PROIECTAREA ALGORITMILOR

Adrian Runceanu

Curs 3

Alocarea dinamică de memorie în C++ (partea II)

Conţinutul cursului

1. Pointeri la funcţii
2. Folosirea pointerilor şi tablourilor ca parametri în funcţii
3. Alocarea dinamică a memoriei
    3.1. Operatorul new
    3.2. Operatorul delete
4. Structuri implementate dinamic:
    4.1. Stiva
    4.2. Coada
    4.3. Lista simplu înlănţuită
    4.4. Lista dublu înlănţuită

1. *Pointeri la funcţii*

Numele unei funcţii este un pointer spre funcţia respectivă.

✓El poate fi folosit ca parametru efectiv la apeluri de funcţii.

✓În felul acesta, o funcţie poate transfera funcţiei apelate un pointer spre o funcţie.

✓Aceasta, la rândul ei, poate apela funcţia care i-a fost transferată în acest fel.

1. *Pointeri la funcţii*

În instrucţiunile în care se atribuie pointeri la funcţii, tipurile de funcţii trebuie să corespundă exact.

Sintaxa de declarare este:

tip functie(*pointer_functie)(lista_param);

Exemple:

1. Se observă importanţa *folosirii parantezelor rotunde pentru pointeri la funcţii*.

Fără acestea nu s-ar mai considera un pointer la funcţie ci o funcţie care întoarce un pointer de un anumit tip.

a) Fie declaraţia

int *s(int *s1, int *s2);

- ✓ În acest caz avem o funcţie care *întoarce pointer la int.*

b) În declaraţia

int (*s)(int *s1, int *s2);

avem un *pointer la o funcţie care întoarce o valoare de tip int*.

c) ***int (*f (int,int)) [10];***

✓ În acest caz se defineşte o variabilă de tipul unei *funcţii cu doi parametri de tip int care întoarce ca valoare un pointer la un tablou de 10 întregi*.

2. Schemă de program care foloseşte un tablou de pointeri la funcţii predefinite.

```
// tablou de pointeri la funcţii predefinite
#include<iostream>
#include<math.h>
using namespace std;
double(*func_mat[ ])(double) = { sin, sinh, cos, cosh, tan,
    atan };
int main()
{
    int i;
    double x;
    for (x=0.01; x<1.01; x+=0.01)
        for(i=0; i<6; i++)
                cout<<(*func_mat[i])(x)<<"   ";
}
```

2. Schemă de program care foloseşte un tablou de pointeri la funcţii predefinite.

Online C++ Compiler IDE

```
1  #include<iostream>
2  #include<math.h>
3  using namespace std;
4  double(*func_mat[ ])(double) = { sin, sinh, cos, cosh, tan, atan };
5  int main()
6  {
7      int i;
8      double x;
9      for (x=0.01; x<1.01; x+=0.01)
10         for(i=0; i<6; i++)
11             cout<<(*func_mat[i])(x)<<"    ";
12 }
13
```

Execute Mode, Version, Inputs & Arguments

Result

CPU Time: 0.00 sec(s), Memory: 4132 kilobyte(s)

```
0.00999983   0.0100002   0.99995   1.00005   0.0100003   0.00999967   0.0199987   0.0200013   0.9998   1.0002   0.0200027   0.0199973
```

3) În continuare, un exemplu de program care foloseşte un tablou de pointeri la funcţii definite:

```
#include<iostream>
using namespace std;
typedef int (*functie)(); // Pointer la funcţie de tip
    întreg
void tiparire()
{
    cout<<"\nFuncţia de tiparire";
}
void afisare()
{
    cout<<"\nFuncţia de afisare";
}
```

```
void scriere(){
cout<<"\nFuncţia de scriere";
}
void verificare(){
cout<<"\nFuncţia de verificare";
}
/* declaratia tabloului de pointeri la funcţii */
functie tf[ ] = { "scriere", "verificare", "afisare",
    "tiparire" };
int main(){
    int i;
    for(i=0; i<4; i++)
/* Apelul unei funcţii din tabloul de pointeri la funcţii */
(*tf[i])();
}
```

```
1   #include<iostream>
2   using namespace std;
3   typedef int (*functie)(); // Pointer la funcție de tip întreg
4   void tiparire()
5   {
6       cout<<"\nFuncția de tiparire";
7   }
8
9   void afisare()
10  {
11      cout<<"\nFuncția de afisare";
12  }
13  void scriere()
14  {
15      cout<<"\nFuncția de scriere";
16  }
17  void verificare()
18  {
19      cout<<"\nFuncția de verificare";
20  }
21  /* declaratia tabloului de pointeri la funcții */
22  functie tf[ ] = { "scriere", "verificare", "afisare", "tiparire" };
23  int main()
24  {
25      int i;
26      for(i=0; i<4; i++)
27  /* Apelul unei funcții din tabloul de pointeri la funcții */
28          (*tf[i])();
29  }
```

Conţinutul cursului

1. **Pointeri la funcţii**
2. **Folosirea pointerilor şi tablourilor ca parametri în funcţii**
3. **Alocarea dinamică a memoriei**
   - **3.1. Operatorul new**
   - **3.2. Operatorul delete**
4. **Structuri implementate dinamic:**
   - **4.1. Stiva**
   - **4.2. Coada**
   - **4.3. Lista simplu înlănţuită**
   - **4.4. Lista dublu înlănţuită**

2. Folosirea pointerilor şi tablourilor ca parametri în funcţii

- ✓ În lista de parametri a unei funcţii pot apărea atât parametri de tip tablou cât şi pointeri la un tip de dată.
- ✓ ***În cazul tablourilor, nu se încarcă în stivă tot conţinutul tablourilor ci numai adresa primului element.***
- ✓ *Este recomandată utilizarea ca parametri a pointerilor în locul tablourilor.*
- ✓ Pentru a vedea mai clar, vom defini aceeaşi funcţie folosind pe rând tablouri şi apoi pointeri.

Exemple:

1. Funcţia ***lungime*** calculează lungimea unui şir de caractere.

Varianta din stânga defineşte funcţia prin utilizarea tabloului, varianta din dreapta defineşte funcţia prin utilizarea pointerilor.

```
int lungime (char s[ ])
{
   int i;
   for(i=0; s[i]; i++);
   return i;
}
```

```
int lungime(char *s)
{
   char *p;
   for(p=s; *p; p++);
   return p-s;
}
```

2. Următoarea funcţie face *transferul datelor între două zone de memorie*.

Funcţia copiere are ca parametri:

- dest = adresa zonei destinaţie
- sursa = adresa zonei sursă
- nr = numărul de octeţi care vor fi copiaţi

```
void copiere(void *dest, void *sursă, int nr)
{
   while (nr--)
        *((char*) dest)++ = *((char*)sursa)++ ;
}
```

- ✓ Pointerii vor fi convertiţi la *char* pentru a face accesul la nivel de octet.
- ✓ La fiecare secvenţă de ciclare se transferă un octet şi se micşorează numărul de octeţi transferaţi cu 1.
- ✓ În momentul în care *nr* devine 0 se iese din ciclul de transfer.

3. Produsul elementelor de pe diagonala principală a unei matrici de tip *int*:

Pointerul p este iniţializat cu adresa de început a matricei

```
long produs(void *a, int n)
{
    int *p=(int*)a,m;
    long k=1;
    for(m=n; m--; k=k * (*p), p = p + n+1) ;
    return k;
}
```

Variabila k este înmulţită pe rând cu elementele de la adresa lui p

Prin adunarea cu valoarea n+1 se face mereu poziţionarea pe următorul element de pe diagonala principală

Conţinutul cursului

1. Pointeri la funcţii
2. Folosirea pointerilor şi tablourilor ca parametri în funcţii
3. Alocarea dinamică a memoriei
   - 3.1. Operatorul new
   - 3.2. Operatorul delete
4. Structuri implementate dinamic:
   - 4.1. Stiva
   - 4.2. Coada
   - 4.3. Lista simplu înlănţuită
   - 4.4. Lista dublu înlănţuită

3. Alocarea dinamică a memoriei

- ✓ Odată cu gestiunea pointerilor apare şi posibilitatea utilizării variabilelor dinamice.
- ✓ Spre deosebire de variabilele statice, aşa cum sugerează şi denumirea, *variabilele dinamice sunt variabile care sunt create şi eliminate la cererea programatorului şi a căror dimensiune se poate modifica pe parcursul execuţiei programului*.
- ✓ Zona de memorie în care se face alocarea dinamică a variabilelor se numeşte ***heap***.

3. Alocarea dinamică a memoriei

- ✓ *Alocarea de zone de memorie şi eliberarea lor în timpul execuţiei programelor* permite gestionarea optimă a memoriei de către programe.

- ✓ Un astfel de mijloc de gestionare a memoriei se numeşte ***alocare dinamică a memoriei***.

3. Alocarea dinamică a memoriei

Alocarea dinamica a memoriei se poate executa prin utilizarea a doi operatori ai limbajului C++:

1. Operatorul de alocare a memoriei - **new**
2. Operatorul de dezalocare(eliberare) a memoriei - **delete**

Conţinutul cursului

1. Pointeri la funcţii
2. Folosirea pointerilor şi tablourilor ca parametri în funcţii
3. Alocarea dinamică a memoriei
    3.1. Operatorul new
    3.2. Operatorul delete
4. Structuri implementate dinamic:
    4.1. Stiva
    4.2. Coada
    4.3. Lista simplu înlănţuită
    4.4. Lista dublu înlănţuită

3.1. Operatorul new

- ✓ Limbajul C++ permite alocări în zona heap prin intermediul operatorului ***new***.
- ✓ Acesta este un ***operator unar*** şi are aceeaşi prioritate ca şi ceilalţi operatori unari.
  - Operatorul new are ca valoare *adresa de început a zonei de memorie alocată în memoria heap* sau *zero (pointerul nul) în cazul în care nu se poate face alocarea*.
  - *Operandul operatorului new în cea mai simpla formă, este numele unui tip* (predefinit sau definit de utilizator).

3.1. Operatorul new

Exemplul 1:

```
int *pint;
pint = new int;
```

- ✓ Prin intermediul acestei expresii, se alocă în memoria heap o zonă de memorie în care se pot păstra date de tip int.
- ✓ Adresa de început a zonei alocate se atribuie pointerului *pint*.

Expresia:

***pint=100;** păstrează întregul 100 în zona respectivă.

3.1. Operatorul new

Exemplul 2:

int& i = *new int;

- ✓ Prin intermediul acestei declaraţii (definiţii) se alocă în memoria *heap* o zonă de memorie în care se pot păstra date de tip int.
- ✓ Numele *i* permite referirea la întregul păstrat în zona respectivă.
- ✓ Expresia de atribuire: ***i=100*** păstrează întregul 100 în zona respectivă.

3.1. Operatorul new

- ✓ Zonele de memorie alocate cu ajutorul operatorului ***new*** pot fi iniţializate.
- ✓ În acest scop se utilizează o expresie de forma:

new *tip(expresie)*

unde:

- ➢ *tip* – este numele unui tip de date
- ➢ *expresie* – este o expresie a cărei valoare iniţializează zona de memorie

3.1. Operatorul new

Exemplul 1:

```
double *pdouble;
pdouble = new double(3.14159265);
```

Această instrucţiune realizează următoarele:

- ✓ alocă în memoria ***heap*** o zonă de memorie în care se pastrează valoarea 3.14159265 în format real dublă precizie
- ✓ adresa de început a acestei zone de memorie se atribuie variabilei *pdouble*

3.1. Operatorul new

Exemplul 2:

double pi = *new double(3.14159265);

Prin această declaraţie se rezervă, în memoria heap, o zonă de memorie în care se păstrează valoarea 3.14159265 în format real dublă precizie.

Data respectivă se poate referi cu ajutorul numelui pi.

De exemplu, pi poate fi utilizat în mod obişnuit în expresii de forma:

- **pi*r*r**
- **sin(pi/2)**
- **x*180/pi,** etc.

3.1. Operatorul new

- ✓ O altă utilizare importantă este aceea de alocare a unei zone de memorie, în memoria *heap*, pentru tablouri.

În acest scop, utilizăm o expresie de forma:

new *tip[expresie]*

unde *expresie* – expresie de tip intreg;

- ✓ Prin această construcţie se rezervă, în memoria *heap*, o zonă de memorie de *expresie*sizeof(tip) octeţi*.
- ✓ Valoarea expresiei de mai sus, este adresa de început a zonei de memorie rezervată prin operatorul *new*.

Exemplu:

```
double *tab;
int m,n;
.....
tab=new double[m*n];
```

- ✓ Această instrucţiune rezervă în memoria *heap* o zonă de *m*n*sizeof(double)* octeţi.
- ✓ Adresa de început a acestei zone de memorie se atribuie pointerului *tab*.
- ✓ Elementele unei astfel de zone de memorie nu pot fi iniţializate decât numai prin secvenţe de program corespunzătoare.

De exemplu, pentru a initializa cu 0 cele *m*n* elemente de tip *double* rezervate ca mai sus, putem folosi instrucţiunea *for* de mai jos:

```
for(int i=0; i<m*n; i++)
    tab[i]=0.0;
```

Conţinutul cursului

1. Pointeri la funcţii
2. Folosirea pointerilor şi tablourilor ca parametri în funcţii
3. Alocarea dinamică a memoriei
   - 3.1. Operatorul new
   - 3.2. Operatorul delete
4. Structuri implementate dinamic:
   - 4.1. Stiva
   - 4.2. Coada
   - 4.3. Lista simplu înlănţuită
   - 4.4. Lista dublu înlănţuită

3.2. Operatorul delete

- ✓ O zonă de memorie alocată prin operatorul ***new*** se eliberează prin operatorul ***delete***.
- ✓ Daca *p* este un pointer spre *tip*:

tip *p;

şi

p=*new tip*;

- ✓ atunci zona din memoria heap alocată cu ajutorul lui new se eliberează folosind construcţia:

delete p

3.2. Operatorul delete

- De asemenea, operatorul ***delete*** se utilizează pentru a dezaloca tablourile alocate prin ***new***.
- Fie alocarea:

```
tip *p=new tip[expresie];
```

- Această zonă se eliberează folosind o construcţie de forma:

```
delete[expresie] p;
```

Conţinutul cursului

1. **Pointeri la funcţii**
2. **Folosirea pointerilor şi tablourilor ca parametri în funcţii**
3. **Alocarea dinamică a memoriei**
   - **3.1. Operatorul new**
   - **3.2. Operatorul delete**
4. **Structuri implementate dinamic:**
   - **4.1. Stiva**
   - **4.2. Coada**
   - **4.3. Lista simplu înlănţuită**
   - **4.4. Lista dublu înlănţuită**

4.1. Stiva

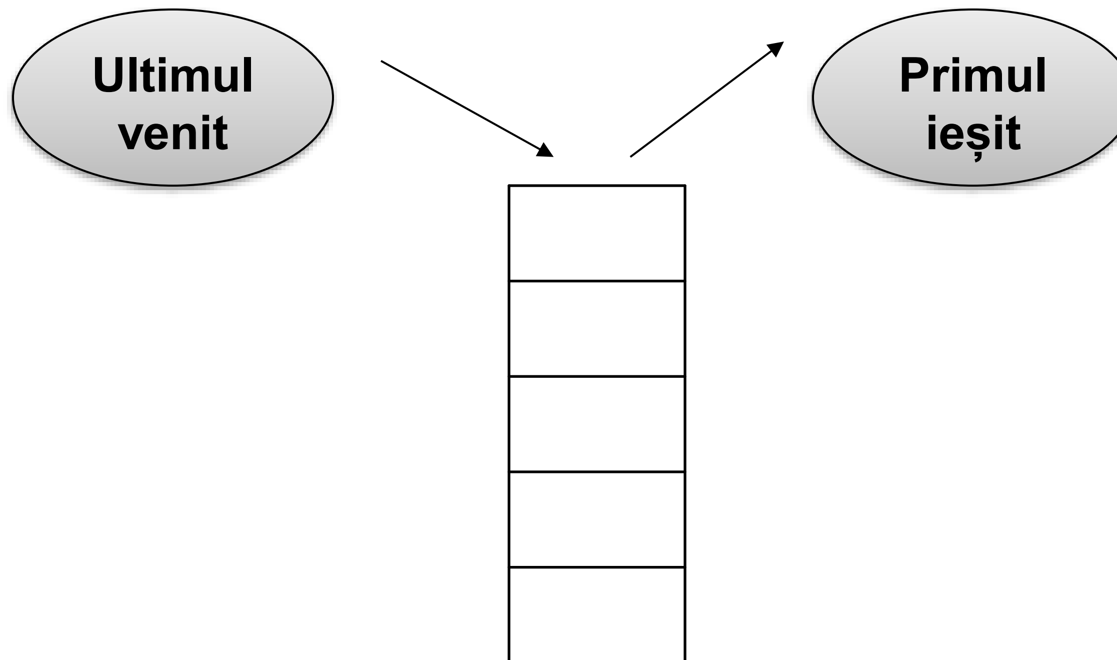
O stivă este un tip de dată ale cărui operaţii de inserare şi de ştergere păstrează această regulă:

ultimul - venit - primul - ieșit
LIFO (Last-In-First-Out)

✓Astfel, într-o stivă pot fi adăugate sau şterse elemente doar în vârful stivei.

4.1. Stiva

Grafic, o stivă se poate reprezenta astfel:



4.1. Stiva

- ✓ Pentru a putea realiza implementarea dinamică a stivei, elementele unei astfel de structuri vor fi de tip **structură**, cu două feluri de informaţii:
- ✓ **informaţia propriu-zisă** (conţinutul efectiv al elementului respectiv şi care variază în funcţie de problemă – notată cu ***inf*** care este de un anumit tip de date ***tip***)
- ✓ **informaţia de legătură** (care este un pointer ce conţine adresa elementului precedent din stivă – notat cu ***leg***).

4.1. Stiva

Tipul de date necesar implementării dinamice a structurii de tip stivă este:

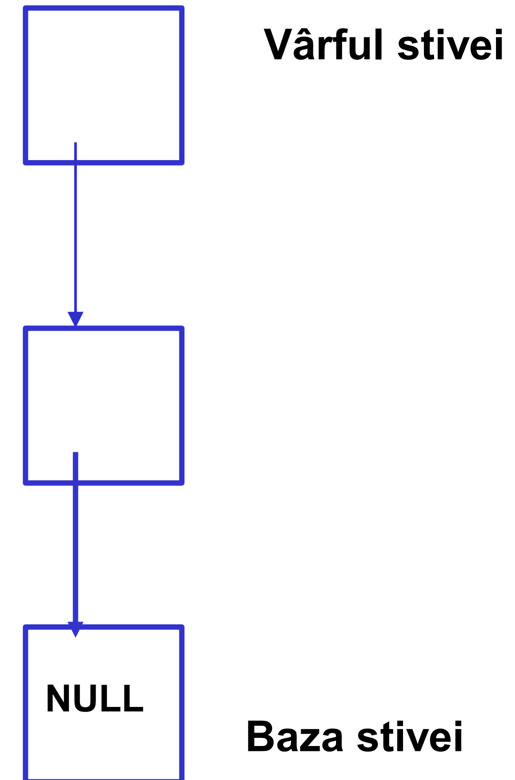
```
typedef struct tnod
{
        tip inf;                    //  informatia propriu-zisa
        struct tnod *leg;   //  informatia de legatura
} TNOD;
```

4.1. Stiva

- ✓ Adăugarea sau extragerea unui element se face la un singur capăt numit **vârful stivei.**
- ✓ Elementul introdus primul în stivă se afla la **baza stivei**.
- ✓ Informaţia de legătură a fiecărui element din stivă reprezintă adresa elementului pus anterior în stivă, excepţie făcând elementul de la bază, a cărui informaţie de legătură este **NULL**.

- Se observă că este necesar şi suficient să fie reţinută ***adresa elementului din vârful stivei*** într-un pointer pe care îl voi nota cu ***vf*(TNOD *vf)**, celelalte elemente putând fi accesate cu ajutorul informaţiei de legătură de care dispune fiecare element al stivei.

- **Stiva** se poate reprezenta grafic astfel:



4.1. Stiva

Operaţiile ce se pot efectua asupra stivei

1) **Iniţializarea** stivei:

```
void init( TNOD* vf )
{
   vf=NULL;
}
```

2) Adăugarea unui element x în vârful stivei:

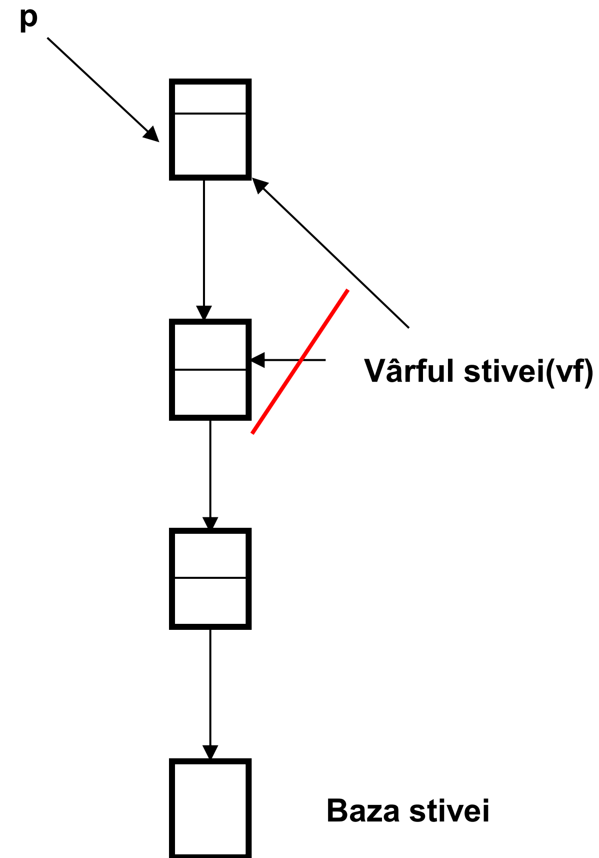
Deoarece toate adăugările şi ştergerile se fac la un singur capăt al stivei, singura posibilitate de a adăuga un element în stivă este în vârf.

Acest lucru se realizează astfel:

- Se alocă memorie pentru noul element (**a**);
- Se iniţializează zona de informaţie propriu-zisă (utilă) a acestuia (**b**);
- Se iniţializează informaţia de legătură cu adresa elementului care a fost anterior în vârful stivei, adresă păstrată în variabila **vf (c);**
- Se actualizează variabila **vf** cu adresa noului element alocat (**d**).
- Putem observa că acest algoritm este corect şi dacă stiva era vidă înainte de adăugare(**vf**=**NULL**).

Această operaţie se poate reprezenta grafic astfel:

```
void adaug(tip x,TNOD *vf)
   {
    TNOD *p;          //pointer de lucru
    p=new TNOD;    //(a)
     p->inf=x;         //(b)
    p->leg=vf;        //(c)
     vf=p;             //(d)
}
```



3) Extragerea elementului din vârful stivei

Prin extragerea unui element se înţelege eliminarea acestuia din stivă.

Această operaţie se poate realiza numai dacă ***stiva este nevidă*** şi în caz afirmativ ***se va şterge elementul din vârful stivei.***

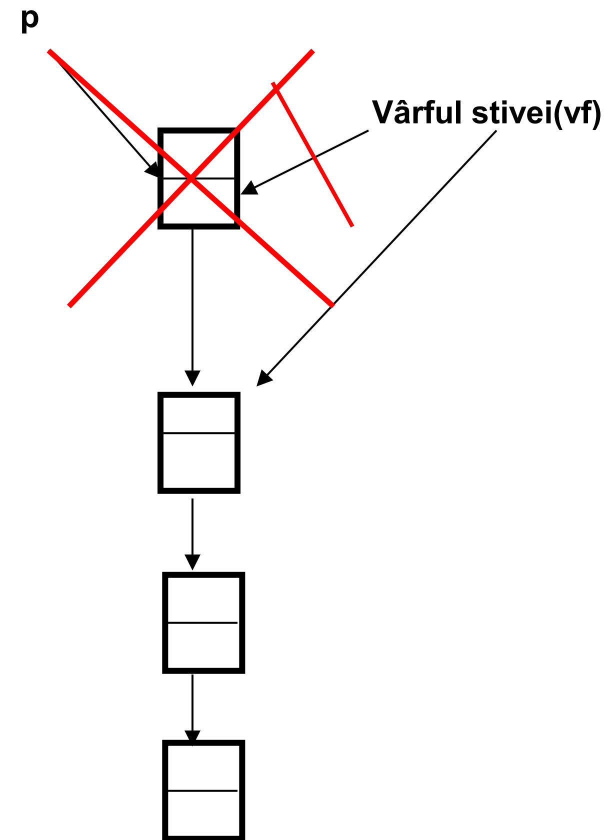
Operaţia se efectuează astfel:

- Într-un pointer de lucru notat **p**, se reţine adresa elementului din vârful stivei; (**a**)
- În variabila **q** se extrage elementul din vârful stivei;
- Adresa elementului următor celui din vârful stivei devine adresa noului vârf al stivei; (**b**)
- Se eliberează memoria ocupată de elementul care a fost anterior în vârful stivei.(**c**)

```
void extragere( TNOD *vf,
   tip *q)
{

   TNOD *p;  //pointer de lucru
   p=vf;          //(a)
   *q=vf->inf; // Se extrage în
   variabila *q valoarea din
   varful stivei
   vf=vf->leg;  //(b)
    delete p;   //(c)
}
```

Această operaţie se poate reprezenta grafic astfel:



4) Verificarea dacă stiva este vidă

```
int stvida( TNOD *vf )
{
  return vf==NULL;
}
```

5) Numărarea elementelor din stivă

```
int cardinal(TNOD *vf)
{
    int m=0; //contorizeaza elementele stivei
    TNOD* p; //pointer de lucru
    p=vf;
    while (p!=NULL)
    {
        m++;
        p=p->leg;
    }
    return m;
}
```

Bibliografie

Mihaela Runceanu, Adrian Runceanu - **STRUCTURI DE DATE ALOCATE DINAMIC. Aspecte metodice. Implementări în limbajul C++**, 2016, Editura Academica Brancusi din Targu Jiu

https://www.researchgate.net/publication/308938197_STRUCTURI_DE_DATE_ALOCATE_DINAMIC_Aspecte_metodice_Implementari_in_limbajul_C/download

Întrebări?