СЛАВЧО ЧУНГУРСКИ

ПОДАТОЧНИ СТРУКТУРИ
+ АЛГОРИТМИ
_____
= ПРОГРАМИРАЊЕ
+ ИМПЛЕМЕНТАЦИЈА ВО C++

СКОПЈЕ, 2011

СОДРЖИНА

Предговор

Оваа книга е наменета за едносеместрален курс од компјутерските науки кој ги третира основните податочни структури и продолжува кон развој на посложени структури и анализи на алгоритми. Покривањето на овие теми во модерните студиски програми од областа на компјутерските науки е неминовно така што, генерално, постои некаков консензус за покривање на темите во овој курс од страна на факултетите, но сепак постои и одредено несогласување во врска со детали кои би се опфатиле во самиот курс.

Во морето од интернационални факултети и студиски програми од областа на компјутерските науки најжестоки дебати се водат околу изборот на вистинскиот јазик со кој најдобро би се проучиле овие теми, така што дилемата се појави и пред пишувањето на оваа книга, дали да биде C++ или да биде Java, што и во двата случаи е подеднакво достапно во странската литература. Се појави дилема и за тоа дали да се оди на некој модерен програмски јазик како што е C#, но со оглед на тоа што тој поседува огромни библиотеки со готови имплементации на податочни структури, изборот сепак падна на C++.

Целта за пишување на оваа книга е да се обезбеди практичен вовед во податочните структури и алгоритми од аспект на апстрактно размислување и решавање на проблеми, така што направен е обид од страна на авторот да се покријат сите поважни детали кои се однесуваат на податочните структури, нивната анализа и нивните C++ имплементации, без посебен осврт на структурите кои се интересни во теоријата, но не се користат во пракса.

Од аспект на претходно стекнати познавања, студентите кои ќе ја користат оваа книга треба да имаат познавање и од објектно-ориентираното и од процедуралното програмирање, задолжително вклучувајќи ги основните карактеристики, како што се примитивни податочни типови, оператори, контролни структури, функции, процедури, рекурзија и слично.

Студентите кои пред овој курс немаат учено C++, препорачливо е прво да ја проучат основната синтакса на овој јазик, а потоа да продолжат со оваа книга, затоа што таа опфаќа некои навистина сложени C++ детали.

Познавањето на дискретна математика е од голема помош, но не е задолжително, затоа што математичките поими и дефиниции, во помала или поголема мерка, се образложени пред самата дефиниција и имплементација на самите податочни структури и алгоритми.

Кодот кој е прикажан во оваа книга е функционален и тестиран, и истиот може да се изврши во било кој стандарден C++ компајлер.

Вовед

Ако погледнеме наоколу ќе забележиме дека компјутерите и мрежите се насекаде, овозможувајќи сложена мрежа од комплексни активности на човештвото: образование, бизнис, забава, истражување, производство и многу други. Сето ова е заслуга на две главни технолошки основи, првата е очигледна, неверојатниот напредок на микроелектрониката и чип дизајнот придонесуваат до **è** побрз и побрз ха рдвер, а другата е нешто што и не е толку очигледно како првото, но во никој случај не е помалку важно, а тоа е интелектуалниот склоп кој ја надахнува компјутерската револуција олицетворен во ефикасни алгоритми.

Слободно може да се каже дека две идеи го променија светот. Во 1448 во германскиот град Мајнц, златарот со име Johann Gutenberg откри начин како да се печатат книги со составување на подвижни метални парчиња. Писменоста се рашири, заврши темното доба, интелектот на човекот се ослободи, науката и технологијата триумфираа, се случи индустриската револуција, па со право многу историчари можат да кажат дека сето ова го должиме на типографијата, но многу други сметаат дека клучниот елемент не е типографијата, туку алгоритмите.

Ако се навратиме назад на периодот кога се користеле римските броеви, тогаш прашањето е како би било собирањето на два броја, или можеби множењето. Комплицирано, секако. Децималниот систем бил револуционерно откритие кое се случило во Индија на почетокот од седмиот век. Се користат 10 симболи, со помош на кои дури и многу големи броеви можат компактно да се запишат, а аритметичките операции можат ефикасно да се извршуваат со последователни елементарни чекори. Ова е разултат на добро осмислена податочна структура која претставува некаков број, а во нејзини рамки се дефинирани операции.

Овој децимален систем и неговите нумерички алгоритми одиграа важна улога кај западните цивилизации, што доведе до развој на науката и технологијата и ги забрзаа индустријата и бизнисот, за да на крај, многу подоцна, конечно да се појави компјутерот и во него да се преточи целиот тој систем овозможувајќи им на научниците низ целиот свет да развиваат се покомплексни и покомплексни алгоритми за секакви видови на проблеми и да откриваат нови апликации безусловно менувајќи го светот.

Основни поими

Кај програмирањето се сретнуваме со два основни аспекти, и тоа статички аспект на некоја програма, кој го претставува она со што се работи

во програмата и динамички аспект на некоја програма, кој го претставува она што се работи во програмата. Статичкиот аспект го претставуваат податочните структури кои се користат во програмата, додека алгоритмите го претставуваат динамичкиот аспект. Оттука доаѓа и името на курсевите кои ги третираат овие содржини, како на пример „Податочни структури и алгоритми“ кој детално ги опфаќа и двата аспекти на програмирањето. Целта на овој курс е студентот да научи подобро да програмира преку запознавање на техниките за различните начини на складирање на информациите, како и различните алгоритми.

Податочните структури и алгоритмите се во тесна врска. Не е можно да се зборува за едното, а да не се спомене другото. Овој курс ја третира токму оваа врска, ќе биде разработено влијанието на податочните структури над алгоритмите за нивна подобра обработка, но и обратно, ќе се види како алгоритмите влијаат на изборот на соодветна податочна структура за приказ на своите податоци. На овој начин, оваа врска ќе придонесе за запознавање на низа важни идеи и поими кои практично ги чинат основите на програмирањето. Покрај овие два главни термини „„податочни структури“ и „алгоритми“, овој курс користи уште неколку наизглед слични поими.

Тип на податоци – множество вредности кои некој податок може да ги прими. Ваков пример е типот ***инт*** кој ги содржи само вредностите од множеството цели броеви, достапни кај компјутерот.

Апстрактен податочен тип (АТП) – се задава со наведување на еден или повеќе типови на податоци, но и една или повеќе операции (функции). Операндите и резултатите од операциите се податоци од наведени типови, а меѓу типовите постои еден по кој целиот апстрактен податочен тип го добива името.

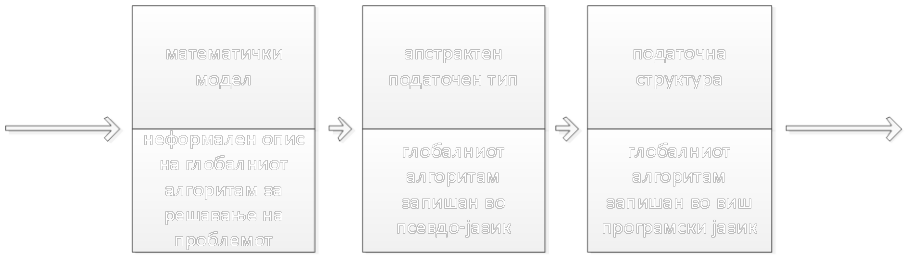
Податочна структура – множество променливи во некоја програма и врска меѓу тие променливи.

Алгоритам – конечна низа инструкции од кои секоја има јасно значење и мже да биде извршена во конечно време. Исти инструкции може да се извршат повеќе пати, под претпоставка дека самите тие укажуваат на повторување. Сепак, потребно е, за било кои вредности на влезните податоци, алгоритмот да заврши по извршување на конечен број на повторувања.

Имплементација на аптрактен податочен тип – конкретна реализација на одреден апстрактен податочен тип во некоја програма. Се состои од дефиниции на структурата на податоци, со која се прикажуваат податоците од апстрактниот податочен тип, но и потпрограми, со кои се остваруваат операциите кај апстрактниот податочен тип со помош на

избрани алгоритми. За ист апстрактен податочен тип обично се можни повеќе различни имплементации кои се разликуваат по тоа што користат различни структури за приказ, но и различни алгоритми за извршување на операциите.

Целта на овој курс е да овозможи унапредување на техниките на програмирање, па развојот на една програма може да се разгледува во согласност со дијаграмот прикажан на сликата.



Развој на програма за решавање некој проблем

Од дијаграмот се гледа улогата на апстрактните податочни типови, податочните структури, како и алгоритмите за постапката на решавање на проблемот. Исто така, се гледа дека програмирањето во подеднаква мера се состои од развој на податочни структури, како и од развој на алгоритми.

Како пример, за апстрактен податочен тип претставен е тип кој одговара на математичкиот поим за комплексни броеви.

Апстрактен податочен тип – COMPLEX

scalar – било кој тип за кој се дефинирани операциите за собирање и множење

COMPLEX – податоците од овој тип се парови од типот scalar

ADD(z1, z2, &z3) – операција за собирање на комплексни броеви. За зададени z_1, z_2 од типот COMPLEX се пресметува нивниот збир z_3, исто така од типот COMPLEX. Значи, за z_1 од облик (x_1, y_1) и z_2 од облик (x_2, y_2) се добива z_3 од облик (x_3, y_3) за кои важи $x_3=x_1+x_2$ и $y_3=y_1+y_2$.

MULT(z1, z2, &z3) – операција за множење на комплексни броеви. За зададени z_1, z_2 од типот COMPLEX се пресметува нивниот производ z_3, исто така од типот COMPLEX. Значи, за z_1 од облик (x_1, y_1) и z_2 од облик (x_2, y_2) се добива z_3 од облик (x_3, y_3) за кои важи $x_3=x_1x_2-y_1y_2$ и $y_3= x_1y_2+ x_2y_1$.

Погодна структура за податоци за приказ на комплексни броеви би била од тип кој би бил соодветен за пар од скаларни вредности *(re, im)* каде

re би го означувал реалниот дел од комплексен број, а *im* би го означувал имагинарниот дел од комплексен број, па структурата во C++ синтакса би била:

```
typedef struct
        {
        scalar re;
        scalar im;
        } COMPLEX;
```

Имплементацијата на апстрактниот податочен тип COMPLEX се состои од претходната дефиниција на типот, т.е. од функциите:

```
void ADD (COMPLEX z1, COMPLEX z2, COMPLEX *z3) {
        z3->re = z1.re + z2.re;
        z3->im = z1.im + z2.im;
        }

void MULT (COMPLEX z1, COMPLEX z2, COMPLEX *z3) {
        z3->re = z1.re*z2.re – z1.im*z2.im;
        z3->im = z1.re*z2.im + z2.re*z1.im;
        }
```

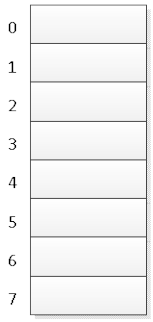
За апстрактниот податочен тип COMPLEX е важно да се присутни операциите ADD и MULT, затоа што без нив би се работело за тип, и во овој случај комплексните броеви не би можеле да ги разликуваме од обичните парови скаларни вредности. Дополнителната димензија на апстрактноста се состои во тоа што не е до крај одредено што е типот scalar, што во овој случај не е важно кога се работи за проучување на податочните структури и алгоритмите.

Градба на податочни структури

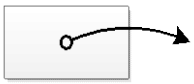
Податочните структури се состојат од помали целини кои се здружуваат во поголеми структури и се поврзуваат меѓу себе. Овде се воведуваат посебни називи за целини, начини на здружување како и начини на поврзување на податоците. Во продолжение е даден опис на елементите од кои се градат податочните структури, а прикажана е и нивната графичка репрезентација.

Ќелија (cell) – променлива која се третира како посебна целина. Тоа е релативен поим, затоа што во еден контекст една иста целина може да се третира како ќелија, а во друг контекст на истата целина да се разгледува нејзината градба. Секоја ќелија си има свој тип и адреса.

Поле (array) – моќна податочна структура која сместува податоци во низа од променливи од ист податочен тип. Полето е составено од повеќе ќелии од ист тип кои се сместени во последователни мемориски адреси. Се вели дека полето е статичка податочна структура затоа што кога еднаш ќе биде декларирана, неговата големина која е специфицирана од страна на програмерот останува непроменета во текот на целата програма и не може да биде променета. Една ќелија се нарекува елемент на полето и еднозначно е определена со вредноста на индексот преку кој се пристапува до елементот. Кај различните програмски јазици индексите кај полињата може да бидат различно дефинирани, на пример во Pascal поле од n елементи би ги содржело елементите со индекси од 1 до n, додека во C поле од n елементи би било определено со индексите од 0 до n-1. Во продолжение на курсот ќе биде користена дефиницијата од програмскиот јазик C.



Запис (слог, record) – уште еден механизам за здружување на помали делови од структура во поголеми. Записот е составен од повеќе ќелии кои, за разлика од кај полето, не мора да бидат од ист тип, но кои се сместени на последователни адреси. Бројот, редоследот и типот на ќелиите се однапред зададени и непроменливи, а одделната ќелија се нарекува компонента на записот. Полињата и запидите може да се комбинираат, на пример, може да се состави поле од записи, запис од полиња, полиња од полиња, запис од записи и слично.



Покажувач (pointer) – служи за поставување на врска меѓу делови од структура. Покажувачот е ќелија која покажува кон некоја друга ќелија, така што содржината на покажувачот е адресата на ќелија кон која покажува тој.

Курсор – уште еден елемент кој служи за воспоставување врска меѓу два дела од структура. Курсорот е ќелија од типот *int* која покажува на елемент од некое поле, а неговата содржина е индексот на елементот кон кој покажува тој.

Запишување и анализирање на алгоритми

Како јазик за запишување на алгоритми ќе се користи програмскиот јазик C++. Ова значи дека во продолжение наместо да се пишува алгоритам, ќе биде пишуван програм (или функција) напишана во C++ која ќе работи според тој алгоритам.

Под анализа на алгоритам се подразбира проценка на времето за извршување на тој алгоритам. Времето го поистоветуваме со бројот на операции кои се потребни за да ги заврши програмот, и истото се озразува како функција од обликот *T(n)* каде *n* е некоја мерка за големина на множеството влезни вредности. На пример, кај алгоритмот кој сортира низа од реални броеви тогаш неговото време на извршување се означува во облик *T(n)* каде *n* е должината на низата реални броеви. Слично, ако се работи за алгоритам за пронаоѓање на инверзна матрица, тогаш неговото време на извршување се изразува како *T(n),* каде *n* е редот на матрицата.

Често се случува *T(n)* да зависи и од вредностите на влезните податоци, а не само од големината на истите. На пример, алгоритмот за сортирање може побргу да сортира низа од броеви која е скоро сортирана, отколку низа од броеви која е добро измешана. Тогаш *T(n)* се дефинира како време во најлош случај, т.е. како максимум за време на извршување по сите множества влезни податоци со големина *n*. Исто така, во вакви случаи, се разгледува и времето за просечна ситуација т.е. $T_{avg}(n)$, кое се добива како математичко очекување од времињата на извршување. За да може да се зборува за очекување, тогаш мора да се претпостави и распределбата на различните множества влезни податоци со големина n. Вообичаено се претпоставува рамномерна распределба, што значи дека веројатноста за појавување на било која комбинација влезни податоци е иста како на секоја друга.

Функцијата *T(n)* нема потреба да се одредува прецизно, затоа што е доволно да се определи нејзиниот ред на големина. На пример, времето за извршување на Гаусовата метода за наоѓање на инверзна матрица е $O(n^3)$, што означува дека $T(n) \leq const * n^3$. Во оваа ситуација не е прецизно утврдено колку секунди ќе трае програмата, но се знае дека ако редот на матрицата се зголеми двојно, тогаш инвертирањето би траело и до 8 пати подолго, т.е. ако редот на матрицата се зголеми тројно, инвертирањето би траело до 27 пати подолго и слично.

I дел

ПОДАТОЧНИ СТРУКТУРИ

1. Листи

1.1 Општа листа (List)

Листа е конечна низа од нула или повеќе податоци од ист тип. Податоците кои ја сочинуваат листата се нарекуваат нејзини елементи, што во теоретските истражувања обично се бележи на следниот начин:

$(a_1, a_2, \ldots, a_n)$

каде $n \geq 0$ и тоа е должина на листата. Ако $n=0$ тогаш се вели дека листата е празна. За $n \geq 1$, a_1 е првиот елемент на листата, a_2 е вториот итн., а a_n е последниот елемент.

Кај листите можно е некои елементи да се еднакви, но идентитетот на еден елемент не е одреден со неговата вредност, туку со неговата позиција т.е. со неговиот реден број.

Важно својство на листите е тоа што нивните елементи се линеарно подредени во однос на својата позиција. Според тоа слободно може да се каже дека елементот a_i е пред a_{i+1}, но и дека елементот a_i е зад a_{i-1}.

Бројот на елементите во една листа не е фиксен со оглед на тоа што во неа може да се додаваат и отфрлаат елементи на било кое место, што влијае на нејзината должина.

Еве неколку примери за листи:

- Еден збор во некој текст е листа од знаци. На пример, зборот СТРУКТУРА може да се интерпретира како листата (С, Т, Р, У, К, Т, У, Р, А). Од овој пример може да се забележи дека два различни елементи може да имаат иста вредност. Исто така еден ред во текст претставува листа од знаци, вклучувајќи ги и празните места, а целиот текст, пак, претставува листа од редови.
- Полиномот во математиката се бележи на следниот начин:

$$P(x) = a_1 x^{e_1} + a_2 x^{e_2} + \ldots + a_n x^{e_n}$$

каде $0 \leq e_1 \leq e_2 \leq \ldots \leq e_n$, но истиот може да се претстави и како листа од облик:

$((a_1, e_1), (a_2, e_2), \ldots, (a_n, e_n))$

- Кај некои програмски јазици, како што е LISP (LISt Processing) листата пак е основен објект од кој се изградени сите други објекти.

- Како резултат на SELECT израз кај базите на податоци, се добива листа од записи. На пример ако се постави израз со кој се бараат сите презимиња и имиња на вработени од IT сектор со нивните датуми на раѓање и кои се родени во 1976 година, сортирани по презимето, би се добил резултат од тип ((Благоевски, Виктор, 01.05.1976), (Милановски, Бобан, 28.05.1976), ..., (Трајковски, Тони, 29.01.1976))

1.1.1 Апстрактен податочен тип ЛИСТА

За да може математичкиот поим „листа“ да се претвори во апстрактен податочен тип, потребно е да се дефинираат операциите кои се извршуваат над листите. Ова може да се изведе на различни начини, така што апстрактниот податочен тип кој е претставен во продолжение е само една од можните варијанти.

IzbranTip – било кој тип на податоци.

Lista – податок од овој тип е низа податоци од типот *IzbranTip*.

pozicija – податокот од овој тип служи за идентификација на елемент во листа, т.е. за задавање на позиција во листа. Се претпоставува, дека во листата $(a_1, a_2, ..., a_n)$ се дефинирани позициите кои одговараат на првиот, вториот, ..., n-тиот елемент, но и позицијата на крајот од листата, која е зад n-тиот елемент.

Kraj(L) – функција која ја враќа позицијата на крајот од листата *L*

Pocetok(L) – функција која ја враќа позицијата на почетокот од листата *L*. Ако *L* е празна, враќа *Kraj(L)*.

Init(&L) – функција која ја претвора листата *L* во празна листа, и ја враќа позицијата *Kraj(L)*

Sleden(p, L) – функција која ја враќа позицијата зад *p* во листата *L*. Ако позицијата *p* е последната позиција во *L*, тогаш *Sleden(p, L) = Kraj(L),* а ако *p==Kraj(L)*, *Sleden()* е недефинирана. Ако позицијата p не постои, тогаш *Sleden()* е повторно недефинирана.

Prethoden(p, L) – функција која ја враќа позицијата пред *p* во листата *L*. Ако *p==Pocetok(L), Prethoden()* е недефинирана. Ако позицијата *p* не постои, тогаш *Prethoden()* е повторно недефинирана.

Dodadi(x, p, &L) – функција која ја додава вредноста *x* на позицијата *p* во листата *L*. При извршување на оваа операција елементите кои до тогаш биле на позицијата *p* и сите што биле зад неа се поместуваат за едно место

понатаму, т.е. ако L е од облик $(a_1, a_2, ..., a_n)$, тогаш L станува $(a_1, a_2, ..., a_{p-1}, x, a_p, ..., a_n)$. Ако $p==Kraj(L)$ тогаш L станува $(a_1, a_2, ..., a_n, x)$. Ако во L не постои позиција p, резултатот е недефиниран.

Izbrisi(p, &L) – функција која го отфрла елементот на позицијата p во листата L. Доколку листата L е од облик $(a_1, a_2, ..., a_n)$ тогаш L станува $(a_1, a_2, ..., a_{p-1}, a_{p+1}, ..., a_n)$. Резултатот не е дефиниран ако L ја нема позицијата p или пак ако $p==Kraj(L)$.

Vrednost(p, L) – функција која го враќа елементот на позицијата p во листата L. Резултатот е недефиниран ако $p==Kraj(L)$ или ако L нема елемент на позицијата p.

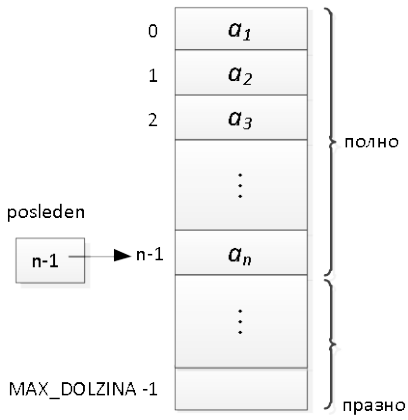
Во продолжение се опишани некои структури на податоци кои можат да послужат за приказ на листа. Постојат два основни пристапи:

- Користење на поле, кога логичкиот редослед на елементите во листата се поклопува со нивниот физички редослед во меморијата
- Користење на покажувач или курсор, кога логичкиот редослед на елементите во листата не се поклопува со физичкиот редослед, па нивната репрезентација мора да биде експлицитна.

И двата пристапи се изводливи со повеќе варијанти, така што во продолжение описот на одделните варијанти не мора да значи дека е единствен.

1.1.2 Имплементација на АПТ ЛИСТА

Имплементација со поле



Слика 1.1. Имплементација на

Елементите на листата се сместени во последователни ќелии од едно поле. За имплементација на овој АПТ потребен искористен е курсор кој покажува каде во полето се наоѓа последниот елемент од листата. Со помош на ваквата имплементација едноставни се операциите кои се во врска со пристапување до елементите било каде во листата, затоа што тие се индексирани, а едноставно

е и додавањето и отфрлањето на елементи на крајот од листата. Но затоа, додавањето и отфрлањето на елементи на почетокот и во средина на листата бара посложени операции за преместување (препишување) на дел од податоците во листата. Дополнителен недостаток е и фактот дека меморискиот простор за полето кое се користи за имплементација на овој АПТ мора да биде алоциран т.е. мора да се резервира меморискиот простор пред да почне да се користи. Ова доведува до ограничување и на самата листа.

Имплементацијата на листата со помош на поле може прецизно да се изрази во програмскиот јазик C++.

```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 30
typedef int IzbranTip;
typedef int pozicija;

typedef struct {
              int posleden;
              IzbranTip element[MAX_DOLZINA];
              } Lista;

pozicija Init(Lista *L) {
       L->posleden = -1;
       return 0;
}

pozicija Prethoden(pozicija p, Lista *L) {
       pozicija q = p - 1;
       if (q < 0) q = 0;
       if (q > L->posleden) q = L->posleden;
       return q;
}
pozicija Sleden(pozicija p, Lista *L) {
       pozicija q = p + 1;
       if (q < 0) q = 0;
       if (q > L->posleden) q = L->posleden;
       return q;
}

pozicija Prv(Lista *L) {
       return 0;
}

pozicija Posleden(Lista *L) {
       return (L->posleden+1);
}
```

```
IzbranTip Vrednost(pozicija p, Lista *L) {
       return L->element[p];
}

void Dodadi(IzbranTip x, pozicija p, Lista *L) {
        pozicija q;
        if (L->posleden >= MAX_DOLZINA-1)
              perror("Listata e polna!!!"); //zapira so rabotata
              else if ((p > L->posleden + 1) || (p < 0))
                            perror("Pozicijata ne postoi!!!");
                      else
                      {
                        for (q = L->posleden; q >= p; q--)
                            // pomesti gi elementite na poziciite
                            // p, p+1, ... edno mesto podolu
                             L->element[q+1] = L->element[q];
                        L->posleden++;
                        L->element[p] = x;
                      }
}

void Izbrisi(pozicija p, Lista *L) {
       if ((p > L->posleden) || (p < 0))
              perror("Pozicijata ne postoi!!!");
       else   {
              L->posleden--;
              for (pozicija q = p; q <= L->posleden; q++)
                     // pomesti gi elementite na poziciite
                     // p+1, p+2, ... edno mesto pogore
                     L->element[q] = L->element[q+1];
              }
}
```

Како што може да се види во кодот дефинирана е константа MAX_DOLZINA која во примерот е поставена на 30, што значи дека листата во примерот е ограничена на најмногу 30 елементи. Исто така дефиниран е тип IzbranTip *за кој е ирелевантно кој податочен тип ќе го репрезентира. Во примерот е поставено дека тој ќе го претставува типот* int*, но ова не значи дека не може да се дефинира како било кој друг податочен тип, вклучително и произволни структури.*

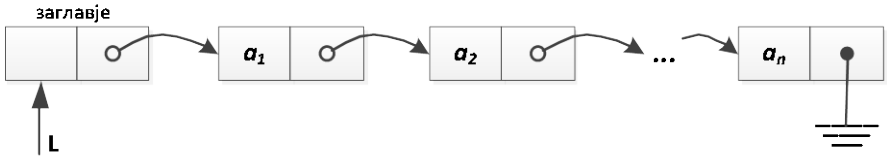
Во однос на времето на извршување, операциите Dodadi() *и* Izbrisi() *имаат време на извршување од* O(n)*, додека останатите операции бараат* O(1).

Имплементацијата на листата со помош на поле се смета дека е погодна доколку при решавање на проблемот е дозволиво да се постави

конечна (не премногу голема) горна граница за должина на листата и доколку нема многу операции од типот додавање и бришење во средина на листата.

Имплементација со покажувачи

Листата се прикажува со помош на низа од ќелии каде секоја од нив содржи еден елемент од листата и покажувач кон иста таква ќелија, која го претставува следниот елемент во листата. Постои и почетна ќелија, која се нарекува и заглавје (header) која го означува почетокот на листата и не содржи никаков елемент. Ваквата структура вообичаено се нарекува поврзана листа (linked list) и графички се претставува како на следната слика.

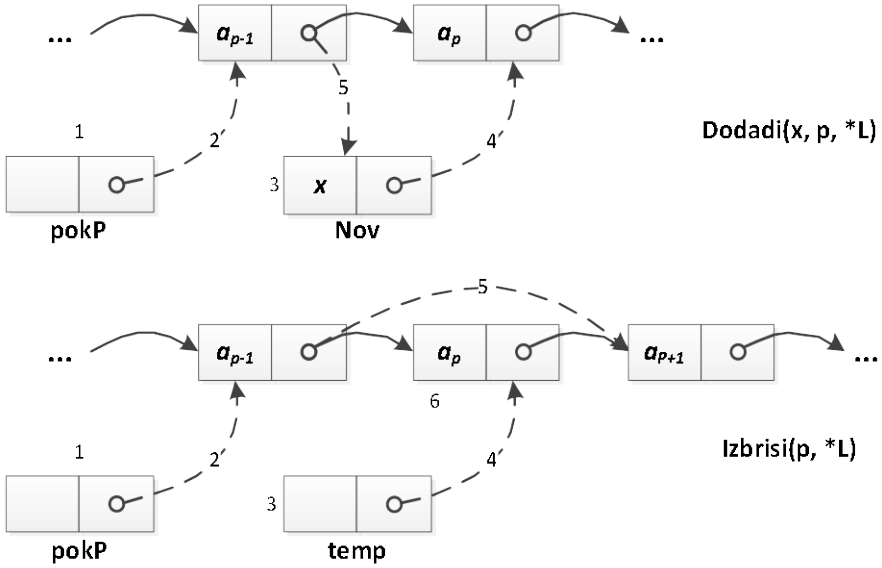


Слика 1.2. Имплементација на општа листа со помош на покажувачи

Кај поврзаните листи, карактеристично е што елементите лесно се додаваат и отфрлаат на било која позиција. Од друга страна, за разлика од имплементацијата со поле, малку покомплицирано е да се прочита *i*-тиот елемент затоа што треба да се прочита прво првиот, па вториот, и така натаму, сè до *i*-тиот елемент, и тоа со помош на дополнителен бројач. Исто така, потешко се одредува крајот на листата, или пак претходниот елемент. Листата се иницијализира со покажувач на заглавјето.

Главна предност на ваквиот начин на имплементација е тоа што теоретски не постои ограничување на должината на листата, туку при секое додавање на нов елемент се алоцира потребниот мемориски простор за новиот јазол и неговата адреса се сместува во покажувачот кој треба да покажува кон тој јазол.

За подобра интерпретација на поврзаните листи, пред самата имплементација за нетривијалните операции *Dodadi* і *Brisi* е даден графички приказ, како и опис на општите алгоритми. Кај дијаграмите, со полни линии е означена состојбата пред влезот во соодветната операција, а со нумерирани испрекинати линии (во секвенца) се означени промените кои се случуваат во тек на алгоритмот.



Слика 1.3. Операции за додавање и бришење на елемент во општа листа на p-та позиција

Како што може да се види и на Слика 1.3 операциите за додавање и бришење на позиција во општа листа воопшто не е тривијален процес. Имено, како што се гледа и во двата случаи потребно е да се генерира покажувач *pokP* (чекор 1 и во двата случаи) кој ќе покажува кон јазолот кој се наоѓа пред *p*-тата позиција (чекор 2 и во двата случаи). Генерирањето на ваков покажувач и негово поврзување кон елементот a_{p-1} е цел процес кој треба да испита дали постои воопшто елемент на *p*-та позиција и доколку постои таков треба да се помине целата листа од почетокот до елементот a_{p-1}.

И во двата случаи, со поставен покажувач на јазолот зад кој ќе се врши операцијата, веќе следните чекори се тривијални. Во првиот случај се генерира нов елемент и му се поставува вредноста *x* (чекор 3), се поставува неговиот покажувач кон јазолот кој е на *p*-тата позиција (чекор 4), а неговата адреса се сместува во покажувачот на елементот a_{p-1} (чекор 5). Во вториот случај се генерира нов јазол кој ќе покажува кон елементот a_p (чекори 3 и 4), а сѐ со цел да се ослободи меморискиот про стор кој го зафаќа јазолот кој се брише од листата (чекор 6).

Според ова во имплементацијата во C++ избран е рекурзивен пристап за решавање на операциите *Dodadi()* и *Izbrisi()*, со што самиот код е во добра мера поедноставен.

```
#include <iostream>
#include <stdio.h>
typedef int IzbranTip;
typedef int pozicija;

typedef struct Lista_Oznaka {
		IzbranTip vrednost;
		Lista_Oznaka *sleden;
	} Lista;

void Init(Lista *L) {
	L->sleden = NULL;
}

Lista * Prv(Lista *L) {
	return L->sleden;
}

Lista * Posleden(Lista *L) {
	if (L->sleden==NULL)
		return NULL;
	else
		if (L->sleden->sleden==NULL)
			return L->sleden;
	else
		return Posleden(L->sleden);
}
Lista * Prethoden(Lista *jazol, Lista *L) {
	if (L->sleden==NULL)
		return NULL;
	else
		if (L->sleden->sleden == jazol)
			return L->sleden;
	else
		return Prethoden(jazol, L->sleden);
}

Lista * Sleden(Lista *jazol) {
	if (jazol->sleden==NULL)
		return NULL;
	else
		return jazol->sleden;
}

void Dodadi(IzbranTip x, pozicija p, Lista *L) {
	if (p==0)
	{
		Lista *Nov = (Lista*)malloc(sizeof(Lista));
		Nov->vrednost = x;
```

```
		Nov->sleden = L->sleden;
		L->sleden = Nov;
	}
	else
		if (L->sleden != NULL)
			Dodadi(x, p-1, Sleden(L));
}

void Izbrisi(pozicija p, Lista *L) {
	if (p==0)
	{
		Lista * temp = L->sleden;
		L->sleden = L->sleden->sleden;
		free(temp);
	}
	else
		if (L->sleden!=NULL)
			Izbrisi(p-1, L->sleden);
}

IzbranTip Vrednost(Lista *jazol) {
	if (jazol==NULL)
		return NULL;
	else
		return jazol->vrednost;
}
```

Како што може да се види и во кодот повеќето потпрограми се едноставни за имплементација. Најкомплицирана е операцијата *Prethoden()* затоа што бара дополнително поминување на елементите од листата од заглавјето до бараната позиција, што е и причина што мора листата да се појави како дополнителен параметар. Од аспект на времето, *Kraj()* и *Prethoden()* имаат *O(n)* време на извршување, додека останатите операции имаат *O(1)* време на извршување.

Имплементацијата на општата листа на ваков начин се смета дека е погодна доколку проблемот бара многу додавања и отфрлања на елементи од листата, доколку должината на листата постојано варира и доколку може да се реши без *Prethoden()*.

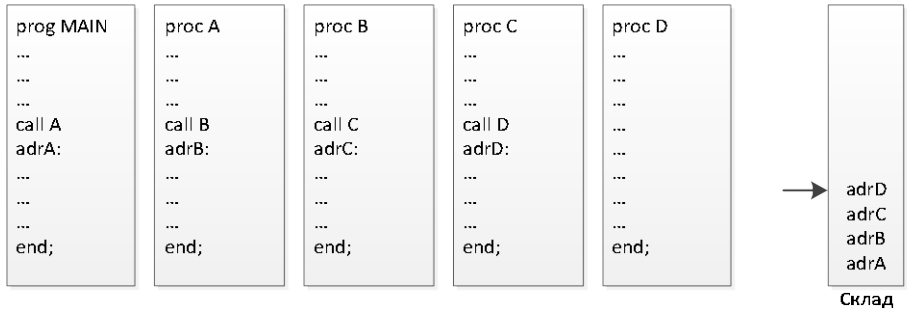
1.2 Склад (Stack)

Склад (Stack, стек, pushdown листа) е специјален вид на листа во која сите додавања и отфрлања на елементи се прават на едниот крај кој се нарекува врв на складот. Складот претставува LIFO (Last In First Out) листа

што значи дека елементот кој последен ќе влезе во структурата, прв ќе биде опслужен.

Во реалниот свет постојат многу примери кои се решаваат и репрезентираат со оваа структура. Таков е примерот со чинии или книги кои се наоѓаат на куп една преку друга. Првиот до кој може да се пристапи е најгорниот елемент, вториот ќе биде достапен откако првиот ќе се отфрли итн.

Складот се користи многу и во компјутерската архитектура, на пример, повикувањето на потпрограми едно по друго. Имено, за време на извршување на секвенца од инструкции кога ќе се наиде на повик на потпрограма, тогаш се зема адресата на следната инструкција и се става на врвот од складот. На овој начин е направен механизам за памтење на адресите на враќање од потпрограмите. Откако потпрограмата ќе заврши со своето извршување ќе го „обработи“ врвниот елемент од складот и ќе оди на адресата од врвниот елемент. Складот е применлив и кога се работи за рекурзивни потпрограми. Овој пример е илустриран на следната слика (Слика 1.4) каде од десната страна се наоѓа содржината складот во било кој момент при извршување на потпрограмата D.



Слика 1.4. Повикувања на потпрограми во програма и содржина на складот во текот на повиците

1.2.1 Апстрактен податочен тип СКЛАД

Складот се користи како специјален случај на листа, но се третира и како посебен апстрактен податочен тип. Складот може да се имплементира и преку имплементацијата на општата листа, но ваквиот пристап можеби би го поедноставил решението, меѓутоа како што беше и дискутирано, времето на извршување би било многу поголемо, со оглед на тоа што потпрограмите од листата се значително комплицирани. Со оглед на тоа што во множеството

на можни имплементации постојат многу поедноставни решенија, во продолжение се прикажани едноставни имплементации .

IzbranTip – било кој тип на податоци.

Sklad – податок од типот Sklad е конечна низа од податоци од типот *IzbranTip*.

Init(&S) – функција која го претвора складот S во празен склад

Prazen(&S) - функција која враќа булова вредност дали складот е празен или не

Push(x, &S) - функција која додава елемент со вредност *x* на врвот од складот S. Оваа операција со термините од АПТ Lista би била еквивалентна со *Dodadi(x, Prv(S), &S)*

Pop(&S) - функција која отфрла елемент од врвот од складот S. Оваа операција со термините од АПТ Lista би била еквивалентна со *Izbrisi(Prv(S), &S)*

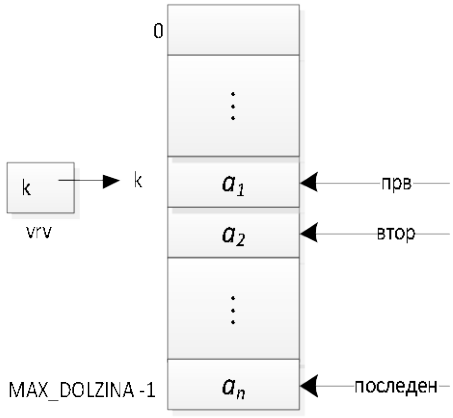
Vrv(&S) - функција која ја враќа вредноста на елементот кој е на врвот од складот и при тоа не го менува складот. Со помош на термините од АПТ Lista, операцијата би се извела со изразот *Vrednost(Prv(S), S)*

1.2.2 Имплементација на АПТ СКЛАД

Имплементација со поле

Оваа имплементација се базира на податочната структура која е претставена и за аптрактниот податочен тип Lista, меѓутоа со една мала измена. Имено, како што може да се види и од алтернативното решение на операциите кај складот со помош на изрази од апстрактниот податочен тип Lista, при додавања и отфрлања на елементи од складот, т.е. на елементи на почетокот од листата, потребни се препишувања на останатите елементи надолу т.е. нагоре во самиот склад.

Измената се состои во тоа што наместо податоците да бидат сместени на почетокот од полето, тие ќе бидат сместени на дното од полето. Со оваа измена се избегнуваат препишувањата на елементите со што се поедноставува алгоритмот за додавање и елиминирање на елементите на врвот од складот, при што индексот на врвниот елемент се намалува како складот расте, т.е. се зголемува, како складот се намалува.



Слика 1.5. Имплементација на склад со помош на поле

И во овој случај, како и во сите случаи кога се користат статички структури, ограничување на должината на структурата мора да постои, па така празен склад би бил оној за кој индексот на врвот ја надминува вредноста MAX_DOLZINA-1, а полн склад би бил оној за кој индексот на врвот би бил еднаков на 0.

Во продолжение се наоѓа имплементација на склад со помош на поле во C++.

```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 30
typedef int IzbranTip;

typedef struct {
              int vrv;
              IzbranTip element[MAX_DOLZINA];
              } Sklad;

void Init(Sklad *S) {
       S->vrv=MAX_DOLZINA;
}

bool Prazen(Sklad *S) {
       return (S->vrv >= MAX_DOLZINA);
}

void Push(IzbranTip x, Sklad *S) {
       if (S->vrv == 0)
              perror("Skladot e poln!!!");
       else
       {
```

```
                S->vrv--;
                S->element[S->vrv]=x;
        }
}

void Pop(Sklad *S) {
        if (Prazen(S))
                perror("Skladot e prazen!!!");
        else
                S->vrv++;
}

IzbranTip Vrv(Sklad *S) {
        IzbranTip rez = NULL;
        if (Prazen(S))
                perror("Skladot e prazen!!!");
        else
                rez = S->element[S->vrv];
        return rez;
}
```

Како што може да се види и во кодот, поголемиот дел од операциите се сведуваат на по една наредба. Имплементацијата на складот со помош на поле е многу погодна со оглед на тоа што времето на извршување на било која операција е константно т.е. *O(1)*. Единствен недостаток е ограничувањето кое е последица на статичка структура.

Имплементација со покажувачи

Оваа имплементација се базира на поврзана листа, но со оглед на тоа што кај складот не е потребен поимот за позиција, не е потребно и заглавјето при имплементацијата на овој АПТ, па доволно е само покажувач на првата ќелија. На овој начин структурата се поедноставува, а до складот се доаѓа со покажувач на почетокот на поврзана листа. Следува код за имплементација на складот со помош на покажувачи.

Слика 1.6. Имплементација на склад со помош на покажувачи

```
#include <iostream>
#include <stdio.h>
typedef int IzbranTip;
```

```
typedef struct Sklad_Oznaka {
              IzbranTip podatok;
              Sklad_Oznaka * sleden;
              } Sklad;

void Init(Sklad* S) {
       S->sleden=NULL;
}

bool Prazen(Sklad *S) {
       return (S->sleden==NULL);
}

void Push(IzbranTip x, Sklad *S) {
       Sklad *Nov = (Sklad *)malloc(sizeof(Sklad));
       Nov->podatok = x;
       Nov->sleden = S->sleden;
       S->sleden=Nov;
}
void Pop(Sklad *S) {
       if (Prazen(S))
              perror("Skladot e prazen!!!");
       else
       {      Sklad * temp = S->sleden;
              S->sleden = temp->sleden;
              free(temp);
       }
}

IzbranTip Vrv(Sklad *S) {
       IzbranTip rez = NULL;
       if (Prazen(S)) perror("Skladot e prazen!!!");
              else rez = S->sleden->podatok;
       return rez;
}
```

1.3 Ред (Queue)

Ред (queue) е специјален вид на листа во која елементите се додаваат на едниот крај на листата (крај), а се отфрлаат на другиот крај на листата (почеток). Редот е структура од типот FIFO (First In First Out) и е определена со почетокот кој го означува првиот елемент во редот и крајот кој го означува последниот елемент од листата.

Слика 1.7. Ред

Примери за ред во реалниот свет има насекаде. Редицата на луѓе кои чекаат на каса во маркет е типичен ред. Ако касата е празна, првиот што ќе дојде ќе биде опслужен. Ако во моментот се опслужува некој, секој нареден застанува во редот и чека сите луѓе пред него да бидат опслужени. Кај компјутерската техника пример за користење на ред е кај делениот печатач. Кога има локална мрежа со еден печатач, тогаш за него се креира редица во која се редат документите од целата мрежа еден по друг, така што првиот кој дошол во редицата ќе биде испечатен прв. Доколку печатачот во моментот печати, а некој корисник сака да испечати друг документ, тогаш документот доаѓа на крајот од креираниот ред на принтер-серверот.

1.3.1 Апстрактен податочен тип РЕД

Редот е уште еден специјален случај на листа, кој се обработува како посебен апстрактен податочен тип. Слично како кај складот, и редот може да се имплементира и преку имплементацијата на општата листа, но со ваквиот пристап времето на извршување би било многу поголемо, поради комплексноста на потпрограмите од имплементацијата на листата. Во продолжение се прикажани едноставни имплементации.

IzbranTip – било кој тип на податоци.

Red – податок од типот Red е конечна низа од податоци од типот *IzbranTip*.

Init(&R) – функција која го претвора редот R во празен ред

Prazen(R) - функција која враќа булова вредност дали редот е празен

Enqueue(x, &R) - функција која додава елемент со вредност *x* на крајот од редот R. Оваа операција со термините од АПТ Lista би била еквивалентна со *Dodadi(x, Kraj(R), &R)*

Dequeue(&R) - функција која отфрла елемент од почетокот на редот R. Оваа операција со термините од АПТ Lista би била еквивалентна со *Izbrisi(Prv(R), &R)*

Pocetok(R) - функција која ја враќа вредноста на елементот кој е на почетокот од редот R и при тоа не го менува редот. Со помош на термините од АПТ Lista, операцијата би се извела со изразот *Vrednost(Prv(R), R)*

И во овој случај имплементациите на редот може да се добијат од имплементацијата на листа, меѓутоа постојат многу поедноставни имплементации, така што имплементацијата нема да биде реализирана преку АПТ Lista.

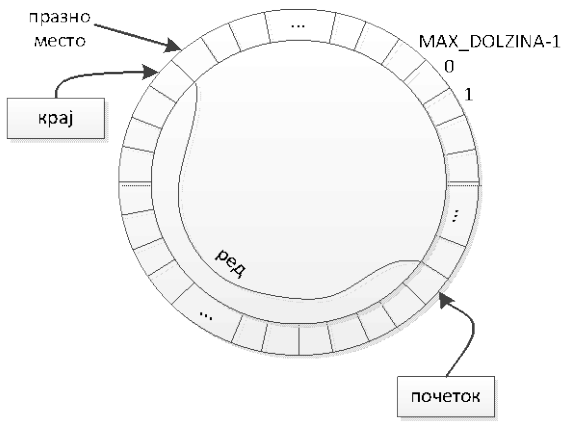
1.3.2 Имплементација на АПТ РЕД

Имплементација со поле

Имплементацијата на овој апстрактен податочен тип може да биде буквално иста со имплементацијата на листата со помош на поле. На овој начин, операцијата Enqueue() ќе се извршува во константно време бидејќи не бара поместување на веќе сместените елементи во листата, туку само додава елемент на крајот од листата. Меѓутоа, Dequeue() не е многу ефикасна. Имено, отфрлањето на елементите се случува на почетокот на редот, па тогаш ќе мора целиот ред да се помести за едно место погоре, како редот повторно би започнал на позицијата 0.

Решение за овој проблем е воведување на уште еден курсор кој ќе го означува почетокот на редот. На овој начин нема веќе да има потреба од препишување на елементите, затоа што редот нема да започнува на позицијата 0, туку на позицијата определена со курсорот за почеток. Решението е добро, но се појавува нов недостаток, со додавање и бришење на елементи во редот, редот се поместува надолу кон долниот крај во полето, така што во еден момент ќе се стигне до долниот крај, иако во почетниот дел би имало место.

Проблемот се решава со воведување на поимот циркуларно поле. Идејата е да кога треба да се додаде нов елемент во редот, во ситуација кога крајот на редот се поклопува со крајот на полето, новиот елемент да се смести во првата позиција од полето т.е. полнењето на редот да почне од почетокот на полето во кое е сместен редот. На Слика 1.8 е претставено полето, но во вид на циркуларно поле.



Слика 1.8. Имплементација на ред со помош на циркуларно поле

Имплементацијата се изведува со помош на курсори за почеток и крај на редот, но се поставува прашањето што би значело редот да е празен, а што би значело редот да е полн. Се дефинира дека, доколку почетокот и крајот се еднакви, тогаш редот има еден елемент т.е. елементот со индекс почеток т.е. крај. Доколку почетокот се наоѓа на првата позиција зад крајот, тогаш ова е празен ред, но тука се појавува една дилема, дали ова е празен или полн ред.

Поради избегнување на двосмисленоста се дефинира дека секогаш елементот зад крајот е празен, освен кога редот е празен, што значи дека доколку почетокот се наоѓа на следниот елемент од крајот, тогаш редот е празен, а доколку почетокот се наоѓа на две места (преку едно празно место) зад крајот, тогаш редот е полн.

Кај ова решение, поради воведување на празно место, мора да се внимава кога се декларира максималната должина на редот затоа што се губи еден елемент на сметка на празното место, па ако се постави дека максималната должина е 30, тогаш треба да се знае дека во редот може да се сместат најмногу 29 елементи.

Имплементацијата е дадена во C++. Треба само да се напомене дека инкрементацијата на курсорите се извршува по модул од MAX_DOLZINA преку функцијата INC, што е клучно за изведување на циркуларната листа со статичко поле.

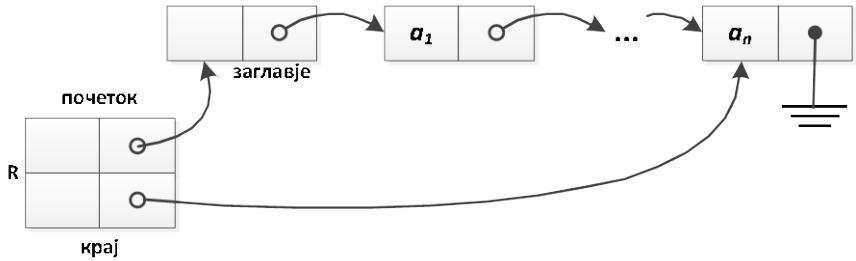
```
#include <iostream>
#include <stdio.h>

#define MAX_DOLZINA 30
```

```
typedef int IzbranTip;

typedef struct {
              int pocetok, kraj;
              IzbranTip element[MAX_DOLZINA];
              } Red;

int INC(int k) {
       return ((k+1) % MAX_DOLZINA);
}

void Init(Red *R) {
       R->pocetok=0;
       R->kraj = MAX_DOLZINA-1;
}

bool Prazen(Red *R) {
       return (INC(R->kraj)==R->pocetok);
}


void Enqueue(IzbranTip x, Red *R) {
       if (INC(INC(R->kraj))==R->pocetok)
              perror("Redot e poln!!!");
       else
       {
              R->kraj = INC(R->kraj);
              R->element[R->kraj] = x;
       }
}

void Dequeue(Red *R) {
       if (Prazen(R))
              perror("Redot e prazen!!!");
       else
              R->pocetok = INC(R->pocetok);
}

IzbranTip Pocetok(Red *R) {
       IzbranTip rez = NULL;
       if (Prazen(R))
              perror("Redot e prazen!!!");
       else
              rez = R->element[R->pocetok];
       return rez;
}
```

Имплементација со покажувачи

Имплементацијата на ред со помош на покажувачи е со сличен пристап како имплементацијата со помош на поле. Имаме два покажувачи, кон почетокот и кон крајот на редот, а поради поедноставна репрезентација на редот, тој секогаш започнува со заглавје т.е. елемент кој не фигурира како елемент од редот.

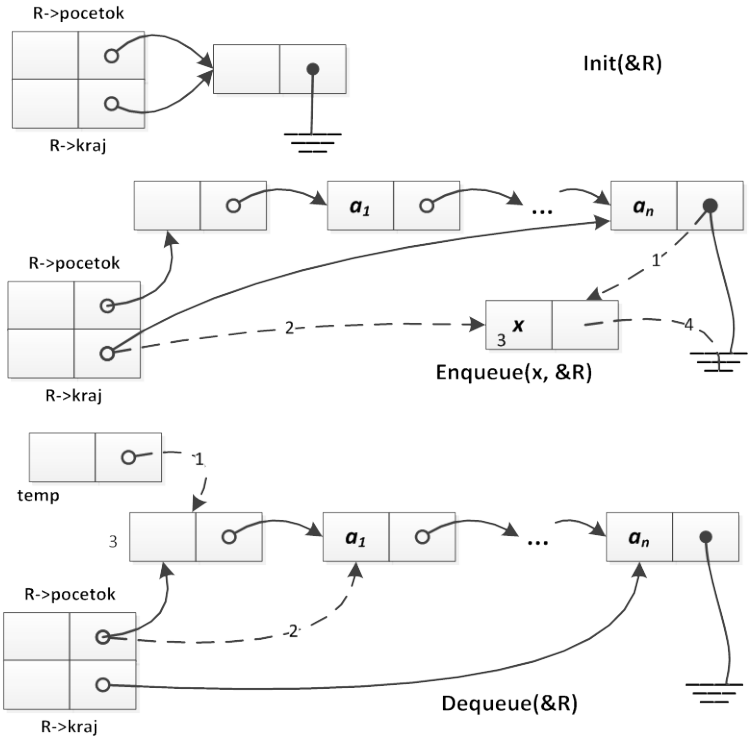
Покажувачите се сместени во заедничка структура која го репрезентира редот, што е илустрирано на Слика 1.9.



Слика 1.9. Имплементација на ред со помош на покажувачи

Во продолжение е дадена имплементацијата на редот со помош на покажувачи во C++. Поради подобра илустрација алгоритмите за операциите Init(), Enqueue() и Dequeue() се прикажани и графички на Слика 1.10.

```
#include <iostream>
#include <stdio.h>

typedef int IzbranTip;

typedef struct Jazol_Oznaka {
              Jazol_Oznaka *sleden;
              IzbranTip podatok;
              } Jazol;

typedef struct Red_Oznaka {
              Jazol *pocetok, *kraj;
              } Red;

bool Prazen(Red *R) {
       return (R->kraj==R->pocetok);
}
```



Слика 1.10. Дијаграми за операциите Init, Enqueue и Dequeue

```
void Init(Red *R) {
	R->pocetok = (Jazol*)malloc(sizeof(Jazol));
	R->pocetok->sleden = NULL;
	R->kraj = R->pocetok;
}

void Enqueue(IzbranTip x, Red *R) {
	R->kraj->sleden = (Jazol*)malloc(sizeof(Jazol));
	R->kraj = R->kraj->sleden;
	R->kraj->podatok = x;
	R->kraj->sleden = NULL;
}

void Dequeue(Red *R) {
	Jazol * temp;
	if (Prazen(R)) perror("Redot e prazen!!!");
	else
	{
		temp = R->pocetok;
		R->pocetok = R->pocetok->sleden;
		free(temp);
	}
}
```

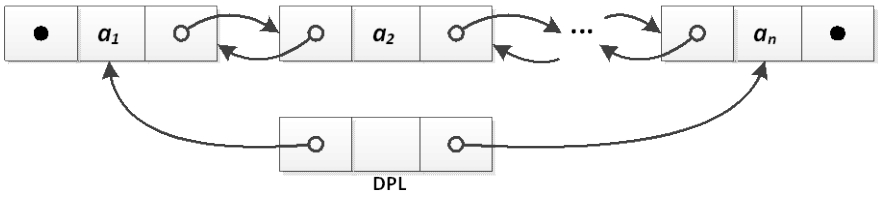
```
IzbranTip Pocetok(Red *R) {
	IzbranTip rez = NULL;
	if (Prazen(R)) perror("Redot e prazen!!!");
	else
		rez = R->pocetok->sleden->podatok;
	return rez;
}
```

1.4 Други видови листи

Постојат повеќе видови на варијации на листите и комбинации од нив. Во ова поглавје ќе бидат претставени само некои покарактеристични видови за кои ќе бидат извршени имплементации во C++, но само со помош на покажувачи.

1.4.1 Двојно-поврзана листа (Doubly Linked List)

Кај стандардната поврзана листа, елементите се поминуваат само во една насока. Но можеби за одредени решенија е потребно да се овозможи поминување и во другата насока. Решението се состои во воведување на уште еден покажувач, во секој од јазлите од стандардната поврзана листа, кој служи за овозможување на поминување (traversal) низ листата во другата насока. Оваа структура се нарекува двојно-поврзана листа (Doubly Linked List) и истата е претставена на Слика 1.11. Од сликата се гледа дека секој јазол е составен од информациски дел и од покажувач кон лево и покажувач кон десно.



Слика 1.11. Имплементација на двојно-поврзана листа со покажувачи

Во оваа имплементација, листата се определува со јазол чијшто лев покажувач покажува на почетокот на листата, а десниот покажува на крајот од листата.

Апстрактен податочен тип ДВОЈНО-ПОВРЗАНА ЛИСТА

IzbranTip – било кој тип на податоци.

DPLista – податок од типот *DPLista* е конечна низа од податоци од типот *IzbranTip*.

Init(&DPL) – функција која ја претвора двојно-поврзаната листа *DPL* во празна

Prazna(DPL) - функција која враќа булова вредност дали листата е празна или не

VmetniDesno(x, &Jazol, &DPL)* и *VmetniLevo(x, &Jazol, &DPL) - функција која додава елемент со вредност *x* пред(зад) јазолот *Jazol* во двојно-поврзаната листа *DPL*.

***DodadiLevo(x, &DPL)* и *DodadiDesno(x, &DPL)* -** функција која додава елемент со вредност *x* на левиот (десниот) крај во двојно-поврзаната листа *DPL*. Во случај *DPL==NULL*, двете функции се поистоветуваат, т.е. едната може да се изведе преку другата. Во случај кога *DPL* е непразна, и двете функции се изведуваат преку функциите за вметнување и тоа

DodadiLevo(x, &DPL)=VmetniLevo(x, DPL->Lpok, DPL)

DodadiDesno(x, &DPL)=VmetniDesno(x, DPL->Dpok, DPL).

Во последниот случај останува уште да се врзат соодветните врски со почетокот (крајот) на листата т.е. *DPL->Lpok=DPL->Lpok->Lpok (DPL->Dpok=DPL->Dpok->Dpok).*

Izbrisi(&Jazol, &DPL) - функција која го брише јазолот *Jazol* од двојно-поврзаната листа *DPL*. Карактеристично за оваа операција е тоа што пред да се премине на бришење треба да се испита позицијата на јазолот во листата т.е. дали е на некој крај, или не.

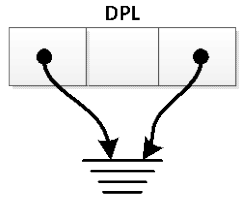
Vrednost(Jazol) - функција која ја враќа вредноста на јазолот *Jazol* од двојно-поврзаната листа DPL и при тоа не ја менува листата. Имплементацијата на оваа операција е тривијална.

Имплементација на АПТ ДВОЈНО-ПОВРЗАНА ЛИСТА

Имплементацијата на двојно-поврзаната листа во C++, како и графичкиот приказ на некои дијаграми за нетривијалните операции е дадена во продолжение. Операциите кои се однесуваат на ориентацијата се

прикажани само во една варијанта - левата, затоа што операциите кои се однесуваат на десната страна се исти како и на левата.

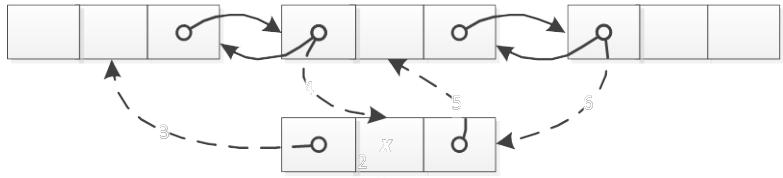
```
#include <iostream>
#include <stdio.h>
typedef int IzbranTip;

typedef struct DPLista_Oznaka {
              DPLista_Oznaka *Lpok;
              DPLista_Oznaka *Dpok;
              IzbranTip podatok;
              } DPLista;

void Init(DPLista *DPL) {
       DPL->Dpok = NULL;
       DPL->Lpok = NULL;
}

bool Prazna(DPLista *DPL) {
       return ((DPL->Dpok==NULL)&&(DPL->Lpok==NULL));
}
```



Слика 1.12. Операција Init() кај АПТ двојно-поврзана листа

```
void VmetniDesno(IzbranTip x, DPLista *Jazol, DPLista *DPL) {
       if (Jazol==NULL) perror("Takov jazol ne postoi!!!");
       else
       {
              DPLista *Nov = (DPLista*)malloc(sizeof(DPLista));
              Nov->podatok = x;
              Nov->Dpok = Jazol->Dpok;
              Jazol->Dpok = Nov;
              Nov->Lpok = Jazol;
              if (Nov->Dpok!=NULL)
                     Nov->Dpok->Lpok = Nov;
              else DPL->Dpok = Nov;
       };
}

void VmetniLevo(IzbranTip x, DPLista *Jazol, DPLista *DPL) {
       if (Jazol==NULL) perror("Takov jazol ne postoi!!!");
       else
```

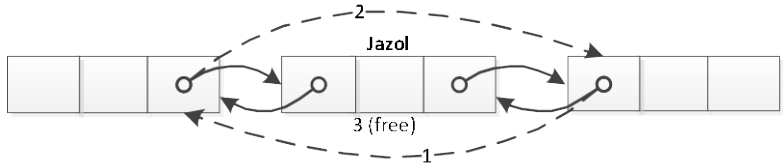
```
	{
		DPLista *Nov = (DPLista*)malloc(sizeof(DPLista));
		Nov->podatok = x;
		Nov->Lpok = Jazol->Lpok;
		Jazol->Lpok = Nov;
		Nov->Dpok = Jazol;
		if (Nov->Lpok!=NULL)
			Nov->Lpok->Dpok = Nov;
		else DPL->Lpok = Nov;
	};
}
```



Слика 1.13. Операција VmetniLevo() кај АПТ двојно-поврзана листа

```
void DodadiLevo(IzbranTip x, DPLista *DPL) {
	if (Prazna(DPL))
	{
		DPL->Dpok = (DPLista*)malloc(sizeof(DPLista));
		DPL->Dpok->podatok = x;
		DPL->Dpok->Lpok=NULL;
		DPL->Dpok->Dpok=NULL;
		DPL->Lpok = DPL->Dpok;
	}
	else
		VmetniLevo(x, DPL->Lpok, DPL);
}

void DodadiDesno(IzbranTip x, DPLista *DPL) {
	if (Prazna(DPL))
		DodadiLevo(x, DPL);
	else
		VmetniDesno(x, DPL->Dpok, DPL);
}

void Izbrisi(DPLista *Jazol, DPLista *DPL) {
	if (Jazol==NULL)
		perror("Takov jazol ne postoi!!!");
	else
	{
		if (Jazol->Dpok==NULL)
		{
			DPL->Dpok=Jazol->Lpok;
```

```
                if (Jazol->Lpok==NULL)
                        Init(DPL);
                else
                        Jazol->Lpok->Dpok=Jazol->Dpok;
        }
        else
        {
                if (Jazol->Lpok==NULL)
                {
                        DPL->Lpok = Jazol->Dpok;
                        Jazol->Dpok->Lpok = Jazol->Lpok;
                }
                else
                {
                        Jazol->Dpok->Lpok=Jazol->Lpok;
                        Jazol->Lpok->Dpok=Jazol->Dpok;
                }
        }
        free(Jazol);
    }
}
```



Слика 1.14. Операција Izbrisi() кај АПТ двојно-поврзана листа

```
IzbranTip Vrednost(DPLista *Jazol) {
        IzbranTip rez = NULL;
        if (Jazol==NULL) perror("Takov jazol ne postoi!!!");
        else
                rez = Jazol->podatok;
        return rez;
}
```

1.4.2 Циркуларна листа (Circular Linked List)

Циркуларна листа е поврзана листа која може да се имплементира со покажувач на крајот од листата, а покажувачот од последниот елемент да покажува на првиот елемент на листата. Структурата од оваа имплементација на циркуларна листа е прикажана на Слика 1.15. Ваквата листа е мало проширување на редот, така што главната предност на циркуларната листа е што сите случувања во врска со елементите се случуваат на крајот на листата, ако може така да се каже (дека е крај).

Циркуларната листа е определена со покажувач на крајот така што листата „кружи“ околу покажувачот.

Слика 1.15. Имплементација на циркуларна листа

Апстрактен податочен тип ЦИРКУЛАРНА ЛИСТА

IzbranTip – било кој тип на податоци.

CLista – податок од типот *CLista* е конечна низа од податоци од типот *IzbranTip*.

Init(&CL) – функција која ја претвора циркуларната листа *CL* во празна

Prazna(CL) - функција која враќа булова вредност дали листата е празна или не

Pomesti(CL) - функција која ја поместува циркуларната листа *CL* од десно кон лево т.е. функција која го префрла првиот елемент на крајот од циркуларната листа *CL*

Dodadi(x, &CL) - функција која додава елемент со вредност *x* на почетокот (зад крајот) од циркуларната листа *CL*.

Izbrisi(&CL) - функција која го брише првиот јазол од циркуларната листа *CL*, а покажувачот на листата останува непроменет

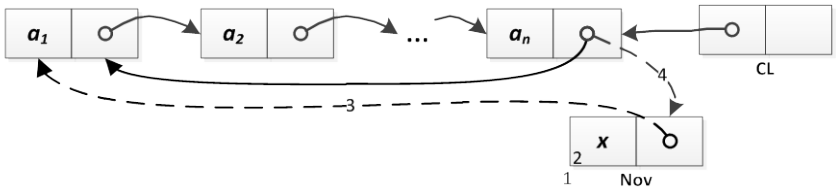
Vrednost(CL) - функција која ја враќа вредноста на последниот јазол од циркуларната листа *CL* и при тоа не ја менува листата. Имплементацијата на оваа операција е тривијална.

Имплементација на АПТ ЦИРКУЛАРНА ЛИСТА

Имплементацијата на овој апстрактен податочен тип е во продолжение.

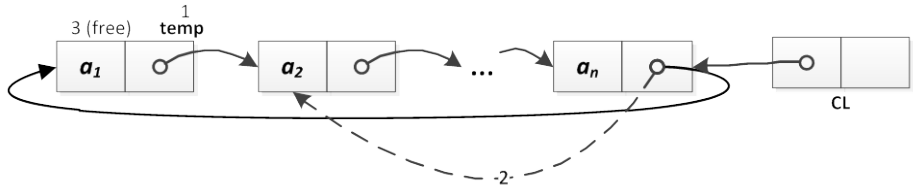
```
#include <iostream>
#include <stdio.h>
typedef int IzbranTip;

typedef struct CLista_Oznaka {
              CLista_Oznaka *sleden;
```

```
                IzbranTip podatok;
                } CLista;

void Init(CLista *CL) {
        CL->sleden = NULL;
}

bool Prazna(CLista *CL) {
        return (CL->sleden==NULL);
}
```



Слика 1.16. Операција Dodadi() кај АПТ циркуларна листа

```
void Dodadi(IzbranTip x, CLista *CL) {
        CLista *Nov = (CLista*)malloc(sizeof(CLista));
        Nov->podatok = x;
        if (Prazna(CL))
        {
                CL->sleden = Nov;
                Nov->sleden = Nov;
        }
        else
        {
                Nov->sleden = CL->sleden->sleden;
                CL->sleden->sleden = Nov;
        }
}
```



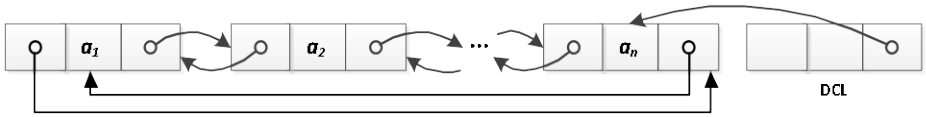
Слика 1.17. Операција Izbrisi() кај АПТ циркуларна листа

```
void Izbrisi(CLista *CL) {
        if (Prazna(CL))
                perror("Listata e prazna!!!");
        else
                if (CL->sleden->sleden==CL->sleden)
        {
                free(CL->sleden);
```

```
		Init(CL);
	}
	else
	{
		CLista *temp = CL->sleden->sleden;
		CL->sleden->sleden = CL->sleden->sleden->sleden;
		free(temp);
	};
}

void Pomesti(CLista *CL) {
	if (!Prazna(CL))
		CL->sleden = CL->sleden->sleden;
}

IzbranTip Vrednost(CLista *CL) {
	IzbranTip rez = NULL;
	if (Prazna(CL)) perror("Listata e prazna!!!");
	else rez = CL->sleden->podatok;
	return rez;
}
```

1.4.3 Двојна циркуларна поврзана листа (Doubly Linked Circular List)

Циркуларната листа е едноставен и моќен апстрактен податочен тип, но постојат бројни проблеми за кои не е доволно кружењето на листата во една насока, туку бараат кружење на јазлите во двете насоки. Ова се реализира со воведување на дополнителен покажувач во јазлите кој е наменет да го обезбеди кружењетово другата насока. На овој начин се добива структура која се нарекува двојна циркуларна поврзана листа. На Слика 1.18 е прикажана имплементација на двојна циркуларна листа. Имено, над имплементацијата на циркуларната листа се применуваат минорни измени во делот со покажувачите на лево, и се добива имплементацијата на двојната циркуларна листа.



Слика 1.18. Имплементација на АПТ двојна циркуларна листа

Апстрактен податочен тип ДВОЈНА ЦИРКУЛАРНА ЛИСТА

IzbranTip – било кој тип на податоци.

DCLista – податок од типот *DCLista* е конечна низа од податоци од типот *IzbranTip*.

Init(&DCL) – функција која ја претвора двојната циркуларна листа *DCL* во празна

Prazna(DCL) - функција која враќа булова вредност дали листата е празна или не

PomestiDesno(DCL) и ***PomestiLevo(DCL)*** - функција која ја поместува циркуларната листа *DCL* од десно кон лево (лево кон десно) т.е. функција која го префрла првиот (последниот) елемент на крајот (почетокот) од двојната циркуларна листа *DCL*

DodadiDesno(x, &DCL) и ***DodadiLevo(x, &DCL)*** - функција која додава елемент со вредност *x* зад крајниот (на почетокот на листата) т.е. пред крајниот јазол од двојната циркуларна листа *DCL*. Функциите на едноставно се изведуваат една од друга, на пример

DodadiLevo(x,DCL)=PomestiDesno(PomestiDesno(DodadiDesno(x, PomestiLevo(DCL))))

Vrednost(DCL) - функција која ја враќа вредноста на последниот јазол од двојната циркуларна листа *DCL* и при тоа не ја менува листата.

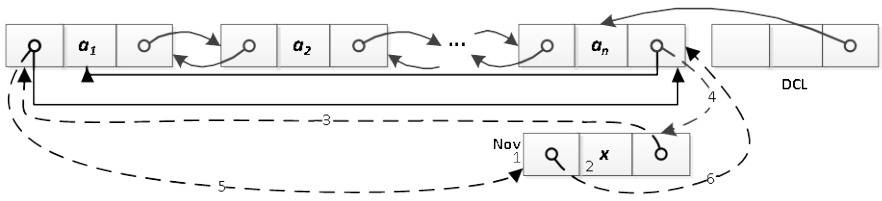
Izbrisi(&DCL) - функција која го брише последниот јазол од двојната циркуларна листа *DCL*, покажувачот на листата се поставува на првиот елемент т.е. почетниот елемент на циркуларната листа се префрлува на крај од листата.

Имплементација на АПТ ДВОЈНА ЦИРКУЛАРНА ЛИСТА

Ова е имплементацијата во програмскиот јазик C++ на ваквата двојно поврзана циркуларна листа:

```
#include <iostream>
#include <stdio.h>

typedef int IzbranTip;

typedef struct DCLista_Oznaka {
              DCLista_Oznaka *Lpok;
              DCLista_Oznaka *Dpok;
              IzbranTip podatok;
              } DCLista;

void Init(DCLista *DCL) {
```

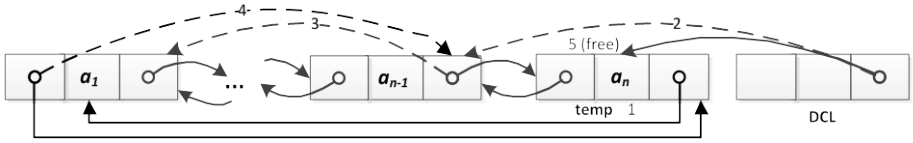
```
	DCL->Dpok = NULL;
}

bool Prazna(DCLista *DCL) {
	return (DCL->Dpok==NULL);
}

void PomestiDesno(DCLista *DCL) {
	if (!Prazna(DCL))
		DCL->Dpok = DCL->Dpok->Dpok;
}

void PomestiLevo(DCLista *DCL) {
	if (!Prazna(DCL))
		DCL->Dpok = DCL->Dpok->Lpok;
}


void DodadiDesno(IzbranTip x, DCLista *DCL) {
	DCLista *Nov = (DCLista*)malloc(sizeof(DCLista));
	Nov->podatok = x;
	if (Prazna(DCL))
	{
		DCL->Dpok = Nov;
		Nov->Dpok = Nov;
		Nov->Lpok = Nov;
	}
	else
	{
		Nov->Dpok = DCL->Dpok->Dpok;
		DCL->Dpok->Dpok = Nov;
		Nov->Dpok->Lpok = Nov;
		Nov->Lpok = DCL->Dpok;
	}
}
```



Слика 1.19. Операција DodadiDesno() кај АПТ двојна циркуларна листа

```
void DodadiLevo(IzbranTip x, DCLista *DCL) {
	PomestiLevo(DCL);
	DodadiDesno(x, DCL);
	PomestiDesno(DCL);
	PomestiDesno(DCL);
```

```
}

void Izbrisi(DCLista *DCL) {
	if (Prazna(DCL)) perror("Listata e prazna!!!");
	else if (DCL->Dpok->Dpok==DCL->Dpok)
	{
		free(DCL->Dpok);
		Init(DCL);
	}
	else
	{
		DCLista *temp = DCL->Dpok;
		PomestiLevo(DCL);
		DCL->Dpok->Dpok = temp->Dpok;
		temp->Dpok->Lpok = DCL->Dpok;
		free(temp);
	};
}
```



Слика 1.20. Операција Izbrisi() кај АПТ двојна циркуларна листа

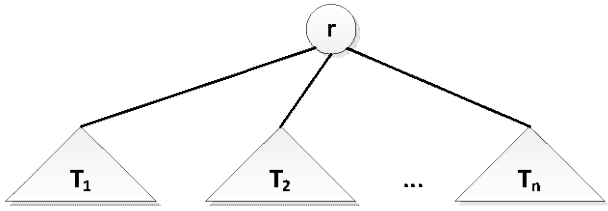
```
IzbranTip Vrednost(DCLista *DCL) {
	IzbranTip rez = NULL;
	if (Prazna(DCL))
		perror("Listata e prazna!!!");
	else
		rez = DCL->Dpok->podatok;
	return rez;
}
```

2. Стебла

2.1 Општо стебло (Tree)

Стеблата се структури кои се базираат на хиерархиско уредување, за разлика од листите кои се базираат на линеарно уредување на податоците. Хиерархиското уредување значи дека меѓу податоците постојат надредени и подредени т.е. родители и деца.

Општо стебло претставува непразно конечно множество на податоци од ист тип кои се нарекуваат **јазли**, при што постои еден јазол кој се нарекува **корен** на стеблото, а другите јазли градат конечна низа од 0 или повеќе дисјунктни стебла. Дисјунктни стебла се стебла кај кои не постои јазол, било каде во нивната хиерархија, кој припаѓа на повеќе од едно стебло. На Слика 2.1 е прикажано општо стебло како што е дефинирано во претходните ставови т.е. рекурзивно.



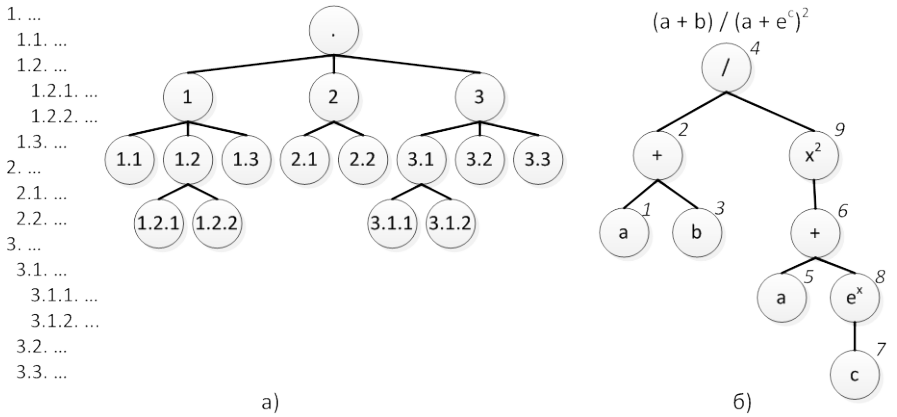
Слика 2.1 Стебло T

Како што може да се види и на сликата помалите стебла означени со T_1, T_2 … T_n се нарекуваат **подстебла** на коренот *r*. Корените r_1, r_2, … r_n на подстеблата T_1, T_2 … T_n се нарекуваат **деца** на јазолот *r*, а *r* се нарекува нивни **родител**. Може да се забележи дека коренот е единствениот јазол во целата структура кој нема родител, а сите други јазли имаат точно по еден. Важно е линеарното подредување на стеблата, т.е. се знае кое е прво, кое второ итн. Максималниот број на деца кој го има некој јазол од едно стебло го одредува **степенот** на стеблото.

Во реалниот свет постојат многу примери кои е наједноставно да се решаваат со помош на ваква структура. Таков пример е содржината од една книга (Слика 2.2-а) или пак алгоритми кои треба да парсираат градба на аритметички изрази (Слика 2.2-б).

Кај стеблото од примерот на Слика 2.2-б може да се забележат и некои дополнителни податоци покрај секој од јазлите. Ова се **ознаки** кои може да

ги има едно стебло, а во таков случај се работи за **означено стебло**. Треба да се напомене дека ознаките може да носат дополнителни информации за секој јазол.



Слика 2.2. Примери за стебла

Низа од јазли i_0, i_1, ..., i_n каде i_k е родител на i_{k+1} за $0 \le k < n$ се нарекува **пат** од i_0 до i_n, а **должината** на тој пат е n. За секој јазол кој е различен од коренот на стеблото, постои единствен пат од коренот на стеблото до него.

Ако постои пат од некој јазол i_k од пониско ниво до некој јазол i_m со повисоко ниво тогаш јазолот i_k е **предок** на i_m, а i_m е **потомок** на i_k. Ова доведува до заклучокот дека коренот е предок на сите јазли во стеблото, а тие се негови потомци.

Ниво на еден јазол е должината на единствениот пат од коренот до него. По дефиниција, коренот е единствениот јазол од ниво 0, децата на коренот се од ниво 1, нивните деца од ниво 2 итн. Максималната вредност на нивото која може да се достигне со барем еден јазол во стеблото се нарекува **висина** (некаде може да се сретне и како **длабочина**) на стеблото. Јазлите кои немаат деца се нарекуваат **листови**, а јазлите кои не се листови се **внатрешни јазли**. Децата на еден ист јазол се нарекуваат **браќа**, а кај нив постои и уредувањето во однос на подредувањето т.е. термините **претходен** брат или **следен** брат.

2.1.1 Апстрактен податочен тип СТЕБЛО

За да може стеблото да се претвори во апстрактен податочен тип, како и кај другите поими, потребно е да се дефинираат и операции кои се извршуваат кај стеблата. И кај стеблата, како и претходно, имплементацијата

може да се изведе на различни начини, така што овде е дадена една од можните варијанти.

IzbranTip - било кој тип на податоци

oznaka - било кој тип наменет за означување на јазлите

jazol - јазол на едно стебло составен од дел со информации од типот *IzbranTip* и покажувачки дел, т.е. 0 или повеќе покажувачи кон други јазли. Секој јазол може да биде означен со ознака од типот *oznaka*.

steblo - податок од овој тип е стебло чии јазли се податоци од типот *jazol* кои се различни меѓу себе и не се *NULL*.

Init(o, d, &T) - функција која го претвора стеблото *T* во стебло кое се состои само од коренски јазол на кој му се доделува вредност *d* и ознаката *o* која може да биде и *NULL.*

DodadiDete(o, d, j, &T) - функција која додава јазол со вредност *d*, како прво дете на јазолот *j* во стеблото *T* и го означува со *o*

DodadiBrat(o, d, j, &T) - функција која додава нов јазол со вредност *d* како следен брат на јазолот *j* во стеблото *T* и го означува новиот јазол со *o*. Ако јазолот веќе има следен брат, новиот јазол се додава меѓу двата јазли. Не е дефинирана ако *j* е коренот на стеблото.

Brisi(j, &T) - функција која го брише јазолот *j* од стеблото *T*. Не е дефинирана ако *j* е коренот на стеблото, ако *j* не припаѓа на *T* или ако *j* има деца.

Koren(T) - функција која го враќа коренот на стеблото *T*

EKoren(j, T) - функција која испитува дали јазолот *j* е корен на стеблото *T*

PrvoDete(j, T) - функција која го враќа првото дете на јазолот *j* од стеблото *T*.

SledenBrat(j, T) - функција која го враќа следниот брат на јазолот *j* во стеблото *T*. Ако *j* е последно дете или ако *j* не припаѓа на *T*, тогаш враќа *NULL*.

Roditel(j, T) - функција која го враќа јазолот кој е родител на јазолот *j* во стеблото *T.*

Vrednost(j, T) - функција која ја враќа вредноста на јазолот *j* од стеблото *T*

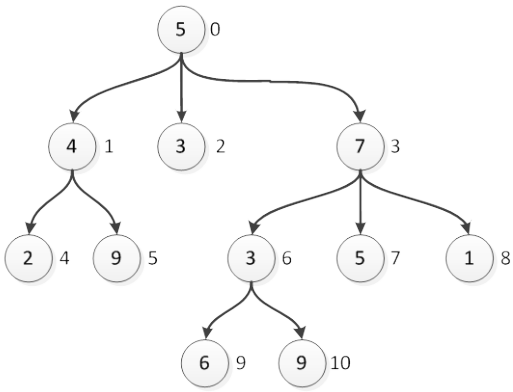
Oznaka(j, T) - функција која ја враќа ознаката на јазолот *j* од стеблото *T*

SmeniOznaka(o, j, &T) - функција која ја менува ознаката на јазолот *j* од стеблото *T* во нова - *o*. Ако *j* не припаѓа на *T* тогаш функцијата враќа *NULL*.

2.1.2 Имплементација на АПТ СТЕБЛО

Имплементација преку релација јазол-родител

Оваа имплементација се базира на тоа да на секој јазол експлицитно му се поврзе неговиот родител. Во множеството на различни варијанти за реализација на овој модел, без голема загуба на генералноста, избран е модел со *n* јазли, каде ознаките на јазлите се цели броеви *0, 1, 2, ..., n-1*.



о нака	подато	родите
0	5	-1
1	4	0
2	3	0
3	7	0
4	2	1
5	9	1
6	3	3
7	5	3
8	1	3
9	6	6
10	9	6

Слика 2.3. Имплементација на АПТ стебло со поле преку релацијата јазол-родител

Стеблото се прикажува со поле од тројки, каде *i*-тата ќелија од полето опишува за кој јазол станува збор според ознаката, која содржина ја содржи и кој е родителот на тој јазол. По дефиниција се зема дека во нултата ќелија е опишан коренот на стеблото чијшто родител е означен со некоја ознака за „непостоење“ (*NULL* или *-1*). Идејата е прикажана на Слика 2.3.

Во продолжение е дадена имплементацијата на моделот во C++.

```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 30

typedef int IzbranTip;
typedef int OznakaTip;
```

```
typedef struct {
	IzbranTip podatok;
	OznakaTip oznaka;
	OznakaTip roditel;
	} JazolTip;

typedef struct {
	int brojJazli;
	JazolTip *jazol[MAX_DOLZINA];
	} Steblo;

void Init(OznakaTip o, IzbranTip d, Steblo *T) {
	T->brojJazli = 1;
	T->jazol[0] = (JazolTip*)malloc(sizeof(JazolTip));
	T->jazol[0]->oznaka = o;
	T->jazol[0]->podatok = d;
	T->jazol[0]->roditel = NULL;
	for(int i=1; i<MAX_DOLZINA; i++)
		T->jazol[i]=NULL;
}

bool EKoren(JazolTip *j, Steblo *T) {
	return ((j->roditel == NULL)&&(T->jazol[0] == j));
}

bool ImaOznaka(OznakaTip o, Steblo *T) {
	bool rez = false;
	for(int i=0;i<T->brojJazli;i++)
		if (T->jazol[i]->oznaka == o) rez = true;
	return rez;
}

JazolTip *Koren(Steblo *T) {
	return T->jazol[0];
}


IzbranTip Vrednost(JazolTip *j, Steblo *T) {
	return j->podatok;
}

void DodadiDete(OznakaTip o, IzbranTip d, JazolTip *j, Steblo *T) {
	if (T->brojJazli>=MAX_DOLZINA)
		perror("Nema mesto za novi jazli...");
	else if (!ImaOznaka(j->oznaka, T))
		perror("Ovoj jazol ne se naodja vo stebloto...");
	else if (ImaOznaka(o, T))
		perror("Oznakata postoi vo stebloto...");
	{
```

```
                int br = 0;
                while ((br < T->brojJazli) &&
                                (T->jazol[br]->roditel != j->oznaka))
                                        br++;
                T->brojJazli++;
                for(int i=T->brojJazli-1; i>br; i--)
                        T->jazol[i] = T->jazol[i-1];
                T->jazol[br] = (JazolTip*)malloc(sizeof(JazolTip));
                T->jazol[br]->oznaka = o;
                T->jazol[br]->podatok = d;
                T->jazol[br]->roditel = j->oznaka;
        }
}

void DodadiBrat(OznakaTip o, IzbranTip d, JazolTip *j, Steblo *T) {
        if (EKoren(j,T))
                perror("Korenot ne moze da ima brat...");
        else if (j==NULL)
                perror("Korenot e NULL...");
        if (T->brojJazli >= MAX_DOLZINA-1)
                perror("Stebloto e polno");
        else if (ImaOznaka(o, T))
                perror("Oznakata postoi vo stebloto...");
        else
        {
                int br=0;
                while (j!=T->jazol[br]) br++;
                for(int i=T->brojJazli; i>br+1; i--)
                        T->jazol[i] = T->jazol[i-1];
                T->brojJazli++;
                br++;
                T->jazol[br] = (JazolTip*)malloc(sizeof(JazolTip));
                T->jazol[br]->oznaka = o;
                T->jazol[br]->podatok = d;
                T->jazol[br]->roditel = j->roditel;
        }
}

OznakaTip Oznaka(JazolTip *j, Steblo *T) {
        return j->oznaka;
}

JazolTip *PrvoDete(JazolTip *j, Steblo *T) {
        if (ImaOznaka(j->oznaka, T)) {
                bool rez = false;
                int br=0;
                while ((!rez)&&(br<T->brojJazli-1))
                {
                        br++;
                        rez = (T->jazol[br]->roditel==j->oznaka);
```

```
                }
                if (rez) return T->jazol[br];
                else return NULL;
        }
        else return NULL;
}

void Brisi(JazolTip *j, Steblo *T) {
        if (EKoren(j, T))
                perror("Korenot ne moze da se izbrise...");
        else if (PrvoDete(j, T) != NULL)
                perror("Jazolot ima deca. Ne moze da se izbrise...");
        else
        {
                int br=0;
                while (j != T->jazol[br]) br++;
                for(int i=br; i<T->brojJazli-1; i++)
                        T->jazol[i] = T->jazol[i+1];
                T->brojJazli--;
                T->jazol[T->brojJazli] = NULL;
        }
}

JazolTip *SledenBrat(JazolTip *j, Steblo *T) {
        if (!ImaOznaka(j->oznaka, T))
                perror("Ne postoi takov jazol");
        else
        {
                int br=0;
                while (j!=T->jazol[br]) br++;
                br++;
                if (br>=MAX_DOLZINA) return NULL;
                else if (j->roditel == T->jazol[br]->roditel)
                                return T->jazol[br];
                        else return NULL;
        }
}

JazolTip *Roditel(JazolTip *j, Steblo *T) {
        if (!ImaOznaka(j->oznaka, T))
                perror("Ne postoi takov jazol");
        else if (Koren(T)==j)
                perror("Korenot nema roditel");
        else
        {
                int br=1;
                while (j->roditel!=T->jazol[br]->oznaka) br++;
                return T->jazol[br];
        }
}
```

```
void SmeniOznaka(OznakaTip o, JazolTip *j, Steblo *T) {
	if (ImaOznaka(o,T))
		perror("Oznakata vekje postoi");
	else
	{
		for(int i=0;i<T->brojJazli;i++)
		{
			if (T->jazol[i]->roditel == j->oznaka)
				T->jazol[i]->roditel = o;
		}
		j->oznaka = o;
	}
}
```

Како што може да се види и во кодот, реализацијата на стеблото е со низа од јазли од каде секој содржи ознака, податоци и ознака за родителскиот елемент т.е. запис кој одговара на редица од табелата од Слика 2.3. Операциите кај оваа имплементација не се едноставни бидејќи секое додавање и бришење на јазол од стеблото бара поместување на елементите надолу т.е. нагоре во полето. Ова значи дека оваа имплементација е добра ако нема многу додавања и бришења на јазли. Доколку не е потребно подредување на братските јазли, оваа имплементација во голема мера може да биде упростена, особено во делот за имплементација на операциите DodadiBrat(), PrvoDete(), SledenBrat() и слично, во спротивно, редоследот на децата на еден јазол е определен со редоследот на јазлите-деца во полето.

Имплементација преку релација јазол-(прво дете, следен брат)

Оваа имплементација е слична како претходната т.е. се користи поле составено од неколку компоненти кои содржат ознаки на јазли кои треба да претставуваат врски кон други јазли. Во претходната имплементација полето содржеше ознака на јазолот, податоци и ознака на родителскиот јазол, при што првото дете беше утврдувано според позицијата во табелата.

Доколку при решавање на проблемот, подредувањето на јазлите од едно ниво е важно и има многу операции во врска со овој редослед имплементацијата преку релацијата јазол и двојката (прво дете, следен брат) е подобар избор во однос на претходниот. Со оглед на тоа што пристапите се слични, а и операциите се слични како во претходната имплементација, оваа имплементација нема да биде дадена како комплетно решение, туку само одредени сегменти.

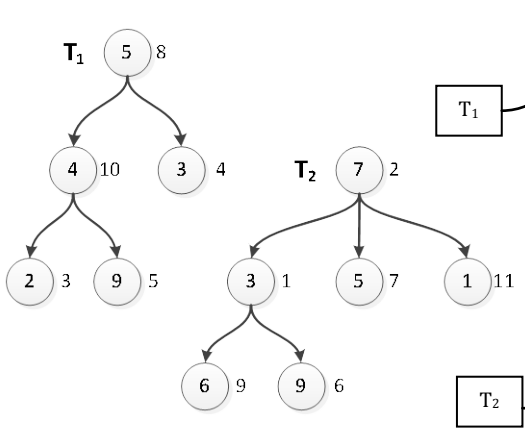
Оваа имплементација се реализира така што на секој јазол експлицитно се запишува неговото прво дете и неговиот следен брат. Врската од јазолот до детето, т.е. до следниот брат, може да се реализира и со курсор и со покажувач.

Во продолжение е дефиницијата на структурата:

```
typedef struct {
       IzbranTip podatok;
       OznakaTip oznaka;
       OznakaTip prvoDete;
       OznakaTip sledenBrat;
       } JazolTip;

JazolTip Prostor[MAX_DOLZINA];
```

Овој модел има уште една дополнителна предност, а тоа е што е погоден за проблеми кога има потреба за комбинирање на повеќе стебла во едно. Ова се овозможува со резервација на целокупниот простор со глобалното поле *Prostor[]*, при што полето претставува залиха од ќелии од кои ќе се градат стеблата (Слика 2.4).

Стеблото е имплементирано како поврзана структура од ќелии. Стеблото се определува со курсор на коренот т.е. typedef int Steblo, а различните стебла со кои се работи користат ќелии од исто (единствено) поле *Prostor[]*.



ознака	податок	прво дете	следен брат
8	5	10	NULL
10	4	3	4
7	5	NULL	11
11	1	NULL	NULL
6	9	NULL	NUL
4	3	NULL	NULL
3	2	NULL	5
5	9	NULL	NULL
2	7	1	NULL
9	6	NULL	6
1	3	9	7

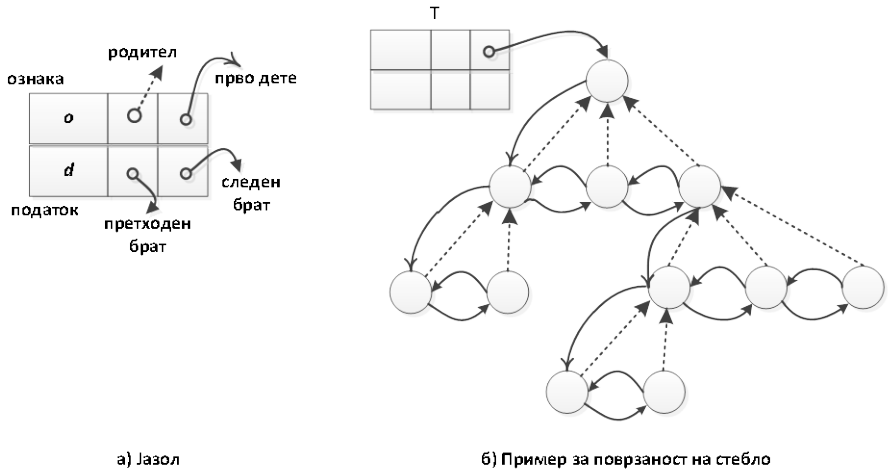
Слика 2.4. Стеблата T_1 и T_2 користат ќелија од исто поле

Ако се игнорира информацијата за следниот брат, подстеблата може да се референцираат како посебни стебла, затоа што во ваквата имплементација недостасува информацијата од јазол кон неговиот родител.

Операцијата *Roditel()* е најкомплицирана и би се имплементирала со испитување на сите јазли со ред во просторот *Prostor[]* или пак со воведување на дополнителната ознака за родител во секој јазол.

Имплементација со покажувачи

АПТ (општо) стебло е навистина комплексна структура за имплементација со покажувачи. Најчесто ваквите стебла се трансформираат во бинарни стебла (кои ќе бидат разгледани понатаму), но во продолжение е дадена една од можните имплементации на стебло со покажувачи. Како илустрација, на Слика 2.5-а е прикажан јазол за оваа структура, а на Слика 2.5-б е прикажан примерок на стеблото кое во продолжение е имплементирано.



Слика 2.5 Имплементација на АПТ стебло со покажувачи

```
#include <iostream>
#include <stdio.h>

typedef int IzbranTip;
typedef int OznakaTip;

typedef struct Jazol_Oznaka {
             IzbranTip podatok;
             OznakaTip oznaka;
             Jazol_Oznaka  *sledenBrat, *prethodenBrat,
                           *roditel, *prvoDete;
             } Steblo;
```

```
void Init(OznakaTip o, IzbranTip d, Steblo *T) {
	T->prvoDete = (Steblo *)malloc(sizeof(Steblo));
	T->prvoDete->prvoDete = NULL;
	T->prvoDete->oznaka = o;
	T->prvoDete->podatok = d;
	T->prvoDete->roditel = NULL;
}

bool EKoren(Steblo *j) {
	return (j->roditel==NULL);
}

Steblo *Roditel(Steblo *j) {
	if (j!=NULL)
		return j->roditel;
	else	return NULL;
}
void DodadiDete(OznakaTip o, IzbranTip d, Steblo *j) {
	if (j->prvoDete==NULL) {
		j->prvoDete = (Steblo*)malloc(sizeof(Steblo));
		j->prvoDete->sledenBrat = NULL;
	}
	else {
		j->prvoDete->prethodenBrat =
				(Steblo*)malloc(sizeof(Steblo));
		j->prvoDete->prethodenBrat->sledenBrat = j->prvoDete;
		j->prvoDete = j->prvoDete->prethodenBrat;
	}
		j->prvoDete->oznaka = o;
		j->prvoDete->roditel = j;
		j->prvoDete->podatok = d;
		j->prvoDete->prvoDete = NULL;
		j->prvoDete->prethodenBrat = NULL;
}

void DodadiBrat(OznakaTip o, IzbranTip d, Steblo *j) {
	if (EKoren(j))
		perror("Korenot ne moze da ima brat...");
	else if (j==NULL)
		perror("Korenot e NULL...");
	else {
		if (j->sledenBrat==NULL) {
			j->sledenBrat =
				(Steblo *)malloc(sizeof(Steblo));
			j->sledenBrat->sledenBrat = NULL;
		}
		else {
			j->sledenBrat->prethodenBrat =
				(Steblo *)malloc(sizeof(Steblo));
```

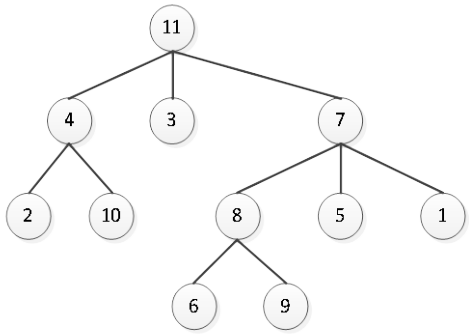
```
                        j->sledenBrat->prethodenBrat->sledenBrat =
                                j->sledenBrat;
                        j->sledenBrat = j->sledenBrat->prethodenBrat;
                };
                j->sledenBrat->prethodenBrat = j;
                j->sledenBrat->roditel = j->roditel;
                j->sledenBrat->oznaka = o;
                j->sledenBrat->podatok = d;
                j->sledenBrat->prvoDete = NULL;
        }
}

Steblo *PrvoDete(Steblo *j) {
        if (j!=NULL) return j->prvoDete;
        else   return NULL;
}


Steblo *SledenBrat(Steblo *j) {
        if (j!=NULL)
                return j->sledenBrat;
        else   return NULL;
}

Steblo *Koren(Steblo *T) {
        return T->prvoDete;
}

void Brisi(Steblo *j) {
        if (EKoren(j))
                perror("Korenot ne moze da se izbrise...");
        else if (j->prvoDete!=NULL)
                perror("Jazolot ima deca. Ne moze da se izbrise...");
        else
        {
                if (j->roditel->prvoDete==j)
                {
                        j->roditel->prvoDete=j->sledenBrat;
                        if (j->sledenBrat!=NULL)
                                j->sledenBrat->prethodenBrat = NULL;
                }
                else
                {
                        if (j->sledenBrat!=NULL)
                                j->sledenBrat->prethodenBrat =
                                        j->prethodenBrat;
                        j->prethodenBrat->sledenBrat = j->sledenBrat;
                }
                        free(j);
        }
```

```
}

IzbranTip Vrednost(Steblo *j) {
	return j->podatok;
}

OznakaTip Oznaka(Steblo *j) {
	return j->oznaka;
}

void SmeniOznaka(OznakaTip o, Steblo *j) {
	if (j!=NULL) j->oznaka = o;
}
```

2.1.3 Поминување низ стебло

Поминување низ стебло е алгоритам со кој се гарантира еднократна „посета“ на секој јазол во едно стебло. Ваквиот алгоритам се користи кога треба да се изврши одредено процесирање на секој јазол во стеблото, со што алгоритмот поставува редослед за еднократно процесирање на сите јазли.

Најпознати алгоритми за поминување низ стебло се *PREORDER()*, *INORDER()* и *POSTORDER()*, па во продолжение се дадени нивните рекурзивни дефиниции.



PREORDER : 11, 4, 2, 10, 3, 7, 8, 6, 9, 5, 1
INORDER: 2, 4, 10, 11, 3, 6, 8, 9, 7, 5, 1
POSTORDER: 2, 10, 4, 3, 6, 9, 8, 5, 1, 7, 11
Слика 2.6. Примери за поминување низ

Нека стеблото T е составено од корен r и подстеблата на коренот T_1, T_2, … T_n. Тогаш:

- *PREORDER()* е алгоритам кој прво го процесира коренот *r*, а потоа подстеблата T_1, T_2 до T_n

- *INORDER()* е алгоритам кој прво го процесира подстеблото T_1, потоа го процесира коренот r и на крај подстеблата T_2, ... T_n
- *POSTORDER()* е алгоритам кој прво ги процесира подстеблата T_1, T_2 до T_n и на крај коренот r

Разликите меѓу овие три методи се видливи во примерот на Слика 2.6.

Имплементацијата на овие алгоритми може да се направи со запишување на алгоритмите во облик на потпрограми во рамките на апстрактниот податочен тип стебло.

Во продолжение се дадени алгоритмите во C++ кои ги испишуваат вредностите на јазлите и се имплементираат во рамки на имплементацијата на овој АПТ со покажувачи.

```
void PreOrder(Steblo *j) {
	if (j==NULL) return;
	std::cout<<Vrednost(j)<<" ";
	Steblo * pd = PrvoDete(j);
	PreOrder(pd);
	while (SledenBrat(pd)!=NULL)
	{	pd = SledenBrat(pd);
		PreOrder(pd);
	}
}
void InOrder(Steblo *j) {
	if (j==NULL) return;
	Steblo * pd = PrvoDete(j);
	InOrder(pd);
	std::cout<<Vrednost(j)<<" ";
	while (SledenBrat(pd)!=NULL)
	{	pd = SledenBrat(pd);
		InOrder(pd);
	}
}
void PostOrder(Steblo *j) {
	if (j==NULL) return;
	Steblo * pd = PrvoDete(j);
	PostOrder(pd);
	while (SledenBrat(pd)!=NULL)
	{	pd = SledenBrat(pd);
		PostOrder(pd);
	};
	std::cout<<Vrednost(j)<<" ";
```

```
}
```

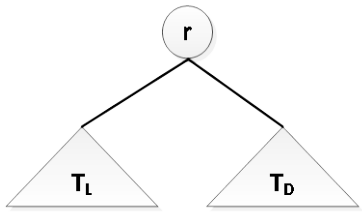
Како што може да се види и во самата имплементација, разликите меѓу алгоритмите се минорни т.е. се работи само за промена на редоследот на извршување на наредбите во внатрешноста на рекурзијата.

Во продолжение е дадена и потпрограма која го креира стеблото од примерот на Слика 2.6, а потоа ги повикува трите алгоритми за поминување низ стеблото.

```
int main()
{
	Steblo * TT = (Steblo *)malloc(sizeof(Steblo));
	Init(0,11,TT);
	DodadiDete(1,7,Koren(TT));
	DodadiDete(2,3,Koren(TT));
	DodadiDete(3,4,Koren(TT));
	DodadiDete(4,10,PrvoDete(Koren(TT)));
	DodadiDete(5,2,PrvoDete(Koren(TT)));
	DodadiDete(6,1,SledenBrat(SledenBrat(PrvoDete(Koren(TT)))));
	DodadiDete(7,5,SledenBrat(SledenBrat(PrvoDete(Koren(TT)))));
	DodadiDete(8,8,SledenBrat(SledenBrat(PrvoDete(Koren(TT)))));
	DodadiDete(9,9,PrvoDete(SledenBrat(SledenBrat(
                                PrvoDete(Koren(TT))))));
	DodadiDete(10,6,PrvoDete(SledenBrat(SledenBrat(
                                PrvoDete(Koren(TT))))));
	std::cout<<"PREORDER: ";
	PreOrder(Koren(TT));
	std::cout<<"\nINORDER: ";
	InOrder(Koren(TT));
	std::cout<<"\nPOSTORDER: ";
	PostOrder(Koren(TT));
	system("pause");
}
```

2.2 Бинарно стебло (Binary tree)

Кај компјутерските науки бинарните стебла се најупотребуваниот вид на стеблата. Тоа се стебла со степен 2 и претставуваат поедноставни и поправилно изградени структури кои се многу поедноставни за реализација кај компјутерите.



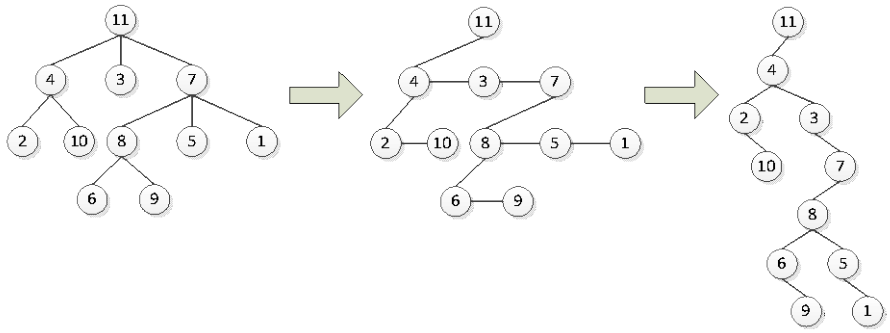
Слика 2.7. Бинарно стебло

Бинарно стебло T е конечно множество од податоци од ист тип кои се нарекуваат јазли, каде T е празно множество (празно стебло) или постои

еден јазол r кој се нарекува **корен** на стеблото T, а останатите јазли градат две помали дисјунктни бинарни стебла T_L и T_D, кои претставуваат **подстебла** на јазолот r, и T_L се нарекува лево подстебло, а T_D десно подстебло. Ако постои коренот на T_L тогаш тој е **лево дете** на r, а ако постои коренот на T_D тогаш тој е **десно дете** на r, а r е нивниот **родител**. Останатата терминологија е иста како кај стеблата и се применуваат истите алгоритми за поминување низ стеблото.

Важно е да се напомене дека бинарното стебло не е специјален случај на стеблата, имено бинарно стебло може да биде и празно, а и ако еден јазол има само едно дете тогаш кај бинарните стебла е важно дали е тоа левото или десното дете.

Секое (општо) стебло може да се интерпретира како бинарно стебло, и тоа врз основа на врската јазол-(прво дете, следен брат). Оваа интерпретација се врши на тој начин што за секој јазол од едно стебло како лево дете се поврзува неговото прво дете, а како десно неговиот следен брат. Пример за ваква интерпретација е прикажан на Слика 2.8.



Слика 2.8. Интерпретација на стебло како бинарно стебло

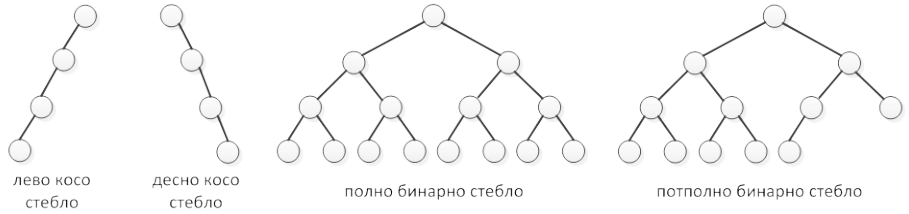
Примери за проблеми кои може да се решат со бинарните стебла се среќаваат насекаде во реалниот свет. Бинарните стебла се користат и за имплементација на други посложени структури, но и за поедноставување на најразлични алгоритми кои ќе бидат тема на следните поглавја, па детално разгледување на оваа структура и сите операции кои би биле корисни за неа се од важно значење.

Од дефиницијата на бинарните стабла следат заклучоците дека:

- максимален број на јазли на k-тото ниво е 2^k
- максимален број на јазли на бинарното стебло со висина k е $2^{k+1}-1$

Стеблото со висина k кое има $2^{k+1}-1$ јазли се нарекува **полно бинарно стебло**, а бинарното стебло во кое разликата на нивоата на крајните јазли е

најмногу 1 се нарекува **потполно бинарно стебло.** Бинарното стебло со висина *k* и вкупно *k+1* јазли од кои *k* имаат само лево (десно) дете, тогаш тоа бинарно стебло се нарекува **лево (десно) косо стебло**.



Слика 2.9. Видови бинарни стебла

2.2.1 Апстрактен податочен тип БИНАРНО СТЕБЛО

Бинарните стебла можат на различни начини да се дефинираат како апстрактен податочен тип. Операциите кои работат на ниво на јазол, но и операциите кои работат на ниво на подстебло лесно може да се дефинираат. Од друга страна некои операции се изведуваат и едни од други.

IzbranTip - било кој тип на податоци

oznaka - било кој тип наменет за означување на јазлите

jazol - јазол на бинарно стебло составен од дел со информации од типот *IzbranTip* и два покажувачи кон други јазли, едниот за левото, другиот за десното дете. Секој јазол може да има ознака од типот *oznaka*.

BINsteblo - податок од овој тип е бинарно стебло чии јазли се податоци од типот *jazol* кои се различни меѓу себе и не се *NULL*.

Init(&Bt) - функција која го претвора стеблото *Bt* во празно стебло.

Prazno(Bt) - функција која враќа информација ако и само ако *Bt* е празно стебло

SpojVoSteblo(o, d, Tl, Td, &Bt) - функција која го спојува бинарното стебло *Tl* како лево и *Td* како десно подстебло на новокореираниот јазол со ознака *o* и податок *d*.

LevoPodsteblo(j) и ***DesnoPodsteblo(j)*** - функции кои враќаат лево т.е. десно подстебло од јазолот *j*

DodadiLevo(o, d, &j) и ***DodadiDesno(o, d, &j)*** - функции кои додаваат нов јазол со вредност *d* како лево т.е. десно дете на јазолот *j* и го означуваат

новиот јазол со *o*. Ако јазолот веќе има лево т.е. десно дете, функциите не се дефинирани.

Brisi(j, &T) - функција која го брише јазолот *j* од стеблото *T*. Не е дефинирана ако *j* е корен на стеблото, ако *j* не припаѓа на *T* или ако *j* има деца.

Koren(T) - функција која го враќа коренот на стеблото *T*

EKoren(j, T) - функција која испитува дали јазолот *j* е корен на стеблото *T*

LevoDete(j)* и *DesnoDete(j) - функции кои го враќаат левото т.е. десното дете на јазолот *j* од стеблото.

Roditel(j, T) - функција која го враќа јазолот кој е родител на јазолот *j* во стеблото *T.*

Vrednost(j) - функција која ја враќа вредноста на јазолот *j*

Oznaka(j) - функција која ја враќа ознаката на јазолот *j*

SmeniOznaka(o, &j) - функција која ја менува ознаката на јазолот *j* во нова - *o*.

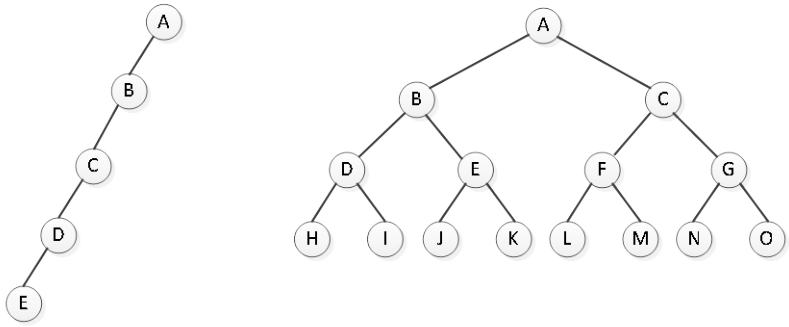
2.2.2 Имплементација на АПТ БИНАРНО СТЕБЛО

Имплементација со поле

Бинарните стебла едноставно се прикажуваат со еднодимензионална низа без податоци за поврзување и користат правила за одредување на релациите во стеблата. Користењето на низата започнува од членот со индекс *1* поради поедноставување на изразите. Правилата за бинарно стебло со *n* јазли, за *i*-тиот јазол се:

- *Roditel(i) = [i/2]* за *i≠ 1;* кога *i=1*, јазолот е корен па нема родител
- *LevoDete(i)=2*i* ако *2*i≤ n*; кога *2*i>n* јазолот нема лево дете
- *DesnoDete(i)=2*i+1* ако *2*i+1≤n*; кога *2*i+1>n* јазолот нема десно дете

На овој начин може да се прикажат сите бинарни стебла, но тогаш не се користи ефикасно меморијата. Најлош случај е кога треба да се смести косо стебло кое користи само *k* локации од 2^k-1 локации предвидени за тоа стебло.



КОСО

ПОЛНО

Слика 2.10. Имплементација на АПТ бинарно стебло со поле

Оваа метода е практична ако се применува на потполно бинарно стебло, каде тоа ќе биде замислено како статичка структура и не се планирани операции од типот *LevoPodsteblo()* или *DesnoPodsteblo()*, туку стеблото ќе служи само за поминување низ стеблото, читање или менување на ознаки, или евентуално додавање или бришење на јазли на десниот крај на последното ниво. Ова е затоа што имплементацијата е доста комплексна и бара многу математички операции, па во продолжение може да се види истата.

```
#include <iostream>
#include <stdio.h>

#define MAX_DOLZINA 127 //od tip (2^k) - 1

int Log2(int k) {
	if (k<1) return NULL;
	return (int)floor(log((float)(k)) / log(2.0));
}

int pow2(int k) {
	return 1 << k;
}

int MAX_VISINA = Log2(MAX_DOLZINA);

typedef int IzbranTip;
typedef int OznakaTip;


typedef struct {
```

```
                IzbranTip podatok;
                OznakaTip oznaka;
                } Jazol;

typedef struct {
        int visina;
        Jazol *jazol[MAX_DOLZINA+1];
        } BINsteblo;

void Init(BINsteblo *Bt) {
        for(int i=1;i<=MAX_DOLZINA;i++)
                Bt->jazol[i] = NULL;
        Bt->visina = -1;
}

int PresmetajVisina(BINsteblo *Bt) {
        int ind = MAX_DOLZINA;
        while ((ind>=1)&&(Bt->jazol[ind]==NULL)) ind--;
        if (ind==0)
                        return -1;
                else
                        return Log2(ind);
}

bool Prazno(BINsteblo *Bt) {
        return (Bt->visina<0);
}

int Nivo(Jazol *j, BINsteblo *Bt) {
        return Log2(j->oznaka);
}

bool EKoren(Jazol *j, BINsteblo *Bt) {
        return (Bt->jazol[1]==j);
}

Jazol *KreirajJazol(OznakaTip o, IzbranTip d) {
        Jazol *q = (Jazol*)malloc(sizeof(Jazol));
        q->oznaka = o;
        q->podatok = d;
        return q;
}

void Brisi(Jazol *j, BINsteblo *Bt) {
        if (j!=NULL) free(j);
        Bt->visina = PresmetajVisina(Bt);
}


void SpojVoSteblo(IzbranTip d, BINsteblo *Tl,
```

```
                              BINsteblo *Td, BINsteblo *Bt) {
        Init(Bt);
        Bt->jazol[1]->oznaka = 0;
        Bt->jazol[1]->podatok = d;
        for (int k=0;k<MAX_VISINA-1;k++)
               for (int i=0;i<pow2(k);i++)
               {
               Bt->jazol[pow2(k+1)+i]->podatok =
                              Tl->jazol[pow2(k)+i]->podatok;
               Bt->jazol[pow2(k)*3+i]->podatok =
                              Td->jazol[pow2(k)+i]->podatok;
               Bt->jazol[pow2(k+1)+i]->oznaka = pow2(k+1)+i;
               Bt->jazol[pow2(k)*3+i]->oznaka = pow2(k)*3+i;
               }
               Bt->visina = PresmetajVisina(Bt);
}

Jazol *LevoDete(Jazol *j, BINsteblo *Bt) {
        if (j==NULL) return NULL;
        if (j->oznaka*2>MAX_DOLZINA) return NULL;
        return Bt->jazol[j->oznaka*2];
}

Jazol *DesnoDete(Jazol *j, BINsteblo *Bt) {
        if (j==NULL) return NULL;
        if (j->oznaka*2+1>MAX_DOLZINA) return NULL;
        return Bt->jazol[j->oznaka*2+1];
}

Jazol *Koren(BINsteblo *Bt) {
        if (Prazno(Bt)) return NULL;
        return Bt->jazol[1];
}

BINsteblo *LevoPodsteblo(Jazol *j, BINsteblo *Bt) {
        BINsteblo *q = (BINsteblo*)malloc(sizeof(BINsteblo));
        Init(q);
        int p = j->oznaka;
        for(int k=0;k<Bt->visina-Nivo(j,Bt);k++)
               for(int i=0;i<pow2(k);i++)
                 q->jazol[pow2(k)+i] =
                      KreirajJazol(pow2(k)+i,
                              Bt->jazol[pow2(k+1)*p+i]->podatok);
        q->visina = PresmetajVisina(q);
        return q;
}



BINsteblo *DesnoPodsteblo(Jazol *j, BINsteblo *Bt) {
```

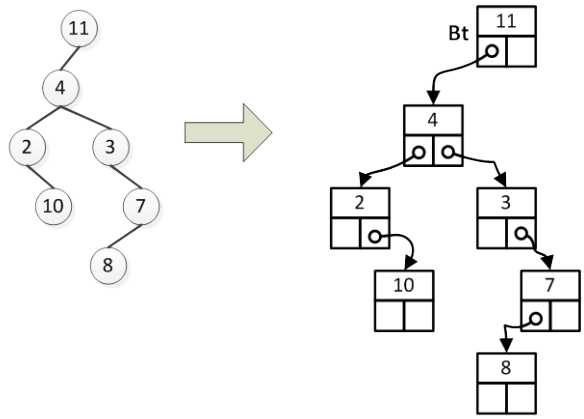
```
        BINsteblo *q = (BINsteblo*)malloc(sizeof(BINsteblo));
        Init(q);
        int p = j->oznaka;
        for(int k=0;k<Bt->visina-Nivo(j,Bt);k++)
                for(int i=0;i<pow2(k);i++)
                  q->jazol[pow2(k)+i] =
                        KreirajJazol(pow2(k)+i,
                                Bt->jazol[3*pow2(k)*p+i]->podatok);
        q->visina = PresmetajVisina(q);
        return q;
}

void DodadiLevo(IzbranTip d, Jazol *j, BINsteblo *Bt) {
        if (j->oznaka*2>MAX_DOLZINA)
                        perror("Nema mesto...");
        else if (Bt->jazol[j->oznaka*2]!=NULL)
                perror("Jazolot ima levo dete...");
        else
        {
                Bt->jazol[j->oznaka*2] =
                        KreirajJazol(j->oznaka*2,d);
                if (Bt->visina < Nivo(Bt->jazol[j->oznaka*2], Bt))
                        Bt->visina = Nivo(Bt->jazol[j->oznaka*2], Bt);
        };
}

void DodadiDesno(IzbranTip d, Jazol *j, BINsteblo *Bt) {
        if ((j->oznaka*2+1)>MAX_DOLZINA)
                        perror("Nema mesto...");
        else if (Bt->jazol[j->oznaka*2+1]!=NULL)
                perror("Jazolot ima desno dete...");
        else
        {
                Bt->jazol[j->oznaka*2+1] =
                        KreirajJazol(j->oznaka*2+1,d);
                if (Bt->visina < Nivo(Bt->jazol[j->oznaka*2+1], Bt))
                        Bt->visina =
                                Nivo(Bt->jazol[j->oznaka*2+1], Bt);
        }
}

Jazol *Roditel(Jazol *j, BINsteblo *Bt) {
        if (EKoren(j, Bt))
                perror("Korenot nema roditel...");
        else
                return Bt->jazol[j->oznaka / 2];
}


IzbranTip Vrednost(Jazol *j) {
```

```
        if (j==NULL) return NULL;
        return j->podatok;
}

OznakaTip Oznaka(Jazol *j) {
        if (j==NULL) return NULL;
        return j->oznaka;
}
```

Евидентно е дека ваквата имплементација на операциите се базира на сложени алгоритми. Операциите за манипулација со цели подстебла, SpojVoSteblo(), LevoPodsteblo(), DesnoPodsteblo(), се базира на креирање на нови полиња и пресметување на нови индекси на јазлите во новите полиња.

Имплементација со покажувачи

Најприродна метода за имплементација на бинарно стебло е методата со покажувачи. На секој јазол експлицитно му се поврзува неговото лево и неговото десно дете, а врската може да се опише со покажувач. На Слика 2.11 е прикажана имплементацијата на бинарно стебло со помош на покажувачи. Секој јазол се прикажува со една ќелија, а тој е еднозначно определен со покажувач на таа ќелија.



Слика 2.11. Имплементација на АПТ бинарно стебло со

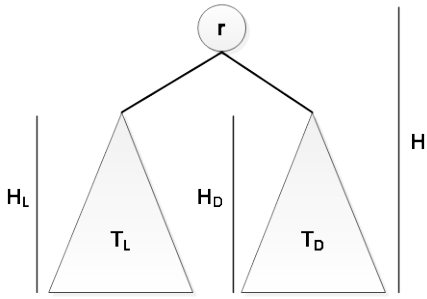
Бинарното стебло се гради како структура од меѓусебно поврзани ќелии, а самото стебло се референцира со покажувач на коренот. Празно стебло значи дека покажувачот на коренот покажува на *NULL*. Доколку во имплементацијата е потребна операцијата *Roditel(),* тогаш згодно е да во ќелијата се додаде уште еден покажувач кон родителскиот елемент. Во последниот случај, покажувачот кон родителот би покажувал на *NULL*.

Со оглед на тоа што имплементацијата на општото стебло во претходното поглавје опфаќа врска од дете кон родител, јасна е идејата и начинот на користењето на оваа врска, па поради поедноставување на оваа имплементација, оваа врска, а и операцијата *Roditel()*, ќе биде испуштена. Дополнително поедноставување на самата имплементација е направено со испуштање и на ознаките, вклучително и сите операции во врска со нив.

Поради внимателно избраниот модел, каде секој јазол е третиран како посебно стебло, добиено е дополнително поедноставување на имплементацијата поради тоа што операциите Koren(), LevoDete() и DesnoDete() се поклопуваат со самото стебло, операцијата LevoPodsteblo() и операцијата DesnoPodsteblo(), соодветно.

```
#include <iostream>
#include <stdio.h>
typedef int IzbranTip;

typedef struct Jazol_Oznaka {
              IzbranTip podatok;
              Jazol_Oznaka *lDete, *dDete;
              } BINsteblo;

void Init(BINsteblo *Bt) {
       Bt = NULL;
}

bool Prazno(BINsteblo *Bt) {
       return (Bt==NULL);
}

BINsteblo *KreirajJazol(IzbranTip d) {
       BINsteblo *q = (BINsteblo*)malloc(sizeof(BINsteblo));
       q->podatok = d;
       q->lDete = NULL;
       q->dDete = NULL;
       return q;
}

void SpojVoSteblo(IzbranTip d, BINsteblo *Tl,
                                    BINsteblo *Td, BINsteblo *Bt) {
       Bt = KreirajJazol(d);
       Bt->lDete = Tl; Bt->dDete = Td;
}
BINsteblo *LevoPodsteblo(BINsteblo *Bt) {
       return Bt->lDete;
}

BINsteblo *DesnoPodsteblo(BINsteblo *Bt) {
       return Bt->dDete;
```

```
}

void Brisi(BINsteblo *Bt) {
	if (Bt!=NULL) free(Bt);
}

BINsteblo *DodadiLevo(IzbranTip d, BINsteblo *Bt) {
	if (Bt->lDete!=NULL) return NULL;
	Bt->lDete = KreirajJazol(d);
	return Bt->lDete;
}

BINsteblo *DodadiDesno(IzbranTip d, BINsteblo *Bt) {
	if (Bt->dDete!=NULL) return NULL;
	Bt->dDete = KreirajJazol(d);
	return Bt->dDete;
}

int Visina(BINsteblo *Bt) {
	if (Prazno(Bt))
		return -1;
	if (Visina(Bt->lDete)>Visina(Bt->dDete))
		return 1+Visina(Bt->lDete);
	else
		return 1+Visina(Bt->dDete);
}
```



H = 1 + MAX(H_L, H_D)

каде

H е висина на стеблото

H_L е висина на левото подстебло

H_D е висина на десното подстебло

Слика 2.12. Рекурзивна дефиниција на операцијата Visina() кај бинарно стебло

```
IzbranTip Vrednost(BINsteblo *Bt) {
	if (Bt==NULL) return NULL;
	return Bt->podatok;
}
void PreOrder(BINsteblo *Bt) {
	if (Bt==NULL) return;
	std::cout<<Vrednost(Bt)<<" ";
	PreOrder(Bt->lDete);
	PreOrder(Bt->dDete);
}
```

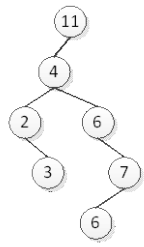
```
void InOrder(BINsteblo *Bt) {
       if (Bt==NULL) return;
       InOrder(Bt->lDete);
       std::cout<<Vrednost(Bt)<<" ";
       InOrder(Bt->dDete);
}

void PostOrder(BINsteblo *Bt) {
       if (Bt==NULL) return;
       PostOrder(Bt->lDete);
       PostOrder(Bt->dDete);
       std::cout<<Vrednost(Bt)<<" ";
}
```

Како илустрација за едноставноста на моделот и едноставната имплементација на операции над овој модел, во кодот е вклучена и имплементацијата на алгоритмите за поминување низ бинарно стебло *InOrder()*, *PreOrder()* и *PostOrder()*. Овие алгоритми се дадени како можно решение за гореспоменатата операција Roditel() ако е потребна имплементација на оваа операција во овој модел. Имено, со било кој од алгоритмите ќе може да се помине низ стеблото до наоѓање на јазолот чие дете е влезниот јазол во операцијата Roditel(). За разлика од претходната имплементација, операцијата Visina() најдобро е да биде решена рекурзивно, што е случај во ова решение (Слика 2.12).

2.2.3 Апстрактен податочен тип БИНАРНО ПРЕБАРУВАЧКО СТЕБЛО

Бинарно пребарувачко стебло е специјален случај на бинарно стебло каде јазлите од тоа стебло се означени со податоци над кои постои дефинирано подредување и каде за било кој јазол ознаките во целото негово лево подстебло се помали од неговата ознака, а ознаките во целото негово десно подстебло се поголеми или еднакви од неговата ознака, според дефинираното подредување. На Слика 2.13 е прикажан пример за бинарно пребарувачко стебло.

Идејата за оваа структура е тоа што со поминување на стеблото со *INORDER* методата се добива сортирана листа во однос на дефинираното подредување. Примената на ваквиот вид стебло е широка, кај различни имплементации на други апстрактни податочни типови, како што е примерот со речниците, но и кај имплементација на различни видови алгоритми, како што се алгоритмите за сортирање (Види Поглавје 7.7.).



Слика 2.13. Бинарно пребарувачко стебло

АПТ бинарно пребарувачко стебло ги има следните карактеристики:

IzbranTip - било кој тип на податоци

oznaka - било кој тип наменет за означување на јазлите

jazol - јазол на бинарно стебло составен од дел со информации од типот *IzbranTip* и два покажувачи кон други јазли, едниот за левото, другиот за десното дете. Секој јазол може да биде означен со ознака од типот *oznaka*.

BSsteblo - податок од овој тип е бинарно стебло чии јазли се податоци од типот *jazol* кои се различни меѓу себе и не се *NULL*.

Init(&BS) - функција која го претвора стеблото *BS* во празно бинарно пребарувачко стебло.

KreirajSteblo(o, d) – функција која враќа ново бинарно стебло со корен со ознака *o* и содржина *d*

LevoDete(j) **и** ***DesnoDete(j)*** - функции кои го враќаат левото т.е. десното дете на јазолот *j* од стеблото.

Dodadi(o, d, &j) – рекурзивна функција која додава нов јазол со вредност *d* во подстеблото чиј корен е јазолот *j* и го означуваат новиот јазол со *o*. Ако јазолот *j* има ознака помала или еднаква од ознаката *o*, тогаш новиот јазол се додава на неговото десно дете, инаку се додава на неговото лево дете. Ако јазолот *j* е *NULL*, тогаш се креира новиот јазол и му се доделува на *j*

Brisi(o, &T) - функција која ги брише сите јазли со ознака *o* од стеблото *T*. Оваа функција најчесто не се користи.

Inorder(T) **–** функција за *Inorder* процесирање на стеблото *T*

Vrednost(j) - функција која ја враќа вредноста на јазолот *j*

Oznaka(j) - функција која ја враќа ознаката на јазолот *j*

2.2.4 Имплементација на АПТ БИНАРНО ПРЕБАРУВАЧКО СТЕБЛО

Имплементацијата на АПТ бинарно пребарувачко стебло е многу слична како имплементацијата на обичното бинарно стебло. Разликата се состои во имплементацијата на операциите за додавање и бришење на нови јазли во стеблото, затоа што овие операции не се однесуваат на експлицитно референцирани јазли, туку зависат од вредноста на ознаките. Поради гореспоменатата природност на имплементацијата на бинарни стебла со помош на покажувачи и оваа имплементација е решена со динамички структури.

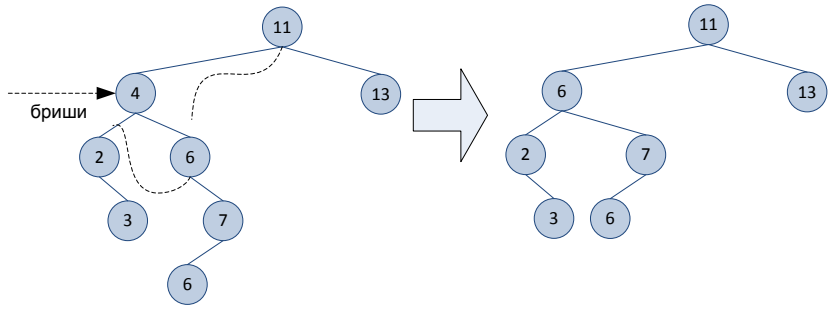
Без губење на општоста во продолжение е дадена имплементацијата на овој АПТ во C++, но со неозначена структура т.е. без користење на ознаки. Подредувањето во оваа имплементација е во однос на подредувањето дефинирано во рамките на доменот на податокот од секој јазол, т.е. *IzbranTip.*

```
#include <iostream>
#include <stdio.h>

typedef int IzbranTip;

typedef struct Jazol_Oznaka {
              IzbranTip podatok;
              Jazol_Oznaka *lDete, *dDete;
              } BSsteblo;

void Init(BSsteblo *BS) {
       BS = NULL;
}

BSsteblo *KreirajJazol(IzbranTip d) {
       BSsteblo *q = (BSsteblo*)malloc(sizeof(BSsteblo));
       q->podatok = d;
       q->lDete = NULL; q->dDete = NULL;
       return q;
}

BSsteblo *KreirajSteblo(IzbranTip d) {
       return KreirajJazol(d);
}

BSsteblo *LevoDete(BSsteblo *BS) {
       return BS->lDete;
}

BSsteblo *DesnoDete(BSsteblo *BS) {
       return BS->dDete;
```

```
}

void DodadiJazol(BSsteblo *jazol, BSsteblo *BS) {
	if ((BS==NULL)||(jazol==NULL)) return;
	if (BS->podatok > jazol->podatok)
		if (BS->lDete==NULL)
			BS->lDete = jazol;
		else
			DodadiJazol(jazol, BS->lDete);
	else
		if (BS->dDete==NULL)
			BS->dDete = jazol;
		else
			DodadiJazol(jazol, BS->dDete);
}

void Dodadi(IzbranTip d, BSsteblo *BS) {
	DodadiJazol(KreirajJazol(d),BS);
}

IzbranTip Vrednost(BSsteblo *BS) {
	if (BS==NULL) return NULL;
	return BS->podatok;
}

void BrisiPreOrder(IzbranTip d, BSsteblo *BS, BSsteblo *b) {
	if (BS==NULL) return;
	if (BS->podatok!=d) Dodadi(BS->podatok, b);
	BrisiPreOrder(d, LevoDete(BS), b);
	BrisiPreOrder(d, DesnoDete(BS), b);
}


BSsteblo *Brisi(IzbranTip d, BSsteblo * BS) {
	if (BS==NULL) return NULL;
	BSsteblo *r = BS;
	while (r->lDete!=NULL)
		r = r->lDete;
	r = KreirajSteblo(r->podatok);
	BrisiPreOrder(d, BS, r);
	return r->dDete;
}

void InOrder(BSsteblo *BS) {
	if (BS==NULL) return;
	InOrder(LevoDete(BS));
	std::cout<<Vrednost(BS)<<" ";
	InOrder(DesnoDete(BS));
}
```

Како што може да се види во самата имплементација најкомплицирана операција е операцијата за бришење на јазли од стеблото. Со оглед на тоа што дефиницијата на оваа операција во овој АПТ се однесува на бришење на сите елементи со дадена ознака тогаш операцијата мора да се имплементира со помош на некој од алгоритмите за еднократно поминување на сите јазли во стеблото. Во оваа имплементација се користи алгоритмот *PREORDER* за да се зачува структурата, така што се креира ново стебло во кое по овој редослед се додаваат сите елементи различни од оние со вредноста која се брише.

Доколку е потребно да се брише точно еден јазол од бинарното пребарувачко стебло, тогаш неговото лево дете се додава како јазол на неговото десно подстебло (преку функцијата *DodadiJazol()*), со што се зачувува дефиницијата на бинарно пребарувачко стебло (Слика 2.14).



Слика 2.14. Бришење на единечен јазол во АПТ бинарно пребарувачко стебло

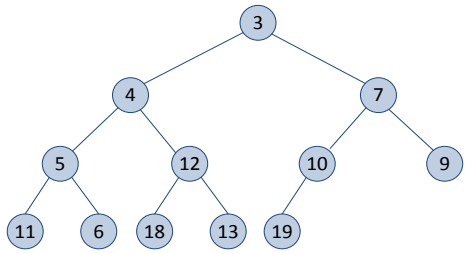
2.2.5 Апстрактен податочен тип КУП (Heap)

Куп или heap-стебло е уште еден специјален вид на бинарно стебло кој се користи за имплементација на други сложени АПТ-ови и алгоритми. Потполно бинарно стебло претставува куп (heap) ако неговите јазли се означени со податоци од некој тип во кој има дефинирано подредување и каде за секој јазол важи дека неговата ознака е помала од ознаките на сите негови деца, во однос на дефинираното подредување (Слика 2.15).

Својствата и методите на купот се:

IzbranTip - било кој тип на податоци

oznaka - било кој тип наменет за означување на јазлите



Слика 2.15. Апстрактен податочен тип КУП (Heap)

jazol - јазол на стеблото составен од дел со информации од типот *IzbranTip* и два покажувачи кон други јазли, едниот за левото, другиот за десното дете. Секој јазол може да биде означен со ознака од типот *oznaka*.

Kup - податок од овој тип е стебло чии јазли се податоци од типот *jazol* кои се различни меѓу себе и не се *NULL* и важат својствата за овој АПТ.

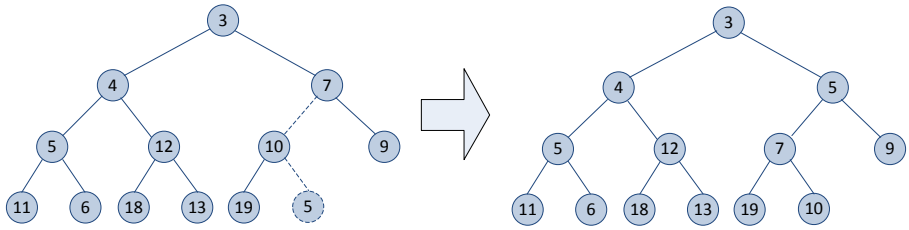
Init(&K) - функција која го претвора стеблото *K* во празен куп.

Prazen(K) - функција која враќа информација ако и само ако *K* е празно стебло

Dodadi(o, d, &K) - функција која додава нов јазол со вредност *d* на купот *K* и го означува новиот јазол со *o*, со задржување на својствата на купот

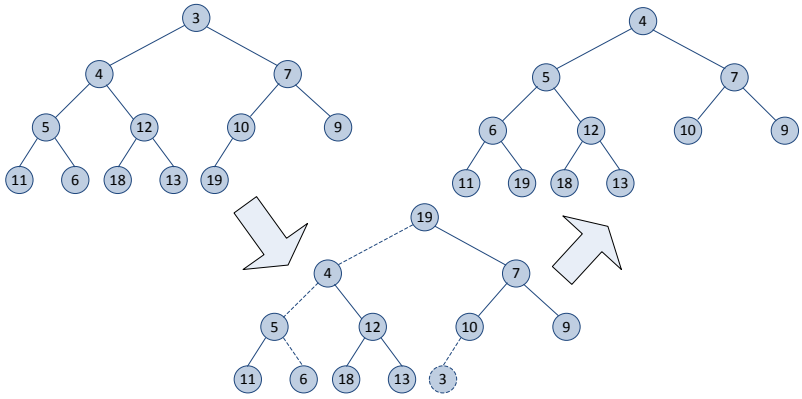
IzbrisiNajmal(&K) - функција која го брише најмалиот елемент од купот *K* со задржување на својствата на купот и ја враќа неговата вредност

Операцијата *Dodadi()* се имплементира на тој начин што се наоѓа првата слободна локација во структурата и на тоа место се креира јазол со вредност *d* и ознака *o*. Меѓутоа, при самото додавање на нов јазол, постои можност својствата на купот да се нарушат т.е. вредноста на новиот јазол да биде помала од вредноста на неговиот родител. За таа цел вредностите на новиот јазол и неговиот родител се заменуваат, па потоа се менуваат вредностите на родителот на новиот јазол и неговиот родител и така натаму, сè додека има потреба. Секвенцата на замени оди најмногу до коренот, со оглед на тоа што во коренот секогаш е сместен најмалиот елемент. Ова може да се види и на Слика 2.16.



Слика 2.16. Операција Dodadi() кај АПТ КУП

Слична е ситуацијата и со операцијата *BrisiNajmal().* Оваа операција всушност секогаш ја брише коренската вредност, меѓутоа бришењето на коренот води до распаѓање на стеблото, па операцијата се извршува на тој начин што коренот и последниот јазол во структурата ги менуваат своите вредности, а потоа последната вредност се брише. Меѓутоа, како и кај додавањето, постои можност да се нарушат својствата на структурата, па затоа мора да се спроведе процес за „средување“ на стеблото.



Слика 2.17. Операција IzbrisiNajmal() кај АПТ КУП

Имено, се врши замена на вредностите на коренот и неговото дете кое има помала вредност, потоа се врши замена на вредностите на ова дете со неговото дете кое има помала вредност и така додека не се исполнат својствата на купот, а најмногу до дното на стеблото. Ова е прикажано на Слика 2.17.

2.2.6 Имплементација на АПТ КУП

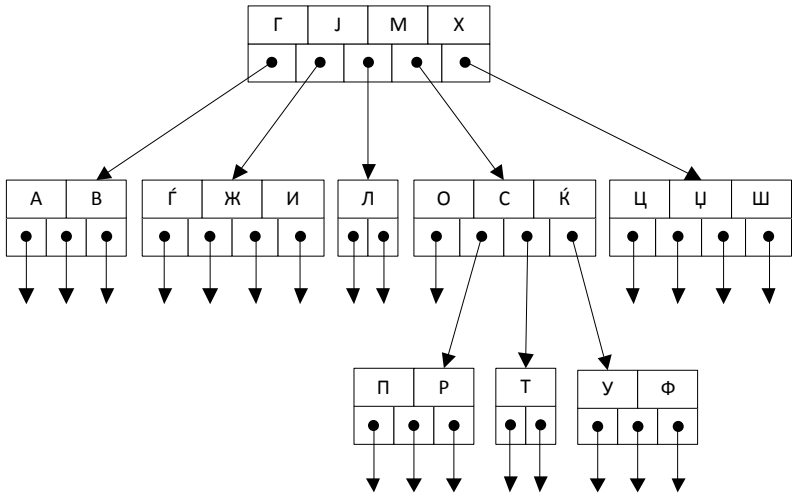
Имплементацијата на овој АПТ е наједноставна со статичка структура за имплементација на бинарно стебло. Приодот е сличен, па без губење на општоста, во оваа имплементација не се користат ознаките, туку подредувањето е според дефинираното подредување на доменот *IzbranTip*. Имплементацијата е дадена во C++, па треба да се забележи дека во C++ *NULL* е дефинирано како 0, што имплицира елиминација на 0 како вредност на некој елемент.

```
#include <iostream>
#include <stdio.h>

#define MAX_DOLZINA 127 //od tip (2^k) - 1

typedef int IzbranTip;

typedef struct {
	IzbranTip podatok[MAX_DOLZINA+1];
	} Kup;

void Init(Kup &K) {
	for(int i=1;i<=MAX_DOLZINA;i++)
		K.podatok[i] = NULL;
}

bool Prazen(Kup &K) {
	return (K.podatok[1]==NULL);
}

void ZameniVrednosti(int j, int k, Kup &K) {
	IzbranTip t = K.podatok[j];
	K.podatok[j] = K.podatok[k];
	K.podatok[k] = t;
}


void Dodadi(IzbranTip d, Kup &K) {
	if (d==0) return; // Poradi NULL = 0
	int poz = 1; //Prvo slobodno mesto
	while (K.podatok[poz]!=NULL) poz++;
	if (poz>MAX_DOLZINA) {
		perror("Kupot e poln!!!");
		return;
	}
	K.podatok[poz] = d;
	while((K.podatok[poz]<K.podatok[poz/2])&&(poz>1))
	{
		ZameniVrednosti(poz, poz/2, K);
		poz = poz/2;
	};
}

IzbranTip IzbrisiNajmal(Kup &K) {
	int poz = 1;
	IzbranTip rez = K.podatok[1];
	while (K.podatok[poz]!=NULL) poz++;
	poz--; //Posleden element
	ZameniVrednosti(1, poz, K);
	K.podatok[poz]=NULL;
```

```
        poz = 1;
        int cel = -1;
        while (cel!=0)
        {
                cel = 0;
                if (K.podatok[poz*2+1]==NULL)
                {
                        if (K.podatok[poz*2]!=NULL)
                                cel = poz*2;
                }
                else
                        if (K.podatok[poz*2]==NULL)
                                cel = poz*2+1;
                        else
                                if (K.podatok[poz*2]>K.podatok[poz*2+1])
                                        cel = poz*2+1;
                                else
                                        cel = poz*2;
                if (K.podatok[poz]<K.podatok[cel]) cel=0;
                if (cel!=0)
                {
                        ZameniVrednosti(poz, cel, K);
                        poz = cel;
                };
        }
        return rez;
}
```

2.3 B-стебло (B-Tree)

B-стебло е специјализирано повеќекратно стебло кое се користи кај дисковите. Во B-стебло секој јазол може да содржи многу клучеви, што значи дека и подстеблата на секој јазол може да бидат многу. B-стеблото е дизајнирано за да се разграна на многу подстебла и да содржи многу клучеви во секој јазол што го прави самото стебло релативно „ниско“. Ваквиот дизајн овозможува со мал број на прочитани јазли да се дојде до бараната информација. Целта на овие стебла е да се обезбеди брз пристап до податоците што доведува до малку исчитани податоци од дисковите. Треба да се забележи дека за ваквите структури се претпоставува дека дискот овозможува брзо вчитување на податоците во големи јазли кои содржат многу клучеви.



Слика 2.18. Пример за B-стебло

На Слика 2.18 е прикажано повеќекратно подредено стебло со степен 5. Како што може да се забележи во секој јазол има низа од клучеви кои може да претставуваат и цели записи каде секој клуч се однесува на еден од низата покажувачи. Алтернативен начин за имплементација на јазолот, кој обично се користи кога податочните записи се големи, е тој да претставува низа од записи во која секој запис покрај податочниот дел содржи и покажувач кон поврзаниот јазол.

Кај B-стеблата низата од клучеви е подредена во растечки редослед и за секој јазол *X* од стеблото кој има *n* клучеви *k*-тото подстебло на *X* ги содржи сите клучеви поголеми од вредноста на *k*-тиот клуч на *X,* но помали од вредноста на (*k+1)*-иот клуч на *X*, освен во случајот кога *k=n* при што важи само дека *k*-тото подстебло на *X* ги содржи сите клучеви поголеми од вредноста на *k*-тиот клуч на *X.*

B-стеблото и неговите варијанти се користат кај големите комерцијални бази на податоци за обезбедување на брз пристап до самите податоци. Всушност, може да се каже и дека тие се стандардната податочна организација кога се работи за апликации кои бараат вметнување, бришење и пребарување по клучеви. Варијантата наречена *B+* стебло е највообичаената, а *B** стебло е многу слично на *B+* стебло, само што кај него се внимава јазлите да бидат барем 2/3 наполнети во однос на нивниот капацитет.

Кај *B+* стеблото записите се сместуваат само во листовите, а внатрешните јазли содржат само клучеви. Овие клучеви се користат за насочување на пребарувањето кон бараниот резултат. Во случај целниот

клуч да е помал од клучот во внатрешниот јазол, пребарувањето продолжува на неговата лева страна, инаку пребарувањето продолжува на неговата десна страна. Кај *B+* стеблото листовите се исто така поврзани со што се овозможува поминување на листовите во растечки редослед, што значи дека пребарувањето би можело да се извршува и само по дното од стеблото.

Кога *B+* стебло се имплементира кај дисковите, листовите содржат (клуч, покажувач) пароци каде покажувачот покажува на податокот поврзан со клучот. Ова овозможува датотеката со податоци да биде сместена одделно од самото *B+* стебло, кое функционира како индекс за сортирање на податоците во датотеката. Ова практично го претставува и начинот како *B+* стеблата се користат кај базите на податоци, така што оваа шема се користи за секој од индексите кои се потребни за одредена табела, што значи дека за секој атрибут од една табела може да се дефинира индекс.

2.3.1 Апстрактен податочен тип B-СТЕБЛО

За да може оваа структура да се претвори во апстрактен податочен тип, слично како и кај другите поими, потребно е да се дефинираат и операции кои се извршуваат.

IzbranTip - било кој тип на податоци

kluc - било кој подреден тип

ElementNaJazol - јазол составен од дел со информации од типот *IzbranTip*, индекс од типот *kluc* и покажувач кон подстебло.

Bsteblo - податок од овој тип е стебло чии јазли се низи од податоци од типот *ElementNaJazol* кои се различни меѓу себе и не се *NULL*.

Init(k, d, &BT) - функција која го претвора стеблото *BT* во стебло кое се состои само од коренски јазол чија низа од клучеви содржи само еден елемент од типот *ElementNaJazol* на кој му се доделува вредност *d* и индексот *k* кој може да биде и *NULL.*

Dodadi(k, d, &BT) - функција која додава елемент со вредност *d* и клуч *k* во стеблото *BT*

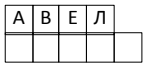
Brisi(k, &BT) - функција која го брише елементот со клуч *k* од стеблото *BT*.

Vrednost(k, BT) - функција која ја враќа вредноста од типот *IzbranTip* на елементот од стеблото *BT* со клуч *k*

Prazno(BT) – функција која испитува дали *B*-стеблото *BT* е празно

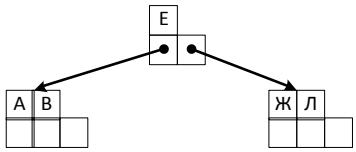
Операцијата *Dodadi()* за додавање на елемент во *B*-стеблото се реализира на тој начин што прво се пребарува елементот во *B*-стеблото. Доколку елементот веќе се наоѓа во *B*-стеблото, тогаш нема никаква промена. Во спротивно, пребарувањето завршува во лист од *B*-стеблото. Ако во листот има доволно простор за сместување на елементот во него, тогаш тоа и се прави, но притоа треба да се води сметка за редоследот на клучевите во овој јазол, т.е. клучевите од листот кои се поголеми од новиот клуч се поместуваат надесно, за да се ослободи простор за новиот клуч. Во случај листот да е полн, тогаш листот се дели на два дела со што се добиваат два листа во кои новиот клуч се додава во соодветниот лист. При оваа поделба, се избира средниот елемент од сите елементи од двата јазли и се сместува во нивниот родител. Секако, можно е и овој јазол да е полн, при што се применува истата постапка т.е. негово разбивање на два дела и избор на среден елемент кој оди едно место погоре во хиерархијата. Оваа постапка може да се повтори најмногу до самиот корен на *B*-стеблото, при што ако се дели и коренот, се добива нов коренски јазол на целото *B*-стебло со што се зголемува и самата длабочина на стеблото.

Оваа постапка е илустрирана преку следниот пример. На располагање е *B*-стебло со степен 5. Потребно е низата В Л Е А Ж Д И Н К Ѓ Ќ Ј Р Х Г М О У Ф Љ да се смести во структурата. На Слика 2.19 е прикажан јазол во кој се сместени првите четири елементи. Очигледно е дека за првите четири елементи од низата постапката доаѓа до коренот и веднаш наоѓа место во него.



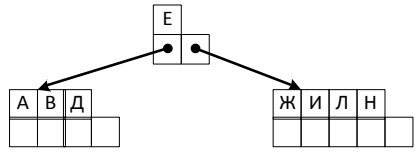
Слика 2.19. Операција за додавање на елемент во B-стебло (пример, слика 1/8)

При додавање на елементот Ж, во овој јазол нема место за него, па јазолот се разбива на два дела, при што средниот елемент се сместува во нов коренски јазол (Слика 2.20).



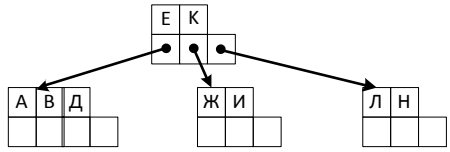
Слика 2.20. Операција за додавање на елемент во B-стебло (пример, слика 2/8)

Понатаму, вметнувањето на Д, И и Н не бара никакви преструктуирања, туку само се пополнуваат слободни места (Слика 2.21).



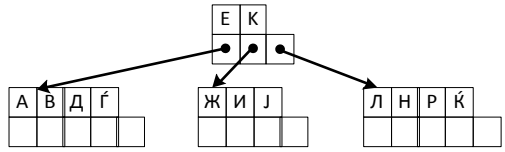
Слика 2.21. Операција за додавање на елемент во B-стебло (пример, слика 3/8)

Елементот К бара разбивање на јазол, при што во коренскиот јазол се сместува средниот клуч од разбиениот јазол (Слика 2.22).



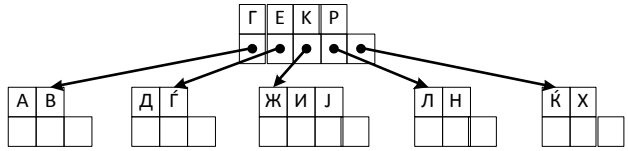
Слика 2.22. Операција за додавање на елемент во B-стебло (пример, слика 4/8)

Следните четири елементи (Ѓ, Ќ, Ј, Р) како и претходно само пополнуваат празни места во јазлите и не бараат разбивања на јазли (Слика 2.23).



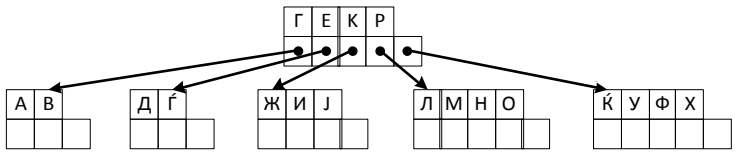
Слика 2.23. Операција за додавање на елемент во B-стебло (пример, слика 5/8)

Елементите Х и Г се два елементи кои бараат разбивање на двата полни листови од претходниот чекор. Нивното додавање во *B*-стеблото резултира со полнење на капацитетот на коренскиот јазол (Слика 2.24).



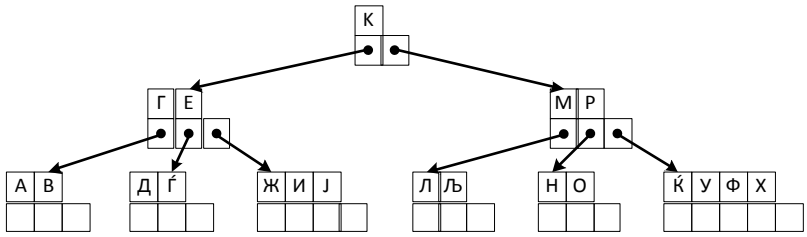
Слика 2.24. Операција за додавање на елемент во B-стебло (пример, слика 6/8)

Следните четири елементи М, О, У и Ф само се додаваат во празните места (Слика 2.25).



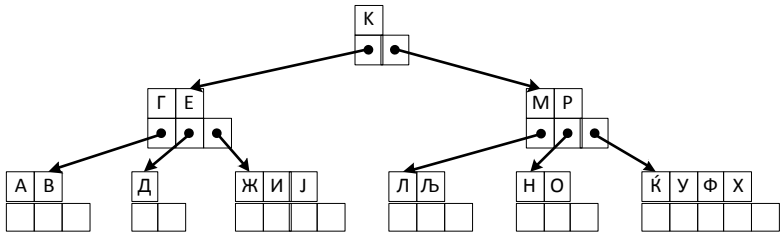
Слика 2.25. Операција за додавање на елемент во B-стебло (пример, слика 7/8)

Конечно, додавањето на последниот елемент Љ придонесува до две разбивања т.е. разбивање на лист и разбивање на коренот. Ова е прикажано на Слика 2.26.



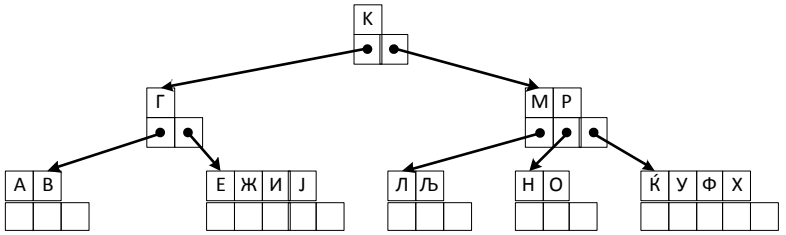
Слика 2.26. Операција за додавање на елемент во B-стебло (пример, слика 8/8)

Операцијата *Brisi()* може да се однесува на два различни случаи, кога се брише елемент од лист или кога се брише елемент од внатрешен јазол. Тривијален случај е кога се брише елемент од лист при што елементот што се брише не е единствен елемент во листот. Оваа постапка се извршува со едноставно отстранување на елементот од јазолот. Ваков пример е случајот кога во *B*-стеблото од Слика 2.26 треба да се отстрани на пример елементот Ѓ, при што се добива *B*-стеблото на Слика 2.27.



Слика 2.27. Пример за бришење на елемент во B-стебло (случај 1)

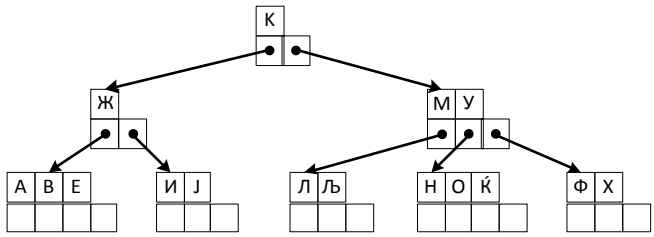
Доколку елементот што се брише е единствениот елемент во јазолот (елементот Д на претходната слика), тогаш неговиот родителски елемент во структурата (во случајот Е) се спушта едно ниво подолу и се групира со сите негови деца во ист јазол (Слика 2.28).



Слика 2.28. Пример за бришење на елемент во B-стебло (случај 2)

Групирањето на елементите во ист јазол, слично како кај операцијата за додавање на елемент во *B*-стебло може да резултира со преполнет јазол кој треба да се разбие, па и во овој случај разбивањето се решава на истиот начин т.е. со издвојување на средниот елемент и негово сместување во родителскиот јазол и разделување на јазолот на два дела, едниот со помалите елементи од средниот елемент, другиот со поголемите елементи од средниот елемент.

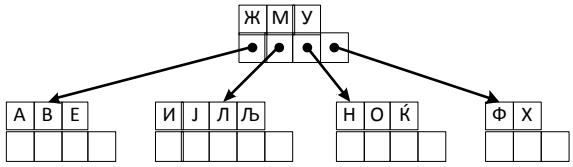
Бришењето на елемент од внатрешен јазол се реализира на тој начин што елементите кои се наоѓаат во децата-јазли на елементот кој се брише се спојуваат во еден јазол кој најверојатно ќе треба да се разбие. При ова разбивање, средниот елемент го заменува елементот кој се брише. На пример, во структурата од претходната слика ако треба да се избришат елементите Г и Р се добива *B*-стеблото од Слика 2.29.



Слика 2.29. Пример за бришење на елемент во B-стебло (случај 3)

Постапката е иста и кога се брише (не/единствен) елемент од коренскиот јазол. Ваков случај е илустриран на Слика 2.30 и го опфаќа бришењето на коренскиот елемент од претходната слика. Но, очигледно тука

настанува проблем. Имено, спојувањето на двата јазли од второто ниво (Ж и М-У) ќе резултира со преклопување на последното дете на левиот јазол (И-Ј) и првото дете од вториот јазол (Л-Љ). Решението на овој проблем е спојување на овие два јазли во еден, и негово разбивање ако има потреба за тоа. Спојувањето во примерот е тривијално, затоа што во примерот се спојуваат листови, па затоа оваа постапка треба да биде рекурзивна за општиот случај т.е. спојувањето на јазли да го спушти до листовите.



Слика 2.30. Пример за бришење на елемент во B-стебло (случај 4)

2.3.2 Имплементација на АПТ B-СТЕБЛО

3. Множества

3.1 Општо множество (Set)

Кај многу алгоритми како математички модел се јавува множеството кое претставува колекција од податоци од ист тип кои се нарекуваат елементи на множеството, при што во едно множество не може да има два елементи со исти вредности. Меѓу елементите не постои никаков линеарно или хиерархиско уредување, туку сите елементи се рамноправни меѓу себе и меѓу нив нема никаква врска. Зависно од типот на податокот кој го содржат елементите се дефинира имплицитна релација на тотално уредување, т.е. може да се користи терминот помал т.е. поголем елемент.

Множествата имаат широка примена во реалниот свет. Конкретно кај базите на податоци, резултатите од пребарувањето даваат множества. На пример, нека А е множеството на сите студенти од факултетот за информатика, а В е множеството на сите студенти од Куманово. Тогаш, студентите од факултетот за информатика и студентите од Куманово со еден израз ги поистоветуваме како унија на А и В т.е. A U B, или студентите од Куманово кои студираат информатика би биле пресекот на А и В т.е. A ∩ B, а студентите од Куманово кои не студираат информатика се разликата на В и А, т.е. B \ A.

3.1.1 Апстрактен податочен тип МНОЖЕСТВО

Множеството се дефинира како апстрактен податочен тип со вообичаените математички операции.

IzbranTip - било кој тип на податоци

Mnozestvo – конечно множество различни елементи од типот *IzbranTip*

Init(&M) - функција која го претвора множеството *M* во празно множество

Prazno(M) – булова функција која кажува дали множеството *M* е празно или не

Dodadi(x, &M) – функција која го додава елементот *x* во множеството *M*, при што ако $x \in M$, *M* останува непроменето

Izbrisi(x, &M) – функција која го брише елементот *x* од множеството *M*, при што ако $x \notin M$, *M* останува непроменето

Pripadja(x, M) – булова функција која враќа дали $x \in M$

MIN(M) и ***MAX(M)*** – функции кои го враќаат минималниот т.е. максималниот елемент од множеството *M* според дефинираното подредување

Podmnozestvo(A, B) – булова функција која испитува дали $A \subseteq B$

Unija(A, B, &C), Presek(A, B, &C), Razlika(A, B, &C) - го менуваат *C* при што $C = A \cup B$, т.е. $C = A \cap B$ т.е. $C = A \setminus B$

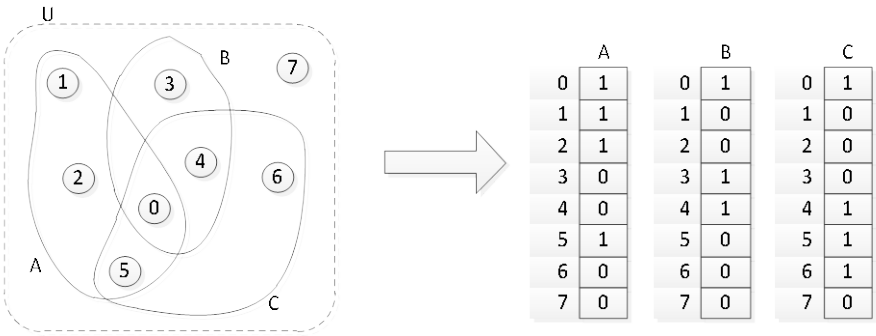
Komplement(A, &B) – го менува *B* при што во него го сместува комплементот на *A* во однос на универзалното множество (кај имплементации каде е можно)

Вака дефинираниот апстрактен податочен тип множество е навистина комплициран за имплементација, затоа што ако се постигне ефикасно извршување на едни операции, тогаш други операции ќе се извршуваат споро, и обратно, па според тоа зависно според намената во оваа глава се објаснети неколку типа на множества кои сите се базираат на општото множество, објаснето во ова поглавје каде е направена имплементација на АПТ множество со цел: ефикасно извршување на операциите за унија, пресек, разлика, комплемент и подмножество, што се користи кај алгоритмите во кои нема многу додавања и бришења на елементи во множествата.

3.1.2 Имплементација на АПТ МНОЖЕСТВО

Имплементација со низа од битови

Оваа имплементација се реализира на тој начин што множеството се претставува како низа од битови каде секој елемент се наоѓа на соодветна позиција во полето. Соодветниот бит во низата го означува присуството на елементот во множеството на кое се однесува низата. Карактеристично за оваа имплементација на множество е тоа што на овој начин може да се имплементира операцијата за наоѓање на комплемент, затоа што самата метода бара дефинирање на универзален домен на елементи врз основа на кој може да се пресмета комплемент на множество.



Слика 3.1. Имплементација на АПТ множество со низа од битови

Покрај дефинираните операции на АПТ множество, кај оваа имплементација потребни се уште неколку карактеристични операции. Во примерот е претпоставено дека избраниот тип за податоци е цел број, па со методата *SetDomen()* во универзалниот домен се сместуваат целите броеви од 0 до MAX_DOLZINA. Функцијата *IndeksNaElement(x)* е функција која го враќа индексот на битот од низата кој соодветствува на елементот *x*, а *PripadjaNaDomen(x)* е булова функција која испитува дали елементот *x* припаѓа на доменот. *PomaloOd(x,y)* испитува дали *x* е помало од *y,* во однос на уредувањето на елементите (за целите броеви е *x<y*).

Овие функции се користат кај дефинираните операции за АПТ множество.

Оваа имплементација станува неупотреблива како расте MAX_DOLZINA.

```
#include <iostream>
#include <stdio.h>

#define MAX_DOLZINA 30

typedef int IzbranTip;

IzbranTip Domen[MAX_DOLZINA];

void SetDomen() {
	for(int i=0;i<MAX_DOLZINA;i++)
		Domen[i] = i;
}

int IndeksNaElement(IzbranTip x) {
	int br = 0;
	while (br<MAX_DOLZINA)
	{
		if (Domen[br] == x) return br;
		br++;
```

```
        }
        return -1;
}

bool PripadjaNaDomen(IzbranTip x) {
        return (IndeksNaElement(x)>=0);
}

bool Pomalo(IzbranTip x, IzbranTip y) {
        return x<y;
}

typedef bool Mnozestvo[MAX_DOLZINA];

void Init(Mnozestvo &A) {
        for(int i=0; i<MAX_DOLZINA; i++)
                A[i] = false;
}

bool Prazno(Mnozestvo A) {
        for(int i=0;i<MAX_DOLZINA;i++)
                if (A[i]) return false;
        return true;
}

void Dodadi(IzbranTip x, Mnozestvo &A) {
        if (!PripadjaNaDomen(x))
                perror("Elementot ne se naodja vo domenot...");
        else A[IndeksNaElement(x)] = true;
}

void Izbrisi(IzbranTip x, Mnozestvo &A) {
        if (!PripadjaNaDomen(x))
                perror("Elementot ne se naodja vo domenot...");
        else
                A[IndeksNaElement(x)] = false;
}


bool Pripadja(IzbranTip x, Mnozestvo &A) {
        if (!PripadjaNaDomen(x))
                perror("Elementot ne se naodja vo domenot...");
        else return A[IndeksNaElement(x)];
}
IzbranTip MIN(Mnozestvo A) {
        if (Prazno(A)) return NULL;
        IzbranTip min = NULL;
        for(int i=0;i<MAX_DOLZINA;i++)
                if (A[i])
                        if (min==NULL)
```

```
                        min = Domen[i];
                else
                        if (Pomalo(A[i],min)) min=A[i];
        return min;
}

IzbranTip MAX(Mnozestvo A) {
        if (Prazno(A)) return NULL;
        IzbranTip max = NULL;
        for(int i=0;i<MAX_DOLZINA;i++)
                if (A[i])
                        if (max==NULL) max = Domen[i];
                        else
                                if (Pomalo(max,A[i])) max=A[i];
        return max;
}

bool Podmnozestvo(Mnozestvo A, Mnozestvo B) {
        for(int i=0;i<MAX_DOLZINA;i++)
                if (A[i] && (!B[i]))
                        return false;
        return true;
}

void Unija(Mnozestvo A, Mnozestvo B, Mnozestvo &C) {
        for(int i=0;i<MAX_DOLZINA;i++)
                C[i] = A[i] || B[i];
}

void Presek(Mnozestvo A, Mnozestvo B, Mnozestvo &C) {
        for(int i=0;i<MAX_DOLZINA;i++)
                C[i] = A[i] && B[i];
}

void Razlika(Mnozestvo A, Mnozestvo B, Mnozestvo &C) {
        for(int i=0;i<MAX_DOLZINA;i++)
                C[i] = A[i] && !B[i];
}

void Komplement(Mnozestvo A, Mnozestvo &B) {
        for(int i=0;i<MAX_DOLZINA;i++)
                B[i] = !A[i];
}
```

Имплементација со сортирана листа

Множеството се претставува како поврзана листа. Со оглед на тоа што со операциите унија, пресек, разлика итн., должината на листата може многу

да варира, па листата се дефинира дека е сортирана во однос на дефинираното подредување. На овој начин бришењето и додавањето на елементи во множеството се поистоветува со бришење и додавање на елементи во листата, но со позиција утврдена со вредноста која ја носи елементот, па затоа и процедурите за бришење и додавање не бараат позиција на елемент како параметар, што е случај кај имплементацијата на листите.

Оваа имплементација е претставена со следниот код во C++.

```
#include <iostream>
#include <stdio.h>

typedef int IzbranTip;

typedef struct CLista_Oznaka {
              CLista_Oznaka *sleden;
              IzbranTip podatok;
              } SetLista;

void Init(SetLista *SL) {
       SL->sleden = NULL;
}

bool Prazno(SetLista *SL) {
       return (SL->sleden==NULL);
}

IzbranTip MAX(SetLista *SL) {
       if (Prazno(SL))
              return NULL;
       else   if (SL->sleden->sleden==NULL)
                     return SL->sleden->podatok;
              else
                     return MAX(SL->sleden);
}

IzbranTip MIN(SetLista *SL) {
       if (Prazno(SL)) return NULL;
       else return SL->sleden->podatok;
}



void Dodadi(IzbranTip x, SetLista *SL) {
       if (Prazno(SL))
       {
              SL->sleden = (SetLista*)malloc(sizeof(SetLista));
              SL->sleden->podatok = x;
```

```
                SL->sleden->sleden = NULL;
        }
        else
                if (SL->sleden->podatok<x) Dodadi(x, SL->sleden);
                else
                 if (SL->sleden->podatok!=x)
                        {
                        SetLista *temp = SL->sleden;
                        SL->sleden =
                                (SetLista*)malloc(sizeof(SetLista));
                        SL->sleden->podatok = x;
                        SL->sleden->sleden = temp;
                        }
}

void Izbrisi(IzbranTip x, SetLista *SL) {
        if (Prazno(SL)) return;
        else
        {
                if (SL->sleden->podatok<x) Izbrisi(x, SL->sleden);
                else if (SL->sleden->podatok==x)
                                {
                                        SetLista * temp = SL->sleden;
                                        SL->sleden = temp->sleden;
                                        free(temp);
                                }
        }
}

bool Pripadja(IzbranTip x, SetLista *SL) {
        if (Prazno(SL)) return false;
        else
        if (SL->sleden->podatok==x)
                return true;
        else
                return Pripadja(x, SL->sleden);
}

bool Podmnozestvo(SetLista * A, SetLista * B) {
        if (Prazno(A))
                return true;
        else
                return Pripadja(A->sleden->podatok,B)
                                && Podmnozestvo(A->sleden,B);
}
void Unija(SetLista * A, SetLista * B, SetLista * C) {
        if (!Prazno(A))
        {
                Dodadi(A->sleden->podatok,C);
                Unija(A->sleden,B,C);
```

```
	}
	else
	if (!Prazno(B))
		{
			Dodadi(B->sleden->podatok,C);
			Unija(A,B->sleden,C);
		}
}

void Presek(SetLista * A, SetLista * B, SetLista * C) {
	if (!Prazno(A))
	{
		if (Pripadja(A->sleden->podatok,B))
			Dodadi(A->sleden->podatok,C);
		Presek(A->sleden,B,C);
	}
}

void Razlika(SetLista * A, SetLista * B, SetLista * C) {
	if (!Prazno(A))
	{
		if (!Pripadja(A->sleden->podatok,B))
			Dodadi(A->sleden->podatok,C);
		Razlika(A->sleden,B,C);
	}
}
```

Како што може да се види, поголемиот дел од операциите се лесно изводливи со помош на рекурзија, на пример:

Unija(A, B, &C) = Unija(A\{x},B\{x},&C U {x}), каде $x \in A$ или $x \in B$ и $A \neq \emptyset$ и $B \neq \emptyset$

Presek(A, B, &C) = Presek(A\{x},B,&C U {x}), каде $x \in A$ и $x \in B$ и $A \neq \emptyset$

Razlika(A, B, &C) = Razlika(A\{x},B,&C U {x}), каде $x \in A$ и $x \notin B$ и $A \neq \emptyset$

3.2 Речник (Dictionary)

Кај множествата често не се бараат сложените операции како што се пресекот, унијата, разликата, подмножество или комплемент, што значи дека кај ваквите проблеми се чува состојбата на едно множество и повремено се додаваат или отфрлуваат елементи од него, како и испитување дали некој елемент припаѓа на множеството или не. Ваквото множество се нарекува речник (dictionary) кој има широка примена во реалниот свет на пример

листа на правилно напишани зборови од некој јазик, што го чини правописот на тој јазик, па за да се види дали некој збор е правилно напишан се проверува дали зборот постои во речникот или не, или пак кај компјутер со повеќе корисници кога се најавува корисник се испитува дали неговото корисничко име постои или не во речникот кој го чини директориумот на корисници за тој компјутер.

3.2.1 Апстрактен податочен тип РЕЧНИК

Апстрактниот податочен тип речник се дефинира слично како и АПТ множество, со тоа што листата на операции во однос на АПТ множество е редуцирана, па овој АПТ би ја имал следната форма:

IzbranTip - било кој тип на податоци

Recnik – конечно множество различни елементи од типот *IzbranTip*

Init(&D) - функција која го претвора речникот *D* во празен

Prazen(D) – булова функција која враќа информација дали речникот *D* е празен

Dodadi(x, &D) – функција која го додава елементот *x* во речникот *D*, при што, како кај множеството, ако $x \in D$, *D* останува непроменет

Izbrisi(x, &D) – функција која го брише елементот *x* од множеството *D*, при што ако $x \notin D$, *D* останува непроменет

Pripadja(x, D) – булова функција која враќа дали $x \in D$

3.2.2 Имплементација на АПТ РЕЧНИК

Речникот може да се имплементира на потполно ист начин како што се имплементира и АПТ множество. Кај првиот случај, со помош на бит-вектор постојат истите правила како кај множеството т.е. операциите *Dodadi(), Izbrisi()* и *Pripadja()* се извршуваат во *O(1)* време, а имплементацијата станува неупотреблива ако доменот на типот *IzbranTip* е премногу голем.

Од друга страна, речниците се подразбира дека се големи, па од големо значење е оптимизацијата на алгоритмите кои се користат при негова имплементација. Ако се врши имплементација на АПТ речник со помош на сортирана листа со помош на статичка структура во однос на подредувањето кое е дефинирано во рамките на *IzbranTip*, се врши подобрување на алгоритмот за пребарување кој се користи кај операциите *Dodadi()*, *Izbrisi()* и

Pripadja() т.е. се применува некој понапреден алгоритам за пребарување во однос на секвенцијалниот, како на пример бинарното пребарување (опишано во делот за алгоритми).

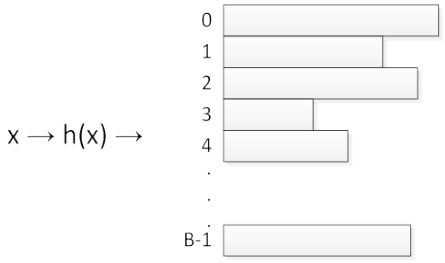
Она што е интересно за овој АПТ се два дополнителни пристапи за негова имплементација и тоа со користење на т.н. хеш таблици и со користење на бинарно пребарувачко стебло.

Имплементација со помош на хеш табела

При имплементацијата на овој АПТ со помош на бит-вектор на секоја можна вредност од *IzbranTip* се придружува еден посебен бит во меморијата, што значи дека се воспоставува биекција од доменот *IzbranTip* во кодоменот "меморија". Оваа имплементација од една страна може да се третира како идеална, со оглед на тоа што операциите Dodadi(), Izbrisi() и Pripadja() се извршуваат за време O(1), меѓутоа, оваа имплементација бара многу меморија т.е. користи мемориски ресурси за сите можни елементи од *IzbranTip* без разлика дали тие се користат или не. Затоа оваа имплементација во пракса е неупотреблива, со оглед на тоа што бара голема меморија која воопшто не се користи. Како компромис пресликувањето се менува од биекција во сурјекција. Ова значи дека се врши пресликување на помал дел од меморијата, при што треба да се води сметка дека повеќе различни вредности може да се пресликаат на иста адреса.

Хеш (hash) функција е алгоритам која пресликува големи количества податоци во помали, наречени клучеви. Хеш функциите најчесто се користат за забрзување на пребарување низ табели или споредба на податоци, при што таа треба да биде таква што ако иста функција се повика двапати над ист влез, резултатот од неа треба да биде ист.

Кај оваа имплементација хеш функцијата *h()* е програмска рутина која сурјективно го пресликува доменот на *IzbranTip* во множеството цели броеви меѓу *0* и *B-1* каде *B* е целобројна константа. Хеш табела е поле од *B* ќелии кои се нарекуваат слотови (slot, bucket). Индексите на слотовите се *0, 1, ... B-1*. Имплементацијата на речникот со помош на хеш табела се базира на тоа што елементот *x* се сместува во слотот со индекс *h(x)*, а потоа го бараме во истиот слот со истата функција.



Слика 3.2. Hash функција и hash табела

Оваа идеја за да функционира важно е функцијата *h(x)* на точно одреден начин да ги распоредува вредностите од множеството *IzbranTip* низ слотовите *0, 1, ..., B-1*. На пример ако *IzbranTip* е множество на сите стрингови, тогаш може да се дефинира хеш функција како должина на стрингот или како реден број на првата буква од стрингот во азбуката, на следниот начин:

```
int h(string x) {
    return x.length();
}
```

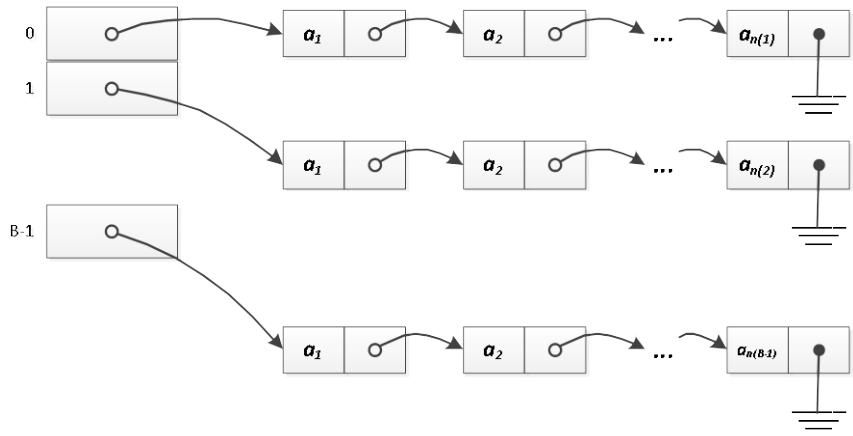
```
int h(string x) {
    return (int)toupper(x[0])-64;
}
```

Од примерот се гледа дека може да има различни имплементации со помош на хеш табела кои се разликуваат по градбата на слотовите, а се однесуваат на решавањето на колизиите. Тоа се отворено и затворено хеширање кои ќе бидат прикажани како две различни имплементации на овој АПТ.

Важно е да се напомене дека кај двата типа на хеширање, табелите треба да бидат добро димензионирани во однос на речникот кој се сместува во самата структура. Ако *n* е број на елементи во речникот, препорачливо е $n \leq 2 \times B$ кај отвореното хеширање, т.е. $n \leq 0.9 \times B$ кај затвореното хеширање. Ако ова биде исполнето, тогаш очекувано е да било која од операциите *Dodadi(), Izbrisi()* или *Pripadja()* да се изврши во само неколку читања од табелата. Во случај кога табелата би се преполнила, тогаш истата треба да се препише во нова поголема.

Отворено хеширање

Кај отвореното хеширање слотовите се градени како поврзана листа. Секогаш кога треба да се додаде нов елемент *x* во *i*-тиот слот, *i*-тата листа се продолжува за еден јазол, што придонесува капацитетот на еден слот да е неограничен и променлив (Слика 3.3).



Слика 3.3. Имплементација на АПТ речник со отворена hash табела

Имплементацијата на АПТ речник со отворена хеш табела во C++ е прикажана со следниот код:

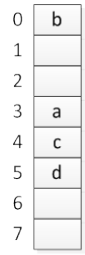
```
#include <iostream>
#include <stdio.h>

#define BrojSlotovi 10

typedef int IzbranTip;

typedef struct Kelija_Oznaka {
              Kelija_Oznaka *sleden;
              IzbranTip podatok;
              } Kelija;
typedef struct {
              Kelija * Slot[BrojSlotovi];
              } Recnik;

int h(IzbranTip x) {
              return x % BrojSlotovi;
              }

void IscistiSlot(Kelija *K) {
       if (K!=NULL)
       {
              IscistiSlot(K->sleden);
              free(K);
       };
}

void Init(Recnik *D) {
       for(int i = 0 ; i<BrojSlotovi; i++)
       {
```

```
                D->Slot[i] = (Kelija*)malloc(sizeof(Kelija));
                D->Slot[i]->podatok=NULL;
                D->Slot[i]->sleden=NULL;
        }
}

bool Prazen(Recnik *D) {
        for(int i = 0 ; i<BrojSlotovi; i++)
                if (D->Slot[i]->sleden!=NULL)
                        return false;
        return true;
}

bool PripadjaVoSlot(IzbranTip x, Kelija *K) {
        if (K==NULL) return false;
        else if (K->podatok==x)
                                return true;
                        else
                                return PripadjaVoSlot(x, K->sleden);
}

bool Pripadja(IzbranTip x, Recnik *D) {
        return PripadjaVoSlot(x, D->Slot[h(x)]);
}

void DodadiVoSlot(IzbranTip x, Kelija * K) {
        Kelija * Nov = (Kelija*)malloc(sizeof(Kelija));
        Nov->podatok = x;
        Nov->sleden = K->sleden;
        K->sleden = Nov;
}
void Dodadi(IzbranTip x, Recnik *D) {
        if (!Pripadja(x, D))
                DodadiVoSlot(x, D->Slot[h(x)]);
}

void IzbrisiOdSlot(IzbranTip x, Kelija * K) {
        if (K==NULL) return;
        Kelija * temp = K;
        while (temp->sleden!=NULL)
                if (temp->sleden->podatok==x)
                        {
                                Kelija * temp2 = temp->sleden;
                                temp->sleden = temp->sleden->sleden;
                                free(temp2);
                        }
                else temp = temp->sleden;
}

void Izbrisi(IzbranTip x, Recnik *D) {
```

```
        IzbrisiOdSlot(x,D->Slot[h(x)]);
}
```

Затворено хеширање

Кај затвореното хеширање, слотовите имаат фиксен капацитет. Поради поедноставување, а без губење на општоста, нека еден слот собира само еден елемент. Ова значи дека хеш табелата е поле од ќелии од типот *IzbranTip*, па ако треба да се додаде елемент *x*, а ќелијата *h(x)* е веќе полна, тогаш се бара алтернативна локација преку функции од типот *hh(1,x), hh(2, x) …*, сè додека не се најде празна локација во некој слот. Често пати се користи линеарно хеширање што значи *hh(i, x) = (h(x)+i) % B.* На Слика 3.4 е прикажана ситуација кога *B=8*, а речникот се гради со додавање на елементи *a, b, c, d* по ред, при што *h(x)* враќа *3, 0, 4, 3*, соодветно.



Слика 3.4. Пример за затворено хеширање

Кај оваа имплементација треба да се внимава на препознавањето на празните слотови, при што се користи специјална резервирана вредност која би била различна од било која вредност која би можела да се додаде во табелата. Оваа вредност се нарекува *ПРАЗНА (EMPTY)* вредност, па барањето на некој елемент *x* во табелата би се одвивало со пребарување на слотовите *h(x), hh(1, x), hh(2, x) …* сè додека не се најде *x* или *ПРАЗНА.* Оваа постапка за пребарување е правилна ако нема бришења на елементи во речникот, но ако треба да се воведе и ваков механизам, тогаш се воведува уште една резервирана вредност *ИЗБРИШАНА (DELETED)* која би означувала дека елементот е избришан, па истата ќелија би можела да се искористи како да е празна.

Во продолжение е имплементацијата во C++ на АПТ речник со помош на затворена хеш табела:

```
#include <iostream>
#include <stdio.h>
#define BrojSlotovi 5
#define KapacitetNaSlot 5
```

```
typedef int IzbranTip;
#define PRAZNA -1
#define IZBRISANA -2

typedef IzbranTip Recnik[BrojSlotovi][KapacitetNaSlot];

int h(IzbranTip x) {
    return x % BrojSlotovi;
}

int hh(IzbranTip k, IzbranTip x) {
    return (h(x)+ k) % BrojSlotovi;
}

void Init(Recnik &D) {
     for(int i = 0 ; i<BrojSlotovi; i++)
             for(int j=0;j<KapacitetNaSlot;j++)
                     D[i][j] = PRAZNA;
}

bool Prazen(Recnik &D) {
     for(int i = 0 ; i<BrojSlotovi; i++)
             for(int j=0;j<KapacitetNaSlot;j++)
                     if ((D[i][j] != PRAZNA) &&
                                  (D[i][j]!=IZBRISANA))
                                         return false;
     return true;
}
bool PripadjaVoSlot(IzbranTip x, int s, Recnik D) {
     for(int i=0;i<KapacitetNaSlot;i++)
             if (D[s][i]==x) return true;
     return false;
}

bool Pripadja(IzbranTip x, Recnik D) {
       bool prip = PripadjaVoSlot(x, h(x), D);
       int iteracija=0;
       while ((!prip)&&(iteracija<BrojSlotovi))
       {
              iteracija++;
              prip = PripadjaVoSlot(x, hh(iteracija, x), D);
       }
       return prip;
}

bool DodadiVoSlot(IzbranTip x, int s, Recnik &D) {
     for(int i=0;i<KapacitetNaSlot;i++)
             if ((D[s][i] == PRAZNA)||(D[s][i] == IZBRISANA))
                {
                  D[s][i] = x;
```

```
                return true;
              }
     return false;
}

void Dodadi(IzbranTip x, Recnik &D) {
     if (Pripadja(x, D)) return;
     int SlotIndeks = h(x);
     bool dodadena = DodadiVoSlot(x,h(x),D);
     int iteracija = 0;
     while ((!dodadena)&&(iteracija<BrojSlotovi))
     {
           iteracija++;
           dodadena = DodadiVoSlot(x, hh(iteracija,x), D);
     }
     if ((!dodadena)&&(iteracija==BrojSlotovi))
        perror("Nema mesto vo recnikot...");
}

bool IzbrisiOdSlot(IzbranTip x, int s, Recnik &D) {
       for(int i=0;i<KapacitetNaSlot;i++)
             if (D[s][i]==x)
             {
                    D[s][i] = IZBRISANA;
                    return true;
             }
       return false;
}
void Izbrisi(IzbranTip x, Recnik &D) {
       if (!IzbrisiOdSlot(x, h(x), D))
       {
             int iteracija = 1;
             while(!IzbrisiOdSlot(x, hh(iteracija, x), D) &&
                          (iteracija<=BrojSlotovi))
                                iteracija++;
       }
}
```

Имплементација со помош на бинарно пребарувачко стебло

Идејата на имплементацијата е следна: речникот се прикажува со бинарно пребарувачко стебло, при што на секој елемент од речникот одговара точно по еден јазол од стеблото и обратно. Потребни се само мали модификации на имплементацијата на АПТ бинарно пребарувачко стебло, така што во оваа имплементација се прикажани само модифицираните функции.

Во имплементацијата на бинарно пребарувачко стебло се додава операцијата *Pripadja()*, која е потребна за имплементацијата на АПТ речник.

```
bool Pripadja(IzbranTip d, BSsteblo *BS) {
       if (BS==NULL)
              return false;
       if (d==Vrednost(BS))
              return true;
       if (d<Vrednost(BS))
              return Pripadja(d, LevoDete(BS));
       else
              return Pripadja(d, DesnoDete(BS));
}
```

Операцијата *Dodadi()* има само мала модификација, а тоа е што треба да спречува дупликати во самото стебло. Модифицираната варијанта на операцијата е прикажана со следниот код:

```
void Dodadi(IzbranTip d, BSsteblo *BS) {
       if (!Pripadja(d, BS))
              DodadiJazol(KreirajJazol(d),BS);
}
```

Операцијата *Izbrisi()* од АПТ бинарно пребарувачко стебло функционира и за овој АПТ, но со оглед на тоа што кај речникот нема дупликати на вредности, применлива е и варијантата од Слика 2.14.

Секоја од функциите *Pripadja(), Dodadi()* и *Izbrisi()* поминува само по еден пат од коренот на стеблото до некој јазол, па затоа времето на извршување на операциите е ограничено со висината на стеблото.

Ако речникот има *n* елементи, тогаш висината на стеблото варира меѓу $(\log_2(n+1))-1$ и n-1 при што граничните случаи се однесуваат на полно т.е. косо стебло. Според ова, и времето на извршување варира меѓу O(log n) и O(n).

Оваа имплементација е само една од многуте слични имплементации, каде речникот се прикажува со помош на некаква „стеблеста“ структура.

3.3 Приоритетен ред (Priority queue)

Кај некои алгоритми се појавува множество на чии елементи се преидружуваат цели или реални броеви кои влијаат на редоследот на процесирање на самите елементи. Овие придружени вредности се нарекуваат приоритети, а вакво множество на елементи со дефинирани

операции за додавање на нов елемент и бришење на елементот со најмал приоритет, се нарекува приоритетен ред (Priority Queue).

Во реалниот свет постојат бројни примери каде може да се сретнат приоритетните редови, на пример чекална во ординација е типичен пример за приоритетен ред. Во чекалната доаѓаат пациенти и од таму одат на преглед кај лекарот, но не е на ред првиот пациент кој дошол во чекалната, туку оној чија состојба е најтешка. Во компјутерската техника се користат приоритетните редови кај процесите кои чекаат на извршување т.е. на секој процес му е доделен некаков приоритет, па процесорот го извршува процесот со највисокиот (најмалата вредност) приоритет.

3.3.1 Апстрактен податочен тип ПРИОРИТЕТЕН РЕД

Изложените проблеми се сведуваат на поедноставен проблем за додавање на елементи во множество и бришење на најмалиот елемент од него. Имено, наместо оригиналните елементи *x*, се разгледуваат подредени парови *(p(x), x)* каде *p(x)* го означува приоритетот на елементот *x*, и за нив се дефинира следното подредување:

$$(p(x_1), x_1) \leq (p(x_2), x_2),$$

ако $[\ p(x_1) < p(x_2)\ ]$ или $[\ p(x_1) = p(x_2)$ и $x_1 \leq x_2\ ]$

Според ова, добиена е дефиниција на подредување според кое може да се подредат елементите на множеството, па може да се дефинира АПТ приоритетен ред со следните карактеристики.

IzbranTip - било кој тип на податоци

PrioritetenRed – конечно множество различни елементи од типот *IzbranTip*

Init(&PQ) - функција која го претвора приоритетниот ред *PQ* во празен

Prazen(D) – булова функција која враќа информација дали *PQ* е празен

Dodadi(x, &PQ) – функција која го додава елементот *x* во приоритетниот ред *PQ*, при што ако $x \in D$, *D* останува непроменет

IzbrisiNajmal(&PQ) – функција која го брише најмалиот елемент од приоритетниот ред *PQ*. Не е дефинирана ако *PQ* е празен.

Овој АПТ може да се разгледува и како рестрикција на АПТ множество, со тоа што операцијата *IzbrisiNajmal()* е комбинација од операциите *MIN()* и *Izbrisi().*

3.3.2 Имплементација на АПТ ПРИОРИТЕТЕН РЕД

Имплементација со помош на сортирана поврзана листа

Приоритетен ред може да се имплементира со помош на листа, и тоа сортирана поврзана при што подредувањето не се врши по вредноста туку по подредените парови *(p(x), x)*, при што *IzbrisiNajmal()* го наоѓа првиот елемент и го брише, и тоа за време *O(1)*. Од друга страна *Dodadi()* треба да ја најде соодветната локација, па во просек ќе прочита половина од листата додека да го најде вистинското место, па ова би се извршило за време *O(n)*, каде *n* е бројот на елементи во листата. Имплементацијата не е дадена со код, бидејќи се работи за иста имплементација со сортирана листа.

Имплементација со помош на бинарно пребарувачко стебло

Бинарното пребарувачко стебло, како што беше објаснето и во неговата имплементација, е структура која најчесто се користи за имплементација на други посложени АПТ-ови, како и специфични алгоритми. Оваа структура беше искористена кај имплементацијата на АПТ речник, така што потполно иста имплементација се применува и за АПТ приоритетен ред, бидејќи се искористуваат истите функции *Dodadi(), IzbrisiNajmal()* и *Init()*, а *Prazen()* е тривијална, затоа кодот за имплементација не е даден за овој АПТ.

Имплементација со помош на куп

Приоритетниот ред може да се имплементира и со куп. На секој елемент од редот одговара точно еден јазол од купот и обратно, при што сите јазли од купот ќе имаат различни ознаки, иако тоа не е услов во дефиницијата на купот.

Кај оваа имплементација, функциите *Init()* и *Prazen()* се тривијални, а функциите *Dodadi()* и *IzbrisiNajmal()* поминуваат само еден пат во потполно бинарно стебло, при што во најлош случај нивното време на извршување е *O(log n)* каде *n* е бројот на јазли во стеблото т.е. бројот на елементите во приоритетниот ред. За разлика од претходната имплементација, оваа имплементација има подобро време на извршување за операциите, со оглед на тоа што во претходната имплементација логаритамско време на извршување се добива само во просечни случаи. За разлика од оваа имплементација, имплементацијата со бинарно пребарувачко стебло има

предност во имплементација на дополнителни операции како што се *Pripadja(), Izbrisi(), Min()* или *Max().*

3.4 Пресликување (Mapping)

Пресликувањето е еден од најосновните поими во математиката. Практично и не постои дел од математиката во кој не се вклучени пресликувањата, а не постои ни математичка теорија која на некој начин не ги вклучува пресликувањата. Нека се дадени две непразни множества A и B. Ако постои барем еден елемент $a \in A$ на кој според некое правило му е придружен елемент $b \in B$, тогаш се вели дека постои придружување (кореспонденција) од множеството A во B. Ако на елементот $a \in A$ му е придружен елементот $b \in B$ се бележи $a \rightarrow b$ при што a е оригинал, а b е слика. За овој пример множеството A претставува дефиниционо множество или домен, а множеството B е множество слики или кодомен. Ако придружувањето е f, а придружените елементи се означени како подреден пар од видот $a \rightarrow b \equiv (a,b)$, тогаш придружувањето може да се дефинира како

$$f \subseteq A \times B = \{ (x,y) \mid x \in A, y \in B \} \text{ и } f(a) = b$$

При придружувањето на елементи од двете множества не се наведени никакви услови кои придружувањето мора да ги задоволи, туку само се формираат парови од елементи при кои едниот елемент му е придружен на другиот, па за да едно придружување од множество A во множество B биде пресликување мора да го задоволува својството $\forall x \in A, \exists! y \in B$ при што $x \rightarrow y$, што значи дека за секој оригинал од доменот кај едно пресликување постои единствена слика во множеството слики.

Нека F е дадено пресликување, така што ако $(x, y) \in F$, се вели дека $F(x)$ е дефинирано и се означува со $y=F(x)$, инаку се вели дека $F(x)$ не е дефинирано.

Како што беше напоменато, пресликувањето има широка примена, но не само во математиката. Телефонскиот именик е класичен пример за пресликување од доменот на луѓе во кодоменот од телефонски броеви. Имињата може да се додаваат и бришат, а броевите може да се менуваат. Слична е ситуацијата со речници со странски зборови, индекс на поими, содржина на книга и сл. Вработен во една компанија е опишан со запис од податоци така што ако множеството од тие записи е кодомен, може да се дефинира пресликување од множеството матични броеви во кодоменот од записи.

3.4.1 Апстрактен податочен тип ПРЕСЛИКУВАЊЕ

Пресликувањето може да се дефинира како АПТ со имплементација на стандардните математички операции кои важат за него.

domen – било кој тип кој ќе претставува домен на пресликувањето

kodomen – било кој тип кој ќе претставува кодомен на пресликувањето

Preslikuvanje – множество на парови од податоци каде $a \rightarrow b$ и a е вредност од доменот на пресликувањето, а b е вредност од кодоменот на пресликувањето

Init(&P) – функција која го претвора пресликувањето *P* во празно

Preslikaj(d, k, &P) – функција која го дефинира $k = P(d)$, без оглед на тоа дали било дефинирано или не

IzbrisiSlika(d, &P) – функција која ја брише дефиницијата *P(d)*, без оглед на тоа дали било дефинирано или не

Presmetaj(d, P, &k) – булова функција која враќа информација дали постои дефиниција на *P(d)*, и притоа доколку постои дефиницијата на *k* му ја придружува вредноста *P(d)*

3.4.2 Имплементација на АПТ ПРЕСЛИКУВАЊЕ

АПТ пресликување се имплементира со истите податочни структури како што се имплементира речникот, т.е. поле, хеш табела или бинарно пребарувачко стебло, само што според дефиницијата на самото пресликување во множеството се сместуваат подредени парови од облик *(x, P(x))* при што дефиницијата на подредувањето на овие парови се поистоветува со подредувањето на првата компонента *x*, затоа што според дефиницијата на пресликувањето, не може да се повтори иста вредност за првата компонента т.е. еден елемент од доменот има само една слика во кодоменот, па операциите *Init(), Preslikaj(), IzbrisiSlika(), Presmetaj()* од овој АПТ, се имплементираат скоро исто како *Init(), Dodadi(), Izbrisi()* и *Pripadja()* од АПТ речник.

Пример: Нека е дадено пресликувањето P од *int* во *double* со табелата

k	3	5	8	9	11	17	21
P(k)	3.13	5.14	8.16	9.16	11.07	17.22	21.25

Тогаш пресликувањето P се претставува како множество подредени парови P={(3,3.13),(5,5.15),(8,8.16),(9,9.16),(11,11.07),(17,17.22),(21,21.25)}.

Имплементација со помош на хеш табела

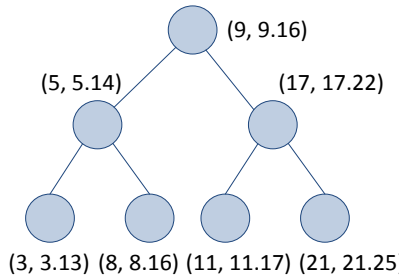
Множеството парови се поистоветува со речник и се имплементира на начини кои беа претставени за АПТ речник. Во продолжение е дадена дефиницијата на *IzbranTip* и имплементацијата на хеш функцијата *h(x)* во C++ кои соодветствуваат на АПТ пресликување, така што само со нивна замена во било која од двете имплементации на АПТ речник со хеш табели и со тривијални измени во врска со управувањето со новиот *IzbranTip* и називите на новите операции се добива имплементацијата на АПТ пресликување.

```
typedef struct {
              int original;
              double slika;
              } IzbranTip;

int h(IzbranTip x) {
              return x.original % BrojSlotovi;
              }
```

Имплементација со помош на бинарно пребарувачко стебло

Множеството парови може да се прикаже и со помош на бинарно пребарувачко стебло. Подредените парови *(x, y)* се сместуваат во јазлите од стеблото, а разгранувањето во стеблото се одредува само врз основа на првата компонента *x* затоа што таа се појавува само еднаш, од дефиницијата на пресликувањето т.е. важи подредувањето:

$(x_1, y_1) < (x_2, y_2)$ акко $x_1 < x_2$

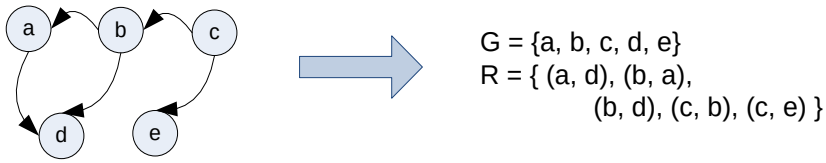


Слика 3.5. Пример за АПТ пресликување во бинарно пребарувачко стебло

3.5 Релација (Relation)

Поопшт модел за придружување на податоци од пресликувањето е поимот за релација. Бинарна релација R е множество подредени парови од обликот (x, y), каде вредноста $x \in X$ и $y \in Y$, каде X и Y се домени на релацијата. Ако релацијата R го содржи парот (x, y) тогаш се вели дека x е во релација R со y и се бележи со xRy, во спротивно се вели дека x не е во релација R со y и се бележи со $x\not Ry$.

Релациите се насекаде во реалниот свет, филмови и кина, лекови и дијагнози, производи и производители итн., а релациите може да се искористат и за дефинирање и на други математички поими, како што е на пример графот. Графовите може да се дефинираат со помош на релација каде домените се исто множество т.е. множеството јазли, а пресликувањето би ги содржело ребрата во графот (Слика 3.6)



Слика 3.6. Релација во функција на граф

3.5.1 Апстрактен податочен тип РЕЛАЦИЈА

Релацијата е најдобро да се дефинира како АПТ со имплементација на стандардните математички операции кои важат за неа.

domen1, domen2 – било кои типови кои ќе претставуваат домени на релацијата

Relacija – множество на парови од податоци кои претставуваат бинарна релација со прв домен *domen1* и втор домен *domen2*

Init(&R) – функција која ја претвора релацијата R во празна

Povrzi(d_1, d_2, &R) – функција која ја поставува d_1Rd_2 без разлика дали била поставена или не

Odvrzi(d_1, d_2, &R) – функција која ја поставува $d_1\not Rd_2$ без разлика дали била поставена или не

Presmetaj1(x, R, &M) – функција која во M го враќа множеството $\{ y \mid xRy \}$

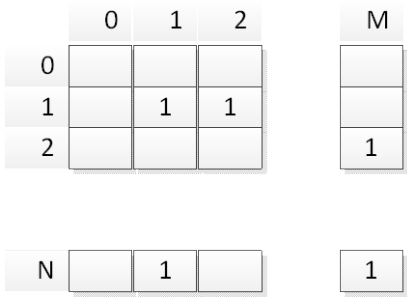
Presmetaj2(y, R, &M) – функција која во *M* го враќа множеството *{ x | xRy }*

3.5.2 Имплементација на АПТ РЕЛАЦИЈА

АПТ релација се имплементира со истите податочни структури кои се користат за имплементација на АПТ речник. Слично како кај пресликувањето, и за релацијата во структурата се сместуваат парови, а операциите *Init(), Povrzi()* и *Odvrzi()* од АПТ релација се имплементираат аналогно како и *Init(), Dodadi()* и *Izbrisi()* од АПТ речник, соодветно. За поефикасно извршување на операциите *Presmetaj1()* и *Presmetaj2()*, потребни се мали измени и дополнувања на структурата, како што беше случај и со АПТ пресликување во однос на АПТ речник.

Имплементација со помош на бит-матрица

Имплементацијата на АПТ релација е наједноставна со помош на статичка структура бит-матрица. Пристапот е сличен со имплементацијата на АПТ множество со помош на бит-вектор, со тоа што во овој случај се работи за дводимензионално поле. Имено, на секој елемент од првиот домен му се доделува редица во матрицата, а на секој елемент од вториот домен му се доделува колона од матрицата.



Слика 3.7. Имплементација на АПТ релација со бит-матрица

Вредностите на матрицата се битови кои кажуваат кои податоци од првиот домен се поврзани со кои податоци од вториот домен.

Во оваа имплементација, без губење на општоста, се зема дека првиот домен е множеството {0, 1, …, N}, а вториот домен е множеството {0, 1, …, M}, па имплементацијата може да се прикаже на Слика 3.7.

Следува кодот во C++ кој ја претставува оваа имплементација.

```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 30
int Domen1[MAX_DOLZINA], Domen2[MAX_DOLZINA];

void PostaviDomeni()
{
	for(int i=0;i<MAX_DOLZINA;i++)
	{
		Domen1[i] = i;
		Domen2[i] = i;
	};
}

typedef bool Relacija[MAX_DOLZINA][MAX_DOLZINA];

typedef bool Mnozestvo[MAX_DOLZINA];

void Init(Relacija &R) {
	for(int i=0; i<MAX_DOLZINA; i++)
		for(int j=0; j<MAX_DOLZINA; j++)
			R[i][j] = false;
}

void Povrzi(int d1, int d2, Relacija &R) {
	R[d1][d2] = true;
}

void Odvrzi(int d1, int d2, Relacija &R) {
	R[d1][d2] = false;
}

void Presmetaj1(int d1, Relacija R, Mnozestvo &M) {
	for(int j=0; j<MAX_DOLZINA; j++)
		M[j] = R[d1, j];
}

void Presmetaj2(int d2, Relacija R, Mnozestvo &M) {
	for(int i=0; i<MAX_DOLZINA; i++)
		M[i] = R[i, d2];
}
```

II дел

АЛГОРИТМИ

4. Нумерички алгоритми

Ова поглавје третира развој на алгоритми за различни проблеми и пресметковни задачи во врска со броеви.

Прв проблем кој е разработен е Фибоначиевата низа, т.е. низата на броеви каде секој број е збир на неговите два претходници, а првите два броја се 0 и 1. Оваа низа покрај низата на степени на бројот 2, е една од најпопуларните низи од броеви и детално е проучувана и применувана во многу полиња: биологија, демографија, уметност, архитектура, музика итн.

Понатаму се опфатени главните теми на ова поглавје, факторите и простите броеви. Разбивањето на фактори значи даден цел број да се претстави како производ од прости броеви (фактори) и тоа воопшто не е тривијален проблем, а испитување на прост број е испитување дали цел број е прост, што исто така не е тривијален проблем.

4.1 Низата на Фибоначи

Во XV-ти век италијанскиот математичар Leonardo Fibonacci го увидел потенцијалот на децималниот систем и истражувал во неговото унапредување и пропагирање, така што тој ја поставил низата

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

каде секој елемент е збир на неговите два претходници, позната како Фибоначиева низа.

Формално, низата на Фибоначи може да се претстави на следниот начин:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

Фибоначиевата низа заедно со низата степени на 2 се најистражуваните низи на броеви. Всушност, Фибоначиевите броеви растат толку брзо, како и степените на 2, F_{30} е преку милион, а F_{100} е долг 21 цифра, па може да се покаже дека $F_n \approx 2^{0{,}694n}$. Сепак, прецизната вредност на Фибоначиевите броеви со индекс 100, или 200 е прашање чиј одговор може

да се добие само со помош на алгоритам кој ја дава вредноста на *n*-тиот Фибоначиев број.

4.1.1 Експоненцијален алгоритам

Секако, првата идеја до која може да се дојде за решавање на овој проблем е да се искористи рекурзивната дефиниција на F_n. Во продолжение е дадена имплементација на овој алгоритам во C++:

```
int fibonacci(int n)
{
	if (n==0) return 0;
	else if (n==1) return 1;
	else
		return fibonacci(n-1)+fibonacci(n-2);
}
```

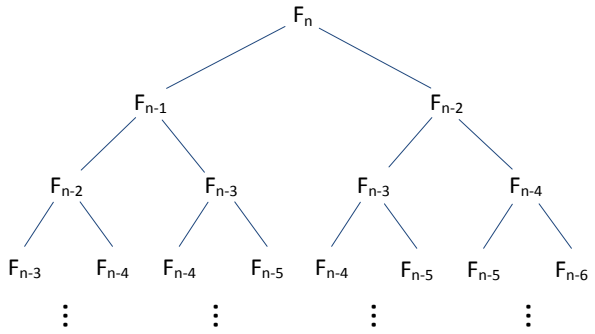
Со сигурност може да се каже дека овој алгоритам работни коректно, затоа што тој е практично Фибоначиевата дефиниција на F_n, меѓутоа прашање е колку е ефикасен овој алгоритам. Нека *T(n)* е бројот на чекори потребни за извршување на *fibonacci(n)*, тогаш можеме да кажеме дека за првите броеви од низата т.е. за *n<2*, процедурата завршува скоро веднаш, додека за поголемите вредности на *n*, се повикуваат два рекурзивни повици на процедурата *T(n-1)* и *T(n-2)*, па ако се додадат дополнителните 3 команди (двете споредби, со 0 и 1, и собирањето на добиените вредности од рекурзивните повици) комплетната дефиниција на *T(n)* би била:

$$T(n)=\begin{cases}1 & n=0\\ 2 & n=1\\ T(n-1)+T(n-2)+3 & n>1\end{cases}$$

Според оваа дефиниција лесно може да се види дека $T(n)\geq F_n$, што е лоша вест затоа што времето на извршување расте, како што растат и самите броеви од низата. Експоненцијално извршување на *T(n)* значи дека алгоритмот воопшто не е практичен, освен за мали вредности на *n*. Ако на пример треба да се пресмета F_{200} тогаш *fibonacci(200)* би се извршувала во барем 2^{138} чекори ($T(200)\geq F_{200}\geq 2^{0.694*200}\geq 2^{138}$). Фибоначиевата низа има примена кај алатките за тестирање на перформансите на процесорите, па во моментот процесорот со најдобри резултати на овој тест (процесорот на Intel од фамилијата i7 модел Extreme 980X со 12 [6HT] јадра кој работи на 4611MHz) има постигнато индекс од 187753,88 милиони инструкции во секунда (според MaxxPI[2] Professional). Ова значи дека пресметувањето на F_{200}

на овој процесор би се извршило за приближно 2^{101} секунди што би се извршувало многу подолг период отколку што научниците очекуваат да живее сонцето.

Сепак, технологијата напредува брзо, така што според некои истражувања брзините на процесорите се дуплираат на секои 18 месеци, така што со ваков раст може да се очекува *fibonacci(n)* да работи многу побрзо на машините од наредната година (или приближно 1,6 пати побрзо). Ако се спореди времето на извршување на *fibonacci(n)* со $2^{0,694n} \approx 1,6^n$, може да се заклучи дека времето за пресметување на F_{n+1} е подолго за 1,6 пати од времето за пресметување на F_n. Па според сето ова може да се заклучи дека ако оваа година во нормално време може да се пресмета F_n, тогаш идната година во нормално време ќе може да се пресмета F_{n+1}, што значи откривање на само еден број од низата на Фибоначи секоја година, а со заслуга на експоненцијалното време.



Слика 4.1. Извршување на рекурзивни повици за пресметување на Фибоначиев број

Ако се постави прашањето зошто овој алгоритам е спор и ако се разгледа начинот на неговото извршување на Слика 4.1, на која е прикажано рекурзивното извршување на само еден повик на *fibonacci(n)*, лесно може да се забележи дека многу пресметки се повторуваат.

4.1.2 Полиномен алгоритам

Проблемите кои произлегуваат од претходниот алгоритам водат кон изнаоѓање на решение во кое ќе се искористуваат меѓурезултатите F_0, F_1, … F_{n-1} секаде каде тоа ќе биде потребно. Следниот код го претставува решението на проблемот:

```
int fibonacci2(int n)
{
       if (n==0) return 0;
```

```
        else if (n==1) return 1;
        else
        {
                int pretprethodnik = 0;
                int prethodnik = 1;
                int pom;
                for (int k = 2; k<n;k++)
                {
                        pom = prethodnik;
                        prethodnik = prethodnik + pretprethodnik;
                        pretprethodnik = pom;
                }
                return prethodnik + pretprethodnik;
        }
}
```

За коректноста на алгоритмот нема потреба да се дискутира, затоа што и овој алгоритам ја користи самата дефиниција на F_n. Се поставува прашањето колку долго се извршува овој алгоритам.

Како што може да се види и во самиот код, внатрешноста на јазолот која е составена од 3 елементарни чекори се извршува *n-1* пати, што значи дека *fibonacci2(n)* линеарно зависи од *n*, па во овој случај од експоненцијално време на извршување, добиено е полиномно време на извршување, што претставува огромен напредок во однос на подобрувањето на операцијата, така што со овој алгоритам не само што е достижно пресметувањето на F_{200} , но и на F_{200000}.

Овој алгоритам има и своја подобрена варијанта од аспект на бројот на операции кои се потребни за негово извршување, но подобрувањето бара мемориски простор пропорционален на *n*. Имено, се работи за алгоритам кој во јазолот ги избегнува двете операции за доделување и користи само една операција за собирање.

```
int fibonacci3(int n)
{
        int F[100];
        F[0] = 0;
        F[1] = 1;
        for(int k = 2; k<=n; k++)
                F[k] = F[k-1] + F[k-2];
        return F[n];
}
```

Како што може да се види и во кодот, оваа варијанта во однос на претходната е побрза со оглед на тоа што во јазолот има само една операција,

но затоа пак е ограничена на резервираниот простор за полето во кое се сместуваат броевите.

Гледајќи ги сите имплементации на алгоритмот за низата на Фибоначи, собирањето е земано како основен чекор што е случај за мали броеви т.е. броеви кои ги покрива рангот на типот (во овие примери int, т.е. 2^{32}). Бројот на Фибоначи со реден број n е составен од приближно $0,694n$ бита, така што рангот на било кој тип многу брзо може да биде надминат. Од друга страна аритметичките операции на доволно големи броеви не можат да се извршат во само еден чекор, па затоа мора да се изврши корекција на претходното излагање.

Собирањето на броеви може да се изврши и со процедурата за собирање на цели броеви која се учи уште во основното образование, така што оваа процедура собира само една цифра со еден чекор. Според ова, времето на извршување за сите алгоритми би се зголемило онолку пати колку што има цифри т.е. n, па времето на извршување на *fibonacci(n)* би било nF_n, а времето на извршување на *fibonacci2(n)* би било n^2, што и не влијае многу на перформансите на вториот алгоритам во однос на првиот.

4.2 Основни аритметички операции

Компјутерите работат со броеви кои припаѓаат на одреден преддефиниран ранг зависно од типот и оперираат само во неговите рамки. Ова поглавје го решава проблемот на основните аритметички операции над произволно големи цели броеви и тоа во било кој броен систем со база меѓу 2 и 16. Алгоритмите кои се користат во голема мера се познати, така што нивната имплементација во C++ користи стрингови за декларација на влезните параметри, освен базата која е од типот *int*. Имплементацијата е условена од одредени операции кои се користат во споменатите алгоритми, така што следниот код ги содржи сите потребни помошни операции кои се користат при имплементација на основните аритметички операции за произволно големи броеви за произволен броен систем.

```
#include <iostream>
#include <stdio.h>
#include <string>

int Vrednost(char c, int b)
	{
		int k = toupper(c);
		if ((k>=48)&&(k<=57))
			k = k - 48;
		else if ((k>=65)||(k<=70))
```

```
                        k = k - 55;
                else k = -1;
                if ((k==-1)||(k>=b))
                {
                        perror("Nedozvolena cifra");
                        return NULL;
                };
                return k;
        }

char VrednostInv(int i)
{
        if (i<10) return i+48;
        else return i+55;
}

bool Pomalo(std::string x, std::string y, int b)
{
        if (x.length()!=y.length())
                return (x.length()<y.length());
        else for(int i=0;i<x.length();i++)
                if (Vrednost(x[i],b)!=Vrednost(y[i],b))
                        return (Vrednost(x[i],b)<Vrednost(y[i],b));
        return false;
}

std::string TrimLead0(std::string x)
{
        if ((x[0]!='0')||(x=="0")) return x;
        else
                return TrimLead0(x.substr(1));
}
std::string PodeliSo2(std::string x, int b)
{
        std::string h = "";
        char carry = '0';
        int prvaCifra = 0;
        while(x.length()>0)
        {
                prvaCifra = Vrednost(x[0],b) + Vrednost(carry,b) * b;
                carry = VrednostInv(prvaCifra % 2);
                h = h + VrednostInv(prvaCifra / 2);
                x = x.substr(1);
        }
        if (h[0]=='0') h = h.substr(1);
        return h;
}

bool DozvolenaBaza(int b)
{
```

```
        if ((b<2)||(b>16))
        {
                perror("Nedozvolena baza...");
                return false;
        }
        else
                return true;
}
```

Функцијата *Vrednost()* на влез прима знак и база на броен систем, а како резултат во декаден броен систем ја дава вредноста на знакот во соодветниот броен систем. Обратно, *VrednostInv()* на влез прима цел број, а како резултат враќа знак кој претставува цифра на „стринг“ броевите (0→0, …, 10→A, …, 15→F).

Функцијата *Pomalo()* е булова функција која испитува дали првиот број е помал од вториот, каде двата параметри се произволно големи цели броеви, а третиот параметар го одредува бројниот систем во кој работи оваа операција. Без разлика на бројниот систем, секој број би требало да биде „исчистен“ од водечки нули, па за ова се грижи операцијата *TrimLead0()*. Кај некои од операциите ќе биде потребно и целобројно делење со 2, па ова се изведува со операцијата *PodeliSo2()* во даден броен систем. Секако, последнава операција е наједноставна за бинарниот броен систем бидејќи се реализира само со елиминација на првата цифра оддесно. Последната функција *DozvolenaBaza()* испитува дали базата е дозволена или не, т.е. испитува дали влезниот параметар на оваа функција е меѓу 2 и 16, инклузивно.

4.2.1 Собирање

Сумата на било кои три едноцифрени броеви е најмногу двоцифрен број. Ова е едно од основните својства на децималните броеви. Ако се направи проверка, 9+9+9=27, тогаш правилото се потврдува, но ова не е правило кое важи само за декадниот броен систем, туку важи за било кој броен систем почнувајќи од бинарниот (1+1+1=11).

Ова просто правило нуди можност да се соберат два броја во било кој броен систем: десните краеви им се порамнуваат, а потоа се извршува еднократно поминување од десно кон лево при што се сумираат цифра по цифра, а евентуално добиената дополнителна цифра се префрла на следната итерација. Со оглед на тоа што дополнителната цифра е секогаш едноцифрен број, тогаш таа кога ќе се додаде на збирот на двете цифри од бројот на кои им одговара, ќе се добие најмногу двоцифрен број, поради претходното правило.

Овој алгоритам е всушност алгоритмот кој најверојатно ја претставува првата математичка постапка која ја учи секој човек – постапка за собирање на два броја. Следниот пример претставува постапка за собирање на два броја од окталниот броен систем, како пример за функционирање на постапката во произволен броен систем.

$$\begin{array}{ccccccc} 1 & 1 & & 1 & & & \\ & 4 & 6 & 2 & 2 & _8 & (2450) \\ 2 & 3 & 6 & 4 & 7 & _8 & (10151) \\ \hline 3 & 0 & 4 & 7 & 1 & _8 & (30471) \end{array}$$

Следниот код претставува имплементација на овој алгоритам каде првите два параметри ги претставуваат броевите кои се собираат, а третиот параметар ја претставува базата на бројниот систем во кој се собираат двата броја.

```
std::string Zbir(std::string x, std::string y, int b)
{
	if (!DozvolenaBaza(b)) return NULL;
	while(x.length()<y.length()) x = "0"+x;
	while(x.length()>y.length()) y = "0"+y;
	std::string z = "";
	int suma;
	bool carry = false;
	for(int i = x.length()-1; i>=0; i--)
	{
		suma = Vrednost(x[i],b) + Vrednost(y[i],b);
		if (carry) suma++;
		carry = (suma >= b);
		if (carry) suma = suma - b;
		z = VrednostInv(suma) + z;
	}
	if (carry) z = "1" + z;
	return TrimLead0(z);
}
```

Нека x и y се n-цифрени броеви, тогаш нивната сума е најмногу $(n+1)$-цифрен, а секоја од неговите цифри е пресметана во фиксен број на чекори. Според ова, вкупното време на извршување на алгоритмот за собирање е од видот c_0+c_1n, каде c_0 и c_1 се константи. Ваквата форма претставува линеарно време на извршување, па точните вредности на c_0 и c_1 се заменуваат со време на извршување $O(n)$.

4.2.2 Разлика

Операцијата разлика, како што е и познато, користи многу слична метода на собирањето со исто време на извршување, па за оваа метода е дадена само имплементацијата без посебен осврт на нејзините перформанси.

```
std::string Razlika(std::string x, std::string y, int b) {
	if (!DozvolenaBaza(b)) return NULL;
	std::string z = "";
	bool neg = Pomalo(x,y,b);
	if (neg)
	{
		z = x; x = y; y = z; z = "";
	}
	while(x.length()<y.length()) x = "0"+x;
	while(x.length()>y.length()) y = "0"+y;
	int raz;
	bool carry = false;
	for(int i = x.length()-1; i>=0; i--)
	{
		raz = Vrednost(x[i],b) - Vrednost(y[i],b);
		if (carry) raz--;
		carry = (raz<0);
		if (carry) raz = raz + b;
		z = VrednostInv(raz) + z;
	}
	z = TrimLead0(z);
	if (neg) z = "-" + z;
	return z;
}
```

4.2.3 Множење

Слично како и за собирањето, и за множењето може да се искористи школската метода за множење на два броја заснована на низа од суми кои претставуваат производ на првиот број со секоја цифра од вториот број. Овие вредности соодветно се поместуваат налево и потоа се собираат.

На пример ако треба да се пресмета производот на броевите 136_8 и 273_8, тогаш постапката може да се види на следната шема:

$$136_8 \times 273_8$$

		4	3	2	8
1	2	2	2		8
2	7	4			8
4	2	2	5	2	8

Постапката е едноставна, при што поместувањето налево е само брз начин за множење со базата на бројниот систем, што во овој пример е 8, па следниот код ја претставува имплементацијата на овој алгоритам.

```
std::string Proizvod(std::string x, std::string y, int b) {
       if (!DozvolenaBaza(b)) return NULL;
       std::string strNuli = "";
       std::string vkupnaSuma = "";
       int carry = 0;
       for(int i=y.length()-1; i>=0; i--)
       {
              std::string partSuma = strNuli;
              strNuli = strNuli + "0";
              for(int j=x.length()-1; j>=0; j--)
              {
                int k = Vrednost(x[j], b)*Vrednost(y[i], b)+carry;
                carry = k / b;
                partSuma = VrednostInv(k % b) + partSuma;
              }
              if (carry>0) partSuma = VrednostInv(carry) + partSuma;
              vkupnaSuma = Zbir(partSuma, vkupnaSuma, b);
       }
       return TrimLead0(vkupnaSuma);
}
```

Од аспект на времето може да се заклучи следното: ако и двата множители се *n*-цифрени тогаш има *n* редици кои треба да се соберат од каде секоја редица е со најмногу *2n* цифри, ако се земе во предвид и поместувањето налево. Според тоа, ако времето за множење на првиот број со една цифра т.е. наоѓање на една редица е *O(n)*, и ако се потребни *n-1* операции собирање за собирање на *n* редици, тогаш вкупното време би било *(n-1)×O(n)* или $O(n^2)$ т.е. квадратно во однос на големината на влезовите. Ова е полиномно време на извршување, но сепак е поспоро во однос на собирањето.

Но, постои и една друга метода за пресметување на производ на два броја. Се работи за метода која се базира на множење и делење со 2. На следната шема е покажана оваа постапка. Се поставуваат двата броја во две колони и потоа првиот број се дели со 2 без остаток, а вториот број се множи со 2. Постапката одиѐдо добивање на 1 во првата колона. Тогаш, сите

редици, во кои првата колона е непарен број , се бришат, а збирот на останатите броеви во втората колона го дава резултатот.

~~454_8~~	~~37_8~~
~~226_8~~	~~76_8~~
113_8	174_8
45_8	370_8
~~22_8~~	~~760_8~~
11_8	1740_8
~~4_8~~	~~3700_8~~
~~2_8~~	~~7600_8~~
1_8	17400_8
	$= 22163_8$

Имплементацијата на оваа метода е дадена во следниот код.

```
std::string Proizvod2(std::string x, std::string y, int b)
{
	if (!DozvolenaBaza(b)) return NULL;
	std::string vkupnaSuma = "";
	while(x!="")
	{
		if (Vrednost(x[x.length()-1],b) % 2 == 1)
			vkupnaSuma = Zbir(vkupnaSuma, y, b);
		y = Zbir(y, y, b);
		x = PodeliSo2(x, b);
	}
	return TrimLead0(vkupnaSuma);
}
```

Времето на извршување на овој алгоритам може да се пресмета на следниот начин: алгоритмот мора да заврши по n повторувања на циклусот, поради тоа што x се преполовува во секој циклус, а секој циклус опфаќа тест за парност на x, едно множење и едно делење со 2 и едно можно собирање, т.е. вкупно $O(n)$ операции, што значи дека вкупното време на извршување е $O(n^2)$, исто како и во претходниот алгоритам.

4.2.4 Делење

Слично како и за множењето, делењето може да се имплементира со алгоритам со квадратно време на извршување. Методата се изведува на тој начин што еден цел број x за да се подели со друг цел број $y \neq 0$, треба да се добие количник q и остаток r, така што $x=yq+r$ и $r<y$. Следниот код претставува рекурзивна имплементација на оваа метода, но за разлика од претходните операции, оваа операција враќа две вредности, количникот и

остатокот од делењето, па затоа истата е имеплементирана како процедура, а не како функција која враќа низа од знаци.

```
void Delenje(std::string x, std::string y, std::string &q,
std::string &r, int b)
{
	if (!DozvolenaBaza(b)) return;
	if (x=="")
	{
		q = "0";
		r = "0";
	}
	else
	{
		Delenje(PodeliSo2(x, b), y, q, r, b);
		q = Zbir(q, q, b);
		r = Zbir(r, r, b);
		if (Vrednost(x[x.length()-1], b) % 2 == 1) r =
Zbir("1", r, b);
		if (Pomalo(y, r, b)||(y == r))
		{
			r = Razlika(r, y, b);
			q = Zbir(q, "1", b);
		}
	}
}
```

Како што може да се види и во самата имплементација, функцијата завршува после *n* рекурзивни повици. Секој од повиците претставува низа од конечен број на операции, 2 (не сметајќи го тестот за базата) ако се работи за терминален случај, и 2 теста со најмногу 4 собирања и 1 одземање, покрај рекурзивниот повик. Според ова може да се заклучи дека и кај овој алгоритам времето на извршување е $O(n^2)$.

4.3 Најголем заеднички делител

Најголем заеднички делител (НЗД) за два цели броја е најголемиот цел број кој е делител и на едниот и на другиот број.

Секако, наједноставниот пристап би бил да се најдат факторите (простите делители) на влезните броеви, а потоа да се измножат само заедничките за влезните цели броеви. На пример, 975 = 3 × 5 × 5 × 13 и 312 = 2 × 2 × 2 × 3 × 13,би дале НЗД = 3 × 13 = 39. Но, поради недостаток на ефикасен алгоритам за факторирање на броеви, се бара подобро решение за алгоритам за НЗД.

Најпознат алгоритам за наоѓање на НЗД на два броја е Евклидовиот алгоритам кој е откриен пред околу 2000 години од старогрчкиот математичар Евклид и истиот се базира на следното рекурзивно правило:

Ако x и y се позитивни цели броеви, при што x≥y, тогаш nzd(x, y) = nzd(x mod y, y)

Според оваа формула може да се генерира следниот код кој го претставува овој алгоритам.

```
int NZD(int x, int y)
{
	if (y==0)
		return x;
	else
		return NZD(y, x % y);
}
```

Како што може и да се види во кодот, имплементацијата на алгоритмот е коректна затоа што таа директно ја користи самата дефиниција, со една мала разлика, а тоа е што никаде не се користи споредбата на влезните броеви *x* и *y*. Ова е од причина што *x mod y* кога *x ≤ y* е секогаш еднаков на *x*, па доколку на влезот во процедурата првиот број е помал од вториот, при наредното рекурзивно повикување на процедурата броевите си ги менуваат местата.

Со оглед на тоа што овој алгоритам користи само основни аритметички операции, тогаш тој би можел да се имплементира и за произволно големи цели броеви, во произволен броен систем (меѓу 2 и 16), со користење на операциите кои беа имплементирани во претходното поглавје. Оваа имплементација е дадена со следниот код.

```
std::string NZD2(std::string x, std::string y, int b)
{
	if (y=="0")
		return x;
	else
	{
		std::string q;
		std::string r;
		Delenje(x, y, q, r, b);
		return NZD2(y, r, b);
	}
}
```

За да се одреди времето на извршување на овој алгоритам потребно е прво да се утврди колку брзо се намалуваат влезните параметри на секој

нареден рекурзивен повик. Во рамките на еден повик, параметрите *(x, y)* стануваат *(y, x mod y)*, при што нивниот редослед е сменет, а поголемиот од нив, *x,* станува *x mod y*, што претставува значително намалување на бројот.

Ако се земе в предвид лемата: *ако x≥y, тогаш x mod y < x/2*, тогаш по било кои два последователни повици на процедурата, двата аргументи, и *x* и *y*, барем се преполовуваат, што значи дека должината на секој од двата броја се намалува за барем еден бит. Ако иницијално, двата влезни параметри се *n*-битни цели броеви, тогаш функцијата ќе се заврши во вкупно *2n* рекурзивни повици, а со оглед на тоа што секој повик вклучува делење кое се извршува во квадратно време, тогаш вкупното време на извршување е $O(n^3)$.

4.4 Тестирање на прости броеви

Тестирањето дали еден број е прост или не е важна задача во компјутерските науки. Најважно од ѐ е тоа што простите броеви се корист ат кај криптографските алгоритми. Постојат два типа на алгоритми кои испитуваат дали еден број е прост или не: детерминистички и пробабилистички.

Што е прост број? Тоа е број кој има точно два различни природни броја кои се негови делители, и тоа 1 и тој самиот.

4.4.1 Тривијален алгоритам

Како последица на претходната дефиниција, најлогичниот и најтривијален пристап за тестирање дали еден број е прост или не, би било да се следи неговата дефиниција, т.е. тестот да се реализира со кодот:

```
bool eProst(int N) {
       if (N == 1)
            return false;
       if (N == 2)
            return true;
       for(int i = 2 ; i < N ;i++)
                if (N % i == 0) return false;
       return true;
}
```

Секако кај овој алгоритам треба да се земе в предвид дека 1 не е прост број, поради тоа што тој не ја исполнува дефиницијата за прост број. Но, дали може да се подобри овој алгоритам? Како прво, може да се покаже дека може

да се тестираат само делители помали или еднакви на $\sqrt{n}$ т.е. ова непосредно следува од следниот израз:

$$(\exists d)\ (1 < d < n)\ d|n \Rightarrow (\exists d_0)\ (1 < d_0 < \sqrt{n}) d|n$$

Ако $\sqrt{n}$ е цел број, тогаш n не е прост, инаку се претпоставува дека првиот најден делител на n е d_1, при што $\sqrt{n} < d_1 < n$, но, ако е ова случај, тогаш има и помал делител на n, а тоа е $d_2 = n/d_1$, кој е помал од $\sqrt{n}$. Според ова, и претпоставката е неточна т.е. ако има делител поголем од $\sqrt{n}$, тогаш постои негов „пар“ помал од $\sqrt{n}$. Со ова изразот е докажан, што значи дека горниот алгоритам може да се подобри на тој начин што циклусот не мора да оди до вредноста на бројот што се испитува, туку може да оди до најмногу $\sqrt{n}$.

Дополнително, постои уште едно многу едноставно подобрување, а тоа е што ако се испита парноста на бројот што се тестира, можно е во циклусот да се прескокнуваат парните броеви. Алгоритмот со овие две подобрувања би изгледал како во следниот код:

```
bool eProst2(int N) {
	if (N == 1)
		return false;
	if (N == 2)
		return true;
	if (N % 2 == 0)
		return false;
	for (int d = 3; d <= (int)sqrt((double)N); d = d + 2)
		if (N % d == 0)
			return false;
	return true;
}
```

Последната идеја може да се обопшти, па во таа ситуација би се испитувале само простите делители. Претпоставувајќи дека делењето на два броја е единечна операција, комплексноста на алгоритмот е $O(\sqrt{n})$, а последното подобрување ја менува само константата. Дополнителното унапредување на алгоритмот придонесува комплексноста на алгоритмот да стане $O(\sqrt{n}/ln(n))$.

4.4.2 Пробабилистички алгоритми

Главната примена на простите броеви е криптографијата, која бара генерирање и тестирање на прости броеви со стотици цифри, а од друга страна алгоритмот нема да испита броеви околу 10^{100} во реално време, па

алгоритмот се применува во пракса само за прелиминарни тестирања, т.е. големите броеви се тестираат со овој алгоритам со циклус за испитување на делители до првите милион броеви, а потоа, ако не се пронајде соодветен делител, во продолжение се користат пробабилистичките алгоритми.

Во контекст на ова, се поставува прашање за едноставен пробабилистички алгоритам со кој ќе може да се испита дали некој цел број е прост или не?

Пробабилистичките тестови се најпопуларните тестови за прости броеви и тие користат, независно од испитуваниот број *x*, некои други броеви *a* кои случајно се избрани од некој дефиниран простор. Ваквите тестови никогаш не покажуваат дека прост број е сложен, но можно е да покажат дека сложен број е прост.

Веројатноста на грешка се редуцира со повторување на алгоритмот за неколку независно избрани вредности на *a* и тоа за 2^{-k} каде *k* е бројот на повторувањата на постапката.

Основната структура на тестовите за прости броеви со случајни броеви е следната:

> **Чекор 1**. Случајно се избира број *a*
>
> **Чекор 2**. Зависно од избраниот тест се испитува одредена еквивалентност зависна од *a* и тестираниот број *n*. Ако еквивалентноста не важи, тогаш се заклучува дека *n* е сложен број, а *a* е доказ за сложеност на тестот и постапката запира.
>
> **Чекор 3**. Се повторува постапката од чекор 1, **è** додека не се постигне одредена точност на резултатот.

Ако по неколку итерации не се докаже дека *n* е сложен број, тогаш тој се декларира за веројатно прост, од каде и доаѓа името на тестовите – пробабилистички. Наједноставниот пробабилистички алгоритам за тестирање на прости броеви се крие зад малата теорема на Fermat од 1640 година, но со оглед на тоа што таа се базира на модуларна аритметика, прво е даден краток осврт на овој проблем.

Модуларна аритметика

Со постојано додавање и множење броевите можат да станат навистина огромни, но за среќа во одредени ситуации од реалниот живот се вршат соодветни ресетирања, т.е. повторно започнување од некоја почетна вредност и продолжување со истите операции. Таков е случајот со часовите

од едно денонoќие кога часот се поставува на 0 кога ќе ја достигне вредноста 24, или пак со месеците, кога се поставува месец Јануари кога месеците ќе ја достигнат вредноста 13. Слична е и ситуацијата со компјутерските процесори каде броевите се ограничени на некоја големина (на пример 32 бита) која се смета за доволна за повеќето намени. Но, како што беше претставено и претходно, постојат ситуации кога се потребни и потенцијално неограничени броеви за одредени` операции, како на пример испитување дали некој број е прост, или во криптографија, каде се користат многу поголеми броеви отколку 32 битните.

Модуларната аритметика претставува систем за справување со ограничен ранг на цели броеви. Се дефинира *x* модуло *N* да го претставува остатокот од делење на *x* со *N*. Ако ова се претстави формално така што ако *x=qN+r*, каде *0 ≤ r < N*, тогаш *x* модуло *N* е еднакво на *r*, се добива можност за подобрена нотација на оваа еквиваленција меѓу броевите: *x* и *y* се конгруентни по модуло *N*, ако тие се разликуваат само во некој производ на *N* со цел број, или формално:

$$x \equiv y (\mathrm{mod}\, N) \Leftrightarrow N \mid (x - y)$$

Пример: 274 ≡ 24 (mod 25), затоа што 274 = q × 25 + 24, или пак може да се користат и негативни броеви, на пример 274 ≡ -1 (mod 25), затоа што 274 = q × 25 – 1.

Едно толкување на модуларната аритметика е дека таа се справува со сите можни цели броеви, при тоа делејќи ги на *N* класи на еквиваленција каде секој член во рамките на една еквивалентна класа е заменлив со било кој друг од неговата класа, но од аспект на модуло *N*.

Класите на еквиваленција се претставуваат со следната формална дефиниција:

$$C_i = \{i + kN | k \in Z\}, \text{ за } 0 \le i < N$$

На пример, постојат четири класи на еквиваленција за *N=4*, и тоа:

$C_0 = \{4k | k \in Z\} = \{\dots, -8, -4, 0, 4, 8, \dots\}$

$C_1 = \{1 + 4k | k \in Z\} = \{\dots, -7, -3, 1, 5, 9, \dots\}$

$C_2 = \{2 + 4k | k \in Z\} = \{\dots, -6, -2, 2, 6, 10, \dots\}$

$C_3 = \{3 + 4k | k \in Z\} = \{\dots, -5, -1, 3, 7, 11, \dots\}$

Од дефиницијата на класите на еквивалентност и од самата заменливост на елементите во внатрешноста на класите, лесно се изведуваат собирањето и множењето на конгруенциите, па ако $x \equiv x' \ (mod\ N)$ и $y \equiv y' \ (mod\ N)$, тогаш

$x + y \equiv x' + y' \pmod N$ и $xy \equiv x'y' \pmod N$

Со вака дефинираните операции за собирање и множење лесно се покажува дека важат и асоцијативниот, комутативниот и дистрибутивниот закон над овие операции, т.е.

$(x + y) + z \equiv x + (y + z) \pmod N$

$xy \equiv yx \pmod N$

$(x + y)z \equiv xz + yz \pmod N$

Изложените факти укажуваат на тоа дека е дозволена редукција на некои меѓурезултати на нивните остатоци по модуло N во било која фаза на секвенца од аритметички операции, затоа што ваквите упростувања драматично можат да го поедностават извршувањето на алгоритмите. Пример за вакво упростување:

$2^{476} \equiv 16^{119} \equiv 1^{119} \equiv 1 \pmod{15}$

Кај модуларната аритметика може да се дефинираат различни алгоритми за модуларно собирање, множење, делење или степенување, така што кај криптографијата и пробабилистичките тестови за прости броеви многу се користи алгоритмот за модуларно степенување т.е. резултат на изразот $x^y \bmod N$, каде вредностите на x, y или N може да се и по неколку стотини бита, при што и самиот резултат може да биде од тој ред на големина.

Имплементацијата на овој алгоритам се состои во идејата да се обезбеди ограничување на големината на меѓурезултатите, така што таа се состои во тоа што $x^y \bmod N$ се пресметува со последователно множење на $x \bmod N$, па резултантанта секвенца би била:

$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^3 \bmod N \rightarrow \ldots \rightarrow x^y \bmod N$

Овој алгоритам е даден со следниот код:

```
int modulo1(int x, int y, int N)
{
    long long rez = 1;
    for(int i = 0; i < y; i++)
        rez = (rez * x) % N;
    return rez % N;
}
```

Како што може да се види и од самиот код, неговото време на извршување е *O(y)*, но тоа може да се подобри до *O(log(y))* со методата за степенување со квадрирање. Идејата е едноставна:

$$x^y = \begin{cases} 1 & y = 0 \\ (x^2)^{\frac{y}{2}} & y \bmod 2 = 0, y > 0 \\ x(x^2)^{\frac{y-1}{2}} & y \bmod 2 = 1 \end{cases}$$

Во овој случај резултантната секвенца би била:

x mod N → x^2 *mod N* → x^4 *mod N* → ... → $x^{2(logy)}$ *mod N*

а ова би одговарало на следниот пример:

$$x^{25} = x^{11001_2} = x^{10000_2} x^{1000_2} x^{1_2} = x^{16} x^8 x^1$$

Имплементацијата на алгоритмот би била:

```
int modulo2(int x,int y,int N)
{
    long long p = 1, q = x;
    while(y > 0)
       {
        if(y % 2 == 1) p = (p * q) % N;
        q = (q * q) % N;
        y = y / 2;
    }
    return p % N;
}
```

4.4.3 Тест на Fermat за испитување на прости броеви

Како што беше споменато и претходно, за решавање на алгоритмот за тестирање прости броеви се користи **малата теорема на Fermat** која гласи:

ако *p* е прост број, тогаш за секој $1 \le x < p$ важи $x^{p-1} \equiv 1 \ (mod\ p)$

Теоремата е дадена без доказ, кој може да се најде во различна литература[1] и таа нуди можност да се испита дали еден број е прост без користење на фактори. Всушност, овој тест не испитува дали еден број е прост или не, туку испитува дали бројот е сложен, па често ваквите алгоритми се нарекуваат и како тестови за сложеност на броеви.

```
unsigned int GenerirajBrojDo(unsigned int x)
```

[1] Р. Малчески, В. Малческа, „Математика 1“, ФОН Универзитет, 2010, стр. 93

```
{
       time_t s;
       time(&s);
       srand((unsigned int) s);
       return ((rand()*rand()) % (x-1)) + 1;
}

bool prostPoFermat(long long N,int k)
{
    if (N == 1) return false;
    for(int i=0;i<k;i++)
       {
        long long a = GenerirajBrojDo(N);
        if (modulo2(a,N-1,N) != 1)
            return false; /* N e slozen */
       }
    return true; /* N e verojatno prost */
}
```

Како што може да се види и во кодот, со циклусот се дава можност на процедурата да врати *false* вкупно *k* пати, така што самата математичка дефиниција се дополнува со овој циклус затоа што повеќе итерации ќе придонесат до подобра прецизност на резултатот, но ќе земат повеќе време, па затоа оваа имплементација го прима бројот на итерации како влезен параметар.

Но, иако овој алгоритам е доста прецизен во пракса, постојат одредени сложени броеви *p* кои се нарекуваат броеви на *Carmichael* за кои сите вредности $a < p$ за кои $НЗД(a,p)=1$, $(a^{p-1}) \bmod p = 1$. Ако се примени тестот на *Fermat* на ваков број, веројатноста да се избере број *a* за кој ќе важи $НЗД(a, p) \neq 1$ е мала, па во таа ситуација тестот ќе врати погрешен резултат со висока веројатност. Овој проблем може да се надмине, во поголема мера, на тој начин што дополнително од овој тест, може да се истестира пронаоѓање на делител на бројот што се тестира, во множеството на малите прости броеви. Секако, постојат и други пробабилистички тестови за прости броеви кои ја немаат оваа аномалија и кои не се опфатени овде.

5. „Раздели и владај“ стратегија

Стратегијата „Раздели и владај“ го решава проблемот со:

Чекор 1. Разбивање на проблемот на потпроблеми кои самите претставуваат проблем за себе

Чекор 2. Рекурзивно решавање на овие потпроблеми

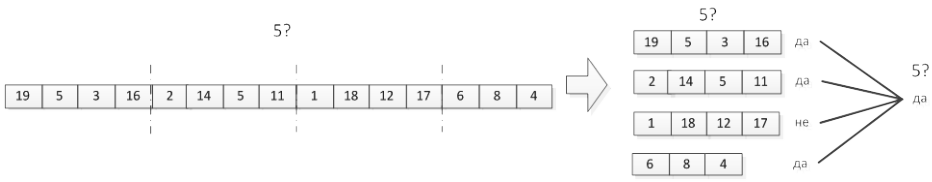
Чекор 3. Соодветно комбинирање на одговорите од решенијата.

Кај овие алгоритми вистинската работа се завршува по делови на три различни места: при парцијализирањето на проблемите во потпроблеми; на дното од рекурзијата каде проблемите се толку мали што се решаваат тривијално и при составувањето на парцијалните одговори. Овие три места се поврзани и управувани од страна на рекурзивната структура од која е изграден и самиот алгоритам.

Стратегијата може најдобро да се увиди преку примери на вакви алгоритми.

5.1 Пребарување на елемент во листа

Треба да се утврди дали во зададена листа постои елемент со дадена вредност. Стратегијата „раздели и владај“ го сугерира следниот пристап. Што е подолга листата, толку е потешко истата да се пребарува. Според ова влезната листа треба да се разбие на N подлисти, (на Слика 5.1 е даден пример каде N=4), па потоа во секоја од малите листи да се пребарува одделно.



Слика 5.1. Пример за пребарување со методата „Раздели и владај“

Бараната вредност се појавува во големата листа, само ако се појавува во барем една од малите листи, што значи дека крајниот резултат се добива со спојување на парцијалните резултати со логичко ИЛИ.

Следниот код ја содржи имплементацијата на овој алгоритам каде влезната листа во секоја итерација се разбива на 3 подлисти:

```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 30

typedef int IzbranTip;

typedef struct {
              int posleden;
              IzbranTip element[MAX_DOLZINA];
              } Niza;

bool SeNaodjaVo(IzbranTip srchEL, Niza *P, int pozOD, int pozDO)
{
       if (pozOD==pozDO)
              return (P->element[pozOD]==srchEL);
       else
              if (pozOD + 1 == pozDO)
                     return SeNaodjaVo(srchEL, P, pozOD, pozOD)
                            || SeNaodjaVo(srchEL, P, pozDO, pozDO);
              else if (pozOD + 2 == pozDO)
                     return SeNaodjaVo(srchEL, P, pozOD, pozOD) ||
                            SeNaodjaVo(srchEL, P, pozDO+1, pozDO+1)
                            || SeNaodjaVo(srchEL, P, pozDO, pozDO);
       int k = (pozDO - pozOD + 1) / 3;
       int k0 = (pozDO - pozOD + 1) % 3;
       if (SeNaodjaVo(srchEL, P,
                            pozOD, pozOD + (k-1) + k0))
              return true;
       if (SeNaodjaVo(srchEL, P,
                            pozOD + k + k0, pozOD + 2*k-1 + k0))
              return true;
       if (SeNaodjaVo(srchEL, P,
                            pozOD + 2*k-1 + k0, pozOD + 3*k-1 + k0))
              return true;
       return false;
}
```

Постои можност овој алгоритам да се поедностави ако се работи за сортирана листа при што поделбата на помали листи се врши така што листата се дели на 3 дела од кои средниот дел содржи само еден елемент (Слика 5.2). На овој начин една операција за споредба на бараната вредност и вредноста на средниот елемент од листата може да го најде бараниот елемент уште во наредниот чекор (во случај на еднаквост), или во најлош случај да се елиминираат две подлисти во наредниот тек на пребарувањето.

Слика 5.2. Пример за подобрена верзија на алгоритмот за пребарување со методата „Раздели и владај“

Овој алгоритам е имплементиран со следниот код:

```
bool SeNaodjaVo2(IzbranTip srchEL, Niza *P, int pozOD, int pozDO)
{
       int brElementi = pozDO+1-pozOD;
       if (P->element[pozOD + brElementi / 2]==srchEL)
              return true;
       else
              if (pozOD==pozDO)
                     return false;
              else
              if (P->element[pozOD + brElementi / 2]>srchEL)
                  return SeNaodjaVo2(srchEL, P,
                            pozOD, pozOD + brElementi / 2 - 1);
              else
                  return SeNaodjaVo2(srchEL, P,
                            pozOD + brElementi / 2 + 1, pozDO);
}
```

Специјален вид на алгоритам за пребарување кој ја користи оваа стратегија е методата за бинарно пребарување. Оваа метода пребарува клуч *k* во голема датотека која содржи клучеви *z[0, 1, ..., n-1]* кои се подредени.

Слично како во претходниот алгоритам, првин *k* се споредува со *z[n/2]*, и зависно од резултатот се повикува истата функција рекурзивно за левата половина од листата *z[0, ..., n/2 -1]* т.е. десната половина *z[n/2,...n-1]* од листата. Во овој случај рекурзијата е *T[n] = T([n/2]) + O(1)*, па извршувањето на целиот алгоритам би било *O(log n)*.

Следниот код ја користи методата за бинарно пребарување во една листа:

```
bool BinPrebaruvanje(IzbranTip srchEL, Niza *P,
                                    int pozOD, int pozDO)
{
       int brElementi = pozDO+1-pozOD;
       if (pozOD==pozDO)
              return (P->element[pozOD]==srchEL);
       if (P->element[pozOD + brElementi / 2]>=srchEL)
```

```
            return BinPrebaruvanje(srchEL, P,
                        pozOD, pozOD + brElementi / 2);
        else
            return BinPrebaruvanje(srchEL, P,
                        pozOD + brElementi / 2 + 1, pozDO);
}
```

5.2 Множење на големи броеви

Гаус забележал дека производот на два комплексни броеви покрај тоа што се решава со 4 множења, може да се реши и со 3:

(a + bi)(c + di) = ac – bd + (ad + bc)i

ad + bc = (a + b)(c + d) – ac – bd

Како што може да се види во претходните изрази овие три множења се *ac, bd* и *(a + b)(c + d)*, па редуцирањето од 4 множења на 3 можеби на прв поглед не претставува кој знае какво подобрување, но во рекурзивна смисла, подобрувањето е навистина значајно.

Ова подобрување има своја примена и кај регуларното множење. Ако *x* и *y* се два *n*-битни броеви, каде поради поедноставување без губење на општоста $n=2^k$, каде *k* е природен број, тогаш множењето на двата броја може да се изведе и со нивно разбивање на две половини со должина *n/2* :

$$x = 2^{\frac{n}{2}} x_l + x_d$$

$$y = 2^{\frac{n}{2}} y_l + y_d$$

Тогаш, производот на *x* и *y* може да се претстави со:

$$xy = (2^{\frac{n}{2}} x_l + x_d)(2^{\frac{n}{2}} y_l + y_d) = 2^n x_l y_l + 2^{\frac{n}{2}} (x_l y_d + x_d y_l) + x_d y_d$$

Производот *xy* се пресметува со изразот оддесно, што значи дека, поради тоа што собирањата бараат линеарно време на извршување, како и множењата со степени од 2, бидејќи тоа се практично поместување на лево со додавање на 0 на крајот. Значи, она што бара повеќе време се четирите *n/2*- битни множења $x_l y_l$, $x_l y_d$, $x_d y_l$ и $x_d y_d$ кои се решаваат со 4 рекурзивни повици. Според ова, оваа метода за множење на два *n*-битни броја започнува со рекурзивни повици за множење на четири пара *n/2*-битни броеви т.е. четири потпроблеми со половина големина, а потоа пресметување на претходниот израз во *O(n)* време, па според тоа вкупното време на извршување би било:

$$T(n) = 4T(\frac{n}{2}) + O(n)$$

Но, ако се употреби воочувањето на Гаус и множењето на два броја, наместо да се пресметува со 4 множења, да се пресметува со 3 и тоа:

$x_l y_l$, $x_d y_d$ и $(x_l + x_d)(y_l + y_d)$

Од трите производи $x_l y_l$ и $x_d y_d$ се исти како во претходниот случај, а третиот ги заменува другите два од претходниот израз на следниот начин:

$$x_l y_d + x_d y_l = (x_l + x_d)(y_l + y_d) - x_l y_l - x_d y_d$$

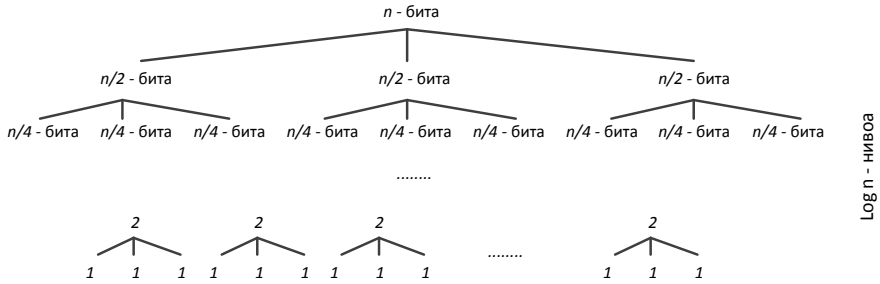
Алгоритмот кој го користи ова подобрување е претставен со следниот код.

```
unsigned int LPolovina(unsigned int x, int n) {
	return x >> n/2;
}

unsigned int DPolovina(unsigned int x, int n) {
	return x - (LPolovina(x, n) << n/2);
}

unsigned int Proizvod(unsigned int x, unsigned int y) {
	unsigned int max = x;
	if (x<y) max = y;
	if (max <= 1)
		return x && y;
	int n = pow2(Log2(Log2(max)+1));
	if (n < Log2(max)+1) n = n << 1;
	unsigned int xL = LPolovina(x, n), xD = DPolovina(x, n);
	unsigned int yL = LPolovina(y, n), yD = DPolovina(y, n);
	unsigned int P1 = Proizvod(xL, yL);
	unsigned int P2 = Proizvod(xD, yD);
	unsigned int P3 = Proizvod(xL + xD, yL + yD);
	return (P1 << n) + ((P3 - P1 - P2) << (n/2)) + P2;
}
```

Како што беше наведено, овој алгоритам користи 3 рекурзивни повици и дополнителни линеарни операции, така што неговата реализација има подобрено време на извршување, и тоа:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

Ова време на извршување може да се изведе ако се разгледа неговата шема на рекурзивни повици која формира стебло (Слика 5.3).



Слика 5.3. Раздели и владај множење. Нивои на рекурзија

Како што може да се види и на сликата на секое ниво на рекурзија, потпроблемите се преполовуваат така што на нивото *log n* потпроблемите се намалуваат до еден бит, каде што практично завршува и рекурзијата, што укажува на длабочина на стеблото од *log n*. Степенот на стеблото е 3, што резултира со 3^k потпроблеми на *k*-тото ниво. Секој потпроблем користи броеви од $n/2^k$ бита.

За секој потпроблем постои и линеарна количина на задачи за да се искомбинираат одговорите од неговите потпроблеми, па вкупното време на извршување на *k*-тото ниво е

$$3^k \times O(n/2^k) = (3/2)^k \times O(n)$$

Во коренот на стеблото *k=0*, па времето на извршување е *O(n)*, а на листовите од стеблото *k=log n*, па времето на извршување е $O(3^{\log n})$.

Нека $p = 3^{\log n}$, тогаш $\log p = \log 3^{\log n} = \log n \times \log 3 = \log n^{\log 3}$, па се добива $p = n^{\log 3}$. Оттука се добива дека времето на извршување на претходниот алгоритам е $O(n^{\log 3})$. Во внатрешните јазли на стеблото, времето на извршување се зголемува геометриски од *O(n)* кон $O(n^{\log 3})$, со фактор 3/2 по ниво. Поради тоа што сумата на било која геометриска низа со константен фактор е последниот елемент од низата, вкупното време на извршување на целиот алгоритам е $O(n^{\log 3})$ или приближно $O(n^{1,59})$.

5.3 Множење на матрици

Производ на две квадратни матрици од ред *n* е трета матрица од ред *n*. Нека *A* и *B* се двете *n×n* матрици, тогаш *C=AB* чиј *(i, j)*-ти елемент се пресметува со следната формула:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} ,$$

каде a_{ij} е *(i, j)*-ти елемент на *A*, а b_{ij} е *(i, j)*-ти елемент на *B*.

Овој алгоритам е претставен со следниот код.

```
#define MAX_DOLZINA 30

typedef int Matrica[MAX_DOLZINA][MAX_DOLZINA];

void Proizvod(int n, Matrica a, Matrica b, Matrica &c) {
       for(int i=0; i<n; i++)
              for(int j=0; j<n; j++) {
                     c[i][j] =0;
                     for(int k=0; k<n; k++)
                            c[i][j] += a[i][k]*b[k][j];
              };
}
```

Како што може да се види и во самиот алгоритам кој директно ја користи формулата, оваа формула користи $O(n^3)$ алгоритам за множење на матрици затоа што има вкупно n^2 елементи чија вредност треба да се пресмета, а од друга страна секој елемент се пресметува во $O(n)$ време.

Во 1969 година германскиот математичар Strassen претставил значително побрз алгоритам за множење на матрици, врз основа на стратегијата „Раздели и владај“.

Множењето на матрици кога редот на матриците е 2^k значително лесно може да се разбие на потпроблеми, за тоа што ова разбивање може да се направи по блокови. Доколку матриците не се од ред $2^k \times 2^k$, тогаш колоните и редиците на матриците се дополнуваат со нули додека не се добие бараниот формат.

Пристапот е следен: A и B се разбиваат на четири $n/2 \times n/2$ блокови:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \text{тогаш}$$

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Со оваа метода се применува стратегијата „Раздели и владај“, така што за да се пресмета производот AB од ред n, рекурзивно се пресметуваат 8 производи од ред $n/2$, а потоа се сумираат со неколку собирања со време $O(n^2)$. Вкупното време на извршување за овој алгоритам би било:

$$T(n) = 8T(n/2) + O(n^2)$$

Ако се пресмета ова време, ќе се добие време на извршување од $O(n^3)$, што значи дека не е добиено никакво подобрување во однос на тривијалниот

алгоритам кој ја користи дефиницијата за производ на две матрици. Но сепак, ефикасноста може да се подобри со еден трик кој Strassen го воочил, а кое се состои во пресметувањето на производот да се редуцира еден потпроблем, така што проблемот би се решил со разбивање на 7 потпроблеми наместо 8. Во овој случај новото време на извршување би било:

$$T(n) = 7T(n/2) + O(n^2)$$

што води до $O(n^{\log 7}) \approx O(n^{2,81})$, што сепак е подобрување во однос на $O(n^3)$.

Подобрувањето т.е. редукцијата на едно $n/2 \times n/2$ множење се состои во следната формула:

$$AB = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 + P_7 \end{bmatrix}, \text{ каде}$$

$$P_1 = A_{11}(B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12})B_{22}$$
$$P_3 = (A_{21} + A_{22})B_{11}$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$P_7 = (A_{21} - A_{11})(B_{11} + B_{12})$$

Овој алгоритам е претставен со следниот код.

```
void Zbir(int n, Matrica a, Matrica b, Matrica &c) {
	for(int i=0; i<n; i++)
		for(int j=0; j<n; j++)
			c[i][j] = a[i][j] + b[i][j];
}

void Razlika(int n, Matrica a, Matrica b, Matrica &c) {
	for(int i=0; i<n; i++)
		for(int j=0; j<n; j++)
			c[i][j] = a[i][j] - b[i][j];
}

void Razdeli(int n, Matrica a, Matrica &A, Matrica &B,
                                Matrica &C, Matrica &D) {
	for(int i=0; i<n/2; i++)
		for(int j=0; j<n/2; j++)
		{
			A[i][j] = a[i][j];
			B[i][j] = a[i][n/2+j];
			C[i][j] = a[n/2+i][j];
			D[i][j] = a[n/2+i][n/2+j];
		}
}
```

```
void Spoj(int n, Matrica A, Matrica B, Matrica C,
                                    Matrica D, Matrica &a) {
	for(int i=0; i<n; i++)
		for(int j=0; j<n; j++)
		{
			a[i][j] = A[i][j];
			a[i][n+j] = B[i][j];
			a[n+i][j] = C[i][j];
			a[n+i][n+j] = D[i][j];
		}
}

void Proizvod2K(int n, Matrica a, Matrica b, Matrica &c) {
	if (n==1)
		c[0][0] = a[0][0] * b[0][0];
	else
	{
		Matrica A11, A12, A21, A22;
		Matrica B11, B12, B21, B22;
		Matrica P1, P2, P3, P4, P5, P6, P7;
		Matrica POM1, POM2, POM3, POM4, POM5, POM6;
		Razdeli(n, a, A11, A12, A21, A22);
		Razdeli(n, b, B11, B12, B21, B22);

		Razlika(n/2, B12, B22, POM1);
			Proizvod2K(n/2, A11, POM1, P1); //P1
		Zbir(n/2, A11, A12, POM1);
			Proizvod2K(n/2, POM1, B22, P2); //P2
		Zbir(n/2, A21, A22, POM1);
			Proizvod2K(n/2, POM1, B11, P3); //P3
		Razlika(n/2, B21, B11, POM1);
			Proizvod2K(n/2, A22, POM1, P4); //P4
		Zbir(n/2, A11, A22, POM1);
			Zbir(n/2, B11, B22, POM2);
			Proizvod2K(n/2, POM1, POM2, P5); //P5
		Razlika(n/2, A12, A22, POM1);
			Zbir(n/2, B21, B22, POM2);
			Proizvod2K(n/2, POM1, POM2, P6); //P6
		Razlika(n/2, A21, A11, POM1);
			Zbir(n/2, B11, B12, POM2);
			Proizvod2K(n/2, POM1, POM2, P7); //P7
		Zbir(n/2, P5, P4, POM2);
			Razlika(n/2, POM2, P2, POM3);
			Zbir(n/2, POM3, P6, POM1); //C11
			Zbir(n/2, P1, P2, POM2); //C12
			Zbir(n/2, P3, P4, POM3); //C21
		Zbir(n/2, P1, P5, POM5);
			Razlika(n/2, POM5, P3, POM6);
			Zbir(n/2, POM6, P7, POM4); //C22
```

```
                Spoj(n/2, POM1, POM2, POM3, POM4, c);
        }
}

void Proizvod2(int n, Matrica a, Matrica b, Matrica &c) {
        int m = pow2(Log2(n));
        if (m < n) m *= 2;
        for(int i=0; i<m; i++)
                for(int j=n; j<m; j++) {
                        a[i][j] = 0; b[i][j] = 0;
                };
        for(int i=n; i<m; i++)
                for(int j=0; j<m; j++) {
                        a[i][j] = 0; b[i][j] = 0;
                };
        Proizvod2K(m, a, b, c);
}
```

Како што може да се види и во самиот код, се користат функциите *Log2()* и *pow2()* кои се користеа и кај другите имплементации (Поглавје 2.2.2). Функциите *Zbir()* и *Razlika()* се за собирање на матрици од ист ред, а функциите *Razdeli()* и *Spoj()* се за имплементација на оваа стратегија, т.е. едната за разделување, а другата за спојување на блокови матрици. Вистинската функција која рекурзивно го решава производот на матриците е функцијата *Proizvod2K()* затоа што со последната функција *Proizvod2()* го сведува редот на матриците од n на ред $2^k>n$ и дополнителните редици и колони ги пополнува со нули, а потоа ја повикува рекурзивната функција.

6. Метода со откажувања (Backtracking)

Backtracking метода или метода со откажувања е метода која се базира на разгледување на сите можности за конструкција на решението кое треба да се реши. Се работи за општа метода која се користи за решавање на тешки комбинаторни проблеми со помош на разгледување на потпроблеми и донесување правилни одлуки во однос на исходот. Методата бара тестирање кое разгледува потпроблем и брзо донесува одлука дали:

- потпроблемот нема решение т.е. грешка

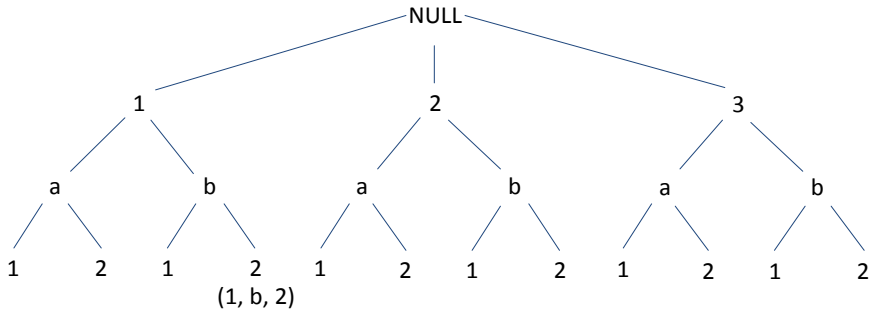
- најдено е решение на потпроблемот т.е. успех

- сеуште не е можно да се одреди исходот т.е. неодредено

Бараното решение треба да се изрази во форма на *n*-торка од облик $(x_1, x_2, \ldots, x_n)$, каде x_i е елемент на некое конечно множество X_i. Декартовиот производ $X_1 \times X_2 \times \ldots \times X_n$ е надмножество на просторот на решенија така што за да некоја *n*-торка биде решение треба да бидат задоволени некои услови, карактеристични за самиот проблем. Ако m_i е бројот на елементи на множеството X_i, тогаш има вкупно $m_1 \times m_2 \times \ldots \times m_n$ можни *n*-торки.

Просторот на можни решенија може да се посматра како подредено стебло каде коренот е *NULL*, а неговите вкупно m_1 деца на број ги претставуваат сите можни *n*-торки за кои првата компонента ја има вредноста на соодветниот коренски јазол за тоа подстебло. Понатаму, вкупно m_k-те јазли од *k*-тото ниво ги содржат сите вредности за *k*-тата компонента, и сè така до *n*-тото ниво каде секој лист од него содржи конкретна *n*-торка која претставува елемент од надмножеството на просторот на решенија. Тука уште останува да се испита дали оваа *n*-торка го задоволува проблемот или не.

На Слика 6.1 е претставено стебло со решенија за случајот кога $n=3$ и $X_1=\{1, 2, 3\}$, $X_2 = \{a, b\}$ и $X_3 = \{1, 2\}$. Во овој случај има вкупно 3×2×2 = 12 можни решенија, што се гледа и на самата слика, а до секое можно решение се доаѓа преку еден од листовите на стеблото, на пример четвртиот лист од третото ниво го одредува решението $(1, b, 2)$.



Слика 6.1. Стебло со решенија

Како што може да се види и на сликата, решенијата се сместуваат во стебло чиишто јазли се испитуваат рекурзивно, што е и основа на самата метода. Јазлите се стават во склад, така што еден чекор од алгоритмот е да се земе јазол од врвот на складот и да се испита дали тој јазол претставува решение на проблемот. Ако е најдено решение, се превзема одредена акција, а доколку тој не е решение, се генерираат неговите деца кои повторно завршуваат во складот. Алгоритмот започнува така што во складот се наоѓа само коренот, а завршува кога ќе се наиде на решение или кога ќе се испразни складот. Од оваа постапка не е тешко да се воочи дека редоследот за обработка на јазлите т.е. вадење од складот се одвива по *Preorder* методата.

Големината на просторот на решенија се зголемува експоненцијално со големината на проблемот *n*, па добриот backtracking алгоритам никогаш не го генерира целото стебло од решенија, туку ги отсекува гранките за кои алгоритмот утврдува дека не водат до решение. Имено, при самата постапка на генерирање на стеблото, на *k*-тото ниво веќе се фиксирани првите *k* компоненти. Доколку се утврди, според условите на проблемот, дека овие *k* компоненти не задоволуваат никакво решение, тогаш воопшто не се ни генерираат децата на тој јазол од *k*-тото ниво, затоа што ни тие нема да доведат до решение.

6.1 Проблем на N кралици

Проблемот се состои во поставување на *N* кралици на шаховска табла со димензии *N×N* и притоа не постојат две кралици кои се напаѓаат една со друга во ниту една насока. За решавање на овој проблем, очигледно е дека во секоја редица мора да биде по една кралица. Кај решението кое е дадено се користи низа од *N* елементи кои примаат целобројни вредности. Самите вредности ја одредуваат колоната на која е поставена кралицата од соодветната редица. Имплементацијата е дадена со следниот код:

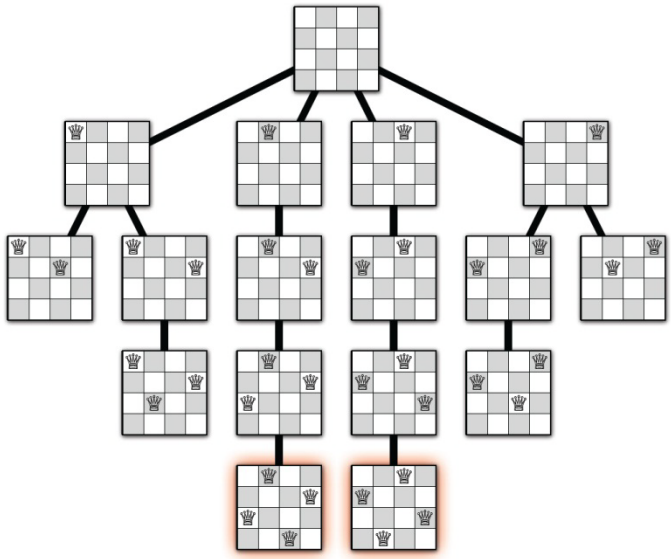
```
#include <iostream>
#include <stdio.h>
#define MAX_DOLZINA 20
typedef int Resenie[MAX_DOLZINA];
int BR = 0;

void DodadiVoResenija(Resenie A, int N) {
	for(int i=0; i<N; i++) {
		for(int j=0; j<N; j++)
			if (A[i]==j) std::cout<<"*";
			else std::cout<<char(176);
		std::cout<<"\n";
	}
	std::cout<<"\n";
	BR++;
}

bool Moze(Resenie A, int nivo, int pozicija, int Red) {
	if (nivo==0) return true;
	for(int i=nivo-1; i>=0; i--) {
		if (A[i] == pozicija) return false;
		int pozL = pozicija - nivo + i;
		int pozD = pozicija + nivo - i;
		if (pozL>=0)
			if (A[i]==pozL) return false;
		if (pozD<=Red)
			if (A[i]==pozD) return false;
	}
	return true;
}

void NQueens(Resenie &A, int nivo, int N) {
	if (nivo + 1 == N) {
		for(int i=0; i<N; i++)
			if (Moze(A, nivo, i, N)) {
				A[nivo] = i;
				DodadiVoResenija(A, N);
			}
		return;
	}
	for(int i=0; i<N; i++)
		if (Moze(A, nivo, i, N)) {
			A[nivo] = i;
			NQueens(A,nivo+1,N);
		}
}
```

Како што може да се види и во самиот код, главната процедура е *NQueens()*, која има три параметри, низата *A* која е парцијалното решение до *nivo*-тото ниво од стеблото со степен *N*. Се гледа дека условот *(nivo+1==N)* го претставува терминалниот случај на рекурзијата, а со условот *Moze(A, nivo, i,*

N) практично се извршува „отсекувањето“ на гранките кои не водат до решение. Условот *Moze()* е со четири параметри и тој испитува дали во парцијалното решение *A* во редицата *nivo*, е дозволена позицијата *pozicija*. Секако, редот *Red* на шаховската табла е неминовен. Во однос на акцијата која се превзема кога ќе се најде решение, се користи процедурата *DodadiVoResenija()* која во оваа имплементација го испишува решението на екран и ја инкрементира глобалната променлива *BR*, која по завршувањето на алгоритмот ќе го даде вкупниот број на можни решенија. Со оглед на тоа што по дефиниција почетокот на алгоритмот се реализира со поставување на коренот во складот, повикот на алгоритмот е на следниот начин

```
Resenie Q;
for(int i=0; i<N; i++)
       Q[i] = -1;
NQueens(Q, 0, N);
```

Надмножеството на просторот на можни решенија е со вкупен број N^N, што значи дека и стеблото кое се развива со рекурзијата е полно стебло од степен *N*. На следната слика е прикажано стеблото за случај *N=4*.



Слика 6.2. Стебло со решенија за проблемот со 4 кралици

Од сликата се гледа начинот на барање на решенија и прекинување на гранките кои не водат до решение. Јазлите кои се расчленуваат на потпроблеми се означени. Во следната табела е даден бројот на решенија на проблемот со *N* кралици.

N	Број на решенија

N	Број на решенија

N	Број на решенија

1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352

10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512
17	95815104
18	666090624

19	4968057848
20	39029188884
21	314666222712
22	2691008701644
23	24233937684440
24	227514171973736
25	2207893435808350
26	22317699616364000

6.2 Судоку

Судоку понекогаш пишуван и како Су Доку, е логички-базирана игра (сложувалка, задача), позната и како „Бројка на место“ (Number Place) во САД. Целта на оваа канонична сложувалка е да се внесе број од 1 до 9 во секое поле од 9×9 матрица, која пак е составена од 3×3 подматрици наречени „региони“ , почнувајќи со разни броеви кои се дадени во некои полиња (Слика 6.3а). Секоја редица, колона и регион мора да содржи една и само една инстанца од секој од броевите 1 до 9. Решен Судоку проблем е прикажан на Слика 6.3б.

	8	4	5					9
		6						5
		1			6			8
			1		8	7		
			4		5			
		3	6		9			
4			9			3		
6						5		
7					2	1	9	

а) Поставен проблем

2	8	4	5	1	3	6	7	9
3	9	6	7	8	4	2	1	5
5	7	1	2	9	6	4	3	8
9	6	5	1	2	8	7	4	3
8	2	7	4	3	5	9	6	1
1	4	3	6	7	9	8	5	2
4	1	2	9	5	7	3	8	6
6	3	9	8	4	1	5	2	7
7	5	8	3	6	2	1	9	4

б) Решение

Слика 6.3. Судоку проблем

И покрај тоа што прв пат се појавува во американско енигматско списание во 1970, Судоку добива на популарност и во Јапан во 1986-та, за во 2005-та да се здобие со меѓународна популарност.

Во ова поглавје е дадено решение за решавач на Судоку проблеми. Поради тоа што се работи за комбинаторна сложувалка, решението е изведено со backtracking стратегијата, но и решението со backtracking бара вистински избор на пристапот како ќе се развие стеблото на решенија.

Можно е решение со сместување на цели редици во јазлите од стеблото, но ваквото решение бара користење на склад во кој ќе се сместуваат целите редици, што значи дека ќе се користи голема количина на меморија. Овој мемориски простор дополнително се зголемува поради фактот што за секоја пермутација на броевите во една редица се формира посебна гранка. Ова значи дека стеблото во најлошата ситуација би било со степен 9!, иако длабочината на стеблото би одела најмогу до 9.

Решението кое е дадено овде користи стебло на решенија во кое сместува цифри во секој од јазлите. Со ваквата имплементација се добива длабоко стебло (максимум 9×9=81) со степен 9. Поради природата на самата backtracking стратегија, отсекувањето на гранките кои не водат до решение е навистина ефективно и води до отсекување на навистина долги гранки во стеблото од решенија, а решението е дадено во следниот код.

```
#include <iostream>
#include <stdio.h>

typedef short Resenie[9][9];
typedef short Redica[9];
unsigned int BR = 0;

bool Moze(Resenie A, short Cifra, short red, short kol) {
	if (A[red][kol]!=0) return false;
	for(int i=0;i<9; i++) {
		if ((i!=red)&&(A[i][kol]==Cifra)) return false;
		if ((i!=kol)&&(A[red][i]==Cifra)) return false;
	}
	short KvadrantNivo = red / 3;
	short KvadrantPoz = kol / 3;
	for(int i=KvadrantNivo*3;i<KvadrantNivo*3+3;i++)
		for(int j=KvadrantPoz*3;j<KvadrantPoz*3+3;j++)
			if ((i!=red)&&(j!=kol)&&(A[i][j]==Cifra))
					return false;
	return true;
}

void IspisiRedica(Redica A) {
	for(int i=0; i<9; i++)
		std::cout << A[i] << " ";
	std::cout << "\n";
}

void DodadiVoResenija(Resenie A) {
	for(int i=0; i<9; i++)
		IspisiRedica(A[i]);
	std::cout << "\n";
	BR++;
```

```
}

void Sudoku(Resenie &A, short red, short kol) {
	if ((red==8)&&(kol==8)) {
		if (A[red][kol]!=0)
			DodadiVoResenija(A);
		else
		for(int i=1; i<=9; i++)
			if (Moze(A, i, 8, 8)) {
				A[8][8] = i;
				DodadiVoResenija(A);
			}
	}
	else {
		if (A[red][kol]!=0) {
			if (kol<8)
				Sudoku(A, red, kol+1);
			else
				Sudoku(A, red+1, 0);
		}
		else
			for(int i=1; i<=9; i++)
				if (Moze(A, i, red, kol)) {
					A[red][kol] = i;
					if (kol<8)
						Sudoku(A, red, kol+1);
					else
						Sudoku(A, red+1, 0);
					A[red][kol] = 0;
				}
	}
}
```

Како што може да се види и во самиот код, во процедурата *DodadiVoResenija()* се врши испис на најденото решение, но се инкрементира и глобалната променлива *BR* која го означува бројот на најдени решенија. Главната акција се случува во процедурата *Sudoku()* во која се пушта шаблонот со дадените броеви, кој шаблон рекурзивно се пополнува, според условите на сложувалката. Оваа процедура користи булова функција *Moze()* со четири влезни параметри и која враќа информација дали е дозволена позиција на цифрата *Cifra*, на позицијата *(red, kol)* во парцијалното решение *A*.

Алгоритмот бара шаблонот да биде поставен во облик на 9×9 матрица од типот *short*, каде со 0 се означени местата за кои треба да се најде соодветна цифра. Како илустрација, со следниот код е даден повик на алгоритмот за проблемот даден на Слика 6.3.

```
int main()
{
	Resenie A = {
	{0, 8, 4, 5, 0, 0, 0, 0, 9},
	{0, 0, 6, 0, 0, 0, 0, 0, 5},
	{0, 0, 1, 0, 0, 6, 0, 0, 8},
	{0, 0, 0, 1, 0, 8, 7, 0, 0},
	{0, 0, 0, 4, 0, 5, 0, 0, 0},
	{0, 0, 3, 6, 0, 9, 0, 0, 0},
	{4, 0, 0, 9, 0, 0, 3, 0, 0},
	{6, 0, 0, 0, 0, 0, 5, 0, 0},
	{7, 0, 0, 0, 0, 2, 1, 9, 0}};

	Sudoku(A, 0, 0);
	std::cout << BR << " resenija.\n";
	system("pause");
}
```

7. Сортирање

Кај проблемот на сортирање во компјутерските науки, влезот е низа од броеви, а излезот мора да биде низа од истите броеви, но сортирана во неопаѓачки редослед. Од аспект на компјутерските науки, ова е интересен проблем од две причини. Како прво, многу апликации бараат сортирање на голем број елементи (телефонски именици, уплати и сл.), па ефикасноста на извршување на овие алгоритми може да биде многу важна. Како второ, постојат повеќе добро-проучени алгоритми за сортирање, што овозможува солидна практична тема за унапредување на вештините за анализирање и споредба на алгоритми. Во ова поглавје, без губење на општоста алгоритмите кои се обработени сортираат низи од цели броеви.

7.1 Сортирање со избор (Selection Sort)

Најслична метода на човечката интуиција за сортирање е методата со избор. Се наоѓа најголемиот елемент од листата и се поместува на соодветна локација, потоа ја наоѓа следната најголема и неа ја поместува на соодветна локација, иècтака додека листата не стане сортирана. За да премести еден елемент на друга позиција, оваа метода врши замена (swap) на елементот што се поместува со елементот кој се наоѓа на целната позиција. Како резултат, низата ќе има дел кој е сортиран во растечки редослед до крајот на низата, и дел од низата кој сеуште не е сортиран.

A[]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	замена
влез	26	9	7	4	8	5	38	7	
i=7	26	9	7	4	8	5	**38**	7	да
i=6	**26**	9	7	4	8	5	7	38	да
i=5	7	**9**	7	4	8	5	26	38	да
i=4	7	5	7	4	**8**	9	26	38	не
i=3	**7**	5	7	4	8	9	26	38	да
i=2	4	5	**7**	7	8	9	26	38	не
i=1	4	**5**	7	7	8	9	26	38	не
i=0	**4**	5	7	7	8	9	26	38	не
излез	4	5	7	7	8	9	26	38	

Слика 7.1. Пример за сортирање со избор

На Слика 7.1 е даден пример за сортирање со овој алгоритам. Во примерот во сивите полиња се наоѓаат елементите кои се споредуваат со крајот на несортираната подниза. Границата меѓу несортираниот и сортираниот дел од низата е означена.

Следниот код го имплементира овој алгоритам.

```
void Zamena(int &a, int &b) {
	int pom = a; a = b;  b = pom;
}

void SelectionSort(Niza &A, int N) {
	for(int i=N-1; i>=0; i--) {
		int max = A[0];
		int maxInd = 0;
		for(int j=0; j<=i; j++)
			if (A[j] > max) {
				max = A[j];
				maxInd = j;
			}
		if (i != maxInd)
			Zamena(A[i],A[maxInd]);
	}
}
```

Внатрешниот циклус на овој алгоритам ја наоѓа позицијата на максималниот елемент во несортираниот дел од низата, така што променливата *max* ја содржи вредноста на максималниот елемент кој е тековно поминат во циклусот, а *maxInd* е позицијата на тој елемент. Кога циклусот ќе заврши со прегледот, *max* и *maxInd* ќе ги содржат максималната вредност и нејзината позиција во целиот несортиран дел од низата.

Надворешниот циклус ја чува позицијата на следниот максимален елемент, така што со овој алгоритам, како што може да се види и во кодот, во еден момент од неговото извршување, несортираниот дел од низата е на почетокот од низата, а сортираниот е на крајот.

Кај овој алгоритам, податоците воопшто не влијаат на бројот на споредби, затоа што циклусите во кодот контролираат колку споредби ќе се направат и тие воопшто не ги ни гледаат самите податоци од низата. Бројот на замени може да варира, со оглед на тоа што во алгоритмот е предвиден услов за испитување дали е потребна замена или не, т.е. услов кој испитува дали максималниот елемент се наоѓа веќе на својата позиција или не. Ова значи, дека доколку на влез се појави веќе сортирана листа, алгоритмот е доволно паметен да не направи ниту една замена.

За да се пресмета времето на извршување, се тргнува од фактот дека бројот на споредби на елементи не зависи од самите вредности на низата. Надворешниот циклус се извршува *n-1* пати, а внатрешниот циклус во најлош случај се извршува *n* пати, па според тоа споредбата не може да се случи повеќе од $O(n^2)$ пати. Но, ова е само горна граница, затоа што во последното извршување од надворешниот циклус се извршува само една споредба, па попрецизно, ако во првото извршување на надворешниот циклус се прават *n* споредби, во второто *n-1*, ... , во *n*-тото една, тогаш вкупниот број на споредби е:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Но, и оваа вредност е $O(n^2)$, што ја претставува и долната граница на времето на извршување, па може да се заклучи дека ова е пресметаното време на извршување на овој алгоритам без разлика на влезната низа.

7.2 Сортирање со меурче (Bubble Sort)

Кај методата за сортирање со меурче се разгледуваат соседни елементи од низата и истите се подредуваат. Прво се разгледуваат првите два елементи, се подредуваат по потреба, па се разгледуваат вториот и третиот елемент, се подредуваат по потреба, па се разгледуваат третиот и четвртиот итн. На овој начин, по едно поминување на низата, најголемиот елемент завршува на крајот од низата. Ако се направи уште едно вакво поминување на низата, вториот најголем елемент ќе заврши на претпоследната позиција, а ако се направат *n* вакви поминувања, целата низа ќе биде сортирана.

Слично на претходната метода, и оваа метода во секој момент има сортиран дел кој е на крајот од низата и сеуште несортиран дел на почеток на низата.

Со оглед на тоа што во *k*-тото поминување на низата, во сортираниот дел има *k-1* елементи, тоа значи дека во несортираниот дел има *n-k+1* елементи, што значи дека споредувањето на соседни елементи нема потреба да ја поминува целата низа, туку треба да заврши до почетокот на сортираниот дел. На Слика 7.2 е даден пример за сортирање со меурче.

A[]	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	Број замени
влез	26	9	7	4	8	5	38	7	

i=7	26↔	9↔	7↔	4↔	8↔	5	38↔	7	6
i=6	9↔	7↔	4↔	8↔	5	26↔	7	38	5
i=5	7↔	4	8↔	5	9↔	7	26	38	3
i=4	4	7↔	5	8↔	7	9	26	38	2
i=3	4	5	7	7	8	9	26	38	0
i=2	4	5	7	7	8	9	26	38	0
i=1	4	5	7	7	8	9	26	38	0
i=0	4	5	7	7	8	9	26	38	0
излез	4	5	7	7	8	9	26	38	

Слика 7.2. Пример за сортирање со меурче

Во примерот се прикажани замените на елементите со соодветните следни соседни елементи. Границата меѓу несортираниот и сортираниот дел од низата е означена.

Како што може да се види во примерот во четири од осум поминувања на низата не се случува ниту една промена, па според ова може да се изврши и подобрување на овој алгоритам така што ќе се дозволат поминувања сè до првото поминување во кое нема да има ниту една промена.

Алгоритмот за сортирање со меурче во подобрената варијанта во C++ е прикажан со следниот код. И во овој алгоритам се користи истата процедура за замена на вредности на две променливи.

```
void BubbleSort(Niza &A, int N) {
	for(int i=N-1; i>=0; i--) {
		bool ImaPromena = false;
		IspisiNiza(A, N);
		for(int j=0; j<i; j++)
			if (A[j] > A[j+1]) {
				Zamena(A[j],A[j+1]);
				ImaPromena = true;
			}
		if (!ImaPromena) return;
	}
}
```

Внатрешниот циклус оди низ низата барајќи соседни елементи и менувајќи им ги местата доколку има потреба за тоа. Прво ги проверува првиот и вториот елемент, потоа втората и третата итн. Главниот ефект на овој циклус е што најголемиот елемент ќе заврши на крајот од низата која овој циклус ја проверува. Тука треба да се напомене дека овој циклус оди до крајот на несортирана подниза од низата што се сортира.

Надворешниот циклус го повторува внатрешниот циклус *n* пати. Првиот пат најголемиот елемент оди на својата позиција, вториот пат, вториот најголем елемент оди на својата позиција и ќе така до поставување на најмалиот елемент на својата позиција.

Како и во претходниот алгоритам, бројот на споредби не зависи од самите податоци, но затоа вредностите на елементите влијаат на бројот на замени кои се случуваат.

Кога податоците се сортирани на влезот, бројот на замените е 0, па подобрувањето на алгоритмот кое беше дополнително имплементирано ќе го заврши извршувањето веднаш по првото поминување, но без подобрувањето немаше да има замени, но ќе се извршеа сите можни споредби предвидени со двата циклуси.

Кога податоците не се сортирани, тогаш бројот на замени ќе варира со податоците.

Во врска со времето на извршување, дискусијата и заклучоците се потполно исти како и кај претходниот алгоритам за сортирање со избор т.е. времето на извршување и на овој алгоритам е квадратно т.е. $O(n^2)$.

7.3 Сортирање со вметнување (Insertion Sort)

Оваа метода ги сортира елементите на начин на кој многу луѓе сортираат карти за играње. Се започнува со празна рака во која по случаен избор се зема карта, од купче затворени карти на маса, и се вметнува во раката на соодветна позиција. За да се најде соодветната позиција на картата која треба да се вметне во раката, се споредува нејзината вредност со картите во раката, од лево кон десно.

Пример за сортирање со методата на вметнување е даден на Слика 7.3. Како што може да се види и во примерот, кај овој алгоритам сортираниот дел е на почетокот на низата, а несортираниот е на крајот од низата. Итераторот *i* ја одредува позицијата на последниот елемент од сортираниот дел во секој момент на извршување на алгоритмот.

A[]	0	1	2	3	4	5	6	7	Број замени
влез	26	9	7	4	8	5	38	7	
i=1	26	↔9	7	4	8	5	38	7	1
i=2	9	↔26	↔7	4	8	5	38	7	2
i=3	7	↔9	↔26	↔4	8	5	38	7	3
i=4	4	7	↔9	↔26	↔8	5	38	7	3

i=5	4	↔7	↔8	↔9	↔26	↔5	38	7	5
i=6	4	5	7	8	9	26	38	7	0
i=7	4	5	7	↔8	↔9	↔26	↔38	↔7	5
излез	4	5	7	7	8	9	26	38	

Слика 7.3. Пример за сортирање со вметнување

Имплементацијата на алгоритмот за сортирање со вметнување во C++ е дадена со следниот код. Функцијата *Zamena()* е истата како во претходните алгоритми.

```
void InsertionSort(Niza &A, int N) {
       for(int i = 0; i < N-1; i++) {
              int j=i+1;
              while(A[j]<A[j-1] && j>0) {
                     Zamena(A[j], A[j-1]);
                     j--;
                     }
       }
}
```

Внатрешниот циклус од алгоритмот го поместува елементот, кој во соодветниот чекор се вметнува, од неговата оригинална позиција (со вредност *i+1*) до позицијата на лево каде треба да се вметне.

Надворешниот циклус го превзема наредниот елемент кој треба да се вметне, т.е. го инкрементира *i*.

Од аспект на времето на извршување на алгоритмот со вметнување може да се види дека во *k*-тото поминување наназад се поминува низ сортираниот дел со должина *k*. Елементите по потреба се поместуваат за едно место надесно, сè додека не се ослободи место за елементот кој треба да се вметне. Поради должината на сортираниот дел од низата ваквите поместувања во најлош случај се вкупно *k* на број. Ова значи дека вкупниот број на операции во најлошиот случај е повторно

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

што значи дека и овој алгоритам има квадратно време на извршување т.е. $O(n^2)$, но и покрај ваквата асимптотска оценка, во пракса овој алгоритам се покажува како побрз од претходните алгоритми.

7.4 Повеќекратно сортирање со вметнување (Shell Sort)

Оваа метода е подобрена верзија на сортирањето со вметнување. За зададено k се разгледуваат k различни поднизи составени од елементи од оригиналната низа кои се на растојание од точно k позиции. Поточно за дадено k се разгледуваат поднизите:

P_0: a_0, a_k, a_{2k}, …

P_1: a_1, a_{k+1}, a_{2k+1}, …

…

P_{k-1}: a_{k-1}, a_{2k-1}, a_{3k-1}, …

Процесот се одвива со k-кратна примена на обичната метода за сортирање со вметнување над овие k поднизи, а секое сортирање на ваква подниза се нарекува k-потсортирање. Овој алгоритам работи така што за дефинирана опаѓачка низа $k_1>k_2>…>k_m=1$ извршува k_1-потсортирање, па k_2-потсортирање и така сè до k_m-потсортирање. Може да се забележи дека последното потсортирање е всушност обичното сортирање со вметнување над целата оригинална низа, што практично е доказ дека не треба да постои сомнеж за коректноста на алгоритмот.

Потсортирањата служат за делумно уредување на целата низа за да се намали работата на секое наредно потсортирање, а најчесто употребувана низа за реализација на овој алгоритам за сортирање е низата која се добива со последователно делење со 2 до 1 т.е. $[n/2],[n/4], …,1$, па пример за ваквото сортирање е даден на Слика 7.4. Во секоја редица од примерот со иста боја се означени елементите кои градат една подниза во рамките на соодветното k-потсортирање.

A[]	0	1	2	3	4	5	6	7
Влез на 4-потсортирање	26	9	7	4	8	5	38	7
Излез на 4-потсортирање	8	5	7	4	26	9	38	7
Влез на 2-потсортирање	8	5	7	4	26	9	38	7
Излез на 2-потсортирање	7	4	8	5	26	7	38	9
Влез на 1-потсортирање	7	4	8	5	26	7	38	9
Излез на 1-потсортирање	4	5	7	7	8	9	26	38

Слика 7.4. Пример за повеќекратно сортирање со вметнување

. Алгоритмот е имплементиран во C++ со следниот код.

```
void ShellSort(Niza &A, int N) {
	int stepen = N/2;
	while (stepen >= 1) {
		for(int i=0; i<stepen; i++) {
			for(int j=i; j<N-stepen; j = j + stepen) {
				int k = j + stepen;
				while(A[k] < A[k-stepen] && k>i) {
					Zamena(A[k], A[k-stepen]);
					k = k - stepen;
				}
			}
		}
		stepen = stepen / 2;
	}
}
```

Алгоритмот за повеќекратно сортирање со вметнување претставува вистински предизвик за истражувачите. Неговото време на извршување зависи од секвенцата на растојанијата меѓу елементи кои градат една подниза во рамките на едно *k*-потсортирање, но сеуште не постојат јасни и целосни оценки за сложеноста на алгоритмот. Просечното време на извршување е отворен проблем, а експерименталните мерења покажуваат дека алгоритмот е изненадувачки брз.

Постојат некои специјални облици на опаѓачката низа $k_1>k_2>...>k_m=1$, за која се изведени оценките на сложеноста на овој алгоритам, но ниту една од тие оценки не ги покрива сите можни случаи на низата. Во табелата во продолжение се дадени некои од изведените оценки за најлошиот случај на времето на извршување.

Израз	**Низа**	**Сложеност**	**Автор и година на публикација**
$\left\lceil \frac{n}{2^k} \right\rceil$	$\left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{n}{4} \right\rceil, \left\lceil \frac{n}{8} \right\rceil, \ldots, 1$	$O(n^2)$	Shell, 1959
$2\left\lceil \frac{n}{2^{k+1}} \right\rceil + 1$	$2\left\lceil \frac{n}{4} \right\rceil + 1, \ldots, 3, 1$	$O(n^{3/2})$	Frank и Lazzarus, 1960
2^k-1	1, 3, 7, 15, 31, 63, ...	$O(n^{3/2})$	Hibbard, 1963

2^k-1 (со 1)	1, 3, 5, 9, 17, 33, 65..	$O(n^{3/2})$	Papernov и Stasevich, 1965
последователни броеви само со фактори 2 и 3	1, 2, 3, 4, 6, 8, 9, 12..	$O(n \log^2 n)$	Pratt, 1971
$(3^k-1)/2$, не поголеми од $[n/3]$	1, 4, 13, 40, 121, …	$O(n^{3/2})$	Knuth, 1973
$4^k+3\times2^{k-1}+1$ (со 1)	1, 8, 23, 77, 281, …	$O(n^{4/3})$	Sedgewick 1986
$9(4^{k-1}-2^{k-1}) + 1,4^{k+1}-6\times2^k+1$	1, 5, 19, 41, 109, …	$O(n^{4/3})$	Sedgewick 1986

Слика 7.5. Извадок од листа на изведени низи за растојанија и нивни оценки кај Shellsort

7.5 Сортирање со спојување (Merge Sort)

Овој алгоритам ја користи стратегијата „раздели и владај“ т.е. влезната низа која треба да се сортира ја дели на две помали поднизи и се повикува рекурзивно самиот себе над двете поднизи. По успешно завршената операција над двете поднизи врши нивно спојување во една низа која практично претставува и резултат на самиот алгоритам.

Алгоритмот е рекурзивен, така што ако влезната низа има 2 елементи, тогаш ќе има два рекурзивни повици над низи со по 1 елемент. Кога ќе влезе низа од 1 елемент во процедурата, алгоритмот ја препознава низата дека е сортирана, со оглед на тоа што содржи само 1 елемент, и ја враќа непроменета. Откако ќе бидат вратени два резултати со по 1 елемент, се врши нивно спојување во една сортирана низа со 2 елементи.

Алгоритмот за сортирање со спојување е даден со следниот код.

```
void MergeSort(Niza &A, int pocetok, int kraj) {
       if (pocetok < kraj) {
              int sredina = (pocetok + kraj) / 2;
              MergeSort(A, pocetok, sredina);
              MergeSort(A, sredina+1, kraj);
              Spoj(A, pocetok, sredina, kraj);
       }
}
```

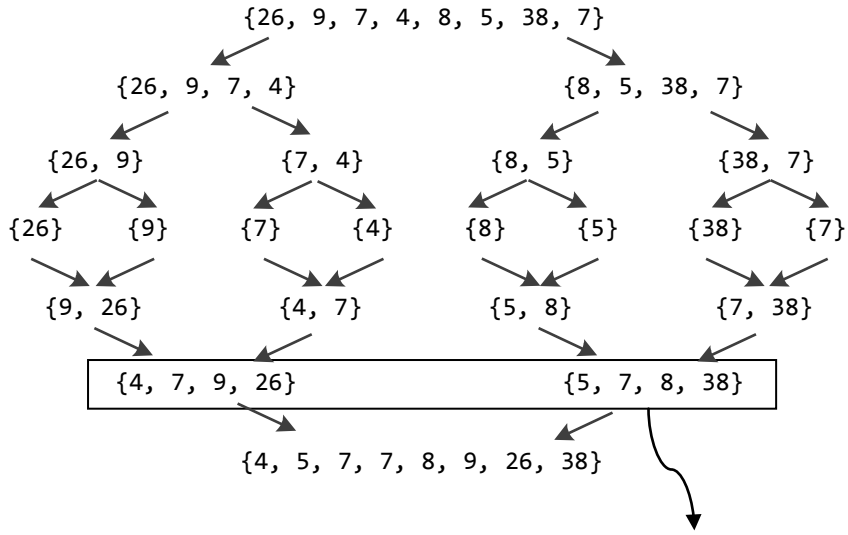
Како што може да се види и во самиот код, процедурата се повикува над низата со дополнителни два параметри кои го определуваат рангот во

низата кој треба да се сортира. Ова значи дека иницијалното повикување на оваа процедура е со 0 како почеток на рангот и *n-1* како крај на рангот, ако низата има *n* елементи.

Алгоритмот е едноставен, но вистинскиот проблем кај него е постапката како да се спојат две понизи во една. Постапката може да се изведе во оригиналната низа, но таа е многу поедноставна ако се користи помошна низа во која ќе се сместуваат резултатите од спојувањата, а потоа да се ископира во оригиналната низа. Процедурата е дадена со кодот:

```
void Spoj(Niza &A, int pocetok, int sredina, int kraj) {
	Niza B;
	int i = pocetok, j = sredina + 1, indNov = pocetok;
	while ((i <= sredina) && (j <= kraj))
		if (A[i] <= A[j])
			B[indNov++] = A[i++];
		else
			B[indNov++] = A[j++];
	if (i > sredina)
		for(int k = j; k <= kraj; k++)
			B[indNov++] = A[k];
	else
		for(int k = i; k <= sredina; k++)
			B[indNov++] = A[k];
	for(int k = pocetok; k <= kraj; k++)
		A[k] = B[k];
}
```

Постапката се изведува едноставно со повикување на целата низа која се спојува определена со ранг дефиниран со две позиции, и дополнителна позиција на елементот кој е граничен елемент меѓу двете поднизи т.е. прв елемент на втората подниза. Се поставуваат два индекси на почетоците на двете низи (*i* и *j* во кодот) и се споредуваат вредностите на елементите кои се наоѓаат на овие две позиции. Како се додава елемент во привремената низа од една од двете поднизи, така се инкрементира индексот од соодветната подниза. Ова трае додека една од двете поднизи не се испразни. Тогаш, остатокот од другата низа само се препишува на крајот од привремената низа, со што влезната низа е сортирана во помошната низа. Ова значи дека по копирање на елементите од привремената низа во оригиналната, постапката завршува. На Слика 7.6 е прикажан пример за сортирање со спојување.



	A[]								B[]							
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
i=0; j=4	4	7	9	26	5	7	8	38								
i++	4	7	9	26	5	7	8	38	4							
j++	4	7	9	26	5	7	8	38	4	5						
i++	4	7	9	26	5	7	8	38	4	5	7					
j++	4	7	9	26	5	7	8	38	4	5	7	7				
j++	4	7	9	26	5	7	8	38	4	5	7	7	8			
i++	4	7	9	26	5	7	8	38	4	5	7	7	8	9		
i++	4	7	9	26	5	7	8	38	4	5	7	7	8	9	26	
j++	4	7	9	26	5	7	8	38	**4**	**5**	**7**	**7**	**8**	**9**	**26**	**38**

Слика 7.6. Пример за сортирање со спојување

Од аспект на времето на извршување, очигледно е дека процедурата за спојување има време на извршување пропорционално со должината на двете низи, па сложеноста е *O(kraj – pocetok)* т.е. *O(n)* каде *n* е должина на целата низа која се спојува со оваа процедура. Понатаму, секој рекурзивен повик на алгоритмот има време на извршување пропорционално со должината на низата која се произведува со него. Множеството на сите рекурзивни повици може да се прикажат со бинарно стебло слично како на Слика 7.6, каде може да се забележи дека сите рекурзивни повици на исто ниво од стеблото работат со низи чиј збир на елементи е точно *n* т.е. должината на иницијалната низа. Ова значи дека ако се соберат времињата на извршување

на спојувањата на едно ниво се добива точно *O(n)*. Со оглед на тоа што има вкупно $log_2\ n+1$ нивоа на бинарното стебло, вкупната сложеност на алгоритмот во најлош случај е $O(n)\times(\log_2 n+1)=O(n \log n)$. Од оваа оценка може да се заклучи дека станува збор не само за еден од најбрзите алгоритми за сортирање, туку и за еден од најпопуларните, иако троши меморија за препишување на елементите при спојувањето.

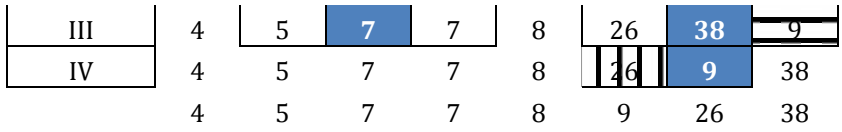
7.6 Брзо сортирање (Quick Sort)

Иако наизглед сортирањето со спојување теоретски има добро време на извршување и има најлоша сложеност од *O(n log n),* овој алгоритам не се користи многу во пракса. Наместо него, повеќето програми за сортирање се базираат на методата со брзо сортирање.

И двете методи ја користат „раздели и владај“ стратегијата. Кај претходниот алгоритам, оригиналната низа се дели на две половини кои се сортираат независно, за да потоа се спојат и дадат резултат. Но што ќе се случи доколку сите елементи од првата половина се помали од најмалиот елемент во втората половина? Во оваа ситуација спојувањето не е потребно, а самото спојување е најкомплицираниот дел од претходниот алгоритам. Според ова, може да се изведе некакво претпроцесирање на низата така што пред да се подели низата ќе се преуредат елементите така што во првата подниза би се сместиле сите елементи помали од најмалиот елемент од втората подниза.

Методата за брзо сортирање го работи токму ова. Имено, овој алгоритам избира еден референтен елемент кој се нарекува оска или стожер (pivot) во однос на кој ќе се подели оригиналната низа на две поднизи така што во едната подниза ќе се сместат сите елементи помали од оската, а во другата ќе се сместат сите елементи поголеми од оската. Пример за работата на алгоритмот за брзо сортирање е даден на Слика 7.7.

	A[]							
Ниво на рекурзија	0	1	2	3	4	5	6	7
I	26	9	7	**4**	8	5	38	7
II	4	9	7	26	**8**	5	38	7
II	4	7	7	26	**8**	5	38	9
III	4	7	**7**	5	8	26	**38**	9



Слика 7.7. Пример за брзо сортирање

Алгоритмот за брзо сортирање може да се имплементира на повеќе начини зависно од изборот на оската. Во примерот од Слика 7.7 оската се избира во средина на низата и овие елементи се означени со затемнето поле. Од левата страна на оската треба да се постават сите елементи помали од неа, а од десната страна сите елементи поголеми од неа. Полињата со вертикална шрафура ги означуваат елементите од левата подниза кои треба да се заменат со некој елемент од десната подниза, а елементите означени со хоризонтална шрафура се елементи од десната подниза кои треба да се заменат со елементи од левата подниза. Како што може да се види и во примерот на првото ниво на рекурзија, оската е најмала во целата низа, па затоа од нејзината десна страна нема избран елемент, затоа што индексот за десната подниза се поставува на самата оска. Според ова, се врши замена на елементите 26 и 4, а потоа се повикува рекурзивно сортирање на поднизата составена од сите елементи освен првиот. Како што се гледа, после секој рекурзивен повик, има елементи кои доаѓаат на својата конечна позиција – елементите без означена ќелија. Во второто ниво на рекурзија, се врши избор на нова оска, па според неа се врши замена на елементите 9 и 7, па замена на 26 и 5. Во тој момент, оската се фиксира, и се врши рекурзивно повикување на алгоритмот за две поднизи, и така до конечно сортирање на целата оригинална низа. Алгоритмот за брзо сортирање со избор на средниот елемент на низата како оска е имплементиран со следниот код во C++.

```
void QuickSort(Niza &A, int pocetok, int kraj) {
	int i=pocetok, j=kraj;
	int oska = A[(pocetok+kraj)/2];
	while (i <= j) {
		while (A[i] < oska) i++;
		while (A[j] > oska) j--;
		if (i <= j) {
			Zamena(A[i], A[j]);
			i++;
			j--;
			}
		}
	if (pocetok < j) QuickSort(A, pocetok, j);
	if (kraj > i) QuickSort(A, i, kraj);
}
```

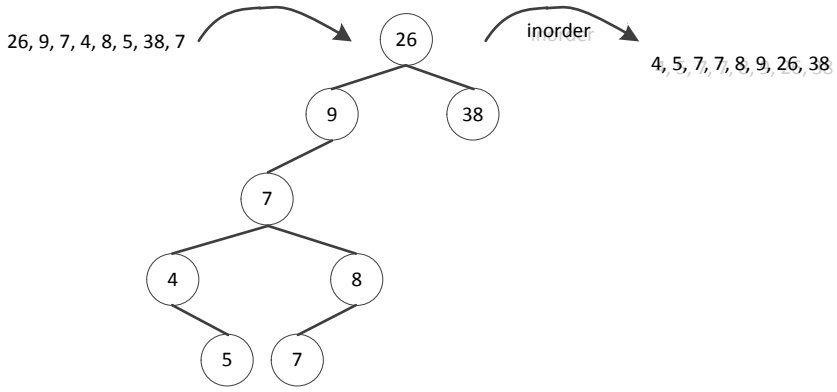
Слично како и кај сортирањето со спојување, повикувањето на алгоритмот се врши со означување на рангот во кој се сортира низата, т.е. ако треба да се сортира целата низа, се повикува со аргументи 0 за почетокот и *n-1* за крајот на низата, ако низата има *n* елементи.

Доколку овој алгоритам ја избира оската во близина на средината на низата, постапката за разбивање на низата, ја дели низата на приближно еднакви низи. Во спротивно, доколку оската се бира кон краевите на низата, концептот „раздели и владај“ паѓа во вода, затоа што разликата во должината на двете поднизи е голема. Во низа со случајно избрани елементи, методата за брзо сортирање се извршува солидно, со просечна сложеност од $O(n \log n)$, додека во најлошото сценарио, кога на влез има веќе сортирана низа, перформансите паѓаат дури до $O(n^2)$. Ако се отфрли инфериорното однесување на алгоритмот во најлошото сценарио, тој е многу побрз во пракса од повеќето други алгоритми за сортирање, што е причина за негов избор како стандард за општите процедури за сортирање.

7.7 Сортирање со бинарно пребарувачко стебло (Tree Sort)

Овој алгоритам се базира на апстрактниот податочен тип бинарно пребарувачко стебло кој беше обработен во поглавје 2.2.3. Значи, во бинарно пребарувачко стебло по ред се сместуваат елементите од низата која се сортира, а откако ќе се сместат сите, се врши исчитување на стеблото назад во оригиналната низа со методата *INORDER*.

На Слика 7.8 може да се види пример за сортирање со помош на бинарно стебло. Елементите прво се сместуваат во стеблото, а потоа стеблото се исчитува со *INORDER*.



Слика 7.8. Пример за сортирање со бинарно стебло

Алгоритмот за сортирање со бинарно пребарувачко стебло е даден со следниот код во C++. Важно е да се напомене дека овој код за да биде функционален мора да биде придружен со кодот за имплементација на АПТ бинарно пребарувачко стебло.

```
void TS_InOrder(BSsteblo *BS, Niza &A, int &N) {
       if (BS==NULL) return;
       TS_InOrder(LevoDete(BS), A, N);
       A[N++] = Vrednost(BS);
       TS_InOrder(DesnoDete(BS), A, N);
}

void TreeSort(Niza &A, int N) {
       if (N==0) return;
       BSsteblo *B = KreirajSteblo(A[0]);
       for(int i=1; i<N; i++)
              Dodadi(A[i], B);
       int n = 0;
       TS_InOrder(B, A, n);
}
```

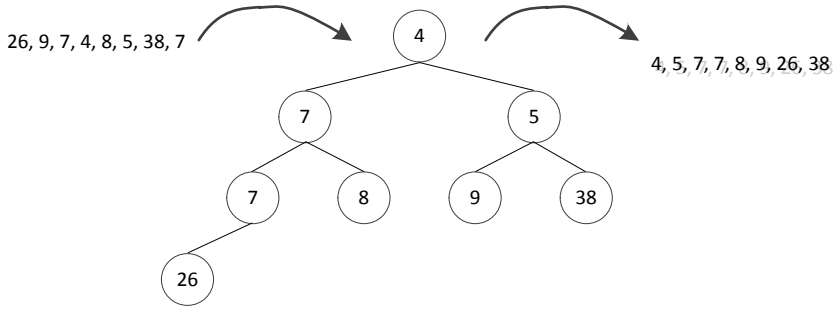
Ако податоците во низата што треба да се сортира се добро измешани, тогаш се добива добро разгрането бинарно пребарувачко стебло, т.е. неговата висина ќе биде приближно $log_2 n$, каде n е бројот на елементите во низата. Со оглед на тоа што секоја операција за вметнување во стеблото бара време пропорционално на висината на стеблото и со оглед на тоа што има вкупно n вметнувања, процесот на градење на самото бинарно стебло бара време $O(n \log n)$. Бидејќи *INORDER* методата има линеарна сложеност т.е. $O(n)$, тогаш вкупното време на извршување на целиот алгоритам е $O(n \log n)$, но ова е само просечното време. Најлошото сценарио е кога низата е веќе

сортирана, па во тој случај се добива косо стебло со висина *n*, па целиот алгоритам бара $O(n^2)$ време на извршување.

7.8 Сортирање со куп (Heap Sort)

Слично и како во претходниот алгоритам, и АПТ куп (Поглавје 2.2.5) може да се употреби за сортирање на низи. Најпрвин, податоците од низата се сместуваат во купот, а потоа со користење на функцијата *IzbrisiNajmal()*, која го брише најмалиот елемент од структурата и ја враќа неговата вредност, се вадат сите елементи од купот, сè додека е непразен.

Овој алгоритам се нарекува сортирање со куп (Heap Sort) и на Слика 7.9 е даден пример за сортирање со него.



Слика 7.9. Пример за сортирање со куп

Алгоритмот за сортирање со помош на овој АПТ е даден со следниот код. И овој код, како и во претходниот случај, мора да биде придружен со имплементацијата на АПТ куп.

```
void HeapSort(Niza &A, int N) {
       if (N==0) return;
       Kup B; Init(B);
       for(int i=0; i<N; i++) Dodadi(A[i], B);
       N = 0;
       while (!Prazno(B))
              A[N++] = IzbrisiNajmal(B);
}
```

Од аспект на сложеноста, алгоритмот се сведува на *n*-кратна примена на вметнување на елемент во АПТ куп, т.е. негово отфрлање. Со оглед на тоа што едно уфрлање т.е. отфрлање има сложеност $O(\log n)$, времето за сортирање на целата низа во најлош случај е $O(n \log n)$. Ова значи дека овој алгоритам е еден од најбрзите познати алгоритми за сортирање кој успешно сортира и големи низи.

8. Динамичко програмирање

Динамичко програмирање е моќна техника за ефикасно решавање на рекурзивни проблеми, но тоа не е тоа многу пати, кога е најден основен алгоритам со динамичко програмирање, може да се најдат дополнителни подобрувања на самиот алгоритам за да се намали времето на извршување или користењето на меморијата. Ваков пример е подобрувањето на рекурзивниот алгоритам за низата на Фибоначи. Имено, експоненцијалниот алгоритам на Фибоначи (Поглавје 4.1.1) е рекурзивен проблем кој ја следи дефиницијата на низата, но тој може да се подобри со користење на техники на динамичко програмирање за добивање на полиномен алгоритам (Поглавје 4.1.2), т.е. елиминација на повторување на пресметки. Според ова, може да се заклучи дека, динамичко програмирање е метода која бара сите потпроблеми да се решат и нивните решенија да се сместат во соодветна податочна структура. Кога алгоритмот ќе побара некое решение на друг потпроблем, тоа не се пресметува повторно, туку се користи неговиот резултат од меморија, доколку се наоѓа таму.

Методата „раздели и владај“ знае да резултира со неефикасни алгоритми затоа што бројот на потпроблемите кои треба да се решат расте експоненцијално со големината на поставениот проблем. На овој начин, може бројот на различни потпроблеми и да не е толку голем, но еден ист проблем може да се појави на повеќе места во рекурзијата, па повторно треба да се решава (што беше случајот и со низата на Фибоначи). Токму на ваквите случаи како одговор се наметнува динамичкото програмирање.

Гореспоменатата метода „раздели и владај“ ги решава проблемите од горе надолу т.е. од поголемите проблеми кон помали, но методата со динамичко програмирање одат од долу нагоре т.е. прво во меморијата се сместуваат решенија на проблемите со најмала големина, па поголемите и сè така до големината на поставениот проблем.

Секако, и оваа метода има свој недостаток, а тоа е дека оваа постапка понекогаш бара решение и на потпроблеми кои и нема да бидат потребни за решавањето на крајниот проблем, но секако во споредба со повторното пресметување на потпроблеми, овој недостаток е далеку поисплатлив.

Динамичкото програмирање всушност претставува рекурзија без повторување. Развојот на ваквите алгоритми секогаш се одвива во две фази: рекурзивна формулација на проблемот и градење на решенијата на рекурзијата од долу нагоре. Последното се одвива едноставно во неколку чекори, и тоа:

- алгоритмот се започнува со тривијалните случаи на рекурзијата, се идентификуваат потпроблемите и различните начини на кои алгоритмот може да се повика самиот себеси;
- се избира податочната структура во која ќе се меморираат меѓурезултатите. Најчесто, рекурзивните потпроблеми се идентификуваат со нумерички податоци, па затоа и најчесто се користат полињата од броеви, но ако тоа не е случај, се дефинира соодветна податочна структура.
- се врши анализа на времето и меморискиот простор, при што бројот на различни можни потпроблеми ја одредува мемориската сложеност, додека за временската сложеност се собираат времињата на извршување на сите можни потпроблеми. Секако, последното се врши така што не се сметаат рекурзивните повици, на пример ако се познати F_{i-1} и F_{i-2} кај низата на Фибоначи, F_i се пресметува во $O(1)$ време, што значи дека времето за наоѓање на n-тиот број на Фибоначи е $O(n)$.
- се наоѓаат зависностите меѓу потпроблемите, освен за тривијалните случаи, и според нив се утврдува соодветен редослед на инструкциите така што секој потпроблем треба да дојде по решавање на потпроблемите од кои тој потпроблем зависи.

Динамичкото програмирање ги сместува решенијата на потпроблемите во некаква податочна структура. Често оваа структура е табела или низа, што придонесува до погрешно размислување на програмерите да се фокусираат на табелата т.е. низата, затоа што овие структури се едноставни, наместо да се фокусираат на поважниот дел – наоѓање на правилна рекурзија. Затоа, мора да се води сметка дека динамичкото програмирање не се однесува на полнење на табели, туку на паметна рекурзија.

8.1 Растојание на Levenshtein

Растојание на Levenshtein (попознато како уредувачко растојание - Edit Distance) меѓу два збора е минималниот број на вметнувања, бришења или замени на букви, потребни за трансформација на едниот збор во другиот. На пример, минималното растојание меѓу зборовите „мед“ и „раце“ е 4:

мед → ме_ → ц̲е → а̲це → р̲аце

Но, во општ случај проблемот и не е толку едноставен, така на пример „алгоритам“ и „милиграм“ е 6, што може да се види на Слика 8.1, но дали ова решение е најоптималното?

чекор	A[]									операција
	0	1	2	3	4	5	6	7	8	
0	А	Л	Г	О	Р	И	Т	А	М	бришење
1		А	Л	Г	О	Р	И	А	М	бришење
2			А	Л	Г	О	Р	А	М	бришење
3				А	Л	Г	Р	А	М	вметнување
4			А	Л	И	Г	Р	А	М	замена
5			И	Л	И	Г	Р	А	М	вметнување
6		М	И	Л	И	Г	Р	А	М	

Слика 8.1. Пример за растојание на Levenshtein

Овој проблем се решава со техниката на динамичко програмирање, па за оваа цел потребна е рекурзивна дефиниција на проблемот. На Слика 8.2 е прикажана репрезентација на влезните параметри од претходниот пример, со приказ на празни места.

	А	Л		Г	О	Р	И	Т	А	М
М	И	Л	И	Г		Р			А	М

Слика 8.2. Репрезентација на влезни параметри со празни места кај алгоритмот за растојание на Levenstein

Ако ги споредиме двете редици од табелата на сликата може да се види дека ако се елиминираат колоните во кои и двете редици се еднакви ќе се добијат точно 6 колони, колку што треба да биде и резултатот. До овој резултат може да се добие ако се разгледуваат двете редици од крајот кон почетокот. Имено, ако се елиминираат последните две колони, резултатот кој се бара ќе биде истиот како и без овие две колони, па според ова може да се направи рекурзивната дефиниција.

Нека двата збора за кои се пресметува растојанието се *X[1..m]* и *Y[1..n]*. Растојанието може да се означи со *Rastojanie(X[1..m], Y[1..n])*, па во зависност од состојбата со последната колона од табелата од сликата може да се добијат три можни случаи, и тоа кога:

- последниот елемент од првата редица е празен, при што резултатот ја има вредноста *Rastojanie(X[1..m], Y[1..n-1])+1*. Ова е случај кога се врши трансформација од првиот кон вториот збор, така што во овој случај таа трансформација е вметнување. Инкрементацијата на резултатот за 1 е токму поради пребројување на оваа трансформација.
- последниот елемент од втората редица е празен. Во овој случај се работи за бришење на буква, па слично како и во претходниот случај

се додава 1 кон резултатот, кој во ваков случај ја има вредноста *Rastojanie(X[1..m-1], Y[1..n])+1*.

- двете редици имаат букви во последната колона, при што овој случај се дели на два подслучаи и тоа кога:
  - буквите се исти, при што резултатот не се менува кога би се отфрлила последната колона т.е. резултатот е *Rastojanie(X[1..m-1], Y[1..n-1])*
  - буквите се различни, при што настанува замена на буква т.е. растојанието е *Rastojanie(X[1..m-1], Y[1..n-1]) + 1*

Според разгледаните случаи, најкраткото растојание меѓу зборовите *X* и *Y* е дадено со следната рекурзивна формула:

$$Rastojanie(X[1..m],Y[1..n]) =$$

$$\begin{cases} m, & n=0 \\ n, & m=0 \\ \min\left\{\begin{array}{c} Rastojanie(X[1..m-1],Y[1..n])+1, \\ Rastojanie(X[1..m],Y[1..n-1])+1, \\ Rastojanie(X[1..m-1],Y[1..n-1])+(\text{int})(X[m]!=Y[n]) \end{array}\right\} & \textit{инаку} \end{cases}$$

Рекурзивната формула лесно може да се искористи за креирање на алгоритам во C++, па со следниот код е дадена оваа имплементација.

```
int Rastojanie(std::string X, std::string Y) {
       int xd = X.length(), yd = Y.length();
       if (xd == 0) return yd;
       if (yd == 0) return xd;
       std::string xl = X.substr(0,xd-1), yl = Y.substr(0,yd-1);
       char xt = X[xd-1], yt = Y[yd-1];
       int min = Rastojanie(xl, Y) + 1;
       int min2 = Rastojanie(X, yl) + 1;
       if (min > min2) min = min2;
       min2 = Rastojanie(xl, yl);
       if (xt != yt) min2++;
       if (min > min2) min = min2;
       return min;
}
```

Времето на извршување лесно може да се преслика од самата дефиниција на проблемот т.е.

$$T(m,n)=\begin{cases} O(1) & n=0 \vee m=0 \\ T(m,n-1)+T(m-1,n) & \\ +T(m-1,n-1)+O(1) & \textit{инаку} \end{cases}$$

Времето на извршување на најлошиот случај може да се добие со воведување на функцијата

$$T'(N)=\max_{n+m=N} T(m,n)$$

од која се добива

$$T'(N)=\begin{cases} O(1) & N=0 \\ 2T(N-1)+T(N-2)+O(1) & \textit{инаку} \end{cases}$$

Со методата на анулатор кај диференцијалните равенки[2], се добива дека

$$T'(N)=O((1+\sqrt{2})^N)$$

што значи дека времето на извршување на алгоритмот е најмногу

$$T(m,n)=O((1+\sqrt{2})^{m+n})$$

Овој алгоритам е особено спор. Слично како и кај низата на Фибоначи, има многу потпроблеми кои се решаваат повеќекратно, па како и претходно, и во овој случај може да се внесе меморирање на резултатите од потпроблемите и нивно искористување при повторените повици. Токму овој пристап го имплементира динамичкото програмирање.

Очигледно е дека аргументите на потпроблемите од претходната имплементација се секогаш префикси на оригиналните зборови *X* и *Y*. Според тоа, ознаките можат да се упростат само со користење на должините на овие префикси кај рекурзивните повици, наместо самите префикси, па според тоа може да се забележи со *R(i, j)* наместо *Rastojanie(X[1..i], Y[1..j])*, па горната формула може да се изрази со новата формулација:

[2] ОВА ДА СЕ ИСПИТА И РЕФЕРЕНЦИРА

$$R(i,j)=\begin{cases} i, & j=0 \\ j, & i=0 \\ \min\begin{Bmatrix} R(i-1,j)+1 \\ R(i,j-1)+1 \\ R(i-1,j-1)+(\text{int})(X[i]!=Y[j]) \end{Bmatrix} & \textit{инаку} \end{cases}$$

Како што може да се види во дефиницијата, секој потпроблем се идентификува со два индекси, за секој збор одделно. Ова значи дека меѓурезултатите може да се сместат во матрица со димензии *m*×*n*. Од прва, не е баш најјасно како рекурзијата ја пополнува матрицата, но јасно е дека елементите се пополнуваат по пополнување на нивните соседни елементи во редицата над нив и во колоната на лево од нив. Ова е сосема доволно да експлицитно да се утврди редоследот на пресметување, имено, ако се пополнува матрицата редица по редица од горе надолу, и секоја редица од лево кон десно, тогаш елементот кој во еден момент треба да се пресмета зависи од веќе пресметаните елементи.

Со користење на ова подобрување се добива алгоритмот со динамичко програмирање кој користи матрица *R*, за секој потпроблем за кој се врши рекурзивен повик. Имплементацијата на овој алгоритам е дадена со следниот код.

```
int RastojanieDP(std::string X, std::string Y) {
	int R[50][50];
	int min, min2;
	int m = X.length(), n = Y.length();
	for(int i=1; i<=n; i++)
		R[0][i] = i;
	for(int i=1; i<=m; i++)
		R[i][0] = i;
	for(int i=1; i<=m; i++)
		for(int j=1; j<=n; j++) {
			if (R[i-1][j] < R[i][j-1])
				min = R[i-1][j] + 1;
			else
				min = R[i][j-1] + 1;
			min2 = R[i-1][j-1];
			if (X[i-1]!=Y[j-1]) min2++;
			if (min > min2) min = min2;
			R[i][j] = min;
```

```
            }
        return R[m][n];
}
```

На Слика 8.3 е дадена матрицата која се користи кај овој алгоритам за зборовите „алгоритам“ и „милиграм“. Лесно може да се забележи дека означените полиња се единствените полиња во целата матрица за кои при пресметување на минимумот во рекурзијата не се инкрементирал елементот кој се користи за пресметување на соодветното поле т.е. овие полиња ги означуваат буквите кои се појавуваат и во двата збора.

		А	Л	Г	О	Р	И	Т	А	М
	0	1	2	3	4	5	6	7	8	9
М	1	1	2	3	4	5	6	7	8	**8**
И	2	2	2	3	4	5	**5**	6	7	8
Л	3	3	**2**	3	4	5	6	6	7	8
И	4	4	3	3	4	5	**5**	6	7	8
Г	5	5	4	**3**	4	5	6	6	7	8
Р	6	6	5	4	4	**4**	5	6	7	8
А	7	**6**	6	5	5	5	5	6	**6**	7
М	8	7	7	6	6	6	6	6	7	**6**

Слика 8.3. Матрица за пресметување на растојание меѓу АЛГОРИТАМ и МИЛИГРАМ

Лесно може да се воочи дека времето на извршување на овој алгоритам е *O(nm).*

8.2 Проблем на пакување ранец

Еден човек се шетал низ шума и наишол на богатство. Богатството било составено од n елементи, така што секој елемент имал своја тежина t_i и имал ознака за неговата вредност v_i. Човекот со себе имал ранец кој бил ограничен во однос на тежината која може да ја понесе, означена со C. Проблемот на пакување ранец се состои во тоа како да се спакува највредната комбинација на елементите од богатството во ранецот така што нивната вкупна тежина нема да ја надмине дозволената тежина на ранецот.

Нека $C=20$ и нека се дадени пет елементи со следните карактеристики:

Елемент	1	2	3	4	5
Тежина	8 кг	5 кг	6 кг	3 кг	2 кг
Вредност	40 ден	24 ден	22 ден	12 ден	9 ден

Слика 8.4. Проблем на пакување ранец со 5 елементи

Постојат две верзии на проблемот:

- со повторување, при што секој елемент го има во неограничен број, па во овој случај решението би било да се земат 2 парчиња од елементот 1 и 2 парчиња од елементот 5, при што вкупната вредност би била 2 × 40 + 2 × 9 = 98 ден, со вкупна тежина од 20 кг
- без повторување, при што секој елемент го има само еднаш, при што решението е да се земат елементите 1, 2 и 3, при што вкупната вредност би била 40 + 24 + 22 = 86 ден, со вкупна тежина од 19 кг

Двете верзии се претставени одделно.

8.2.1 Верзија со повторување

Како и секогаш, главното прашање кај динамичкото програмирање е кои се потпроблемите. Во овој случај за да се разложи поставениот проблем постојат два начина, да се разгледуваат помали капацитети на ранецот $t \le C$ или да се разгледуваат помалку елементи, на пример елементите $1..j$ каде $j \le n$. За да се види кој пристап е добар се испробуваат двата случаи.

Според првата рестрикција се дефинира $V(c)$ како максималната вредност во ранец со капацитет c. За да се разложи проблемот на потпроблеми се тргнува од ситуацијата ако оптималната вредност на $V(c)$ го вклучува елементот i, тогаш со негова елиминација од ранецот тој останува со оптимално решение $V(c-t_i)$ или со други зборови $V(c)$ за некое i ја има вредноста $V(c-t_i)+v_i$. Поради тоа што не се знае i-то треба да се испробаат сите варијанти, па решението може да се дефинира со формулата $V(c) = \max_{i:t_i \le c}\{V(c-t_i)+v_i\}$, каде како специјален случај се зема по дефиниција дека максимумот од празно множество е 0. Според оваа формула може да се напише и алгоритмот кој е претставен со следниот код.

```
#define MAX_DOLZINA 100
typedef struct {
	int tezina;
	int vrednost;
} Element;

typedef Element Bogatstvo[100];

int Ranec1(Bogatstvo B, int n, int C) {
	int V[MAX_DOLZINA]; V[0] = 0;
	for(int c=1; c <= C; c++) {
		int max = 0;
		for(int i = 0; i < n; i++) {
```

```
                        int vr = 0;
                        if (c >= B[i].tezina)
                                vr = V[c-B[i].tezina]+B[i].vrednost;
                        if (max < vr)
                                max = vr;
                }
                V[c] = max;
        }
        return V[C];
}
```

Овој алгоритам на влез прима низа од целобројни парови (t_i, v_i), број кој ја означува должината на низата n и капацитетот на ранецот C. Од друга страна пополнува еднодимензионална низа со должина $C+1$, од лево кон десно. Секој елемент може да потроши најмногу $O(n)$ време за негово пресметување, па вкупното време на извршување е $O(nC)$. Низата која го дава резултатот за влезот од Слика 8.4 е следната:

0, 0, 9, 12, 18, 24, 27, 33, 40, 42, 49, 52, 58, 64, 67, 73, 80, 82, 89, 92, 98

а резултатот е последниот елемент т.е. 98.

8.2.2 Верзија без повторување

Во оваа верзија на проблемот, потпроблемите од претходната постапка се неупотребливи. Така на пример ако се знае дека $V(c - t_n)$ е многу големо не помага затоа што не се знае дали елементот n бил или не бил употребен за решавање на овој потпроблем. Според ова, мора да се вклучат некои дополнителни информации за елементите кои се користат. Се додава дополнителен параметар $0 \le j \le n$ така што $V(c, j)$ е максималната вредност која може да се добие за ранец со капацитет c и елементите $1..j$, а вредноста која се бара е $V(C, n)$.

За да се разложи еден потпроблем $V(c, j)$ на помали потпроблеми се одлучува дали елементот j е потребен за достигнување на оптималната вредност или не. Одлучувањето се состои во формулата $V(c, j) = \max\{V(c - t_j, j-1) + v_j, V(c, j-1)\}$ од која може да се изведе и алгоритмот.

```
int Ranec2(Bogatstvo B, int n, int C) {
        int V[MAX_DOLZINA][ MAX_DOLZINA];
        for (int i = 0; i <= C; i++)
                V[i][0] = 0;
        for (int i = 0; i <= n; i++)
                V[0][i] = 0;
        for(int j = 0; j <= n; j++)
```

```
                for(int c=1; c <= C; c++)
                        if (B[j].tezina > c)
                                V[c][j] = V[c][j-1];
                        else {
                                int max = V[c][j-1];
                                int max2 = V[c - B[j].tezina][j-1]
                                                + B[j].vrednost;
                                if (max < max2)
                                                max = max2;
                                V[c][j] = max;
                        }
        return V[C][n];
}
```

Овој алгоритам пополнува матрица со *C+1* редици и *n+1* колона, при што секој елемент од матрицата бара константно време, па дури и матрицата да е многу поголема од претходниот случај, времето на извршување е исто како и во претходниот алгоритам т.е. *O(nC)*.

9. Алгоритми со алчен (greedy) пристап

Оптимизациски проблем е проблем за кој не се бара само решение, туку најдоброто решение. Понекогаш, „алчните“ алгоритми работат добро за оптимизациските проблеми. Ваквите алгоритми работат во фази, при што во секоја фаза се избира најдобриот пат, без да се мисли што ќе се случи понатаму. Од друга страна, постои надеж дека локалниот оптимум на секој чекор ќе заврши во глобален оптимум.

Човекот секојдневно е опкружен со проблеми кои ги решава најчесто со алчен алгоритам. Пример за тоа е кога патува на работа. Го избира најкраткиот пат, без да размисли дали може тој пат да биде блокиран поради густ сообраќај.

Алчните алгоритми се базираат на следните елементи:

- Множество кандидати, од кое се креира решението
- Функција за избор, која го избира најдобриот кандидат за додавање во решението
- Функција за изводливост, која проверува дали кандидатот може да придонесе во креирањето на решението
- Објективна функција, која доделува вредност на решението т.е. парцијалното решение
- Функција за решение, која покажува дека е постигнато комплетно решение

Алчните алгоритми даваат добри решенија за некои математички проблеми, но за некои воопшто не се погодни. Во еден момент, може да се направи било каков избор кој изгледа дека е најдобар, а потоа да се решаваат потпроблемите кои подоцна се јавуваат. Изборот може да зависи од претходните избори, но не и од наредните и итеративно се вршат избори еден за друг, при што секој проблем се сведува на помал. Ова значи дека алчниот алгоритам никогаш не ги преиспитува своите избори.

Често оваа стратегија може да биде погрешен избор кога се решава некој проблем. На пример, ако се примени овој алгоритам кај шахот, секогаш ќе се избира потез кој во моментот на избор ќе изгледа најдобро, без да размислува за неговите последици.

9.1 Проблем на монети

Класичен пример за алгоритам со алчен пристап е уште еден секојдневен пример т.е. проблемот со монети. Нека на располагање се

монети од 1, 2, 5, 10 и 50 денари, а треба да се соберат вкупно 73 денари. Инстинктивно се прави избор 1×50 + 2×10 + 1×2 + 1×1 = 73при што не само што е собран точниот износ, туку и е најдено најоптималното решение т.е. најкратката низа. Алгоритмот работи така што се бира најголемиот износ кој ја има една монета, а кој не преминува преку 73. Тоа е 50, се додава во листата на избори, а потоа го применува истиот алгоритам над остатокот 23. Но како што беше напоменато, не секогаш се добива оптимално решение со овој алгоритам. Што ако монетите се составени од 1, 6 и 13 денари, а треба да се соберат 18 денари. Тогаш алгоритмот ќе издвои една монета од 13 денари, а потоа поради тоа што со 6 денари би се надминал вкупниот износ, се избираат 5 монети од по 1 денар. Ова решение дава низа од 6 монети, иако постои оптимално решение од 3 монети од по 6 денари. Имплементација во C++ за овој алгоритам е дадена во следниот код:

```
#define MAX_DOLZINA 100
typedef int Niza[MAX_DOLZINA];

void GreedyMoneti(int Iznos, Niza Moneta, int BrojMoneti,
                                    Niza &Rezultat, int &M) {
       if (Iznos == 0) return;
       int i = BrojMoneti - 1;
       int k = Moneta[i];
       while (k>Iznos) k = Moneta[--i];
       Rezultat[M++] = k;
       GreedyMoneti(Iznos-k, Moneta, BrojMoneti, Rezultat, M);
}
```

Процедурата на влез го прима износот кој треба да се добие, низата од расположливи монети сортирана во растечки редослед со вкупна должина *BrojMoneti,* низата за сместување на резултатот, како и бројач кој треба да преброи колку вкупно монети ќе го дадат решението (должина на резултантната низа).

Како што беше напоменато претходно овој алгоритам не дава оптимално решение во општ случај, а има случаи кога алгоритмот може и да не се изврши при прецизно избран систем на монети. Таков е случајот со систем во кој има парички од 5, 7 и 8 денари, а треба да се соберат 43 денари. Решение постои (8+8+8+7+7+5=43), но до него овој алгоритам не може да дојде. Според ова, може да се заклучи дека постојат проблеми кај кои алчниот пристап дава коректен алгоритам, но ова треба да се докаже, а постојат и проблеми кај кои алчниот алгоритам не е потполно коректен, но претставува добра евристика, што треба експериментално да се потврди. Оттука може да се заклучи дека овој алгоритам е само добра илустрација на „алчниот“

пристап за постоечките системи на монети, но во никој случај не смее да се користи за општи системи на монети.

9.2 Проблем на пакување ранец со деливи елементи

Овој проблем на пакување ранец е сличен како и претходниот проблем, само што во овој случај секој од елементите може да се подели, па ако еден елемент не го собира цел во ранецот, тој може да се подели и во ранецот да се смести само негов дел.

Проблемот формално може да се зададе на следниот начин: ако е даден природниот број n и позитивни реални броеви C, t_i, v_i, каде $i=1..n$, да се најдат реалните броеви x_i, каде $i=1..n$, такви што $\sum_{i=1}^{n} v_i x_i$ е максимално, при што $\sum_{i=1}^{n} t_i x_i \le C$ и $0 \le x_i \le 1$, за $i=1..n$.

Оваа формулација може да се интерпретира на следниот начин:

- C е капацитетот на ранецот
- t_i е тежината на i-тиот елемент
- v_i е вредноста на i-тиот елемент
- x_i е процентот на учество на i-тиот елемент во пакувањето

Алчниот алгоритам за решавање на овој проблем се состои во пресметување на профитабилноста на секој од елементите $p_i = v_i / t_i$ за $i=1..n$, а потоа низата се сортира во опаѓачки редослед и по тој редослед се пакуваат елементите во ранецот додека има место. При секое сместување во ранецот се настојува да се смести што е можно поголем дел од елементот кој се пакува. Имплементацијата на овој алгоритам е дадена со следниот код.

```
double GreedyRanec(Bogatstvo A, int n, double C) {
       Bogatstvo B;
       for (int i=0; i< n; i++) B[i] = A[i];
       for(int i = 0; i < n-1; i++) {
         int j=i+1;
         while (((double)B[j].vrednost / (double)B[j].tezina >
              (double)B[j-1].vrednost / (double)B[j-1].tezina) &&
              (j>0)) {
                           Element d = B[j];
                           B[j] = B[j-1];
                           B[j-1] = d;
                           j--;
                           }
              }
       double rez = 0;
```

```
        double spakuvano = 0;
        int i = 0;
        while ((C > spakuvano) && (i<=n))
                if (B[i].tezina <= C - spakuvano) {
                        rez += B[i].vrednost;
                        spakuvano += B[i].tezina;
                        i++;
                }
                else {
                        rez += ((C - spakuvano) / B[i].tezina)
                                * B[i].vrednost;
                        spakuvano = C;
                }
        return rez;
}
```

Ако се искористи овој алгоритам над влезните параметри од Слика 8.4, тогаш како решение ќе се добие

1×40 + 1×24 + 1×9 + 1×12 + 0,33×22 = 92,33.

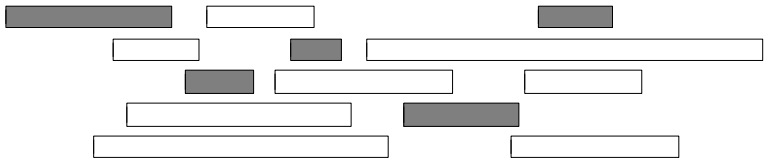
9.3 Проблем на распоредување

Се одржува конференција на која има повеќе паралелни предавања во повеќе сали. Предавањата имаат различни времиња на траење, па според тоа и времето на почнување на секое од предавањата е различно од предавањата во другите сали. Врз основа на овие времиња на почнување и завршување е подготвена програма на конференцијата. Проблемот се состои во наоѓање на избор на предавања кои еден посетител треба да ги слуша со цел да присуствува на што е можно повеќе предавањата и притоа не е дозволено доцнење или порано напуштање на предавањата.

Формално, овој проблем може да се опише со две низи *P[1..n]* и *K[1..n]* кои ги претставуваат времињата на почетокот и крајот на секое предавање. Задачата е да се најде најголемото можно подмножество $X \subseteq \{1, 2 \ldots, n\}$ за кое важи

$$\forall (i, j) \in X \times X, i \neq j, \text{ важи } P[i] > K[j] \text{ или } P[j] > K[i]$$

Проблемот може да се илустрира со Гантов дијаграм т.е. со исцртување на секое предавање како посебна отсечка (или лента) на временската оска која започнува во почетното време, а завршува во крајното време соодветно за тоа предавање. Како што може да се види, на Слика 9.1 е дадена оваа илустрација, а целта е да се најде најголемото множество на ленти кои не се преклопуваат вертикално.



Слика 9.1. Максимално множество на непреклопени предавања

Проблемот може лесно да се реши со помош на рекурзија каде треба да се реши дали да се слуша предавањето 1. Во овој случај нека A_1 е множеството предавања кои завршуваат пред да започне предавањето 1, а B_1 е множеството предавања кои почнуваат по завршувањето на предавањето 1, што формално може да се претстави на следниот начин:

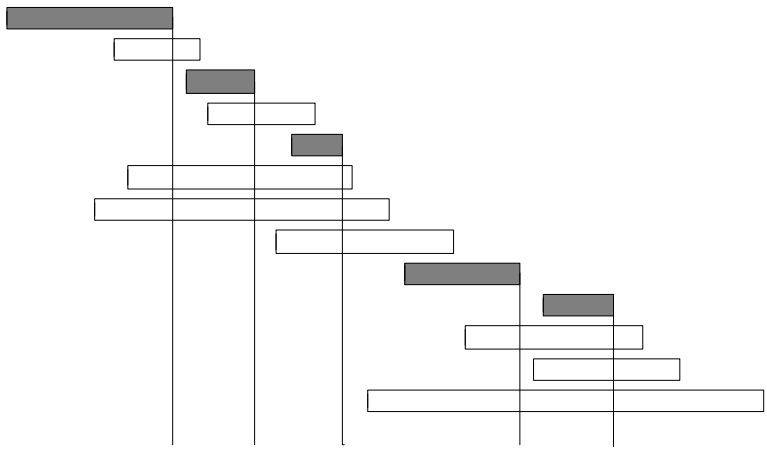
$A_1=\{i \mid 1 < i \leq n \ u \ K[i] < P[1]\}$

$B_1=\{i \mid 1 < i \leq n \ u \ P[i] > K[1]\}$

Ако предавањето 1 е дел од оптималното решение, тогаш и A_1 и B_1 се оптимални, што може да се најде рекурзивно, инаку оптималното решение може да се најде рекурзивно над множеството $X\backslash\{1\} = \{2,..., n\}$. Ова значи дека се испробуваат двете решенија и се избира подоброто. Ова секако се решава со динамичко програмирање, при што овој алгоритам е со сложеност $O(n^2)$ (Како вежба, ова може да се провери).

Овој проблем е доказ дека постојат проблеми кои ако се решат со алчен алгоритам имаат подобри перформанси од алгоритмите со динамичко програмирање. Имено, ако се оди по интуицијата на човекот, изборот на предавањата би се одвивал така што на почеток ќе се избере првото предавање кое треба да заврши, што е логично, затоа што по неговото завршување ќе има најголем број на преостанати предавања врз кои се повторно се применува алчната стратегија.

Ова значи дека овој проблем може многу лесно да се реши на тој начин што прво ќе се сортираат предавањата според времето на завршување и при секоја итерација да се избира предавање кое уште не е почнато, а најрано ќе заврши. Оваа идеја е прикажана на Гантовиот дијаграм на Слика 9.2 , каде се сортирани предавањата од примерот на Слика 9.1 од горе надолу.



Слика 9.2. Предавања сортирани по времето на завршување, наменети за алчниот алгоритам

Овој алгоритам не го дава единствениот максимален распоред, но дефинитивно дава еден. За да се докаже ова тврдење се тргнува од следниот факт кој треба да се докаже, а тоа е дека барем еден максимален распоред без преклопување го вклучува предавањето кое прво завршува. За да се докаже ова тврдење нека g е ова предавање. Нека X е максимален распоред без преклопување кој не го содржи g. Нека h е предавање од X кое прво завршува, па бидејќи g завршува пред h, g не се преклопува со ниту едно предавање од множеството $X \setminus \{h\}$. Ова значи дека распоредот $Y = (X \cup \{g\}) \setminus \{h\}$ е исто така максимален распоред без преклопување.

Но, ова се однесува само на предавањето кое завршува прво. Што е со другите случаи? Нека $\{g_1, g_2, \ldots, g_k\}$ е секвенца од предавања кои се избрани со алчниот алгоритам и нека постои максимален распоред без преклопување од видот $\{g_1, g_2, \ldots, g_{j-1}, h_j, h_{j+1}, \ldots, h_m\}$, каде h_i се предавања кои не се избрани од овој алгоритам. Според дефиницијата на алгоритмот, g_j е неговиот j-ти избор и тој не се преклопува со претходните предавања $\{g_1, g_2, \ldots, g_{j-1}\}$, но ова важи и за h_j. Но според дефиницијата на алгоритмот важи и дека g_j е предавањето кое прво ќе заврши од сите кои не се преклопуваат со претходните, т.е. g_j ќе заврши пред h_j, што значи дека g_j не се преклопува ниту со следните предавања $\{h_{j+1}, \ldots, h_m\}$, па распоредот $\{g_1, g_2, \ldots, g_{j-1}, g_j, h_{j+1}, \ldots, h_m\}$ е без преклопување. Индуктивно, следува дека постои оптимален распоред $\{g_1, g_2, \ldots, g_{k-1}, g_k, h_{k+1}, \ldots, h_m\}$ кој ги вклучува сите предавања избрани со алчниот алгоритам, но ова е можно само ако $k=m$ затоа што ако постои предавање h_{k+1} кое не се преклопува со g_k, алчниот алгоритам би го избрал и него, што ја докажува коректноста на алчниот алгоритам.

Според тоа, алгоритмот има смисла да се имплементира, што е направено со следниот код. Овој алгоритам се извршува за *O(n logn)* време.

```
#include <iostream>
#include <stdio.h>
#include <time.h>

typedef struct {
	time_t pocetok;
	time_t kraj;
} Predavanje;

typedef Predavanje Raspored[100];

void GreedyRaspored(Raspored B, int n, Raspored &R, int &m) {
	Raspored A;
	for(int i=0; i < n; i++) A[i] = B[i];
	for(int i = 0; i < n-1; i++) {
		int j=i+1;
		while ((A[j].kraj < A[j-1].kraj) && (j>0)) {
			Predavanje p = A[j];
			A[j] = A[j-1];
			A[j-1] = p;
			j--;
			}
	}
	m = 1; R[m-1] = A[0];
	for(int i=1; i<n; i++)
		if (A[i].pocetok >= A[m-1].kraj)
			R[m++] = A[i];
}
```