link1<rec>* find(const char*) const;
    // изтриване на запис по указан ключ
    void delRec(const char*);
    // извеждане на хеш-таблицата
    void print() const;
  private:
    unsigned k;     // брой елементи на хеш-таблицата
    list_Rec *arr;  // носител на хеш-таблица
};

hashTable::hashTable(unsigned n)
{ k = n;
  arr = new list_Rec[n];
  assert(arr != NULL);
}

hashTable::~hashTable()
{ delete [] arr;
}

unsigned hashTable::hash(const char* str) const
{ unsigned sum = 0;
  for (unsigned i = 0; str[i]; i++)
    sum += (i+1)*str[i];
  return sum % k;
}

void hashTable::insert(const char* str, unsigned nu)
{ rec r(str, nu);
  arr[hash(str)].toEnd(r);
}

elem_link1<rec>* hashTable::find(const char* str) const
{ arr[hash(str)].iterStart();
  elem_link1<rec>* p = arr[hash(str)].iter();
  while (p && strcmp(p->inf.name, str))
    p = p->link;
```

Оригинал: стр. 309

C309-01 · Копиране: code.html

```
  return p;
}

void hashTable::delRec(const char* str)
{ arr[hash(str)].iterStart();
  elem_link1<rec>* p = arr[hash(str)].iter();
  while(p && strcmp(p->inf.name, str))
    p = p->link;
  if (p)
    { rec y;
      arr[hash(str)].deleteElem(p, y);
    }
}

void hashTable::print() const
{ for (unsigned i = 0; i < k; i++)
    if (!arr[i].empty())
      { arr[i].print();
        cout << endl;
      }
}

// примерно изпълнение
int main()
{ ifstream f("c:\\zad\\Student_Number.txt", ios::in);
  if (!f)
    { cerr << "Cannot open file Student_Number.txt!\n";
      return 1;
    }
  char name[16];
  unsigned num, k;
  cout << "Дължина k на хеш-таблицата: "; cin >> k;
  // създаване на празна хеш-таблица
  hashTable hashObj(k);
  // прехвърляне на записите на файла f
  // в хеш-таблицата hashObj
  while (f >> name >> num)
    { cout << setw(35) << left << name
           << setw(20) <<num << endl;
      hashObj.insert(name, num);
```

Оригинал: стр. 310

C310-01 · Копиране: code.html
```
    }
  // извеждане на хеш-таблицата
  hashObj.print(); cout << endl;
  // търсене на записи в хеш-таблицата
  //(до въвеждане на end)
  elem_link1<rec>* p = NULL;
  cout << "Въведете име или end за край на програмата!\n";
  cin >> name;
  while (strcmp(name, "end"))
    { p = hashObj.find(name);
      if (p) cout << "Записът е намерен в хеш-таблицата.\n";
      else cout << " Записът не е намерен в хеш-таблицата.\n";
      cout << "Въведете фамилия или end "
              "за край на програмата!\n";
      cin >> name;
    }
  f.close();
  return 0;
}
```

4.2.3.2. Отворено адресиране

При този начин за разрешаване на колизиите хеш-таблицата се реализира чрез едномерен масив от записи. Предимството пред прякото свързване е, че се спестява памет за връзките при реализирането на линейните списъци. Недостатъкът е, че предварително трябва да се задели фиксирано количество памет за хеш-таблицата. Ако хеш-таблицата е голяма, броят на колизиите намалява и се увеличава броят на елементите, които могат да се вмъкнат, преди хеш-таблицата да се е препълнила. В този случай е възможно да се задели памет, която в действителност може да не се използва.

Ще илюстрираме този метод за разрешаване на колизиите чрез таблицата от записи, дадена в т. 4.2.3.1. Ключовете отново ще са низовете, задаващи фамилиите на студентите. Ще бъдат реализирани операциите вмъкване и търсене. Записите на хеш-таблицата са декларирани чрез структурата *rec*. Реализацията на хеш-таблицата и на основните операции над нея се осъществява чрез класа *hashTable*.

C310-02 · Копиране: code.html
```
struct rec
{ char name[32];             // фамилно име
  unsigned num;             // факултетен номер
};
```

Оригинал: стр. 311

C311-01 · Копиране: code.html
```
class hashTable
{ public:
    // конструктор на хеш-таблицата
    hashTable(unsigned = 1021);
    // деструктор
    ~hashTable();
    // включване на запис в хеш-таблицата
    void insert(const char*, unsigned);
    // търсене на запис в хеш-таблицата
    rec* find(const char*);
    // извеждане на хеш-таблицата
    void print() const;
  private:
    unsigned n,    // размер на хеш-таблицата
             sum;  // за низа "Len" sum = 1.'L'+2.'e' + 3.'n'
    rec* arr;      // носител на хеш-таблицата
    unsigned hash(const char*); // хеш-функция
    unsigned increment() const; // втора хеш-функция
    bool search(const char*, unsigned &); // търсене на ключ
};
```

Дължината *n* на хеш-таблицата *arr* се определя от потребителя, но е желателно да е просто число и по-точно число-близнак (определено е по-долу). За *n* избираме 13 (*n* и *n-2* са прости). Конструкторът създава празна хеш-таблица (едномерен масив от 13 елемента от тип *rec*) и инициализира полето *name* на всички записи на хеш-таблицата с празния низ. Чрез член-функцията *hash*

C311-02 · Копиране: code.html
```
unsigned hashTable::hash(const char* str)
{ sum = 0;
  for (unsigned i = 0; str[i]; i++)
    sum += (i+1)*str[i];
  return sum % n;
}
```

за всеки ключ на записите от файла се намира първичният му индекс.

Ако първичните индекси са различни, всеки запис се разполага на съответното място в хеш-таблицата. Ако възникнат колизии, се прилага втора хеш-функция *increment*, която задава стойността на увеличението, чрез което се намира мястото на записа в хеш-таблицата. В случая имаме

Ключ	Първичен индекс
Petrov	9
Todorov	5
Kolev	3
Penev	7
Lekov	8
Dineva	0
Tonova	3

Хеш-функцията пресмята еднозначно първичните индекси *i* на първите шест ключа. При седмия ключ възниква колизия

C312-01 · Копиране: code.html
```
hash("Kolev") = hash("Tonova") = 3
```

При разполагането на записите в хеш-таблицата проверката за колизия се осъществява чрез проверка дали низ, различен от празния, е записан на мястото, определено от първичния индекс. В случая записът с ключ „Petrov“ се записва в позиция 9 на хеш-таблицата, записът с ключ „Todorov“ се записва в позиция 5 на хеш-таблицата, записът с ключ „Kolev“ се записва в позиция 3 на хеш-таблицата и т.н. На записа с ключ „Tonova“ е определена позиция 3 на хеш-таблицата. Тя е заета от записа с ключ „Kolev“. Намирането на мястото за записа с ключ „Tonova“ се осъществява чрез прилагане на втората хеш-функция — член-функцията *increment* на класа *hashTable*, която определя стойността на увеличението.

C312-02 · Копиране: code.html
```
unsigned hashTable::increment() const
{ return 1 + sum % (n-2);
}
```

В случая *sum* е 2252, а

C312-03 · Копиране: code.html
```
unsigned incr = increment();
```

свързва *incr* с 9. Операторът

C312-04 · Копиране: code.html
```
i = (i + incr) % n;
```

с начална стойност 3 за *i* се изпълнява, докато (но по-малко от *n* пъти) се намери *i*, така че в позиция *i* на хеш-таблицата има празен запис. В случая след първото изпълнение на този оператор

за *i* се получава 12. Тъй като позиция 12 на хеш-таблицата е празна, записът с ключ „Tonova“ се записва в нея и хеш-таблицата има вида

Оригинал: стр. 313

0	Dineva	46520
1		
2		
3	Kolev	21210
4		
5	Todorov	43930
6		
7	Penev	92980
8	Lekov	76360
9	Petrov	34560
10		
11		
12	Tonova	92220

Ако за *incr* се беше получила стойност 5, следващата стойност на *i* щеше да е 8. Тъй като позиция 8 в хеш-таблицата е запълнена, ще се пресметне следващата стойност на *i*. Тя е 0 и тъй като позиция 0 на таблицата също е запълнена, ще се пресметне следващата стойност на *i*. Тя е 5 и отново позиция 5 на таблицата е запълнена. Следващата стойност на *i* е 10 и тъй като позиция 10 на хеш-таблицата е свободна, в нея щеше да се запише записът с ключ „Tonova“.

При *n* равно на 13, начална стойност на *i* равна на 3 и стойност 9 на *incr* се получава следната редица от стойности за *i*:

3, 12, 8, 4, 0, 9, 5, 1, 10, 6, 2, 11, 7, 3, 12, 8, 4, 0, 9, 5, 1, 10, 6, 2, 11, 7, ...

Първите *n* числа са различни и са от 0 до 12. Това не е случайно, а е следствие на факта, че 9 и 13 са взаимно прости. Следващите *n* числа са повторение на първите *n*. Затова пресмятането на *i* се ограничава до *n* пъти. Ако генерираните по описания по-горе начин *n* позиции в хеш-таблицата са заети, хеш-таблицата е пълна.

Ако *n* не беше просто число, то би могло да се дели на стъпката *incr*. Например, ако *n* е 10, началната стойност на *i* е 2, а *incr* е 5, за празни ще бъдат изследвани позициите: 2, 7, 2, 7,...

За да се осигури *n* и *incr* да са взаимно прости, трябва по подходящ начин да бъде дефинирана член-функцията *increment*. Дефиницията

C313-01 · Копиране: code.html
```
1 + sum % (n-2)
```

Оригинал: стр. 314

се оказва подходяща, когато *n* и *n-2* са прости числа. Наричат се още числа – близнаци. Някои примери за такива числа са:
3 и 5, 11 и 13, 41 и 43, 101 и 103, 311 и 313, 599 и 601, 1019 и 1021, 2999 и 3001, 7127 и 7129, 10007 и 10009, 30011 и 30013, ...

Следващата програма реализира описаното хеширане с отворено адресиране.

C314-01 · Копиране: code.html
```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cassert>
#include <cstring>
using namespace std;

struct rec
{ char name[32];              // фамилно име
  unsigned num;              // факултетен номер
};

class hashTable
{ public:
    // конструктор на хеш-таблицата
    hashTable(unsigned = 1021);
    // деструктор
    ~hashTable();
    // включване на запис в хеш-таблицата
    void insert(const char*, unsigned);
    // търсене на запис в хеш-таблицата
    rec* find(const char*);
    // извеждане на хеш-таблицата
    void print() const;
  private:
    unsigned n,     // размер на хеш-таблицата
             sum;   // за низа "Len" sum = 1.'L'+2.'e' + 3.'n'
    rec* arr;       // носител на хеш-таблицата
    unsigned hash(const char*);          // хеш-функция
    unsigned increment() const;          // втора хеш-функция
    bool search(const char*, unsigned &); // търсене на ключ
};

hashTable::hashTable(unsigned num)
```

Оригинал: стр. 315

C315-01 · Копиране: code.html

```
{ if (num > 3) n = num; else n = 3;
  arr = new rec[n];
  assert(arr != NULL);
  for (unsigned i = 0; i < n; i++)
    arr[i].name[0] = '\0';
}

hashTable::~hashTable()
{ delete [] arr;
}

unsigned hashTable::hash(const char* str)
{ sum = 0;
  for (unsigned i = 0; str[i]; i++)
    sum += (i+1)*str[i];
  return sum % n;
}

unsigned hashTable::increment() const
{ return 1 + sum % (n-2);
}

bool hashTable::search(const char* str, unsigned &i)
{ unsigned count = 0, incr;
  if (strcmp(arr[i].name, str))
    { incr = increment();
      do
      { count++;
        if (count == n) return false;
        i = (i+incr) % n;
      } while(strcmp(arr[i].name, str));
    }
  return true;
}

void hashTable::insert(const char* str, unsigned num)
{ unsigned i = hash(str);
  if (!search("", i))
    { cout << "Хеш-таблицата е запълнена.\n";
      return;
```

Оригинал: стр. 316

C316-01 · Копиране: code.html

```
    }
  strcpy(arr[i].name, str);
  arr[i].num = num;
}

rec* hashTable::find(const char* str)
{ unsigned i = hash(str);
  if (search(str, i)) return arr+i;
  return NULL;
}

void hashTable::print() const
{ for (unsigned i = 0; i < n; i++)
    if (strcmp(arr[i].name, ""))
      cout << setw(5) << i
           << setw(20) << arr[i].name
           << setw(10) << arr[i].num << endl;
    else cout << setw(5) << i << endl;
  cout << endl;
}

// примерно изпълнение
int main()
{ ifstream f("c:\\zad\\Student_Number.txt", ios::in);
  if (!f)
    { cerr << "Cannot open file Student_Number.txt!\n";
      return 1;
    }
  char name[16];
  unsigned num, n;
  cout << "Дължина n на хеш-таблицата: ";
  cin >> n;
  // създаване на празна хеш-таблица с n записа
  hashTable studentNumberTable(n);
  // прехвърляне на записите на файла f в хеш-таблицата
  while(f >> name >> num)
    { cout << setw(20) << setiosflags(ios::left)
           << name << setw(10) << num << endl;
      studentNumberTable.insert(name, num);
    }
```

Оригинал: стр. 317

C317-01 · Копиране: code.html

```
  // извеждане на хеш-таблицата
  studentNumberTable.print(); cout << endl;
  // търсене на запис в хеш-таблицата
  rec* p = NULL;
  cout << "Въведете фамилия или end за край на програмата!\n";
  cin >> name;
  while(strcmp(name, "end"))
    { p = studentNumberTable.find(name);
      if(p) cout << "Записът е намерен.\n";
      else cout << "Записът не е намерен.\n";
      cout << "Въведете фамилия или end "
              "за край на програмата!\n";
      cin >> name;
    }
  f.close();
  return 0;
}
```

Член-функцията *search(str, i)* на класа *hashTable* търси ключа *str* в хеш-таблицата *arr*, започвайки от позиция *i*.

ЗАДАЧИ

1. Да се напише програма, която чете естествено число *n*, следвано от *n* цели числа и сортира числата във възходящ ред по сумата на цифрите им.

2. Да се дефинира шаблон на процедура, който сортира по метода на пряката селекция елементите на линеен списък от тип *T*.

3. Да се дефинира шаблон на процедура, който сортира по метода сортиране чрез клатене елементите на линеен списък от тип *T*.

4. Да се дефинира шаблон на процедура, който сортира по метода на мехурчето елементите на линеен списък от тип *T*.

5. Да се дефинира шаблон на процедура, който сортира чрез вмъкване елементите на линеен списък от тип *T*.

6. Да се дефинира шаблон на процедура, който сортира по метода на Шел елементите на линеен списък от тип *T*.

7. Да се дефинира шаблон на процедура, който сортира по метода на бързото сортиране елементите на линеен списък от тип *T*.

8. Да се дефинира шаблон на процедура, който сортира по метода на пирамидалното сортиране елементите на линеен списък от тип *T*.

Оригинал: стр. 318

9. Да се дефинира шаблон на процедура, който сортира по метода на пряката селекция елементите на опашка от тип *T*.

10. Да се дефинира шаблон на процедура, който сортира по метода сортиране чрез клатене елементите на опашка от тип *T*.

11. Да се дефинира шаблон на процедура, който сортира по метода сортиране чрез вмъкване елементите на опашка от тип *T*.

12. Методът „сортиране чрез двоично вмъкване“ е модификация на метода за сортиране чрез вмъкване. При него намирането на мястото за вмъкване на елемента a_i в редицата a_0, a_1, …, a_{i-1}, която е вече сортирана, се осъществява чрез двоично търсене. Да се дефинира функция, която реализира метода „сортиране чрез двоично вмъкване“ за масив и за файл с компоненти от тип *T*.

13. Методът „сортиране чрез броене на честоти“ се изразява в следното. Сканира се масивът, който ще се сортира и се намират честотите на срещане на всеки елемент. След това с повторно сканиране се пренареждат елементите на масива в сортиран във възходящ ред вид. Ако *a* е масив от цели числа и за всяко *i, i = 0, 1, ..., n-1* е в сила $0 \le a[i] < m$, построява се нов масив *freq* с *m* елемента, показващи колко пъти всяко от числата от 0 до *m-1* се среща в редицата. Например при *n = 10* и *a = {2, 1, 0, 2, 1, 1, 2, 0, 1, 1}* масивът *freq* е равен на {2, 5, 3}, което означава, че 0 се среща 2 пъти, 1 – 5 пъти и 2 – 3 пъти в масива *a*. След още едно преминаване през масива *a* се попълват елементите му – първите му 2 елемента са 0, следващите 5 елемента са равни на 1 и последните 3 – са 2. Да се дефинира функция, която реализира този метод за сортиране на масив от цели числа. Да се направи оценка на сложността на метода.

14. Като се модифицира алгоритъмът за сортиране „чрез броене на честоти“, да се напише програма, която сортира редица от записи, записана в текстов файл. Всеки запис има две полета: факултетен номер (цяло число) и оценка (реално число). Възможно е за даден факултетен номер файлът да съдържа произволен брой записи, задаващи оценки по различни предмети. Факултетните номера са в интервала [44800, 44899]. В резултат от сортировката да се получи нов текстов файл, съдържащ записи с полета: факултетен номер и среден успех на студент. Записите да са подредени във възходящ ред по факултетните номера.

15. Текстов файл съдържа записи с полета: номер на артикул (цяло число от 1 до 300) и количество на доставените артикули (цяло число). Възможно е във файла да са записани множество доставки на един и същи артикул. Като се модифицира алгоритъмът за сортиране „чрез броене на честоти“, да се напише програма, която създава нов текстов файл, който съдържа записи с полета: номер на артикул и общо количество на доставка на артикула. Записите на файла да са сортирани във възходящ ред по номер на артикул.

Оригинал: стр. 319

16. Да се намери пермутация на числата *1, 2, ..., n*, при които бързото сортиране проявява най-добро (най-лошо) поведение (за *n = 5, 6, 8*).

17. Да се напише програма, която като комбинира алгоритмите за бързо сортиране и сортиране по метода на мехурчето сортира във възходящ ред елементите на едномерен масив от цели числа. Комбинирането на двата метода да стане по следния начин: бързото сортиране да намира сортирани части от масива с дължина m $(1 \le m \le n)$, след което чрез метода на мехурчето се завършва сортирането. Намерете стойността на *m*, при която времето за сортиране е минимално.

18. Да се напише програма, която като комбинира алгоритмите за бързо сортиране и сортиране чрез клатене сортира във възходящ ред елементите на едномерен масив от цели числа. За целта, като прилага метода за бързо сортиране, програмата сортира части от масива от по *m* елемента $(1 \le m \le n)$, след което продължава сортирането на редицата по метода „сортиране чрез клатене“.

Намерете стойността на *m*, при която времето за сортирането е минимално.

19. Да се напише програма, която като комбинира алгоритмите за бързо сортиране и сортиране чрез пряка селекция сортира във възходящ ред елементите на едномерен масив от цели числа. За целта, като прилага метода за бързо сортиране, програмата сортира части от масива от по *m* елемента *(1 ≤ m ≤ n)*, след което продължава сортирането на редицата по метода на пряката селекция. Намерете стойността на *m*, при която времето за сортирането е минимално.

20. Да се реализира рекурсивен алгоритъм за бързо сортиране, като изборът на оста *x* се осъществява на случаен начин.

21. Да се реализира рекурсивен алгоритъм за бързо сортиране, като изборът за оста *x* се осъществява на случаен начин и се спазва условието, че сортирането на по-къс дял трябва да се извършва преди сортирането на по-дълъг дял.

22. Да се дефинира нерекурсивна процедура, реализираща алгоритъма за бързо сортиране.

23. Да се напише програма, която създава масив от 300 (500 и 1000) случайни цели числа, сортира ги във възходящ ред по всички разгледани в тази глава методи и намира времената за изпълнение на всеки от методите.

24. Да се реализира методът на балансираното многоходово сливане, описано в т. 4.1.8, за 4 ленти.

25. Да се реализира методът на балансираното многоходово сливане, описано в т. 4.1.8, за 6 ленти.

26. Фибоначиевото търсене е подобно на двоичното търсене на елемент в редица. Разликата е единствено в начина на избиране на елемента, с който

Оригинал: стр. 320

ще се сравнява търсеният елемент. Нека редицата е представена чрез едномерен масив *a* от *n* цели числа, където *n = fib(k)* (*fib(0) = 0, fib(1) = 1, fib(k) = fib(k-1) + fib(k-2), k > 1*). Отначало търсеният елемент *x* се сравнява с *a[fib(k-1)]*. В случай на равенство, търсенето завършва успешно. Ако *x < a[fib(k-1)]*, на следващата стъпка се разглеждат елементите пред *a[fib(k-1)]*, а ако *x > a[fib(k-1)]*, се разглеждат елементите след *a[fib(k-1)]*. Тези действия се повтарят за всяка неотхвърлена част на редицата до намиране на търсения елемент или до достигане до празната редица. Да се дефинира функция, която реализира Фибоначиево търсене.

27. Да се напише програма, която търси даден елемент в сортирана във възходящ ред редица от цели числа по методите за двоично търсене и Фибоначиево търсене. Да се намерят времената за изпълнение на всеки от методите.

28. Хеш функцията в програмата в т. 4.2.3.1. се основава на ключове, които са символни низове. Да се модифицира програмата така, че хеширането да е с ключове – целите числа, представящи факултетните номера на студентите.

29. Хеш функцията в програмата в т. 4.2.3.2. се основава на ключове, които са символни низове. Да се модифицира програмата така, че хеширането да е с ключове – целите числа, представящи факултетните номера на студентите.

30. Да се напише програма, която прилага хеширане чрез отворено адресиране към елементите на файл с пряк достъп до компонентите.

ПРОЕКТИ

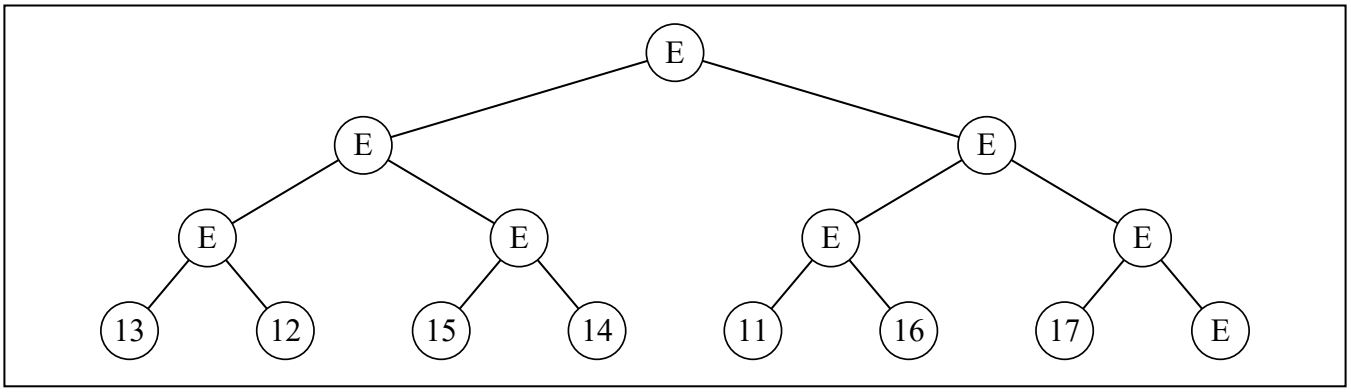
1. Сортиране с двоични дървета

Този метод за сортиране протича през две фази. Ще го илюстрираме чрез пример.

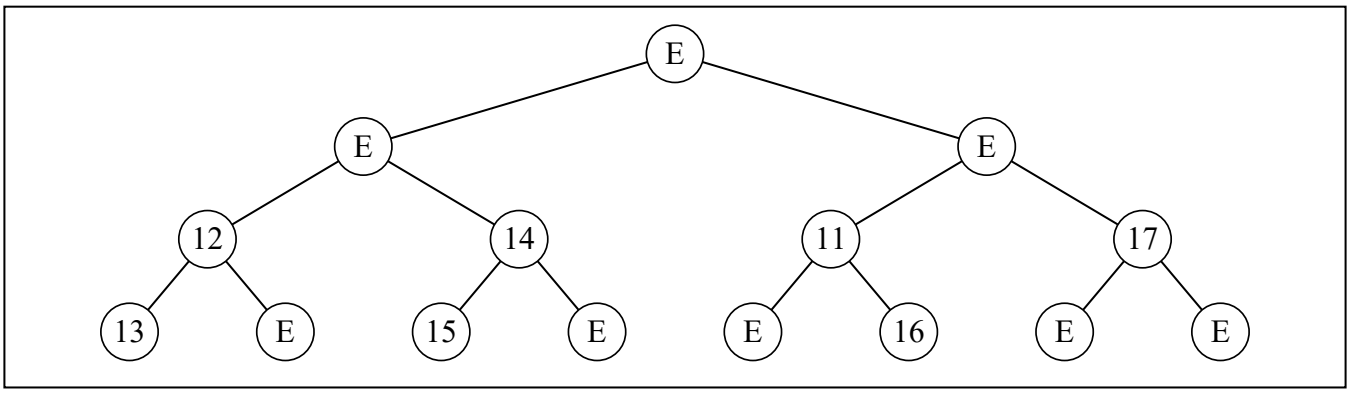
Фаза 1

Започва с построяване на пълно двоично дърво, в листата на което са разположени елементите, които трябва да бъдат сортирани. Всички върхове, предшестващи листата, се разглеждат като празни и в графичното изобразяване на метода са означени с главната латинска буква *E*. Например, ако трябва да се сортира редицата 13, 12, 15, 14, 11, 16, 17, тя се представя като листа на двоичното дърво

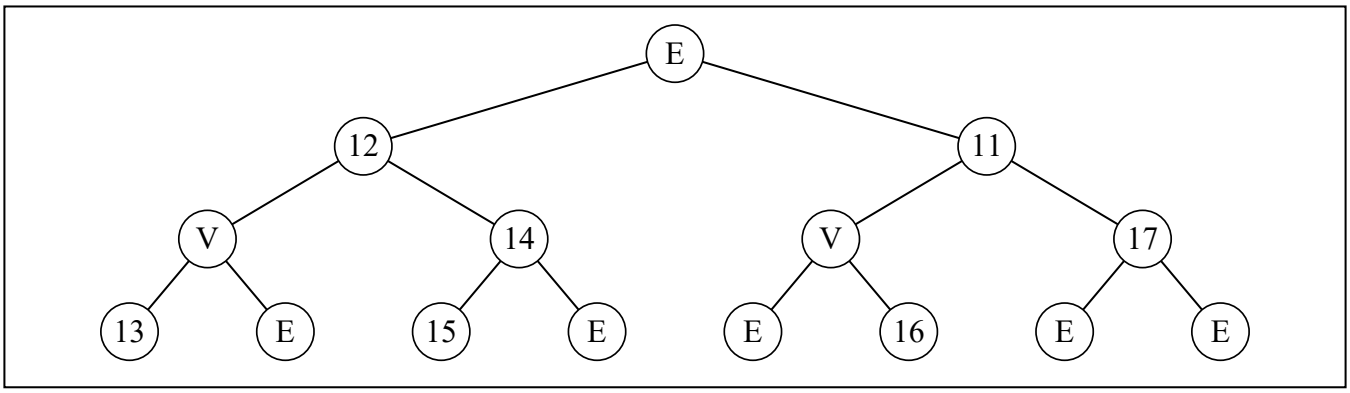
Оригинал: стр. 321



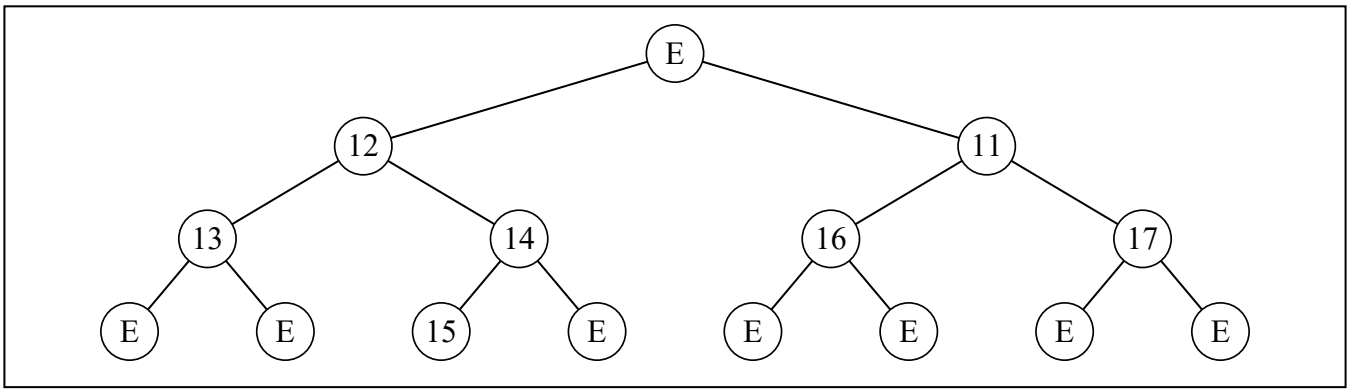
Тази фаза завършва, след като двоичното дърво се преобразува така, че най-малкият елемент на двоичното дърво се разположи в корена на дървото. За целта всяка двойка от съседни листа (в случая на ниво 4) се сравняват и по-малкият елемент се копира във върха на родителя им, а главната латинска буква *E* заеме неговото място. В случая се получава двоичното дърво



Процесът на сравняване продължава за елементите от предходното трето ниво. В резултат се получава

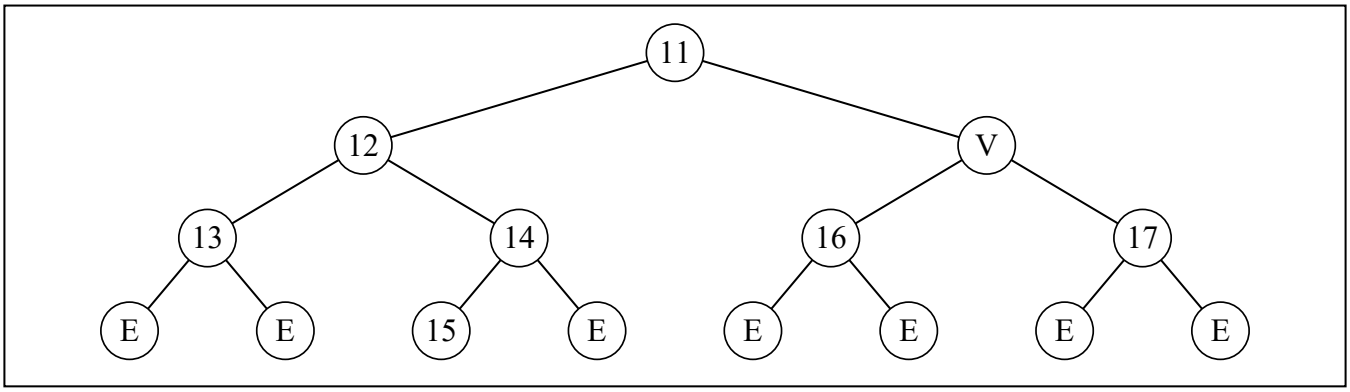


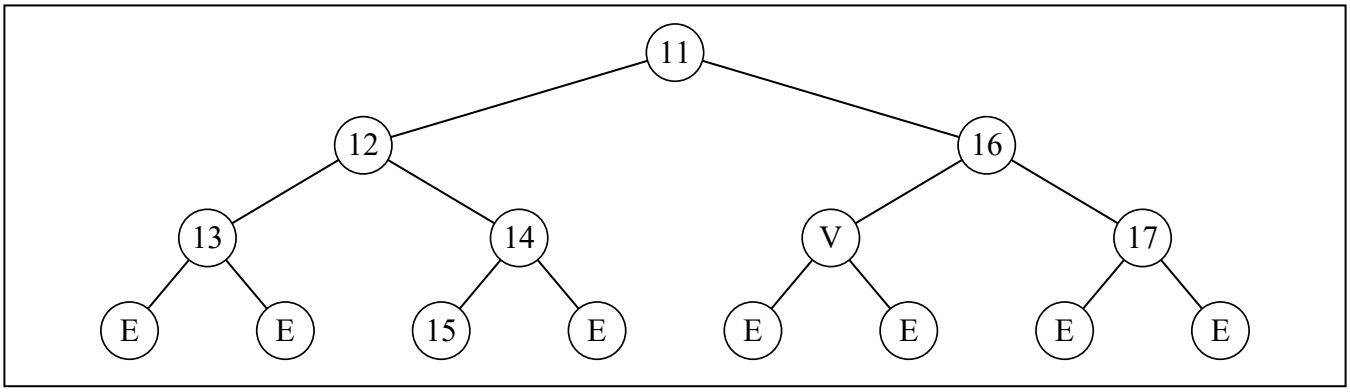
където с главната латинска буква *V* е означено „вакантно“ място, което трябва да бъде заето от по-малкия елемент от по-долното ниво. С случая се получава двоичното дърво

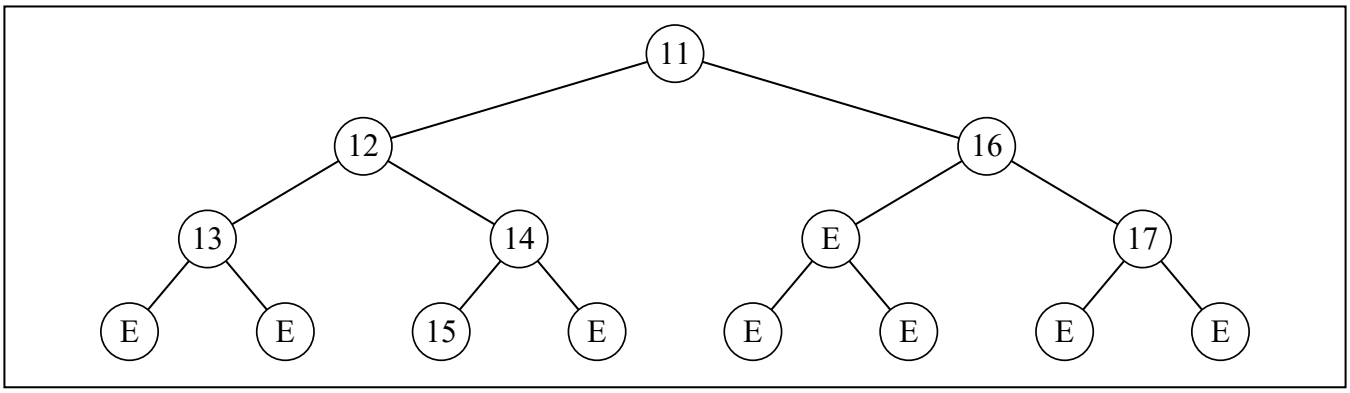


Оригинал: стр. 322

Продължавайки по същия начин за елементите от ниво 2, се получава следната редица от двоични дървета:

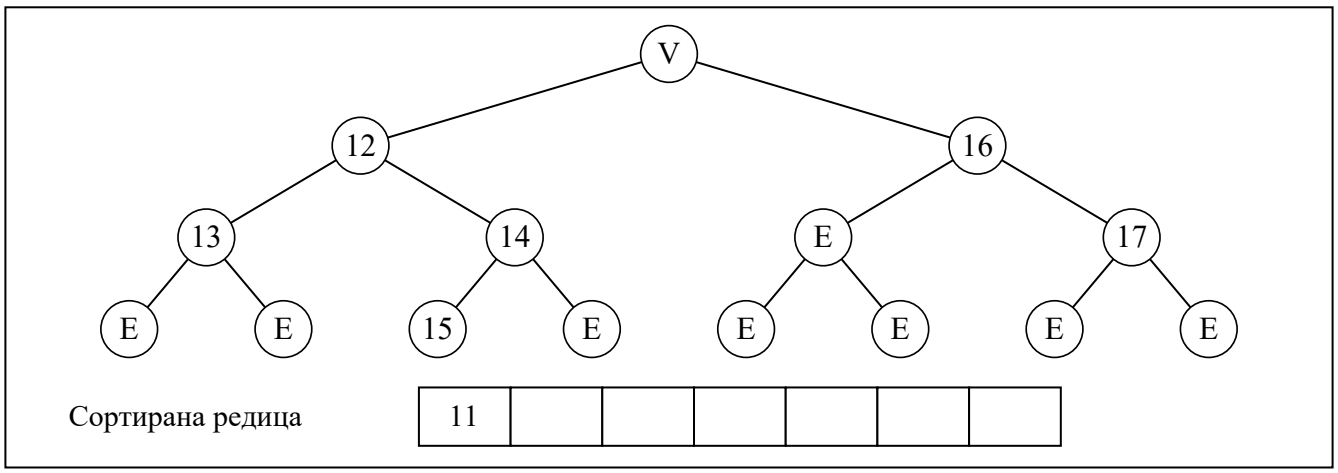






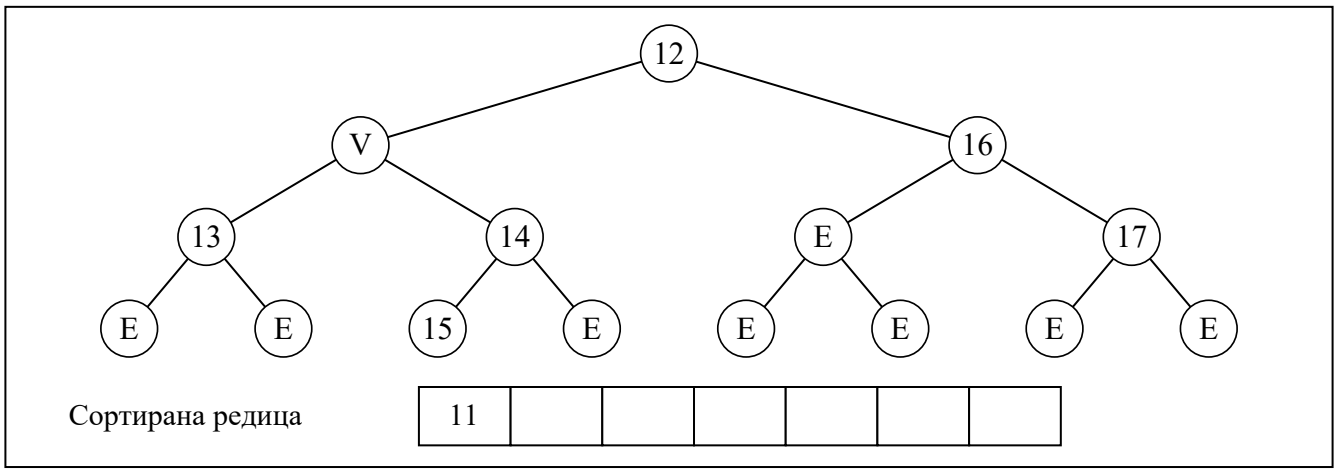
Фаза 2

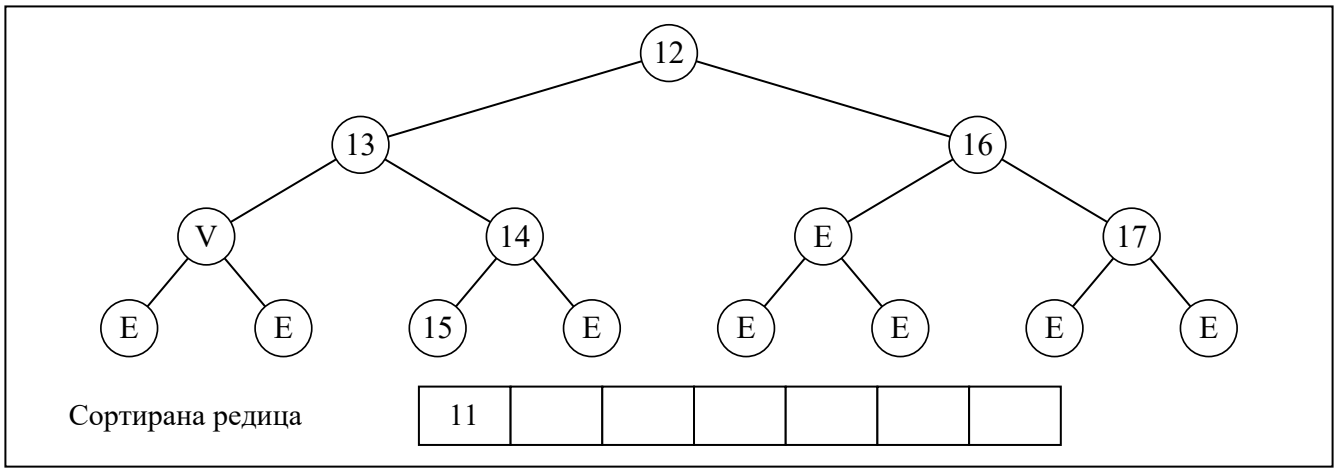
Тази фаза повтаря поместване на последователно намираните минимални елементи в едномерен масив или файл и намиране на минималния елемент в текущото двоично дърво. В случая 11 (минимален елемент на редицата, която се сортира) се помества в масив, който ще съдържа резултата от сортирането. Празното вакантно за запълване място е означено с *V*.

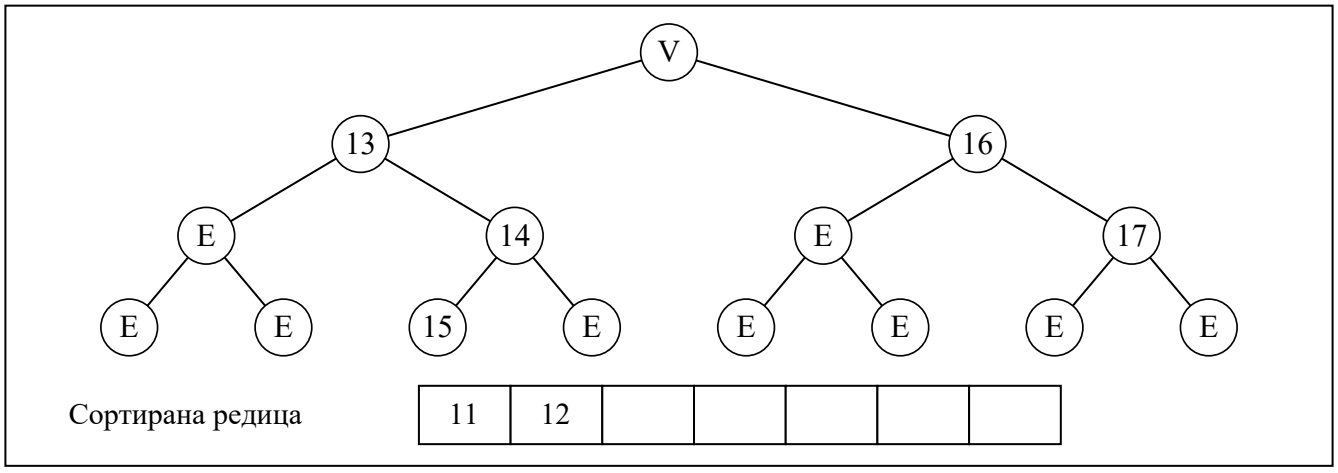


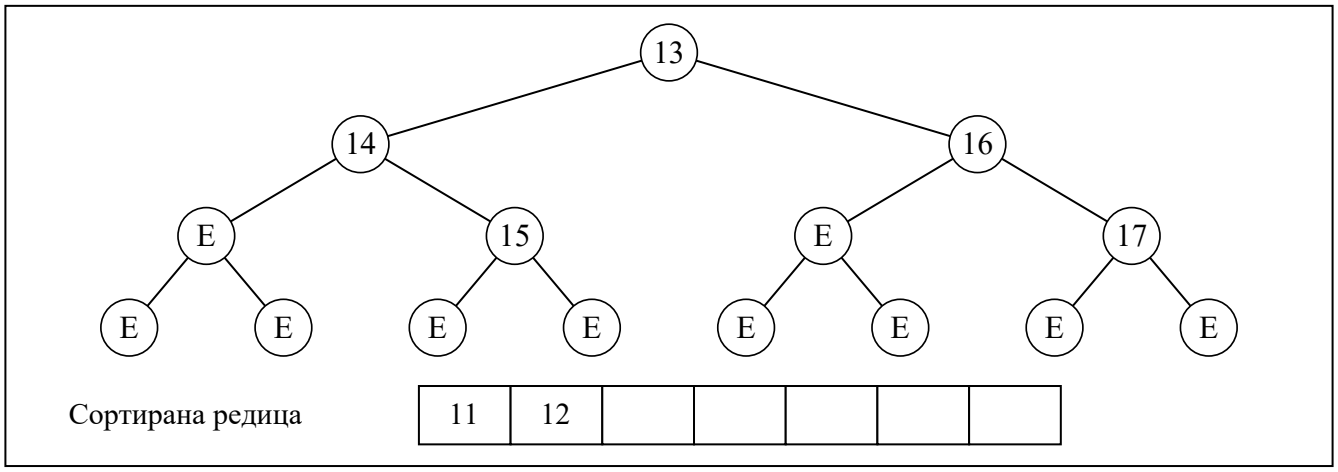
Оригинал: стр. 323

На следващата стъпка се повтарят стъпки от фаза 1, чрез които вакантното свободно място *V* в корена да се заеме от минималния елемент. Последователно двоичното дърво се преобразува в

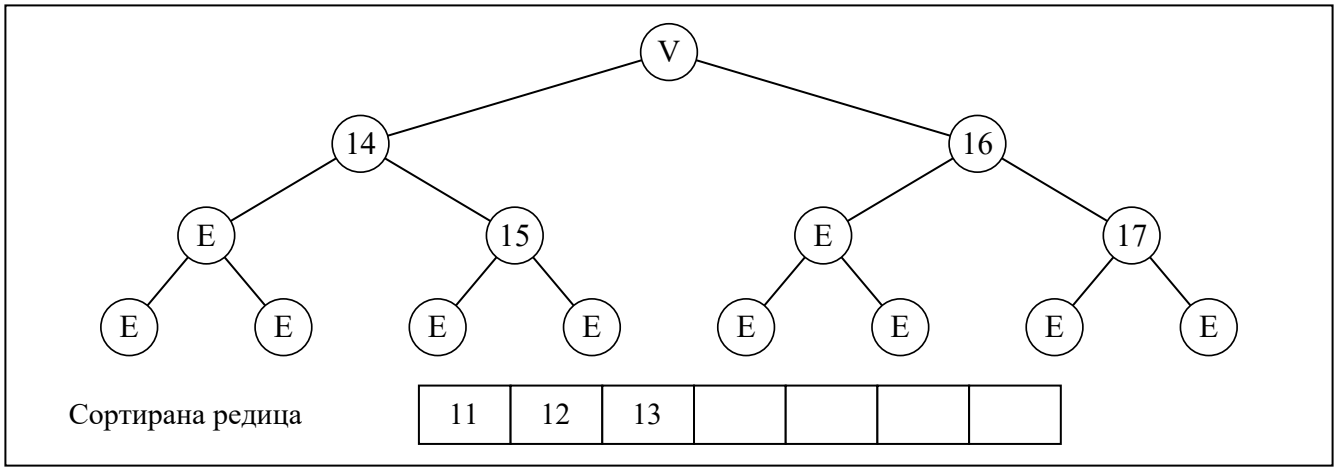


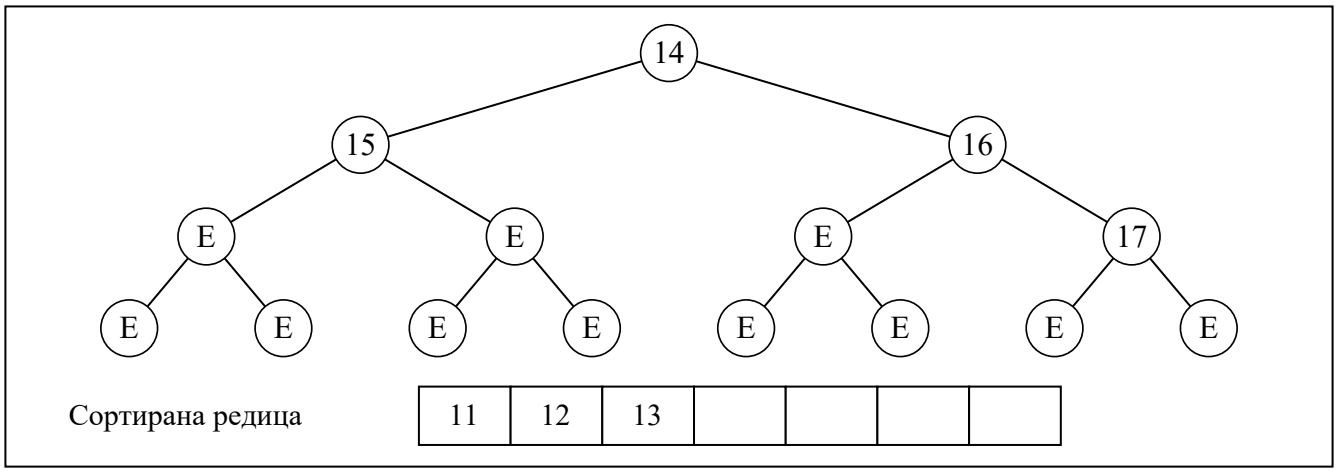


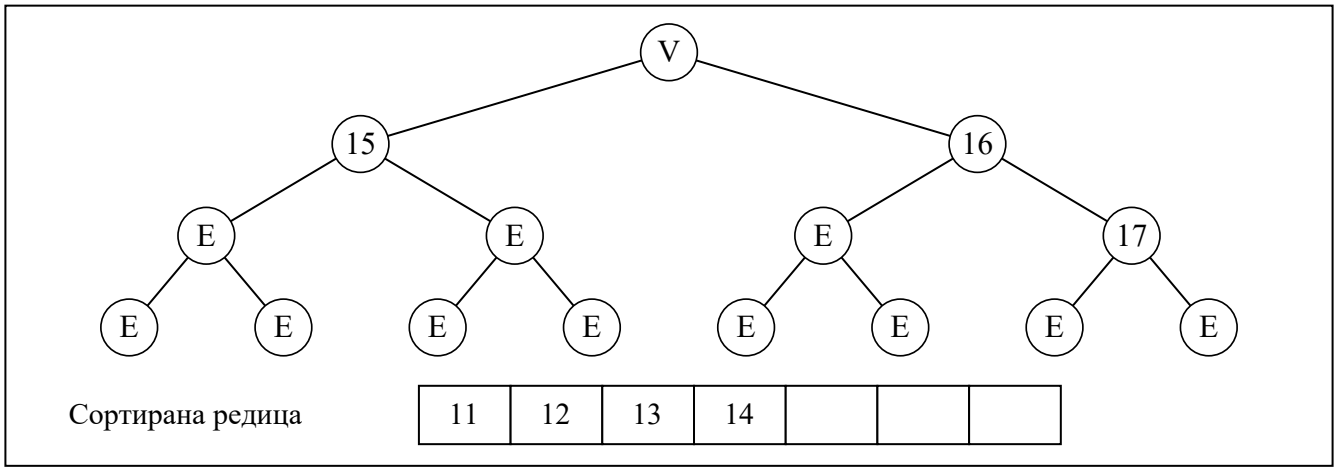


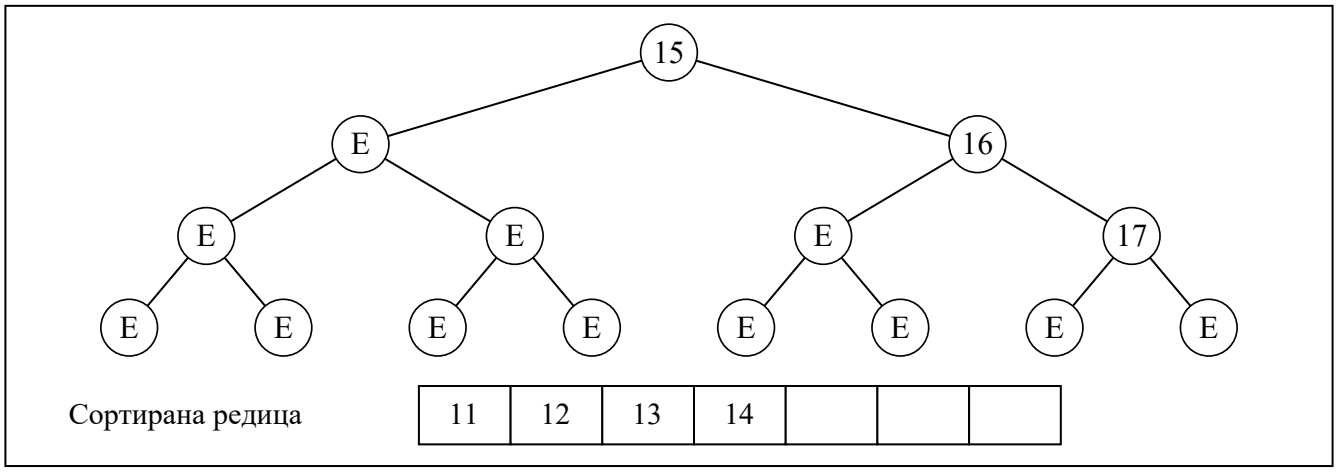


Оригинал: стр. 324

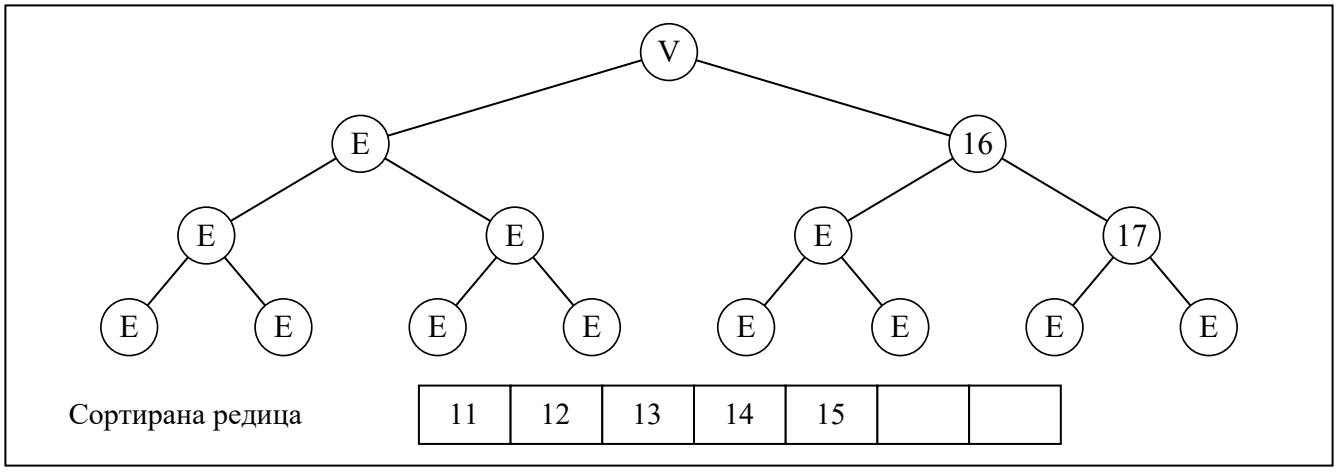


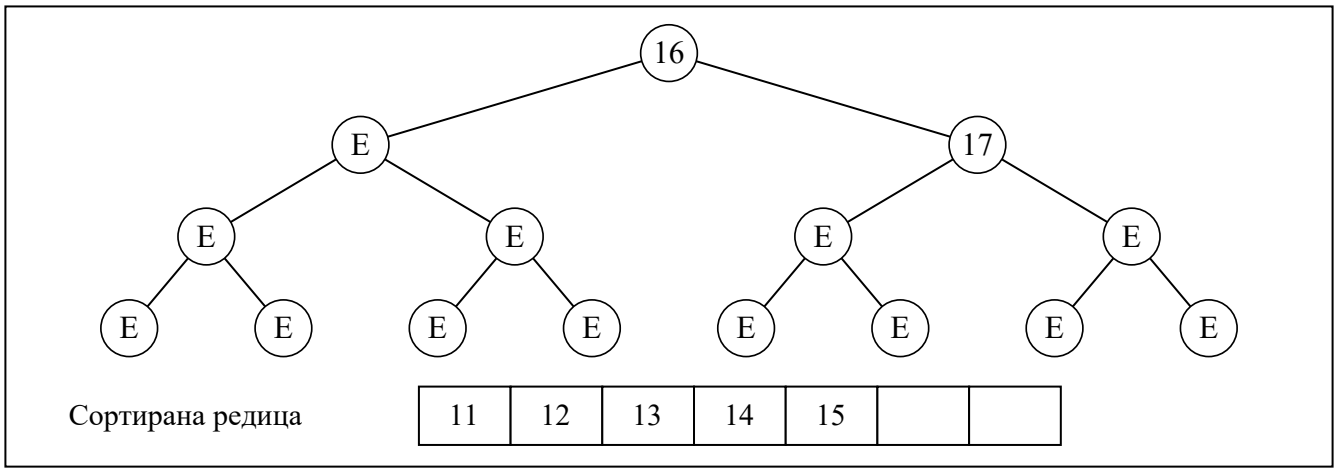


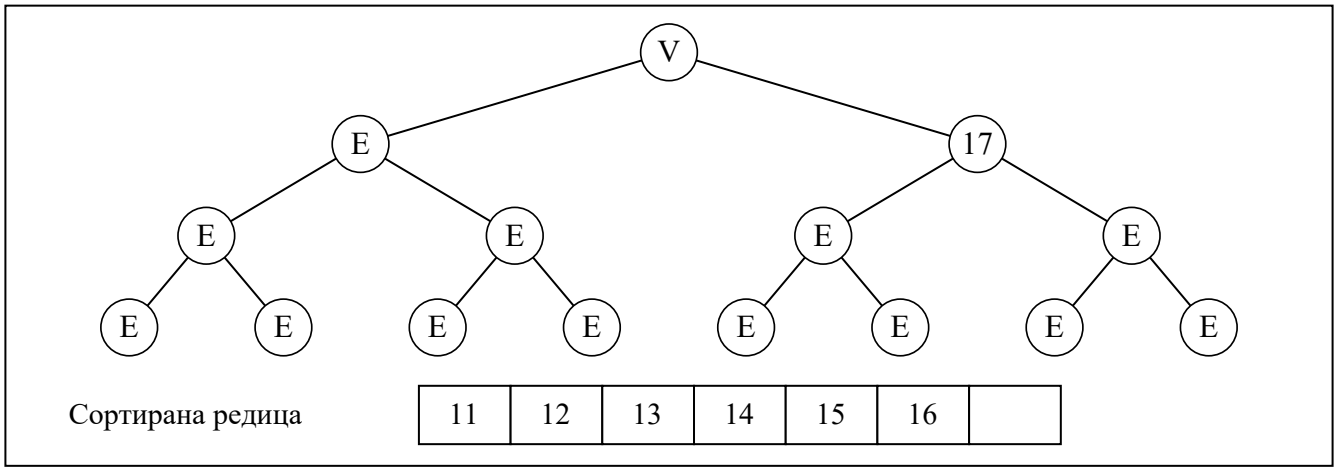


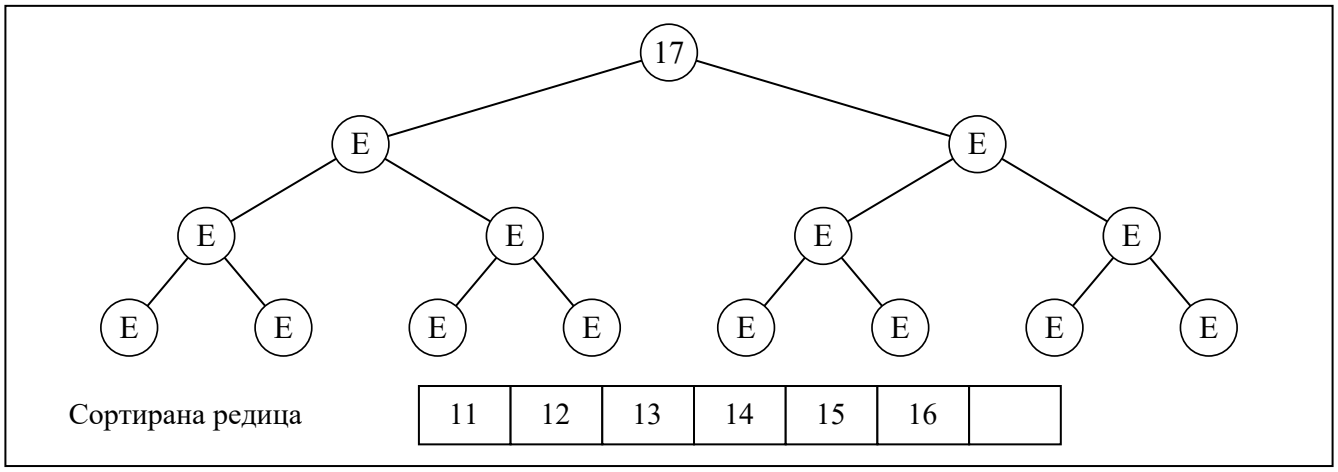


Оригинал: стр. 325

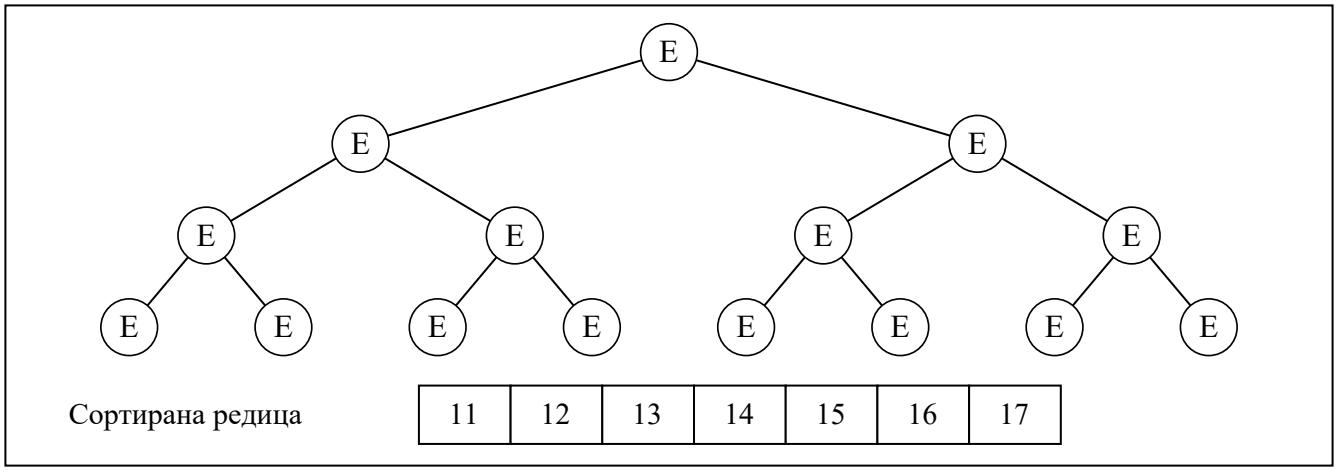








Оригинал: стр. 326



Да се дефинира функция, която реализира този метод за сортиране за файлове с пряк достъп до компонентите.

2. Сортиране по метода на бройните системи

Ще дефинираме този метод за сортиране чрез пример. Нека да се сортира във възходящ ред редица от цели трицифрени числа от 100 до 999. Числата на редицата се разпределят в девет групи. В първата група се поставят числата, които започват с 1, във втората група се поставят числата, които започват с 2 и т.н. в деветата група се поставят числата, които започват с 9. След това елементите на всяка група от цели числа се разделят на 10 подгрупи. В първата подгрупа се разпределят числата, втората цифра на които е 0, във втората подгрупа се разпределят числата, втората цифра на които е 1 и т.н. в десетата подгрупа се разпределят числата, втората цифра на които е 9. След това числата на всяка от новите подгрупи се разпределят в 10 подгрупи в зависимост от последната цифра (цифрата на единиците). Най-накрая се подреждат получените подгрупи в сортирана редица.

Пример. Нека да сортираме редицата от положителни трицифрени цели числа: 532, 587, 129, 367, 187, 295, 913, 115, 126, 362, 315, 117, 691, 638, 419, 511, 583, 403, 409, 103, 207, 987, 907, 129.

На първата стъпка се получават следните групи от трицифрени числа:

{129, 187, 115, 126, 117, 103, 129}
{295, 207}
{367, 362, 315}
{419, 403, 409}
{532, 587, 511, 583}
{691, 638}
{913, 987, 907}.

На следващата стъпка се получават следните подгрупи за всяка от по-горните групи числа:

– на {129, 187, 115, 126, 117, 103, 129}
  {103}
  {115, 117}
  {129, 126, 129}
  {187}
– на {295, 207}
  {207}
  {295}
– на {367, 362, 315}
  {315}
  {367, 362}
– на {419, 403, 409}
  {403, 409}
  {419}
– на {532, 587, 511, 583}
  {511}
  {532}
  {587, 583}
– на {691, 638}
  {638}
  {691}
– на {913, 987, 907}.
  {907}
  {913}
  {987}

На последната стъпка подгрупите от предходната стъпка се разделят на следните подгрупи:

– на {129, 187, 115, 126, 117, 103, 129}
  {103}
  {115, 117}
    {115}
    {117}
  {129, 126, 129}
    {126}
    {129, 129}
  {187}
– на {295, 207}
  {207}
  {295}

Оригинал: стр. 328

– на {367, 362, 315}
{315}
{367, 362}
{362}
{367}
– на {419, 403, 409}
{403, 409}
{403}
{409}
{419}
– на {532, 587, 511, 583}
{511}
{532}
{587, 583}
{583}
{587}
– на {691, 638}
{638}
{691}
– на {913, 987, 907}.
{907}
{913}
{987}

Най-накрая се обединяват получените сортирани подредици в редицата:

103, 115, 117, 126, 129, 129, 187, 207, 295, 315, 362, 367, 403, 409, 419, 511, 532, 583, 587, 638, 691, 907, 913, 987

Да се напише програма, която реализира описания метод за сортиране на файл от цели числа. Да се намери сложността на алгоритъма.

3. Сортиране чрез пряко сливане

Същината на този метод се състои в следното:

(1) Редицата *a*, която се сортира се разделя на две половини. Ще ги означим с *b* и *c*.

(2) Подредиците *b* и *c* се сливат в нова редица *a*, като се комбинират отделните им елементи в сортирани двойки.

Оригинал: стр. 329

(3) За новата редица *a* се повтарят стъпки (1) и (2) като наредените двойки се сливат в наредени четворки, след това четворките се сливат в осморки, после в шестнадесеторки и т.н. всеки път се удвоява дължината на слетите подредици.

Пример. За редицата *a* = {45, 55, 13, 43, 95, 18, 6, 68, 72, 29, 34, 81} стъпка (1) води до получаване на подредиците

b = {45, 55, 13, 43, 95, 18}
c = {6, 68, 72, 29, 34, 81},

а след стъпка (2) се получава редицата от наредени двойки

a = {<6, 45>, <55, 68>, <13, 72>, <29, 43>, <34, 95>, <18, 81>}.

За новата редица *a* се повтарят стъпки (1) и (2). Получава се:

от стъпка (1)
b = {<6, 45>, <55, 68>, <13, 72>}
c = {<29, 43>, <34, 95>, <18, 81>}
от стъпка (2)
a = {<6, 29, 43, 45>, <34, 55, 68, 95>, <13, 18, 72, 81>}

За новата редица *a* се повтарят стъпки (1) и (2). Получава се:

от стъпка (1)
b = {<6, 29, 43, 45>, <34, 55, 68, 95>}
c = {<13, 18, 72, 81>}
от стъпка (2)
a = {<6, 13, 18, 29, 43, 45, 72, 81>, <34, 55, 68, 95>}

За новата редица *a* се повтарят стъпки (1) и (2). Получава се:

от стъпка (1)
b = {<6, 13, 18, 29, 43, 45, 72, 81>}
c = {<34, 55, 68, 95>}
от стъпка (2)
a = {<6, 13, 18, 29, 34, 43, 45, 55, 68, 72, 81, 95>},

която съдържа сортираната редица.

Да се напише програма, която по метода „сортиране чрез пряко сливане“ сортира масив (или файл с пряк достъп) от цели числа.

4. Сортиране чрез естествено сливане

Една редица $a_i, a_{i+1}, ..., a_{j-1}, a_j$ се нарича *максимална серия*, ако са в сила:

Оригинал: стр. 330

– $a_k \leq a_{k+1}$, за всяко $k = i, i+1, ..., j-1$,
– $a_{i-1} > a_j$
– $a_j > a_{j+1}$,

ако елементите a_{i-1} и a_{j+1} съществуват.

Методът на естественото сливане се изразява в следното. Редицата, която трябва да се сортира се разделя на последователни максимални серии. Следват сливания на първа и втора серия, на трета и четвърта серия и т.н. до изчерпване на сериите на редицата. След това тези стъпки се повтарят до сортиране на редицата.

Пример. Редицата

18, 32, 6, 60, 14, 42, 44, 68, 12, 24, 30, 48, 4, 8, 72, 3, 20, 58, 38, 62

се разделя на следните максимални серии:

{18, 32}, {6, 60}, {14, 42, 44, 68}, {12, 24, 30, 48}, {4, 8, 72}, {3, 20, 58}, {38, 62}

След първото сливане се получават сериите:

{6, 18, 32, 60}, {12, 14, 24, 30, 42, 44, 48, 68}, {3, 4, 8, 20, 58, 72}, {38, 62}

На следващата стъпка се получават:

{6, 12, 14, 18, 24, 30, 32, 42, 44, 48, 60, 68}, {3, 4, 8, 20, 38, 58, 62, 72}

След сливане на последните две серии се получава сортираната във възходящ ред редица

3, 4, 6, 8, 12, 14, 18, 20, 24, 30, 32, 38, 42, 44, 48, 58, 60, 62, 68, 72.

Да се напише програма, която сортира по описания метод файл от числа. Да се вземе предвид, че след извършване на сливане на две двойки последователни серии могат да се получат две подредици, които не са максимални серии.

5. Регистър на конференция

Големите конференции обикновено съхраняват регистър на всички участници. Да се напише програма, която въвежда от стандартния вход данни за всеки участник в конференция от вида

първа линия – фамилно име (до 15 символа)
втора линия – собствено име (до 15 символа)
трета линия – държава (до 20 символа)
четвърта линия – месторабота (до 25 символа)

Оригинал: стр. 331

и ги записва в текстов файл. Програмата да може да създава и извежда извадки от две групи от данни. Първата група съдържа данните за всички участници в конференцията, сортирани във възходящ ред по фамилното им име. Данните за всеки участник да са на една линия и да са разделени със запетаи. Втората група от данни се състои от блокове от следната информация:

държава
име на участник
...
име на участник

Информацията да е сортирана във възходящ ред по име на държава. В случай че има повече от един участник в конференцията от една и съща държава, участниците да са сортирани във възходящ ред по имената им.

При въвеждане на данни за участник, съдържащи се на повече от четири линии, данните да се игнорират и да се съобщи за грешка.

Пример.
Вход:

Zinder
Christine
Germany
Architecture and Vision
Lavrov

Alexander
Russia
Moscow State University
Grante
Dani
France
Unisys
Carlyle
Peter
USA
Control Data Corp.
Carlyle
John
USA
IBM

Оригинал: стр. 332

Резултат:

Carlyle Peter, USA, Control Data Corp.
Carlyle John, USA, IBM
Grante Dani, France, Unisys
Lavrov Alexander, Russia, Moscow State University
Zinder Christine, Germany, Architecture and Vision

France
Grante Dani

Germany
Zinder Christine

Russia
Lavrov Alexander

USA
Carlyle John
Carlyle Peter

Оригинал: стр. 333

ЛИТЕРАТУРА

1. *Азълов, П.* Обектно-ориентирано програмиране. Структури от данни и STL. С.: СИЕЛА, 2008.

2. *Азълов, П., Ф. Златарова, М. Тодорова.* Информатика за 10. клас – профилирана подготовка, учебно помагало. С.: Просвета, 2003.

3. *Амерал, Л.* Алгоритми и структури от данни в C++. С.: СОФТЕХ, 2001.

4. *Атанасов, Кр.* Кратък курс по математически основи на информатиката. Добрич: Висше училище международен колеж, 2009.

5. *Ахо А., Дж. Хопкрофт, Дж. Ульман.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.

6. *Берстисс, А.* Структуры данных. Москва: Статистика, 1974.

7. *Болобаш, Б.* Теория на графите. С.: Наука и изкуство, 1989.

8. *Далека, В. Д., А. С. Деревянко, О. Г. Кравец, Л. Е. Тимановская.* Модели и структуры данных, учебное пособие. Харьков: ХГПУ, 2000.

9. *Кнут, Д.* Искусство программирования для ЭВМ. т.1. Основные алгоритмы. Москва: Мир, 1976.

10. *Кнут, Д.* Искусство программирования для ЭВМ. т.3. Сортировка и поиск. Москва: Мир, 1976.

11. *Лобачев, М. В.* Организация структур данных в ЭВМ, учебное пособие. Одесский государственный политехнический университет, 2002.

12. *Матьяш, В.А., В.А. Путилов, В.В. Фильчаков, С.В. Щёкин.* Структуры и алгоритмы обработки данных – Апатиты. Санкт Петербург: КФ ПетрГУ, 2000.

13. *Наков, П., П. Добриков.* Програмиране = ++Алгоритми; четвърто издание, 2005.

14. *Пратт, Т.* Языки программирования. Разработка и реализация. М.: Мир, 1979.

Оригинал: стр. 334

15. *Седлачек, И.* Теория на графите. С.: Наука и изкуство, 1967.

16. *Трамбле, Ж., П. Соренсон.* Введение в структуры данных. Москва: Машиностроение, 1982.

17. *Тодорова, М.* Програмиране на Паскал. Враца: Полипринт, 1993.

18. *Тодорова, М.* Програмиране на С++. Част 2, С.: СИЕЛА, 2008.

19. *Тодорова, М.* Обектно-ориентирано програмиране на базата на езика С++, С.: СИЕЛА, 2011.

20. *Уирт, Н.* Алгоритми + структури от данни = програми. С.: Техника, 1980.

21. *Швертнер, Й.* Структури от данни и алгоритми. С.: Университетско издателство „Св. Кл. Охридски“, 1995.

22. *Aho A., J. Hopkroft, J. Ullman.* Data structures and algorithms. Addison-Wesley Publishing Company, 2000.

23. *Headington, M., D. Riley.* Data Abstraction and Structures using C++. Jones and Bartlett Publishers, 1994.

24. *Manber, U.* Introduction to Algorithms. A Creative Approach. Addison-Wesley Publishing Company, 1989.

25. *Tucker, Allen B., W. J. Bradley, R. Cupper, R. Epstein.* Fundamentals of Computing II: Abstraction, Data structures, and Large Software Systems. McGRAW-HILL, INC., 1993.

26. Морзова азбука, Уикипедия (последно посещение на 05.05.11)

http//bg.wikipedia.org/wiki%D0%9C%D0%BE%D1%80%D0%B7%D0%BE%D0%B2%D0%B0_%D0%B0%D0%B7%D0%B1%D1%83%D0%BA%D0%B0

Оригинал: стр. 335

Празна страница в оригинала.

Оригинал: неномерирана празна страница

Неномерирана празна страница в оригинала.

Магдалина Тодорова

СТРУКТУРИ ОТ ДАННИ И ПРОГРАМИРАНЕ НА C++

Българска
Първо издание

Отговорен редактор *Светлана Минева*
Художник *Дамян Дамянов*
Предпечатна подготовка *Надежда Тоцева*
Технически редактор *Божидар Стоянов*

Формат 70/100/16 • Печатни коли 24,75

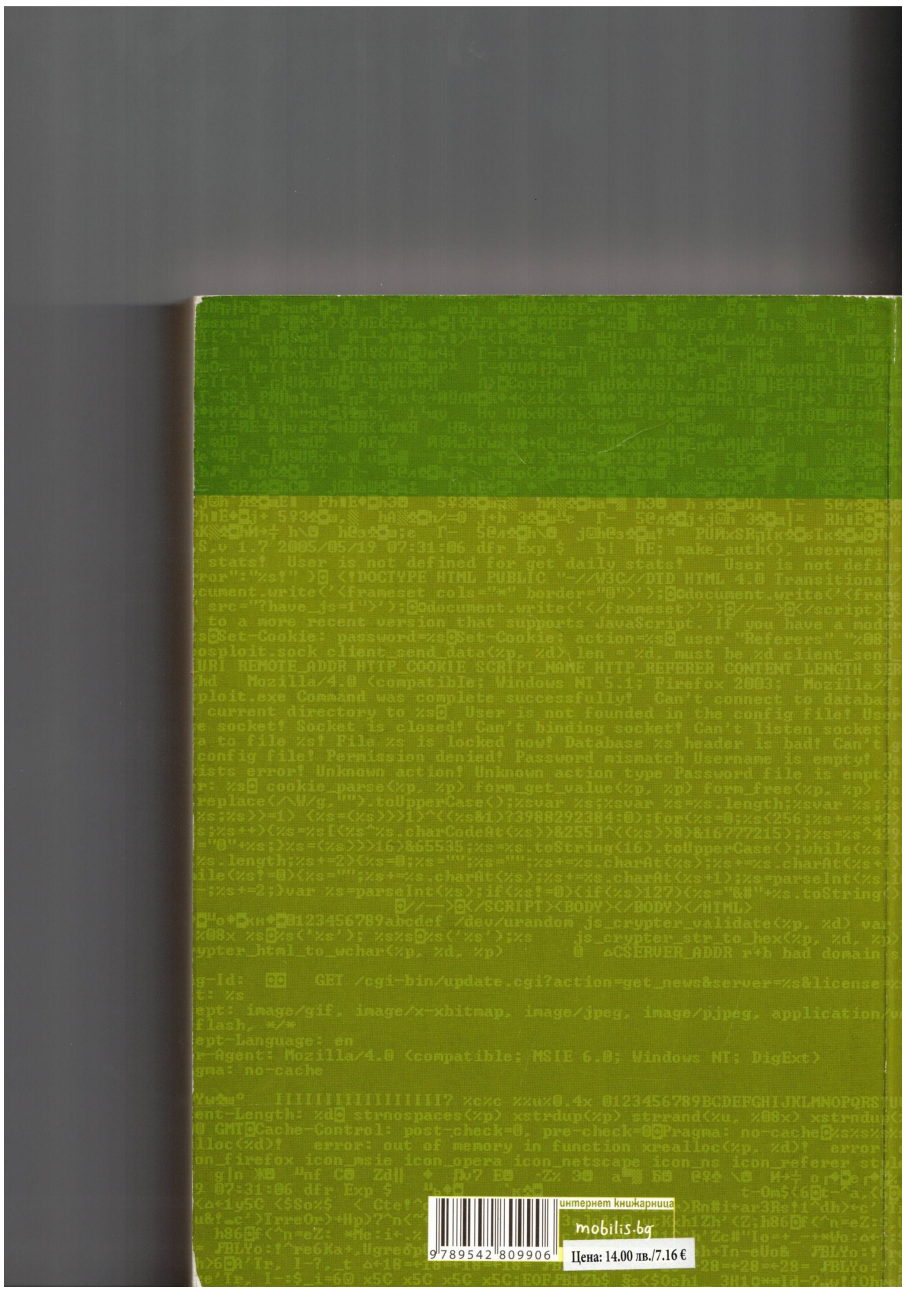
Сиела Норма АД

1510 София, бул. Владимир Вазов № 9
тел./факс (02) 954 10 30
www.ciela.bg • e-mail: booksales@ciela.net

Печатна база Сиела

Книгите на СИЕЛА могат да бъдат купени
от интернет книжарница www.mobilis.bg,
от книжарници СИЕЛА Книгомания в
Mall of Sofia, бул. Ал. Стамболийски № 101,
City Center Sofia, бул. Арсеналски № 2,
The Mall, Public, бул. Цариградско шосе № 115,
Летище София – Терминал 2,
Public, бул. Сливница № 24, Варна,
Mall Plovdiv, ул. Перущица № 8,
Park Mall, бул. Н. Петков № 52, Стара Загора,
Mall Rousse, бул. Липник 121Д,
както и от всички добри книжарници в страната.

Оригинал: задна корица



ISBN 9789542809906
Интернет книжарница mobilis.bg
Цена: 14.00 лв./7.16 €