# 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 меморски 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 првиот 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 инцијализација 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.Притоа 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 a 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;
}
}
```

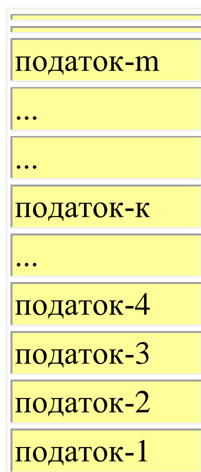
Како што можеме да забележиме низите се од голема корист, но зошто да не се употребат за било каков податочен тип? Во случај кога компонентите во низата не се сортирани, можеме да сместуваме компоненти во O (1) време, но времето потребно за пребарување е O (n).

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 пополнети. 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.

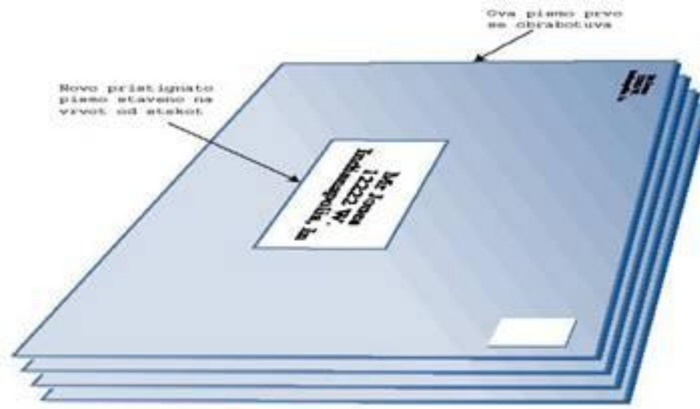
Купот исто така може да се дефинира како структура која зафаќа посебна меморија со одредена големина. Доколку капацитетот на купот е n, тогаш во него може да се сместат само n податоци.

Со купот се извршуваат следните основни операции:

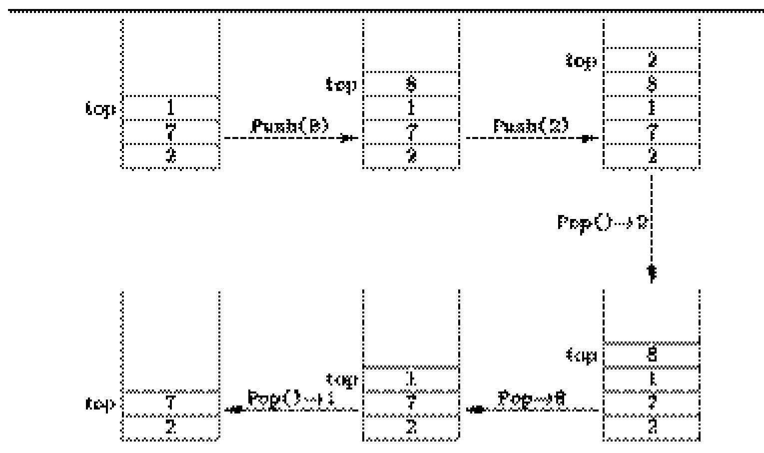
- иницијализација на празен куп
- проверка дали е купот полн
- ставање податок во купот
- проверка дали е купот празен
- земање податок од куп

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

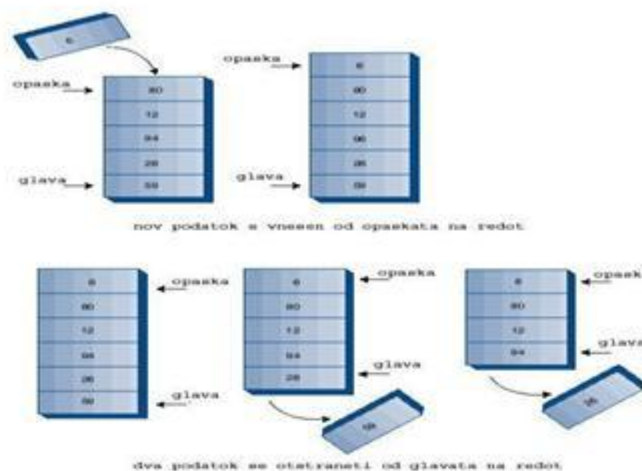
Ред (анг. gueue) е линеарна листа во која елементите се ставаат на крајот, а се земаат од почетокот на редот. Затоа велиме дека филозофијата на работа на редот е " кој прв влегува, прв излегува" (анг. Frst In Frst Out-FIFO). Редот има два покажувачи едниот покажува на почетокот на редот, а другиот на 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

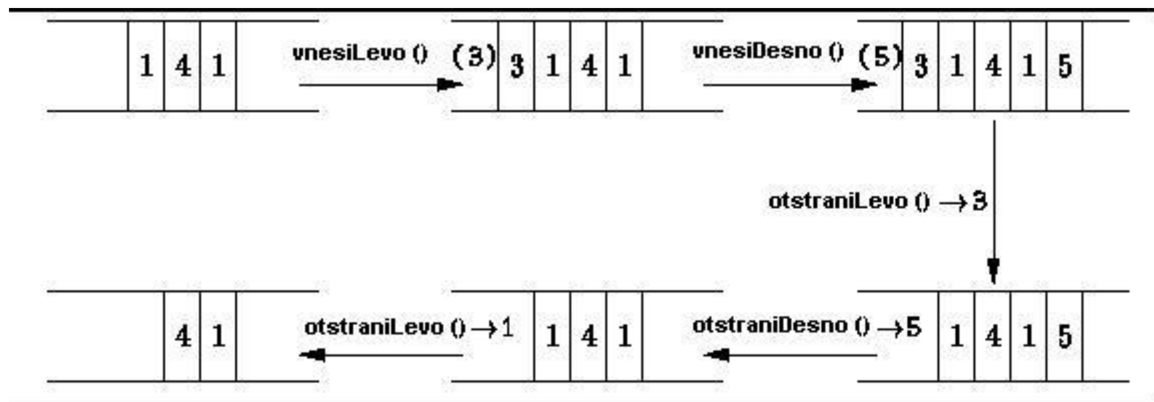
# Шпил

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