Алгоритми и нивна сложеност

1. Што е алгоритам?
2. Претставување на алгоритмот
3. Својства на алгоритмите
4. Анализа на алгоритмот
5. Ефикасност

Што се алгоритмите? Зошто воопшто се учат алгоритмите? Што е улогата на алгоритмите во споредба со другите технологии користени во компјутерите? Во продолжение, ќе ги одговориме овие прашања.

1. Што е алгоритам?

Зборот алгоритам (Algorithmi) е земен од латинскиот јазик и е даден по името на узбекистанскиот математичар од IX век Мухамед Ал Хорезми(Abu Jafar Mohammed Ibn Musa Al Khowarizmi), кој прв ги формирал правилата за извршување 4 основни операции со арапски цифри.

Неформално, алгоритам е било која добро-дефинирана сметачка процедура, што зема некоја вредност, или множество од вредности како влез и резултира некоја вредност, или множество од вредности како излез.

Решавајќи задачи од областа на математиката, физиката, статистиката и други ние користиме познати правила и методи. Како на пример, правило за множење на два броја, делење на броеви, методи за решавање на систем линеарни равенки и друго.

Исто така ќе го гледаме алгоритмот како алатка за решавање на добро специфициран сметачки проблем.

Алгоритмите се процедури за решавање на одредени проблеми. Многу едноставен пример е проблемот на множење на два броја. Множењето на мали броеви како што се 7 и 9 е тривијално, затоа што можеме да го меморираме одговорот однапред. Но ако множиме големи броеви како што се 1234 и 789 ни треба чекор по чекор процедура или алгоритам. Сите ние во училиште сме учеле некои алгоритми кои биле составени од други помали алгоритми но не сме обрнувале внимание како тие процедури стручно се викаат и од што се составени, туку сме ги памтеле онакви какви што биле. Алгоритмите во сметањето имаат долга историја, можеби исто толку долга колку што постои цивилизацијата воопшто.

И во секојдневието извршуваме работи по некои правила. Ќе наведеме едноставен пример. Проблемот што треба да се реши е да се направи чоколадна торта по рецепт.

Рецепт	
Чоколадна торта	
400 гр чоколада	3 јајца
1 маргарин	1 пакетче ванила
2 чаши шеќер	1 чаша брашно

Стопи ги чоколадата и маргаринот.Истури го шеќерот во стопеното чоколадо и измешај.Потоа истури ги јајцата и ванилата и измешај.Истури го и брашното и измешај.Измешаното истури о во плех.Се пече на 250 степени околу 40 минути или додека вилушката кога ќе ја забодеме во тестото и извлечеме ќе биде скоро чиста.Се остава да се излади и потоа се јаде.

Програмскиот код за овој проблем е:

променливи

```
cokolada,margarin,seker,jajca,vanilla,brasno,izmesaj

izmesaj=stopi((400*cokolada)+margarin)
izmesaj=isturi(izmesaj+(2*seker))
izmesaj=isturi(izmesaj+(3*jajca)+vanila)
izmesaj= izmesaj+brasno
isturi(izmesaj)
sedodekanecista(viluska)
peci(izmesaj,250)
```

Значи и правењето торта може да се опише со алгоритам.

Пропишаните правила што ги користиме во врска со решавањето на одредената задача се вика алгоритам. Со други зборови алгоритмот се дефинира како конечно множество правила (инструкции) со кое се дефинира низа од операции со точно зададен редослед чиешто извршување е потребно за решавање на даден проблем.

Од дефиницијата за алгоритам можеме да заклучиме:

- алгоритамот завршува по конечен број на операции
- редоследот на операциите е точно зададен
  со што се добива решение на проблемот, што е и наша цел.

Секое поединечно дејство од множеството правила (инструкции) дефинирани во алгоритмот се нарекува алгоритамски чекор. Врз основа на ова можеме да кажеме дека алгоритмот се состои од низа алгоритамски чекори кои се извршуваат по однапред зададен редослед.

Пример1

Да се состави алгоритам за пресметување на плоштина на правоаголник со страни a и b.

Според дефиницијата ние треба да одредиме низа од дејства (алгоритамски чекори) така што со нивната примена врз податоците ќе го добиеме точниот резултат.

Најпрво тргнуваме од правилото за пресметување плоштина на правоаголник со страни a и b.
Формулата гласи: $P=a\cdot b$
Сега ги дефинираме алгоритамските чекори:

Чекор 1. читање на вредностите на променливите a и b
Чекор 2. пресметување на вредноста на P по формулата $P=a\cdot b$
Чекор 3. прикажување на вредноста на променливата P
Чекор 4. крај на алгоритмот

Да ги појасниме чекорите

Чекор1 е алгоритамски чекор за влез со кој на познатите променливи се доделуваат почетни вредности. Во алгоритмот на a и на b им се доделуваат почетни вредности.

Чекор 2 е алгоритамски чекор за пресметување на вредноста на променливата што е зададена со формула зависна од веќе познатите променливи. Во примерот се пресметува вредноста на P.

Чекор 3 е алгоритамски чекор за излез со кој се печатат или прикажуваат на екран вредностите на одделни променливи. Во примерот се прикажува вредноста на променливата P.

Мора да напоменеме дека не може да се наведе алгоритамски чекор за пресметување на вредноста на променливата ако не се познати вредностите врз основа на кои таа се пресметува. Исто така ни алгоритамски чекор за прикажување на вредност на променлива, на која не и е доделена или пресметана вредноста.

За еден алгоритам велиме дека е точен, ако за секоја внесена инстанца завршува со точен излез. Велиме дека точен алгоритам го решава дадениот проблем.

Алгоритмот мора да се направи така да биде испланиран да се извршува од човек или од машина. Во општ случај, алгоритмот мора да се формулира во чекори, доволно едноставни за да се извршат од човек или машина (компјутер).

Алгоритмите имаат многу заеднички работи со програмите, но постојат и значајни разлики меѓу нив.

Прво, алгоритмите се поопшти од програмите. Еден алгоритам може да биде решен од човек или машина, или од двете. Програмата мора да биде извршена од компјутер.

Второ, алгоритмите се поапстрактни од програмите. Еден алгоритам може да биде изразен во било кој конвенционален јазик или нотација, а програматамора да биде изразена во некој програмски јазик.

Ако алгоритмот имаме намера да го извршиме на компјутер, прво треба да го кодираме во одреден програмски јазик и може да бираме во кој програмски јазик сакаме.

Сортирањето е без сомнение единствениот сметачки проблем за кој алгоритмот бил развиван.

Секој алгоритам треба да ги има следниве карактерични делови:

- влез
- излез
- дефиниција
- ефективност
- крај

2. Претставување на алгоритмот

Алгоритмот може да се прикаже на два начини:

- текстуално
- графички

Преку пример ќе ги разгледаме двата начина на преставување на алгоритмот.

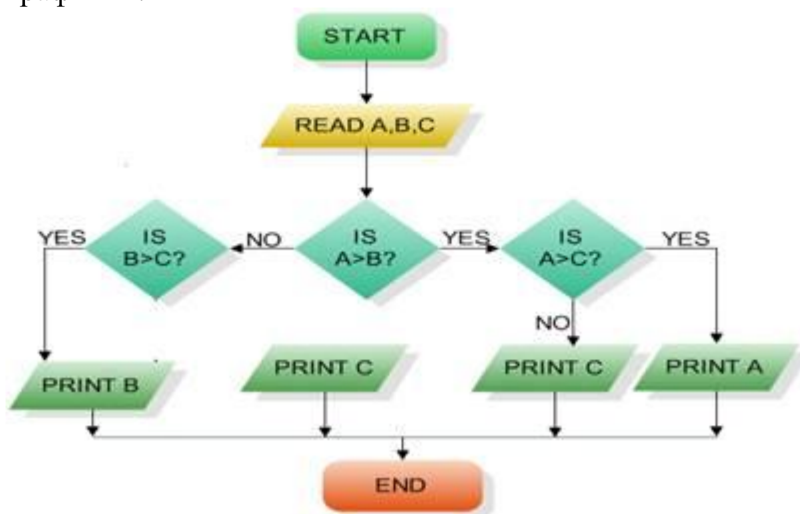
Пример2
Да се напише алгоритам за наоѓање на најголем број од три внесени броја.

Текстуално:

```
алгоритам
Најголем;
почеток
          читај a,b,c;
           ако a>b
                      тогаш
                                 p←a;
                      инаку
                                 p←b;
          крај_ако(a>b)
           ако p>c
                       тогаш
                                  n←p
                    инаку
                                  n←c
          крај_ако(p>c)
           печати n;
крај(Најголем)
```

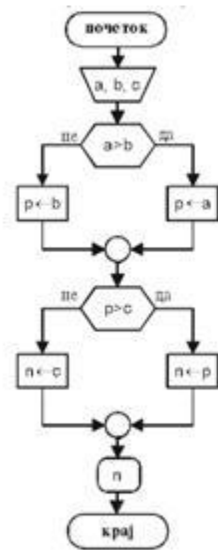
Во табелата горе е претставен текстуално алгоритмот за дадениот пример. Можеме да воочиме дека претставувањето се врши во чекори како што е кажано предходно. Се користи т.н. псевдојазик за опишување на алгоритмите чии зборови во примерот се со задебелени букви.

Графички:



Слика 1

На слика 1 е дадено графичко преставување на алгоритмот со помош на блок-дијаграм. Во блок-дијаграмот се користат посебни графички симболи за одредени дејства (операции).



Графичкото преставување има свои предности и недостатоци. Предноста во однос на текстуалното преставување е во поголемата прегледност на текот на дејствата во алгоритмот, бидејќи човекот подобро перцепира (осознава) слика од текст.

Бидејќи често се прави измена на првичниот алгоритам, полесно е таа да се направи кога алгоритмот е преставен графички, а и подоцна истата промена полесно може да ја увиди секој кој би го погледал алгоритмот.

Но од друга страна, графичкото преставување е непогодно за поголем алгоритам кој може да зафати повеќе страници, во кои тешко би се снаоѓале. Со графичкото преставување поголем проблем може да се сведе на помали елементарни проблеми кои би се поделиле на повеќе луѓе за да ги решат и потоа да се имплементираат во еден алгоритам кој го решава целиот проблем.

3. Својства на алгоритмите

При решавањето на даден проблем, треба да се води сметка за тоа на кој начин е напишан алгоритмот т.е.по кој редослед се извршуваат операциите во него.

При негово пишување треба особено да се внимава на:

- алгоритмот да има само еден почеток
- алгоритмот да има само еден крај
- во него да нема бесконечни циклуси
- да нема делови (чекори) кои никогаш не се извршуваат

Пример 3

```
алгоритам X;
почеток
          читај a,b;
          додека a<b извршувај
                    c←a*b;
                    печати a,b,c;
          крај_додека(a<b)
          печати a,b;
крај(X)
```

Пример 4

```
алгоритам Y;
почеток
      читај m,n;
       ако m≥n
              тогаш
                   скок на oznaka
               инаку
                     ако m<n
                         тогаш
                               печати m,'<',n;
                               скок на oznaka
                          инаку
                               печати m,'=',n;
                      крај_ако(m<n)
        крај_ако(m>n)

  oznaka: печати 'Каде е грешката?';
крај(Y)
```

Дадени се алгоритми во пример 3 и 4. Кој од нив не е правилен алгоритам? Зошто?

(Одговор:Алгоритмот од пример 3 не е правилен, бидејќи ако бројот a е помал од бројот b, тогаш циклусот ќе се извршува бесконечно и никогаш нема да се добие резултат (решение).

Исто така и алгоритмот во пример 4 не е правилен, бидејќи во него постои дел кој никогаш не се извршува. Делот печати m,'=',n; во внатрешната ако-тогаш-инаку контролна структура никогаш нема да се изврши.)

За даден проблем може да се напишат различни алгоритми и сите тие да бидат правилни. Ако се правилни секој од нив ќе даде точно решение. Но тоа не значи дека сите тие алгоритми се подеднакво ефикасни. Некој можеби ќе биде побрз, друг да дава поточни резултати, трет да има пократок запис или да биде појасен итн.

Но секој од тие алгоритми треба да ги има следниве особини:

- секоја операција е точно определена
- во алгоритмот е прецизно утврден редоследот на извршувањето на операции
- алгоритмот може да се расчлени на елементарни чекори
- дава резултати по конечен број чекори
- со алгоритмот се опфатени сите можни решенија за различни вредности на влезните податоци
- не зависи од тоа дали се извршува од човек или на сметач,и тоа независно од типот на сметачот
- да е разбирлив за секој,независно кој го напишал

Споменавме дека може да се напишат повеќе правилни алгоритми за ист проблем. Тогаш се поставува прашањето кој алгоритам да се користи? Кој алгоритам е подобар?
Лесно е да кажеме овој алгоритам е подобар од другиот, но како сме дошле до тоа сознание, зошто едниот е подобар од другиот алгоритам?
Овие прашања имаат одговор кој води до анализа на алгоритмот и одредување на неговата сложеност.

4. Анализа на алгоритмот

Претпостави дека компјутерите се бесконечно брзи и компјутерската меморија слободна.Дали би имале било каква причина да учите алгоритми? Одговорот е да, ако не за друго да докажите дека вашето решение завршува и го дава точниот одговор.

Ако компјутерите се бесконечно брзи, секој точен метод за решавање на одреден проблем ќе работи. Вие сигурно сакате вашата имплементација да биде добро дизајнирана и документирана но многу често го користите методот што е најлесен за имплементација.

Секако, компјутерите можеби се брзи, но не се бесконечно брзи. И меморијата можеби е ефтина но не е неограничена. Времето за пресметка е исто така ограничувачки ресурс, како што е просторот во меморијата. Овие ресурси треба да се користат мудро, и алгоритмите ќе бидат поефикасни во однос на времето и меморијата.

Анализата на алгоритмот овозможува да се даде квантитативна оценка за предноста на еден алгоритам во однос на друг алгоритам за ист проблем. Оценката може да преставува како алгоритмот се однесува во:

- најдобар случај
- најлош случај
- просечен случај

Значи анализата на алгоритмот се прави за да може полесно да се спореди со друг алгоритам за истиот проблем. Со анализирањето на алгоритмите и определување на најдобриот алгоритам од понудените за тој проблем значително се заштедува на време и пари и се добива поголема ефикасност во решавањето на зададените задачи.

Ќе се запрашаме што точно се оценува кај еден алгоритам? На кој начин се споредуваат два алгоритми?
Еден начин за споредба на алгоритмите е да се споредат кој побрзо го решава дадениот проблем. Некои доаѓаат побрзо до решението од другите. Најчесто и ние го одбираме алгоритмот што најбргу доаѓа до решението.

Друг начин за споредба на алгоритмите е да се види колку мемориски простор е потребен при извршување на алгоритмот напишан во програмски јазик. Некои од нив зафаќаат повеќе мемориски простор, но бргу доаѓаат до решението, а други помалку мемориски простор, а до решението доаѓаат поспоро.

Значи сложеност е ниво на тежина при решавање на одреден проблем мерен во време, број на чекори или аритметички операции, меморија.

Ние ќе се задржиме на временската сложеност на алгоритмот.
Времето што е потребно за решавање на одреден проблем зависи од одреден број на фактори:

- колку е брз компјутерот
- капацитетот на RAM-от на компјутерот
- оперативниот систем
- квалитетот на кодот генериран од компајлерот итн.

Ако некој од факторите се промени тогаш и времето на извршување се менува.

Затоа оваа големина не е доволно добра како мерка за алгоритамските перформанси. Нас ни е потребна мерка со која ќе може да споредиме 2 алгоритми.

Анализата на алгоритмот се состои од 2 фази:

- претходна анализа
- тестирање

Во првата фаза, претходна анализа, се формира функција на сложеност зависна од релевантни фактори која го определува времето на извршување. Функцијата всушност преставува оценка на алгоритмот кога тој не зависи од тоа на каков компјутер се извршува.

На кој начин се прави тоа?

Секој алгоритам се состои од конечен број инструкции. Колку повеќе иструкции, толку подолго тој ќе се извршува. Еден начин е да се бројат инструкциите (операциите) од кој е составен алгоритмот за решавање одреден проблем. Потребно е да се знае колку пати се извршува инструкцијата и која е “нејзината цена” т.е. времето (честота) на едно извршување на инструкцијата.

Нека се дадени следниве програмски сегменти:

```
а) x = x + y
                                              в) For i = 1 to n do
б) For i = 1 to n do                              For i =1 to n do
      x = x + y                                  x = x + y
    end                                            end
                                                    end
```

а) честота на извршување е 1
б) честота на извршување е n
в) честота на извршување е n^2

Бројот на инструкции ќе варира зависно од бројот на внесените податоци. Програма за пресметка на плати, за влез од 100 лица ќе изврши повеќе инструкции одколку за 10 лица. Затоа може да го изразиме бројот на инструкции како функција на сложеност од бројот на внесени податоци.

Пример 5

```
Алгоритам Збир              1
 Почеток                    1
   читај n;                 1
    s←0;                    1
   за i← 1 зголемувај до n  n
       s←s+i;               n
    крај_за(i)              1
  печати s;                 1
 крај(Збир)                 1
```

Функцијата што го преставува бројот на инструкции за овој алгоритам може да се запише како f(n)=2n+7. Оваа функција на сложеност е линеарна, што значи и сложеноста на овој алгоритам е линеарна. Но мора да напоменеме дека оваа функција е зависна од сметачот и програмскиот јазик во кој се извршува.

Нека f(n) е функција на сложеност што зависи од компјутерот на кој се извршува алгоритмот, а g(n) функција на сложеност што не зависи од тоа на каков компјутер се извршува алгоритмот. Тогаш важи f(n)=O(g(n)).

Кога ќе се каже ...Алгоритмот има O(g(n)) време на пресметување (извршување)... Тоа значи дека ако алгоритмот се извршува повеќе пати на иста машина, на ист тип податоци, но за различно (растечко) n, времето на извршување секогаш ќе биде помало од некоја константна вредност |g(n)|.

Значи функцијата на сложеност која не зависи од тоа на каков сметач се извршува алгоритмот во примерот 5 е g(n)=n, и велиме алгоритмот има сложеност O(n) (се чита “од ред n ”).

Но дали треба да ги земеме во предвид сите инструкции во алгоритмот?

Најчесто алгоритмите се состојат од циклус и инструкции што се извршуваат во него или надвор од него. Бројот на инструкциите надвор од циклусот во алгоритми што се однесуваат за ист проблем незначително се разликува.

Од нив не зависи дали алгоритмот ќе има помала или поголема сложеност. Ако бројот на влезни податоци расте тогаш бројот на инстукции што се извршуваат во циклусот го надминува бројот на оние што се надвор од него.

За да ја упростиме нашата пресметка нема да ги броиме инструкциите надвор од циклусот.

Дали треба да ги земеме во предвид сите инструкции во циклусот?

Бројот на инструкции во циклусот незначително се разликуваат во 2 алгоритми напишани за ист проблем. Но, тоа што се разликува е колку пати циклусот се извршува.

Пример 6

```
алгоритам NajdiElement
  почеток
     читај n,baran;
      за i=1 зголемувај до n
           читај a[i];
      крај_за(i)
      i=1;
      додека ((i≤n) и (baran ≠ a[i] ))
          i=i+1;
      крај_додека((i≤n) и (baran ≠ a[i] ))
     ako i=n
               тогаш
                    печати "Елементот не е во низата";
                инаку
                    печати "Елементот се наоѓа на ",i,"-тата
позиција";
 крај(NajdiElement)
```

Потребно е да го пресметаме бројот на извршувања на циклусот. Во алгоритам кој пребарува низ низа елементи, во циклусот се споредува тековниот елемент од низата со елементот што треба да се најде.

Во овој случај го бараме бројот на споредувања на елементите во дадениот алгоритам. Но бројот на споредби пак зависи од должината на низата. Нека низата има должина n, во алгоритам за линеарно пребарување. Бројот на споредувања ќе биде n кога елементот што се бара не е во низата. Тогаш велиме дека алгоритмот има сложеност O(n) или линеарна сложеност.

Ако ја зголемиме двојно низата, двојно се зголемува и бројот на споредби во алгоритмот. Но некогаш имаме среќа да го најдеме елементот блиску до крајот, а некогаш блиску до почетокот на низата. Во тие случаи ни треба да ја пребараме половина од низата за да го најдеме тој елемент. Па тогаш би имале O(n/2) споредувања.

Ова се нарекува просечна сложеност на алгоритмот. Ако пак елементот не се наоѓа во низата, се прават n споредувања и ова се нарекува најлош случај на сложеност на алгоритмот. Но да забележиме дека сепак сложеноста на алгоритмот е O(n) наспроти фактот дека во просек се прават n/2 споредби.

Ова е поради тоа што ако n расте многу големо, разликата помеѓу n и n/2 станува се помалку значајна. Кога се споредува алгоритам важно е да се знаат и двете сложености (просечна и најлошиот случај).

Ќе опишеме друг алгоритам за барање низ низа наречен бинарно пребарување (Binary Search) каде бројот на споредувања е значајно помал. Тој изнесува O(log n).

Пример 7

```
алгоритам БинарноБарање
  почеток
        печати 'Внеси број на елементи во низата';
        читај n;
        печати 'Внеси ги елементите';
         за i=1 зголемувај до n
                 читај ai;
         крај_за(i)
         печати 'Внеси ја вредноста што се бара';
         читај v;
         levo ← 1;
         desno ← n;
         повторувај
                   sredina ← [(levo+desno)/2];
                   ако v>asredina
                         тогаш
                                levo ← sredina+1;
                         инаку
                                desno ← sredina-1;
                   крај_ако( v>asredina)
         до asredina=v или levo>desno;
         крај_повторувај
         ако asredina=v
                    тогаш
                          печати 'Елемент со таква вредност
е',asredina;
                     инаку
                          печати 'Елемент со таква вредност не
постои';
          крај_ако(asredina=v)
 крај(БинарноБарање)
```

Ако имаме низа со должина 1000 со бинарно пребарување би го нашле бараниот елемент во најлош случај со 10 споредби (обратно од линеарното пребарување). Но низа со должина 1 билион во најлош случај го наоѓа елементот по 30 споредувања. Очигледно е дека бинарно пребарување е со помала сложеност од линеарното пребарување.

Бинарното пребарување се состои од делење на низата на половина и воочување во која од двете добиени низи е бараниот елемент. Потоа таа низа повторно се дели на половина инт., се додека не се пронајде елементот или пак додека не се воочи дека елементот не е во низата.

Во најлош случај бинарното пребарување престанува кога нема да го најде елементот во низата. По i-тото делење на низата на пола, кандидати за елементот што се бара се најмногу n/2i. Кога ќе заврши процесот i е најмалиот цел број така што (n/2i)log n. Значи алгоритмот се изведува во време O(log n).

Бинарно пребарување има ограничено користење т.е. само кога низата е сортирана. Тоа е метод кој го користиме кога листаме телефонски именик. Отвараме на “средина” и во зависно од тоа каде сме отвориле знаеме дали бројот е во едната половина од именикот или во втората половина. Со ова елиминираме половина од именикот со едно споредување. Оваа постапка се повторува на делот што останува додека не се најде бројот или додека не се утврди дека не се наоѓа во низата, со што секоја натамошна споредба ја намалува низата за половина.

Но како што рековме за да го користиме овој метод низата треба да е сортирана. Алгоритмите за пребарување и сортирање се повеќе се користат. Па всушност скоро во секој алгоритам за поголем проблем има сортирање и пребарување.

Ќе покажеме алгоритам за сортирање наречен метод на меурче (Bubble Sort).

```
алгоритам СортирањеСоМетодНаМеурче
  почеток
        печати 'Внеси број на елементи во низата';
        читај n;
        печати 'Внеси ги елементите';
         за i=1 зголемувај до n
                 читај ai;
         крај_за(i)
          brojac ←2;
         повторувај
                    за i=n намалувај до brojac
                      ако ai<ai-1
                         тогаш
                              pom ←ai;
                              ai ←ai-1;
                              ai-1 ←pom;
                      крај_ако(ai<ai-1)
                    крај_за(i)
              brojac=brojac+1;
         до brojac=n;
         крај_повторувај
         печати 'Сортираната низа е';
         за i=1 зголемувај до n
                 печати ai;
         крај_за(i)
 крај(СортирањеСоМетодНаМеурче)
```

Има многу начини да се сортира низа. Подолу ќе ја објасниме метода на меурче(Bubble Sort) и ќе ја покажеме нејзината сложеност. Ќе ја сортираме низата во растечки редослед. Нека низата е: 10, 34, 2, 16, 23, 8, 12. Сакаме да добиеме 2, 8, 10, 12, 16, 23, 34. За да ја сортираме мора да ги знаеме вредностите на елементите во низата, така што елементите со помала вредност ќе ги заменат местата со оние со голема вредност, со што големите вредности ќе дојдат на крајот, а малите на почетокот на низата.

Сакаме да ги поместиме големите вредности на крајот од низата. Може да споредуваме само 2 вредности од низата. Во овој метод почнуваме од почетокот на низата. Ги споредуваме првите 2 елементи L[0] и L[1].Ако L[0]>L[1] тогаш смени им ги местата. Потоа ако L[1]>L[2] тогаш смени им ги местата итн. Ова го поместува најголемиот елемент на крајот на низата. Другиот дел од низата не е сортирана. Сме поминале низ низата еднаш.

Низата ја има формата 10, 2, 16, 23, 8, 12, 34. Потоа почнуваме пак од почеток со горенаведената постапка. И се така до моментот кога ќе ја пројдеме низата по n-ти пат ако таа има должина n. По завршувањето на методот низата е следна: 2, 8, 10, 12, 16, 23, 34 и е сортирана. Оваа метода се нарекува така затоа што елементот се движи кон крајот на низата како меурче во чаша вода за да излезе на површина. Поголемото се движи побрзо кон површината.

Задача:

Со метод на меурче да се сортира низата 2,4,15,6,7,9,10. Колкава е сложеноста на алгоритмот во овој случај?

Сложеноста на метод на меурче за низа од n елементи е $O(n^2)$. Зошто?
Низа од n елементи, со овој метод ја поминуваме n пати, а во секое поминување на низата правиме n споредби на елементите. Од таму сложеноста на овој алгоритам е $O(n^2)$.

Досега видовме сложеност на 3 алгоритамски методи:

- Линеарно пребарување O(n)
- Бинарно пребарување O(log n)
- Метод на меурче O(n2)

Сложеноста на алгоритмите со низи е најмалку O(n) бидејќи скоро секогаш мора да поминеме барем еднаш низ низата. Бинарното пребарување е на некој начин исклучок. Предноста е во тоа што не мора да се поминува целата листа која е сортирана. Но што ако низата не е сортирана? Тогаш или ќе користиме линеарно пребарување што се користи и за несортирани и сортирани низи или прво ќе ја сортираме низата, а потоа ќе употребиме бинарно пребарување. Линераното пребарување ќе биде подобро од комбинацијата сортирање и бинарно пребарување. Алгоритам со сложеност O(n) е поефикасен алгоритам во споредба со метод на меурче со сложеност $O(n^2)$.

Според сложеноста на алгоритмите разликуваме:

- константни O(1)
- линеарни O(n)
- логаритамски O(log2 n)
- линеарно-логаритамски O(n·log2 n)
- квадратна O(n2)
- експоненцијални O(2n) (n>1)
- факториелни O(n!)

Колку помала сложеност толку поголема ефикасност има алгоритмот. Нормално ние ќе го избереме оној алгоритам со најмала сложеност. Од горе наведените алгоритми најефикасни се логаритамските, а видовме и зошто.
Постојат голем број на линеарни алгоритми кои се ефикасни,а квадратните алгоритми имаат добра ефикасност.

Експоненцијалните и факториелните алгоритми имаат многу мала ефикасност и се користат само кога мора. Постојат проблеми за кои има пронајдено само експоненцијални или факториелни алгоритми. Наоѓањето на алгоритам со помала сложеност за таков проблем е голем успех.

Втората фаза од анализата на алгоритмот е неговото тестирање на машина во некој програмски јазик. Многу од програмите кои се користат содржат багови (грешки). Секој (чесен) програмер ќе ги преброи глупавите, смешни и сериозни грешки што ги направил во програмата. Дури и ќе се обиде да ги поправи и во секое наредно пишување на алгоритам за било кој проблем ќе гледа да не ги повтори. 70% од конструирањето на нов комплексен софтвер се троши за поправање на грешките. Почетниците најчесто веруваат дека нивните алгоритми го прават токму она што тие сакаат да го направат и дека тие алгоритми се најефикасните.

При тестирањето не може да се задржиме на функцијата на сложеност од претходната анализа, туку треба да се води сметка и за константите што се јавуваат во таа функција.

Прашање:

Ако имаме два алгоритми кои истата задача ја извршуваат со n влеза, и првиот има време на пресметување O(n), а вториот O(n2), кој е побрз?

Со сигурност ќе тврдиме дека првиот е побрз. Лесно е да се забележи дека за доволно големи вредности на n, времето на извршување на вториот алгоритам ќе биде поголемо од времето на извршување на првиот.

Нека вистинско време на извршување на алгоритмот на сметач е 2n и n2 соодветно.Тогаш првиот алгоритам е побрз за сите n>2.

Проблем: Што ако 104n и n2 се соодветните вистински времиња на извршување?

Тогаш вториот алгоритам е побрз за сите n104, побрз е првиот алгоритам. Значи не може да одлучиме кој алгоритам е побрз, ако не знаеме ништо за константите поврзани со функцијата на сложеност.

Методите во оваа фаза може да се групираат во две категории: тестирање и докажување. Тестирањето се состои од извршување на програмата за разни влезни податоци и дали ќе даде точен резултат за тие податоци.Тестирањето се врши само над одбрано множество влезни податоци. Докажувањето на програмата значи дека програмата е исправна за сите дозволени влезни податоци (од различен тип), дури и за голем број на влезни податоци. Тестирањето повеќе се користи бидејќи е полесна техника за проверка на исправноста на алгоритмот. Докажувањето на алгоритмот се користи за резонирање на однесувањето на програмата. Со него може да се зголеми довербата во исправноста на програмата.

5. Ефикасност

Алгоритмите што се смислени да решат ист проблем често драматично се разликуваат во нивната егикацност. Овие разлики можат да бидат позначајни од разликите во хардверот и софтверот.
Како за пример, ќе видиме два алгоритми за сортирање. Првиот, или уште попознат како инсертион сорт, зема време n2 за да сортира n работи. Вториот, merge sort, истите работи ќе ги сортира за време log2n. Вреди да се спомене тоа што insertion sort е обично побрз од merge sort за мали вредности на n, но кога бројот на внесени податоци доволно ќе нарасне merge sort е убедлив во однос на insertion sort.
За конкретниот пример, нека имаме компјутер А којшто е побрз и ќе равоти со insertion sort и компјкутер Б којшто е поспор и ќе работи со merge sort. Нивна задача е да сортираат низа од 1000000 броеви.

Да претпоставиме дека компјутерот А извршува 1 билион инструкции во секунда, а компјутерот Б извршува само 10 милиони инструкции во секунда. Значи компјутерот А е 100 пати побрз од компјутерот Б. За да ја направиме разликата уште повеќе драматична нека константите ц1 за компјутерот А биде 2, а ц2 за компјутерот Б биде 50. За да сортира еден милион броеви, на компјутерот А ќе му требаат

$$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}} = 2000 \text{ seconds} ,$$

додека на компјутерот Б ќе му требаат:

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 100 \text{ seconds} .$$

Со користење на алгоритам чиешто времм на извршување расте многу поспоро, дури и со послаб компајлер, компјутерот Б работи 20 пати побрзо отколку компјутерот А! Предноста на merge sort ќе биде уште поизразена кога би сортирале 10 милиони броеви: кога на insertion sort би му требале приближно 2,3 дена, додека на merge sort помаалку од 20 минути. Генерално, колку големината на проблемот се зголемува, толку се золемува предноста на merge sort.

Псевдокод на insertionSort во Pascal

```
insertionSort(array A)
begin
    for i := 1 to length[A]-1 do
    begin
        value := A[i];
        j := i-1;
        while j ≥ 0 and A[j] > value do
        begin
            A[j + 1] := A[j];
            j := j-1;
        end;
        A[j+1] := value;
    end;
end;
```

Псевдокод на insertionSort во Java

```
InsertionSort(A) **sort A[1..n] in place
    for j = 2 to n do
        key  = A[j] **insert A[j] into sorted sublist A[1..j - 1]
        i  = j - 1
        while (i > 0 and A[i] > key) do
            A[i+1] =  A[i]
            i  = i - 1
            A[i+1]  = key
```

Псевдокод на mergeSort во Pascal

```
MergeSort(List[], leftIndex, rightIndex)
Begin:
    if leftIndex < rightIndex then
        mid = (leftIndex + rightIndex) / 2
        MergeSort(List[], leftIndex, mid) // split left sublist
        MergeSort(List[], mid + 1, rightIndex) // split right sublist
        Merge(List[], leftIndex, mid, rightIndex) // merge sorted
sublists
    endif
End
```

Псевдокод на mergeSort во Java

```
function merge(left[], right[], output_tape[])

     do

        if left[current] ≤ right[current]

            append left[current] to output_tape

            read next record from left tape

        else

            append right[current] to output_tape

            read next record from right tape

    while left[current] < left[next] and right[current] < right[next]

    if left[current] < left[next]

        append current_left_record to output_tape

    if right[current] < right[next]

        append current_right_record to output_tape

    return
```

Задачи:

1.Мирко се пријавил на конкурс за работа на проект. На интервјуто на кое бил повикан, работодавецот му објаснил дека ќе го плаќа на следен начин:

- прв ден 1 евро
- втор ден 2 евра

  ……………

Мирко очекувал од проектот да заработи најмалку 200 евра, во спротивно не би ја прифатил работата. Проектот трае n денови. Помогнете му на Мирко да пресмета дали му се исплати да работи на проектот.

Решение:

```
Алгоритам Заработка                        1
   почеток                                     1
    читај n;                                   1
   s← n*(n-1)/2;                                1
   ако s<200                              1
      тогаш
        печати "Не ја прифаќам работата"; 1
      инаку
        печати "Ја прифаќам работата";    1
    крај_ако(s<200)
  крај(Заработка)
```

Значи f(n)=7,бидејќи кога алгоритмот се извршува на сметач секоја инструкција има определено време на извршување. Сега треба да го одредиме најмалото g(n) за да важи f(n)=O(g(n)). Согледуваме дека најмалото g(n)=1 и запишуваме дека алгоритмот има сложеност O(1) (се чита "од ред 1" т.е. константна сложеност.)

2. За дадена низа a да се создаде низа b така што i-тиот елемент во низата b е аритметичка средина на елементите до i-тата позиција во низата a.

Решение:

```
Алгоритам НизаОдНиза          1
 почеток                      1
  читај n;                     1
  за i←1 зголемувај до n        n
   читај ai;                    n
  крај_за(i)
  b1 ←a1;                     1
  за i← 2 зголемувај до n        n-1
   bi ←(bi-1 * (i-1)+ai)/i;           1
  крај_за(i)
   за i←1 зголемувај до n         n
   печати bi;                   n
  крај_за(i)
крај (НизаОдНиза)
```

Значи f(n)=1+1+1+n+n+1+(n-1)+1+n+n)=5*n+4. Најмалото g(n)=n, па имаме сложеност O(n) т.е. линеарна сложеност.

3. Следните Java методи имплементираат собирање и множење на матрици. Секоја матрица е претставена со n x n дво-димензионални низи од float броеви

```
static void matrixAdd (int n, float[][] a, float[][] b, float[][] sum) {
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            sum[i][j] = a[i][j] + b[i][j];
        }
    }
}
static void matrixMult (int n, float[][] a, float[][] b, float[][] prod){
    for(int i = 0; i < n; i++){
        for (int j = 0; j < n; j++){
            float s = 0.0;
            for (int k = 0; k < n; k++){
                s += a[i][k] * b[k][j];
            }
            prod[i][j] = s;
        }
    }
}
```

Анализирајте ги овие методи. Која е комплексноста на секој метод?

(Одговор: На собирањето е $O(n^2)$, а на множењето е $O(n^3)$.)

Низи, куп, ред и шпил

Низи

Податоците кои се состојат од прости (неструктурирани) типови на податоци и кои што се третираат како сложена целина се нарекуваат структурирани типови на податоци.Такви се низа битови, низа знаци, општа низа, множество запис, стек, ред, датотеки.

Честопати е потребно да се сместат голем број на податоци како една целина.За сместување на ваквите податоци се употребува структуриран тип на податоци наречен низа.

Низата претставува група од мемориски локации кои имаат исто име и чии елементи се од ист тип.За да означиме специфичен елемент на низата, го задаваме името на низата и позицијата на тој елемент во низата. Доколку сакаме да пристапиме до елемент од низата тоа се врши преку името на низата и редниот број на елементот во низата.Со оглед на големината на низите може да имаме еднодимензионални и повеќедимензионални низи.

Еднодимензионални низи

Низите во C++ се декларираат со:

тип_на_елементите име[број на елементи];

а додека елемент на еднодимензионална низа се означува со:

име[индекс]

каде што *индекс* може да биде од 0 до *број_на_елементи-1*.

Пример за декларација:

```
int a[5];
```

a[0]	12
a[1]	23
a[2]	34
a[3]	45
a[4]	1

Низата a е составена од 5 елементи. Сите елементи на низата имаат исто име a, а различен индекс. 0 е индекс на прviот елемент на низата a, а 4 е индекс на последниот елемент на низата a.

Пример за декларација:

```
const int brojelementi=100;

int c[brojelementi];
```

тука i-тиот елемент е c[i], а индексот i може да биде од 0 до 99. Исто така важно е да се напомени дека при ваква декларација бројот на елементи мора да биде константа како што е запишано во горенаведениот пример. Доколку бројот на елементи не е константа тогаш декларацијата е погрешна.

Пример за погрешна декларација:

int k=10;

float u[k];

Доделување вредности и иницијализирање на низи

На елементите на низата им се доделуваат вредности со наредба за доделување или при иницијализација.

Примери на доделување на вредности на елементи на низа:

int a[10];

a[0]=1; a[1]=2; a[3]=5; a[4]=7; a[5]=-9; a[6]=9; a[7]=11; a[8]=12; a[9]=15;

Доделувањето може да се врши и преку изрази во кои се пресметува вредноста на индексите на низата, т.е ако е дадена следната низа наредби:

int c=3;
int d=2;
a[c+d]+=6;

тогаш на елементот a[5] му се зголемува вредноста за 6 т.е неговата вредност беше a[5]=-9 што значи сега ќе биде -3.

Исто така доделување вредности на елементите на низа може да се врши и со инцијализација на елементите при декларирањето:

int a[5]={ 5, 2, 7, -2, 6};

Вредностите на елементите се: a[0]=5, a[1]=2, a[2]=7, a[3]=-2, a[4]=6.

Пример за иницијализацција на елемтите на низа со користење на наредбата for.

```
#include
using namespace std;
int main()
{
int a[10];
int i;
for( i=0; i<10; i++)
a[ i ]=0;
//pechatenje na nizata a
count << "Element Vrednost "<< endl;
for ( i=0; i<10; i++)
count<< i<< " "<< a[ i ]<< endl;
return 0;
}
```

Повеќедимензионални низи

Низите може да имаат повеќе од една димензија. При што првиот индекс го претставува бројот на редицата а вториот бројот на колоната, елементот се означува со името на низата проследен од индексите по редици и колони ставени секој во посебни средни загради, т.е a[i][j] каде a е името на низата, i е индексот по редици, а j е индексот по колони. Најкористени повеќедимензионални низи се **дводимензионалните**, уште наречени матрици.

Декларацијата на дводимензионалните низи е:

тип_на_елементите име[димензија1] [димензија 2];

кај *димезија1* е првата димензија, а *димензија2* е втората димензија на низата. Низата има вкупно *димензија1 x димензија2* елементи.

Ваквата додимензионална низа може да се претстави како табела со редици и колони, каде со првиот индекс се означуваат редиците, а со вториот колоните.

На пример со декларацијата:

int a[4][3];

се декларира низата а од 12 елементи, на кои првиот индекс им е 0,1,2 или 3, а вториот индекс им е 0,1 или 2, односно оваа низа е составена од 4редици и 3 колони.

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]

Како и кај еднодимензионалните низи и тука кај дводимензионалните низи, димензиите на низите може да се зададат и преку константи:

const int m=10;

const int n=5;

float b[m] [n];

А вредности на елементите на дводимензионалните низи се доделуваат со наредба за доделување.

Пример:

float c[10][15];

c[0][4]=3; c[1][2]=10;

се доделува вредност на елементите c[0][4] и c[1] [2].

Исто така дводимензионалните низи може да се иницијализираат и со иницијализирачка листа при декларирање.Приота ако вредностите се зададат во една редица тогаш тие ќе се доделат на

елементите редица по редица, а ако се одвојат во посебна листа за секоја редица, тогаш елементите ќе се иницијализираат според поделбата на листата.

Пример:

int a[3][2]={ -3, 5, 0, 1, 7, -2};

елементите на а ќе ги добијат истите вредности како и при:

int a[3][2]= { {-3, 5}, {0, 1}, {7, -2}};

Иницијализација и печатење на дводимензионална низа.

```
#include
using namespace std;
void main()
{
int i, j;
//inicijalizacija na niza so inicijalizatorska lista
int niza1 [2] [3] ={ {1,2,3}, {4,5,6}};
cout<< "Prvata niza e inicijalizirana"<<endl;
int niza2[2][3];
cout<<"\n Vnesete elementi za vtorata niza"<<endl;
for(i=0; i<=1; i++)
{
cout<<"\n Vnesete elementi vo "<< i+1<<"-ta redica"<<endl;
for(j=0; j<=2; j++)
{
cout<<"Vnesete go "<<"-ot element";
cin>>niza2[i][j];
}
}
cout<<"\n Pechatenje na nizite"<< endl;
cout<<"\n Pechatenje na prvata niza "<<endl;
for(i=0; i<=1; i++)
{
for(j=0; j<=2; j++)
cout<<niza1[i][j] << " ";
cout<<endl;
}
cout<<"\n Pechatenje na niza2"<
for(i=0; i<=1; i++)
{
for(j=0; j<=2; j++)
cout<<niza2[i][j] << " ";
cout<<endl;
}
}
```

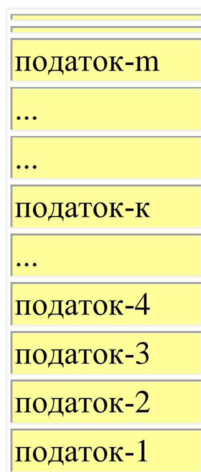
Како што можеме да забележиме низите се од голема корист, но зошто да не се употребат за било каков податочен тип? Во случај кога компонентите во низата не се сортирани, можеме да сместуваме компоненти во O (1) време, но времето потребно за пребарување е O (n).

Кога компонентите во низата се подредени, можеме да пребаруваме брзо т.е во O (logn) време, но за внесување на нови компоненти е потребно O (n) време. Друг проблем што се јавува кај низата е тоа

што тие имаат фиксирана големина. Обично откако програмот ќе се стартува, ние не знаеме точно колку компоненти треба да се внесат, па претпоставуваме колкава би била должината на низата. Ако претпоставиме дека должината на низата е премногу голема, тогаш би потрошиле меморија, така што ќе имаме ќелии во низата што никогаш нема да бидат попол нети. Ако претпоставиме дека должината е мала, тогаш ќе ја преплавиме низата, така што ќе дојде до паѓање на програмата.

Куп

Куп (анг. stack) е линеарна листа која се карактеризира со операциите кои можат да се извршуваат со неа. Елементите од купот се додаваат според стратегијата на работа наречена "кој последен влегува прв излегува" (анг. Last In Frst Out- LIFO). Тоа значи дека секогаш нов податок врз купот се става врз последниот податок, додека вадењето податоци се врши во спротивен редослед од ставањето т.е прв се зема последниот ставен податок.



На сликата е даден графички приказ на куп со m елементи. Значи елементот кој што прв влегол во купоте податок 1, а елемент т.е податок кој што прв ќе излезе од купот е податок m односно ова е и последниот податок кој што влегол во купот.

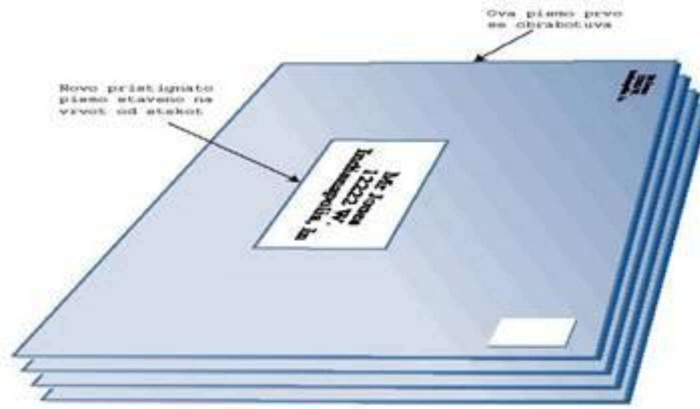
Купот исто така може да се дефинира како структура која зафаќа посебна меморија со одредена големина. Доколку капацитетот на купот е n, тогаш во него може да се сместат само n податоци.

Со купот се извршуваат следните основни операции:

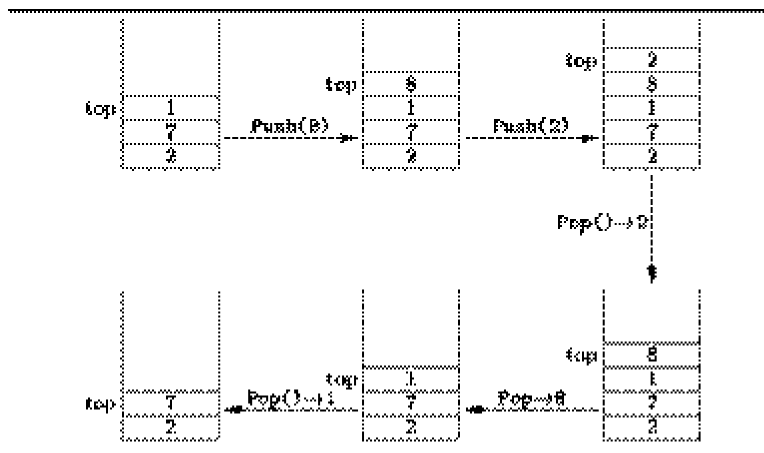
- иницијализација на празен куп
- проверка дали е купот полн
- ставање податок во купот
- проверка дали е купот празен
- земање податок од куп

Предноста на купот е дека сместувањето и отстранувањето на податок од врвот на купот се врши во O (1) време. Додека голем недостаток е тоа што освен првиот, има спор пристап до останатите податоци што се сместени во купот.

За да се разбере идејата за куп, ќе разгледаме пример за Македонска пошта. Многу луѓе, кога ја добиле поштата, ја фрлаат во купот (корпа за пошта). Потоа, кога тие имаат слободно време ја процесираат акумулираната пошта од горе па надоле. Прво тие го отвараат писмото на врвот од купот, и превземаат соодветно дејство - ја плаќаат сметката, потоа ја фрлаат итн. Кога првото пимо е распоредено, тие го прегледуваат наредното писмо, кое е сега на врвот на купот, и го проследуваат. Тие работат по ред до долното писмо на купот (кое е сега на врвот).



Друга аналогија на купот е задача која се изведува во типичен работен ден. Вие работите на долготраен проект (А), но вас ве прекинува вашиот соработник кој ве прашува за привремена помош за друг проект (Б). Додека вие работите на проектот (Б), некој ве прекинува од сметководство околу сметките за патување , (Ц), а за време на оваа средба вие имате итен повик од продажба и користите неколку минути на проблемите за некој голем производ (Д). Кога ќе завршите со повикот Д, вие го решавате Ц, кога ќе завршите со Ц, вие продолжувате со проектот Б, и кога ќе завршите со Б, (конечно!) се враќате на проектот А.



Како што можеме да забележиме на сликата купот е базиран на низа. Па така на него гледаме како низа од податоци. Иако е базирано на низа, купот има ограничен пристап, па така не можеме да пристапиме на него како во низа, т.е немаме пристап до било кој податок. Купот на сликата започнува со веќе внесени податоци. Доколку сакаме да започнеме со празен куп, креираме метод new што ќе конструира куп без податоци. За се внесе податок во купот, креираме метод push. Како што гледаме на сликата се врши внесување на елементите 8, па потоа 2. Внесувањето податоци во купот е на таков начин што се оди по редослед од дното па нагоре до врвот. Доколку сакаме да отстраниме податок од врвот на купот, креираме метод Pop(). Како што гледаме на сликата се врши отстранување на елементите 2,8 и 1. При пишување на програмата важно е да се напише код кој нема да ги дозволи следните два случаи:

- кога купот е празен, а ние сакаме да отстраниме податоци од празен куп
- коге купот е полн, а ние сакаме да сместиме уште податоци

Ред

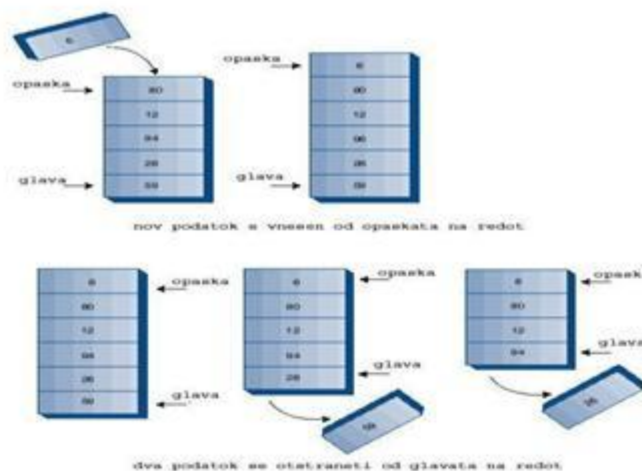
Ред (анг. gueue) е линеарна листа во која елементите се ставаат на крајот, а се земаат од почетокот на редот. Затоа велиме дека филозофијата на работа на редот е " кој прв влегува, прв излегува" (анг. Frst In Frst Out-FIFO). Редот има два покажувачи едниот покажува на почетокот на редот, а другиот на крајот на редот. Исто така редот може да биде со фиксна големина или со неограничена, ако е имплементиран како поврзана листа.

Со редот се извршуваат слични операции како и со купот:

- иницијализација на празен ред
- проверка дали е редот полн
- ставање податок на крајот на редот
- проверка дали е редот празен
- земање податок од почетокот на редот

Иницијализацијата се врши така што се задава покажувачот на почетокот и покажувачот на крајот да имаат вредност 0.

Двете основни операции за ред се сместување (inserting) на податок, каде податокот се складира во позадина од редот, и операцијата отстранување (removing) на податок , каде што се зема од предниот дел на редот. Задниот дел од редот каде што податоците се сместуваат се нарекува опашка (tail) или крај (end) на редот. Предниот дел каде што податоците од редот се отстрануваат се нарекува глава (head). Тоа е покажано во наредната слика:



Како што можеме да видиме на сликата, на опашката од редот се наоѓа елементот 80, но со користење на операцијата сместување (inserting), ќе биде внесен нов елемент на опашката од редот и ќе се инкрементира стрелката за опашка така што ќе покажува на новиот елемент (во случајот тоа е елементот 6). Слично, можеме да отстрануваме елементи од главата од редот користејќи ја операцијата отстранување (removing). Исто како кај купот, при пишување на програмата важно е да се напише код кој нема да ги дозволи следните два случаи:

- кога редот е празен, а ние сакаме да извлечеме податоци од празен ред
- коге редот е полн, а ние сакаме да внесеме уште податоци

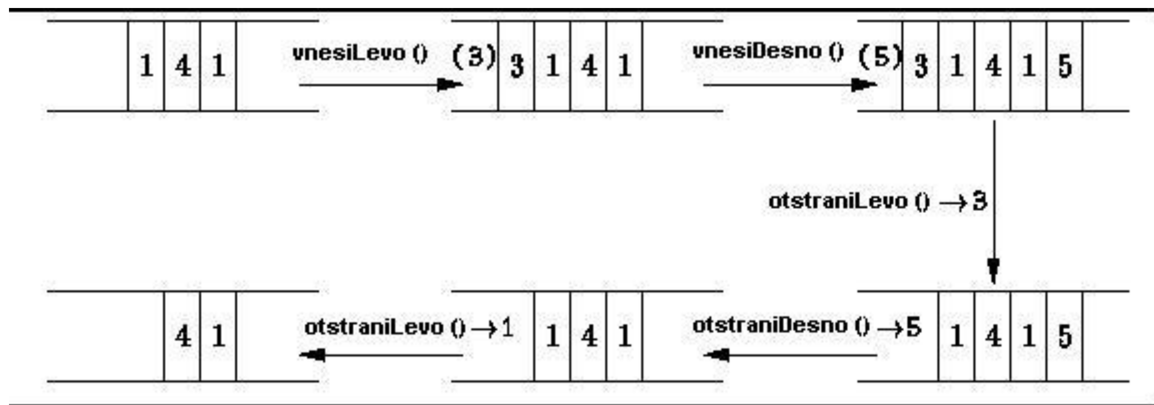
Шпил

Шпил (анг. DEQUE) е двостран ред. Можеме да сместуваме податоци од едниот крај, а да отстрануваме податоци од другиот крај, и обратно. Шпилот ги овозможува методите:

- внесиЛево (insertLeft ()), сместување елемент од главата на шпилот
- внесиДесно (insertRight ()), сместување елемент од опашката на шпилот
- отстраниЛево (removeLeft ()), отстранување елемент од главата на шпилот
- отстраниДесно (removeRight ()), отстранување елемент од опашката на шпилот

Ако само ги користиме методите за сместување и отстранување елементи од левата страна (или

нивните еквиваленти од десната страна), тогаш шпилот ќе делува како куп. Ако ги користиме методите за сместување елементи од левата страна и отстранување елементи од десната страна (или спротивниот пар), тогаш шпилот ќе функционира како ред.
На сликата е покажан начинот на функционирање на податочната структура шпил. Во првиот чекор се врши сместување на елементот 3 од левата страна, во вториот сместување на елементот 5 од десната страна, во третиот отстранување на елементот 3 од левата страна, во четвртиот отстранување на елементот 5 од десната, а во последниот се врши отстранување на елементот 1 од левата страна.



При пишување на програмата важно е да се напише код кој нема да ги дозволи следните два случаи:

- кога шпилот е празен, а ние сакаме да извлечеме податоци од празен шпил
- кога шпилот е полн, а ние сакаме да внесеме уште податоци

Сортирање

Поим

Сортирањето може да се каже дека претставува процес на преуредување на дадено множество објекти по некој однапред зададен критериум.
Кога пребаруваме по некој телефонски број во именик или збор во речник ние го користиме нашето искуство и знаење за тоа како се организирани податоците во именикот или речникот (алфабетски подредени т.е сортирани).

Ако податоците не се подредени во некој логички редослед пронаоѓањето некој податок претставува вистинска тешкотија или одзема многу време.
Целта на сортирањето е да се олесни и забрза пребарувањето на објектите.
За сортирањето може да се каже дека е една од најважните активности при обработката на податоците.

Сортирањето е проблем за кој постојат голем број разновидни алгоритми наречени алгоритми за сортирање кои меѓусебно се разликуваат по времето потребно за извршување како и потребните мемориски ресурси односно сложеност.

Формална дефиниција
Нека земеме една произволна низа $S=\{sk_1, sk_2, ..., sk_n\}$ составена од $n \geq 0$ елементи од некое универзално множество U.
Целта на сортирањето е да се преорганизираат елементите од низата S создавајќи некоја нова низа S' во која елементите од низата S ќе бидат подредени.
Но со што всушност се пртеставува подреденоста на елементите?.

Треба да дефинираме релација < која ќе претставува тотално подредување над елементите од множеството U на следниов начин:

Дефиниција:
Тотално подредување е релацијата '< 'дефинирана над елементите од некое универзално множество со следните атрибути:

1.За сите парови елементи (i,j) од UxU точно едно од следните тврдења е вистинито: ij.
2. За сите тројки (i,j,k) од UxUxU i i
(релацијата е транзитивна).

За да ги сортираме елементите од множеството S ние треба да најдеме пермутација

$P=\{p_1, p_2, ..., p_n\}$ така што за елементите од множеството S ке важи:
$sp_1 <= sp_2 <=...<= sp_n$

Во суштина ние не сме заинтересирани за пермутацијата туку нашата цел е сортираната низа $S'=\{s_1', s_2', ..., s_n'\}$ за чии елементи важи
$s_i'=sp_i$ за $1<=i<=n$.

Алготитми за сортирање

Ќе обработиме неколку алгоритми за сортирање организирани во следниве категории според главниот медот на кој се засновани:

- Сортирање со вметнување –(eng. Insertion sort)
- Сортирање со замена – (eng. Exchange sort)
- Сортирање со избор – (eng. Selection sort)
- Сортирање со мешање – (eng. Merge sort)
- Сортирање со распределба – (eng. Distribution sort)

Алгоритми за сортирање со вметнување(Insertion sort)

Основна идеја

Нека е дадена некоја низа од броеви која треба да се сортира.
Во секој чекор од алгоритмот постои дел од низата во кој елементите се сортирани (на почетокот се зема дека првиот елемент претставува сортирана низа), и исто така треба да се најде "точна" позиција на која треба да се вметне нов елемент во низата.

Ако низата се сортира во растечки редослед потребно е да се најде позиција во низата со вредност поголема од вредноста на елементот кој го вметнуваме.
Ако таква позиција нема тогаш позицијата на новиот елемент е после последниот елемент во сортираната низа.
Новиот елемент се вметнува на пронајдената позиција но притоа сите елементи од таа позиција до крајот на низата се поместуваат за едно место во десно за да се ослободи простор за елементот кој се вметнува.

Пример
Да претпоставиме дека во некој чекор од алгоритмот ја имаме следната ситуација:

A [0] = 0, A [1] =5, A [2] = 8, A [3] = 12, A [4] = 15

0	1	2	3	4	5	6	7	8	9
0	5	8	12	15					

7

и 7 е вредноста на новиот елемент кој треба да го вметнеме.

Според сугестиите наведени претходно пребаруваме низ низата A[1],A[2],…A[i] додека не го пронајдеме првиот елемент со вредност поголема од 7 (во овој случај тоа е 8),индексот на тој елемент е позицијата на која треба да биде вметнат новиот елемент додека елементите A[4],A[3],A[2] мора да бидат поместени за една позиција во десно за да се ослободи место за новиот елемент.

0	1	2	3	4	5	6	7	8	9
0	5	8	12	15					

7

позиција на која треба
да се внесе

После вметнувањето се добива следната ситуација:

0	1	2	3	4	5	6	7	8	9
0	5	7	8	12	15				

Алгоритам

Алгоритмот за реализација на овој метод се состои од следните чекори:

1.Претпостави дека првиот елемент од низата претставува сортирана низа.
2.Започни со следниот елемент од низата (несортиран дел).
3.Вметни го новиот елемент во сортираниот дел од низата на точната позиција .
4.Сортираниот дел од низата зголеми го за еден.Повторувај ја постапката се додека не се сортира целата низа.

Имплементација

Следниов програмски код напишан во програмскиот јазик Pascal покажува една имплементација на алгоритмот за сортирање со вметнување.

```
procedure insertionsort(var a:niza;n:integer);
                                        {зема за аргументи низа и должина на низата}
var i,j,temp:integer;
begin
    For i:=2 to n do
    begin
      j := i;
        temp := a[j];
        While (j>1) and (a[j-1]>temp) do
            begin
             a[j] := a[j-1];
             Dec(j);
            end;
    a[j] := temp;
    end;
end;
```

Анализа за сложеност

Најдобар случај : ако низата е веќе сортитана тогаш внатрешниот while циклус нема да се изврши,а надворешниот for циклус ке се изврши n-1 пати од каде се добива сложеност O(n).
Најлош случај:ако низата е сортирана во обратен редослед тогаш имаме максимален број на извршување на внатрешниот while циклус од каде се добива сложеност O().

Алгоритмот кој го анализиравме користи линеарно пребарување за позицијата на која вршиме вметнување, но бидејќи може да се забележи дека целната подниза во која вршиме вметнување е веќе сортирана , може да се искористи побрз алгоритам за пронаоѓање на местото на кое треба да се вметни следниот елемент.

Идеален алгоритам за пребарување е алгоритмот на бинарно барање од каде и алгоримот за сортирање го добива името алгоритам за сортирање со бинарно вметнување.

Значи во оваа модифицирана верзија на алгоритмот за сортирање со вметнување ние всушност користиме бинарно пребарување (поефикасен алгоритам ,со логаритамската комлексност) за позицијата на која треба да вршиме вметнување,
додека останатиот дел од алгоритмот е потполно идентичен.

Останува непроменет бројот на поместувања на податоците во десно за ослободување на место за вметнување.

Ефикасноста на овој алгоритам во најдобар случај е O(nlogn),
додека во најлош случај O().

Следниот програмски код напишан во програмскиот јазик Pascal е една имплементација на алгоритмот за сортирање со бинарно вметнување.

```
procedure binaryinsertionsort(var a:niza;n:integer);
var
    temp,i,j,l,r,m: integer;
begin
  for i:=2 to n do
   begin
   temp := a[i];

      l := 1;
      r := i-1;
       while l<=r do
         begin
           m := (l+r) div 2;        {koristime binarno baranje za pozicijata na koja}
            if  temp < a[m] then   {treba da go vmetneme noviot element}
                  r := m-1
                     else
                  l := m+1
         end;{while}

      for j:= i-1 downto l do
      a[j+1]:= a[j];
   a[l]:= temp;
  end;{for}
end;{procedure}
```

Сложеноста на овој алгоритам се разгледува во однос на пребарување за позицијата на новиот елемент и во најдобар случај е O(nlogn).

Алгоритми за сортирање со замена (eng Exchange Sort)

Втората класа на алгоритми за сортирање кои ке ги разгледаме се алгоритмите за сортирање со замена на пар елементи се додека низата не е сортирана.

Ќе ги разгледаме методите:

- Метода на меурче (eng Bubble Sort)
- Брзо сортирање (eng Quick Sort)

Сортирање со метода на меурче (eng Bubble Sort)

Основна идеја

За да се сортира низата $a_1, ..., a_n$ со оваа метода потребно е да се направат n-1 премини во низата.Во секој премин се споредуваат парови соседни елементи и ако има потреба т.е. ако елементите кои се споредуваат не се во бараниот редослед , тие си ги заменуваат местата.

После првиот премин најмалиот елемент од низата ќе дојде на првата позиција од низата (избива на површината како меурче во чаша вода). После k премини над низата последните k елементи од низата се на "точната" позиција и не треба повеќе да се разгледуваат. За да се осигураме дека низата е сортирана комплетно потребни се n-1 премини иако таа може да биде сортирана и порано.

Пример
Да се сортира низата 234,77,0,113,404,60 со помош на методата на меурче:

```
Премин број 1:
   77  234    0  113  404   60
   77    0  234  113  404   60
   77    0  113  234  404   60
   77    0  113  234  404   60
   77    0  113  234   60  404

 Број на замени во овој премин = 4

 Премин број 2:
   0    77   113    234   60  404
   0    77   113    234   60  404
   0    77   113    234   60  404
   0    77   113     60  234  404
  Број на замени во овој премин=2

Премин број 3:

    0  77  113  60  234  404
    0  77  113  60  234  404
    0  77  60  113  234  40
Број на замени во овој премин=1

Премин број 4:
  0  77  60  113  234  404
  0  60  77  113  234  404
Број на замени во овој премин=1
```

```
После овој премин низата е веќе сортирана иако алгоритмот за да го провери тоа мора да
направи уште еден премин над елементите.
```

Алгоритам

1.Повторувај ги чекорите од 2 до 4 се додека се можни замени
2. Од еден до должината на низата минус еден извршувај чекор 3
3.Ако соседните елементи (првиот со вториот,вториот со третиот итн.)не се во бараниот редослед замени им ги местата.
4.Должината на низата над која се извршува алгоритмот намали ја за еден.
5.Ако се можни замени оди на чекор 2.
Овој алгоритам гарантира дека после секој премин сигурно еден елемент од низата доаѓа на точната позиција.

Имплементација

Овој алгоритам е многу едоставен за имплементација и разбирање.
Следниов програмски код прикажува една негова имплементација во програмскиот јазик Pascal.

```
Procedure bubblesort(var a:niza;n:integer);
var temp,i,j: integer;
begin
    for i:= n downto 2 do
        begin
            for j:= 2 to i do
                begin
                    if (a[j-1] > a[j]) then
                        begin
                          temp:= a[j-1];
                          a[j-1]:= a[j];
                          a[j]:= temp;
                        end;{if}
                end;{for j}
        end;{for i}

End;
```

Анализа

Овој алгоритам има квадратна сложеност $O(n^2)$ бидејќи и во најдобар случај кога низата е сортирана потребни се истиот број на споредби за да со сигурност се потврди дека низата е сортирана.Можни се некои модификации над првичниот алгоритам со кои се добива на забрзување со тоа што ако низата се сортира пред сите можни премини алгоритмот ќе ја открие таквата состојба и ќе го прекине понатамошното извршување.

Метод на брзо сортирање (eng QuickSort)

Овој алгоритам е од типот раздели па владеј (eng divide-and-conquer).

Еден алгоритам е од типот раздели па владеј ако проблемот го решава така што најпрвин го разделува на помали подпроблеми,а потоа решавајќи ги подпроблемите и комбинирајќи ги нивните решенија доаѓа до бараното решение т.е решението на основниот проблем.

Основна идеја

Нека $a_1, ..., a_n$ е дадена низа која треба да се сортира.
Методот на брзо сортирање проблемот на сортирањето го решава на следниот начин:

-еден елемент од низата се прогласува за главна точка-пивот(eng pivot),се пронаоѓа и поставува на “точната“ позиција на која треба да стои.
-елементите од низата се делат на две поднизи во првиот дел одат елементите кои се помали од главната точка ,а во вториот дел оние кои се поголеми.
-алгоритмот рекурзивно се применува за двете поднизи се додека не се дојде до интервал со должина еден.

Пример:
Да се сортираат елементите 16,7,9,44,2,18,8,53,1,5,17 со помош на методата за брзо сортирање.Алгоритмот се изведува со помош на користење на два индекси иницијално поставени едниот на почеток а другиот на крајот од низата.

Го избираме првиот елемент #1(16) како главна точка и продолжуваме со пребарување,откриваме дека четвртиот елемент #4(44) е првиот поголем елемент од 16 и застануваме тука ,сега започнуваме со елементот #11(17) на крајот од низата и работиме наназад,елементот #10(5) е првиот елемент помал од 16 и тука вршиме смена со елементот на четвртата позиција #4(44).
Ја добиваме следната ситуација:

16 7, 9, 5, 2, 18, 8, 53, 1, 44, 17,

Подвлечените елементи се тие два елементи кои си ги заменија местата се разбира на индексите каде што застанавме со броењето од почетокот и крајот на низата.

Продолжуваме,следните два елементи кои треба да си ги заменат местата затоа што се наоѓаат на погрешни страни од низата се елементите #6(18) и #9(1)со што добиваме:

16 7, 9, 5, 2, 1, 8, 53, 18, 44, 17,

Продолжуваме следниот поголем елемент од пивотот е елементот #8(53) но тука веќе индексите кои ни покажуваат до каде сме стигнале се изедначуваат со што не се можни повеќе замени па пивотот се поставува на точната позиција (индекс минус еден) вршејќи замена со елементот кој е на таа позиција со што се добива:

8, 7, 9, 5, 2, 1, 16, 53, 18, 44, 17,

Сега низата е поделена на две поднизи,а елементот 16 е на “точната“ позиција и не треба повеќе да се разгледува (оттука името главна точка).Размислувајќи рекурзивно истата постака може да се примени за двете поднизи со што би се сортирала целата низа.

Повторувајќи ја постапката за левата подниза добиваме:

8 7, 9, 5, 2, 1, 53, 18, 44, 17,

8 7, 1, 5, 2, 9, 16 53, 18, 44, 17,

2, 7, 1, 5, 8, 9, 16 53, 18, 44, 17,

Повторувајќи ја постапката за левата подниза околу 8 , се добива:

2 7, 1, 5, 8, 9, 16 53, 18, 44, 17,

2 1, 7, 5, 8, 9, 16 53, 18, 44, 17,

1, 2, 7, 5, 8, 9, 16 53, 18, 44, 17,

Левата подниза со главна точка 2 е сортирана(еден елемент).Кога и десната подниза со главна точка 2 (два елементи) ке се исортира добиваме:

1, 2, 5, 7, 8, 9, 16 53, 18, 44, 17,

Со повторување на истата процедура ќе се добие следната слика:

1, 2, 5, 7, 8, 9, 16 53, 18, 44, 17,

1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

1, 2, 5, 7, 8, 9, 16 17, 18, 44, 53,

Алгоритам

Алгоритмот за реализација на овој метод се состои од следните чекори:

1.Постави го првиот елемент од низата за главна точка-пивот.
2.Најди ја точната позиција на тој елемент и постави го таму.
3.Распореди ги елементите од низата т.ш. елементите кои се пред
главната точка-пивот се помали,а тие после неа поголеми од неа.
4.Повторувај ги чекорите рекурзивно за левата подниза се додека има повеќе од еден елемент.
5.Повторувај ги чекорите рекурзивно за десната подниза се додека има повеќе од еден елемент.

Алгоритмот гарантира дека во секој чекор еден елемент ке дојде на вистинското место.

Имплементација:

Следниот програмски код напишан во програмскиот јазик Pascal е една имплементација на алгоритмот за брзо сортирање.

```
procedure quicksort(var a:niza;lo,hi:integer);
procedure Sort(l, r: integer);
 var
     i, j, x, y: integer;
 begin
    i := l;
    j := r;
    x := a[l];
       repeat
               while a[i] < x do i := i + 1;
               while x < a[j] do j := j - 1;
                 if i <= j then begin
                      y := a[i];
                      a[i] := a[j];
                      a[j] := y;
                      i := i + 1;
                      j := j - 1;
                 end;
       until i > j;
if l < j then Sort(l, j);
if i < r then Sort(i, r);
end;

begin
Sort(lo,hi);
end;
```

Анализа

Најдобар случај за овој алгоритам има кога во секој чекор од рекурзијата од поделбата на низата настануваат точно два дела со еднаква должина,тогаш длабочината на рекурзијата е log(n).Од тука следува дека комлексноста е O(nlog(n)).

Најлош случај има кога во секој чекор на рекурзијата настанува небалансирана поделба,имено кога едната подниза се состои само од еден елемент а другата од останатите,тогаш длабочината на рекурзијата е n-1 па за сложеноста на алгоритмот се добива $O(n^2)$. Можни се модификации за забрзување на овој алгоритам и тоа во делот при изборот на елемент пивот.

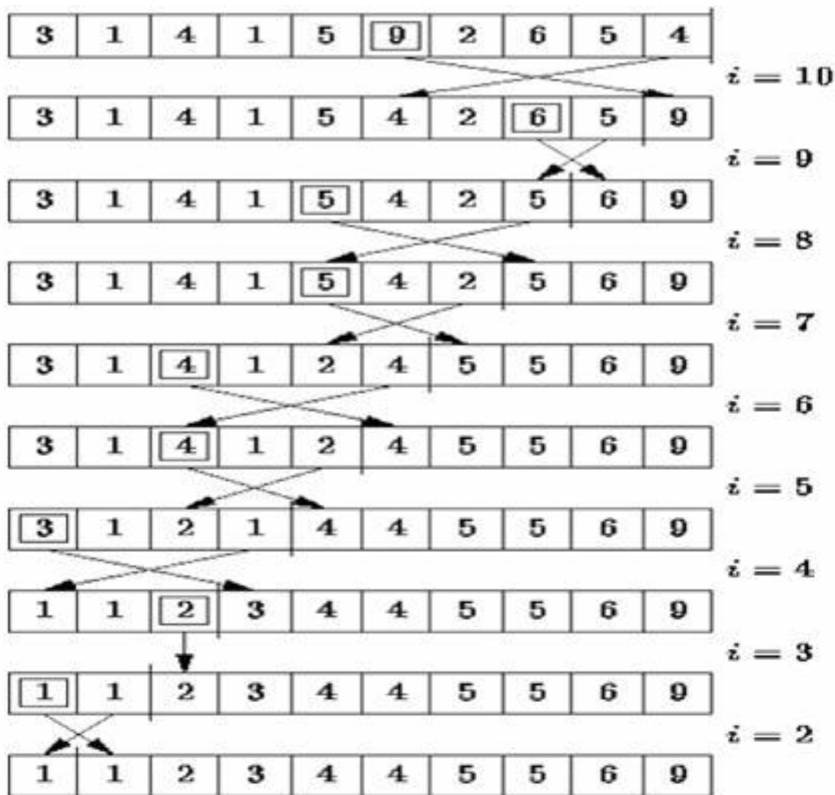
Алгоритми за сортирање со избор (eng Selection Sort)

Идејата за решавање на проблемот на сортирање кај овие алгоритми е следната:
во секој чекор елементот кој се додава во сортираната низа се избира од останатите елементи.
Кај овие видови алгоритми вметнувањето нови елементи се врши од еден крај на низата што ги прави различни од алгоритмите за сортирање со вметнување каде вметнувањето се врши на произволна позиција.

Основна идеја

За реализација на овој алгоритам потребни се n-1 премини во низата. Во секој чекор на алгоритмот вршиме линеарно барање низ несортираниот дел од низата барајќи ја позицијата на најголемиот елемент.Тој елемент го поставуваме на крајната позиција вршејќи замена со елементот кој се наоѓа на таа позиција,после секој чекор должината на низата над која се извршува алгоритмот ја намалуваме за еден.

Пример
Да се сортира низата 3,1,4,1,5,9,2,6,5,4 со помош на методата за сортирање со избор.При секој премин го избираме најголемиот елемент и го поставуваме на точната позиција вршејќи замена со елементот кој е на таа позиција.



Алгоритам

Алгоритмот може да се опише со следните чекори:

1. Со линеарно барање најди го максимум на елементите од низата.
2.Замени ги крајниот елемент од низата и максумумот на низата
3.Повтори ја постапката за преостанатите n-1 елементи,потоа за n-2 итн се додека не остане само еден т.е. најмалиот елемент.

Имплементација

Следниот програмски код напишан во програмскиот јазик Pascal е една имплементација на алгоритмот сортирање со избор.

```
Procedure selectionsort(var a:niza;n:integer);
Var
        i,j,temp,m: Integer;
Begin
    for i:=n  downto 2 do
      begin
        m:=1;
           for j:=1 to i do
             begin
           if (a[j] > a [m]) then
                  m:=j;
           end;

                    temp:=a[i];
                    a[i]:=a[m];
                    a[m]:=temp;

        end;
end;
```

Анализа

Овој алгоритам има константна сложеност и тоа O(n^2) бидејќи секогаш се извршуваат истиот број на споредби и замени.

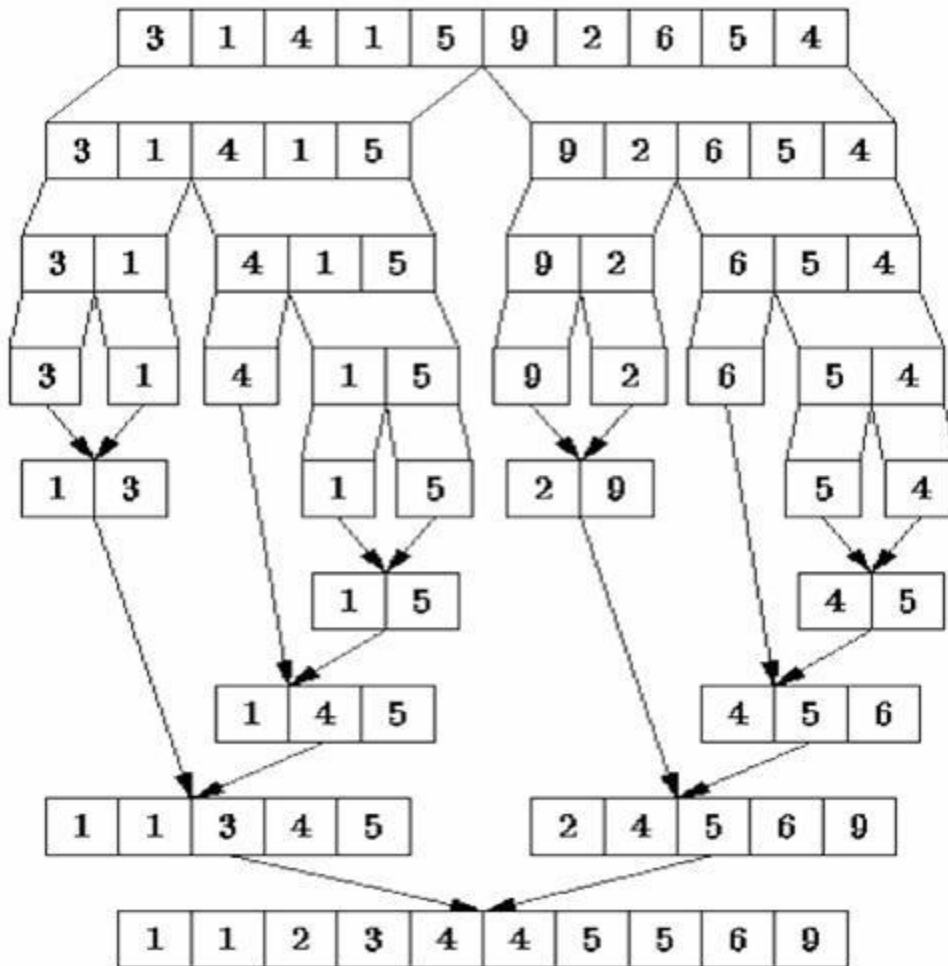
Друг алгоритам од оваа класа на алгоритми за сортирање со избор е алгоритмот за сортирање со Heap.Овој алгоритам има оптимална сложеностO(nlog(n)).

Алгоритми за сортирање со мешање (eng Merge Sort)

Основна идеја

Идејата за сортирање кај овој алгоритам е слична на идејата на алгоритмот за брзо сортирање.И овој алгоритам е од типот раздели па владеј,најпрво низата која треба да се сортира ја разделува на половина т.е. на две поднизи и потоа секоја подниза се сортира независно, на крај двете сортирани поднизи се мешаат со што се добива оригиналното решение.

Пример
Со помош на метод на претопување да се сортира низата 3,1,4,1,5,9,2,6,5,4.
Во секој чекор низата се разделува на половина и рекурзивно се повикува процедурата за сортирање за левата и десната подниза,и потоа процедурата за мешање.



Алгоритам

Алгоритмот може да се опише со следните чекори:

1.Раздели ја низата на две поднизи со должина n/2 ,n/2
2.Рекурзивно сортирај ја секоја од двете поднизи.
3.Измешај ги сортираните поднизи за да се добие крајниот резултат.

Базен случај за алгоритмот е подниза со точно еден елемент и таа се зема за сортирана.

Имплементација

Постојат повеќе начини за имплементација на процедурата за мешање (merge) но таа најчесто се имплементира на следниот начин:

Двете поднизи кои се мешаат најпрво се копираат во една помошна низа,потоа двете поднизи сега составен дел од помошната низа со помош на индекси кои покажуваат кои се нивни делови се преминуваат и помалиот елемент од поднизите се копира назад во почетната низа.Може да настане ситуација така што едниот индекс да дојде до крајот а другиот не, што означува дека во едната пониза има уште елементи и тогаш природно е преостанатите елементи да се ископираат на крајот од почетната низа.

Следниот програмски код напишан во програмскиот јазик Pascal е една имплементација на алгоритмот за сортирање со мешање.

```
Procedure mergesort(Var a: niza; lo, hi: Integer);
Label return;
Var end_lo,k,mid,start_hi: Integer;
   T: integer;
Begin
if lo>=hi then goto return;
  mid := (lo + hi) Div 2;
  {Podeli ja listata na dve podnizi i sortiraj gi rekurzivno}
  mergesort(a, lo, mid);
  mergesort(a, mid + 1, hi);
  {Izmesaj gi dvete podnizi vo edna sortirana}
  start_hi := mid + 1;
  end_lo := mid;
  while (lo <= end_lo) and (start_hi <= hi) do
  begin
   if a[lo] < a[start_hi] then
     Inc(lo)
   else
   begin
     T := a[start_hi];
     for k := start_hi - 1 Downto lo do a[k+1] := a[k];
     a[lo] := T;
     Inc(lo);
     Inc(end_lo);
     Inc(start_hi)
     end
   end;
  return: End;
```

Анализа

Процедурата за мешање побарува најмногу 2n чекори (n чекори за копирање на низата во помошната низа и n чекори за враќање на елементите назад),од каде за комплексноста на овој алгоритам добиваме :

$T(n) \leq 2n + 2\ T(n/2)$ и

$T(1) = 0$

од каде со решавање на оваа рекурентна равенка се добива

$T(n) \leq 2n \log(n)\ \varepsilon\ O(n \log(n))$.

Алгоритми за сортирање со распределба (eng Distribution Sort)

Главна карактеристика на овие алгоритми е тоа што тие не користат споредби за да извршат сортирање.

Bucket Sort

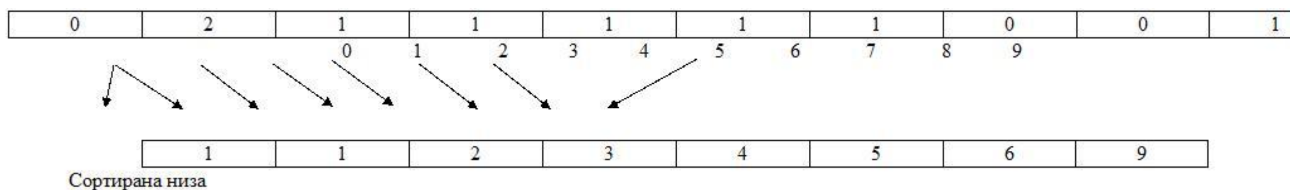
Овој алгоритам е можеби наједноставниот алгоритам за сортирање со распределба. Суштинското барање на овој алгоритам е тоа што големината на множеството од каде што се елементите кои ги сортираме е мало и константно.

На пример да претпоставиме дека треба да ги сортираме елементите од множеството {0,1,... n-1} т.е. подмножество од множеството цели броеви.
Овој алгоритам користи n бројачи, i-от бројач чува број на појавувања на i-от елемент од множеството.
Пример:
Да се сортира низата 3,1,4,1,5,9,2,6 со методот BucketSort.

3	1	4	1	5	9	2	6

Почетна низа

Низа од бројачи со должина колку максималниот елемент плус еден



Имплементација

Следниот програмски код напишан во програмскиот јазик Pascal е една имплементација на овој алгоритмот за сортирање.

```
procedure bucketsort(var a:niza;n:integer;m:integer);
              {argumenti: a-niza,n-broj na elementi,m-maksimalen element na nizata}
 var bukets:niza;     { treba array[1..maximalen_od_nizata] of integer;}
      pom,i,j,k:integer;
begin
for j:=0 to m do {inicijalizacija}
begin  bukets[j]:=0; end;
for i:=1 to n do
     begin
       pom:=a[i];
       bukets[pom]:=bukets[pom]+1;
     end;



   i:=0; {generiranje na nizata}
   for j:=0 to m do
     begin
        for k:=1 to bukets[j]  do
         begin
            i:=i+1;
            a[i]:=j;
         end;
     end;
end;
```

Анализа

Времето потребно за извршување на овој алгоритам е од ред O(m+n).

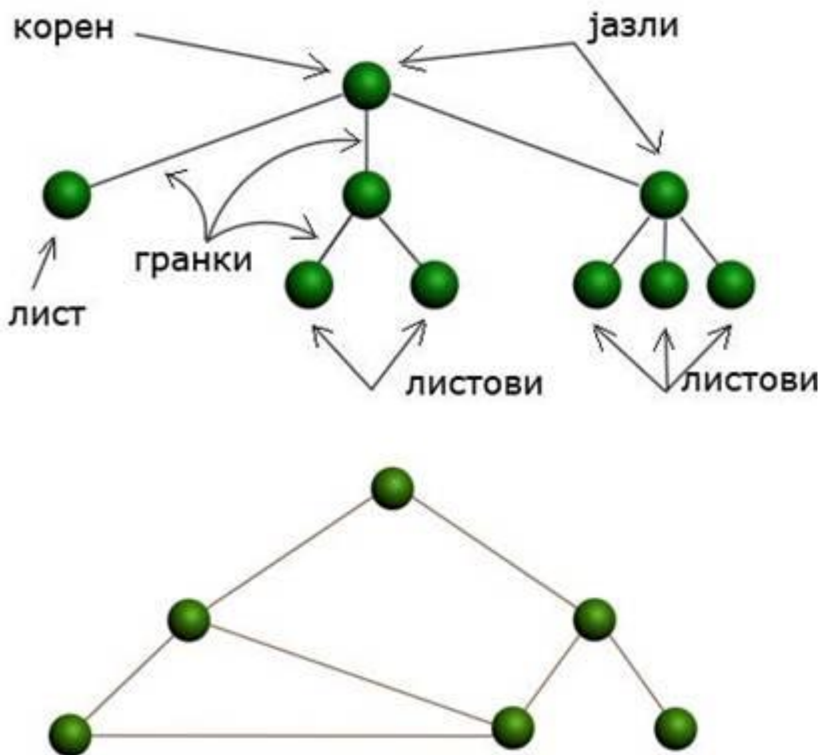
Дрва

Вовед

Структурата на дрвата е навистина пробив во организацијата на податоците бидејќи тие дозволуваат да имплементираме алгоритми кои се многу побрзи отколку кога користиме линеарна податочна структура. Дрвата се основна податочна структура за зачувување на податоците користена во програмирањето. Тие даваат многу поголема предност во однос на податочните структури што ги видовме досега(низи,куп,ред,шпил). Релациите во дрвото се хиерархиски подредени со некои објекти "над" и некој "под" другите. Главната терминологија за податочната структура е "родител" – "дете".

Што е Дрво?

Дрво е апстрактна податочен тип кој ги чува елементите хиерархиски. Со исклучок на најгорниот елемент, сите имаат елемент "родител" и ниедно или повеќе "деца". Најгорниот елемент го викаме корен на дрвото и тој се црта како највисок елемент.



Зошто да користиме дрва?

Дрва би користеле затоа што ни даваат комбинирани предности на две други структури : подредена низа и поврзана листа. Во дрвото можеме да пребаруваме многу брзо исто како во подредена низа, и можеме брзо да внесуваме и бришеме елементи како во поврзана листа.

Споро внесување на елементи во подредена низа

Замислете си низа каде што елементите се подредени по ред т.е. подредена низа. За ваква низа пребарувањето е многу брзо за некоја вредност, како на пример користејќи binary search. Прво пребаруваме во центарот на низата, ако вредноста на елементот што го бараме е поголема од онаа

вредност на центарот, тогаш продолжуваме со десната страна на низата инаку со пребарувањето продолжуваме со левата половина на низата. Ваквото пребарување ни овозможува брзина од O(log n).

Од друга страна, ако сакаме да внесуваме објект во подредена низа прво треба да ја најдеме позицијата каде што треба да го внесеме елементот, и потоа да ги поместиме сите објекти на десно што се со поголема вредност за да направиме место за нашиот објект. Овие повеќекратни поместувања ни одземаат од процесорското време (N/2 поместувања). Истото важи и за бришење на елемент од подредената низа.

Споро пребарување во поврзана листа

Од друга страна пак, внесувањето и бришењето на елементите во поврзаната листа се одвива многу брзо. Овие операции имаат брзина од O(1) време на извршување. За жал пак, ова наогање на елемент во поврзаната листа се одвива многу споро.

Секогаш мора да се почнува од почеток и да се посети секој елемент додека не се најде оној што го бараме. Така треба да посетиме по просек од N/2 елементи споредувајќи го секој клуч на елементот со посакуваниот клуч на елементот. Ова е споро, има време на извршување од O(N).

Некои би помислиле дека би можеле да ги забрзаат работите користејќи подредена поврзана листа каде што елементите се подредени по ред, но ова не би им помогнало. Сепак мора да се почнува од почеток и да се посетат сите елементи во листата, бидејќи не може да се пристапи до одреден елемент без да се следи синџирот од референци до него.

Предноста на дрвата

Би било убаво ако постои податочна структура со брзо внесување и бришење на елементите како што е поврзаната листа, и исто така брзо пребарување како што е во подредена низа. Дрвото ги овозможува овие две карактеристики.

Апстрактна податочна структура – Дрво

Дрвото T е множество од јазли кој ги зачувува елементите во релација родител – дете со следниве можности :

- T има специјален јазол r, наречен корен.
- Секој јазол v од T различен од r има јазол родител n.

Во компјутерските програми јазлите најчесто ни претставуваат такви ентитети како што се луѓе, делови на коли,авионски резервации, итн, со други зборови кажано, вредности што ги зачувуваме во било која податочна структура.

Гранките се тие коишто ги поврзуваат јазлите. Едно дрво велиме дека е подредено ако има линеарно подредување дефинирано на децата. Во секој јазол, односно дека може да биде прво, второ, трето итн. Подредувањето е одредено според тоа како сакаме ние, обично се подредуваат од лево на десно.

Карактеристики на двата

Пат
Низата што е добиена како резултат на движење низ дрвото од еден јазол кон друг (гранките) се нарекува пат.

Корен
Јазолот којшто се наоѓа на врвот на дрвото се вика корен. Постои само еден корен во дрвото. За една колекција од дрва и гранки што го дефинираат едно дрво, мора да постои еден (и само еден!) пат од коренот до секој останат јазол.

Родител
Секој јазол (освен коренот) има точно една гранка којашто оди нагоре до друг јазол.Јазолот над тој јазол се вика јазол – родител.

Дете
Било кој јазол може да биде поврзан со еден или повеќе јазли надолу по дрвото. Јазлите под даден јазол се викаат негови деца.

Лист (надворешен јазол)
Јазол којшто нема ниту едно едете се вика лист или надворешен јазол. Во едно дрво може да има само еден корен, но повеќе листови.

Поддрво
Поддрво T1 од даденото дрво T со корен во јазолот v е дрво кое се состои од сите елементи под v во T1.
Линиите што се графички претставени кои ги поврзуваат јазлите всушност претставуваат нивните сродни врски.

Поминувања на дрва
Да се помине едно дрво значи да се посетат сите негови јазли по специфичен редослед. Подоцна ќе се запознаеме со поминувања во дрва.

Имплементација на дрва во Pascal

```
type pokazuvac = ^jazol;

jazol = record

Levo, desno: = pokazuvac;

vrednost := char;

end;
```

Алгоритми со дрва

Длабочина на јазол v

Нека v е јазол на дрво T. Длабочина на v е бројот на предци на v вклучувајќи го и него самиот.Длабочината на дрво T може да се дефинира рекурзивно на следниов начин :
Ако v е корен тогаш длабочината на v е 0,
Инаку, длабочината на v е еден плус длабочината на родителот на v

Висина на јазол v

Висина на јазол v исто така се дефинира рекурзивно
Ако v е надворешен јазол, тогаш висината е 0. Инаку, висината на v е 1 + максималната висина на

дете на v.
Висината на дрво T е еднаква на максималната длабочина на надворешен јазол на T.

Видови на дрво

Постојат различни видови на дрва и тоа :

- Бинарни дрва
- АВЛ дрва
- Multi way дрва
- 2 - 4 дрва
- B - дрва

Бинарни дрва

Бинарно дрво е подредено дрво кадешто секој јазол може да има најмногу две деца.
Јазол во едно бинарно дрво не мора да значи дека има точно две деца, може да има само едно лево дете,само едно десно дете или може воопшто да нема деца (во тој сучај станува збор за надворешен јазол). Во правилно бинарно дрво секој внатрешен јазол има точно две деца или ниедно дете. Овие деца се подредени така што левото дете доаѓа пред десното дете.

Можности на бинарни дрва

Бинарните дрва имаат интересни можности во справувањето со врските меѓу висината и бројот на јазли. За множество од сите јазли од дрво T на иста длабочина d велиме дека е ниво d на T. Во бинарно дрво нивото 0 има еден јазол, нивото еден има 2 јазли, нивото два има четири јазли итн. Општо нивото d има 2^d јазли. Максималниот број на јазли на нивоата од бинарното дрво расте експоненцијално надолу по дрвото.
Нека T е правилно бинарно дрво со n јазли и нека h е висина на T.Тогаш T ги има следниве можности:

- Бројот на надворешни јазли во T е најмалку h+1, а најмногу 2^h
- Бројот на внатрешни јазли во T е најмалку h, и најмногу 2^h-1
- Точниот број на јазли во T е најмалку 2*h+1 и најмногу 2^(h+1)-1
- Висината на T е најмалку log(n+1)-1 и најмногу (n-1)/2

Дефиниција на податочна структура на бинарни дрва во Pascal

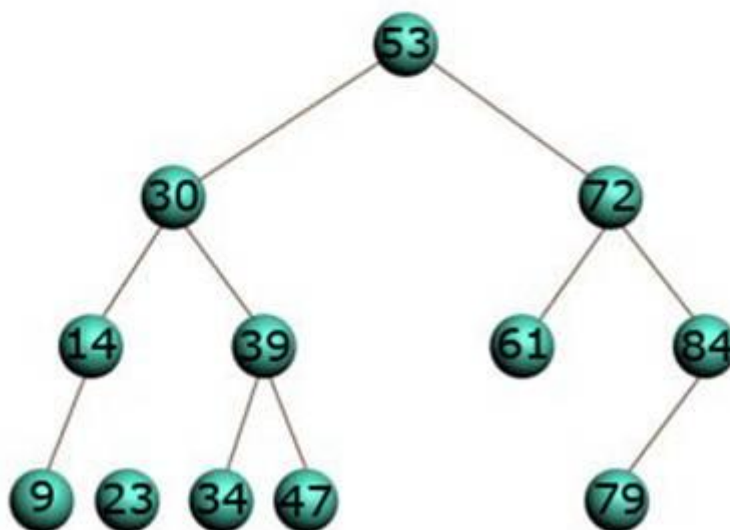
```
type pokazuvac = ^jazol;

jazol = record

vrednost : char;

levo, desno : pokazuvac;

end;
```

PreOrder премин на дрво

PreOrder премин на дрво е кога ќе ги поминеме сите јазли од дрвото по следниов редослед: корен, лево поддрво, десно поддрво. Овој вид на премин на дрвото е многу корисен кога пишуваме програми што анализира алгебарски изрази.

Бинарни пребарувачки дрва

Пребарувачки бинарни дрва се оние дрва кај кои нумеричките вредности на јазлите во левото поддрво на даден јазол се помали од него, а сите нумерички вредности на јазлите во десното поддрво на даден јазол се поголеми од него. На овој начин не мора да се шета низ целото дрво за да се провери дали постои даден елемент во дрвото.



Графички приказ на бинарно пребарувачко дрво

Алгоритмот за пребарување во бинарно дрво е

```
    procedure baraj( kluc : char; koren : pokazuvac );
begin
if koren=nil then {*** Не е најден ***}
        begin
           {* notfound( kluc )    *}
        end
else if koren^.vrednost = kluc then {*** Најден ***}
        begin
           {* notfound( kluc )    *}
        end
else if koren^.vrednost < kluc then baraj( kluc, koren^.vrednost )
     else                 baraj( kluc, koren^.vrednost )
end;
```

Алгоритмот за внесување на елемент во бинарно дрво е

```
procedure vmetnijazol(novjazol:pokazuvac; var koren:pokazuvac);
 begin
      if koren=nil
         then
             begin
                  koren:=novjazol;
                  novjazol^.levo:=nil;
                  novjazol^.desno:=nil
             end
         else
             if novjazol^.vrednost<koren^.vrednost
                then
                    vmetnijazol(novjazol,koren^.levo)
                else
                    vmetnijazol(novjazol,koren^.desno);
 end;
```

Поминувања на дрва

Поминување на дрво значи да се помине секој јазол од дрвото точно еднаш. Имаме 3 видови на поминување на бинарни дрва и тоа :

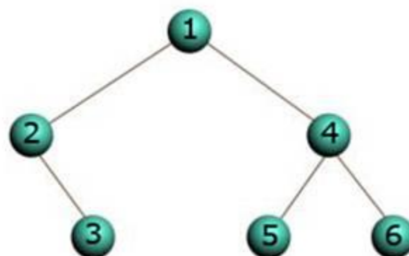
- PreOrder премин
- InOrder премин
- PostOrder премин

Во главно, рекурзивната структурна формула за поминување на непразно бинарно дрво е следната : На јазолот N мора да се направат следниве три работи : ·

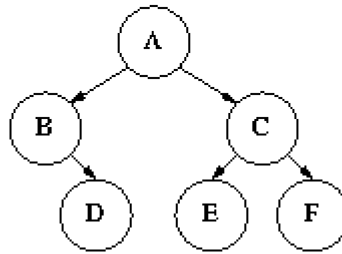
- Рекурзивно помини го левото поддрво. Кога овој чекор е завршен се наоѓате на јазолот N.
- Рекурзивно помини го десното поддрво. Кога овој чекор е завршен се наоѓате на јазолот N.
- Процесирај го јазолот N.

PreOrder премин на дрво

PreOrder премин на дрво е кога ќе ги поминеме сите јазли од дрвото по следниов редослед: корен, лево поддрво, десно поддрво.Овој вид на премин на дрвото е многу корисен кога пишуваме програми што анализира алгебарски изрази.



Графички приказ на PreOrder премин на дрво

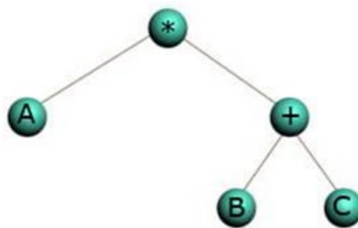


Се чита A-B-D-C-E-F

Имплементација на PreOrder во Pascal

```
procedure preorder(koren:pokazuvac);
begin
     if koren<>nil
        then
            begin
                 write('pritisnete <ENTER> za ispis na vrednosta na sledniot jazol');
                 readln;
                 pisijazol(koren^);
                 inorder(koren^.levo);
                 inorder(koren^.desno);
            end;
end;
```

Бинарно дрво (не бинарно пребарувачко дрво) може да се користи како претставување на аритметички израз вклучувајќи ги бинарните аритметички операции +,-,*,/. Коренот на дрвото содржи оператор и секој од неговите корени претставуваат или име на некоја променлива (како А,В,С итн) или друг израз.



Аритметички израз претставен со дрво

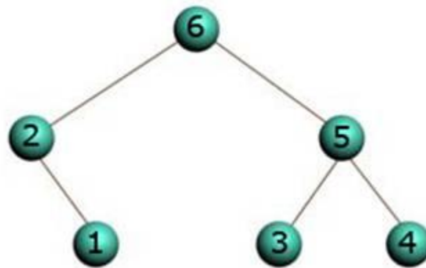
Користејќи го PreOrder преминот на дрво ќе ја добиеме префиксната нотација *A+BC

PostOrder премин на дрво

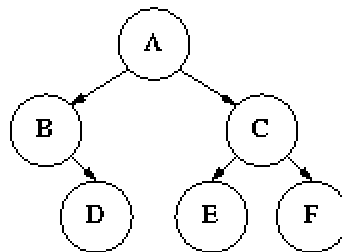
PostOrder премин на дрво е кога ќе ги поминеме сите јазли од дрвото по следниов редослед: лево поддрво, десно поддрво, корен. Посетувајќи го дрвото со PostOrder премин на сите јазли ќе ни ја генерира следниов аритметички израз ABC+*

Имплементација на PostOrder во Pascal

```
procedure postorder(koren:pokazuvac);
begin
     if koren<>nil
        then
            begin
                 inorder(koren^.levo);
                 inorder(koren^.desno);
                 write('pritisnete <ENTER> za ispis na vrednosta na sledniot jazol');
                 readln;
                 pisijazol(koren^);
            end;
end;
```



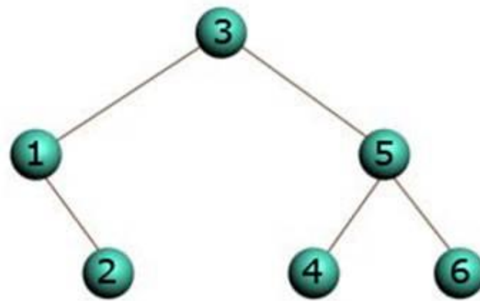
Графички приказ на PostOrder премин на дрво



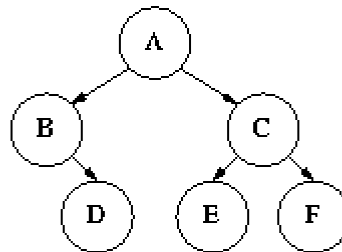
Се чита D-B-E-F-C-A

InOrder премин на дрво

InOrder премин на дрво е кога ќе ги поминеме сите јазли од дрвото по следниов редослед: лево поддрво, корен, десно поддрво.



Графички приказ на InOrder премин на дрво



Се чита B-D-A-E-C-F

Имплементација на Inorder во Pascal

```
procedure inorder(koren:pokazuvac);
begin
     if koren<>nil
        then
            begin
                 inorder(koren^.levo);
                 write('pritisnete <ENTER> za ispis na vrednosta na sledniot jazol');
                 readln;
                 pisijazol(koren^);
                 inorder(koren^.desno);
            end;
end;
```

AVL дрва

Проблемот со постигнувањена логаритамско време на извршување на операциите се разрешува со тоа што во дефиницијата за бинарно пребарувачко дрво секогаш ќе гледаме да се добие логаритамска висина за дрвото. Правило на кое ќе сметаме е балансирање на висината, што ја карактеризира структурата на бинарно пребарувачко дрво T со висина на неговите внатрешни јазли.

Својство за балансирање на висината : За секој внатрешен јазол v на T,висините на децата од v може да се разликуваат најмногу за 1.

Било кое бинарно пребарувачко дрво T кое го задоволува ова својство се нарекува AVL дрво, и е наречено по пронаоѓачите Adelson-Velski и Landis.

Последица од својството за балансирање е тоа дека поддрвото на AVL дрво е AVL дрво .
Својство: Висината на AVL дрво T кое има n членови е O(log n).
Според ова својство операцијата за наоѓање на елементи има време на извршување O(log n).

Внесување на елементи во AVL дрво:

Се внесува нов член во јазолот w во T кој претходно бил надворешен јазол и правиме w да стане внатрешен јазол со додавање на две надворешни "деца" на w. Оваа акција може да влијае на балансирање на висината, за некои јазли се зголемува висината за еден. Затоа треба да го реструктуираме T за да се блалансира висината.

За дадено бинарно пребарувачко дрво T, велиме дека јазолот v од T е балансиран ако апсолутната вредност на разликата меѓу висините на децата на v е најмногу 1 и велиме дека е небалансиран во друг случај. Значи, својството за балансирање на висината која го карактеизира AVL дрвото е исто со тоа секој внатрешен јазол да е балансиран.
Нека T го задоволува својството за балансирање на висината, по внесување нов член на T, висините на некои јазли вклучувајќи го и w, се зголемуваат. Сите такви јазли се на патеката од коренот,и тие се единствените јазли кои не се балансирани. За T да биде AVL дрво треба да ја поправиме оваа небалансираност.

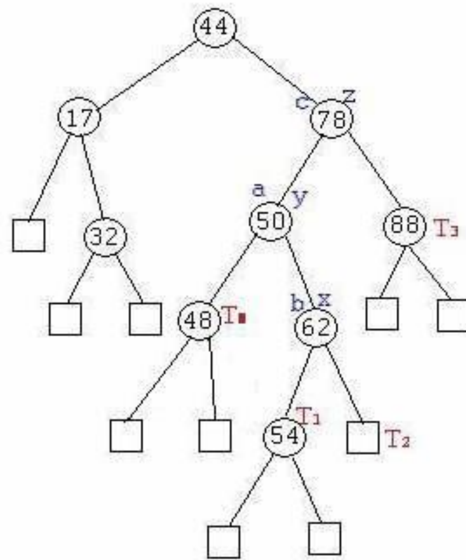
Ја регулираме балансираноста на јазлите од бинарно пребарувачко дрво T со едноставна стратегија "најди и поправи". Нека z е првиот јазол кој сме го вброиле одејќи нагоре од w до коренот на T кој што е небалансиран. Исто така y е дете на z со поголема висина. Нека x е дете на y со поголема висина, притоа x може да биде еднаков на w и x е внук на z. Бидејќи z е небалансиран поради внесување во дрвото со корен во y, висината на y е за 2 поголема од висината на оној што е на исто ниво со него. Сега го ребалансираме поддрвото со корен во z , со тоа што привремено ги преименуваме x,y и z како a,b и c, така што a претходи на b и b на c кога T се поминува inorder.

Имаме четири можни начини на мапирање на x,y и z во a,b и c. Реструктуирањето на трите јазли ги заменува z со јазол b, и негови деца ги прави a и c и прави децац на a и c да бидат претходните четири деца на x,y, и z, при тоа ги задржува inorder врските на сите јазли во T.

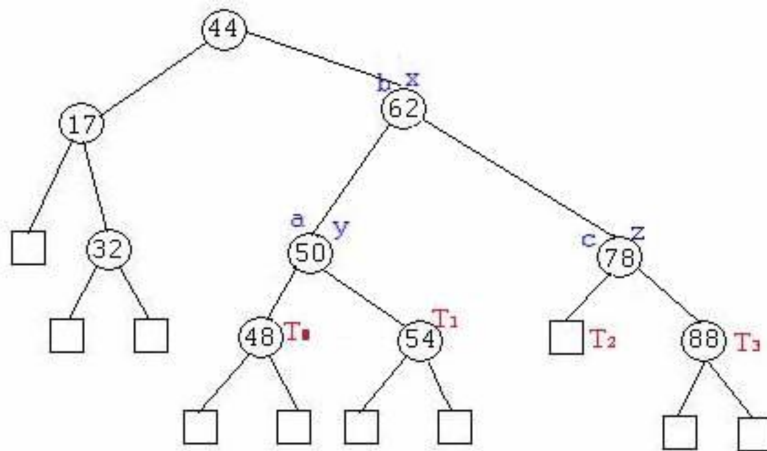
Алгоритам за реструктуирање .

Влез: јазол x од бинарно пребарувачко дрво T што има родител y и надродител z

1. Нека (a,b,c) се inorder од лево кон десно листани јазлите x,y,z и нека (T0,T1,T2,T3) се од лево кон десно поддрва на x,y,z
2. Заменете го поддрвото со корен во z со новото поддрво со корен во b
3. Нека a е левото дете на b и нека T0 и T1 се левото и десното поддрво на a
4. Нека c е десното дете на b и нека T2 иT3 се левото и десното поддрво на c



небалансирано бинарно пребарувачко дрво



Модификација на дрвото T предизвикана со операцијата за реструктуирање се нарекува ротација. Ако b=y тогаш методот на реструктуирање се нарекува еднократна ротација, тоа може да биде визуелизирано како ротирање на y околу z . Ако b=xоперацијата за реструктуирање се нарекува двојна ротација , и може да се визуелизира како ротирање на x околу y и потоа околу z.

Имплементација на ротација во Pascal:

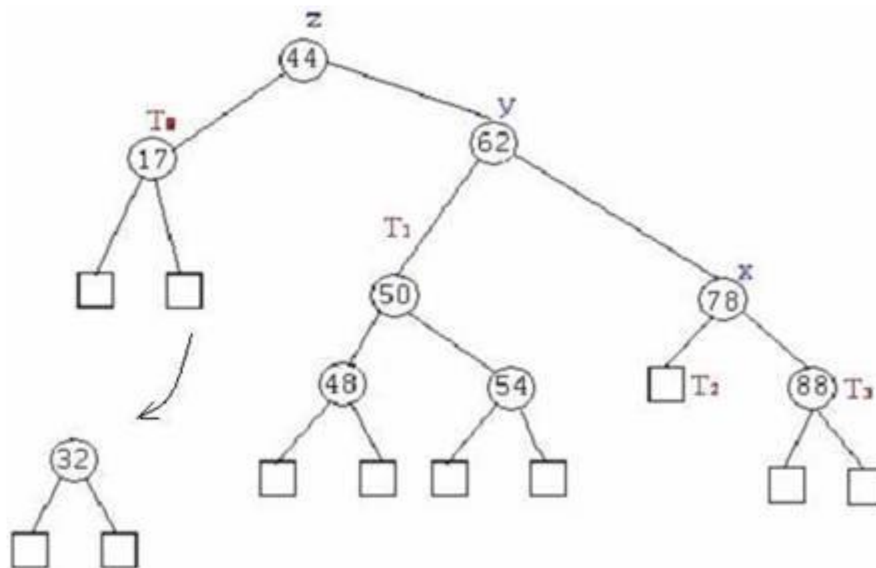
```
procedure lrot( var t : tree );
     var     temp : tree;
     begin
          temp := t;
          t := t^. right;
          temp^. right := t^. left;
          t^. left := temp;
          t^. weight := temp^. weight;
          temp^. weight := wt(temp^. left) + wt(temp^. right);
     end;

procedure rrot( var t : tree );
     var temp : tree;
     begin
          temp := t;
          t := t^. left;
          temp^. left := t^. right;
          t^. right := temp;
          t^. weight := temp^. weight;
          temp^. weight := wt(temp^. left) + wt(temp^. right);
     end;
```

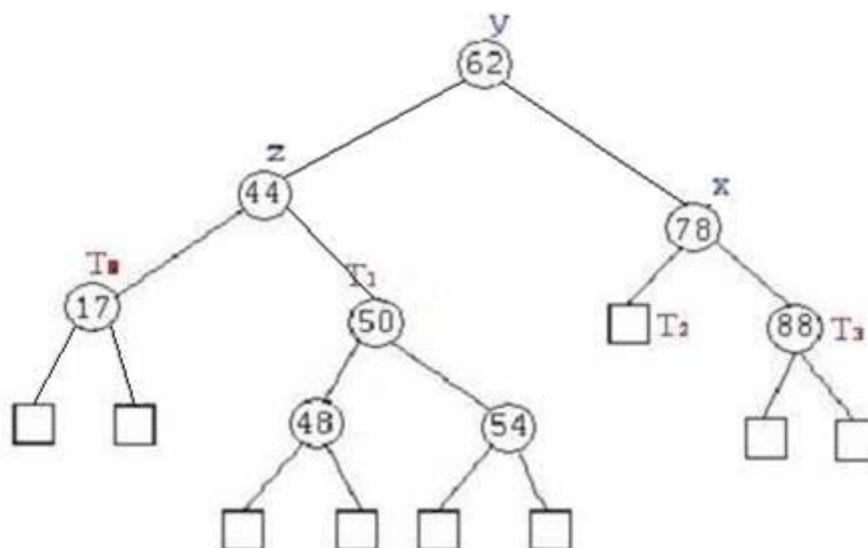
Имплементација на внесување во Pascal:

```
function insert( key : typekey; var t : tree ) : integer;
var     incr : integer;
begin
insert := 0;
if t = nil then begin
               t := NewNode( key, nil, nil );
               t^. bal := 0;
               insert := 1
               end
else if t^. k = key then
               Error {*** Клучот е во  табелата ***}
else    with t^ do     begin
        if k < key then  incr := insert( key, right )
               else     incr := -insert( key, left );
        bal := bal + incr;
        if (incr <> 0) and (bal <> 0) then
               if bal < -1 then
                       {*** левото поддрво е превисоко: потребна е десна ротација ***}
                       if left^. bal < 0 then rrot( t )
                               else    begin   lrot( left ); rrot( t ) end
               else    if bal > 1 then
                       {***  десното поддрво е превисоко: потребна е лева ротација ***}
                       if right^. bal > 0 then        lrot( t )
                               else    begin   rrot( right ); lrot( t ) end
               else    insert := 1;
        end
end;
```

Бришење на елементи на AVL дрво

При бришење на внатрешен јазол и подигање на едно од неговите деца на негово место, може да има небалансиран јазол во T на патот од родителот w на отргнатиот јазол до коренот на T.





Како и при внесувањето користиме реструктуирање за да се регулира балансот на дрвото T. Нека z е првиот небалансиран јазол вброен одејќи нагоре од w до коренот на T. Нека y е дете на z со поголема висина (јазолот y е дете на z кој не е предок на w), и нека x е дете на y со најголема висина. Изборот на x не мора да биде единствен, бидејќи поддрвата на y може да имаат иста висина. Во било кој случај кога ја изведуваме операцијата за реструктуирање, која ја регулира балансира носта на висината локално, поддрвото кое имало корен во z сега има корен во b.

Неформално, ваквото реструктуирање може да ја намали висината на поддрвото со корен во b за 1, што може да предизвика потомците на b да станат небаласирани. Значи еднократното реструктуирање не мора да ја промени балансираноста глобално по бришењето. Па, по ребалансирањето на z, продолжуваме да проверуваме каде во T има небалансирани јазли. Ако најдеме друг, ја изведуваме операцијата за реструктуирање за да се регулира балансираноста , и продолжуваме по T барајќи други се до коренот. Бидејќи висината на T е O(log n), каде n е бројот на членови, значи со ова реструктуирање сме го задоволиле својството за балансираност.

Имплементација на бришење во Pascal:

```
procedure delete( key : typekey; var t : tree );

    begin
    if t = nil then Error {*** клучот не е најден ***}
    else begin
        {*** бараме клуч кој ќе го избришеме ***}
        if      t^. k < key then delete( key, t^. right )
        else if t^. k > key then delete( key, t^. left )

        {*** клучот е најден, избриши го ако нема потомци ***}
        else if t^. left  = nil then t := t^. right
        else if t^. right = nil then t := t^. left

        else if wt( t^. left ) > wt( t^. right ) then
             begin rrot( t );  delete( key, t^. right ) end
        else begin lrot( t );  delete( key, t^. left ) end;

        if t <> nil then begin
            t^. weight := wt( t^. left ) + wt( t^. right );
            checkrots( t )
            end
        end
    end;
```

Multi-way пребарувачки дрва (силно разгранети)

Multi-way пребарувачки дрва се оние со внатрешни јазли кои имааат две или повеќе деца, членовите кои ги складираме во пребарувачкото дрво се парови од форма (к,х) каде к е клуч, а х е членот кој е поврзан со клучот.
Нека v е јазол на подредено дрво. Велиме дека v е d-јазол ако има d деца. Дефинираме Multi-way пребарувачко дрво да биде подредено дрво T кое ги има следниве својства.

1. секој внатрешен јазол од T има најмалку две деца. Значи, секој внатрешен јазол е (к,х) каде к е клуч, а х елемент, каде d ?2.
2. секој внатрешен јазол од T складира колекција од членови од форма (к,х) каде к е клуч, а х е елемент.
3. секој d-јазол v од T, со деца v_1…. . v_d складира d-1 членови ($к_1$,x_1). . . ($к_{d-1}$,x_{d-1}),каде $к_1$<…< $к_{d-1}$
4. конвенционално дефинираме $к_0$= -∞ и кd= +∞. За секој член (к,х) во јазол на поддрвото од v со корен во v_i i=1…d, имаме дека $к_{i-1} \leq к \leq к_i$.

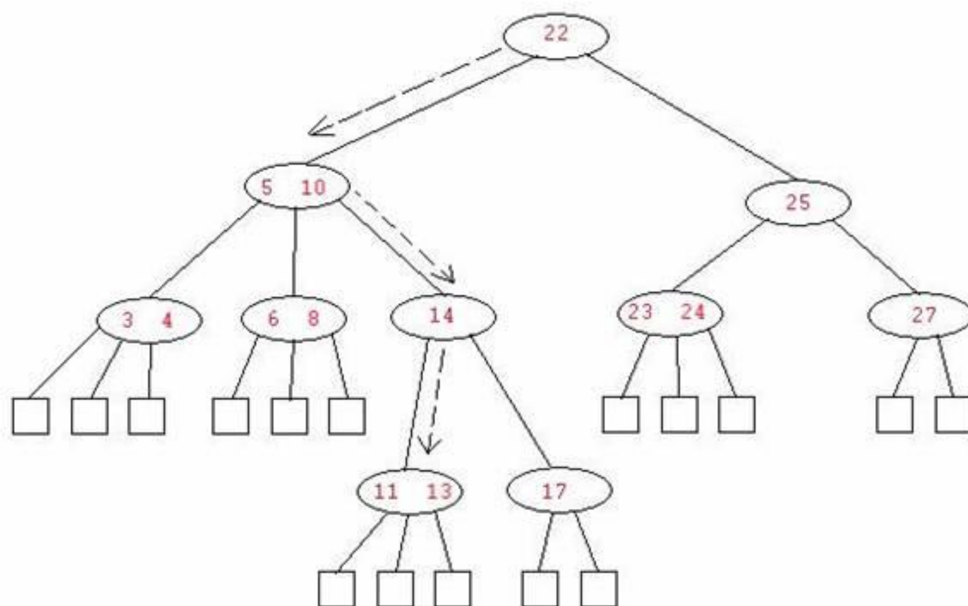
Ако во множеството од клучеви во v ги има и специјалните клучеви $к_0$= -∞ и кd= +∞, тогаш клуч к зачуван во поддрвото на T со корен во јазолот дете v_i мора да биде "измеѓу" два клуча кои ги има во v. Овој поглед укажува на правилото дека јазол со d деца има d-1 клучеви и исто така ги формира основите на алгоритамот за пребарување во multi-way пребарувачките дрва.

Од претходната дефиниција , надворешните јазли на Multi-way дрвата не зачувуваат членови и служат како " државачи на место" . Multi-way пребарувачкото дрво може да има само еден внатрешен јазол кој ги зачувува сите членови. Додека надворешните јазли можат да бидат null или референци до објектот Null-Node, но земаме дека тие се јазли кои не чуваат ништо.

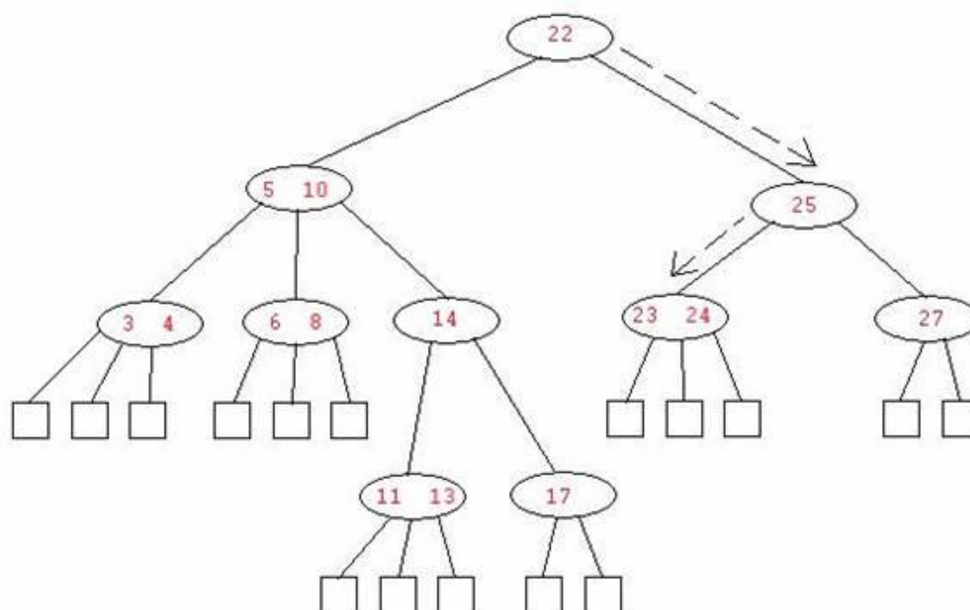
Без разлика дали внатрешните јазли имаат две деца или повеќе, има врска меѓу бројот на членови и бројот на надворешни јазли.
Својство: Multi-way пребарувачко дрво кое има n членови има n+1 надворешни јазли.

Пребарување во Multi-way дрво

Пребарување за елемент со клуч к во Multi-way пребарувачко дрво T, е едноставно. Изведуваме такво пребарување поминувајќи патека во T која започнува во коренот. Кога сме во d –јазолот го споредуваме клучот к со клучевите $к_1 < \ldots < к_{d-1}$ од v . Ако $к = к_i$ за некое i , потрагата е успешно завршена . Инаку, продолжуваме со потрагата во детето vi на v такво што $к_{i-1} \leq к \leq к_i$ (притоа $к_0 = -\infty$ и $кd = +\infty$). Ако дојдеме до надворешен јазол, тогаш знаеме дека нема член со клуч к во T и потрагата завршува неуспешно.



неуспешна потрага на 12



успешна потрага на 24

(2 , 4) ДРВА

Multi-way пребарувачко дрво кое ги исполнува следниве две својства се вика (2,4) или (2,3,4) дрво.

Својство за големина: Секој јазол има најмногу четири деца .

Својство за длабочина: Сите надворешни јазли имааат иста длабочина.

Земаме дека надворешните јазли се празни во опишувањето на методите за пребарување и измена ,земаме дака тие се вистински јазли.

Својство: висината на (2,4) дрво кое зачувува n членови е O(log n) .

Ова својство докажува дека својствата за големина и длабочина се важни за одржување на Multi-way дрвото балансирано и имплицира дека изведувањето на пребарување во (2,4) дрво завзема O(log n) време.

Внесување на елементи во (2,4) дрво

Методот за внесување го засегнува својството за длабочина, бидејќи додаваме нов надворешен јазол на исто ниво како и надворешните јазли, а тоа може да го засегне и својството за големина. Ако јазолот v е прво јазол со 4 члена , потоа може да стане 5-јазол , што предизвикува T веќе да не е (2,4) дрво. Нека v1… v5 се деца на v, и нека к1. . . к4 се клучевите зачувани во v . За да го разрешиме преполнувањето во јазолот v , изведуваме операција за поделба на v како што следи:

1. го заменуваме v со два јазли v' и v'', каде

1. 1 v' е 3-јазол со деца v1, v2, v3 со клучеви к1 и к2

1. 2 v'' е 2-јазол со деца v4, v5 со клуч к3

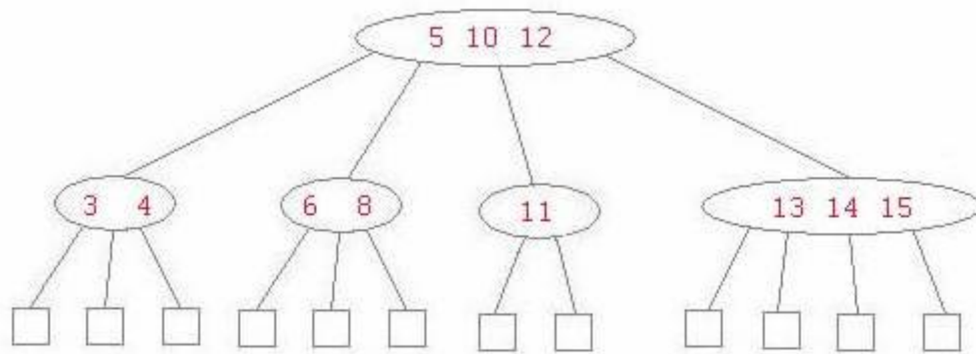
2. ако v е коренот на T, креираме нов корен , јазол u; инаку нека u е родител на v .

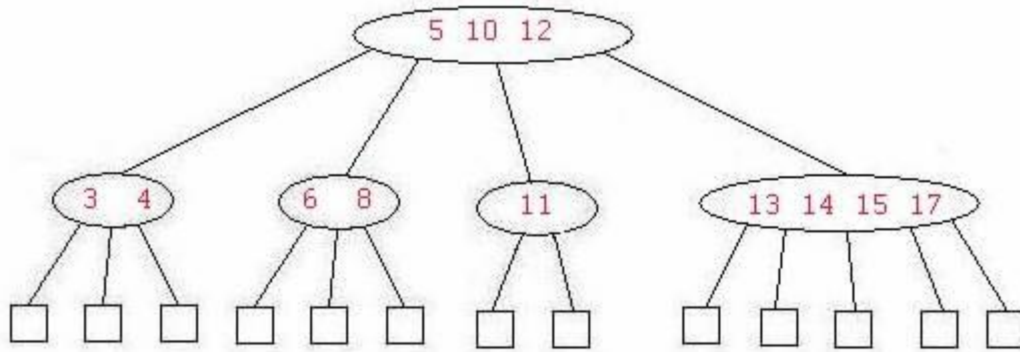
3. внесуваме клуч к3 во u и v' и v'' ги правиме деца на u ,така што ако v беше i-то дете на u, тогаш v' и v'' стануваат i-то иi+1 дете на u.

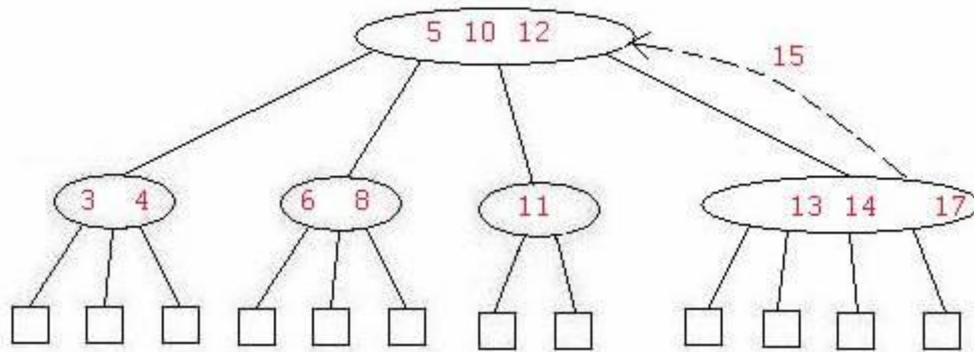
Анализа на внесувањата во (2,4) дрво

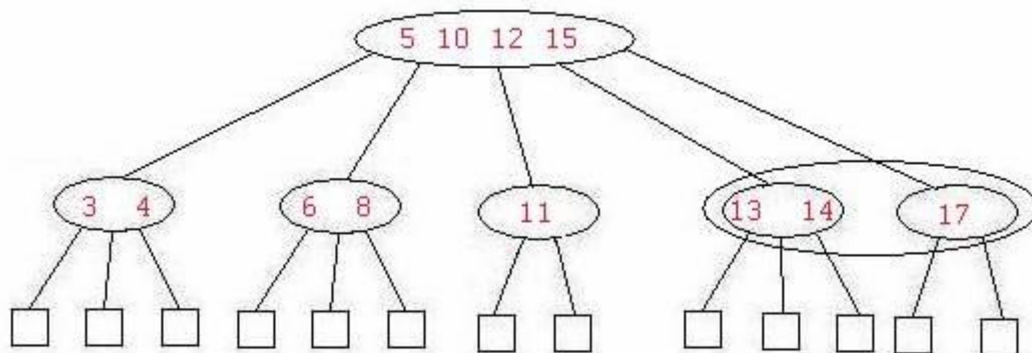
Операцијата на поделба влијае на константниот број на јазли на дрвото и O(1) членовите зачувани во таквите јазли. Значи, може да биде имплементирано за да се изврши во O(1) време.

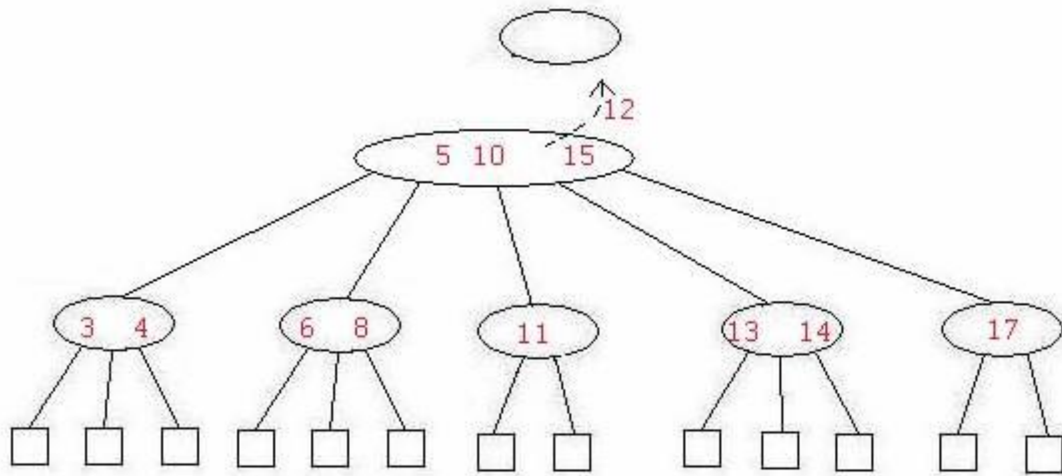
Како последица на операцијата на поделба на јазол v, ново преполнување може да се појави во родителот u на v . Ако се појави такво преполнување , тоа повлекува поделба на јазол u. Операцијата на поделба или го елиминира преполнувањето или го пролонгира во родителот на тој јазол. Бројот на операции на поделба е ограничен со висината на дрвото , кој е O(logn). Затоа целосно време на изведување на внесувањето во (2,4) дрво е O(log n).

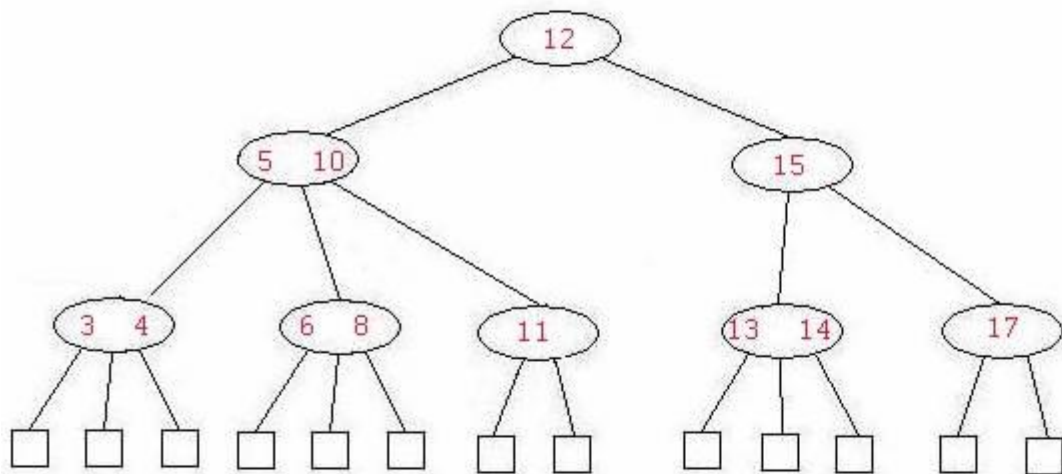

5 10 12
3 4
6 8
11
13 14 15


5 10 12
3 4
6 8
11
13 14 15 17


5 10 12
15
3 4
6 8
11
13 14 17


5 10 12 15
3 4
6 8
11
13 14
17





Бришење на елементи во (2,4) дрво

Почнуваме операција на бришење со изведување на потрага во Т за член со клуч к. Бришењето на таков член од (2,4) дрво може секогаш да се редуцира до случајот каде членот кој треба да се избрише е зачуван во јазол v чии деца се надворешни јазли. Да претпоставиме дека членот со клуч к кој сакаме да го избришеме е зачуван во i-тиот член (кi,хi) во јазолот z кој има единствен внатрешен јазол. Во овој случај, го заменуваме членот (кi,хi) со соодветен член кој е зачуван во јазолот v со надворешен јазол:

1. го наоѓаме најдесниот внатрешен јазол v во поддрвото со корен во i-тото дете на z ,забележуваме дека децата на јазолот vсе надворешни јазли

2. го заменуваме членот (кi,хi) на z со последниот член на v.

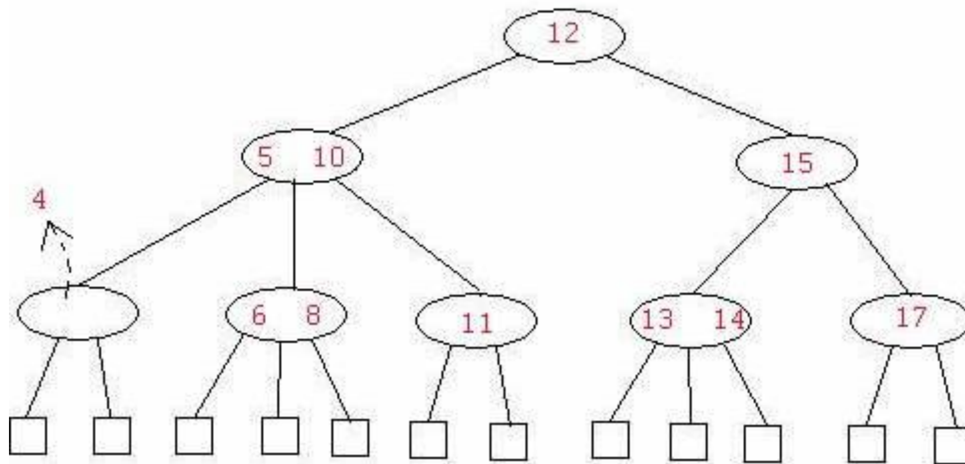
3. кога сме сигурни дека членот кој сакаме да го избришеме од v има деца кои се само надворешни јазли –деца, едноставно го бришеме членот од v и го бришеме i-тиот надворешен јазол од v.

Бришењето на член (и дете) од јазолот v го засега својството за длабочина, затоа секогаш го бришеме детето кое е надворешен јазол од јазолот v со децата надворешни јазли. Во отстранувањето на таков надворешен јазол можеме да влијаеме на својството за големина во v. Ако v бил претходно јазол со два клуча , потоа станува 1-јазол без членови за бришење, што не е дозволено во (2,4) дрво. Овој тип на влијаење на својството за големина се нарекува underflow во v. За да го поправиме ова , провеРуваме дали јазолот на исто ниво со v е 3-јазол или 4-јазол. Ако најдеме таков јазол w, тогаш

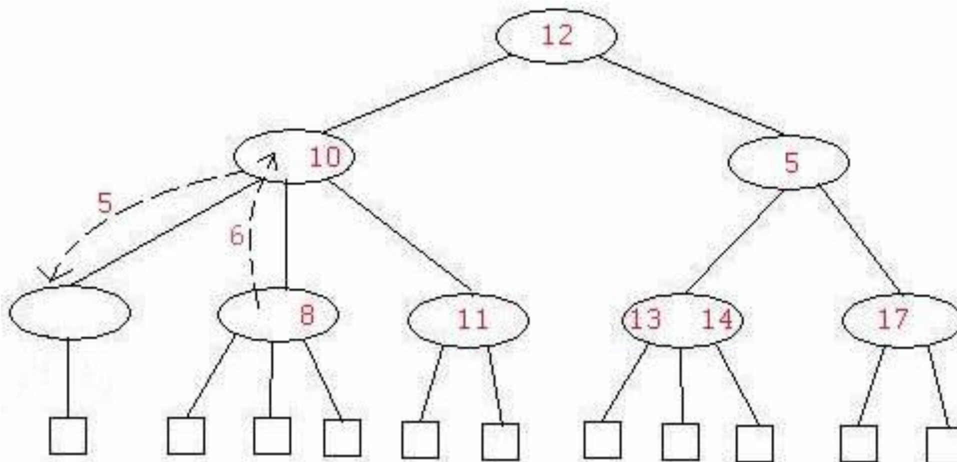
изведуваме операција на трансфер во која ги поместуваме децата од w во v, клучевите од w во родителот u на v и w, и клучот на u во v. Ако v нема „sibling“, или ако и двата се 2-јазли, тогаш изведуваме операција на фузија, во која ги спојуваме v со својот „sibling“, креирајќи нов јазол v' , и го поместуваме клучот од родителот u на v во v'.

Операцијата на фузија во јазолот v може да предизвика нов underflow во родителот u на v, што повлекува трансфер или фузија во u. Бројот на операции на фузија е ограничен со висината на дрвото, која е O(log n). Ако се пролонгира „underflow“ по целиот пат до коренот тогаш тој се брише.

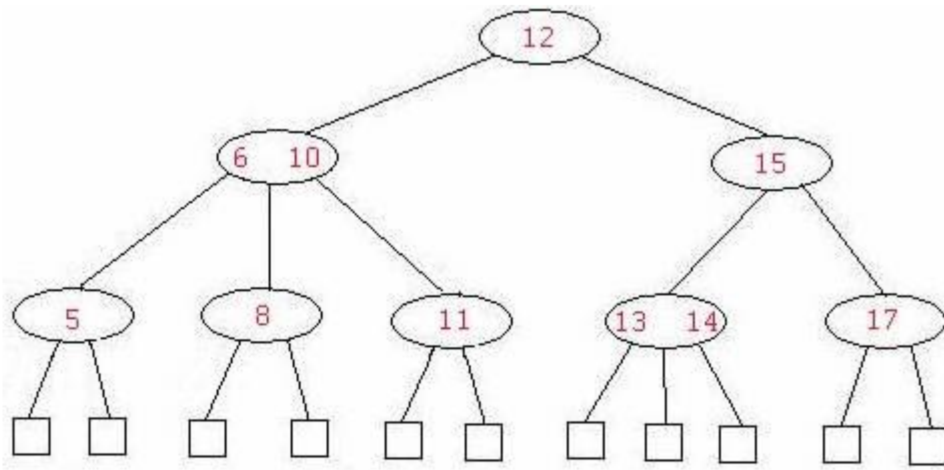
Примери за бришење:



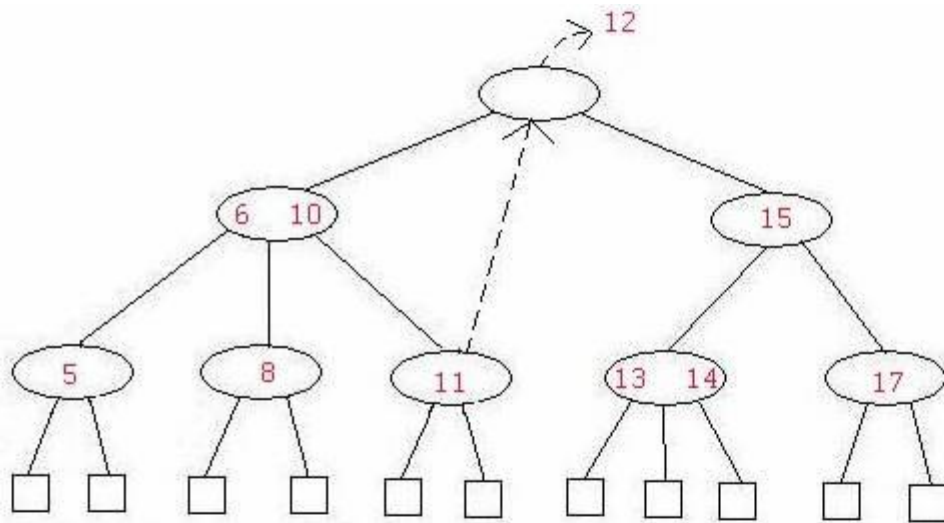
бришење на 4



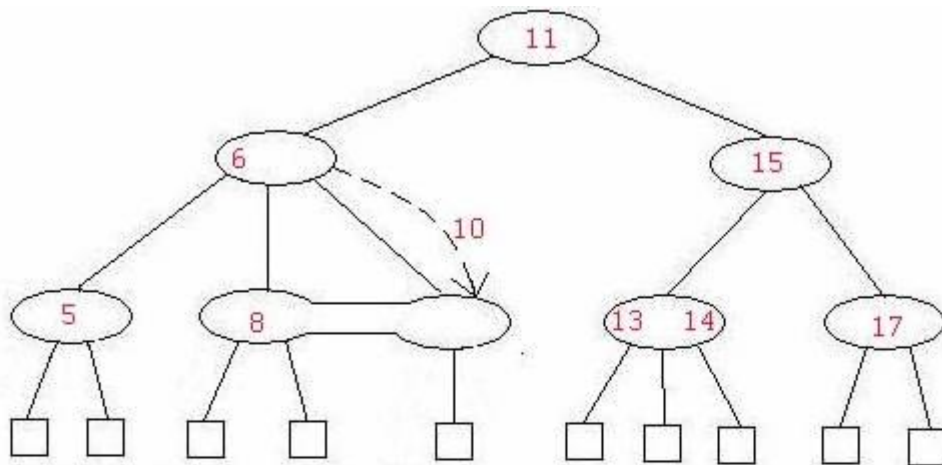
операција на трансфер



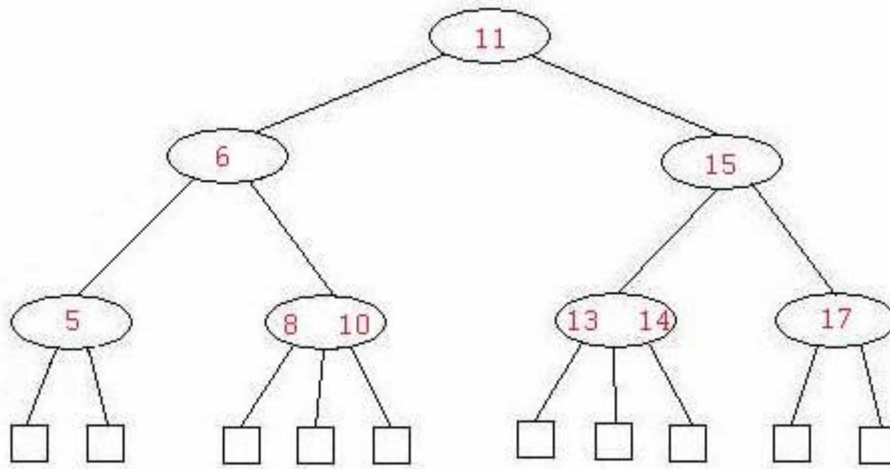
по трансфер операцијата



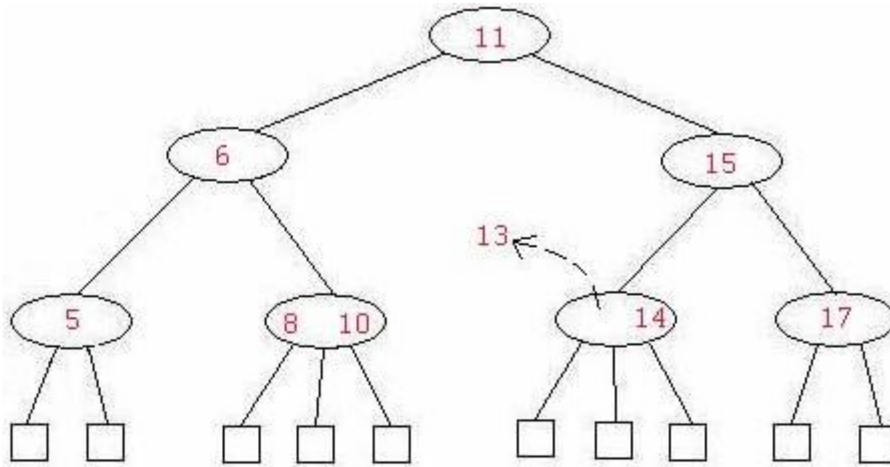
бришење на 12



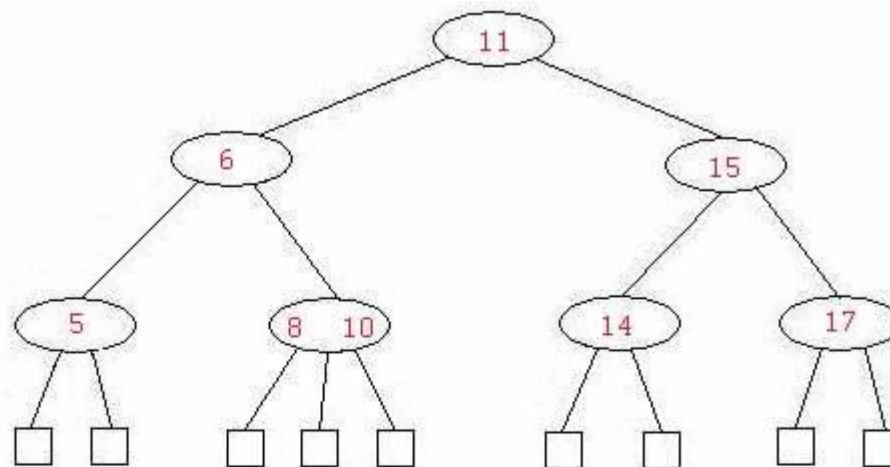
операција на фузија



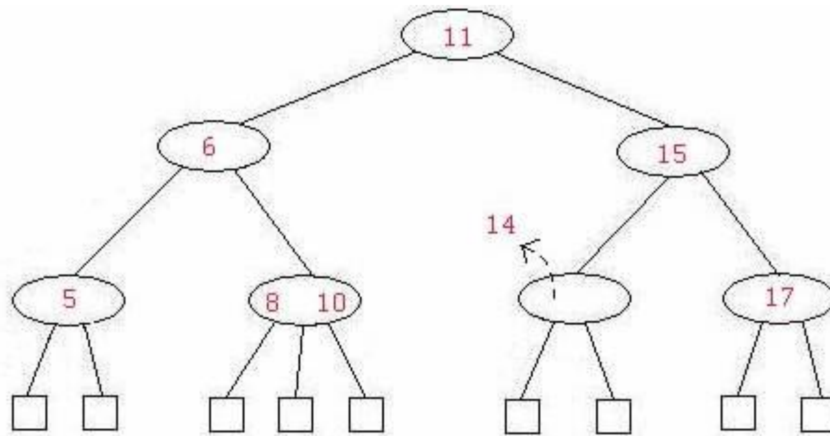
по фузија



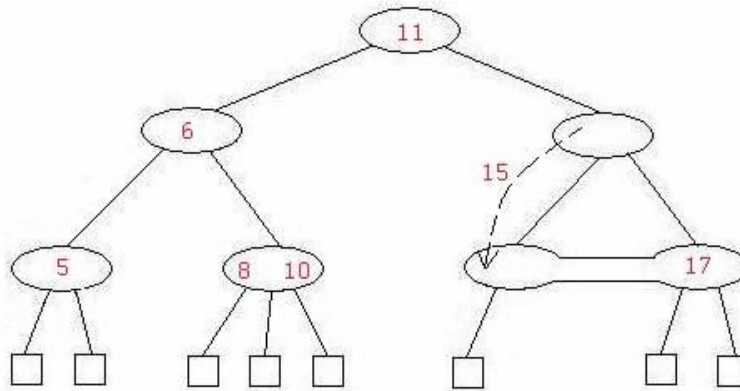
бришење на 13



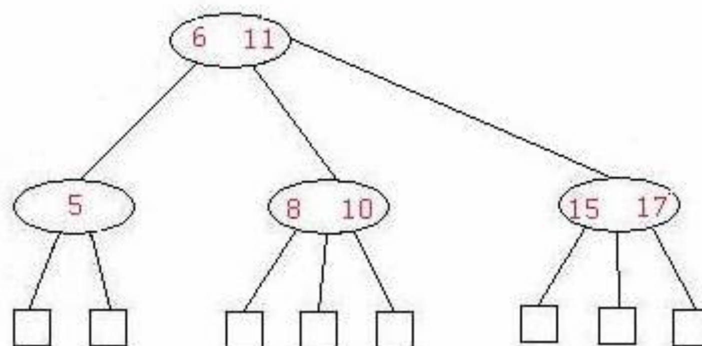
по бришењето



бришење на 14



фузија



Пребарување

Пребарување на податочни структури

Пребарување се нарекува процесот на барање на даден елемент од дадена структура на податоци. До бараниот елемент може да се пристапи на различни начини, кои се помалку или повеќе ефикасни. Понатаму ќе разгледаме пребарување на некои познати податочни структури како низи, листи, дрва итн.

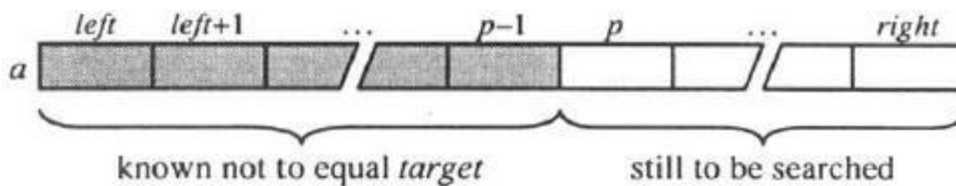
Низи

Ќе разгледаме пребарување во несортирани и сортирани низи.

Несортирани низи

Линеарно пребарување

Со линеарното пребарување бараниот елемент се споредува со секој елемент на низата. Кога ќе се пронајде бараниот елемент, алгоритмот завршува и го враќа индексот на тој елемент, а доколку нема таков елемент, алгоритмот враќа соодветна порака. Карактеристична операција на овој начин на пебарување е споредувањето. Овој алгоритам има линеарна временска сложеност, O(n). Просечниот број на споредби е (n+1)/2 а најлошиот случај е n споредби.




```
function found(a:niza;x,n:integer):boolean;
var i:integer;
begin
    found:=false;
    for i:=1 to n do
    begin
        if a[i]=x
        then
        begin
            found:=true;
            exit;
        end;
    end;
end;
```

target lion

		p = 0	1	2	3	4	5	6	7	8
Initially:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog
		0	*p* = 1	2	3	4	5	6	7	8
After 1 iteration:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog
		0	1	*p* = 2	3	4	5	6	7	8
After 2 iterations:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog
		0	1	2	*p* = 3	4	5	6	7	8
After 3 iterations:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog
		0	1	2	3	*p* = 4	5	6	7	8
After 4 iterations:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog
		0	1	2	3	4	*p* = 5	6	7	8
After 5 iterations:	*a*	rat	cat	pig	cow	fox	lion	tiger	goat	dog

Сортирани низи

Линеарно пребарување

Линеарното пребарување може да се примени и на сортирана низа. Придобивка овде е тоа што доколку бараниот елемент го нема во низата, не мора да бараме до крај на низата, туку до првиот елемент поголем од бараниот (освен ако бараниот елемент не е поголем од сите елементи од низта). Истотака ако бараниот елемент е помал од првиот тогаш него го нема во низата.

Бинарно пребарување

Поефикасно пребарување на сортираните низи е бинарното пребарување. Ние бинарно пребаруваме, на пример кога бараме збор во речник. На почеток го отвораме речникот на средина и ако ја погодиме страната застануваме. Но ако не ја погодиме страната, во зависност од тоа дали почетната буква на бараниот збор е пред или по буквата на која сме наиделе, повторно отвораме во првата или втората половина на речникот итн. се до наоѓањето на зборот.

Поточно постапката се одвива на следниот начин:

1. **Постави dolna=1 и gorna=n(каде n е бројот на елементи на низата)**
2. **Додека dolna<=gorn повторувај**

   2.1 **Постави sredina=(dolna + gorna) / 2**

   2.2 **Ако бараниот елемент е a[sredina] заврши со излез sredina**

   2.3 **Ако бараниот елемент е помал од a[sredina] тогаш gorna=sredina-1**

   2.4 **Ако бараниот елемент е поголем од a[sredina] тогаш dolna=sredina-1**

3**. Заврши со излез 0**

(a) *target* lion

		l = 0	1	2	3	4	5	6	7	r = 8
Initially:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	l = 5	6	7	r = 8
After 1 iteration:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	l = r = 5	6	7	8
After 2 iterations:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger

(b) *target* shark

		l = 0	1	2	3	4	5	6	7	r = 8
Initially:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	l = 5	6	7	r = 8
After 1 iteration:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	5	6	l = 7	r = 8
After 2 iterations:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	5	6	7	l = r = 8
After 3 iterations:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger
		0	1	2	3	4	5	6	r = 7	l = 8
After 4 iterations:	a	cat	cow	dog	fox	goat	lion	pig	rat	tiger

Бројот на итерации на овој алгоритам во најлош случај е еднаков на бројот на делење на должината на низата со два се додека должината не стане нула т.е Log_2n па и временската сложеност е $O(Log_2n)$.

```
    type pokazuvac=^element;
  element=record
        broj:integer;
        sleden:pokazuvac;
      end;
function najdi(baran:integer):boolean;
begin
  tekoven:=prv;
  while (tekoven^.broj<>baran) and (tekoven^.sleden<>nil)do
      tekoven:=tekoven^.sleden;
  end;
  if  tekoven^.broj=baran
      then najdi:=true
      else najdi:=false;
end;
```

Поврзани листи

Линеарно пребарување

Линеарното пребарување на поврзаните листи е идентично како кај низите и временската сложеност е исто O(n).

Бинарно пребарување

Бинарното пребарување е ефикасно кај низи зашто директно може да се пристапи до средниот елемент на низата. Кај поврзаните листи, за да се стигне до средниот елемент треба да се мине половина листа. Затоа временската сложеност на бинарното пребарување кај поврзаните листи е O(n), каде n е должината на листата. (НАПОМЕНА: Листата мора да биде сортирана за кристење на овој метод.)

Дрва

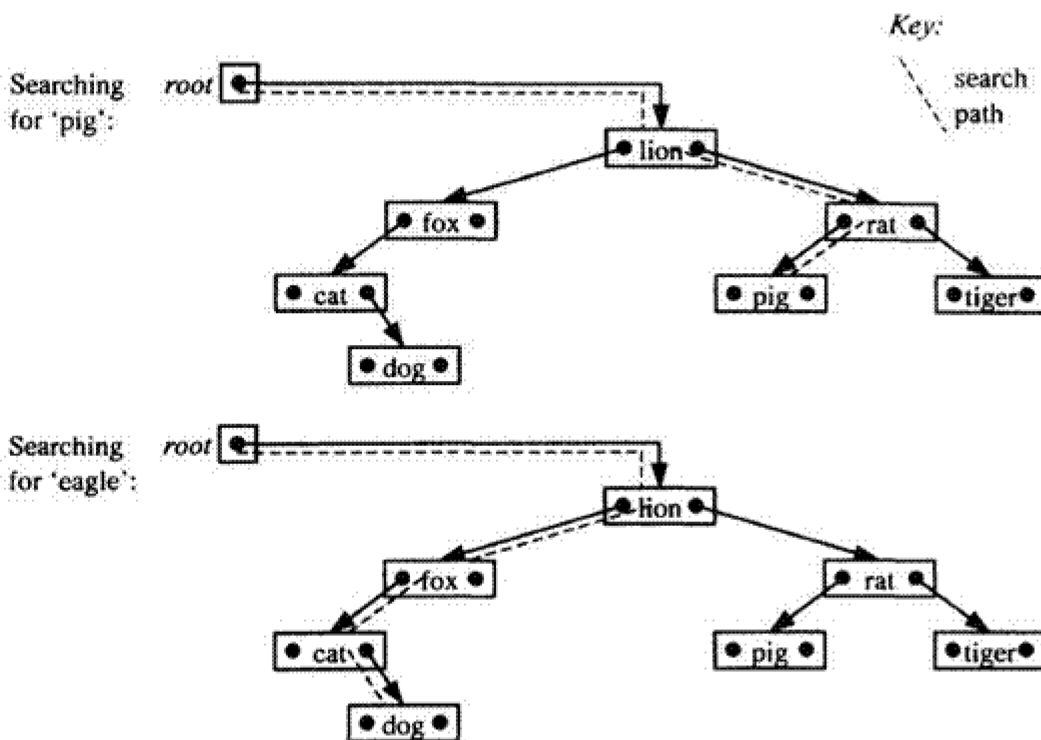
Бинарни пребарувачки дрва

Бинарно пребарувачко дрво е:

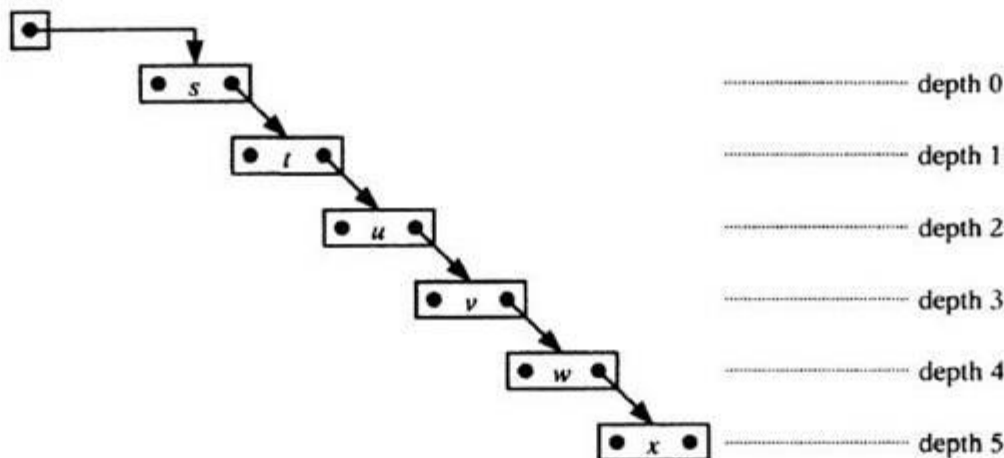
- празно дрво или
- непразно, во кој случај коренот содржи елемент, врска кон лево бинарно пребарувачко дрво со елементи помали од елементот на коренот и врска кон десно бинарно пребарувачко поддрво со елементи поголеми од елементот на коренот.

Совршено бинарно дрво со висина h е бинарно дрво со следните особини

- Ако h=0 тогаш левото и десното поддрво се празни
-Ако h>0 тогаш и левото и десното поддрво се совршени бинарни дрва со висина h-1



Временската сложеност во најлош случај е O(log n). Доколку дрвото не е совршено, временската сложеност ќе биде поголема. Дрвото во примерот се однесува како еднострана листа.



Алгоритам за пребарување на бинарно пребарарувачко дрво

1. Поставете го curr на вредноста на коренот
2. Повторувај:

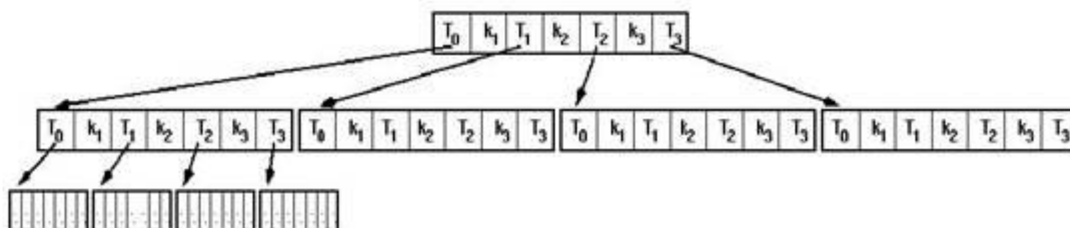
   2.1. Ако curr има вредност nil, заврши
   2.2. Инаку, ako target има иста вредност со curr, заврши со излез curr
   2.3. Инаку, ako target има помала вредност од curr, дајте му ја на curr вредноста на неговото лево дете
   2.4. Инаку, ako target има поголема вредност од curr, дајте му ја на curr вредноста на неговото десно дете

N-арни пребарувачки дрва

N-арно дрво е:

Празно дрво
Еден јазол има n-1 клучеви и n деца. Секој јазол има покажувачи на поддрва и клучеви: поддрво | клуч | поддрво | клуч | поддрво | клуч | поддрво

- Сите клучеви што се лево од некој клуч се помали од него
- Сите клучеви што се десно од некој клуч се поголеми од него



B-дрво е балансирано N-арно дрво кај кое:

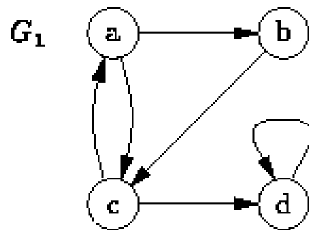
1. Сите листови се на исто ниво
2. Сите јазли освен коренот и листовите имаат најмалку n/2 а најмногу n деца. Коренот има најмалку 2 а најмногу n деца.

```
btree = ^node;
   node  = record
      d : 0..2*M;
      k : array [1..2*M] of typekey;
      p : array [0..2*M] of btree
      end;

procedure search( key : typekey; t : btree );

   var i : integer;
   begin
   if t=nil then {*** Not Found ***}
      notfound( key )
   else with t^ do begin
      i := 1;
      while ( i<d ) and ( key>k[i] ) do i := i+1;
      if key = k[i] then   {*** Found ***}
         found( t^, i )
      else if key < k[i] then search( key, p[i-1] )
         else           search( key, p[i] )
      end
   end;
```

Графови

Повеќе за графови – во предавањето за Графови.
Директен или насочен граф е подреден пар G=(V,E) со следните својства:

- V е конечно, непразно множество. Елементите на V се наречени темиња.
- Е е конечно множество од подредени парови на темиња. Елементите на ова множество се нарекуваат ребра.
- Реброто (a, b) е различно од реброто (b, a)



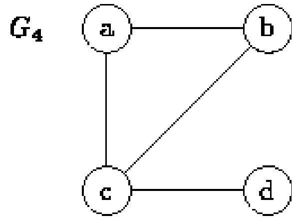
G1=(V1,E1)

V1={a, b, c, d}

E1={(a, b), (a, c), (b, c), (c, a), (c, d), (d, d)}

Индиректен или ненасочен граф со следните својства:

1. V е конечно, непразно множество. Елементите на V се наречени темиња.Е е конечно множество од подредени парови на темиња.
2. Елементите на ова множество се нарекуваат ребра.
3. Реброто (a, b) е исто со реброто (b, a)



G2=(V2,E2)

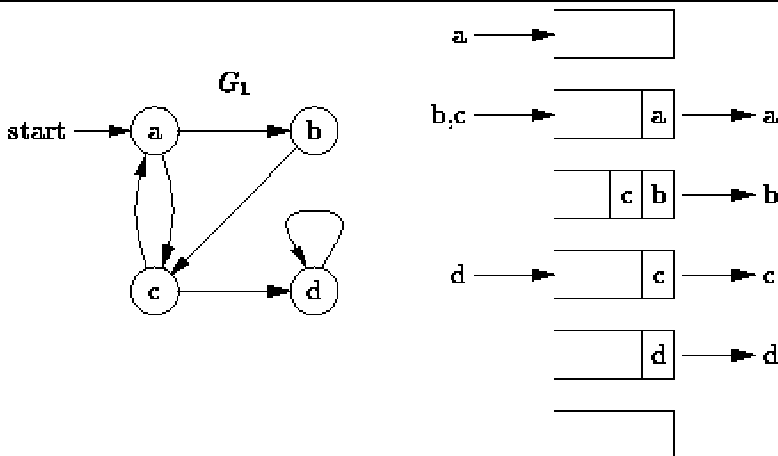
V2={a, b, c, d}

E2={(a, b), (a, c), (b, c), (c, d)}

Пребарување на графови по длабочина (Depth-First Search)

Овој метод може да се примени и на дрва, при што пребарувањето почнува од коренот. Кај графовите мора да го назначиме почетното теме. Со пребарувањето по длабочина се посетува теме а потоа рекурзивно се посетува секое теме поврзано со него. Попрецизно, откако ќе го одбереме почетното теме потоа се посетува теме поврзано со почетното и се продолжува со посетување на сите со него поврзани темиња.

Доколку почетното теме е поврзано со повеќе темиња, потоа се поситуваат и тие како и темињата поврзани со нив. Притоа графот може да содржи циклуси но секое теме мора барем еднаш да биде посетено. Затоа се памтат темињата што биле посетени. Во примерот подолу за почетно теме се зема C, потоа се посетува A, како теме поврзано со C па во длабочина B како теме поврзано со A. Потоа се навраќаме на темето D како второ теме поврзано со C.



```
DepthFirstSearch(G);
begin
   for each vertex x in V do num[x]:=0;
   TreeEdges:= 0;
   i:= 0;
   for each vertex x in V do
begin
            if num[x] = 0 then DepthFirstSearch(x);
    end;
   DepthFirstSearch(v);
   i:= i + 1;
   num[v]:= i;
   for each vertex w in Adj[v] do
   begin
            if num[w] = 0 then { w е ново теме}
            TreeEdges:= TreeEdges (v,w); {(v;w) е ново ребро}
            DepthFirstSearct(w);
   end;
end;
```

Покажувачи

Според операциите што можат да се извршат над одредено множество податоци, разликуваме повеќе типови на податоци. Под тип на податок се подразбира множество на податоци T={t1, t2,….,tn} врз кои можат да се применат кои било операции од множеството O={o1,o2,…om}.За секој податок што се користи во програмата мора да се дефинира неговиот тип.
Типовите податоци можат да бидат:

- статички
- динамички

Статички типови податоци се оние чија големина е однапред дефинирана (најчесто на почетокот на програмата). Тие се сместени на фиксни локации во меморијата.

Динамичките типови на податоци се оние чија големина и/или структура се менува во текот на извршувањето на програмата. Тие не се сместуваат на фиксна локација во меморијата, туку на локација која во моментот на нивното креирање е слободна.Динамичките типови на податоци можат да бидат:

- со променлива големина (низа битови, низа знаци, општа низа, множество, куп, ред и датотека)
- со променлива структура (листа, дрво, покажувач и објект).

При преведувањето на програмата потребно е преведувачот да ги знае сите променливи и нивниот тип. Променливите кои се создаваат за време на извршување на програмата, а потоа се бришат се нарекуваат **динамички променливи**.

Динамичките променливи кои добиваат податоци од тип покажувач се наречени **променливи од тип покажувач или само покажувачи**. При креирањето, динамичките променливи се сместуваат во слободната внатрешна (динамичка) меморија, наречена heap-меморија.

Користењето на променливи од тип покажувач нуди две битни погодности во однос на меморијата:

1. Се проширува меморискиот простор што може да се користи за податоци во една програма
2. Со користење на покажувачките променливи во динамичката меморија, програмата може да се извршува со помала количина неопходна меморија.

На пример, програмата може да поседува две многу сложени структури на податоци што не се користат истовремено. Ако овие две структури се декларираат како глобални променливи, тогаш тие се наоѓаат во сегментот за податоци и завземаат меморија за цело време на извршувањето на програмата, без оглед дали се користат во моментот или не. Но, ако се дефинирани преку покажувачи (динамички), тие се наоѓаат во динамичката меморија и ќе бидат избришани од неа по престанокот на нивното користење

Декларација на покажувачи

Покажувачките променливи не содржат податоци на ист начин како и другите променливи.

Овој тип променливи наместо податоци содржи локација т.е. адреса на податокот што се наоѓа во динамичката меморија.

Покажувачите се декларираат на следниот начин:

*тип *име_на_покажувач;*

каде што:

тип- е типот на променлива на кои ќе покажува покажувачот

*- знак за декларација на покажувач т.е покажувачка променлива

Пример: int *pok;

се декларира покажувачот (покажувачката променлива) pok, кој може да се користи само за да покажува на променливи од тип int.

Адресен оператор &

Операторот & се нарекува адресен оператор. Тој е унарен оператор, кој ја враќа адресата на операндот по него.

Пример:

```
int y=5;

int *yPok;
```

со наредбата

```
yPok=&y;
```

на покажувачот (покажувачката променлива) yPok му се доделува адресата на променливата y. Затоа за покажувачот yPok се вели дека покажува на y.

Оператор за дереференцирање *

Променливата yPok може да се означи и преку покажувачот yPok со *yPok.

Операторот * се нарекува **индиректен оператор** или **оператор за дереференцирање.** Тој ја враќа вредноста на променливата на која покажува неговиот операнд. Односно *yPok ја враќа вредноста на променливата на која покажува покажувачот yPok.

Пример:

```
cout<< *yPok;
```

ја печати вредноста на променливата y, т.е 5.

Пример:

```
int* ip;

int a=47;

int *ipa=&a;
```

На овој начин a и ipa се иницијализирани и покажувачот кон цел број ipa ја содржи адресата на a, а за да се пристапи кон променливата a преку покажувачот, покажувачот се дереференцира на следниот начин:

**ipa=110;*

сега а содржи вредност 110 наместо 47.

Иницијализација на покажувачи

Покажувачите треба да се иницијализираат или при декларација или во некоја наредба за доделување. Покажувачот исто така може да биде иницијализиран на 0, NULL. Покажувачот со вреднот NULL не покажува на ниту една променлива. NULL покажувач не е исто што и неиницијализиран покажувач!

*Пример за користење ба операторите & и **

```
#include
using namespace std;
void main()
{
int a;
int *aPok;
a=7;
aPok=&a;
cout<<" Adresata na a e" << &a << "\n Vrednosta na aPok e " << aPok;
cout<<"Vrednosta na a e "<< a << "\n Vrednosta na *aPok e "<< *aPok;
cout<<"Pokazuvame deka * i & se komplementni &*aPok="<< &*aPok << " *&aPok=" << *&aPok << endl;
}
```

Покажувачи и низи

Постои тесна врска меѓу низите и покажувачите.Доколку е декларирана и иницијализирана низата:

int a[]={1,2,4,8,16};

Нејзиниот идентификатор a има вредност на адресата на првиот елемент од низата и е од *тип покажувач на променливи од типот на елементите на низата.* Во овој случај а е покажувач на целобројни променливи затоа:

a;

&a[0];

ја соджат адресата на првиот елемент од низата.Исто така покажувачите често се користат за пристап до елементите на низа.Тоа е можно бидеќи елементите на низите во меморијата се сместуваат во последователни адреси.

Пример:

int a[10], *p;

со наредбата:

p=a;

на покажувачот p му се доделува вредноста на адресата на првиот елемент на низата a[]. А додека вредноста на првиот елемт од низата може да се добие со *a или a[0].

Наредбите new и delete

Наредбата new има форма

покажувач= new тип;

Со неа во меморијата се креира променлива од соодветен тип и на покажувач му се доделува адресата на креираната променлива. Во спротивно покажувачот добива вредност NULL.

Пример, ако е деклариран покажувач

```
int *ip;
```

тогаш со наредбата

```
ip=new int;
```

во меморијата се креира неименувана променлива и нејзината адреса му се доделува на покажувачот ip.

Наредбата delete се користи за бришење (ослободување) на мемориската локација.

Пример, со:

```
delete ip;
```

се ослободува меморијата на променливата на која покажувал покажувачот ip, при што тој останува недефиниран.

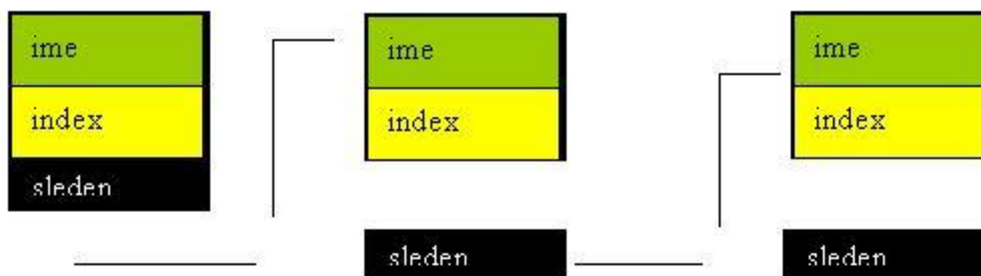
Внимателно со покажувачите!

„Играњето“ со меморијата може да биде ризично и да причинува некои програмски грешки, меѓутоа истовремено и некои програмски задачи не можат (или е потешко возможно) да се реализираат без употреба на покажувачите!

Листи

Динамичката меморија се користи за сместување на големи и сложени типови на податоци, па покажувачките променливи служат за нивна манипулација. Тие можат да покажуваат кон локација од било кој тип на податок вклучувајќи ги и записите.

Запис е множество на податоци. Елементите на записот не се индексирани, туку секој податок има свој идентификатор (име) со кое се пристапува до него. На пример, записот за еден студент нека ги содржи податоците: име, број на индекс и следен. ime е текстуален податок, додека index е целоброен податок. sleden е покажувач од тип pok, кој е дефиниран како покажувач од тип student. Според дефиницијата на student, можеме да креираме листа од записи од ист тип, поврзани со употреба на покажувачи. Листата е илустрирана со следнава слика:



Поврзаната листа е една од основните динамички структури на податоци кои се употребуваат во програмирањето. Се состои од низа од јазли. Секој јазол има дел за сместување податоци и дел за сместување покажувач кој служи како врска кон наредниот јазол од низата. Поврзаните листи овозможуваат вметнување и отстранување на јазли од било кое место во низата за константно време, но не дозволува пристап до јазлите по случаен избор. Односно, секвенцијалниот пристап е ефикасен, меѓутоа директниот не е бидејќи треба да се поминат сите елементи од листата за да се добијат потребните податоци.
Постојат неколку типови на поврзани листи: еднострано поврзани, двострано поврзани и кружни листи.

Низи наспроти поврзани листи

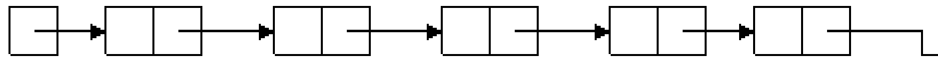
Низите алоцираат меморија за сите нејзини елементи заедно како еден блок. Наспроти тоа листите алоцираат меморија за секој елемент посебно во одделен блок меморија. Тоа значи дека поврзани листи најдобро работат за складирање на низи од податоци на кои не им се знае однапред колкав ќе биде нивниот број, со можност за подоцна да се менува бројот на елементите и нивниот редослед.

Пристапот до елементите кај низите е директен, додека кај листите секвенцијален. Едностраните листи можат да се поминуваат само во еден правец. Ова ги прави поврзаните листи да се неупотребливи каде што е пожелно да се пребаруваат брзо елементите по нивниот индекс. Потребен е дополнителен мемориски простор за референците кај поврзаните листи, што ги прави непрактични за листи со едноставни податоци како char или Boolean.

	Низи	Листи
Пребарување	O(1)	O(n)
Вметнување	O(n)	O(1)
Бришење	O(n)	O(1)

Еднострано поврзани листи

Наједноставниот тип на поврзани листи се еднostaно поврзаните листи кои имаат по еден покажувач за секој јазол, кој покажува кон наредниот јазол од листата, кон null вредност ако станува збор за последниот јазол од листата или кон празна листа. Анкер (anchor) на листата е покажувач кој покажува кон првиот јазол. За да се промени редоследот на листата доволно е само да се промени вредноста на покажувачот.



Секој јазол од листата има свој *следбеник* (освен последниот) и свој *предходник* (освен првиот). *Должината* на поврзана листа се изразува преку бројот на јазли од кои е составена. Листа која не содржи јазли се нарекува *празна листа*.

Креирањето на поврзани листи во Pascal бара одреден напор, но придобивката од користењето на поврзаните листи е зголемена брзина на работа и поефикасно користење на меморијата.

Да разгледаме практичен пример на креирање еднострано поврзана листа и нејзина манипулација. Нека јазлите содржат записи како тип на податоци, дефинирани на начин што предходно го објаснивме со записот `student`.

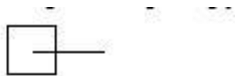
```
Type
     pok=^student;
     student: record
                    ime: string;
                    index: integer;
                   sleden: pok;
    end;
```

Ќе започнеме со **процедура за додавање елементи од назад,** која подоцна ќе ја искористиме во програмата и за креирањето на листата. Кодот во Pascal е:

```
Procedure DodadiNazad (Var anker : PLista; nov : PLista);
begin
  if anker = nil then
    anker := nov
  else
    begin
      pom := anker;
      while pom^.sleden <> nil do
        begin
          pom := pom^.sleden;
        end;
      pom^.sleden := nov;
    end;
end;
```

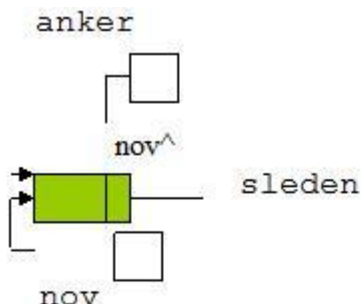
Разгледуваме случај на додавање елементи од назад на: *празна листа* и додавање елементи на *листа со повеќе елементи*.

Дали листата е празна проверуваме со *if anker = nil then*

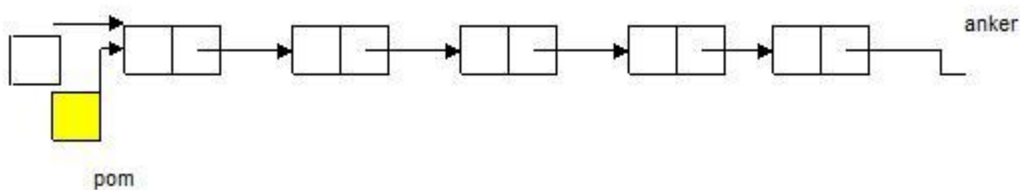


Anker

т.е. ако листата е ***празна,*** *тогаш* *anker := nov*



Ако не е празна воведуваме нова покажувачка променлива `pom` и ја поставуваме да покажува онаму каде што покажува `anker`, односно на почетокот на листата.

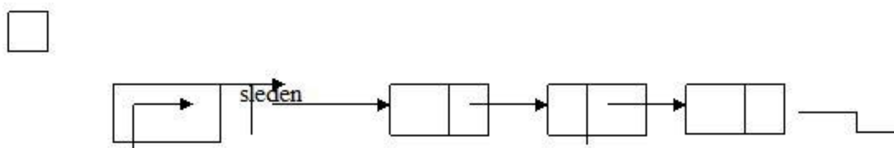


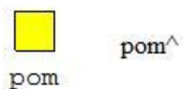
И започнуваме со “шетање” низ листата за да стигнеме до нејзиниот крај, каде ќе го

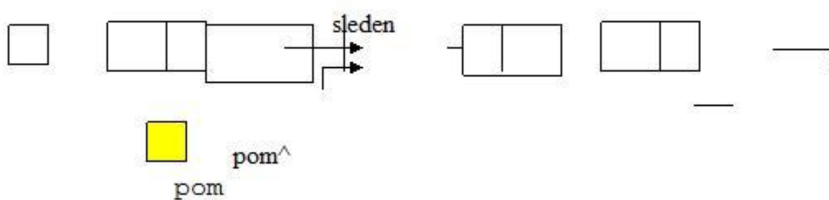
додадеме новиот елемент. Тоа се изведува со програмските линии

while pom^.sleden nil do

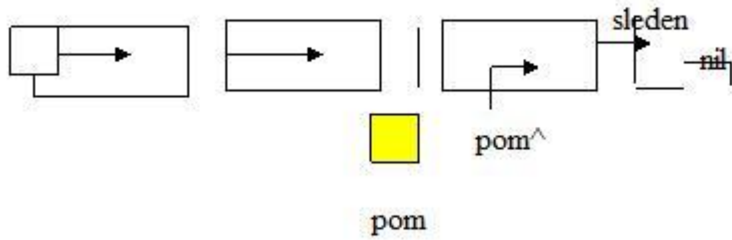
pom := pom^.sleden;



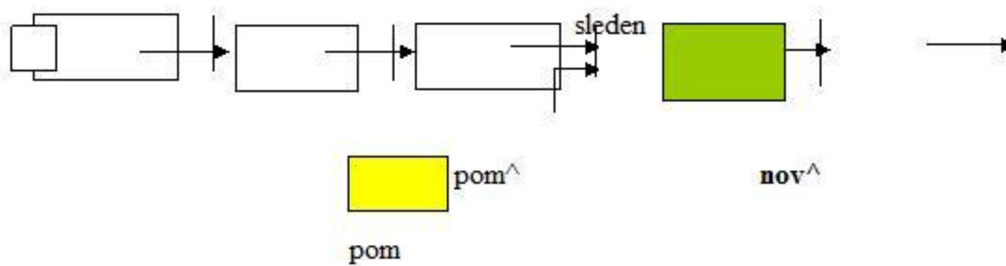




Кога `pom^.sleden=nil` *односно помошниот покажувач ќе покажува на последниот елемент од листата (чиј* `sleden` *покажувач има nil вредност),*



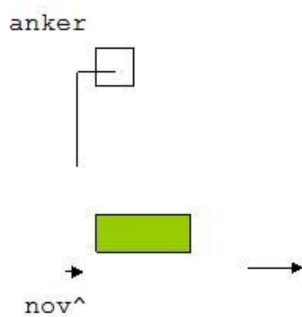
Тогаш со pom^.sleden:=nov ; се поставува новиот елемент на крајот од поврзаната листа.



Процедура за додавање на елементи на почетокот на листата:

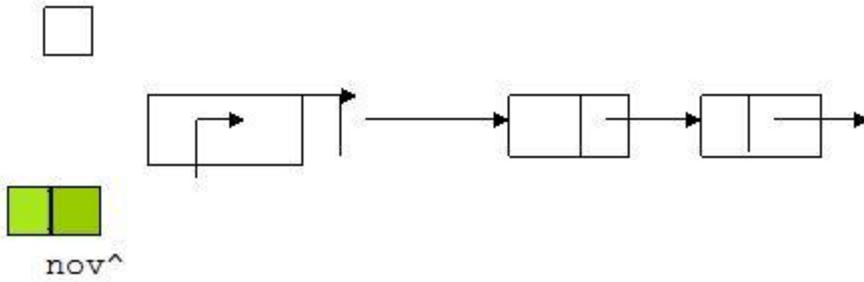
```
Procedure DodadiNapred (Var anker : PLista; nov : PLista);

begin
  if anker = nil then
    anker := nov
  else
    begin
      nov^.sleden := anker;
      anker := nov;
    end;
end;
```

Ако листата е празна, односно главата на листата има вредност nil, ***anker:=nov*** ;

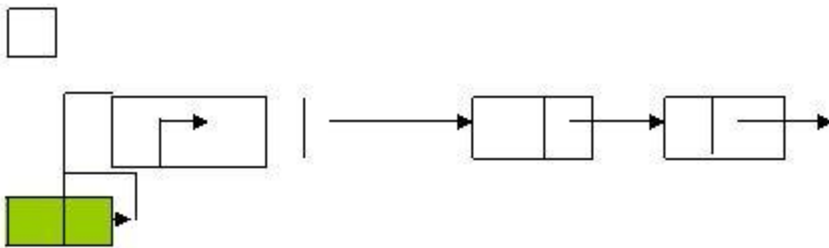


Ако листата има елементи:

nov^.sleden:=anker



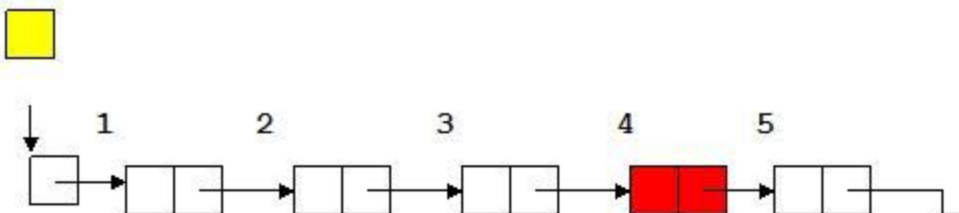
anker:=nov;



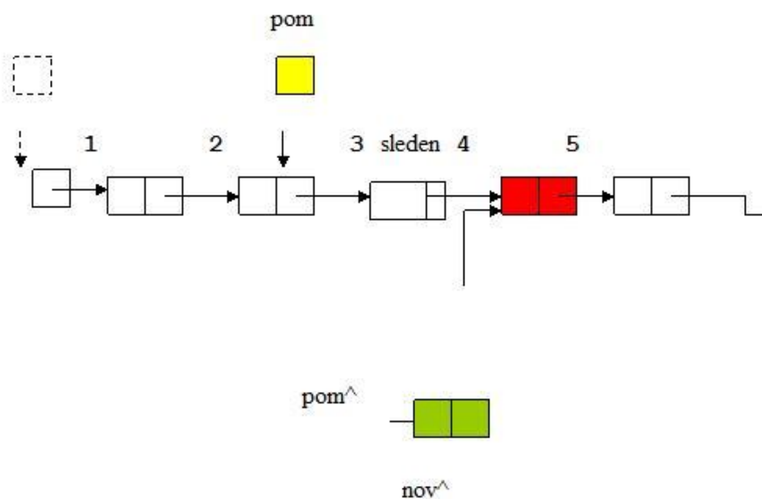
Процедура за додавање на елемент на n-та позиција

```
Procedure DodadiSredina (Var anker: PLista; nov: PLista);
begin

 Writeln('Vnesi go brojot na elementot pred koj sakas da go vmetnes noviot element: ');
 Readln(m);

    pom:=anker;
    for i:=1 to (m-2) do
        pom:=pom^.sleden;

    nov^.sleden:=pom^.sleden;
    pom^.sleden:=nov;
end;
```

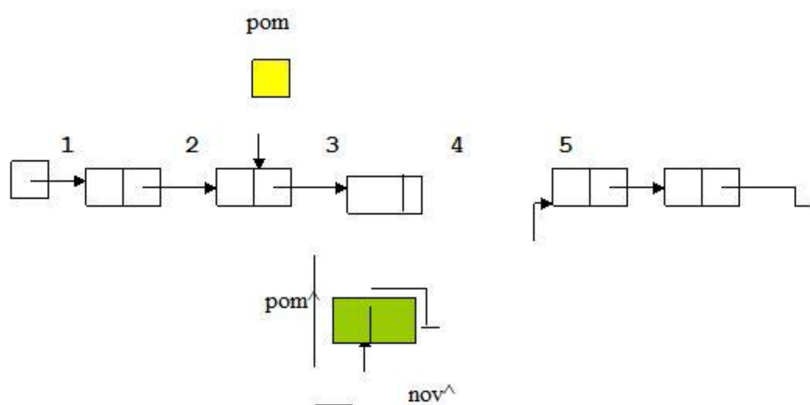
Ако сакаме да го додадеме нов елемент пред m-тиот елемент од поврзаната листа, помошниот покажувач `pom` го поставуваме на почетокот, и почнуваме да „шетаме“ за (m-2) чекори по нејзината должина



Ако сакаме да вметнеме нов пред 4-тиот елемент, го поместуваме помошниот покажувач два пати по должината на листата и застануваме кај неговиот предходник, т.е. на 3-тиот елемент.



Новиот елемент го поставуваме да покажува на исто место со `pom^.sleden` со програмската линија `nov^.sleden:=pom^.sleden`. И потоа правиме “преврзување“ на елементите од листата со тоа што `pom^.sleden` го ставаме да покажува на исто место со `nov (pom^.sleden:= nov;)`.

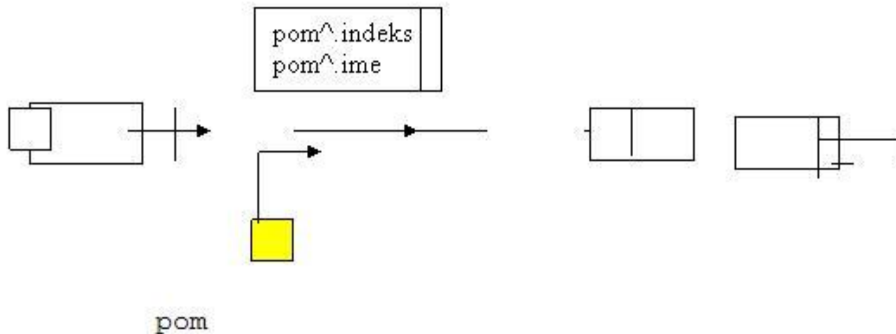


Процедура за листање на елементите од поврзаната листа:

```
procedure BrisiRedenbroj (Var anker : PLista);

begin
 readln(m);
 if m=1 then
   begin
     pom := anker;
     anker := pom^.sleden;
     dispose (pom);
   end
  else
   begin
     pom := anker;
     for i:=1 to (m-2) do
       pom := pom^.sleden;
     pom2 := pom^.sleden;
     pom^.sleden := pom2^.sleden;
     dispose (pom2);
   end;
end;
```

Го поставуваме помошниот покажувач да покажува на почетокот од листата, односно на исто место каде што покажува anker со pom:=anker;

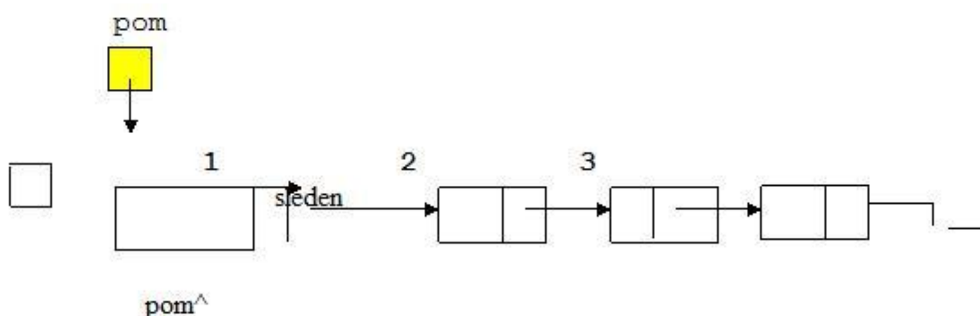
Шетаме низ листата на ист начин објаснет во процедурата DodadiNazad. На секој елемент до кој дошол помошниот покажувач, му пристапуваме до податоците кои ги содржи со pom^.indeks, pom^.ime (во нашиов случај јазлите од листата содржат име и презиме од записот student) и ги печатиме истите.



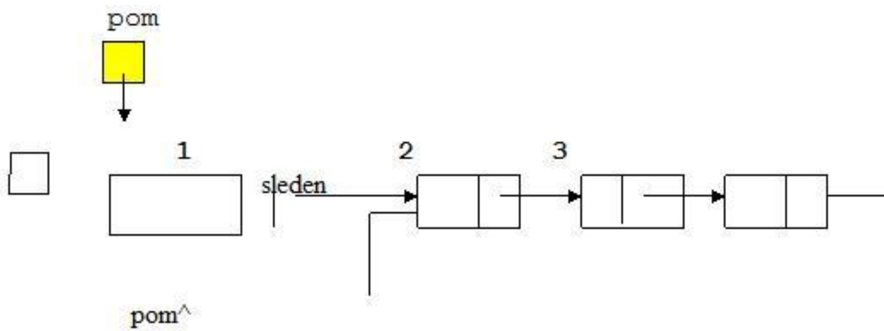
Процедура за бришење на m-ти елемент од поврзана листа:

```
procedure BrisiRedenbroj (Var anker : PLista);

begin
 readln(m);
 if m=1 then
   begin
     pom := anker;
     anker := pom^.sleden;
     dispose (pom);
   end
  else
   begin
     pom := anker;
     for i:=1 to (m-2) do
       pom := pom^.sleden;
     pom2 := pom^.sleden;
     pom^.sleden := pom2^.sleden;
     dispose (pom2);
   end;
end;
```

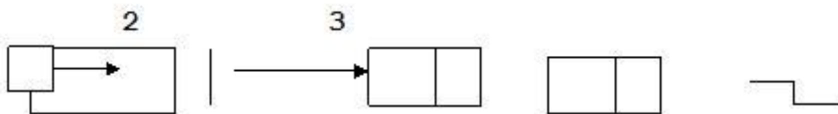
Ако сакаме да го избришеме првиот елемент од листата (m=1), поставуваме помошен покажувач да покажува на почетокот на листата `pom := anker;`



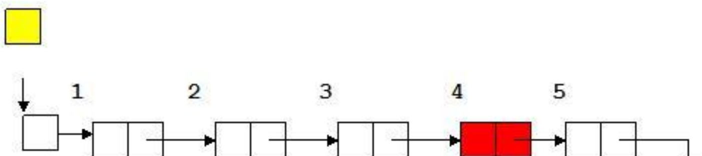
Анкерот се поставува да покажува на исто место со pom^.sleden

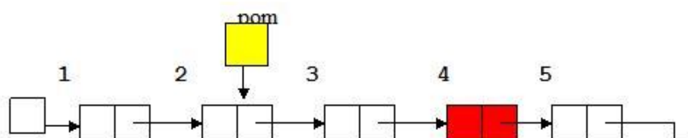


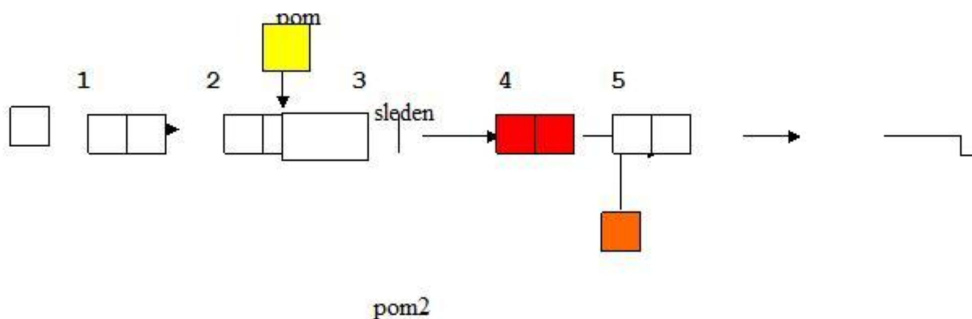
И со `dispose(pom)` се остранува првиот елемент со што се ослободува мемориската локација која ја зафаќал тој.



Ако сакаме да го избришеме **m-тиот елемент** од поврзаната листа, помошниот покажувач `pom` го поставуваме на почетокот, и почнуваме да „шетаме“ за (m-2) чекори по нејзината должина. На пример, ако сакаме да го избришеме 4-тиот елемент, од почетокот на листата правиме 2 поместувања на `pom` по должината на листата и застануваме со покажувачот да покажува на еден елемент пред елементот кандидат за бришење. И воведуваме нов помошен покажувач `pom2` кој ќе ни покажува на исто место `со pom^.sleden (pom2:=pom^.sleden).`

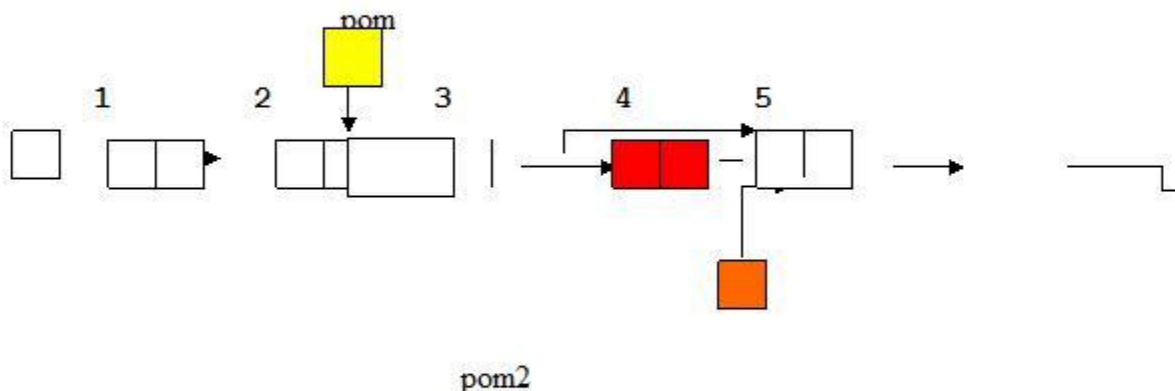






`На овој начин сме се осигурале дека и со отстранувањето на врската која води од 3-тиот елемент кон 4-тиот, сеуште ќе имаме пристап до сите елементи од листата. Сега извршуваме „преврзување“ со тоа што ќе го`

поставиме pom^.sleden да покажува на исто место со pom2^.sleden. Односно врската од 3-тиот елемент која покажуваше кон 4-тиот, сега покажува онаму каде што покажува 4-тиот елемент, а тоа е на 5-тиот.

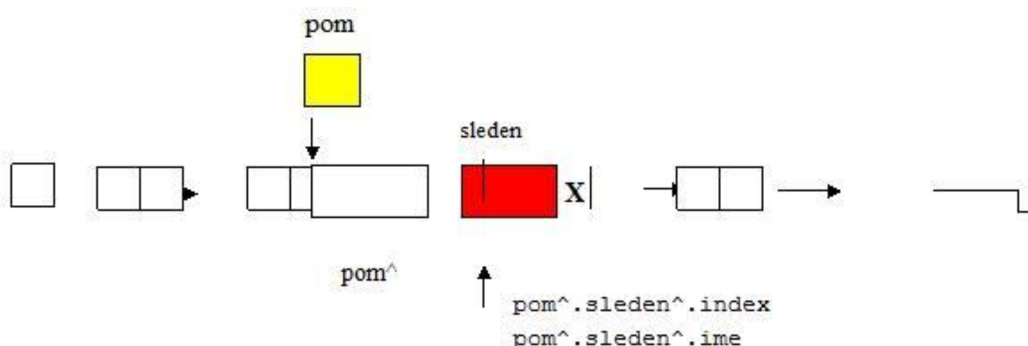


Процедурата за бришење на елемент од листа пребарувајќи го според податокот што го содржи тој елемент е слична со предходно објаснетата процедура:

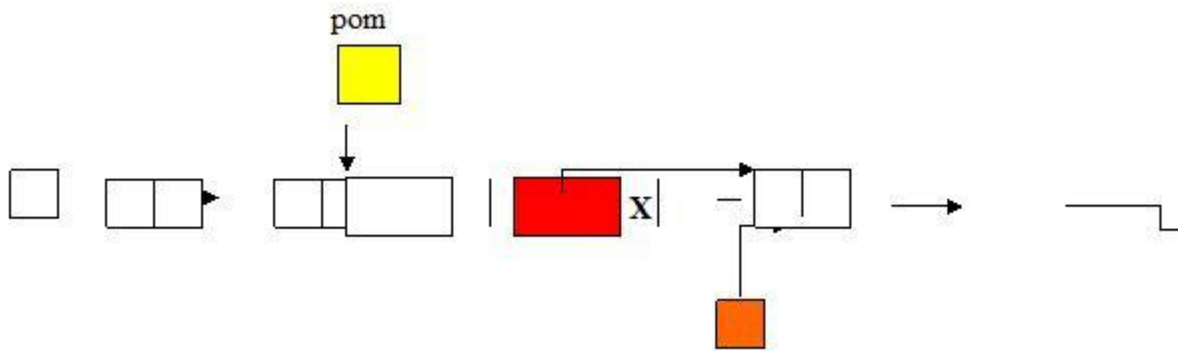
```
Procedure BrisiPodatok (Var anker : PLista; x : integer);
begin

 readln(x);
 writeln;
  if anker^.indeks = x then
    begin
      pom := anker;
      anker := pom^.sleden;
      dispose (pom);
    end
  else
    begin
      pom := anker;
      while pom^.sleden^.indeks <> x do
        pom := pom^.sleden;
      pom2 := pom^.sleden;
      pom^.sleden := pom2^.sleden;
      dispose (pom2);
    end;
end;
```

Помошен покажувач се поставува на почетокот на листата и се шета по листата се додека

pom^.sleden^.indeks x



Кога pom^.sleden^.index ќе биде бараниот податок воведуваме нов помошен покажувач pom2 и постапката е иста како и во предходниот случај.



Комплетна програма за манипулација со еднострана поврзана листа:

```
Program PovrzanaLista;
Type
PLista = ^TLista;
TLista = Record
indeks : integer;
ime : String;
sleden : PLista;
end;
Var
anker, nov, pom, pom2 : PLista;
n, i,m, indeks, x, izbor : integer;
ime : String;

Procedure KreirajLista (Var anker:PLista; nov: PLista);
Begin
anker:=nil;
writeln('Kolu elementi ke vnesuvas vo listata:');
readln(n);
writeln;
for i:= 1 to n do
begin
new(nov);
writeln ('Vnesi go brojot na index na ',i, '-tiot element, pa potoa negovoto ime:');
readln(indeks);
readln(ime);
writeln;
nov^.indeks:=indeks;
nov^.ime:=ime;
nov^.sleden:=nil;
if anker = nil then
anker := nov
else
begin
pom := anker;
while pom^.sleden nil do
begin
pom := pom^.sleden;
end;
pom^.sleden := nov;
```

```
end;
end;
end;
Procedure KreirajNov(Var nov: PLista);
begin
new(nov);
writeln('Vnesi go indexot na noviot element:');
readln(indeks);
writeln('Vnesi go imeto na noviot element:');
readln(ime);
writeln;
nov^.indeks:=indeks;
nov^.ime:=ime;
nov^.sleden:=nil;
end;

Procedure DodadiSredina (Var anker: PLista; nov: PLista);
begin
Writeln('Vnesi go brojot na elementot pred koj sakas da go vmetnes noviot element: ');
Readln(m);

pom:=anker;
for i:=1 to (m-2) do
pom:=pom^.sleden;
nov^.sleden:=pom^.sleden;
pom^.sleden:=nov;
end;


Procedure Listaj;
begin
writeln;
writeln('Elementite na listata se:');
writeln;
pom := anker;
while pom nil do
begin
writeln (pom^.indeks, ' ', pom^.ime);
pom := pom^.sleden;
end;
writeln;
end;


Procedure DodadiNapred (Var anker : PLista; nov : PLista);

begin
if anker = nil then
anker := nov
else
begin
nov^.sleden := anker;
anker := nov;
end;
end;
```

```
Procedure DodadiNazad (Var anker : PLista; nov : PLista);

begin
if anker = nil then
anker := nov
else
begin
pom := anker;
while pom^.sleden nil do
begin
pom := pom^.sleden;
end;
pom^.sleden := nov;
end;
end;

procedure BrisiPodatok (Var anker : PLista; x : integer);

begin
writeln('Vnesi go brojot na index na elemenotot sto ke go brises:');
readln(x);
writeln;
if anker^.indeks = x then
begin
pom := anker;
anker := pom^.sleden;
dispose (pom);
end
else
begin
pom := anker;
while pom^.sleden^.indeks x do
pom := pom^.sleden;
pom2 := pom^.sleden;
pom^.sleden := pom2^.sleden;
dispose (pom2);
end;
end;




procedure BrisiRedenbroj (Var anker : PLista);

begin
writeln('Na koe mesto se naogja elementot sto ke go brises:');
readln(m);
Writeln('Izbravte da go brisete ',m,'-tiot element od listata');
writeln;
if m=1 then
begin
```

```
pom := anker;
anker := pom^.sleden;
dispose (pom);
end
else
begin
pom := anker;
for i:=1 to (m-2) do
pom := pom^.sleden;
pom2 := pom^.sleden;
pom^.sleden := pom2^.sleden;
dispose (pom2);
end;
end;



begin
KreirajLista(anker, nov);
Listaj;
writeln;
Readln;
n:=0;
while n<1 do
begin
writeln;
writeln('OBERI OPCIJA OD MENITO:');
writeln;
writeln('1. Kreiraj nov');
writeln('2. Vmetni na pocetok');
writeln('3. Vmetni na kraj');
writeln('4. Vmetni vo sredina');
writeln('5. Listaj');
writeln('6. Brisi element spored sodrzinata');
writeln('7. Brisi element spored mestoto');
writeln('8. KRAJ');

readln(izbor);
writeln;
case izbor of
1: KreirajNov(nov);
2: DodadiNapred(anker, nov);
3: DodadiNazad(anker, nov);
4: DodadiSredina(anker,nov);
5: Listaj;
6: BrisiPodatok(anker, x);
7: BrisiRedenbroj(anker);
8: n:=2;
end;
end;
writeln;
end.
```

Подалгоритми

Програмирањето е процес на подготовка на некој проблем (задача) за решавање со помош на компјутер. Тој процес се состои од: поставување на проблемот, дефинирање на постапката (алгоритамот) и пишување програма.

При поставување на проблемот потребно е истиот да се разбере, анализира и, по можност, да се разбие на потпроблеми. Потоа секој потпроблем може да се разгледува како посебна целина, за чие решавање може да се применуваат различни алгоритми и да се пишуваат посебни програми. Ваквите програми се викаат потпрограми. Со логичко поврзување на сите потпрограми се формира програма за решавање на поставениот проблем.

На пример, проблемот "Пресметување просечен успех на едно училиште" може да се разбие на следните потпроблеми: пресметување просечен успех на еден ученик, на едно одделение, на една година, на сите години.

Ако за секој од овие потпроблеми напишеме посебни програми - потпрограми и сите нив ги поврземе во една програма, така добиената програма ќе биде многу појасна, попрегледна и поразбирлива.

За секој подалгоритам може да се напише посебна програма која се нарекува потпрограма (анг. subprogram). Програмата напишана за некој алгоритам се нарекува главна програма (анг. main program). Во главната програма може да се повикуваат повеќе потпрограми.

Потпрограмите претставуваат независни програмски целини, кои имаат свои влезни податоци и даваат излезни резултати. Секоја потпрограма може да биде напишана од друг човек. Корисникот на потпрограмата треба да знае само да ја користи: да знае кои и какви влезни податоци се дозволени и кои резултати се добиваат.

Потпрограмите кои се користат за решавање на стандардни проблеми, се напишани најчесто од група програмери и усовршени се чуваат во посебни програмски библиотеки. Потпрограмите од библиотеките можат да се користат од секоја корисничка програма со повикување преку името на потпрограмата. Претходно, програмата треба да биде поврзана со програмската библиотека.

Со постоењето на програмските библиотеки се ослободуваат програмерите од пишување на исти програми за проблеми со кои често се среќаваат. На пример, во секоја програмска библиотека постојат потпрограми за: наоѓање корени на квадратна равенка, собирање на два полиноми, решавање систем линерни равенки итн.

Потпрограмите што најчесто се користат при програмирањето, наречени стандардни потпрограми, се ставени во посебни програмски модули кои се составен дел на преведувачот на соодветниот програмски јазик.

Постојат два вида потпрограми:

- функциски потпрограми или функции (анг. function), и
- процедурални потпрограми или процедури (анг. procedure)

Подалгоритми

Подзадачите се користат за добивање еден или повеќе меѓурезултати, од кои во понатамошните чекори, се добиваат резултатите на задачата. За добивање на меѓурезултатите не мора да се користат сите влезни податоци, туку само некои од нив или некои претходно добиени меѓурезултати.

За секоја подзадача може да се напише посебен алгоритам, кој ќе го наречеме подалгоритам. Подалгоритмите, исто како и алгоритмите, се именуваат. Влезните податоци во подзадачата и меѓурезултатите што се добиваат, се нарекуваат аргументи.

Влезните податоци се нарекуваат влезни аргументи, а меѓурезултатите се нарекуваат излезни аргументи. Аргументите се наведуваат во заграда по името на подалгоритмот. На пример, во подалгоритмот *Поголем*, влезните аргументи можеме да ги запишеме со broj1 и broj2, а излезниот аргумент со pogolem. Тогаш насловот на подалгоритмот Поголем може да се запише со *Поголем*(broj1, broj2, pogolem; На крајот од насловот на подалгоритмите се става знакот ; (точка и запирка).

Алгоритмите за посложени задачи можат да бидат многу долги и непрегледни. Затоа и програмите што ќе се напишат според тие алгоритми можат да бидат тешко разбирливи. Тоа е посебно важно кога ќе се јави потреба од некоја промена или надополнување во нив. За полесно снаоѓање во големите програми, денес тие се пишуваат со техника на програмирање позната како структурно програмирање.

При структурното програмирање се користат две техники на програмирање, и тоа:

- програмирање одгоре надолу (анг. top-down programming) и
- модуларно програмирање (анг. modular programming).

Програмирањето одгоре надолу се врши со разделување (расчленување) на задачата на помали и поедноствни задачи, кои ќе ги наречеме подзадачи. Ако е потребно, и тие подзадачи понатаму се разделуваат на уште поедноставни, додека не се добијат задачи што лесно се програмираат.

Секоја подзадача, од така расчленетата задача, може да се разгледува како посебна целина, независно од другите.

На пример, во алгоритамот на задачата за наоѓање на најголемиот од три дадени броја, двапати се бара одредување на поголемиот од два броја.

```
алгоритам Najgolem;
              почеток
                     читај a,b,c;
                     ако a>b
                            тогаш
                                   pogolem <- a
                            инаку
                                   pogolem <- b;
                     крај_ако(a>b)
                     p <- pogolem;
                     ако p>c
                            тогаш
                                   pogolem <- p
                            инаку
                                   pogolem <- c;
                     крај_ако(p>c)
                     n <- pogolem;
                     печати n;
              крај(Najgolem)
```

Тоа може да се издвои како посебен алгоритам, односно подалгоритам, со кој може да се извршат двете споредувања: *a* и *b* во првото, и *p* и *c* во второто споредување на кои било два броја.

Постојат два вида подалгоритми, и тоа:

- функциски и
- процедурални.

Функциски подалгоритми

Функциските подалгоритми, наречени и функции (анг. function), го добиле името бидејќи имаат само еден излезен параметар, исто како и функциите. Излезниот параметар се нарекува *излезен формален параметар*, додека влезните параметри, кои можат да бидат и повеќе, се нарекуваат *влезни формални параметри* .

Излезниот формален параметар од функциските подалгоритми е **самото име на функцијата**. Влезните формални параметри се ставаат во заграда по името.

За да одлучиме дали за една подзадача може да се напише функциски подалгоритам, потребно е да се одговори на следното прашање:

0. Дали подалгоритмот за подзадачата има една излезна вредност?

Ако одговорот на ова прашање е да, тогаш за подзачата може да се напише функциски подалгоритам. При пишување на функциски подалгоритам за зададена подзадача, потребно е да се одговорат и прашањата:

1. Од кои аргументи директно зависи резултатот на подзадачата?

2. Од кој тип е излезниот резултат на подзадачата?

Одговорот на првото прашање ќе ни укаже на тоа: кои, колку и каков тип влезни формални аргументи ќе има во листата на формални аргументи; додека, пак, одговорот на второто прашање ќе ни го одреди типот на излезниот резултат од функцискиот подалгоритам.

На пример, подалгоритамот за наоѓање на поголемиот од два броја може да се запише како функциски подалгоритам на следниов начин:

```
подалгоритам Pogolem (broj1, broj2);
      почеток
            ако   broj1>broj2
                 тогаш
                      Pogolem <- broj1
                 инаку
                      Pogolem <- broj2;
            kraj_ako  (broj1>broj2)
крај (Pogolem)
```

Еве и еден пример на задача каде решението не може да се добие со функциски подалгоритам: Да се подредат три броја по големина.

Одговорот на основното прашање (ред. бр. 0) е НЕ, затоа што во задачата се бара влезните податоци (трите броја) да бидат и излезни резултати, пак три броја, само подредени по големина. Затоа решението не можеме да го добиеме со функциски подалгоритам.

За функцијата да врати резултат, името на функцискиот подалгоритам мора да се јави на левата страна барем во една наредба за доделување во самиот подалгоритам, бидејќи преку името се пренесува резултатот од подалгоритамот во главниот алгоритам.

Функциските подалгоритми се повикуваат со чекор за доделување. На пример со чекорот:

```
p <- Pogolem(a,b);
```

се повикува подалгоритамот за наоѓање на поголемиот од броевите *a* и *b*. Притоа, влезните формални параметри *broj1* и *broj2* се заменуваат со вистинските *a* и *b* соодветно. Преку името на

подалгоритамот *Pogolem*, со чекорот за доделување *p ß Pogolem(a,b);*, во променлива *p* се пренесува резултатот од споредувањето на *a* и *b*.

Алгоритамот за наоѓање на најголемиот од три дадени броја со користење на функцискиот подалгоритам *Pogolem*, ќе биде

```
алгоритам Najgolem;
почеток
     читај a,b,c;
     p <- Pogolem(a,b);
     n <- Pogolem(p,c);
     печати n;
крај(Najgolem)
```

Во програмите кои користат потпрограми променливите можат да се поделат на: глобални променливи и локални променливи. Променливите кои се декларираат во главната програма се нарекуваат глобални променливи, а променливите кои се декларираат во потпрограмите се нарекуваат локални променливи. Самото нивно име ни зборува и за областа во која со нив може да се оперира, т.е. кога тие се достапни (видливи).

Променливите што се најавуваат на почетокот на програмата се нарекуваат *глобални променливи*. За да се избегне зависност на потпрограмите од главната програма, Паскал дозволува декларирање константи, променливи, типови, па и други функции во декларативниот дел на функцијата и тие имаат значење само во функцијата, т.е. делуваат **локално**.

Пример:

```
Function Stepen (x : Real; N : Integer) : Real;
Var
       Pom : Real;
       i : 1..MaxInt;
Begin
       Pom:=1.0;
       For i:=1 to abs(n) do
            Pom:=Pom*x;
            If n >= 0 then Stepen:=Pom
            Else Stepen:=1.0/Pom;
End;
```

***Pom , i* се локални променливи**.

Накратко, вреднување на функциски повик се одвива на следниот начин:

1. Се креира мемориска локација за функцискиот повик;

2. Вистинските параметри на функцискиот повик формираат парови со формалните параметри на функцијата, од лево на десно. Нивниот број мора да е еднаков;

3. Се креира мемориска локација за секој формален параметар на функцијата, кон кои е придружен соодветниот вистински параметар. Типовите на двата вида параметри мора да се исти;

4. Се креира мемориска локација за секоја локална променлива на функцијата;

5. Се изведуваат наредбите во телото на функцијата;

6. На крај, сите мемориски локации креирани за функцискиот резултат, формалните параметри и локалните променливи се ослободуваат. Вредноста на функцискиот резултат се пренесува како вредност на функцискиот повик.

Процедурални подалгоритми

Функциските подалгоритми се корисни, но ограничени поради тоа што даваат само еден резултат. Некогаш е потребен подалгоритам кој ќе произведе неколку резултати. Такви подалгоритми се *процедуралните подалгоритми - процедури*. **Главна разлика** во однос на **функциските подалгоритми** е тоа што **процедурите не враќаат вредност**.

Процедурите имаат свое име, кое е единствено и по кое се разликуваат и се повикуваат.

Процедурите можат да бидат без параметри и со параметри. Процедурите без параметри претставуваат, всушност, еден блок од програмата, означен со име и крај.

Кај процедурите со параметри, после името, во мали загради се наведуваат параметрите. За секој параметар се наведува и типот. Овие параметри се нарекуваат *формални параметри*, бидејќи тие не се дефинирани додека не се заменат со *вистинските параметри*, кои се задаваат при повикување на процедурата.

До повикувањето, процедурата е непозната и нема никакво дејство.

Во главната програма една процедура може да се повикува повеќе пати.

Процедурите се повикуваат во главната програма преку името, односно со наредба за повикување на процедура.

```
ИмеНаПроцедурата;
```

Кај процедурите со параметри, по името, во мали загради се наведуваат параметрите. Вистинските параметри се задаваат при повикување на процедурата со наредба за повикување на процедура со синтакса:

```
ИмеНаПроцедура(листа на вистински параметри);
```

При повикувањето, формалните параметри се заменуваат соодветно со вистинските параметри. Бројот, редоследот и типот на вистинските параметри, мора да биде ист со формалните параметри.

Да го земеме истиот пример како кај функциите.

```
Procedure Pogolem(broj1,broj2 : integer; var  pog:integer);
Begin
      If  broj1>broj2
           Then
                 pog := broj1
            Else
                 pog := broj2;
End;
Program Najgolem;
Var a,b,c,p,n : integer;
Begin
      ReadLn(a,b,c);
      Pogolem(a,b,p);
      Pogolem(p,c,n);
      WriteLn(n);
End.
```

Анализа: Променливите *a,b* и *p* се вистински параметри. При повикувањето на подалгоритамот се врши замена на формалните параметри со вистинските. Формалниот параметар *broj1* ќе се замени со вистинскиот параметар *a, broj2* со *b* и *pog* со *p*. По извршувањето на подалгоритмот *Pogolem*, променливата p ќе ја содржи вредноста на поголемиот од броевите *a* и *b,* која ја добил преку формалниот параметар *pog*. Значи, *pog* е формален параметар преку кој се пренесува резултатот од подалгоритамот во главниот алгоритам. Слично се случува и ако подалгоритамот го повикаме со: *Pogolem(p,c,n);* Формалните параметри *broj1,broj2* и *pog* се заменуваат со вистинските *p,c* и *n* соодветно.

-Вредносни и променливи параметри

Вредносни параметри (ВП) се формалните параметри кои **внесуваат** вредности во процедурата. Со повикување на процедурата, на секој вредносен параметар му се доделува соодветниот вистински параметар. По ова доделување на вредностите, не постои повеќе заемнодејство меѓу формалните и вистинските параметри. Ако во процедурата се назначи нова вредност на формалните параметри, тоа нема да има ефект над соодветниот вистински параметар. Затоа се дефинираат *променливи*

параметри (ПП) кои можат да ја менуваат својата вредност и **изнесуваат** вредности (резлтати) од процедурата во програмата. Во Паскал се означуваат се зборот VAR пред формалниот параметар.

Пример.

```
Procedure Pogolem(broj1,broj2 : integer; var  pog:integer);
```

Повик на вредносен параметар уште се нарекува *повик по вредност (call by value)*, додека на променлив параметар *повик по адреса(call by adress)*.

Процедуралните алгоритми ќе ги објасниме и со следниов пример, за наоѓање на најмалиот елемент во низата $[a_i]_n$ и неговиот реден број. Бидејќи се бараат два излезни резултати, не може да се напише функциски подалгоритам, туку ќе напишиме процедурален:

```
подалгоритам Најмал_Елемент(n,a,najmal,redbr);
почеток
      najmal <- a1;
      redbr <- 1;
      за k <- 2 зголемувај до n
            ако ak < najmal
                  тогаш
                        najmal <- ak;
                        redbr <- k;
            крај_ако (ak < najmal)
       крај_за(k)
крај(Најмал_Елемент)
```

Процедуралниот подалгоритам се повикува само со неговото име, а во заграда се ставаат вистинските аргументи. На пример, за наоѓање на најмалиот елемент во низата $[c_i]_m$ и неговиот реден број, подалгоритмот *Најмал_Елемент* треба да се повика со чекорот

```
Најмал_Елемент (m,c,min,rb);
```

Променливите m, c, min и br се вистински аргументи. При повикувањето на подалгоритмот се врши замена на формалните аргументи и тоа: n со m, a со c, najmal со min и redbr со rb.

По извршувањето на подалгоритмот, променливата min ќе ја содржи вредноста на најмалиот елемент на низата, која ја добила преку формалниот аргумент najmal, додека променливата rb ќе го содржи

редниот број на најмалиот елемент во низата, кој го добила преку формалниот аргумент redbr. Значи, najmal и redbr се формални аргументи преку кои се пренесуваат резултатите од подалгоритмот во главниот алгоритам. Затоа тие се нарекуваат излезни формални аргументи. Аргументите, пак, n и a во кои се пренесуваат вистинските аргументи од главниот алгоритам во подалгоритмот, се нарекуваат влезни формални аргументи.

При повикување на подалгоритмот, бројот, редоследот и типот на вистинските аргументи мораат да бидат исти со бројот, со редоследот и со типот на формалните аргументи. Алгоритмот за наоѓање на најмалиот елемент во низата [ci]m и неговиот реден број, со повикување на процедуралниот подалгоритам *Најмал_Елемент* е следен:

```
Алгоритам НајмалИРедБрој;

почеток

        читај m;

        за i <- 1 зголемувај до m

              читај ci ;

        крај_за (i)

        Најмал_Елемент(m, c, min, rb);

       печати 'Најмал е ',rb,'-от елемент со вредност ',min;

 крај {НајмалИРедБрој}
```

Накратко, ефектот на процедурален повик е:

1. Се создаваат парови од лево на десно на вистинските со формалните параметри од процедурата. Бројот на параметрите е ист;

2. Се креира мемориска локација за секој вредносен параметар кон кој е придружен соодветниот вистински параметар. Типовите се исти;

3. Секој променлив параметар се поврзува со соодветниот вистински параметар и тој е негов претставник. Вистинскиот параметар мора да е променлива од ист тип како и променливите параметри;

4. Се креира мемориска локација за секоја локална променлива чија иницијална вредност е недефинирана.

5. Се извршуваат наредбите од процедурата. Сега вредносните параметри се однесуваат како и локалните променливи. Секоја промена, пак, на променливите параметри се пренесува на вистинскиот параметар кого го претставува;

6. На крај, сите мемориски локации за формалните параметри и локалните променливи се ослободуваат.

Глобални и локални променливи

Подрачје на декларации

Декларација претставува воведување име и тип на променливите. *Дефиниција* е декларација кога на името на променливата и се дава и значење. *Подрачјето на важење* е множеството на програмски линии во кои е видлива (може да се користи) некоја променлива. Постојат:

- глобална декларација и
- локална декларација

Глобалните променливи имаат подрачје на важење во целата програма. Тие се декларирани надвор од функциите.

Локалните променливи имаат подрачје на важење само во телото на функцијата во која се декларирани или само во блокот во кој се декларирани (локалните променливи имаат *блок подрачје на важење*, помеѓу две загради). *Скриена* променлива е онаа која не е видлива (не може да се користи) во некој блок.(Пример: глобални и локални променливи со исто име)

Следи еден програмски код во кој во коментари се напишани карактеристиките на променливите и нивните вредности:

```
int m=10;             // m e globalna promenliva
void fun(int m, int& n);
main()
{
     int n=20;             // n i p se
     int p=30;             // lokalni za main()
     if (n<p)
     {                     //priv ne moze da se koristi nadvor
          int priv;
          priv=p;
          p=m;                  // m e globalnata promenliva
     }                     // od ova { i ova } zagrada
     cout<<m;              // se pecati globalnata m=10
     int m=44;             // tuka i vo slednata linija vazi
                        //lokalnata m=44,
                      // globalnata m=10 e skriena
     fun(m,p);             // fun se povikuva za m=44 i p=30
}
void fun(int m, int& n) // ovie m i n se lokalni promenlivi za fun
{                     // i vazat samo vo ovaa funkcija
     int p;                // ova p ne e ona od main()
     p=m;
     n=m+p;
}
```

Правила за подрачјето на важење на променливите:

1. Променливата не може да се користи надвор од подрачјето на важење,
2. Глобалните променливи можат да се користат во целата програма,
3. Променливите декларирани во еден блок можат да се користат само во него (освен како формални аргументи)
4. Променлива декларирана во една функција не може да се користи во друга,
5. Променливата може да биде скриена во некој дел од нејзиното подрачје на важење,

6. Не може две различни променливи со исто име да имаат исто подрачје на важење. (Не може да се декларираат две променливи со исто име во различни блокови во иста функција.)
7. Може две функции со исто име да имаат исто подрачје на важење, само ако имаат различни листи на аргументи.

Живот на променливите

Живот на променливите е времето од креирањето до исчезнувањето.

Локалните променливи живеат од моментот на извршување на дефиницијата, до крајот на извршување на блокот (функцијата) во која се дефинирани. Тие се автоматски by default. Автоматските променливи се декларираат со зборот auto. На пример:

```
auto float o1;    // isto so float o1;
int o2;           // isto so auto int o2;
```

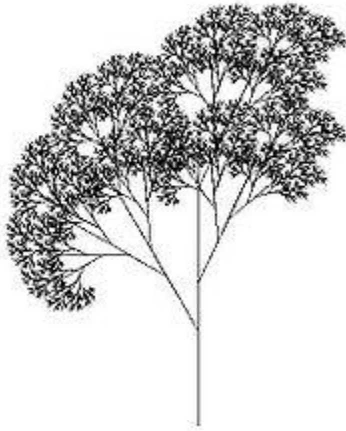
Глобалните променливи се *статички* и тие живеат цело време додека се извршува програмата. И локалните променливи можат да се декларираат со зборот static. На пример:

```
static int broj=n;
static double iznos;
```

Глобалните променливи живеат од моментот на нивното декларирање, до завршување на извршувањето на програмата. Иницијализација: само еднаш. (ако не се иницијализирани автоматски се иницијализираат на 0 - сите битови 0).

Локалните статички променливи живеат од едно до друго повикување на функцијата. При новото повикување на функцијата тие ја имаат состојбата (вредноста) добиена во претходното повикување.

Рекурзија



Рекурзија е дефинирање на некој поим преку самиот себе, односно во дефиницијата се вклучува и поимот кој се дефинира.

Употреба на рекурзијата: дефиниција на мтематички функции, дефиниција на структури на податоци.

латински: **re** = назад + **currere** = извршува;

Да се случи повторно, со нови интервали. Многу математички функции може да се дефинираат рекурзивно:

- факториел
- фибоначиеви броеви
- Евклидов НЗД (најголем заеднички делител)
- аритметички операции

Концепт

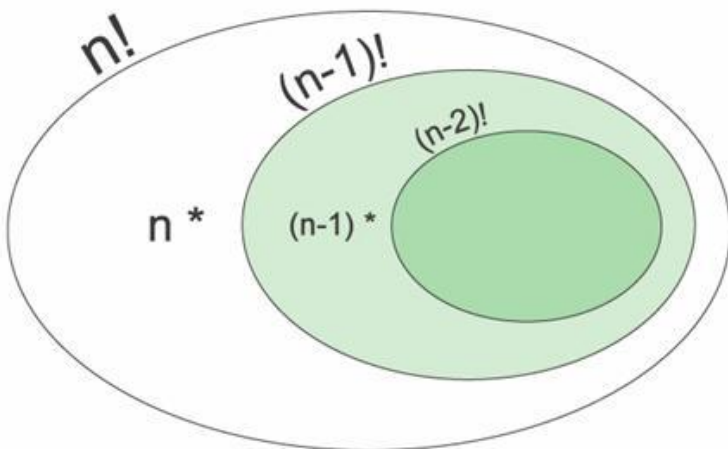
Рекурзијата е важен концепт во компјутерската наука бидејќи многу алгоритми можат со неа најдобро да се прикажат.

Кога се обидуваме да решиме некој проблем со помош на компјутерска програма, се трудиме да го разделиме проблемот на помали делови кои што се полесни за решавање. Често пишуваме посебни методи за да се справиме со овие потпроблеми. Кога потпроблемот е ист како и главниот, но на помало множество податоци, велиме дека проблемот е дефиниран рекурзивно.

Пример1. Фотографијата што ја држи момчево во рака е самата таа фотографија, но со помала големина.

Пример2. Факториел на даден број n.

n!=n*(n-1)!= n*(n-1) (n-2)!=...



Рекурзивни подалгоритми

Покрај повикувањето еден со друг, подалгоритмите можат да се повикуваат и самите себе. Таквото повикување се нарекува повратно (анг. recurrent) повикување, односно повторување (анг. recursion) на повикувањето. Затоа постапката е наречена рекурзија, а подалгоритмите се наречени рекурзивни подалгоритми.

Рекурзивните подалгоритми се многу елегантни и пократки од итеративните, но тешко разбирливи и побавни од нив. Тие можат да се реализираат како процедурални или како функциски подалгоритми.

Факториел

Најрепрезентативен пример на рекурзивен подалгоритам е пресметување на факториел на даден број n:

$$n! = \begin{cases} 1 \dots\dots\dots\dots\dots\dots\dots\dots n = 0 \\ n*(n-1)! \dots\dots\dots n > 0 \end{cases}$$

```
подалгоритам fact(n) ;
почеток
      ако n=0
            тогаш
                    fact <- 1
            инаку
                    fact <- n*fact(n-1);
крај
```

Времето на извршување на алгоритмот е $T(n)=\begin{cases} t1 \dots\dots\dots\dots\dots\dots\dots\dots n = 0 \\ T(n-1)+t2 \dots\dots\dots n > 0 \end{cases}$,

каде што t1 и t2 се константи.

За да се реши оваа рекурентна равенка применуваме техника на **последователна замена**.

Значи имаме:

T(n)=T(n-1)+t2

=(T(n-2)+t2)+t2

=T(n-2)+2t2

=(T(n-3)+t2)+2t2

=T(n-3)+3t2

…

Очигледно дека **T(n)=T(n-k)+kt2**, каде што 1<=k<=n.

Точноста на оваа релација може секогаш да се провери со индукција.

Ако n е познат, тогаш го повторуваме процесот на замена се додека не стигнеме до T(0) на десната страна од равенката. Но n не го знаеме, така што за да се добие T(0) на десната страна ставаме n-k=0, т.е. n=k.

T(n)=T(n-k)+kt2

=T(0)+nt2

=t1+nt2

Значи сложеноста на овој рекурзивен алгоритам за пресм

етување факториел е O(n).

Природен начин за решавање на факториел е пишување на рекурзивна функција која одговара на дефиницијата

```
Example 1.) n!

fact(non-negative integer n)
{
	if (n==0)
	{
		return 1;
	}
	else
	{
		return fact(n-1)*n;
	}
}
```

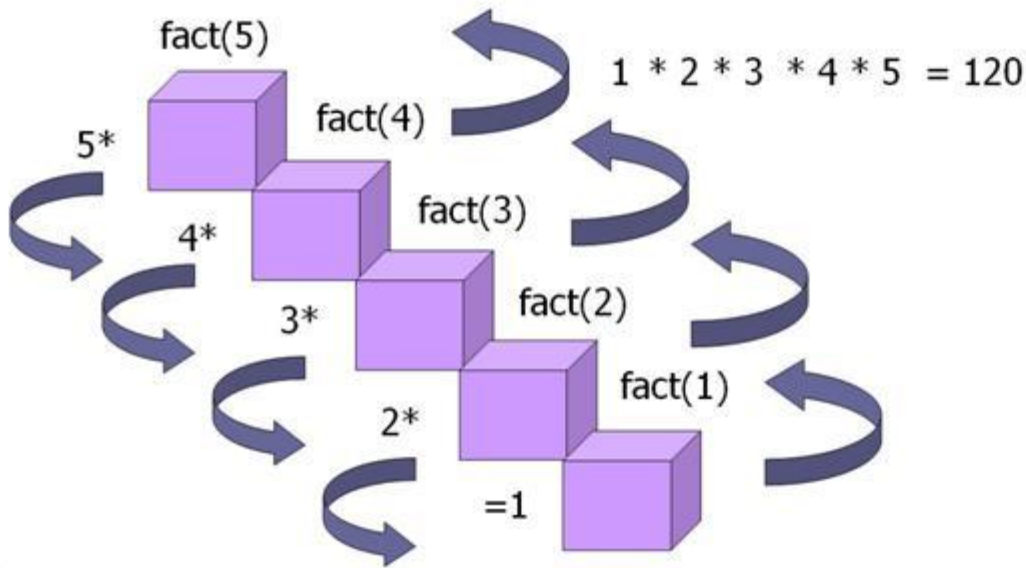
Оваа функција се повикува самата себе за да го пресмета следниот член. На крај ќе дојди до условот за прекин и ќе излези. Меѓутоа, пред да дојде до условот за прекин, таа ќе стави n повици на стек.

Постоењето на услов за прекин е неопходно кога се работи за рекурзивни функции. Ако се изостави, тогаш функцијата ќе продолжува да се повикува самата себеси се додека програмата не го наполни целиот стек, што ќе доведе до несоодветни резултати.

Како работи рекурзијата

При секое самоповикување на подалгоритмот натамошното извршување привремено се прекинува, освен при последното повикување кога целосно се извршува (тоа е граничниот случај). Потоа се продолжува секое прекинато извршување и тоа наназад.

За да се овозможи завршување на подалгоритмот по секое прекинато извршување, потребно е да се паметат вредностите на сите променливи во моментот на прекинувањето, и тоа за сите прекини. Во рекурзивните потпрограми тоа се постигнува со користење на посебен дел од меморијата т.н. стек. Во него се ставаат вредностите на променливите при секој прекин., а при продолжување на извршувањето се вадат.



За да се избегне бесконечното извршување на подалгоритмот:

- Мора да постои одреден критериум (краен критериум, **граничен случај**) за кој подалгоритмот не се повикува самиот себе.
- Секогаш кога подалгоритмот ќе се повика самиот себе мора да биде поблиску до крајниот критериум (граничен случај).

Рекурзијата во компјутерската наука е начин на размислување и решавање на проблеми. Всушност рекурзијата е една од централните идеи во компјутерската наука. Решавањето на проблем со помош на рекурзија значи дека решението зависи од решавањето на помалите инстанци на истиот проблем.

„Моќта на рекурзијата е во можноста на дефинирање на бесконечно множество на објекти со конечна состојба. Бесконечен број на сметања можат да се опишат со конечна рекурзивна програма што не содржи експлицитни повторувања".

Најмногу од комјутерско програмирачките јазици поддржуваат рекурзија дозволувајќи им на функциите да се повикуваат самите себе.

Општ метод на поедноставување е делење на проблемот на подпроблеми од ист тип. Ова е познато како „dialekting". Како и во компјутерската програмирачка техника, ова е познато како раздели па владеј и е клуч на дизјнот на многу важни алгоритми и е фундаментален дел на динамичкото програмирање.

Виртуелно, сите јазици за програмирање што се во употреба денес дозволуваат директна спецификација на рекурзивни функции и процедури. Секоја рекурзивна функција може да биде трансформирана во итеративна функција со користење на стек.

При креирањето на рекурзивна процедура битно побарување е дефинирањето на „основен случај" и потоа дефинирање на правила за решавање на многу покомплексни случаи со основниот случај. Клуч за рекурзивна процедура е тоа што при секој рекурзивен повик, доменот на проблемот мора да се намалува, се додека не се дојде до основниот случај.

Дефиниција: Алгоритамска техника каде функција, во обид да исполни одредена задача, се повикува себеси како дел од самата задача.

Забелешка: Секое рекурзивно решение е составено од два главни делови или случаи, притоа вториот дел има три компоненти.

- **Базен случај** – во кој проблемот е доволно едноставен да биде решен директно, и
- **Рекурзивен случај**. Рекурзивниот случај има три компоненти:
    - **Раздели** го проблемот на еден или повеќе поедноставни или помали делови на проблемот
    - **Повикај** ја функцијата (рекурзивно) за секој дел, и
    - **Комбинирај** ги решенијата на деловите во решение на целиот проблем.

Зависно од проблемот, секое од овие може да биде тривијално или комплексно.

Споредба на рекурзивни со итеративни подалгоритми

Од претходните примери се гледа дека при секое самоповикување на подалгоритмот, извршувањето се прекинува, освен при последното повикување кога целосно се извршува. Потоа, се продолжува секое прекинато извршување и тоа наназад; прво последното прекинато извршување, потоа претпоследното итн. се до првото. За да се овозможи завршување на подалгоритмот по секое прекинато извршување, потребно е да се памети состојбата на сите променливи во моментот на прекинувањето, и тоа за сите прекини. Во рекурзивните потпрограми тоа се постигнува со користење на посебен дел од меморијата т.н. *стек*. Во него се ставаат вредностите на променливите при секој прекин, а при продолжување на извршувањето се вадат.

Рекурзивната постапка го скратува изворниот код на програмата, а во исто време го прави тешко разбирлив. Некои проблеми со рекурзивна постапка се решаваат доста лесно и елегантно, меѓутоа не треба да се сфати дека рекурзијата е начин за ефикасно програмирање.

Секое рекурзивно решение има и своја нерекурзивна варијанта.

Пример:

Проблем: Собери 1 000 000 денари за хуманитарни цели

Претпоставка: Секој е подготвен да донира по 100 денари

- ***Итеративно решение***

Посети 10 000 луѓе, барајќи им на сите по 100 денари

- ***Рекурзивно решение***

Ако е побарано од вас да дадете 100 денари, дајте му ги на човекот што ви ги побарал

Инаку

Посети 10 луѓе и побарај од секој од нив 1/10 од сумата што вам ви ја бараат да ја соберете

Собери ги парите што ви ги дале тие 10 луѓе и дај му ги на човекот што ви ги побарал

Евклидов алгоритам за наоѓање најголем заеднички делител (НЗД) на два цели броја

Се задава рекурзивно на следниот начин:

НЗД на два броја x и y е еднаков на НЗД од y и остатокот при делење на x со y т.е. НЗД(x, y)=НЗД(y, x mod y) (општ случај).

ако y=0 тогаш НЗД ја добива вредноста на x (граничен случај).

$$NZD(x, y)= \begin{cases} NZD(y, x \bmod y) \ldots\ldots\ldots\ldots y \neq 0 \\ x \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots y = 0 \end{cases}$$

Во Паскал:

```
function evklid(m:integer, n:integer):integer;

begin

  if n=0
  then
      evklid:=m;
      evklid:=evklid(n, m mod n);

end;
```

Повикувајќи ја оваа функција со evklid(314159, 271828) ги имаме овие рекурзивни (вгнездени) повици последователно:

```
evklid(271828, 42331)

evklid(42331, 17842)

evklid(17842, 6647)

evklid(6647, 4548)

evklid(4548, 2099)

evklid(2099, 350)

evklid(350, 349)

evklid(349, 1)

evklid(1, 0)


Значи НЗД е: 1
```

Кај евклидовиот алгоритам можевме да постапиме и вака, со што итеративно би го решиле проблемот:

```
function evklid(m:integer, n:integer):integer;
var pom:integer;
begin
  while n<>0 do
  begin
    pom:=n;
    n:=m mod n;
    m:=pom;
  end;
  evklid:=m;
end;
```

Рекурзивното решение е природно и се наметнува за решавање на математичките функции. За некои математички проблеми рекурзивното решение е и единствено. Таков пример е Акермановата функција зададена со релацијата:

$$A(n,m)=\begin{cases} m+1 \ldots\ldots\ldots\ldots\ldots\ldots n=0 \\ A(n-1,1) \ldots\ldots\ldots\ldots\ldots n\neq 0, m=0 \\ A(n-1, A(n,m-1)) \ldots\ldots n>0, m>0 \end{cases}$$

Можно е да се напишат едноставни рекурзивни програми кои се многу неефикасни. Пример, пресметување на првите n членови на Фибоначиевата низа зададена рекурзивно:

F(0)=F(1)=1 (гранични случаи)

F(n)=F(n-1)+F(n-2), n>=2 (рекурзивно правило)

$$F(n)=\begin{cases} F(n-1)+F(n-2) \ldots\ldots n\geq 2 \\ 1 \ldots\ldots\ldots\ldots\ldots\ldots n=0 \\ 1 \ldots\ldots\ldots\ldots\ldots\ldots n=1 \end{cases}$$

Тоа се следните броеви: 1,1,2,3,5,8,13,21,34….

1. <u>Добро решение</u> (не користиме рекурзија).

```
function fibonaci(n:integer):integer;

var f0,f1,i,pom:integer;

begin

       f0:=1;

       f1:=1;

       if n=0 then fibonaci:=f0;

       if n=1 then fibonaci:=f1;

       for i=2 to n do
              begin

                     pom:=f0;

                     f0:=f1;

                     f1:=f1+pom;

              end;

              fibonaci:=f1;

end;
```

Очигледно дека сложеноста на овој алгоритам е **линеарна** O(n).

2. <u>Лошо решение</u> (користиме рекурзија на лош начин)

```
function fibonaci(n:integer):integer;

begin

      if n=0 then fibonaci:=1;

      if n=1 then fibonaci:=1;

      fibonaci:=fibonaci(n-1)+fibonaci(n-2);

end;
```

Кодот изгледа многу компактен и читлив, ама е исклучително неефикасен.

Се покажува дека сложеноста на алгоритмот во погорната имплементација е **експоненцијална**, а тоа е се разбира неприфатливо. Итеративното решение имаше **O(1)** сложеност.

Неефикасноста на ова решение има и свое интуитивно објаснување кое се добива кога ги пратиме вгнездените функциски повици. На пример да пресметаме fibonaci(5):

```
fibonaci(5)=fibonaci(4)+fibonaci(3)

fibonaci(5)=fibonaci(3)+fibonaci(2)+fibonaci(3)
```

Се дуплира пресметувањето за fibonaci(3).

Стратегија за работа со рекурзија

Два основни методи за работа со рекурзија се **раздели па владеј** (divide and conquer) и **динамичко програмирање** (dynamic programming).

· **Раздели па владеј (**анг. divide and conquer**)** -го дели проблемот на потпроблемите кои ги решава. Оваа метода функционира добро кога се потпроблемите независни. Меѓутоа кога ќе се примени директно на проблем чии потпроблеми не се независни како на пример горниот пример со Фибоначиевите броеви добиваме неефикасен алгоритам во кој се повторуваат беспотребни пресметувања. (пребројте колку пати пресметавме fibonaci(0) горе.)

· **Динамичко програмирање (**анг. dynamic programming**)** Постојат два типа

o **Од доле нагоре (bottom-up dynamic programming)** –ги пресметуваме по ред вредностите до зададените. Во секој чекор користиме веќе пресметани вредности со кои пресметуваме нови. Се разбира ова може да го употребиме ако имаме начин како да ги чуваме пресметаните вредности.

o **Од горе надоле (**top-down dynamic programming**)** –ја пресметуваме бараната вредност како во раздели па владеј методите, меѓутоа секоја конечно пресметана вредност ја запамтуваме и секојпат кога пресметуваме нова вредност користиме веќе пресметани вредности за да избегнеме повторни пресметки како кај лошото рекурзивно решение на Фибоначиевиот проблем.

o Во двата случаи може да се чуваат веќе пресметаните вредности во некое (доволно) големо поле или поврзана листа, кои ги користи функцијата што имплементира динамичко програмирање.

Рекурзија во поврзани листи

Ќе наведеме неколку примери на рекурзивни подалгоритми за поврзани листи. Ова е природно затоа што поврзаните листи и самите можат да се дефинираат рекурзивно. Да земеме дека секој јазел има своја нумеричка вредност.:

```
type jazel = record

     vrednost : integer;

     sleden : ^jazel;

end;
```

Сите подалгоритми наведени подоле користат заглавие, наречено анкер што покажува на почетокот од листата.

1. Одредување должина на листа.

За да се избројат јазлите во листата, треба да провериме дали анкерот покажува кон јазел или кон ништо (NIL). Ако покажува кон ништо, тогаш должината на листата е нула. А ако покажува кон јазел, тогаш броиме 1 за првиот јазел и продолжуваме да ги броиме јазлите од остатокот на листата.

Како и да е, покажувачот кон вториот јазел во листата е покажувач кон почеток на листа која е пократка од листата која ја набљудуваме. Броењето на јазлите во пократката листа е ист проблем како и тој што требаше да го решиме, само на помалку податоци. Значи го решаваме овој потпроблем со рекурзивно повикување на истиот метод.

Можеме да ја претставиме должината на листата на следниов начин:

Граничен случај: Ако анкерот покажува кон ништо, тогаш должината е 0.

Општ случај: Должината е 1 плус должината на листата без првиот јазел.

Во Паскал кодот би изгледал вака:

```
function dolzina(anker:^jazel):integer;
begin
    if anker=NIL then
        dolzina:=0
    else
        dolzina:=1+dolzina(anker^.sleden);
end;
```

Со мала измена можеме да ја пресметаме сумата на вредностите на јазлите:

```
function suma(anker:^jazel):integer;
begin
    if anker=NIL then
        suma:=0
    else
        suma:=anker^.vrednost+suma(anker^.sleden);
end;
```

2. Печатење на вредностите на листата.

```
procedure pecati(anker:^jazel);

begin

       if anker=NIL

       then

            break

       else

       begin

            writeln(anker^.vrednost);

            pecati(anker^.sleden);

       end;

end;
```

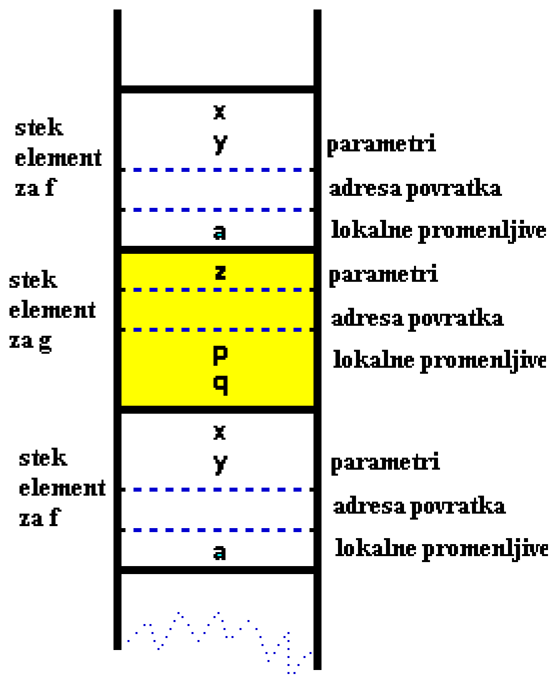
3. Печатење на вредностите на листата во обратен редослед.

Процедурата pecati се повикува рекурзивно дури не се посети последниот јазел. И потоа со враќање наназад од таму кај што биле прекинати се печатат вреднотите на листата во обратен редослед.

```
procedure pecati(anker:^jazel);

begin

      if anker=NIL

      then

           break

      else

      begin

           pecati(anker^.sleden);

           writeln(anker^.vrednost);

      end;

end;
```

По правило, модерните програмирачки јазици користат стек за повикување на работната меморија на сите повикани функции. Кога се повикува некоја процедура или функција, одреден број на повикувачки параметри се ставаат на стек. Кога се врши враќање од функцијата во повикувачкиот редослед, повикувачките параметри се вадат од стекот.

Кога функција повикува друга функција, најпрво се нејзините аргументи, адресата на враќање и конечно просторот за локалните променливи што се ставаат на стекот. Бидејќи секоја функција работи во сопственото опкружување или контекст, постои можност да функцијата се повика самата себе - значи како рекурзија. Оваа можност е екстремно корисна - бидејќи многу проблеми елегантно се решаваат на рекурзивен начин.



Стек, после извршување на неколку рекурзивни функции:

```
int f(int x, int y) {
int a;
    if ( услов_за_прекин ) return ...;
    a = .....;
    return g(a);
}

int g(int z) {
int p,q;
    p = ...; q = ...;
    return f(p,q);
}
```

Гледате дека функциите f и g односно нивните параметри и локални променливи се наоѓаат на стекот. Кога функцијата f се повика по втор пат од функцијата g, се креира нов повикувачки формат за вториот повик на функцијата f.

Структурите на податоци исто така можат рекурзивно да се дефинираат. Една од најважните класи на структури - дрвата, дозволуваат рекурзивни дефиниции кои водат до рекурзивни функции за нивна обработка.

Динамичко програмирање 1

Динамичкото програмирање обично се применува во проблемите на оптимизација: проблемот може да има многу решенија, секое решение има вредност, а се бара решение кое што има оптимална (најголема или најмала) вредност. Во случај да постојат повеќе решенија кои што имаат оптимална вредност, обично се бара кое било од нив. Во една широка класа на проблеми на оптимизација, едно од оптималните решенија може да се најде користејќи го динамичкото програмирање.

Поим и историјат

Динамичко програмирање претставува начин на програмирање односно тип на алгоритми, со помош на кои се доаѓа до **оптимални вредности** на широка класа на проблеми од определен тип, така што алгоритмот во повеќето случаи претставува **оптимално решение на проблемот**.

Самиот збор "програмирање" слично како и кај линеарното програмирање се однесува на пополнување на табели при решавање на проблемот, а не на употребата на компјутери и програмски јазици. Техниките на оптимизација кои имаат елементи на динамичко програмирање биле познати и порано, но денеска за автор на методот се смета професор *Richard E. Belman.* Во средина на 50-те години *Belman* го проучувал динамичкото програмирање и дал цврста математичка основа за овој начин на решавање на проблемите.

Воопштено зборувајќи, проблемот се решава така што се воочува хиерархијата на проблемите од ист тип, содржани во главниот проблем и решавањето започнува од наједноставните проблеми. Вредностите и деловите на решенијата на сите решени потпроблеми се паметат во табела, па потоа со нивните комбинации се добиваат решенија на поголемите потпрограми се до решение на главниот проблем.

Идеја за реализација

Да ја илустрираме идејата на еден од најпознатите проблеми на динамичкото програмирање - проблемот на ранец *(knapsack problem)*.

1) Проблем на ранец

За овој проблем постојат неколку познати варијанти и секоја може да се реши со повеќе формулации. Во продолжение ја даваме онаа формулација (една од варијантите) по која проблемот го добил името: крадец со ранец во кој може да се сместат N волуменски единици, влегол во просторија во која што се чуваат скапоцени предмети.

Во просторијата има вкупно M типови на предмети, секој во многу големи количини (повеќе отколку што може да се собере во ранецот). За секој тип на предмет позната е неговата вредност $V(k)$ и неговиот волумен $Z(k)$, $k=1,M$. Сите големини се целобројни. Крадецот сака да го наполни ранецот со најскапоцена содржина. Потребно е да се одредат предметите кои треба да ги стави во ранецот и нивната вкупна вредност.

Решение:

Најнапред да направиме неколку важни забелешки за подобро да се објасни суштината на проблемот.

- Ранецот кој што е оптимално пополнет не мора да биде пополнет до врвот, или попрецизно, збирот на волуменот на предметите ставени во ранецот не мора да биде еднаков на волуменот на ранецот. Важно е тој збир да не е поголем од волуменот на ранецот, а во исто време збирот на вредностите на тие предмети да биде максимален. На пример, нека ранецот има капацитет од $N=7$ волуменски единици и нека постојат $M=3$ предмети такашто $V=(3,4,8)$ и $Z=(3,4,5)$ односно првиот предмет има вредност 3 и волумен 3, вториот вредност 4 и волумен 4, а третиот предмет има вредност 8 и волумен 5. Една варијанта за пополнување на ранецот е со полнење на ранецот до врвот така што во него ги ставаме првиот и вториот предмет бидејќи $z_1+z_2=3+4=7=N$. Сепак, таквото пополнување не е оптимално бидејќи неговата вредност е $v_1+v_2=3+4=7$, додека со ставање на третиот предмет во ранецот се добива ранец со повредна содржина $v_3=8$, а при тоа ранецот не е пополнет до врвот $(z_3=5<7=N)$.
- Врз основа на претходниот пример, се добива впечаток дека до решението се доаѓа така што се одредува k за кое $V(k)/Z(k)$ е најголемо, па ранецот се полни само со предметот k. Овој начин не претставува решение што исто така ќе го покажеме на поедноставен пример: нека е $N=7$, $M=3$, $V=(3,4,6)$, $Z=(3,4,5)$. Наведената идеја (алчен избор) наложува да се избере третиот предмет, како највреден по единица волумен. Според тоа во ранецот би се ставил само еден од предметите од третиот тип (во иднина: предмет број 3), а вредноста на ранецот би била еднаква на 6. Лесно може да се согледа дека изборот на првите два предмети дава вредност на рабецот 7, што е подобро (како и оптимално) решение. Идејата за алчен избор води кон решение во случај да не мора да се земаат цели предмети и вредноста на дел од некој предмет да е сразмерна со големината на тој дел.

Карактеристики на решението

Од овој пример се заклучува дека проблемот на ранец не е тривијален и до решението не може да се дојде директно. Потребна е повнимателна анализа својствена на проблемот и решението, на основа која што подоцна ќе го конструира решението.

Анализирајќи го проблемот можеме да се согледа следното: ако при оптимално пополнување на ранецот последен избран предмет е x, тогаш претходно избраниот предмет на оптимален начин го пополнува ранецот со капацитет N-z_x. Овој заклучок лесно се докажува со сведување на контрадикција. Имено да претпоставиме дека постои друг начин подобро да се пополни ранецот со капацитет N-z_x. Нека на потполно ист начин се пополнат и првите N-z_x единици волумен на ранецот со големина N и потоа нека го додадеме x-тиот предмет. Со тоа се добива пополнување на целиот ранец, кој е подобар од почетниот, што е невозможно бидејќи почетното пополнување е оптимално.

Според тоа, **оптималното решение на проблемот содржи во себе оптимални решенија на проблемите од истиот тип содржани во главниот проблем**. Вообичаено е во ваков случај да се каже дека проблемот има оптимална подструктура. Наведеното својство се нарекува и својство на *Belman*, по авторот на методите на динамичкото програмирање. Ова е клучна карактеристика за примена на динамичко програмирање. Благодарејќи на оваа карактеристика, можеме да дојдеме до оптималното решение на проблемот, комбинирајќи со оптималните решенија на проблемите.

Ќе докажеме, дека користејќи ја математичката индукција, за секое целобројно N (и дадените типови на предмети, кои не се менуваат) умееме да најдеме најдобро пополнување на ранецот со капацитет N. Оптималното решение за $N=0$ е празен ранец. Нека се познати решенијата за сите ранци со капацитет q и нека е $B(q)$ збир на сите вредности, а $S(q)$ низа на редни броеви на предметите ставени во ранец со капацитет q при оптимален избор.

Секој од M предметите кои што можат да се соберат во празен ранец со капацитет N, да го испробаме како последен избран за ранец со капацитет N. Остатокот на ранецот во секој од овие случаи да ги пополниме на оптимален начин што можеме на основа на индуктивната хипотеза. Најголемата добиена вредност од сите ранци на капацитетот N ќе биде оптимална. Оваа претпоставка следува директно на основа на оптималноста на подструктурите, бидејќи еден предмет мора да биде последен, а ние го испробувавме секој како последен.

Според досега кажаното, решението за ранецот со капацитет N е:

$$B(N) = \max_{\substack{1 \le x \le M \\ Z(x) \le N}} \{V(x) + B(N - Z(x))\}, \quad S(N) = S(N - Z(x_N)) \cup \{x_N\}$$

каде што x_N е она x кое остварува максимум во $B(N)$. Да забележиме дека нема потреба да го паметиме целиот збир $S(q)$ за секој капацитет на ранецот q, односно доволно е како член на низата $C(q)$ да се запамти последниот додаден елемент x_N. Сите елементи тогаш може да се најдат со редот (од последниот кон првиот) и тоа се:

$$a = C(N),\ b = C(N\text{-}Z(a)),\ c = C(N\text{-}Z(a)\text{-}Z(b)),\ d = C(N\text{-}Z(a)\text{-}Z(b)\text{-}Z(c)), \ldots$$

или додека не се добијат сите предмети од ранецот.

Рекурзивно програмско решение

Задачата може да се реши со употреба на рекурзивна потпрограма *napolni(q,B,C)*, која за ранецот со капацитет q ја наоѓа неговата оптимална вредност B и редниот број на последниот додаден предмет C. Ќе сметаме дека низите V и Z како и бројот на типови на предмети M се глобални големини.

```
procedure napolni(q:integer; var B:integer; var C:integer);
var BB, CC : integer;
begin
            B:=0;
            C:=0;
            if q>0 then
            begin
                 for i:=1 to M do
                      if Z[i]<=q then
                      begin
                            napolni(q-Z[i],BB,CC);
                            if BB+V[i]>B then
                            begin
                                  B:=BB+V[i];
                                  C:=i;
                            end;
                      end;
            end;
end.
```

Од главната програма би се повикала потпрограмата *napolni(N,B,C)*, а секое повикување на ова потпрограма може да предизвика M нови повикувања поради решавање на генерализираните потпроблеми. На пример за $N=1000$, $M=10$ и предметите со волумен од $Z_{min}=5$ до $Z_{max}=10$, би требало помеѓу $M^{N/Zmax} = 10^{100}$ и $M^{N/Zmin} = 10^{200}$ рекурзивни повикувања на потпрограмата *napolni* и тоа само на последното ниво на длабочината на рекурзијата (листови во дрвото на рекурзивните повикувања). Кога секое повикување би траело една наносекунда ($10^{-9}s$), би требало повеќе од $10^{91}s > 10^{83}$ години, а староста на комплетната вселена се проценува на помалку од 10^{12} години!

Решение со динамичко програмирање

Наместо рекурзивно, проблемот можеме да го решиме и со редови од $q=1$ до $q=N$, пополнувајќи табели за B и C. За секое q потребно е M преминувањена низ циклусот за да се определи последниот избран елемент и најголемата вредност, што значи дека вкупниот број операции е пропорционален со

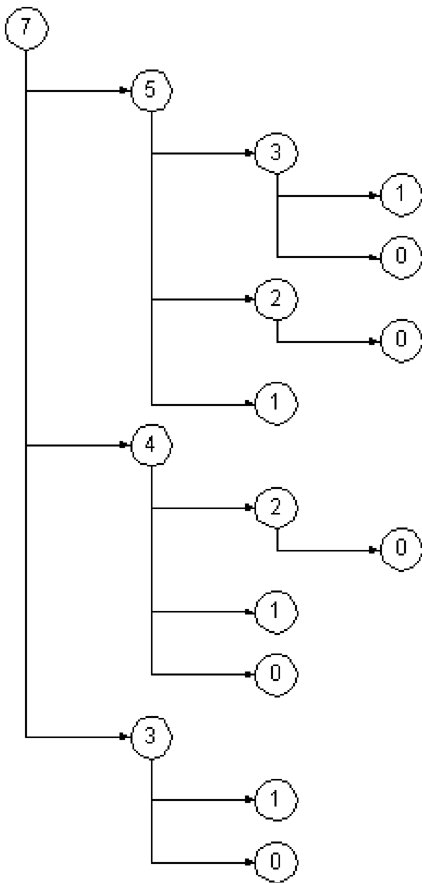
MN. Во горниот пример тоа би било десет илјади циклуси од неколку пресметувачки чекори, што на денешните компјутери се извршува за значително помалку од една секунда.

```
var
            B, C:array[0..1000] of longint;
            V, Z:array[1..10] of integer;
            N, M, q, i : integer;
begin
            readln(N,M);
            for i:=1 to M do
                  readln(V[i], Z[i]);
                  B[0]:=0; C[0]:=0;
                  for q:=1 to N do
                  begin
                       B[q]:=0; C[q]:=0;
                       for i:=1 to M do
                            if Z[i]<=q then
                                   if B[q-Z[i]]+V[i]>B[q] then
                                   begin
                                          B[q]:=B[q-Z[i]]+V[i];
                                           C[q]:=1;
                                   end;
                 end;
            writeln('Optimalna  vrednost  na  sodrzinata  na  ranecot  e',
B[N]);
            writeln('Izbranite  predmeti  se:');  {se  ispisuvaat  nanazad,
sto ne e bitno}
            q:=N;
            while c[q]>0 do
            begin
                  writeln(c[q]);
                   q:=q-z[c[q]];
            end;
end.
```

Од каде ваква драматична разлика во ефикасноста на понудените решенија? Да испитаме подетално на помал пример, како работи рекурзивниот алгоритам. Нека е даден пример за ранец со волумен *N=7* и нека постојат *M=3* предмети со волумен *Z=(2,3,4)*. Вредностите на предметите не се од значење за следење на текот на рекурзивните повикувања. На сликата 1 е прикажана хиерархија на рекурзивните повикувања на потпрограмата *napolni* во облик на дрво. Јазлите на дрвото се наведени по редослед на настанување, т.е. по редослед на повикувања на примерокот на потпрограмата *napolni*, претставен со тие јазли. Ознаките на јазлите претставуваат вредности на аргументот *q*, т.е. на големината на ранецот кој треба да се пополни.

Како што гледаме, при преминувањето по дрвото на рекурзивните повикувања, еден ист проблем (пополнување на ранецот со иста големина *q*) се среќава повеќе пати и секој пат (непотребно) се решава повторно. При тоа бројот на повторените решавања по правило расте експоненцијално со зголемување на димензиите на почетниот проблем. Со други зборови, **потпроблемите имаат заеднички потпотпроблеми, односно делумно се преклопуваат**.

Преклопувањето на потпроблемот е втора особина која е значајна за примена на динамичкото програмирање. Оваа особина не е неопходна за да се примени динамичкото програмирање, но без преклопувања на потпроблемите динамичкото програмирање ја губи својата голема предност во брзина во однос на рекурзијата, бидејќи тогаш со рекурзијата секој потпроблем се креира и решава само еднаш. Кога нема преклопување на потпроблемите, во некои проблеми се случува рекурзивното решение да работи побргу и/или да троши значително малку од меморискиот простор (види глава “Мемоизација”).



Слика 1: Хиерархија на рекурзивни повици на потпрограмата *napolni.*

Варијанти на проблемот на ранец

Веќе е споменато дека проблемот на ранец може да се појави во неколку различни варијанти. Проблемот може да се постави така што за секој предмет да се зададе број на расположливи примероци или така што од секоја врста на предмети да е на располагање точно по еден примерок.

2) Проблем на ранец со само едно појавување на даден предмет

Ќе го разгледаме детално и случајот кога секој предмет може да се земе најмногу еднаш.

Решение:

Нека е познато оптималното пополнување на ранецот со големина *N*. Ако во тоа пополнување не учествува *M*-тиот предмет, тогаш истото пополнување е оптимално и за ранецот со големина *N* и првите *M-1* предмети. Ако во оптималното пополнување учествува и *M*-ти предмет, тогаш останатите предмети од ранецот прават оптимално пополнување за ранецот со големина *N-Z(M)* и првите *M-1* предмети. Заклучуваме дека проблемот и понатаму има оптимална подструктура, но сега големината на проблемот се задава со два пареметри, а тоа се капацитетот на ранецот и бројот на предметите. Да ја означиме со *B(X,Y)* најголемата вредност на ранецот со капацитет *X*, пополнуван со некои од првите *Y* предмети. Тогаш е:

$$B(X,0) = 0$$

$$B(X,Y) = \begin{cases} B(X,Y-1), & \text{ako e } Z(Y) > X \\ \max\begin{cases} B(X,Y-1) \\ V(Y) + B(X - Z(Y)), Y-1) \end{cases} & \text{ako e } Z(Y) \le X \end{cases}$$

Според тоа, потпроблемите може да се решаваат по следниот редослед: прво за сите ранци и еден предмет, па за сите ранци и два предмети, итн. По решавањето на *B(N,M)* имаме решение на почетниот проблем.

И тука како и во претходната варијанта, поради реконструкција на решението доволна е дополнителна низа *C*, каде што во *C(X)* ќе го памтиме последниот ставен предмет при оптимално пополнување на ранецот со капацитет *X*.

При пополнување на матрицата *B* по колоните се користат само вредности од претходната колона (т.е. на колоната *Y-1)*. Благодарејќи на тоа, наместо матрицата *B* со *M* колони можеме да ја користиме матрицата само со две колони, што е значителна заштеда на просторот, која што може пресудно да делува на применливоста на постапката (за некои вредности *M* и *N*). Разгледувајќи ја повнимателно релацијата по која се пресметува *B(X,Y)*, гледаме дека потребните елементи од претходната колона, сите имаат реден број на врста помал или еднаков на *X*. Со тоа при пополнување на *Y*-та колона на матрицата *B* наназад (од последната врста кон првата), можеме сите операции да ги изведеме во иста колона, па за чување на потребните податоци за доволна низа *B* (ако се користи на опишаниот начин). Во продолжение следува програмата:

```
var
            b, c:array[0..1000] of longint;
            v, z:array[0..10] of integer;
            N, M, x, y: integer;
begin
            write ('n,m='); readln(N,M);
            for y:=1 to M do readln(v[y], z[y]);
            for x:=1 to N do begin
                        b[x]:=0;
                        c[x]:=0;
            end;
            for y:=1 to M do
                        for x:=N downto 1 do
                                    if z[y]<=x then
                                                if b[x] < v[y]+b[x-z[y]]
then begin
                                                            b[x]:=v[y]+b[x-
z[y]];
                                                                  c[x]:=y;
                                    end;
            writeln('Optimalna vrednost na sodrzinata na ranecot e',
b[N]);
            writeln('Izbranite predmeti se:'); {se ispisuvaat nanazad,
sto ne e bitno)                         w:=N;
            while c[x]>0 do begin
                        writeln(c[x]);
                        x:=x-z[c[x]];
            end;
end.
```

При решавање на проблемот со оваа варијанта (секој предмет најмногу еднаш), треба да имаме на ум уште една важна претпоставка: решавањето на проблемот со динамичко програмирање се исплаќа единствено ако бројот на предметот *M* е доволно голем, а капацитетот на ранецот *N* релативно мал, така што *MN* операциијата (колку приближно е потребно за овој начин на решавање) да се изврши побргу отколку да се најдат сите 2^M можни поднизи и нивните збирови на вредностите, односно волуменот.

Да речеме за *N=10000, M=100*, динамичкото програмирање на решението го добиваме без чекање, додека 2^{100} поднизи практично не можеме да формираме. Меѓутоа, во случај на мала вредност на *M*

(на пример десет) и голема вредност на *N* (на пример сто милиони) подобро е да се испробат сите поднизи.

Проблемот на ранец е од исклучително значење во практиката и неговите решенија често се користат, на пример при транспортот на стока. На натпреварите исто така можат многу често да се сретнат овие или други варијанти на проблемот на ранец со можни усложнувања или поедноставувања. Да ги спомнеме познатите примери:

3) Подниза со даден збир

Од дадената низа на природни броеви *A*, да се одвои подниза чиј збир на елементите е даден број *M* или да се испише порака дека таква низа не постои.

Решение:

Со *S* да ја означиме сумата, а со *N* бројот на елементите во низата. Можеме да сметаме дека со низата *A* се зададени такви предмети со кои е $v_1=z_1=a; i=1,N$. Сега е јасно проблемот да се сведи на оптимално пополнување на ранецот со капацитет *M*, при што решенија има ако и само ако вредноста на оптималната содржина на ранецот е еднаква на неговиот волумен *M*, т.е. ако и само ако ранецот е наполнет до врв.

4) Подниզи со минимална разлика на збирови

Броевите од даената низа на природни броеви А, да се поделат во две групи така што разликата на збировите на елементите во поедините групи да биде минимална.

Решение:

Во овој пример низата *A* повторно има улога како и низата *Z* и низата *V*. Нека *S* и *N* имаат исти значења како и во претходниот пример. Решението на проблемот се состои во следново: предметите кои сочинуваат оптимално пополнување на ранецот со капацитет *S/2* (делењето е целобројно), треба да се спои во една а сите останати предмети во друга група. Ако ранецот е пополнет до врв групите за парно *S* ќе имаат еднакви збирови. Во спротивно првата група има помал збир, а другата поголем, но тие збирови се најблиску до вредноста *S/2* а со тоа и еден до друг.

5) Постави загради во алгебарски израз со минуси

Даден е изразот a_1-a_2\-...-a_n, каде што сите броеви во низата се цели. Да се поставаат загради во овој израз така што вредноста на изразот да биде еднаков со зададен број *X*.

Решение:

Проблемот може еквивалентно да се постави на следниот начин: броевите од дадената низа на цели броеви *A* да се поделат во две групи така што a_1 задолжително да припадне во првата, а a_2 во втората група и разликата на збировите на елементите на првата и втората група да биде еднаква на *X*. Оваа формулација е еквивалентна, бидејќи заградите секогаш може да се поставаат така што пред броевите од првата група има парен број на минуси, а пред бреовите од втората група непарен број на минуси кои однесуваат на тие броеви. Оваа постапка на поставување загради веројатно е полесно да се изведе, отколку прецизно да се опише, па на читателот му препуштаме после формирањето на двете групи на броеви, заградите да ги постави сам.

Нека S_1 и S_2 се суми на броевите од првата и втората група, а *S* сума на сите *N* броеви така што $X=S_1-S_2=2S_1-S$. Сега решавањето на преформулираниот проблем се сведува на избор на некои од броевите $a_3,a_4,...,a_n$ (внимавајте дека a_1 и a_2 се изоставени), така што збирот на избраните броеви е еднаков на $M=(X+S)/2-a_1$, а таа задача е веќе разгледувана.

Резиме

Во сите овие три проблеми, кои се сведуваат на проблем на ранец, варијантата “секој предмет најмногу еднаш”, до решение може да се дојде на потполно ист начин како и во случајот елементите на низата да се цели броеви (а не природни) броеви. Услов елементите на низата да се позитивни, е даден само затоа што во спротивна интерпретација се губи физичката смисла: мора да се воведат негативни волумени и негативни вредности. Особината на низата која е суштински битна во овие три проблеми е да се вредностите на елментите на низата, како и нивниот вкупен број на умерени целобројни големини. За низа од *N* цели броеви кои се сите по апсолутна вредност помали од *G*, сумите и/или разликите на елементите на сите поднизи (секој елемент може да се земе директно, со променлив предзнак или да се изостави), можат да имат помалку од *2NG* различни вреднсоти, било позитивни, било негативни. Типично, за *N=G=100*, со динамичко програмирање лесно се испитува за секој од можните 20000 броеви, дали може да се појави како сума или разлика на некои елементи од низата, позитивни или негативни. Ваква постапка за решавање не би можела да се примени во случајот можните кандидати за збир (или разлика) на некои (или сите) елементи на низата да има за неколку редови големини повеќе, а тоа се случува ако е ограничувањето на елементите на низа многу големо, или ако се допуштат реални броеви. Тогаш мора да се испробат сите поднизи, што значи дека ефикасна постапка во тој случај нема.

На крајот на ова поглавје да сумираме што се беше потребно за решавање на секоја од варијантите на проблемот на ранец (или кој било друг проблем) со динамичко програмирање:

- Ја анализираме структурата на оптималното решение. Треба да се согледа дека извесни делови на едно оптимално решение на проблемот всушност е оптималното решение на помали проблеми од истиот тип. Да ги одредиме параметрите кои што задаваат големина на проблемот и потпроблемите.
- Вредноста на оптималното решение да ја зададеме рекурентно, т.е. користејќи вредности на оптималното решение на некои помали проблеми.
- Да најдеме ефикасен начин за на основа на вредностите на оптималните решенија на потпроблемите да добиеме вредност на оптималното решение на проблемот (типично користејќи еден циклус, понекогаш само неколку наредби).
- Да ги најдеме и табеларно вредностите на решенијата за наједноставните потпроблеми, а потоа да ги комбинираме вредностите на решенијата на помалите потпроблеми, користејќи го пронајдениот начин наоѓаме вредности на решенијата на поголемите потпроблеми, се до решавањето на самиот проблем. При тоа вредностите ги табелираме вредностите на решенија и делумните информации за начинот на добивање на решенија за сите потпроблеми.
- На основ на табеларните информации го конструираме бараното оптимално решение.

Динамичко програмирање 2

Во наредниот текст решени се уште некои проблеми со примена на динамичкото програмирање. Користени се следните договори и ознаки:

Ако A е низа од N елементи, со A_k ја означуваме низата која што се состои од првите K елементи на низата A, каде K£N. Притоа, A_0 е празната низа, а A_N е еднаква на целата низа A. Елементите на низата A ги означуваме со $a_1, a_2, \ldots, a_N$ или со $A(1), A(2) \ldots A(N)$. Во случај на двојни индекси, поради подобра читливост нема да пишуваме туку $a(b_c)$, или $a(b(c))$. Истите забелешки важат и за матриците.

1) Проблем на максимален збир

Нека е дадена правоаголна шема A со MxN полиња сместени во M редици и N колони и пополнети со цели броеви. Од секое поле е дозволено да се премине само на полето под или на полето десно од тоа поле. Потребно е да се избере пат од горното лево поле до долното десно поле, така што збирот на броевите во полињата преку кои се поминува треба да биде максимален. Да се испише вредноста на оптималниот пат, а потоа и патот како низа на координатни полиња преку кои се поминува.

Решение:

Како што може да се претпостави, генерирањето на сите можни патишта и памтење на оној пат кој што има најголем збир, не е добра идеја, бидејќи бројот на можни патишта расте со експоненцијална брзина со порастот на големината на правоаголникот.

Да се потсетиме на условите за примената на динамичко програмирање:

- **оптималното решение на проблемот содржи во себе оптимални решенија на проблемите од истиот тип содржани во главниот проблем**
- **потпроблемите имаат заеднички потпотпроблеми, односно делумно се преклопуваат**

Да ги провериме условите за примена на динамичкото програмирање. Нека е даден патот P чии што полиња имаат најголем збир. Тогаш секој дел од оптималниот пат го соединува почетното и крајното поле на тој дел на оптимален начин (инаку почетниот пат не би бил оптимален). Згодно е потпроблемите формално да ги зададеме на следниот начин: нека $P(i,j)$ е оптималниот пад од полето (1,1) до полето (i,j) и нека B(i,j) е збирот кој што се постигнува на тој пат. Тогаш се бара патот $P(M,N)$ и збирот $B(M,N)$.

Веќе заклучивме дека проблемот има оптимална потструктура. Можеме ли $B(M,N)$ да го изразиме рекурентно? До полињата (M,N) можеме да дојдеме само преку едно од полињата $(M,N$-1) или (M-1,N). Затоа збирот $B(M,N)$ е еднаков на поголемиот од вредноста $B(M,N-1)$ или вредноста $B(M-1,N)$ зголемена за $A(M,N)$ односно:

$$B(M,N) = max\{B(M,N-1), B(M-1,N\} + A(M,N)$$

Тогаш патот $P(M,N)$ се добива со додавање на полето (M,N) на оној пат $P(M-1,N)$ или $P(M,N-1)$ кој што дава поголем збир.

1,1 →	1,2	....	1,N
2,1	2,2	....	2,N
3,1	3,2 →	....	3,N
....	....	....	....
M,1	M,2 →	...	M,N

1,1			
			M-1,N
		M,N-1	M,N

Според тоа, решавањето на почетниот проблем се сведува на решавање на два проблема од ист тип и нивно едноставно комбинирање односно споредување на два збира и собирање на поголемиот од тие два збира со вредноста на последното поле. За решавање на сите нетривијални потпроблеми важи потполно истото.

Тривијалните проблеми се наоѓање на елементи од првиот ред и првата колона на матрицата B. Бидејќи до полињата (1,Y) или (X,1) постои само еден пат, треба само да се соберат полињата на тој пат.

Потпроблемите и кај овој проблем се поклопуваат. На пример проблемот за полето (M-1,N-1) се појавува и кај потпроблемот (M,N-1) и кај (M-1,N). Затоа за решавање на овој проблем никако не е прифатлива рекурзијата.

Повеќе потпроблеми со рекурзивно решение се сведуваат на два помали проблеми, па според тоа со рекурзија потребно е експоненцијално да се решат многу потпроблеми наместо само M·N, колку вкупно ги има различни. Затоа, по ред ќе ги решиме сите различни потпроблеми, т.е. ќе го користиме динамичкото програмирање.

Координатите на полињата преку кои минува оптималниот пат ги пронаоѓаме од последното кон првото поле. За да ги испишеме од првото кон последното, може да користиме стек, со помош на кој редоследот на податоците го вртиме наопаку.

Меѓутоа, при повикување на потпрограмите, оперативниот систем веќе користи стек за сместување на потребните податоци, па примената на рекурзијата е едноставен и природен начин за да се испишат координатите на полињата во саканиот редослед.

Забелешка: Решението е добиено со динамичко програмирање, а рекурзија се користи само за испишување на решението.

Потпрограмата *ispis*(M,N) која што го испишува патот до полето (M,N), се повикува рекурзивно најмногу онолку пати колку што има полиња на патот од полето (1,1) до полето (M,N), односно вкупно помалку од M+N пати, така што таа бргу се извршува.

```
var A,B:array[1..30,1..30]of integer;
i,j,M,N:integer;
 procedure ispisi(x,y:integer);
begin
     if(x>1)and(y>1)then
            begin
                 if B[x-1,y]>B[x,y-1] then ispisi(x-1,y)
                                      else ispisi(x,y-1);
                  writeln(x,',',y);
            end
               else
                  if(x=1) then
                              for i:=1 to y do writeln(x,',',i)
                          else
                                    for       i:=1       to       x       do
writeln(i,',',y);                                 end;
Begin
     writeln('M N');
     readln(M,N);
     for i:=1 to M do
         for j:=1 to n do
             begin
                  write('A[',i,',',j,']=');
                  readln(A[i,j]);
             end;
     writeln('Matricata A e ');
      for i:=1 to M do
      begin
         for j:=1 to n do
                  write(A[i,j]:3);
         writeln;
      end;
     B[1,1]:=A[1,1];
     for j:=2 to N do B[1,j]:=A[1,j]+B[1,j-1];
     for i:=2 to M do B[i,1]:=A[i,1]+B[i-1,j];
     for i:=2 to M do
         for j:=2 to N do
             if B[i,j-1]<B[i-1,j] then B[i,j]:=A[i,j]+B[i-1,j]
                                  else B[i,j]:=A[i,j]+B[i,j-1];
     writeln('Maksimalniot zbir e ',B[M,N],' a se postignuva vaka:');
     ispisi(M,N);
end.
```

Оваа задача може да се појави во повеќе сродни варијанти. Во секоја варијанта ќе подразбираме (ако на друг начин не се напомене), дека е потребно со движење низ матрицата да се постигне најголем збир и на секое поле да може да се застане само еден пат.

- Со движење надолу и десно да се стигне од полето (1,1) до полето (m,n). Ова е варијанта која тука се разгледува и решава.

- Со движење доле, десно и доле-десно се стигнува од полето (1,1) до полето (m,n). Во оваа варијанта (нетривијална) елементите од матрицата B се сметаат незначително на друг начин:

  $B(X,Y) = max\{B(X,Y\text{-}1),B(X\text{-}1,Y),B(X\text{-}1,Y\text{-}1)\} + A(X,Y).$

  Во склад со оваа измена, треба да ја измениме и постапката за реконструкција на решението како и самата програма.

- Со движење горе-десно, доле-десно и десно се стигнува од кое било поле (x_1,1) од првата колона до кое било поле (x_2,n) од последната колона.

Тука матрицата B мора да се пополнува по колони, за елементите од првата колона важи $B(X,1)=A(X,1)$, а за останатите

$$B(X,Y) = max\{B(X-1,Y-1),B(X,Y-1),B(X+1,Y-1)\} + A(X,Y)$$

при што сметаме дека $B(X-1,0)$ и $B(X-1,M+1)$ се еднакви на -¥, т.е. за првите и последните елементи во колоната на максимум се бира од 2, а не од 3 члена.

- Од едно поле на матрицата се преминува на следното со движење како шаховскиот коњ, но само во десно. Треба да се стигне од првата до последната колона. Задачата се решава слично како и претходните.

- Даден е триаголник пополнет со броевите: во првиот ред еден број, во вториот два броја, итн. до *N*-от ред со *N* броеви. Треба да стигнеме од бројот на врвот до бројот на основата на триаголникот. Триаголникот може да се смести во квадратна матрица на повеќе начини. Нека биде тоа триаголникот над споредната дијагонала. Така добиваме еквивалентна постапка: во квадратната матрица треба со движење надолу и десно да се стигне од горниот лев агол до кој било број на споредната дијагонала. Задачата се решава така што на ист начин како и порано. Формираме квадратна матрица B, но ја пополнуваме само до споредната дијагонала. Решението е одредено со најголемите елементи на споредната дијагонала на матрицата B.

- Целта е со движење надолу и десно да се стигне од горното лево до долното десно поле на матрицата A, при тоа да се застане на такви соседни броеви $x_1,x_2,\ldots,x_k$ во матрицата A ($x_1=a_{11}$, $x_k=a_{mn}$), така што ќе се максимизира не нивниот збир, туку збирот на апсолутните разлики на непрекинатите броеви преку кои се поминува, односно вредноста $\sum_{i=1}^{k-1}|x_{i+1} - x_i|$.

  Лесно се воочува дека за $X > 1$, $Y > 1$ важи релацијата

$$B(X,Y) = \max\begin{cases} B(X,Y-1) + |A(X,Y-1) - A(X,Y)| \\ B(X-1,Y) + |A(X-1,Y) - A(X,Y)| \end{cases}$$

  додека елементите од првата редица и првата колона се пресметуваат според формулата

$$B(1,1) = 0$$

$$B(1,Y) = B(1,Y-1) + |A(1,Y-1) - A(1,Y)|$$

$$B(X,1) = B(X-1,1) + |A(X-1,1) - A(X,1)|$$

  Врз основа на вредностите на пополнетата матрица B не е тешко да се одредат полињата низ кои се поминува поради максимизирање на бараниот збир.

- Треба да се стигне од кој било елемент на првата колона до кој било елемент на последната колона, движејќи се горе, доле и десно.

  Матрицата B ја пополнува по колони така што $B(X,Y)$ да е најголема вредност која може да се достигне со движење низ првите Y колони и застанување во полето $A(X,Y)$. Тогаш имаме:

$$B(X,1) = \max_{1\le k\le M}\left\{\sum_{p=k,X} a_{p,1}\right\}, \quad 1 \le X \le M$$

  каде p се менува од k до X , а за $k>X$ оди и наназад, и

$$B(X,Y) = \max_{1\le k\le M}\left\{B(k,Y-1) + \sum_{p=k..X} a_{p,Y}\right\},\quad 1 \le X \le M,\quad 2 \le Y \le N$$

Вкупниот број на операции во овој пример е поголем и е пропорционален со M^2N, а резултатот е одреден со максимумот на последната колона од матрицата B.

Освен наведените, може да се најдат и други слични варијанти на овој проблем. Сите тие се решаваат на начин многу близок на изложениот.

2) Најдолга заедничка подниза

Низата A е подниза на низата B, ако со прецртување на некои елементи од низата B може да се добие низата A. На пример (1,3,3,5) е подниза од (1,2,3,3,4,5), а не е подниза ниту од (1,5,2,3,3,4), ниту од (1,2,3,4,5).

Дадени се две низи: P од M и Q од N елементи. Да се најде низата со најголема можна должина која е подниза и за P и за Q.

Решение:

Нека $NZP(X,Y)$ е најдолгата заедничка подниза од низите P_X и Q_Y. Во задачата се бара $NZP(M,N)$. Ако е $p_M = q_N$, тогаш $NZP(M,N) = NZP(M\text{-}1,N\text{-}1)$È$\{p_M\}$, ($p_M$ се допишува на крајот од низата $NZP(M\text{-}1,N\text{-}1)$, додека за p_M¹q_N, $NZP(M,N)$ е еднаков на подолгиот пат од $NZP(M\text{-}1,N)$,$NZP(M,N\text{-}1)$.

Ова релација ни овозможува да ги пресметаме сите $NZP(X,Y)$ по ред (по редици или по колони), знаејќи дека е $NZP(X,0) = NZP(0,Y) =$ Æ (празна низа). Доволно е да се паметат должините на најдолгите заеднички поднизи во матрицата B, а $NZP(X,Y)$ лесно се реконструира на основа на матрицата B.

B(X,0) = B(0,Y) = 0

$$B(X,Y) = \begin{cases} B(X-1,Y-1)+1, & \text{ako e } P[X] = Q[Y] \\ \max\{B(X,Y-1), B(X-1,Y)\}, & \text{ako e } P[X] \ne Q[Y] \end{cases}$$

Доколку се бара само должината NZP, можеме да заштедиме простор, така што наместо матрицата B користиме само две низи, кои играат улога на редица од B која моментално се формира и претходната редица (со редослед на пополнување наназад, доволна е само една низа).

За да ја реконструираме NZP неопходна ни е целата матрица B, или дополнително време за повторно пресметување на изгубените информации.

```
var
            B:array[0..100,0..100] of integer
            p,q:array[1..100] of integer;
            i,j,m,n:integer;

procedure ispisi(i,j:integer);
begin
            if(i>0) and (j>0) then
                        if p[i]=q[j] then begin
                                    ispisi(i-1,j-1);
                                    write(p[i]:3);
                        end
            else if B[i-1,j]>B[i,j-1] then
                        ispisi(i-1,j)
            else
                        ispisi(i,j-1);
end;

begin
            {vnesuvanje na nizata P}
            write('m='); readln(m);
            for i:=1 to m do begin
                        write('p[',i,']= ');
                        readln(p[i]);
      end;

            {vnesuvanje na nizata Q}
            write('n='); readln(n);
            for j:=1 to n do begin
                        write('q[',j,']= ');
                        readln(q[j]);
      end;
            for i:=0 to m do B[i,0]:=0;
            for j:=0 to n do B[0,j]:=0;
            for i:=1 to m do
                        for j:=1 to n do
                                    if p[i]=q[j] then
                                                B[i,j]:=B[i-1,j-1]+1
                                    else if B[i-1,j] > B[i,j-1] then
```

Пример :

Нека се дадени низите P={1,3,4} и Q={2,1,5,4,3}.

Да се најде најдолгата подниза на P и на Q.

Решение :

M=3,N=5;

Ја пополнуваме матрицата B која е од ред (M+1)x(N+1)

Нултата редица и нултата колона ја полниме со нули.

Во контекст на програмското решение матрицата B ја полниме на следниот начин:

$P[1]=1 \neq Q[1]=2 \Rightarrow B(1,1)=\max\{B(1,0),B(0,1)\}=0$

P[1]=1=Q[2]=1 => B(1,2)=B(0,1)+1=1

0	0	0	0	0	0
0	0	1			
0					
0					

0	0	0	0	0	0
0	0				
0					
0					

P[1]=1≠ Q[3]=5 => B(1,3)=max{B(1,2),B(0,3)}=1 …

P[2]=3≠ Q[1]=2 => B(2,1)=B(2,0)=0 …

0	0	0	0	0	0
0	0	1	1		
0					
0					

0	0	0	0	0	0
0	0	1	1	1	1
0	0				
0					

P[2]=3≠Q[2]=1 =>B(2,2)=max{B(1,2),B(2,1)}=1...

P[2]=3= Q[5]=3 => B(2,5)=B(1,4)+1=2

0	0	0	0	0	0
0	0	1	1	1	1
0	0	1	1	1	2
0					

0	0	0	0	0	0
0	0	1	1	1	1
0	0	1			
0					

P[3]=4≠ Q[1]=2 => B(3,1)=max{B(2,1),B(3,0)}=0…

P[3]=4 = Q[4]=4 => B(3,4)=B(2,3)+1=2 ...

0	0	0	0	0	0
0	0	1	1	1	1
0	0	1	1	1	2
0	0				

0	0	0	0	0	0
0	0	1	1	1	1
0	0	1	1	1	2
0	0	1	2	2	2

Изгледот на поднизата се наоѓа со помош на матрицата B почнувајќи од последното поле. Се испитуваат условите зададени во процедурата ispisi(i,j) за да се увиди во која насока да треба да се продолжи со повикувањето на процедурата.

ispisi(i,j); i=3,j=5;

P[3]=4 $\neq$ Q[5]=4=>од не (B(2,5)=2>B(3,4)=2) => ispisi(3,4)

P[3]=4 = Q[4]=4=> ispisi(2,3) печати P[3]

P[2]=3 $\neq$ Q[3]=5=>од не (B(1,3)=1>B(2,2)=1) => ispisi(2,2)

P[2]=3 $\neq$ Q[2]=1=>од да (B(1,2)=1>B(2,1)=0) => ispisi(1,2)

P[1]=1 = Q[2]=1=> ispisi(0,1) печати P[1]

Завршивме затоа што i=0

Значи низата што се добива е {1,4}

3) Најефтина исправка на зборови

Дозволени операции над стрингот се: вметнување букви, бришење на една буква, измена на буквите и бришење на сите букви до крајот на стрингот. Секоја од овие операции има зададена цена. Потребно е да се одреди најмалата вкупна цена на операциите со кои од даден стринг A се добива дадениот стринг B.

Решение:

Задачата се решава многу слично како проблемот на максималниот збир и проблемот за најдолга заедничка подниза, па затоа деталното објаснување е изоставено.

```
var
            cena: array[0..50,0..50] of real;
            a,b: string;
            cVmet, cBris1, cZam, cBrisDoKraj:real;
            i,j,k,m,n:integer;

function min3(a,b,c:real);real;
var x:real;
begin
            x:=a; if x>b then x:=b; if x>c then x:=c; min3:=x;
end;
begin
            write('Cena na vmetnuvanja='); readln(cVmet);
            write('Cena na brisenje na eden znak='); readln(cBris1);
            write('Cena na zamena='); readln(cZam);
            write('Cena na brisenje do kraj na stringot='); readln(cBrisdoKraj);
            write('a= '); readln(a); m:=length(a);
            write('b= '); readln(b); n:=length(b);
            cena[0,0]:=0;
            for i:=1 to m do begin
                        cena[i,0]:=cena[i-1,0]+cBris1;
                        if cena;i,0]>cBrisDoKraj then
                                    cena[i,0]:=cBrisDoKraj;
      end;
      for j:=1 to n do
                        cena[0,j]:=cena[0,j-1]+cVmet;
            for i:=1 to m do
                        for j:=1 to n do begin
                              if a[i] = b[j] then
                                                cena[i,j]:=cena[i-1,j-1]
                              else
                                          cena[i,j]:=min3(cena[i-1,j-1]+cZam,
                                                            cena[i,j-1]  + cVmet,cena[i-1,j]
+ cBris1);
                  for k:=1 to i-1 do
                                    if cena[i,j]>cena[k,j]+cBrisDoKraj then
                                                cena[i,j]:=cena[k,j]+cBrisDpKraj;
                  end;
            writeln('najeftinata ispravka cini',cena[m,n]:10:3);
end.
```

4) Максимална сума на несоседни во низа

Дадена е низата A од N цели броеви. Да се одреди поднизата A чиј збир на елементите е максимален, а во кој нема соседни елементи на низата A. Да се смета дека празната низа има збир на елементите нула.

Решение:

Нека за низата A е позната една таква подница $P=P(A)$. Ако елементот a_N не припаѓа на поднизата P, тогаш P е оптимална подница и за низата A_{N-1}. Ако елементот a_N припаѓа на поднизата P, тогаш поднизата P е без последен елемент (т.е. без a_N). Во спротивно би било можно да се подобри оптималната подниза P на низата A така што на подобрата подниза на низата A_{N-2} ќе се допише бројот a_N. Ја установивме оптималноста на подструктурата на проблемот.

Решенијата за низите A_0 и A_1 се тривијални. За X од 2 до N, $P(A_x)$ е оној елемент од низите $P(A_{x-1})$ и $P(A_{x-2})$È$\{a_x\}$, кој што има поголем збир.

Ќе формирање низа B, така што е $B(X)$ збир на елементите на оптималната подниза $P(A_x)$. Според претходното следува

$B(0) = 0$, $B(1) = max(0,a_1)$, $B(X) = max\{B(X\text{-}1), B(X\text{-}2)+a_x\}$, X=2,N.

На основа на низата *B* едноставно се наоѓа поднизата *P*.

```
var
            a,b:array[0..1000] of integer;
            n,i:integer;

procedure ispis(n:integer);
begin
            if n>0 then
                        if b[n]=b[n-1] then
                                    ispis(n-1)
                        else begin
                                    ispis(n-2);
                                    write(a[n]:4);
                        end;
end;

begin
            readln(n);
            for i:=1 to n do readln(a[i]);
            b[0]:=0;
            if a[1]>0 then b[1]:=a[1] else b[1]:=0;

            for i:=2 to n do
                  if b[i-2]+a[i]>b[i-1] then
                                    b[i]:=b[i-2]+a[i]
                  else
                                    b[i]:=b[i-1];

            ispis(n); readln;
end.
```

Пример:

Нека е дадена низата A со n=6 елементи.A={4,5,6,1,8,9}

Решение:

Ја пополнуваме низата B со помош на формулите зададени претходно

```
B[0]=0; B[1]=A[1]=4;

B[2]=max{B[0]+A[2] , B[1]} =max{5,4}=5

B[3]= max{B[1]+A[3],B[2]}=max{10,5}=10

B[4]=max{B[2]+A[4],B[3]}=max{6,10}=10

B[5]=max{B[3]+A[5],B[4]}=max{18,10}=18

B[6]=max{B[4]+A[6],B[5]}=max{19,18}=19
```

За да ја определиме поднизата на низата A почнуваме од крајот на низата B одејќи на напред проверувајќи ги условите во процедурата ispisi(n);

```
Ispis(6); n=6;

B[6]=19 ≠ B[5]=18 => ispis (6-2) , печати A[6]=9

B[4]=10 = B[3]=10 => ispis (4-1)

B[3] =10  ≠ B[2] =5 => ispis (3-2), печати A[3]=6

B[1]=4  ≠  B[0]=0 =>   ispis(1-2), печати A[1]=4
```

Завршивме затоа што се доби n<0

Поднизата со максимален збир на дадената низа А е {4,6,9}

Паскал vs. C++

Краток историјат

Паскал е еден од постарите програмски јазици и според некои луѓе еден од “најпопуларните”.

Паскал е креиран од страна на швајцарскиот професор Никлаус Вирд во седумдесетите години, а го именувал во чест на фрацускиот математичар и физичар Блеиз Паскал, кој пак ја има конструирано првата машина за собирање.

Во 1980 година е конструиран објектниот јазик C наречен “C со класа“, кој работи на многу ниско ниво и е наменет за креирање на оперативни системи и компајлери. Токму на таа основа во 1983 година се раѓа C++ варијанта на C со една голема разлика во својата стабилност во работата.

Како за почеток најголемата разлика помеѓу паскал и C++ е тоа што паскал е структурен (некаде се дефинира како процедурален) јазик а C++ е објектно-ориентиран јазик. Паскал ја доживува својата експанзија токму поради тоа што тој бил првиот јазик кој овозможува структурно програмирање.

Краток опис

Структурното програмирање (прецедуралното програмирање) на паскал му диктира пат на работа со модули, кои пак се посебни независни делови а секој од нив реализира различна функција. Модулот мора да им оперделен почеток и крај. Модулите можат да се менуваат и тестираат независно еден од друг а се повикуваат според пропишана редослед. Недостатокот на процедуралното програмирање е тоа што фукциите и податоците се раздвоени.

Објектно-ориентираното програмирање е пак од друга страна по дефиниција значи: манипулација со објекти низ програмирањето. Па токму оваа дефиниција го носи терминот објектно-ориентирано програмирање. Кај ООЈ имплементиран е апстрактен тип на податоци преку концептот наречен “КЛАСИ“. Таа апстракција всушност е комбинирање на податоците со функциите, со користење на класи. Огромна предност пред структурните јазици е тоа што може да се издвојат делови од програмите како објекти и да се користат во други програми, било да се надополнуваат или не. ОО-јазиците овозможуваат и лесна модификација на постоечките програми без ништо да се менува. Ова се прави со помош на:

- Наследување и
- Полиморфизам

“Преклопувањето на оператори“ е уште една од многуте добри собини на ОО-јазиците.

<u>СЕ ШТО МОЖЕ ДА СЕ НАПРАВИ ВО ПАСКАЛ МОЖЕ ДА СЕ НАПРАВИ И ВО C++, ОБРАТНОТО ТВРДЕЊЕ НЕ ВАЖИ.</u>

Почетни синтаксички разлики

Како прво и основно паскал не е “case-sensitive“ додека C++ е “case-sensitive“.

Кога разгледуваме програма во паскал и програма направена во C++ уште на самиот почеток имаме видливи синтаксички разлики како што се заглавје на програми, дефинирање на променливи, функции и процедури наспроти класи, запишување коментари, структурни наредби, наредби за печатење, циклуси за повторување, аритметички операции итн. Сега со мала демонстрација на програмски код и од двата програмски јазици ќе ја утврдиме разликата.

На самиот врв на програмата во паксал дефинираме име на програмата, од друга страна во C++ тоа не е потребно тоа се изведува на поинаков начин што ќе го покажеме со следниов мал и наједноставен пример:

Паскал код:

```
program PechatenjeIme
 begin
   writeln (' Marjan ');
   {Запишување на коментар}
 end.
```

C++ код:

```
#include
 int main()
 {
   cout << "Marjan" << endl;
   // запишување на коментар
 }
```

Двата кода извршуваат потполно иста работа разликата е само во синтаксата (овие две програмчиња кога би се извршиле на монитор ќе се испише Marjan). Да ги воочиме разликите кај паскал наредбите се пишуваат помеѓу

```
begin
    ;

 end;
```

а кај C++ тоа се прави помеѓу

```
{
    ;
    ;
 }
```

Симболот “;“ е задолжителен после секоја наредба освен пред end во паскал, а во C++ не смее никаде да се изостави.

Аритметички операции

Паскал код:

```
:=    { симбол за доделување на вредност }
+
-
*
/     { "real" делење }
div   { "integer" делење }
mod   { делење по модул}
```

C++ код:

```
=  // симбол за доделување на вредност
+
-
*
/
%  // делење по модул
```

Релациски оператори

Паскал код:

```
=    { еднакво }
   { различно }
<
<=
>
>=
```

C++ код:

```
==   // еднакво
!=   // различно
<
<=
>
```

Логички оператори

Паскал код:

```
and {и}
or  {или}
not {не}
```

C++ код:

```
&& //и
|| //или
!  //не
```

Декларација на променливи, константи и низи: Типови на променливи

Паскал код:

```
·         char
·         integer
·         real
·         boolean
```

C++ код:

```
·         char
·         int
·         float
·         bool
```

Константи:

Паскал код:

```
const
   Pi = 3.14;
   Rate = 0.05;
      { .05 не е дозволува}
   Chash = 3600;
   Denar = 'den';
   Pozdrav  = 'Zdravo';
```

C++ код:

```
 const double Pi =
3.14,
       Rate = .05; //
or 0.05;
 const int Chas = 3600;
 const char Denar =
'den';
 const char Pozdrav[] =
"Zdravo";
 //  Исто така се
дозволува и употреба на
следниов начин
 const double R = 5.,
Pi = 3.14,
    Area = Pi * R * R;
```

Променливи:

Кај паскал променливите и во главната програма и во функциите и процедурите променливите мора да се наведени по пишување на зборот var. Не е дозволено иницијализирање на променливите во нивната декларација.

Паскал код:

```
...
   var
     r : real;
     i, j : integer;
     star : char;
     match : boolean;
...
```

C++ код:

```
  double r = 5.;
   int i = 0, j = i+1;
   char star = '*';
```

Во C++ декларирањето на променливи може да се направи било каде во кодот и тие може да се иницијализираат при нивната декларација. Глобалните променливи може да се декларираат надвор од секоја класа.

Низи

Паскал код:

```
  var
   str : packed array [1..80] of char;
   grid : array [1..32, 1..25] of integer;
```

Резервираниот збор packed е препорачлив за користење на текстуални низи (но ева е отфрлено во пракса бидејќи денешните компјутери располагаат со доволно голема меморија). Првиот член во низата може да почне од било кој број но вообичаен индекс е еден (1).

C++ код:

```
 char str[80];
  int grid[32][25];
```

Исто така низите во C++ можат да бидат иницијализирани при нивната декларација:

```
  int fiboBroevi[5] = {1,1,2,3,5};
  char tekst[80] = "Значи, да а";
```

Во C++ индексот на првиот елемент е нула (0), а индексот на последниот елемент е n-1.

Операторот sizeof(. . .)

Паскал код:

```
Во паскал не постои такво нешто
```

C++ код:

```
Неговата улога е да врати големина во бајти на константи, променливи на пример
sizeof(char)
врќа вредност еден (1).
```

Множества (SETS)

Паскал код:

Пример:

```
if ch in ['0'..'9'] then
   writeln (ch, ' e broj.');
 if ch in ['A'..'Z', 'a'..'z']
  then
    writeln (ch, ' e bukva.');
     ...
```

Компајлерот може да ја лимитира големината на множеството, вообичаено е големината да биде 256.

C++ код:

Не постои такво нешто во C++.

Готови математички функции

Паскал код:

```
 abs(x)
 sqrt(x)
 sin(x)
 cos(x)
 exp(x)
 ln(x)
 sqr(x)
 arctan(x)
```

C++ код:

```
int abs(int x);
 double fabs(double x);
 double sqrt(double x);
 double sin(double x);
 double cos(double x);
 double exp(double x);
 double log(double x); // Природен log
 double pow(double base, double exponent);
 double atan(double x);
```

Комбинирани аритметички операции (Инкремент, Декремент)

Паскал код:

Во паскал не постои такво нешто, во паскал овие работи можат да се решат на следниот начин:

```
a := a + 1;
b := a; a := a + 1;
a = :a + 1; b := a;
a := a - 1;
b := a; a := a - 1;
a := a - 1; b := a;
a := a + b;
a := a - b;
a := a * b;
a := a / b;
a := a mod b;
```

C++ код:

Соодветните од оние кај паскал во C++ би ги запишале вака:

```
a++;      //  a := a + 1
b = a++; //  b := a; a := a + 1
b = ++a; //  a := a + 1; b := a
a--;      //  a := a - 1;
b = a--; //  b := a; a := a - 1
b = --a; //  a := a - 1; b := a
a += b;  //  a := a + b
a -= b;  //  a := a - b
a *= b;  //  a := a * b
a /= b;  //  a := a / b
a %= b;  //  a := a % b
```

Влез и Излез

Паскал код:

```
write (x, y, ...);
writeln (x, y);
writeln;
read (x, y, ...);
readln (x, y);
readln;
```

C++ код:

```
cout << x << ' ' << y << ...;
cout << x << ' ' << y << endl;
cout << endl;
cin >> x >> y >> ...;
```

```
cin >> x >> y;
```

Излезни карактери:

Паскал код:

Во паскал не постојат “излезни карактери“ се што се пишува за печатење се пишува помеѓу два наводника

```
writeln ('Prim''er!');
```

C++ код:

Во C++ постојат “излезни карактери“ тие се пишуваат на следниов начин и нивната акција е следна:

```
'\n'  Нова Линија
'\''  Единечен Наводник
'\"'  Наводница
'\\'  Коса црта
'\a'  Аларм
'\t'  Таб
'\r'  Враќање назад
```

Наредби за разгранување(if-else, case - switch)

Паскал код:

```
if  then
   ;
 if  then
    {употреба на ; не е дозволено}
 else
   ;
 if  then begin
   ;
   ;
 end
 else begin
   ;
   ;
 end;
```

C++ код:

```
if (  )
   ;
 if (  )
   ; // задолжителна уптреба на ;
 else
   ;
 if (  ) {
   ;
   ;
 }
 else {
   ;
```

```
   ;
 }
```

case – switch

Паскал код:

```
case  of
   :
     ;
   ...
   :
     ;
 end;
```

Примери:

```
 case ch of
   'A': Add();
   'D': Delete();
   'M': Modify();
   'Q': ; { do nothing }
 end;
 case d of
   1,2: ...;
    99:   ...;
   100:   ...;
 end;
 case age >= 65 of
   true: ...;
   false: ...;
 end;
```

C++ код:

```
switch (  ) {
   case :
     ; break;
   ...
   case :
     ; break;
   default:  // optional
     ; break;
 }
```

Наредби за повторување (итеративни наредби)

Паскал код:

```
   { while }
 while  do begin
   ;
   ...
 end;
  { for }
 for i := n1 to n2 do begin
```

```
  ...      { зголемуваме за еден }
end;
for i := n2 downto n1 do begin
  ...      {намалуваме за еден }
for ch := ltr1 to ltr2 do
  ...
  { repeat - until }
repeat
  ;
  ...
until ;
```

Кај наредбата repeat, наредбите во циклусот се извршуаат барем еднаш бидејќи излезот е на крајот од циклусот.

C++ код:

```
 // while
while (  ) {
  ;
  ...
}
 // for
for ( ;; )
{
  ...
}
```

Пример за for:

```
for (i = 0;   i < n;   i++)
  ...
 // do - while
do {
  ;
  ...
} while ( );
```

Наредби за прекин (*break* и *continue*)

Паскал код:

Во паскал не постојат наредба за прекин или пак за продолжување.

C++ код:

```
for (i = 0;   i < n;   i++) {
   if (a[i] < 0) break;
     // Излез од for-наредбата
   if (a[i] == 0) continue;
     // Продолжи со следното i
   // Ако се наоѓаме тука тогаш a[i] е >0
   b = a[i];
     ...
 }
```

Процедури и Функции

Паскал код:

```
procedure Primer
     (n : integer; ch : char);
 ...
   begin
     ...
   end;
 function PrimerFunkcija(m, n : integer) : real;
 ...
   begin
     ...
     PrimerFunkcija:= ;
   end;
```

```
Процедурите и функциите примаат аргументи. Процедурите не враќаат вредност, наспроти функциите кои
 враќаат вредност на дефиниран тип на податок т.е вредност на аргументот.
```

C++ код:

```
void Primer (int n, char ch)
 {
     ...
 }
 double Primer1(int m, int n)
 {
     ...
     return ;
 }
```

Во C++ не постојат процедури, се е функција. функциите примаат аргументи и враќаат вредности. Функциите кои не враќаат експлицитна вредност се дефинираат како void функции.

Функциите кои не се void враќаат вредност со помош на наредбата return, можно е повеќекратно враќање на вредности со тоа што ставаме услов за разгранување if.

Место на процедурите и функциите во програмот

Паскал код:

```
program ...
 ...
 procedure Primer (...);
 ...
   begin
     ...
   end;
 ...
 begin { main }
   ...
   Primer(...);
   ...
 end.
```

Во Паскал вообичаено е дефинирањето на функциите и процедурите да биде пред тие да бидат повикани. Дозволено е вгнездување на функциите и процедурите.

C++ код:

```
// Function prototype:
 double MyFunc(int arg1, int arg2);
 int main()
 {
   double x;
   ...
   x = MyFunc(1999, 3);
   ...
 }
 ...
 // Function definition:
 double MyFunc(int arg1, int arg2)
 {
   ...
 }
```

Функцијата мора да биде декларирана пред првото нејзино повикување, но можеме телото или кодот кој таа ќе го извршува да биде напишан по нејзиното повикување. Во C++ вгнездените функции не се дозволени.

Предавање на параметри (аргументи) според адреса

Паскал код:

```
procedure Primer (var x, y : integer);

procedure Primer1 (a, b, c : real; var x1, x2 : real);
```

C++ код:

```
void Primer (int &x, int &y);
 void Primer1 (double a,double b, double c, double &x1, double &x2);
```

Во паскал за да го направиме ова потребно ни е користење на резервираниот збор var, додека тоа кај C++ се прави со симболот “&“.

Покажувачи

Еден прост пример за креирање, бришење на покажувач како и негово печатење т.е на вредноста на која што покажува.

Паскал код:

```
type
   RealArray = array [1..100] of real;
 var
   i : integer;
   pi1, pi2 : ^integer; { pointers to integer }
   pa : ^RealArray;     { pointer to an array of reals }
 begin
   i := 99;
   new (pi1);  { allocates one integer }
   if pi1 = nil then
     writeln ('Greshka vo memoriska alokacija');
   pi1^ := i;
   pi2 := pi1;
   writeln (pi2^);   { output: 99 }
   dispose (pi1);
   new (pa);  { allocate an array of RealArray type }
   pa^[1] := 1.23;
   pa^[2] := pa^[1];
   writeln (pa^[2]); { output: 1.23 }
   dispose (pa);
 end.
```

C++ код:

```
int main()
 {
   int i, *pi1, *pi2; // pi1, pi2 are pointers to int.
   double *pa;        // pointer to a double
   i = 99;
   pi1 = new int;
   if (!pi1)      // или со други зборови if (pi == 0)
     cout << " Greshka vo memoriska alokacija " << endl;
   *pi1 = i;
   pi2 = pi1;
   cout << *pi2 << endl;  // Излез: 99
   delete pi1;
   pa = new double[100]; // allocate an array of 100 doubles
   pa[0] = 1.23;
   pa[1] = pa[0];
   cout << pa[1] << endl; // Излез: 1.23
   delete [] pa;
   return 0;
 }
```

Во C++ со користење на операторот & покажувачот можеме да го наместиме да покажува на променлива која е од истиот тип како и покажувачот.

Пример:

```
int a, *pa = &a;  // Покажувачот pa е поставен на адресата на a.
```

Покажувачи и низи

Паскал код:

Во паскал нема директна врска помеѓу покажувачите и низите. Најчесто покажувачите се користат со поврзани листи, дрва и други поврзани структури.

C++ код:

Во C++ постои силна врска помеѓу покажувачите и низите. Пример:

```
int a[100];

a[0] е исто со *a.

a[i] е исто што и *(a+i).
```

Во употребата со поврзани листи и други поврзани структури операторот “new” подржува алокација на низи со променлива должина.

Пример:

```
int n;
 cin >> n; // Enter n
 // Allocate an array of n integers:
 int *a = new int[n];
 a[0] = ...;
```

Адресни променливи (Reference Variables)

Паскал код:

Во паскал не постои такво нешто.

C++ код:

Во суштина тие се скоро исти со покажувачите меѓутоа користат различна синтакса.

Пример:

```
int main()
 {
   int i = 1, &ri = i;
   // ri becomes an alias for i
   i = 99;
   cout << ri << endl;
   // Излез: 99
   ...
 }
```

Адресните променливи се користат најчесто за предавање на аргументи на функции по адреса и за да вратат вредности од тие функции или на некој “пренаполнети оператори“ (overloaded operators).

Класи

Во C++ класите ги комбинираат елементите и поврзаните функции во еден ентитет. Класите се многу згодни за имплементација и главниот чекор до ОО-програмирање. Кај нив ги среќаваме термините како што се “private“, “public“,“constructor“,“destructor“, “полиморфизам” и слично.