UNIVERSITATEA **TITU MAIORESCU**

Facultatea de Informatică

Conf. univ. dr. DANIELA JOIȚA

ALGORITMI
ȘI
STRUCTURI DE DATE

Curs pentru învăţământul la distanţă

editura universităţii *Titu Maiorescu*

BUCUREŞTI – 2014

Introducere

Acest material este destinat studenților anului II, invatamant la distanță, specializarea Informatică. Modul de prezentare are în vedere particularităţile învăţământului la distanţă, la care studiul individual este determinant. În timp ce **profesorul** sprijină studentul prin coordonarea învățării și prin feedback periodic asupra acumulării cunoștințelor și a deprinderilor, **studentul** alege locul, momentul și ritmul pentru studiu, dispune de capacitatea de a studia independent și totodată își asumă responsabilitatea pentru inițierea și continuarea procesului educațional.

Disciplina **Algoritmi şi structuri de date** utilizează noţiunile predate la disciplinele *Bazele Informaticii*, *Programare procedurală*, *Fundamentele algebrice ale informaticii* şi *Algoritmica grafurilor*, discipline studiate în anul I.

Competențele dobândite de către studenți prin însușirea conținutului cursului **Algoritmi şi structuri de date** sunt des folosite la disciplinele de specialitate precum *Programare orientată pe obiecte*, *Tehnici avansate de programare*, *Proiectarea interfețelor grafice*, *Sisteme de gestiune a bazelor de date*, etc. O neînțelegere a noțiunilor fundamentale prezentate în acest curs poate genera dificultăți în asimilarea conceptelor mai complexe ce vor fi introduse în aceste cursuri de specialitate.

Principalele obiective ale disciplinei **Algoritmi şi structuri de date** sunt:

- Cunoașterea principalelor structuri de date liniare și neliniare folosite in informatică;
- Asimilarea metodelor de analiza a eficientei unui algoritm;
- Cunoasterea principalilor algoritmi de sortare si cautare;
- Implementarea algorimilor si a structurilor de date invatate, într-un limbaj de programare, cu precădere în limbajul C/C++.

Competențele specifice disciplinei **Algoritmi şi structuri de date** se pot clasifica după cum urmează:

1. Cunoaştere şi înţelegere
- Identificarea si clasificarea unor tipuri de structuri de date
- Intelegerea rolului structurilor alocate dinamic in raport cu cele statice
- Cunoasterea si inţelegerea modalitatilor de analiza a eficientei algoritmilor
- Cunoasterea principalilor algoritmi de sortare si cautare

2. Explicare şi interpretare • Explicarea si interpretarea conceptului de eficienta a unui algoritm • Interpretarea modului de organizare a datelor • Explicarea modalitatilor de functionare a algoritmilor specifici disciplinei
3. Instrumental – aplicative • Implementarea într-un limbaj de programare a algoritmilor specifici disciplinei • Proiectarea aplicaţiilor pentru rezolvarea unor probleme utilizând instrumente specifice de structurare a datelor • Corelarea cunoştinţelor teoretice cu abilitatea de a le aplica în practică • Elaborarea unui proiect care sa scoată in evidenţă importanţa algoritmilor specifici disciplinei precum și înțelegerea modalității de alegere a algoritmului optim
4. **Atitudinale** • Manifestarea unor atitudini favorabile faţă de ştiinţă şi de cunoaştere în general • Formarea obişnuinţelor de a recurge la concepte şi metode informatice de tip algoritmic specifice în abordarea unei varietăţi de probleme • Exprimarea unui mod de gândire creativ în structurarea şi rezolvarea problemelor

Structura cursului este următoarea:

Unitatea de învățare 1. ***Structuri de date liniare***

Unitatea de învățare 2. ***Structuri de date neliniare***

Unitatea de învățare 3. ***Analiza algoritmilor***

Unitatea de învățare 4. ***Algoritmi de sortare***

Unitatea de învățare 5. ***Algoritmi de căutare***

Este foarte important ca parcurgerea materialului sa se faca in ordinea unităților de învățare incluse (1 – 5). Fiecare UI (unitate de învățare) conține, pe langa prezentarea notiunilor teoretice, exerciții rezolvate, activități de lucru individual la care sunt prezentate și indicații de rezolvare, exemple iar la finalul fiecărei lecții, un test de autoevaluare. În plus, la sfârșitul fiecărei UI sunt incluse probleme propuse care testeaza cunoasterea notiunilor teoretice de catre student.

Materialul a fost elaborat astfel incat algoritmii prezentati să poată fi implementati în orice limbaj de programare. Pentru a face o alegere, limbajul de programare folosit in aplicații va fi limbajul C/C++.

Pachet software recomandat:

Orice IDE (Integrated Development Environment) pentru limbajul C/C++ poate fi folosit.

Bibliografia recomandată

1. Ioan Tomescu, *Data Structures*, Editura Universitatii din Bucuresti,, 1997
2. Knuth D.E., *Arta programarii calculatoarelor*, Editura TEORA, 1998-2004.
3. Cormen T.H, Leiserson C.E., Rivest R.L., Stein C, *Introduction to Algorithms*, The MIT Press, 2001
4. Brian Kernighan si Denis Ritchie, *Limbajul C*, Editura TEORA
5. Daniela Joiţa, Programare procedurală, Editura Universităţii Titu Maiorescu, 2008

Vă precizăm de asemenea că, din punct de vedere al verificărilor şi al notării, cu adevărat importantă este capacitatea pe care trebuie să o dobândiţi şi să o probaţi de a rezolva toată tipologia de probleme aplicative aferente materialului teoretic prezentat în continuare. Prezentăm în continuare criteriile de evaluare şi ponderea fiecărei activităţi de evaluare la stabilirea notei finale.

La stabilirea notei finale se iau în considerare	Ponderea în notare, exprimată în % {Total = 100%}
- răspunsurile la examen (evaluarea finală)	50%
- răspunsurile finale la lucrările practice de laborator	20%
- testarea periodică prin teme pentru acasa	10%
- testarea continuă pe parcursul semestrului	10%
- activităţile gen proiecte	10%
Modalitatea de evaluare finală: lucrare scrisă descriptivă şi/sau probleme	
Cerinţe minime pentru nota 5	**Cerinţe pentru nota 10**
• Însuşirea cunoştinţelor de bază • Obţinerea unui procent de cel putin 45% din procentul maxim alocat fiecarei activitati care se considera in stabilirea notei finale. • Activitate în timpul semestrului	• Rezolvarea corectă şi completă a subiectelor de examen • Efectuarea corecta si completa a temelor pentru acasa • Participarea activă la curs si laborator • Elaborarea unui proiect corect, complet si bine documentat

În dorinţa de ridicare continuă a standardelor desfăşurării activitatilor dumneavoastra, dupa fiecare unitate de invatare va rugăm să completaţi un formular de feedback şi să-l transmiteţi indrumatorului de an. Acest formular se gaseste la sfarsitul acestui material

In speranta ca organizarea si prezentarea materialului va fi pe placul dumneavoastra, va uram MULT SUCCES!

Coordonator disciplină: Conf. univ. dr. Daniela Joiţa

CUPRINS

UNITATEA DE ÎNVĂȚARE 1

Structuri de date liniare

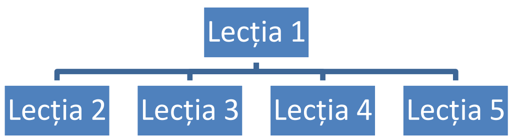
Obiective urmărite:

La sfârşitul parcurgerii acestei UI, studenţii

- vor şti să identifice şi să clasifice principalele tipuri de structuri de date liniare
- vor şti să interpreteze modul de organizare a datelor liniare
- vor cunoaşte operaţiile specifice fiecărui tip de structură de date liniar
- vor şti să implementeze într-un limbaj de programare principalele tipuri de structuri de date liniare
- vor înţelege rolul structurilor alocate dinamic în raport cu cele statice
- vor şti să identifice tipul de structură de date liniară care poate fi folosită pentru organizarea datelor şi informaţiilor în vederea rezolvării unor probleme practice

Ghid de abordare a studiului:

Timpul mediu necesar pentru parcurgerea si asimilarea unitatii de invatare: 8h.
Lecțiile se vor parcurge în ordinea sugerată de diagrama de mai jos.



Rezumat

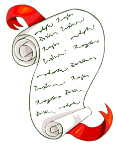
În această modul sunt prezentate principalele tipuri de structuri de date liniare: liste, stive, cozi. Se face o analiză a acestora din punct de vedere a modului de alocare a zonelor de memorie necesare reprezentării lor şi o prezentare detaliată a operațiilor corespunzătoare acestor tipuri de date. Sunt discutate implementările în limbajul de programare C/C++ a acestor structure de date. Sunt analizate problemele practice care pot fi rezolvate prin folosirea acestui mod liniar de organizare a datelor problemelor.

Cuvinte cheie

listă, alocare secvenţială, alocare înlănţuită, pointer, HEAD, eliminare, inserare, listă vidă, UNDERFLOW, OVERFLOW, stivă, coadă, nod listă, liste push-down, LIFO, FIFO

Lecția 1. Liste liniare

In viata de fiecare zi construim liste ca o modalitate de a pune lucrurile in ordine: lista studentilor unei grupe, lista numerelor din cartea de telefon, programul TV, lista de cumparaturi.



O **listă liniară** este un set finit de elemente ordonate liniar, adică există un element considerat primul in listă, un element considerat ultimul și pentru orice alt element din listă există un element precedent și un element următor.

Principalele **operatii cu liste** sunt:

- accesarea sau modificarea unui element din lista
- inserarea unui element in lista
- eliminarea unui element din lista

Modul in care aceste operatii sunt efectuate depinde de modul in care sunt reprezentate listele. Exista doua feluri de reprezentari: secventiala si inlantuita.

1.1 Alocarea secvențială

In **alocarea secventiala**, o lista se reprezinta ca un sir in care elementele sunt memorate in locatii consecutive de memorie.

In C, o astfel de lista este, de fapt, un sir care se defineste, in general:

nume_tip nume_sir[nr_elemente];

unde *nume_tip* este tipul elementelor sirului, *nume_sir* este numele sirului (listei), iar *nr_elemente* este o constanta ce reprezinta numarul maxim posibil de elemente ale sirului.

In C, se poate face referire la elementele sirului prin intermediul indicelui. Primul element in lista are indice 0 si se noteaza *nume_sir[0]*, ultimul este *nume_sir[nr_elemente – 1]*, iar un element de indice *k*, $1 \leq k \leq$ *nr_elemente - 2* este *nume_sir[k]* si are ca precedent elementul *nume_sir[k - 1]* si este urmat de elementul *nume_sir[k + 1]*. Elementele sirului sunt memorate astfel incat:

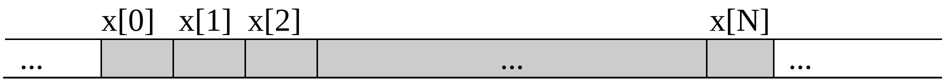
adresa lui *nume_sir[k]* = adresa lui *nume_sir[0]* + *k *sizeof (nume_tip)*

unde *sizeof(nume_tip)* returneaza numarul de octeti necesari memorarii unei singure variabile de tipul *nume_tip*.

In cele ce urmeaza, presupunem ca avem un sir definit

T x[N];

unde *N* este o constanta suficient de mare, *T* este un tip de date definit anterior (eventual printr-o definitie de tipul *typedef*), iar *n* este numarul de elemente din lista, $n \leq N$.



Operatiile cu liste prezentate mai sus, se pot descrie astfel:

– **accesarea/ modificarea unui element** se face prin intermediul indicelui elementului

– **inserarea unui element** se poate face intr-o pozitie data, dupa sau inaintea unui element dat. Prezentam in continuare inserarea unui element numit *elem_nou* intr-o pozitie data *k*, deoarece celelalte se reduc la aceasta.

Algoritm: Presupunem $0 \leq k \leq n$. Daca n = N atunci se produce OVERFLOW adica spatiul alocat listei este ocupat in totalitate si ca nici o alta inserare nu este posibila. In caz contrar se muta elementele sirului x[k], ..., x[n-1], cate un bloc de memorie spre dreapta, incepand cu ultimul. Se introduce elementul nou in pozitia k. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if n = N then OVERFLOW
        else for i = n-1, k, -1
                        x[i+1] = x[i]
             endfor
endif
x[k] = elem_nou
n = n +1
```

– **eliminarea elementului** de indice k din lista.

Algoritm: Daca n = 0 atunci se produce UNDERFLOW adica lista este vida si deci eliminarea unui element din lista nu este posibila. Presupunem $0 \leq k \leq n\text{-}1$. Se salveaza informatia continuta de elementul de indice k pentru cazul in care se doreste prelucrarea ei. Se muta elementele sirului x[k+1], ..., x[n-1], cate un bloc de memorie spre stanga, incepand cu x[k+1]. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if n = 0 then UNDERFLOW
          else elem_sters = x[k]
                 for i = k, n-2,1
                         x[i] = x[i+1]
                 endfor
endif
n = n –1
```

Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de inserare a unui element numit elem_nou intr-o pozitie dată k într-o listă alocată secvențial. Se cunosc numele listei (x), numărul de elemente ale listei (n) și numărul maxim de elemente ale listei.

tip funcție

nume funcție

Spațiul alocat listei este ocupat în totalitate.

fisier1.c

```
int inserare(int x[], int n, int elem_nou, int k)
{
    int i;
    if (n == N) {
                    printf("Inserare nereusita: OVERFLOW\n");
                    return n;
                  }
    for (i = n-1; i >= k; i--)
         x[i+1] = x[i];
    x[k] = elem_nou;
    n = n +1;
    return n;
}
```

Se mută elementele sirului x[k], ..., x[n-1], câte un bloc de memorie spre dreapta, începand cu ultimul.

Se incrementează nr. de elem.

Funcția returnează numărul de elemente ale listei noi.

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o lista liniara alocata secvential.

Indicații de rezolvare: Este bine să scrieți un program în care operațiile de inserare, stergere, creare și afișare listă să fie descrise fiecare printr-o funcție, iar în funcția main să se facă doar apelarea acestor funcții după cum se dorește efectuarea uneia sau a alteia dintre operații. Pentru aceasta puteti să completați următorul program C cu corpul funcțiilor pentru operațiile cu liste.

```c
#include <stdio.h>
#define N 100

void citeste(int x[], int n)
{
}
void scrie(int x[], int n)
{
 }
int inserare(int x[], int n, int elem_nou, int k)
{
}
int eliminare(int x[], int n, int k)
{
}
```

Se vor completa cu instrucțiunile necesare

```c
int main()
{
   int x[N],i, n, val, elem_nou, k;
   char c;
   printf("Introduceti dimensiunea sirului: ");
   scanf("%d", &n);
   citeste(x, n);

//        Accesarea/ modificarea unui element
   printf("\nIntroduceti indicele elementului pe care doriti sa il accesati: ");
   scanf("%d", &i);
   printf("\nAti accesat elementul: x[%d]=%d", i, x[i]);
   printf("\nDoriti sa il modificati?(Y/N) ");
   rewind(stdin);
   c = getchar();
   if(c =='Y' || c=='y')
   {
       printf("\nIntroduceti noua valoare: ");
       scanf("%d", &val);
       x[i]=val;
       printf("\nSirul este acum\n ");
       scrie(x, n);
   }

// Inserarea unui element: elem_nou intr-o pozitie data k
   printf("Introduceti elementul pe care vreti sa il adaugati in lista si pozitia\n");
   scanf("%d %d", &elem_nou, &k);
   n = inserare(x, n, elem_nou, k);
   printf("Sirul este acum\n ");
   scrie(x, n);
```

```
//  Eliminarea elementului de indice k din lista.
    printf("Introduceti pozitia elementului pe care vreti sa-l stergeti din lista: ");
    scanf("%d", &k);
    n = eliminare(x, n, k);
    printf("Sirul este acum\n ");
    scrie(x, n);
    return 0;
}
```

1.2 Alocarea înlănțuită

De multe ori, memoria nu este ocupata la rand, zone de memorie libere alternand cu zone de memorie ocupate. Alocarea secventiala a unei liste se poate face daca exista blocuri de memorie suficient de mari pentru a o memora. Daca, de exemplu, vrem sa alocam spatiu pentru o lista cu M elemente, dar nu exista nici un bloc de memorie de marime M, desi exista trei blocuri de memorie de marime M/2, ar trebui sa gasim o modalitate de reorganizare a memoriei, altfel lista nu poate fi creata sau sa gasim o alta metoda de reprezentare a listei in care elementele sa nu mai fie memorate in locatii consecutive. Acesta din urma este cazul alocarii inlantuite.

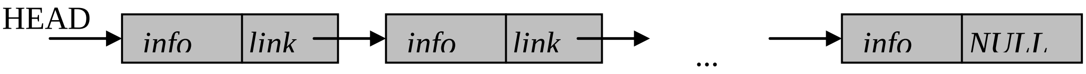
In alocarea inlantuita, pentru a pastra ordinea listei va trebui sa stim care este primul element al listei, al doilea etc. Pentru aceasta vom folosi un pointer la primul element al listei (numit pointerul listei si notat HEAD), iar pentru a sti care este urmatorul element in lista fiecare element va trebui sa contina (sa memoreze) pe langa informatia corespunzatoare lui si un pointer la elementul urmator in lista. Elementele nemaifiind memorate in locatii consecutive, acesti pointeri ajuta la reconstituirea listei. Ultimul element in lista va avea drept componenta pointerul NULL. Din acest motiv, fiecare element va fi de tip structura, continand doua campuri: *info* (folosit pentru memorarea informatiei corespunzatoare elementului) si *link* (un pointer la elementul urmator). Datorita reprezentarii structurate, elementele unei astfel de liste se mai numesc si noduri.

> In **alocarea inlantuită**, fiecare element al listei se reprezintă ca o variabilă de tip structură cu două componente: informația propriu-zisă și un pointer la următorul element din listă. Lista este bine definită prin pointerul la primul element al listei (HEAD).

In C, putem defini tipul de date asociat unui nod astfel:

```
struct nod
{
        T info;
        struct nod *link;
};
typedef struct nod NOD;
```

unde *T* este presupus definit anterior (eventual printr-o definitie *typedef*).



Operatiile cu liste se pot descrie astfel:

– **accesarea/ modificarea unui element** ce contine o valoare data *k*, *k* de tip *T*.

Algoritm: Folosim variabilele *gasit* de tip boolean, *gasit* = *false* daca elementul cu valoarea k nu a fost gasit in lista si *true* in caz contrar si *iter* de tip pointer la un nod care initial pointeaza la primul nod al listei si va parcurge lista pana cand nodul dorit este gasit sau pana cand se ajunge la ultimul nod al listei, in caz contrar.

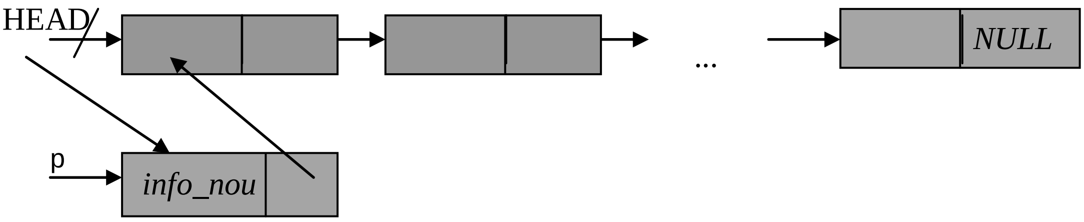
```
gasit = false
iter = HEAD
while (not gasit and iter ≠ NULL)
        if iter -> info = k then gasit = true
        else iter = iter -> link
        endif
endwhile
if gasit then acceseaza nodul pointat de iter
     else afiseaza mesajul "Nu exista nici
un nod cu informatia data."
endif
```

– **inserarea unui nou nod**

- *la inceputul listei:*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod.

```
Aloca memorie pentru un nod nou. Returneaza
p, un pointer la noul nod.
     if p ≠ NULL then
          p -> link = HEAD
          p -> info = info_nou
          HEAD = p
     else OVERFLOW
     endif
```

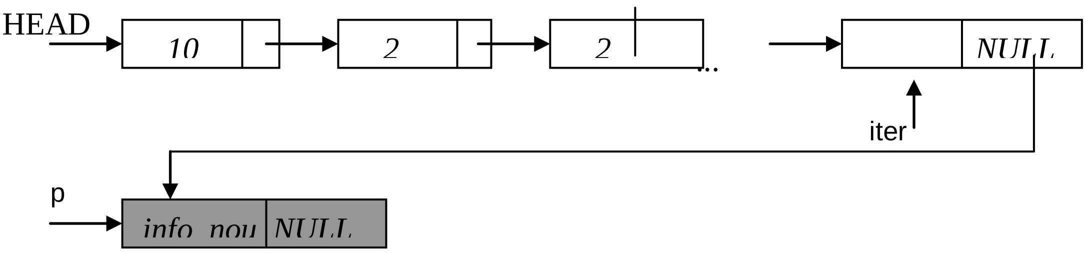


Observatie: Algoritmul functioneaza chiar daca lista este nula si se poate folosi pentru crearea unei liste prin introducerea pe rand, unul cate unul, a elementelor listei.

- *la sfarsitul listei:*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista, *p* pointer la un nod si *iter* care initial pointeaza la primul nod al listei si va parcurge lista pana cand acesta va pointa la ultimul nod din lista.

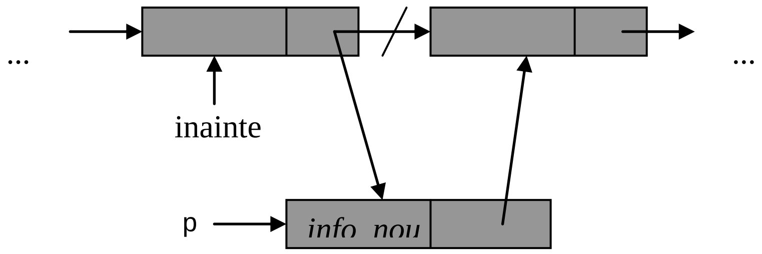
```
Aloca memorie pentru un nod nou. Returneaza p, un
pointer la noul nod.
        if p ≠ NULL
        then
                p -> link = NULL
                p -> info = info_nou
                iter = HEAD
                while (iter ≠ NULL and iter -> link ≠
        NULL)
                iter = iter -> link
                endwhile
                if iter = NULL then HEAD = p
                else iter ->link =p
                endif
        else OVERFLOW
        endif
```



- *dupa un nod dat:*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod si *inainte* pointer la nodul dupa care se doreste introducerea noului nod.

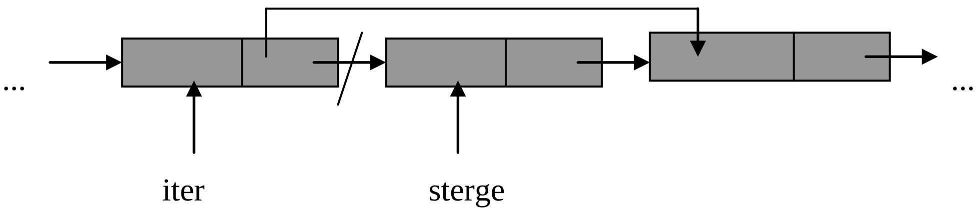
```
Aloca memorie pentru un nod nou. Returneaza p, un
pointer la noul nod.
        if p ≠ NULL then
                p -> link = inainte ->link
                inainte -> link =p
                p -> info = info_nou
        else OVERFLOW
        endif
```



– **eliminarea unui nod dat din lista:**

Algoritm: Folosim variabilele *sterge* si *iter* de tip pointer, *sterge* este pointer la nodul care trebuie eliminat din lista si *iter* care initial pointeaza la primul nod al listei si va parcurge lista pana cand acesta va pointa la nodul dinaintea nodului de sters din lista.

```
iter = HEAD
while (iter ≠ NULL and iter -> link ≠ sterge)
        iter = iter -> link
endwhile
if iter = NULL then tipareste mesajul "Eroare. Lista este vida.
UNDERFLOW."
else recupereaza informatia nodului de sters sterge->info
     iter ->link = sterge -> link
endif
```



Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de ștergere a unui nod dintr-o listă alocată înlănțuit. Se dau pointerul head la începutul listei și sterge, pointer la nodul care trebuie eliminat. Se dau următoarele definiții:

```
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *eliminare(NOD *head, NOD* sterge)
{
   NOD *iter;
   if (head == NULL)
   {
       printf ("Eroare. Lista este vida. UNDERFLOW.");
       return NULL;
       }
   if (sterge == head)
     {
        printf("Valoarea elementului sters este: %d", sterge->info);
        head = head -> link;
        return head;
        }
   iter = head;
   while (iter != NULL && iter -> link != sterge)
   iter = iter -> link;
   printf("Valoarea elementului sters este: %d", sterge->info);
   iter ->link = sterge -> link;
   return head;
}
```

Analizați mereu și cazul în care lista este vidă.

nodul de șters este primul nod

Găsește nodul dinaintea nodului de șters pentru a putea reface legăturile

Funcția returnează un pointer la noua listă.

Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de căutare a unui nod dintr-o listă alocată înlănțuit. Se dau pointerul head la începutul listei și valoarea căutată în listă. Presupunem că valoarea din fiecare nod al listei este un int. Funcția va returna un pointer la nodul găsit sau NULL în cazul în care nu a fost găsit.

Se dau următoarele definiții:

```
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *cauta(NOD *head, int val)
{
   NOD *iter; int gasit;
   iter = head; gasit =0;
         while (!gasit && iter != NULL)
         {
       if (iter -> info == val) gasit = 1;
                   else iter = iter -> link;
   }
   if(gasit) return iter;
   else return NULL;
}
```

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o lista liniara alocata inlantuit.

Indicații de rezolvare: Completați următorul program C cu corpul funcțiilor pentru operațiile cu liste.

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


struct nod
{
         int info;
         struct nod *link;
};
typedef struct nod NOD;

// prototipurile functiilor care trebuie completate

NOD *crearelista();
void afiseazalista(NOD *head);
void accesare(NOD *head, int val);
NOD *inserareinceput(NOD *head, int info_nou);
NOD *inseraresfarsit(NOD *head, int info_nou);
NOD *inseraredupa(NOD *head, int val, int info_nou);
NOD *cauta(NOD *head, int val);
 NOD *eliminare(NOD *head, NOD* sterge);
```

Corpurile funcțiilor se vor adăuga la sfârșitul programului.

```
 int main()
{
    NOD *head, *sterge; int val, alegere, info_nou;
    head = crearelista();
    afiseazalista(head);

//          Accesarea/ modificarea unui element
    printf("\nIntroduceti valoarea elementului pe care doriti sa il accesati: ");
    scanf("%d", &val);
    accesare(head, val);

// Inserarea unui element: elem_nou intr-o pozitie data
    printf("Introduceti elementul pe care vreti sa il adaugati in lista\n");
    scanf("%d", &info_nou);
    printf("Introduceti pozitia noului element in lista:\n");
    printf("1 - pentru inceputul listei\n");
    printf("2 - pentru sfarsitul listei\n");
    printf("3 - pentru dupa un nod dat\n");
    scanf("%d", &alegere);
    switch(alegere)
    {
      case 1: head = inserareinceput(head, info_nou);
              break;
      case 2: head = inseraresfarsit(head, info_nou);
              break;
      case 3: printf("Introduceti valoarea elementului dupa care doriti sa
introduceti nodul: ");
              scanf("%d", &val);
              head = inseraredupa(head, val, info_nou);
              break;
      default:printf("Pozitie invalida");
    }
    printf("Lista este acum\n ");
    afiseazalista(head);

//  Eliminarea unui element
    printf("Introduceti valoarea elementului pe care vreti sa-l stergeti din lista: ");
    scanf("%d", &val);
    sterge = cauta(head, val);
    if (sterge != NULL) head =eliminare(head, sterge);
    else printf("Valoarea %d nu este in lista\n", val);
    printf("\nLista actualizata\n ");
    afiseazalista(head);

    getchar();
    getchar();
    return 0;
}
```

z

Test de autoevaluare 1

1. Scrieti o funcţie C care afişează in ordine toate elementele unei liste de numere întregi alocate înlănţuit. Incorporaţi funcţia într-un program C şi testaţi-o.

2. Cum se reprezintă o listă alocată secvenţial in limbajul de programare C?

3. Ce face următorul algoritm?

```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
if p ≠ NULL then
        p -> link = HEAD
        p -> info = info_nou
        HEAD = p
else OVERFLOW
endif
```

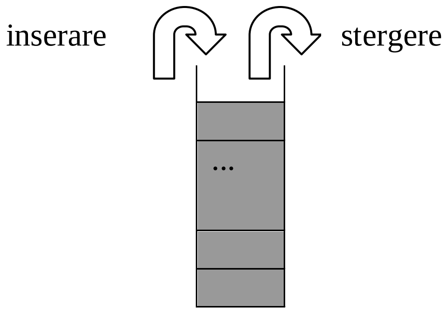
4. Transcrieți algoritmul de mai sus într-un fragment de cod scris în limbajul C.

5. Care sunt instrucţiunile C pentru alocarea dinamică a unui şir cu n elemente? Dar instrucţiunile C++?

Lecția 2. Stive

Stiva este o lista liniara in care inserarile si stergerile din lista se pot face numai pe la un capat al listei, numit <u>varful stivei</u>.

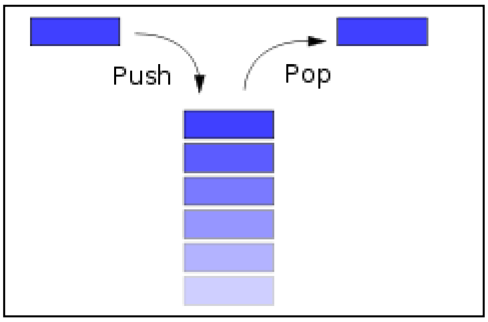
Singurul element care poate fi accesat la un moment dat este cel din varful stivei. Se poate face o analogie intre o stiva folosita in programare si o stiva de carti. Adaugarea unei carti se poate face numai in varful stivei de carti, peste cartile existente si singura carte ce poate fi accesata, eventual eliminata din stiva este cea din varful stivei.



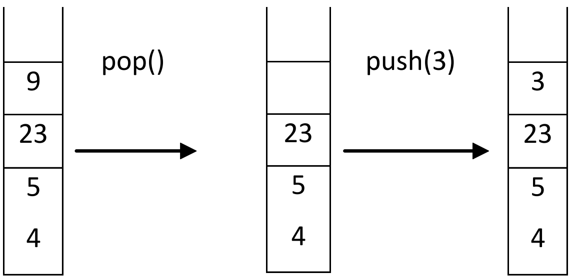
Stivele se mai numesc si liste *push-down* sau *LIFO (last in/first out)* deoarece primul element care este extras din stiva este ultimul introdus in stiva.

Notăm:

- inserarea unui element *a* intr-o stiva S: $a \Rightarrow S$,
- stergerea unui element *a* dintr-o stiva S: $S \Rightarrow a$.



Cele doua operatii se mai numesc si *push* respectiv *pop*.



Stiva originala

2.1 Alocarea secvențială

Presupunem că stiva este definită

T x[N];

unde *N* este o constanta suficient de mare, *T* este un tip de date definit anterior (eventual printr-o definitie de tipul *typedef*), iar *n* este numarul de elemente din stiva, $n \le N$, iar elementul din varful stivei este considerat elementul *x[n-1]*.

Operatiile permise cu stive se pot descrie astfel:

- **accesarea/ modificarea unui element:** numai elementul din vârful stivei poate fi accesat adică elementul *x[n-1]*

- **inserarea unui element:** $elem\_nou \Rightarrow S$

Algoritm: Daca n = N atunci se produce OVERFLOW adica spatiul alocat stivei este ocupat in totalitate si ca nici o alta inserare nu este posibila. In caz contrar elementul nou se adaugă în vârful stivei. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if n = N then OVERFLOW
         else n = n + 1
              x[n-1] = elem_nou
endif
```

- **ştergerea unui element:** $S \Rightarrow elem\_sters$

Algoritm: Daca n = 0 atunci se produce UNDERFLOW adica stiva este vidă si nicio ştergere nu este posibilă. In caz contrar elementul din vîrful stivei este şters. Valoarea acestuia este salvată în variabila *elem_sters*. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if n = 0 then UNDERFLOW
        else elem_sters = x[n-1]
             n = n – 1
endif
```

Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de inserare a unui element numit elem_nou într-o stivă alocată secvențial. Se cunosc numele listei (x), numărul de elemente ale stivei (n) și numărul maxim de elemente ale stivei.

```
int push(int x[], int n, int elem_nou)
{
    if (n == N) {
                printf("Inserare nereusita: OVERFLOW\n");
                return n;
              }
```

Funcția returnează numărul de elemente ale listei noi.

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o stivă alocata secvential.

Indicații de rezolvare: Este bine să scrieți un program în care operațiile de inserare, stergere, creare și afișare să fie descrise fiecare printr-o funcție, iar în funcția main să se facă doar apelarea acestor funcții după cum se dorește efectuarea uneia sau a alteia dintre operații. Pentru aceasta puteti să completați următorul program C cu corpul funcțiilor pentru operațiile cu stive.

```
#include <stdio.h>
#define N 10

void citeste(int x[], int n);
void scrie(int x[], int n);
int inserare(int x[], int n, int elem_nou);
int eliminare(int x[], int n);
```

Se vor completa cu instrucțiunile necesare

```
int main()
{
   int x[N], n, elem_nou, info;
   char c;
   printf("Introduceti dimensiunea stivei: ");
   scanf("%d", &n);
   citeste(x, n);
   scrie(x,n);

// Inserarea unui element: elem_nou
   do
   {
     printf("\nIntroduceti elementul pe care vreti sa il adaugati in stiva \n");
     scanf("%d", &elem_nou);
```

```
     if(inserare(x, n, elem_nou))
     n++;
     printf("Sirul este acum\n ");
     scrie(x, n);
     printf("Mai doriti sa introduceti un element (Y/N)?");
     rewind(stdin);
```

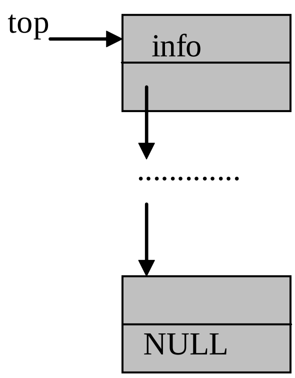
2.2 Alocarea inlantuita

In alocarea inlantuita, folosim aceeasi structura ca si in cazul listei liniare

struct nod
{
T info;
*struct nod *link;*
};
typedef struct nod NOD;

Notam cu *top* pointerul stivei (pointerul care puncteaza la elementul din varful stivei).

*NOD *top;*

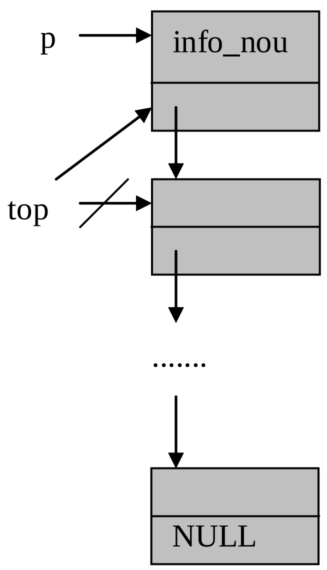


Cu aceste notatii, operatiile cu stive se pot descrie astfel:

inserarea unui nod nou	stergerea/accesarea unui nod

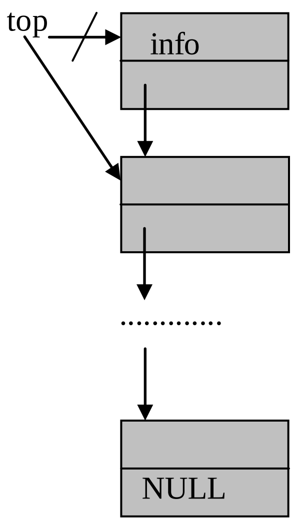
Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod.

```
Aloca memorie pentru un nod nou. Returneaza p, un pointer
la noul nod.
if p ≠ NULL then
        p -> link = top
        p -> info = info_nou
        top = p
else OVERFLOW
endif
```



Algoritm:

```
if top = NULL
    then UNDERFLOW
    else elem_sters = top -> info
             top = top -> link
endif
```



Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de ștergere a unui nod dintr-o stivă alocată înlănțuit. Se da pointerul top la vârful stivei. Se dau următoarele definiții:

```
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *pop(NOD *top)
{
    if (top == NULL)
    {
        printf ("Eroare. Stiva este vida. UNDERFLOW.");
        return NULL;
    }
    printf("\nStergem un element din stiva\n");
    printf("Valoarea elementului sters este: %d", top->info);
    top = top -> link;
    return top;
}
```

stiva vidă

Funcția returnează pointerul la noua stivă.

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o stivă alocata inlantuit.

Indicații de rezolvare: Completați următorul program C cu corpul funcțiilor pentru operațiile cu liste.

```
#include <stdio.h>
#include <stdlib.h>
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *crearestiva();

void afiseazastiva(NOD *top);

NOD *push(NOD *top, int info_nou);

NOD *pop(NOD *top);
```

Se completeazăcu instrucțiunile necesare

```
int main()
{
   NOD *top=NULL;
   int  info_nou;
   char c;
   top = crearestiva();
   afiseazastiva(top);

// Inserarea unui element: elem_nou
   do
   {
         printf("Introduceti elementul pe care vreti sa il adaugati in stiva\n");
         scanf("%d", &info_nou);
         top = push(top, info_nou);
      printf("Stiva este acum\n ");
         afiseazastiva(top);
         printf("Mai doriti sa introduceti un element (Y/N)?");
         rewind(stdin);
         c= getchar();
   } while(c=='Y' || c=='y');

//  Eliminarea unui element
        do
        {
     top = pop(top);
     printf("\nStiva actualizata\n ");
     afiseazastiva(top);
     printf("Mai doriti sa stergeti un element (Y/N)?");
     rewind(stdin);
     c= getchar();
   } while(c=='Y' || c=='y');

   getchar();
   getchar();
   return 0;
}
```

Test de autoevaluare 2

1. Scrieti o funcţie C care goleşte o stivă dată de numere întregi alocată înlănţuit şi afişează pe rând toate elementele scoase. Încorporaţi funcţia într-un program C şi testaţi-o.

2. Se da urmatoarea stiva de numere intregi (5 este valoarea din varful stivei):

5
6
8
3

Scrieti secventa de operatii necesare pentru a-l scoate pe 8 din stiva.

3. Ce face următorul algoritm?

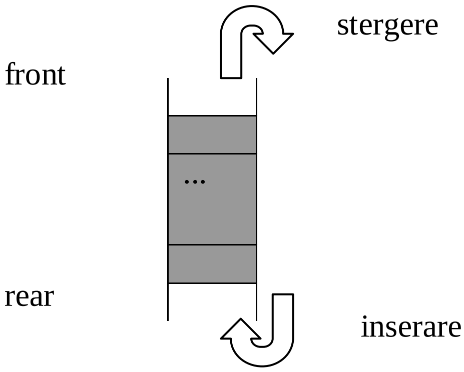
```
if top = NULL
        then UNDERFLOW
        else elem_sters = top -> info
             top = top -> link
endif
```

4. Transcrieţi algoritmul de mai sus într-un fragment de cod scris în limbajul C.

Lecția 3. Cozi

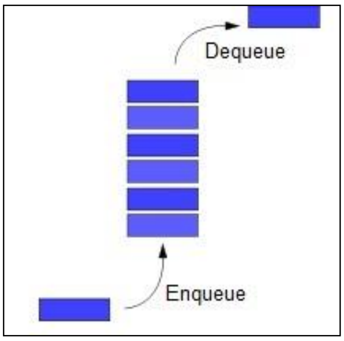
> O **coada** este o lista liniara in care stergerea si accesarea unui element se pot face numai pe la un capat al cozii, numit <u>front,</u> iar inserarea se face la celalalt capat al cozii, numit <u>rear.</u>

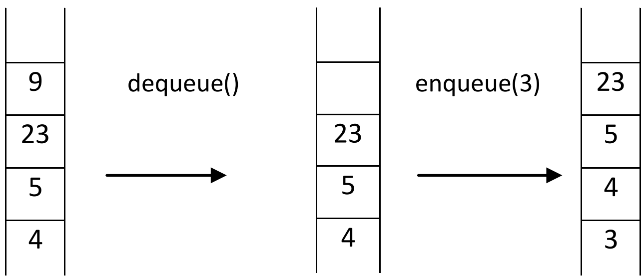
Se poate face o analogie intre o coada folosita in programare si, de exemplu, o coada pentru tiparirea mai multor fisiere text. Pentru tiparirea unui nou fisier va trebui sa asteptam pana cand toate fisierele sunt tiparite in ordinea in care comenzile de tiparire au fost efectuate.



Cozile se mai numesc si liste *FIFO (first in/first out)* deoarece primul element care este extras din coada este primul introdus.

> Notăm:
> - inserarea unui element a intr-o coada C: $a \Rightarrow C$
> - stergerea unui element a dintr-o coada C: $C \Rightarrow a$.





Coada originala

> Cele doua operatii se mai numesc si *enqueue* respectiv *dequeue*.

3.1 Alocarea secventiala

In alocarea secventiala, presupunand ca avem o coada definita
T x[N];
unde *N* este o constanta suficient de mare, *T* este un tip de date definit anterior (eventual printr-o definitie de tipul *typedef*), iar elementele cozii sunt in ordine *x[R], x[R+1], ..., x[F]*, cu indicii $0 \le R \le F \le N - 1$,

	x[R]	x[R+1]			x[F]			
...				...		...		

Operatiile permise cu cozi se pot descrie astfel:

- **accesarea/ modificarea unui element:** numai elementul din vârful cozii poate fi accesat adică elementul *x[F]*

- **inserarea unui element:** *elem_nou* $\Rightarrow$ *C*

Algoritm: Daca R = 0 şi F=N-1 atunci se produce OVERFLOW şi nicio inserare nu este posibila. In caz contrar elementul nou se adaugă la sfârşitul coziii. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if R = 0 and F = N - 1 then OVERFLOW
        else if R > 0 then x[R – 1] = elem_nou
                          R = R –1
                    else for i = F, R, -1
                                 x[i+1] = x[i]
                          endfor
                          x[0] = elem_nou
                          F = F +1
            endif
endif
```

- **ştergerea unui element:** *C* $\Rightarrow$ *elem_sters*

Algoritm: Daca R>F atunci se produce UNDERFLOW adica coada este vidă si nicio ştergere nu este posibilă. In caz contrar elementul din vîrful stivei este şters. Valoarea acestuia este salvată în variabila *elem_sters*. Se actualizeaza numarul de elemente al sirului. Mai precis, algoritmul se scrie:

```
if R > F then UNDERFLOW
               else elem_sters = x[F]
                    F = F -1
        endif
```

Exercițiu rezolvat

Să se scrie o funcție C++ care să implementeze algoritmul de ștergere a unui element dintr-o coadă alocată secvențial. Se cunosc numele cozii (x) și indicii F și R. Funcția returnează valoarea elementului sters.

```
int dequeue(int x[], int R, int F)
{
    if (R >F) {
                    printf("Stergere nereusita:  UNDERFLOW\n");
                    return;
                  }
    elem_sters = x[F];
    F--;
    return elem_sters;
}
```

Lucru individual

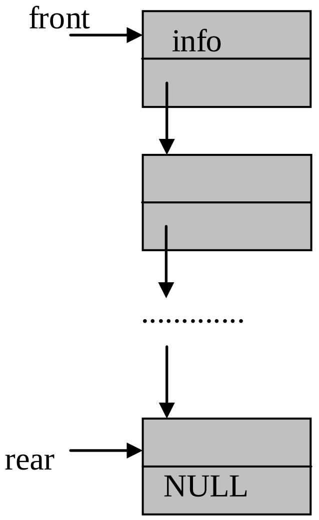
Scrieti un program care implementeaza algoritmii de mai sus pentru o coadă alocata secvential.

Indicații de rezolvare: Ca și în cazul stivei alocate secvențial, este bine să scrieți un program în care operațiile de inserare, stergere, creare și afișare să fie descrise fiecare printr-o funcție, iar în funcția main să se facă doar apelarea acestor funcții după cum se dorește efectuarea uneia sau a alteia dintre operații.

3.2 Alocarea inlantuita

In alocarea inlantuita, folosim aceeasi structura ca si in cazul listei liniare

```
struct nod
{
        T info;
        struct nod *link;
};
typedef struct nod NOD;
```



Notam

- *front* pointer la primul nod al cozii,
- *rear* pointerul la ultimul nod.

*NOD *front, *rear;*

Cu aceste notatii, operatiile cu cozi se pot descrie astfel:

inserarea unui nod nou	**stergerea/accesarea unui nod**

inserarea unui nod nou

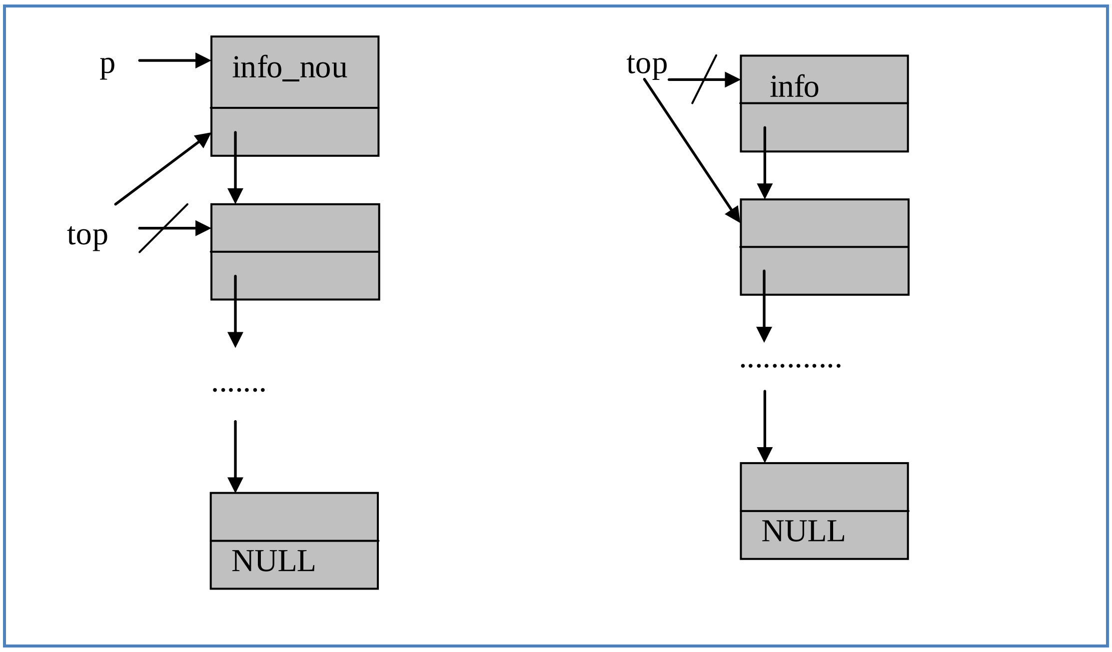
Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod.

```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
if p ≠ NULL then
        p -> link = top
        p -> info = info_nou
        if rear ≠ NULL then
            rear -> link = p
        else front =p
        rear = p
else OVERFLOW
endif
```

stergerea/accesarea unui nod

Algoritm:

```
if front = NULL
   then UNDERFLOW
   else elem_sters = front -> info
        if front=rear then
                front = NULL
                rear=NULL
        else front = front -> link
        endif
endif
```



Exercițiu rezolvat

Să se scrie o funcție C++ care să implementeze algoritmul de inserare a unui nod dintr-o coadă alocată înlănțuit. Se dau pointerii front și rear care se consideră variabile globale. Se dau următoarele definiții:

```
typedef int T;
struct nod
{
        T info;
        nod *link;
};
```

```
void enqueue(int a)
{
        nod *p = new nod;
        if (p != NULL)
            {
                p ->info = a;
                p ->link = NULL;
                if(rear!=NULL) rear->link=p;
                else front=p;
                rear = p;
            }
        else cout << "OVERFLOW"<<endl;
}
```

Memorie disponibilă

Valorile noului nod

coada vidă

Exercițiu rezolvat

Să se scrie o funcție C++ care să returneze valoarea primului nod al unei cozi de numere intregi alocată înlănțuit. Se dau pointerii front și rear care se consideră variabile globale. Se dau următoarele definiții:

```
typedef int T;
struct nod
{
        T info;
        nod *link;
};
```

```
T valfront()
{
        if (front != NULL)
        return front->info;
        else {
                cout << "coada vida"<<endl;
                return -9999;
        }
}
```

Pentru că este o coadă vidă, se returnează o valoare ce nu poate fi în coadă, de exemplu -9999.

Exercițiu rezolvat

Să se scrie o funcție C++ care să returneze true dacă o coadă este vidă și false în caz contrar. Se dau pointerii front și rear care se consideră variabile globale.

```
bool vida()
{

        return front==NULL;
}
```

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o coadă alocata inlantuit.

Indicații de rezolvare: Completați următorul program C++ cu corpul funcțiilor pentru operațiile cu liste.

```
#include <iostream>
using namespace std;

typedef int T;
struct nod
 {
                 T info;
                 nod *link;
 };

 nod *front=NULL, *rear=NULL;

 T valfront(); // returneaza valoarea primului elem din coada
 T valrear(); // returneaza valoarea ultimului elem din coada
 void dequeue(); // sterge un element din coada
 void enqueue(T a); // insereaza elementul a in coada
 bool vida(); // = true coada este vida si =false altfel
 int size(); // returneaza nr de elem din coada fara a le scoate din coada
 void clear(); // goleste  coada
 void input(); // permite introducerea de la tastatura a elem de pus in coada
          // pana cand valoarea tastata nu are formatul dorit (de ex daca
          // se introduc nr intregi si se tasteaza o litera atunci citirea
                          // se opreste)
 void print(); // tipareste componenta cozii


int main()
{
        input();
        enqueue(2);
        enqueue(3);
        print();
        cout << valfront()<<endl;
        dequeue();
        cout << size()<<endl;
        cout << valfront()<<endl;
        print();
        if(vida()) cout << "coada este vida"<<endl;
        else cout <<"coada nu este vida"<<endl;
        clear();
        if(vida()) cout << "coada este vida"<<endl;
        else cout <<"coada nu este vida"<<endl;
        print();
        cout << valfront()<<endl;
         system("PAUSE");
        return 0;
}
```

O posibila functie main.
O puteti modifica dupa
cum doriti ca sa testati
functiile scrise

Test de autoevaluare 3

1. Scrieti o funcţie C care returneaza numărul de elemente din coada fara a le scoate din coada. Încorporaţi funcţia într-un program C şi testaţi-o.

2. Se da urmatoarea coadă de numere intregi (5 este valoarea din varful cozii):

5
6
8
3

- Scrieti secventa de operatii necesare pentru a-l scoate pe 8 din coadă.
- Dacă inițial coada era vidă, scrieti secventa de operatii necesare pentru a se obține coada din figură.

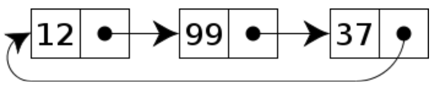
3. Ce face următorul algoritm?

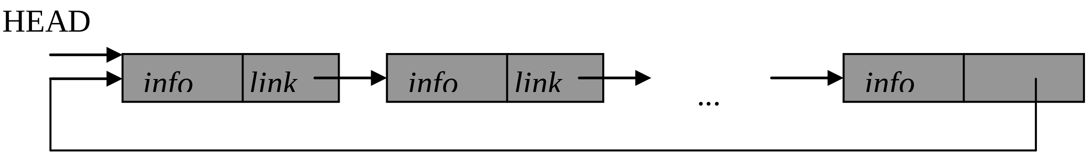
```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
if p ≠ NULL then
        p -> link = top
        p -> info = info_nou
        if rear ≠ NULL then
             rear -> link = p
       else front =p
        rear = p
else OVERFLOW
endif
```

4. Transcrieți algoritmul de mai sus într-un fragment de cod scris în limbajul C.

Lecția 4. Liste circulare

O **lista circulara** este o lista in reprezentare inlantuita care are proprietatea ca ultimul nod puncteaza la primul nod al listei





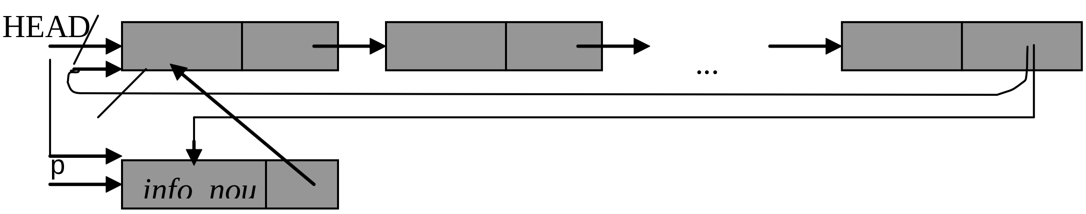
Operatiile permise cu liste circulare se pot descrie astfel:

– **accesarea/ modificarea unui element:** se face la fel ca in cazul listei liniare, algoritm descris in Lecția 1.

– **inserarea unui nou nod:**
  - *la inceputul listei (sau la sfarsitul listei*, este acelasi lucru):

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista, *p* pointer la un nod si *iter* care initial pointeaza la primul nod al listei si va parcurge lista pana cand acesta va pointa la ultimul nod din lista.

```
Aloca memorie pentru un nod nou. Returneaza p, un
pointer la noul nod.
        if p ≠ NULL then
                p -> link = HEAD
                p -> info = info_nou
                iter = HEAD
                while (iter -> link != HEAD)
                        iter = iter -> link
                endwhile
                iter -> link =p
                HEAD = p
        else OVERFLOW
        endif
```



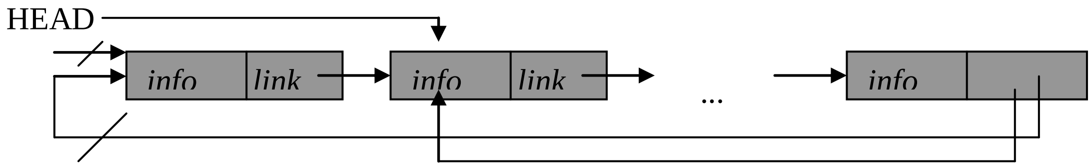
- *dupa un nod dat:* se face la fel ca in cazul listei liniare, algoritm descris in Lecția 1.

– **eliminarea unui nod dat din lista:**

- *eliminarea primului nod*

Algoritm: Folosim variabila *iter* care initial pointeaza la primul nod al listei si va parcurge lista pana cand acesta va pointa la ultimul nod din lista.

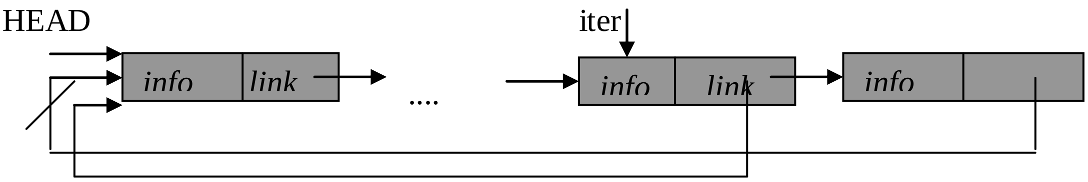
```
elem_sters = HEAD -> info
iter = HEAD
while (iter -> link != HEAD)
            iter = iter -> link
endwhile
iter -> link = HEAD -> link
            HEAD = HEAD -> link
```



- *eliminarea ultimului nod:*

Algoritm: Folosim variabila *iter* care initial pointeaza la primul nod al listei si va parcurge lista pana cand acesta va pointa la penultimul nod din lista.

```
iter = HEAD
            while (iter ≠ NULL and iter -> link  ≠ HEAD and (iter -> link) ->link ≠
HEAD)
            iter = iter -> link
            endwhile
            if iter = NULL then UNDERFLOW
                                    else if iter -> link = HEAD
                                         then elem_sters = iter ->info
                                          else elem_sters = (iter - >link) ->info
                                              iter -> link = HEAD
                                         endif
            endif
```



- eliminarea altui nod din lista se face la fel ca in cazul listei liniare, algoritm descris in Lecția 1.

Exercițiu rezolvat

Să se scrie o funcție C care să implementeze algoritmul de ștergere a primului nod dintr-o listă circulară. Se dă pointerul head la începutul listei. Se dau următoarele definiții:

```
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *eliminare_primul_circular(NOD *head )
{
   NOD *iter;
   if (head == NULL)
   {
       printf ("Eroare. Lista este vida. UNDERFLOW.");
       return NULL;
       }
   printf("Valoarea elementului sters este: %d", head->info);
   iter = head;
   while (iter -> link != head)
          iter = iter -> link;
   iter ->link = head -> link;
   head = head -> link;
   return head;
}
```

Se cauta pointerul la ultimul nod

Returneaza pointer la noua listă

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o lista circulară.

Test de autoevaluare 4

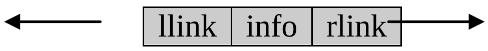
1. Ce face următorul algoritm?

```
iter = HEAD
while (iter ≠ NULL and iter -> link  ≠ HEAD and (iter -> link) ->link ≠ HEAD)
iter = iter -> link
endwhile
if iter = NULL then UNDERFLOW
                    else if iter -> link = HEAD
                         then elem_sters = iter ->info
                          else elem_sters = (iter - >link) ->info
                               iter -> link = HEAD
                         endif
endif
```

2. Transcrieți algoritmul de mai sus într-un fragment de cod scris în limbajul C.

3. Scrieți o funcție C++ care crează o listă circular folosind apelarea repetată a unei functii inserare cu prototipul **NOD *inserare(NOD *head);** ce insereaza un nod la începutul unei liste circulare.

Lecția 5. Liste dublu inlantuite



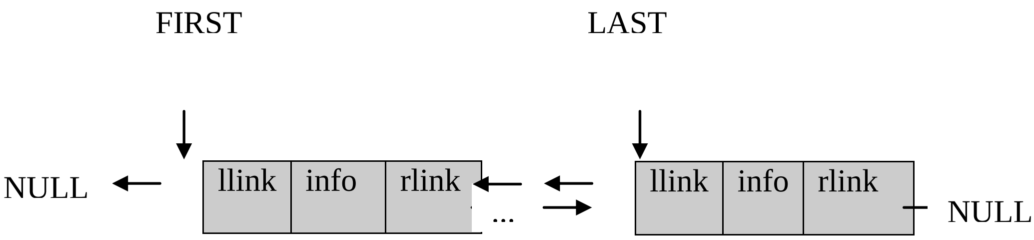
O listă dublu înlănțuită este o listă alocată înlănțuit în care fiecare nod are, pe lângă informația conținută și doi pointeri:

- pointer la nodul următor
- pointer la nodul precedent.

Orice lista dublu inlantuita va avea doi pointeri

- FIRST, care pointeaza la primul nod
- LAST care pointeaza la ultimul nod.

Deasemenea, se fac urmatoarele setari FIRST->llink=NULL si LAST->rlink= NULL.



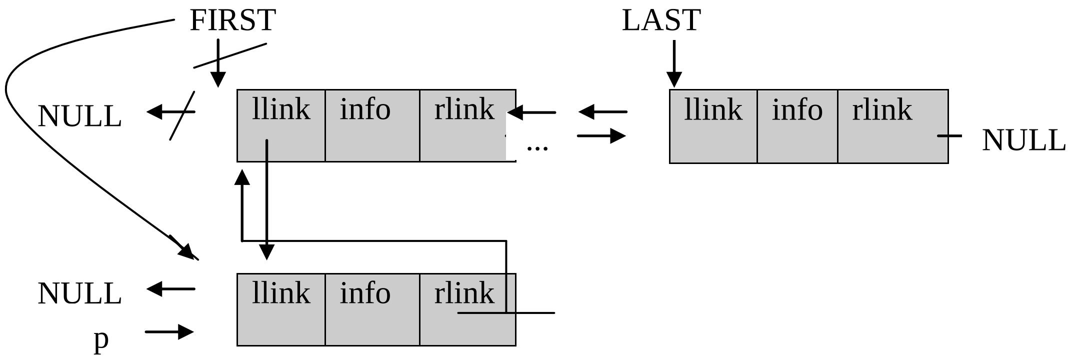
Operatiile cu liste dublu inlantuite se pot descrie astfel:

– **accesarea/ modificarea unui element** ce contine o valoare data se face ca si in cazul listei liniare simplu inlantuite, parcurgandu-se lista fie de la inceput la sfarsit, fie de la sfarsit la inceput.

– **inserarea unui nou nod**
  - *la inceputul listei:*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod.

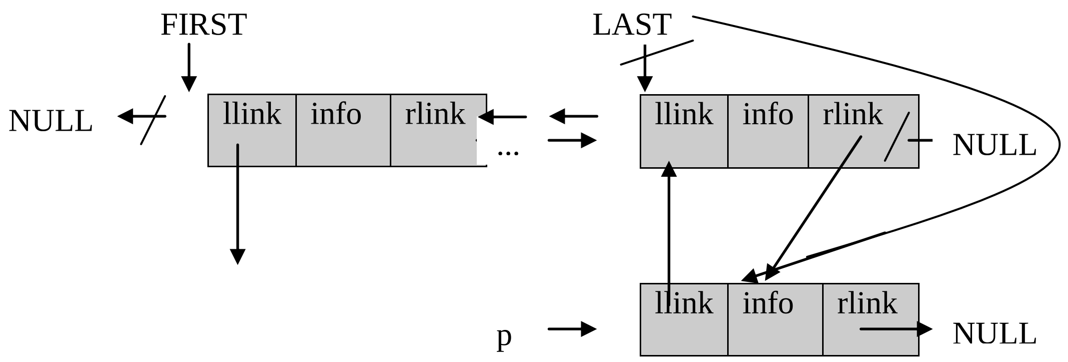
```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
        if p ≠ NULL then
                        p -> llink = NULL
                        p -> rlink = FIRST
                        p -> info = info_nou
                        FIRST -> llink = p
                        FIRST = p
                else OVERFLOW
        endif
```



- *la sfarsitul listei*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista, *p* pointer la un nod

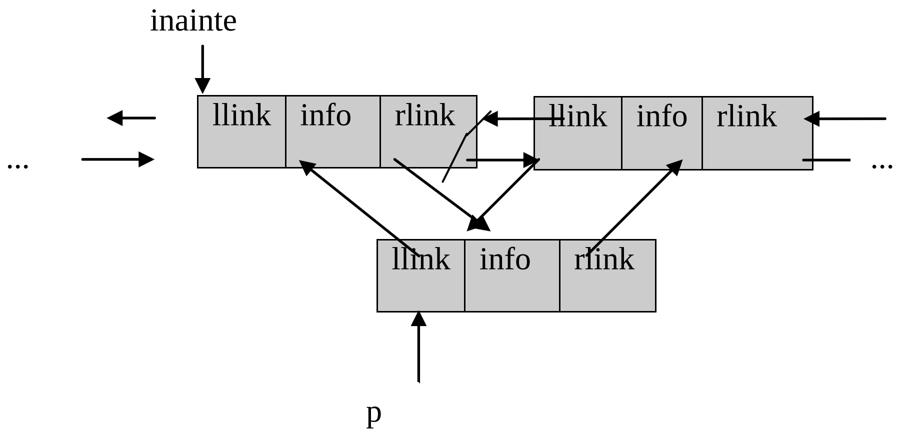
```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
if p ≠ NULL then
                p -> rlink = NULL
                p -> info = info_nou
                p -> llink = LAST
                LAST -> rlink =p
                LAST=p
        else OVERFLOW
endif
```



- *dupa un nod dat:*

Algoritm: Folosim variabilele *info_nou* ce contine valoarea informatiei nodului ce trebuie introdus in lista si *p* pointer la un nod si *inainte* pointer la nodul dupa care se doreste introducerea noului nod.

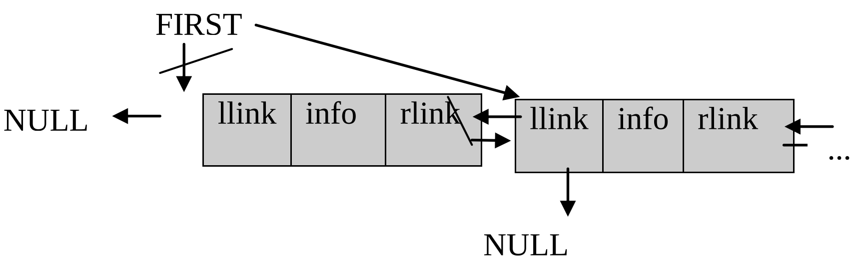
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
if p ≠ NULL then
p -> llink = inainte
p -> rlink = inainte -> rlink
inainte -> rlink = p
(p -> rlink) -> llink = p
p -> info = info_nou
else OVERFLOW
endif



– **eliminarea unui nod**
   ○ *eliminarea primului nod:*

Algoritm:

elem_sters = FIRST -> info
FIRST = FIRST -> rlink
if FIRST ≠ NULL then FIRST ->llink = NULL
else LAST= NULL



   ○ *eliminarea ultimului nod:*

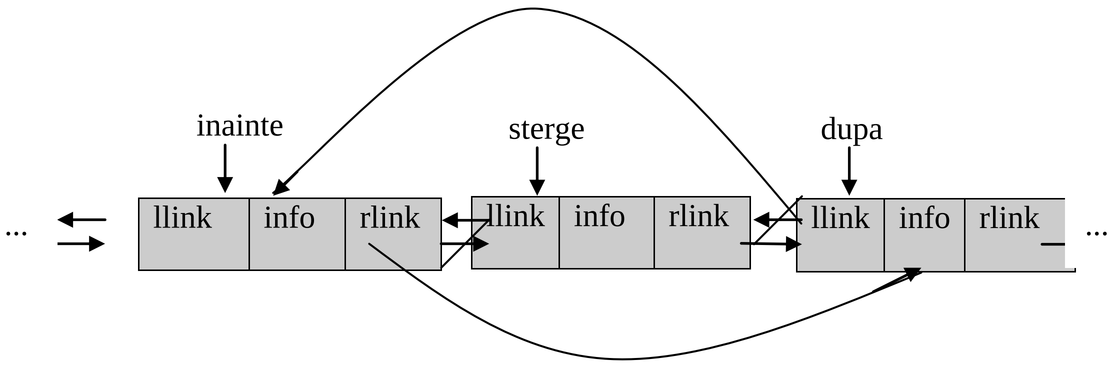
Algoritm:

elem_sters = LAST -> info
LAST = LAST -> llink
if LAST ≠ NULL then LAST ->rlink = NULL else FIRST = NULL

- *eliminarea unui nod dat din lista:*

Algoritm: Folosim variabilele *sterge*, un pointer la nodul care trebuie eliminat din lista, *inainte*, pointer la nodul dinainte si *dupa*, pointer la nodul dupa nodul ce trebuie sters.

```
elem_sters = sterge -> info
inainte = sterge -> llink
dupa = sterge -> rlink
dupa ->llink = inainte
inainte ->rlink = dupa
```



Să se scrie o funcție C care să implementeze algoritmul de ștergere a primului nod dintr-o listă dublu înlănțuită. Se dau pointerii first și last. Se dau următoarele definiții:

```
struct nod
{
        int info;
        struct nod *link;
};
typedef struct nod NOD;
```

```
NOD *eliminare_primul_dbinlantuita(NOD *first, NOD* &last )
{ if (first == NULL)
   {
      printf ("Eroare. Lista este vida. UNDERFLOW."); return NULL;
     }
   printf("Valoarea elementului sters este: %d", first->info);
   first = first -> rlink;
   if (first != NULL ) first ->llink = NULL;
         else last= NULL;
   return first;
}
```

Returneaza pointer la noua listă

Lucru individual

Scrieti un program care implementeaza algoritmii de mai sus pentru o lista dublu înlănțuită.

Test de autoevaluare

5

1. Ce face următorul algoritm?

```
Aloca memorie pentru un nod nou. Returneaza p, un pointer la noul nod.
        if p ≠ NULL then
                p -> llink = inainte
                p -> rlink = inainte -> rlink
                inainte -> rlink = p
                (p -> rlink ) -> llink = p
                p -> info = info_nou
        else OVERFLOW
        endif
```

2. Transcrieți algoritmul de mai sus într-un fragment de cod scris în limbajul C++.

3. Scrieți o funcție C++ care tipărește o listă dublu înlănțuită în două feluri de la început la sfârșit și de la sfârșit la început.

Probleme propuse

1. Să se implementeze o listă dublu înlănţuită ale cărei elemente să fie studenţii unei facultăţi. Programul va conţine funcţii pentru:
    - crearea listei vide
    - afişarea elementelor listei
    - căutarea unui student în listă
    - adăugarea unui student la lista (la început, la sfârşit, după un anumit nod specificat).
2. Intr-o gara se considera un tren de marfa ale carui vagoane sunt inventariate intr-o lista, in ordinea vagoanelor. Lista contine, pentru fiecare vagon, urmatoarele date:

    - codul vagonului (9 cifre);
    - codul continutului vagonului (9 cifre);
    - adresa expeditorului (4 cifre);
    - adresa destinatarului (4 cifre).

Deoarece in gara se inverseaza pozitia vagoanelor, se cere listarea datelor despre vagoanele respective in noua lor ordine.

3. Se consideră o masă circulară la care sunt aşezaţi copii şi fie *n* un număr întreg pozitiv. Pornind de la un anumit copil ei încep să numere pe rând. Atunci când unul din copii ajunge numărul *n* acesta se ridică de la masă şi se reia numărătoarea pornind de la 1. Jocul se opreşte atunci când rămâne un singur copil. Realizaţi un program ce utilizează o listă circulară pentru a afişa ordinea în care copiii se ridică de la masă.
4. Să scrie un program care să calculeze produsul a două polinoame utilizându-se pentru reprezentarea unui polinom o listă simplu înlănţuită.
5. Scrieti un program care evalueaza o expresie aritmetica in care apar operatorii +, -, / si * iar operanzii sunt constante.

UNITATEA DE ÎNVĂȚARE 2

Structuri de date neliniare

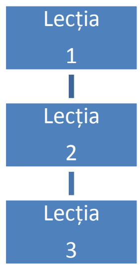
Obiective urmărite:

La sfârşitul parcurgerii acestei UI, studenţii

– vor şti să identifice şi să clasifice principalele tipuri de structuri de date neliniare
– vor şti să interpreteze modul de organizare a datelor neliniare
– vor cunoaşte modurile de reprezentare a structurilor de date neliniare
– vor cunoaşte operaţiile specifice fiecărui tip de structură de date neliniară
– vor şti să implementeze într-un limbaj de programare principalele tipuri de structuri de date neliniare
– vor cunoaşte cele două moduri de reprezentare a structurilor de date neliniare (dinamic şi static)
– vor înţelege rolul structurilor alocate dinamic şi avantajele, precum în raport cu cele statice
– vor şti să identifice tipul de structură de date neliniară care poate fi folosită pentru organizarea datelor şi informaţiilor în vederea rezolvării unor probleme practice

Ghid de abordare a studiului:

Timpul mediu necesar pentru parcurgerea si asimilarea unitatii de invatare: 4h.
Lecțiile se vor parcurge în ordinea sugerată de diagramă.



Rezumat

În această UI sunt prezentate principalele tipuri de structuri de date neliniare: grafuri și arbori. Se accentuează importanța arborilor binari în informatică. Se face o analiză a reprezentării acestor structuri de date și o prezentare detaliată a operațiilor corespunzătoare acestor tipuri de date. Sunt discutate implementările în limbajul de programare C/C++ a acestor structuri de date. Sunt analizate problemele practice care pot fi rezolvate prin folosirea acestui mod liniar de organizare a datelor problemelor.

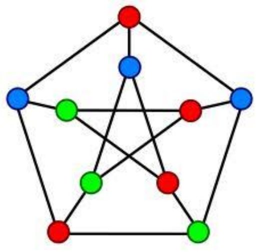
Cuvinte cheie

arbori, grafuri, graf neorientat, graf orientat, arbore binar, drum, ciclu, înălţimea unui arbore binar, noduri terminale, nod rădăcină, matrice de adiacenţă, listă de adiacenţă, noduri, muchii, conex, tare conex, nod tată, nod fiu, fiu stâng, fiu drept, subarbore, subarbore stâng, subarbore drept

Lecția 1. Grafuri

2.1 Definiții

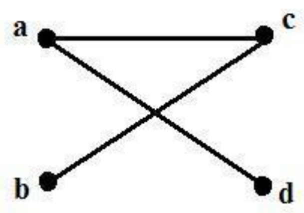
Se numeste <u>graf neorientat</u> o structura G = (V, E) in care V este o multime finita, iar E o multime de perechi neordonate de elemente din V. Mai exact, $E \subseteq \{\{a, b\} \mid a, b \in V, a \neq b\}$.



Multimea V se numeste <u>multimea varfurilor</u> grafului G, iar E <u>multimea muchiilor</u>. Notam muchiile sub forma [a, b], unde $a,b \in V$.

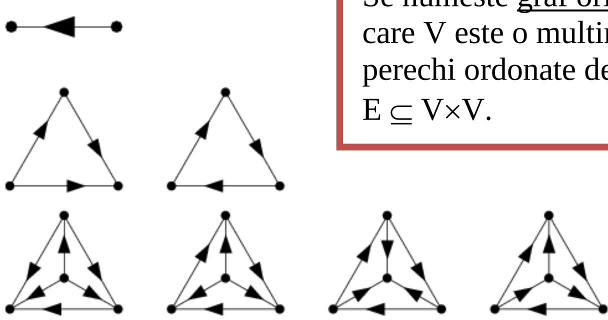
In reprezentare grafica, varfurile se reprezinta prin cercuri, iar muchiile prin linii ce unesc varfurile.

Exemplu:



Graful neorientat $G_1 = (V_1, E_1)$ unde

- $V_1 = \{a, b, c, d\}$
- $E_1 = \{[a, c], [b, c], [a, d]\}$


[1]

Se numeste <u>graf orientat</u> o structura G = (V, E) in care V este o multime finita, iar E o multime de perechi ordonate de elemente din V. Mai exact $E \subseteq V \times V$.

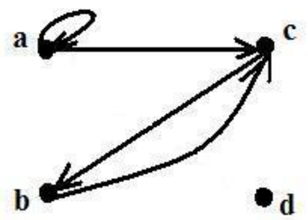
Multimea V se numeste <u>multimea varfurilor</u> grafului G, iar E <u>multimea arcelor</u>.

[1] Imagine preluată din pagina Weisstein, Eric W. "Tournament." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/Tournament.html

Daca (a, b) $\in$ E atunci a se numeste extremitatea initiala a arcului, iar b extremitatea finala.

In reprezentare grafica, varfurile se reprezinta prin cercuri, iar arcele prin sageti de la primul varf la cel de al doilea.

Exemplu:



Graful orientat $G_2 = (V_2, E_2)$
unde

- $V_2 = \{a, b, c, d\}$
- $E_2 = \{(a, c), (c, b), (b, c), (a, a)\}$

Doua varfuri ale unui graf neorientat intre care exista o muchie se numesc adiacente.
Daca a, b sunt doua varfuri ale unui graf orientat astfel incat exista un arc de la a la b atunci spunem ca b este adiacent varfului a.

Gradul unui varf x al unui graf neorientat, notat d(x), este dat de numarul de varfuri adiacente cu el. Un varf de grad zero se numeste izolat.
Pentru grafuri orientate se definesc:

- gradul de intrare al unui varf, notat $d^-(x)$, ca fiind numarul de arcuri care au ca extremitate finala varful respectiv
- gradul de iesire, notat $d^+(x)$, se defineste ca numarul de arcuri ce au ca extremitate initiala varful respectiv.

Exemple:

Graful G_1

x = varf	d(x) = grad
a	2
b	1
c	2
d	1

Graful G_2

x = varf	$d^-(x)$ = grad intrare	$d^+(x)$= grad iesire
a	1	2
b	1	1
c	2	1
d	0	0

Dat un graf G = (V, E), se numeste drum de la varful x la un varf y, o succesiune de varfuri, pe care o notam $[x, x_1, x_2, ..., x_p , y]$ daca G este neorientat si $(x, x_1, x_2, ..., x_p , y)$ daca G este orientat, astfel incat: $[x, x_1]$, $[x_1, x_2]$, ..., $[x_p , y]$ sunt muchii (daca G este neorientat) sau (x, x_1), (x_1, x_2), ..., (x_p , y) sunt arce (daca G este orientat).

Lungimea unui drum este data de numarul muchiilor (arcelor) drumului. Un drum se numeste elementar daca $x, x_1, x_2, ..., x_p$ sunt distincte si $x_1, x_2, ..., x_p , y$ sunt de asemenea distincte.

Un drum intr-un graf neorientat $[x, x_1, x_2, ..., x_p , y]$ pentru care $x = y$, drumul contine cel putin o muchie si toate muchiile sunt distincte se numeste <u>ciclu</u>.

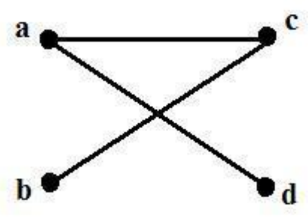
Un drum intr-un graf orientat $(x, x_1, x_2, ..., x_p , y)$ pentru care $x = y$, drumul contine cel putin un arc si toate arcele pe care le contine sunt distincte se numeste <u>circuit</u>. Un graf fara cicli se numeste <u>aciclic</u>.

Exemple:

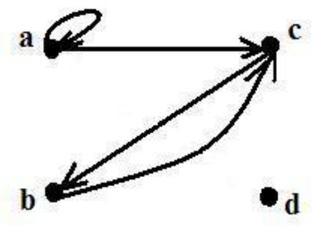
- [d, a, c, b] este un drum elementar de la d la b de lungime 3 in G_1
- (a, a, c) este un drum de la a la c de lungime 2 in G_2
- (a, c, b, c) este un drum de la a la c de lungime 3 in G_2
- (a, c, b) este un drum elementar de la a la b de lungime 2 in G_2
- nu exista nici un drum de la d la orice alt nod in G_2
- G_1 este aciclic
- G_2 are un singur ciclu: (b,c,b) de lungime 2.

Un graf neorientat se numeste <u>conex</u> daca intre orice doua varfuri ale sale exista un drum. <u>Componentele conexe</u> ale unui graf sunt clasele de echivalenta ale varfurilor aflate in relatia "exista drum intre".

Exemplu:



Graful G_1

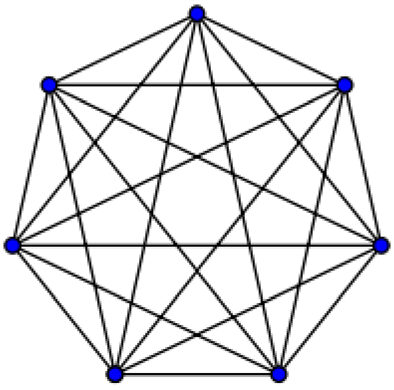


Graful G_2

- G_1 este conex in timp ce G_2 nu este conex, dar are 2 componente conexe.

Un graf neorientat se numeste <u>complet</u> daca exista muchie intre oricare doua varfuri.

Exemplu:

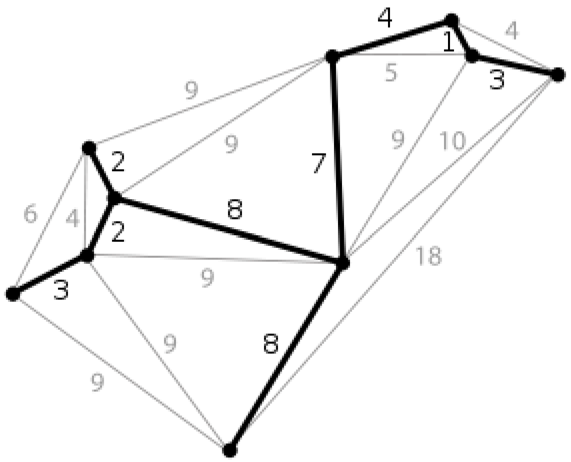


Graf complet cu 7 varfuri.

Un graf $G' = (V', E')$ este <u>subgraf</u> al grafului $G = (V, E)$ daca $V' \subseteq V$ si $E' \subseteq E$.
Un graf $G' = (V', E')$ este <u>graf partial</u> al grafului $G = (V, E)$ daca $V' = V$ si $E' \subseteq E$.

Se numeste <u>graf ponderat</u> (sau cu ponderi) un graf G = (V, E) pentru care fiecarei muchii i se asociaza o pondere data de obicei de o functie $w: E \rightarrow \mathbf{R}$.

Exemplu:



Graf ponderat în care se pune în evidență subarborele de cost minim.

2.2 Reprezentarea grafurilor

Reprezentarea grafurilor, orientate sau neorientate se poate face in doua feluri:

1) Matricea de adiacenta

Fie G = (V, E) un graf dat si n = numarul de varfuri pe care le vom nota 1, 2, ..., n.
Matricea de adiacenta este o matrice $A= (a_{ij})$, $1 \le i, j \le n$ definita astfel:

$$a_{ij} = 1 \text{ daca } (i, j) \in E \text{ si } 0 \text{ in caz contrar}$$

Exemple:

Matricile de adiacenta ale grafurilor de mai sus sunt:

(Pentru ambele grafuri vom nota varfurile: a↔ 1, b↔ 2, c ↔3 si d↔ 4.)

Matricea de adiacenta a lui G_1

	1	2	3	4
1	0	0	1	1
2	0	0	1	0
3	1	1	0	0
4	1	0	0	0

Matricea de adiacenta a lui G_2

	1	2	3	4
1	1	0	1	0
2	0	0	1	0
3	0	1	0	0
4	0	0	0	0

Observati ca matricea de adiacenta a unui graf neorientat este simetrica.

Folosirea acestui mod de reprezentare a unui graf prezinta avantaje si dezavantaje.

- Avantajele sunt date de faptul ca se programeaza mai usor.
- Dezavantajele ar fi in primul rand folosirea unui spatiu mare de memorie, in special in cazul in care numarul de varfuri este mare. De asemenea, de multe ori aceasta matrice este rara (adică conține multe elemente nule) și practic se folosește multă memorie pentru foarte puține date.

Exercițiu rezolvat

Cum arată matricea de adiacență a unui graf neorientat complet cu n vârfuri?

Rezolvare

Fie G=(V, E) un graf complet cu n varfuri numerotate 1, 2,..., n. Deoarece într-un graf complet există muchie între orice două vârfuri, avem $a_{ij} = 1$ pentru orice două noduri $i \neq j$ cu elementele de pe diagonala 0.

Lucru individual

Scrieti o funcție care adauga o muchie dintr-un graf neorientat reprezentat prin matricea de adiacență.

Indicații de rezolvare: Graful poate fi introdus prin citirea de la tastatura a matricii de adiacenta. Se citește numărul de noduri și apoi matricea. Funcția va avea drept parametrii cele două varfuri ale muchiei de adăugat. Se modifică valoarea din matrice corespunzătoare muchiei din 0 în 1.

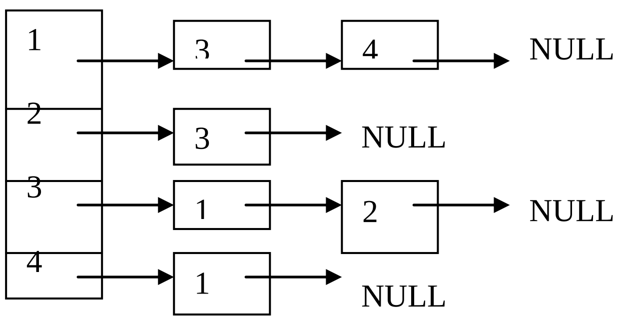
2) *Liste de adiacenta*

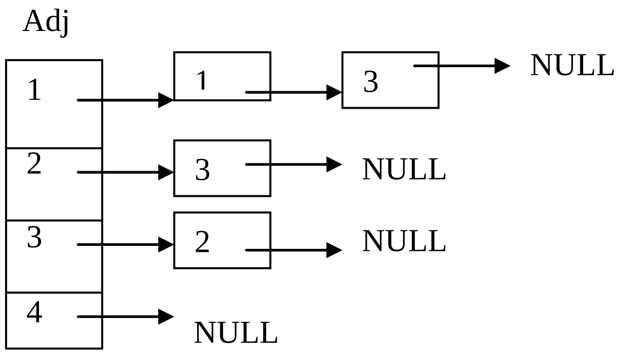
Pentru fiecare varf asociem lista varfurilor adiacente cu varful dat, numita lista de adiacenta. In aceasta reprezentare, pentru un graf dat G = (V, E) avem un sir Adj cu n elemente unde n = numarul de varfuri pe care le vom nota 1, 2, ..., n astfel incat Adj[i] contine toate varfurile j pentru care (i, j) ∈ E. De obicei, varfurile in fiecare lista de adiacenta sunt sortate intr-o anumita ordine, dar acest lucru nu este obligatoriu.

Exemple:

Listele de adiacenta ale celor doua grafuri de mai sus sunt:
(Pentru ambele grafuri vom nota varfurile: a↔ 1, b↔ 2, c ↔3 si d↔ 4.)

Adj

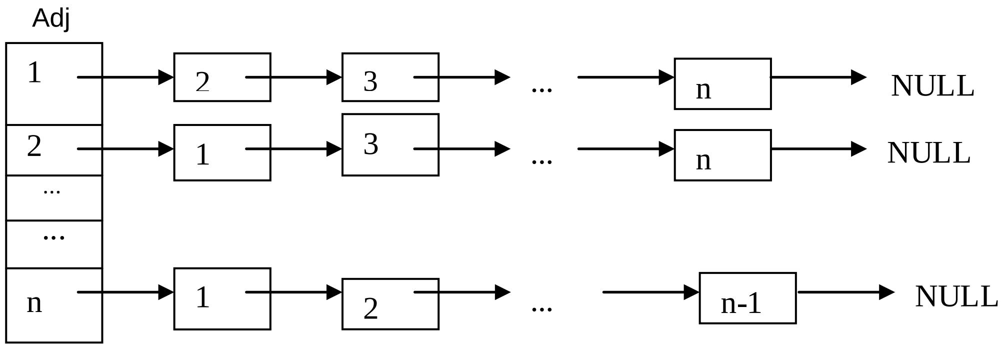




Exercițiu rezolvat

Cum arată listele de adiacență ale unui graf neorientat complet cu n vârfuri?

Rezolvare



Lucru individual

Scrieti o funcție care adauga o muchie dintr-un graf neorientat reprezentat prin liste de adiacență.

Indicații de rezolvare: Se citește numărul de noduri și se creează pentru fiecare nod lista de adiacență. Funcția va avea drept parametrii variabilele i și j, cele două varfuri ale muchiei de adăugat. În lista lui i se va insera un nod cu valoare j, păstrându-se ordinea crescătoare a valorii nodurilor în listă, iar în lista lui j se va insera un nod cu valoare i, păstrându-se deasemenea ordinea crescătoare a valorii nodurilor în listă.

Operatiile cu grafuri se pot descrie astfel, indiferent de metoda de reprezentare:

- **crearea unui graf (identificarea nodurilor și muchiilor)**
- **afișarea părților componente ale unui graf**
- **inserarea unui nod** (împreună cu muchiile corespunzătoare)
- **ștergerea unui nod** (împreună cu muchiile corespunzătoare)

- **inserarea unei muchii**
- **ștergerea unei muchii**

Lucru individual

Scrieti un program care implementează aceste operații pentru un graf neorientat reprezentat prin matricea de adiacență.

Indicații de rezolvare: Este bine să scrieți un program în care operațiile să fie descrise fiecare printr-o funcție, iar în funcția main să se facă doar apelarea acestor funcții după cum se dorește efectuarea uneia sau a alteia dintre operații. Pentru aceasta puteti să completați următorul program C cu corpul funcțiilor pentru operații.

```
#include <stdio.h>
#define N 100

void citestegraf(int x[][N], int n);
void afiseaza(int x[][N], int n);
void adauga_muchie(int a[][N], int vf1, int vf2);
void sterge_muchie(int a[][N], int vf1, int vf2);
void adauga_nod(int a[][N], int &n);
void sterge_nod(int a[][N], int &n);

void actualizeaza_matrice(int a[][N], int &n, int vf1);

int main()
{
   int a[N][N], n;
   char c;
   int gata;
   int vf1, vf2;
   printf("Introduceti numarul de varfuri: ");
   scanf("%d", &n);
   citestegraf(a, n);
   afiseaza(a,n);

//Adaugarea unei muchii
   printf("\nIntroduceti muchia pe care doriti sa o adaugati: ");
   rewind(stdin);
   scanf("%c%d%c%d%c", &c,&vf1,&c, &vf2, &c);
   adauga_muchie(a,vf1, vf2);
   afiseaza(a,n);
```

Functia actualizeaza _ matrice sterge din matricea de adiacenta linia și coloana corespunzătoare nodului care este sters .

```
//  Stergerea unei muchii
    printf("\nIntroduceti muchia pe care doriti sa o stergeti: ");
    rewind(stdin);
    scanf("%c%d%c%d%c", &c,&vf1,&c, &vf2, &c);
    sterge_muchie(a, vf1, vf2);
    afiseaza(a,n);

   // Adaugarea unui nod
    printf("Se adauga un nou nod.........");
    printf("introduceti muchiile corespunzatoare noului nod");
    gata =0;
    while (!gata)
    {
    rewind(stdin);
    scanf("%c%d%c%d%c", &c,&vf1,&c, &
    adauga_muchie(a,vf1, vf2);
    printf("gata?(1/0)");
    scanf("%d", &gata);
    }

    // stergerea unui nod
      printf("Se sterge un nod.........");
      printf("nodul de sters este:";
      scanf("%d", &vf1);
      for(i=0; i<n; i++)
      if (a[i][vf1]==0) sterge_muchie(a, vf1, i);
      actualizeaza_matrice(a,n,vf1);

    getchar();
    getchar();
    return 0;
}
```

Se adauga una câte una muchiile noului nod.

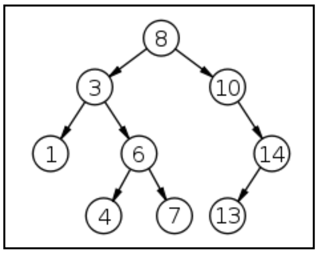
Cand s-au terminat de introdus toate muchiile se introduce valoarea 1 pentru variabila gata.

Lucru individual

Scrieti un program care implementează aceste operații pentru un graf neorientat reprezentat prin liste de adiacență.

Indicații de rezolvare: Este bine să scrieți un program în care operațiile să fie descrise fiecare printr-o funcție, iar în funcția main să se facă doar apelarea acestor funcții după cum se dorește efectuarea uneia sau a alteia dintre operații. Pentru aceasta puteti să completați următorul program C cu corpul funcțiilor pentru operații. Atenție la construcția sirului Adj. El va conține n liste, unde n este numărul de noduri ale grafului.

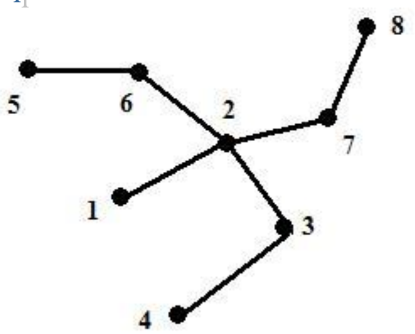
Lecția 2. Arbori

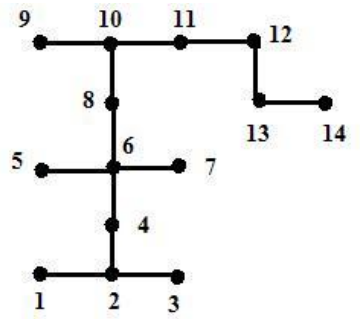


2.1 Notiuni generale

Un graf neorientat conex fara cicluri se numeste <u>arbore</u>.

Exemple:





Arbore 1 Arbore 2

TEOREMA DE CARACTERIZARE A ARBORILOR:

Daca G este un graf neorientat cu n = numar de varfuri, n ≥ 3 atunci urmatoarele conditii sunt echivalente:

i. ***G este arbore;***
ii. ***G este minimal conex (G este conex dar daca este eliminata orice muchie a grafului graful obtinut nu mai este conex);***
iii. ***G este fara cicluri maximal (Intre orice doua varfuri distincte exista exact un singur drum elementar);***
iv. ***G nu are cicluri si are n - 1 muchii;***
v. ***G este conex si are n - 1 muchii.***

Pentru demostratia acestei teoreme puteti consulta Ioan Tomescu, *Data Structures*, Editura Universitatii din Bucuresti, 1997.

<u>Corolar:</u> ***Un arbore cu n varfuri are n - 1 muchii.***

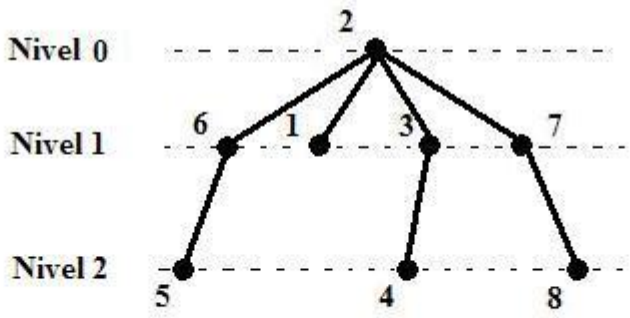
Definitia de mai sus a notiunii de arbore este cea folosita, in literatura de specialitate, in teoria grafurilor. Un tip special de arbore il constituie un arbore, o structura arborescenta, similara cu un arbore din natura sau cu un arbore genealogic, in care se pune in evidenta un nod, numit <u>radacina.</u> Acesta este tipul de arbore folosit in algoritmii computationali si in continuare vom folosi aceasta definitie a notiunii de arbore. (In teoria grafurilor acest tip de arbore se numeste arbore cu radacina).

Deci putem spune ca un arbore este o multime de noduri legate intre ele prin muchii ce indica relatiile dintre noduri, relatii de tip tata-fiu, similare arborilor genealogici.

In informatica arborii sunt vizualizati cu radacina in sus si frunzele in jos. Nodurile sunt aranjate pe nivele. Pe nivelul 0 se afla un singur nod, radacina. Nodurile fiecarui nivel al aborelui sunt fii nodurilor nivelului precedent. Un nod care are fii se numeste tata. Fii cu acelasi tata se numesc frati. Nodurile care nu au fii se mai numesc frunze sau noduri terminale, iar muchiile dintre noduri, ramuri.

Exemplu:

In figura de mai sus, in cazul aborelui 1, daca alegem 2 ca fiind radacina, reprezentarea arborelui pe nivele este:



- Nodul 2 este tatal nodurilor 6, 1, 3, si 7;
- 5 este fiul lui 6;
- 4 este fiul lui 3;
- 8 este fiul lui 7.
- Nodurile 5, 4, 8, si 1 nu au nici un fiu.
- Nodurile 6, 1 , 3 si 7 sunt frati.
- Nodurile 6, 1 , 3 si 7 sunt urmasii lui 2.
- nodurile 5, 4 si 8 sunt urmasii lui 2
- nodul 2 este stramosul tuturor nodurilor (mai putin el insusi), 2 fiind radacina raborelui.

In general, un nod al unui arbore poate avea un numar arbitrar de fii. Daca orice nod al unui arbore nu are mai mult de n fii atunci arborele se numeste arbore n-ar. Un arbore in care orice nod nu are mai mult de 2 fii se numeste arbore binar. Despre acest tip de arbori vom vorbi in capitolul urmator.

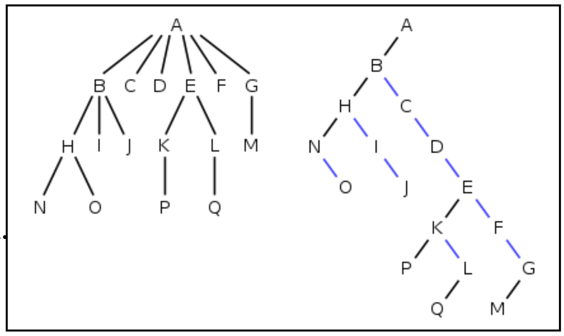
Observati ca intre orice nod si radacina exista exact un singur drum.

Se numeste inaltime a unui arbore lungimea celui mai lung drum de la radacina la un nod terminal din arbore. Pentru arborele de mai sus inaltimea este 2.

Lucru individual

Dați exemplu de un arbore cu cel puțin 5 noduri și înălțime 3.
Dați exemplu de un graf care nu este arbore.
Dati exemplu de un arbore binar cu cel puțin 7 noduri.

Lecția 3. Arbori binari



Arborii binari sunt cele mai importante structuri de date neliniare folosite in informatica.

3.1 Notiuni generale

Un **arbore binar** este un arbore in care orice nod are cel mult doi descendenti facandu-se distincatie clara intre descendentul drept si descendentul stang.

Nota

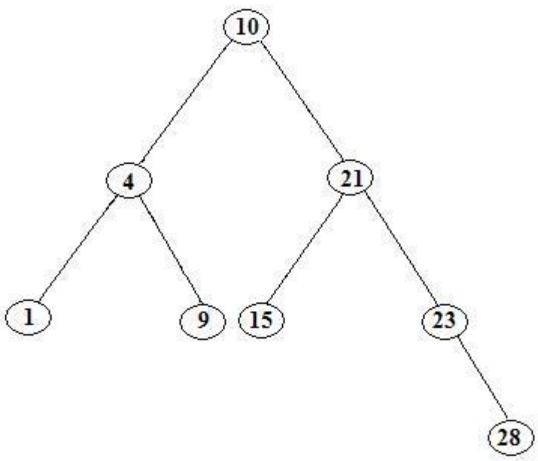
Orice arbore poate fi transformat intr-un arbore binar echivalent. Imaginea de mai sus este un exemplu.

Radacina unui arbore binar are doi subarbori:

- subarborele stang, cel care are drept radacina fiul stang si
- subarborele drept, cel care are ca radacina fiul drept.

Orice subarbore al unui arbore binar este el insusi arbore binar.

Exemplu:



Arbore binar; radacina 10, are drept fiu stang nodul 4, iar fiu drept nodul 21. nodul 21 are subarborele stang

Nota:

Un arbore binar poate fi si vid (adica fara nici un nod).

Un arbore binar pentru care orice nod neterminal are exact doi fii se numeste arbore plin ("full").

3.2 Reprezentare

De obicei, nodurile unui arbore, in particular binar, contin pe langa informatia corespunzatoare si informatii despre cei doi fii stang si drept. In calculator, arborii binari se pot reprezenta in doua moduri.

- **<u>Reprezentarea secventiala</u>**

Pentru fiecare nod al arborelui se precizeaza informatia si descendentii directi ca elemente a trei vector diferiti, INFO[i], ST[i] si DR[i], unde i este indicele asociat unui nod. Cei trei vectori au dimensiunea egala cu numarul de noduri din arbore.

Exemplu:

Pentru arborele de mai sus, daca numerotam nodurile incepand cu nivelul 0, de la stanga la dreapta, obtinem urmatorii vectori, cu conventia ca radacina este nodul 1:

INFO= (10, 4, 21, 1, 9, 15, 23, 28),

ST=(1, 4, 6, 0,0, 0, 0, 0),

DR = (3, 5, 7, 0, 0, 0, 8, 0).

- **<u>Reprezentarea inlantuita</u>**

Pentru fiecare nod al arborelui se precizeaza informatia si descendentii directi ca elemente ale unei structuri definita astfel:

```
struct nodarbore
{
        T info;
        struct nodarbore *stang;
        struct nodarbore *drept;

};
typedef struct nodarbore NODARB;
```

unde

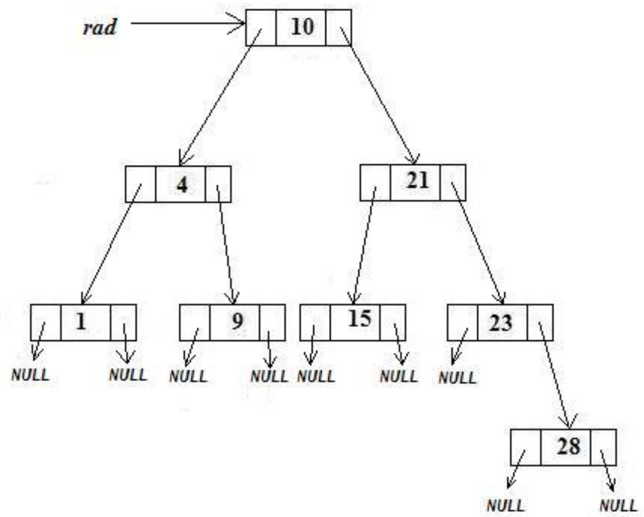
- *T* este presupus definit anterior (eventual printr-o definitie *typedef*),
- *stang* este pointer la subarborele stang al nodului, iar
- *drept* este pointer la subarborele drept al nodului.

Pentru identificarea radacinii arborelui vom defini *NODARB rad;* drept un pointer la radacina arborelui.

Daca unul din subarbori este vid, atunci pointerul la acel subarbore este NULL.

Exemplu:

Pentru arborele de mai sus, reprezentarea inlantuita este:



3.3 Traversare

De multe ori dorim sa accesam ("vizitam") nodurile unei structuri (lista sau arbore). Pentru arbori aceasta accesare, examinare a unui nod, sau, mai exact, examinarea tuturor nodurilor unui arbore se numeste **traversare** si se poate face:

- in **preordine**: intai vizitam radacina arborelui, apoi subarborele stang urmat de subarborele drept
- in **inordine** (simetrica): intai vizitam subarborele stang, , apoi radacina arborelui si apoi subarborele drept
- in **postordine**: intai vizitam subarborele stang si subarborele drept si ultima data radacina arborelui.

Actiunea explicita de vizitare a unui nod depinde de scopul traversarii (de exemplu, aflarea numarului de elemente ale arborelui, gasirea unei valori date in arbore).

Exemplu:

Pentru arborele de mai sus, de exemplu, traversarile sunt:

- preordine: 10, 4, 1, 9, 21, 15, 23, 28
- inordine (simetrica): 1, 4, 9, 10, 15, 21, 23, 28
- postordine: 1, 9, 4, 15, 28, 23, 21.

Algoritmi de traversare a unui arbore binar (recursivi):

```
preordine (rad) /* traversare in preordine a arborelui cu
pointerul la radacina rad */
if rad ≠ NULL then viziteaza nodul pointat de rad
                   call preordine (rad -> stang)
                   call preordine (rad -> drept)
endif
```

```
inordine (rad) /* traversare in inordine a arborelui cu
pointerul la radacina rad */
if rad ≠ NULL then call inordine (rad -> stang)
                   viziteaza nodul pointat de rad
                   call inordine (rad -> drept)
endif
```

```
postordine (rad) /* traversare in postordine a arborelui cu
pointerul la radacina rad */
if rad ≠ NULL then call postordine (rad -> stang)
                   call postordine (rad -> drept)
                   viziteaza nodul pointat de rad

endif
```

Probleme propuse

1. Scrieti un program care printr-o singura traversare a unui arbore binar gaseste elementele maxim si minim din acel arbore.
2. Scrieti un program care calculeaza numarul de noduri terminale ale unui arbore binar dat.
3. Scrieti un program pentru obtinerea listei succesorilor unui nod dintr-un graf reprezentat prin liste de adiacenta.
4. Scrieti un program pentru obtinerea listei predecesorilor unui nod dintr-un graf reprezentat prin liste de adiacenta.
5. Scrieti un program care cauta un element dat intr-un arbore binar.
6. Scrieti un program care insereaza un element dat intr-o pozitie data.
7. Scrieti un program care pentru un arbore binar dat T si un nod dat v, a carui pozitie in arbore se cunoaste, va determina nodul vizitat dupa v in preordine, inordine si respectiv postordine.
8. Scrieti un program care afiseaza un arbore binar dat pe ecran.
9. Scrieti un algoritm nerecursiv pentru traversare in inordine a unui arbore binar si implementati-l.
10. Scrieti un program care tipareste nodurile unui arbore binar, incepand cu cele de pe nivelul 0, de la stanga la dreapta.

UNITATEA DE ÎNVĂȚARE 3

Analiza algoritmilor

Obiective urmărite:

La sfârşitul parcurgerii acestei UI, studenţii

– vor cunoaşte conceptul de eficienta a unui algoritm
– vor cunoaste si inţelege modalitatile de analiza a eficientei algoritmilor
– vor şti să identifice clasele de valori de intrare ale unui algoritm
– vor şti să identifice necesarul de memorie pentru implementarea unui algoritm
– vor putea să precizeze care este timpul de execuție al unui algoritm și să identifice măcar timpul minim și timpul maxim
– vor cunoaşte conceptul de complexitate a algoritmilor şi modul de măsurare a lui

Ghid de abordare a studiului:

Timpul mediu necesar pentru parcurgerea si asimilarea unitatii de invatare: 2h.
Lecțiile se vor parcurge în ordinea sugerată de diagramă.

Lecția 1

Rezumat

În această UI sunt prezentate metodele de analiza a eficientei algoritmilor și se introduce noțiunea de ordin de complexitate a algoritmilor. Se dau câteva exemple de algoritmi pentru care se determină ordinul de complexitate. Se introduc definițiile matematice pentru ca două functii date f(n) si g(n), să avem fie în relația f(n) = O(g(n)) unde n = ordinal de mărime a problemei.

Cuvinte cheie

algoritm, complexitate, analiză, spaţiu de memorie necesar, număr de operaţii, comparaţii, atribuiri, operaţii aritmetice, eficienţa unui algoritm, clase de valori de intrare, timp de execuţie, timp mediu, timp minim, timp maxim, timp mediu, rata de creştere, cel mai bun caz, cel mai rău caz, ordin de complexitate, *O*(n)

Lecția 1. Analiza eficientei algoritmilor

Pentru rezolvarea unei probleme exista de obicei mai multi algoritmi. Cand analizam un algoritm intai trebuie sa ne asiguram ca algoritmul duce la rezolvarea problemei si apoi sa vedem cat de eficient o face. Analiza unui algoritm presupune determinarea timpului necesar de rulare al algoritmului. Acesta nu se masoara in secunde, ci in numarul de operatii pe care le efectueaza. Numarul de operatii este strans legat de timpul efectiv de executie (in secunde) a algoritmului, dar nu acesta nu constituie o modalitate de masurare a eficientei algorimului deoarece un algoritm nu este "mai bun" decat altul doar daca il vom rula pe un calculator mai rapid sau este "mai incet" daca il rulam pe un calculator mai putin performant. Analiza algoritmilor se face independent de calculatorul pe care va fi implementat sau de limbajul in care va fi scris. In schimb, presupune estimarea numarului de operatii efectuate.

Vom determina, de exemplu, numarul de operatii efectuate pentru a inmulti doua matrici de marimi $N \times N$, pentru un algoritm dat care rezolva aceasta problema sau numarul de comparatii necesar sortarii unei liste de N elemente. Analiza algoritmilor ne permite alegerea celui mai bun algoritm pentru o problema data.

Sigur ca eficienta unui algoritm depinde si de spatiul de memorie necesar rularii algoritmului. In cele ce urmeaza insa ne vom concentra in evaluarea numarului de operatii efectuate (si deci a timpului de executie) de catre un algoritm, discutand despre necesarul de memorie numai atunci cand acest lucru devine important.

Un rol important in evaluarea eficientei unui algoritm il au valorile de intrare pentru care se aplica algoritmul. De exemplu, daca se considera problema determinarii celui mai mare element dintr-o lista x de N elemente, putem folosi urmatorul algoritm:

```
max = x[1]
for i = 2, N
        if max < x[i] then max = x[i]
        endif
endfor
write max
```

- Daca lista de intrare este in ordine descrescatoare atunci numarul de instructiuni de atribuire este 1 (cea dinaintea instructiunii for) + N (cele de tipul i=2, i=3, ..., i =N+1), numarul de comparatii este N (cele de tipul $i \leq N$) + (N –1) (cele de tipul max < x[i]).
- Daca lista de intrare este in ordine strict crescatoare atunci numarul de instructiuni de atribuire este 1 (cea dinaintea instructiunii for) + N (cele de tipul i=2, i=3, ..., i =N+1) + N – 1 (cele de tipul max = x[i]) , numarul de comparatii este N (cele de tipul $i \leq N$) + (N – 1) (cele de tipul max < x[i]).

Observati ca numarul de comparatii este acelasi pentru orice tip de lista si ca cele doua cazuri difera prin numarul de instructiuni de atribuire de tipul max = x[i]. Cand se face analiza unui algoritm trebuie sa se ia in considerare toate clasele de valori de intrare posibile. Pentru algoritmul de mai sus acestea sunt:

1. valori de intrare pentru care elementul maxim este in prima pozitie
2. valori de intrare pentru care elementul maxim este in doua pozitie

...

N. valori de intrare pentru care elementul maxim este in a N-a pozitie,

deci exact N clase de valori posibile. Observati ca daca lista de intrare este in ordine descrescatoare atunci ea face parte din prima clasa de valori de intrare, iar daca este in ordine strict crescatoare atunci este face parte din ultima clasa. Observati, de asemenea ca algoritmul determina primul maxim in cazul in care exista doua sau mai multe elemente cu aceeasi valoare maxima.

Numim cel mai bun caz, cazul in care se efectueaza cele mai putine operatii, cel mai rau caz este cazul in care se efectueaza cele mai multe operatii. Pentru algoritmul de mai sus prima clasa de valori de intrare constituie cazul cel mai bun, numarul de operatii in acest caz fiind: (N+1) atribuiri + (2N-1) comparatii, iar ultima clasa constituie cazul cel mai rau, numarul de operatii in acest caz fiind: (N+1+N–1=2N) atribuiri + (2N-1) comparatii.

Asa cum am discutat mai sus, orice algoritm prezinta un necesar de timp si de memorie ce formeaza asa numita complexitatea algoritmului. Procesul de masurare a complexitatii unui algoritm se numeste analiza algoritmului. In cele ce urmeaza ne vom concentra asupra complexitatii timpului de executie al unui algoritm. Sigur ca nu vom putea calcula chiar timpul de executie a algoritmului ci vom determina o functie care depinde de marimea problemei de rezolvat si care este direct proportionala cu timpul de executie. Aceasta functie examineaza modul de crestere al timpului de executie in cazul cresterii marimei problemei rezolvate de catre algoritm. Aceasta functie se numeste rata de crestere a algoritmului. Comparand functiile de crestere a doi algoritmi putem determina care dintre ei este mai rapid si deci eficient.

Pentru un algoritm dat, vom putea estima timpul minim, adica timpul de executie in cazul cel mai bun si timpul maxim, adica timpul de executie in cazul cel mai rau. Cele doua cazuri extreme apar foarte rar si o analiza completa presupune determinarea timpului mediu de executie definit ca media aritmetica a timpilor de executie corespunzatoare tuturor cazurilor posibile. Din pacate, acest lucru este mai greu si uneori ne vom limita la estimarea timpului maxim al unui algoritm. Pentru algoritmul de mai sus, timpii de executie ai celor N cazuri posibile, proportionali cu numarul operatiilor efectuate, sunt urmatorii: 3N, 3N +1, ..., 4N –1 si deci, timpul mediu este dat de

$$\tfrac{1}{N}\sum_{i=1}^{N-1}(3N+i) = \tfrac{1}{N}\big(3N(N-1) + N(N-1)/2\big) = \tfrac{7}{2}(N-1).$$

Exemplu: Sa consideram urmatoarea problema: Dat n > 0 intreg sa se calculeze suma 1 + 2 + ... + n. Prezentam trei algoritmi diferiti pentru rezolvarea acestei probleme. Ne propunem sa facem analiza fiecarui algoritm si sa il determinam pe cel mai eficient.

Algoritm 1	**Algoritm 2**	**Algoritm 3**

suma = 0
for i = 1, n
    suma = suma + i
endfor
	suma = 0
	for i = 1, n
	    for j = 1, i
	        suma = suma + 1
	    endfor
	endfor
		suma = n * (n - 1) / 2

Marimea problemei este data de n. Daca n creste suma are mai multi termeni si deci sunt mai multe operatii ce trebuie efectuate.Vrem sa determinam functiile de crestere ale celor trei algoritmi. Determinam intai numarul de operatii efectuate de fiecare algoritm. Observati ca pentru toti algoritmi, nu exista un "cel mai bun caz" sau "cel mai rau caz" pentru ca dat un n algoritmi efectueaza acelasi numar de operatii, sau mai exact exista o singura clasa de valori intrare.

Algoritm 1	Algoritm 2	Algoritm 3
Atribuiri: 2n+2 Comparatii: n+1 Adunari/scaderi: n Inmultiri/impartiri: 0 Total operatii: 4n+3	Atribuiri: n^2+2 Comparatii: (n^2+3n +2)/2 Adunari/scaderi: n(n-1)/2 Inmultiri/impartiri: 0 **Total operatii: $2n^2$ + n+3**	Atribuiri: 1 Comparatii: 0 Adunari/scaderi: 1 Inmultiri/impartiri: 2 **Total operatii: 4**

Daca presupunem ca pentru efectuarea tuturor acestor operatii este necesar acelasi numar de unitati de timp, putem spune ca:

- rata de crestere a algoritmului 1 este direct proportional cu 4n+3
- rata de crestere a algoritmului 2 este direct proportional cu $2n^2$+ n+3
- rata de crestere a algoritmului 3 este direct proportional cu 4.

La comportarea algoritmilor ne intereseaza comportarea lor pentru valori mari ale problemei, respectiv n in cazul nostru. Deoarece n^2 este mult mai mare decat n+3 (fapt pe care il scriem $n^2 >> n+3$), pentru valori mari ale lui n, putem spune ca $2n^2$+ n+3 se comporta ca si n^2. De asemenea, 4n+3 se comporta ca n, iar rata de crestere a algoritmului 3 este independenta de n.

Informaticienii folosesc o notatie pentru reprezentarea complexitatii unui algoritm si anume, faptul ca rata de crestere a algoritmului 2 este proportionala cu n^2 se spune ca algoritmul 2 este $O(n^2)$, sau ca algoritmul 2 este de ordin cel mult n^2. In exemplul de mai sus avem:

- Algoritmul 1 este O(n),
- Algoritmul 2 este $O(n^2)$,
- Algoritmul 3 este O(1) (adica independent de n).

Deoarece $n^2 >> n >> 1$ pentru n mare, putem spune ca cel mai bun algoritm este Algoritmul 3, urmat in ordine de Algoritmul 1 si Algoritmul 2.

De obicei, algoritmii intalniti pot avea rate de crestere de ordinuri: 1, n, n^2, n^3, log n, 2^n , n! si altele. Tabelul urmator ne arata cum se comporta cateva functii de crestere pentru valori crescatoare ale lui n.

n	**log n**	**n**	**n log n**	**n^2**	**n^3**	**2^n**	**n!**
10	3	10	33	**10^2**	**10^3**	**10^3**	**10^5**
10^2	7	**10^2**	664	**10^4**	**10^6**	**10^{30}**	**10^{94}**
10^3	10	**10^3**	9966	**10^6**	**10^9**	**10^{301}**	**10^{1435}**
10^4	13	**10^4**	132877	**10^8**	**10^{12}**	**10^{3010}**	**10^{19335}**
10^5	17	**10^5**	1660964	**10^{10}**	**10^{15}**	**10^{30103}**	**10^{243338}**
10^6	20	**10^6**	19931569	**10^{12}**	**10^{18}**	**10^{301030}**	**$10^{2933369}$**

Deci avem: $1 << \log n << n << n \log n << n^2 << n^3 << 2^n << n!$.

Definitie: Timpul de rulare al unui algoritm f(n) este de ordin cel putin g(n), i. e. f(n) = O(g(n)) daca $\exists$ c >0, c real $\exists$ N natural astfel incat $f(n) \leq c\, g(n)$, $\forall\, n \geq N$.

Exemplu: Se poate demonstra ca

1. f(n) = 4n+3 = O(n),
2. f(n) = $2n^2$+ n+3 = $O(n^2)$,
3. f(n) = 4 = O(1).

Observatie importanta: In general, pentru analiza unui algoritm nu vom calcula numarul tuturor operatiilor pe care un algoritm le efectueaza, ca de exemplu, nu vom lua in calcul modul de implementare al unui for loop si implicit numarul de atribuiri sau comparatii efectuate pentru stabilirea valorii indicelui, nici chiar fiecare atribuire sau operatie de adunare/scadere/inmultire/impartire ci lua in considerare numai acele operatii importante, de care depinde timpul de rulare al unui algoritm, deci vom defini una sau mai multe operatii de baza pe care le efectueaza un algoritm si vom calcula numarul de operatii de acel tip, in functie de marimea problemei. De exemplu, in cazul unui algoritm de inmultire a doua matrici, numarul de inmultiri este esential, operatia de baza fiind inmultirea, in cazul unui algoritm de cautare a unui element dat intr-o lista, numarul de comparatii intre elementul dat si elementele din lista este esential.

Lucru individual

Pentru fiecare dintre perechile de functii f(n) si g(n), avem fie f(n) = O(g(n)) or g(n) = O(f(n)), dar nu amandoua cazurile in acelasi timp. Determinati care din cele doua relatii este adevarata pentru urmatoarele functii:

a) $f(n) = (n^2 - n)/2$, $g(n) = 6n$
b) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$
c) $f(n) = n + n \log n$, $g(n) = n\sqrt{n}$
d) $f(n) = n^2 + 3n + 4$, $g(n) = n^3$
e) $f(n) = n \log n$ $g(n) = n\sqrt{n}/2$
f) $f(n) = n + \log n$, $g(n) = \sqrt{n}$

Probleme propuse

1. Care este complexitatea urmatorilor algoritmi in cel mai rau caz?
   - Afiseaza toti intregii unui sir de numere intregi
   - Afiseaza toti intregii unei liste inlantuita.
   - Afiseaza al n-lea intreg intr-un sirf de intregi
   - Calculeaza suma primilor n par intregi intr-un sir de intregi.
2. Scrieti doi algoritmi, unul quadratic si unul liniar pentru problema urmatoare:
   Dat un sir x de n elemente, x[1], x[2], ..., x[n], calculati un sir a[i], i = 1, 2, ..., n astfel incat a[i] = media aritmetica a elementelor x[1], ..., x[i]. Analizati cei doi algoritmi.
3. Descrieti o metoda simpla de calcul a valorii unui polinom intr-un punct.
   Pentru aceeasi problema aplicati schema lui Horner. Analizati ordinul de complexitate a algoritmului.
4. Pentru urmatoarele cicluri for, aflati ordinul de complexitate in functie de n:
   a.
   ```
   s = 0
   for i = 1, n
           s = s + i
   ```
   b.
   ```
   p = 1
   ```

```
    for i = 1, 2*n
        p = p* i
c.  p = 1
    for i = 1, n^2
        p = p* i
d.  s = 0
    for i = 1, 2*n
        for j = 1, i
            s = s + i
e.  s = 0
    for i = 1, n^2
        for j = 1, i
            s = s + i
```

UNITATEA DE ÎNVĂȚARE 4

Algoritmi de sortare

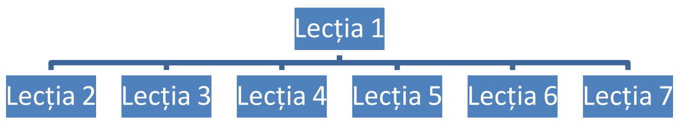
Obiective urmărite:

La sfârşitul parcurgerii acestei UI, studenţii

- vor cunoaste modul de funcționare a principalilor algoritmi de sortare
- vor putea să analizeze algoritmii de sortare studiați
- vor şti să implementeze intr-un limbaj de programare algoritmii de sortare studiați
- vor şti să coreleze cunoştinţele teoretice cu abilitatea de a le aplica în practică
- vor şti să elaboreze în cadrul unui proiect un studiu comparativ ai principalilor algoritmi de sortare care sa scoata in evidenta importanta algoritmilor specifici disciplinei precum si intelegerea modalitatii de alegere a algoritmului optim

Ghid de abordare a studiului:

Timpul mediu necesar pentru parcurgerea si asimilarea unitatii de invatare: 8h.
Lecțiile se vor parcurge în ordinea sugerată de diagramă.



Rezumat

În această UI sunt prezentati principalii algoritmi de sortare: sortare prin numarare, sortare prin inserare, sortarea prin metoda bulelor (Bubblesort), sortarea rapidă (Quicksort), sortarea prin selectie, sortarea prin interclasare (Mergesort). Fiecare algoritm este prezentat în limbaj pseudocod, este analizată complexitatea algoritmului şi se discută implementarea acestuia într-un limbaj de programare.

Cuvinte cheie

sortare, sortare prin numarare, sortare prin inserare, sortarea prin metoda bulelor (Bubblesort), sortarea rapidă (Quicksort), sortarea prin selectie, sortarea prin interclasare (Mergesort), algoritm de interclasare, recursivitate

Lecția 1. Sortare

Orice colectie de obiecte intre care exista o relatie de ordine poate fi sortata adica aranjata in ordine crescatoare sau descrescatoare.

De exemplu, putem aranja cartile dintr-o biblioteca in ordine alfabetica dupa autor sau dupa titlu, sau chiar dupa culoare, daca stabilim in prealabil o ierarhie a culorilor, putem aranja studentii unei grupe in ordine alfabetica dupa numele de familie sau in ordine descrescatoare a mediei fiecarui student.

In cele ce urmeaza presupunem ca avem o colectie de obiecte $R_1, R_2, ..., R_N$ ce trebuie sortata intr-o anumita ordine, sa zicem crescator dupa una din caracteristicile comune ale obiectelor din colectie pe care o vom numi cheie si o vom nota K. Deci vrem sa gasim o permutare p a numerelor 1, 2, ..., N, astfel incat $K_{p(1)} \leq K_{p(2)} \leq ... \leq K_{p(N)}$ sau o modalitate de rearanjare a obiectelor astfel incat $K_1 \leq K_2 \leq ... \leq K_N$.

Lecția 2. Sortare prin numarare

Aceasta tehnica de sortare se bazeaza pe faptul dupa sortare elementul aflat pozitia i are cheia mai mare decat cheile a i -1 elemente si deci numarand pentru fiecare element din lista, cate elemente au cheile mai mici decat cheia lui gasim pozitia elementului in sirul ordonat. Pentru memorarea pozitiei se foloseste un sir auxiliar.

<u>Algoritm de sortare prin numarare:</u> Se dau $K_1, K_2, ..., K_N$. Se determina pozitia fiecarui element in sirul ordonat. Se foloseste sirul auxiliar poz cu N elemente intregi. Dupa terminarea algoritmului, poz[i] va reprezenta pozitia lui K_i in sirul ordonat.

```
for i = 1, N
        poz[i] = 1
endfor
for i = 1, N-1
        for j = i+1, N
                if Kj < Ki then poz[i] = poz[i] + 1
                        else poz[j] = poz[j] + 1
                endif
        endfor
endfor
```

<u>Observatie:</u> ca algoritmul functioneaza chiar si daca elementele au chei egale.

<u>Analiza algoritmului:</u> Observati ca exista o singura clasa de valori posibile de intrare. Asa cum mentionam in finalul capitolului 9 ***Analiza eficientei algoritmilor,*** vom determina numai numarul de operatii de baza efectuate, in cazul nostru de comparatii. Numar total comparatii = $\sum_{i=1}^{N-1}(N-i) = \frac{1}{2}N(N-1)$. Deci algoritmul este $O(N^2)$ (deci nu este un algoritm rapid) si, in plus, necesita si un spatiu de memorie pentru cele N elemente ale sirului auxiliar. De fapt, nu am prezentat acest algoritm pentru eficienta sa ci pentru simplitatea sa..

Lecția 3. Bubblesort (Sortarea prin metoda bulelor)

Idea principala este de a permite elementelor mici sa se mute la inceputul listei, iar cele mari spre sfarsitul listei. Lista se parcurge de mai multe ori. La fiecare trecere oricare doua elemente vecine se compara si daca nu sunt in ordine crescatoare se interschimba. Incepem comparand K_1 si K_2, si daca nu sunt in ordine atunci cele doua elemente se interschimba, comparam apoi K_2 si K_3, apoi K_3 si K_4 s.a.m.d., iar ultima data K_{N-1} si K_N. Observati ca deja dupa prima trecere elementul cel mai mare al listei se va afla in ultima pozitie in lista si deci a doua trecere va trebui sa parcurga lista numai de la K_1 la K_{N-1}. Algoritmul se opreste atunci cand este efectuata o trecere prin lista si nici o interschimbare nu a fost efectuata.

Algoritm Bublesort: Se dau K_1, K_2, ..., K_N.Se foloseste o variabila *schimba* care este true atunci cand se cere o interschimbare si false cand nu mai este nevoie de nici o interschimbare.

```
ultim = N
schimba = true
while schimba = true
          schimba = false
           for i = 1, ultim -1
                     if Ki > Ki+1 then Ki ↔ Ki+1
                                          schimba = true
                     endif
           endfor
           ultim = ultim –1
endwhile
```

Analiza algoritmului: Intai sa observam ca operatia principala care se efectueaza este comparatia dintre elementele listei. Sa estimam intai timpul minim de executie. In cel mai bun caz, lista este ordonata crescator, se efectueaza o singura trecere printre elementele listei, nici o interschimbare nu este efectuata si algoritmul se opreste. Numarul de comparatii= $\sum_{i=1}^{N-1} 1 = N-1$ si deci timpul minim ~ N. In cel mai rau caz, va fi necesara parcurgerea listei de N-1 ori, caz ce corespunde unei liste de intrare ordonata strict descrescator. Deci numarul de comparatii= $\sum_{i=1}^{N-1} i = \frac{1}{2}N(N-1)$ si deci timpul maxim ~ N^2. Se poate demonstra ca timpul mediu ~ N^2 si deci algoritmul este $O(N^2)$. Deci nici acesta nu este un algoritm rapid dar nu necesita deplasari de blocuri mari de elemente.

Lucru individual

Scrieti un program care implementeaza algoritmul de mai sus.

Lecția 4. Quicksort (Sortarea rapidă)

Quicksort alege un element din lista, numit pivot si rearanjeaza lista, prin interschimbari, astfel incat toate elementele mai mici decat pivotul sunt mutate inaintea lui, iar toate elementele mai mari sunt mutate dupa pivot. Elementele din cele doua parti (inainte si dupa element) nu sunt neaparat in ordine. Tot ceea ce stim este ca, daca pivotul este mutat in pozitia i, atunci toate elementele din pozitiile 1, 2, ..., i-1 sunt mai mici decat pivotul, iar toate elementele din pozitiile i+1, ..., N sunt mai mari.

Apoi Quicksort este apelat recursiv pentru cele doua parti ce trebuie sortate. Exista cel putin doua metode de alegere a pivotului. Noi o vom prezenta pe cea mai simpla si anume se alege ca pivot elementul de inceput al listei. Deci se parcurge lista comparand acest pivot cu toate elementele listei. Cand gaseste un element mai mic decat pivotul, pozitia pivotului este incrementata si elementul mai mic este intershimbat cu elementul aflat in noua pozitie a pivotului, deci elementul mai mic se schimba cu un altul mai mare, cel mic mutandu-se mai aproape de inceput, inaintea pivotului. In final, dupa ca a fost parcursa toata lista, se pozitioneaza pivotul in pozitia nou gasita, prin interschimbarea primului element cu elementul aflat in pozitia pivotului.

Algoritm Quicksort: Se dau $K_1, K_2, ..., K_N$.

```
Quicksort (K, prim, ultim) /* Sorteaza elementele Kprim, ..., Kultim. Apelarea
                              initiala este pentru prim = 1 si ultim =N*/
if prim < ultim then  valoarepivot = Kprim
                      pozpivot = prim
                      for i = prim + 1, ultim
                          if Ki < valoarepivot then pozpivot = pozpivot +1
                                                    Ki ↔ Kpozpivot
                          endif
                      endfor
                      Kprim ↔ Kpozpivot
                      call Quicksort (K, prim, pozpivot –1)
                      call Quicksort (K, pozpivot +1, ultim)
endif
```

Analiza algoritmului: Operatia principala care se efectueaza este comparatia dintre elementele listei. Prima parte a algoritmului o constituie partitionarea listei prin gasirea pozitiei pivotului. Aceasta partitionare necesita un numar de comparatii = numarul de elemente din secventa analizata –1 (sau, daca vreti = ultim – prim), pentru acesta neexistand decat un singur caz. Desigur aceasta nu se intampla si pentru intregul algoritm Quicksort, pentru care se distinge timpul minim, mediu si maxim. Surprinzator, cel mai rau caz pentru acest algoritm este cazul in care lista este deja ordonata crescator. Motivatia este urmatoarea: Lista fiind ordonata crescator, pozitia pivotului ramane neschimbata adica de fiecare data pozpivot = prim si deci se imparte lista originala in doua liste, una vida iar cealalta avand cu 1 mai putine elemente decat lista originala. Deci numarul de operatii necesare sortarii unei liste ordonate crescator cu N elemente $K_1, K_2, ..., K_N$ va fi = numarul de operatii necesarii partitionarii (asa cum am vazut, acest numar este N-1) + numarul de operatii necesare sortarii listei $K_2, ..., K_N$ (tot o lista ordonate crescator dar cu N-1 elemente). Obtinem deci urmatoarea relatie de recurenta: $C_N = N-1 + C_{N-1}$, daca $N > 1$ si $C_1 = 0$ deci $C_N = \sum_{i=1}^{N-1} i = \frac{1}{2}N(N-1)$ deci timpul maxim ~ N^2. Se poate demonstra ca acest algoritm este O(N log N).

Lecția 5. Sortare prin selectie

Ideea algoritmului este: gaseste cel mai mic element si muta-l in prima pozitie prin interschimbarea cu elementul de pe prima pozitie si repeta procedeul pentru elementele listei de indici 2, 3, ..., N.

<u>Algoritm de sortare prin selectie (varianta iterativa):</u> Se dau $K_1, K_2, ..., K_N$.

for i = 1, N-1
  Determina indexul celui mai mic element dintre $K_i, K_{i+1}, ..., K_N$.
  Fie index_mic acesta.
  if i $\neq$ index_min then $K_i \leftrightarrow K_{index\_mic}$
  endif
endfor

<u>Algoritm de sortare prin selectie (varianta recursiva):</u> Se dau $K_1, K_2, ..., K_N$.

Selectie (K, prim, ultim) /* Sorteaza elementele $K_{prim}, ..., K_{ultim}$. Apelarea initiala este pentru prim = 1 si ultim =N*/
if (prim < ultim) then
  Determina indexul celui mai mic element dintre $K_{prim}, K_{prim+1}, ..., K_{ultim}$.
  Fie index_mic acesta.
  if prim $\neq$ index_min then $K_{prim} \leftrightarrow K_{index\_mic}$
  endif
  call Selectie (K, prim +1, ultim)
endif

<u>Analiza algoritmului:</u> Operatia principala care se efectueaza este comparatia dintre elementele listei. Acest algoritm prezinta o singura clasa de valori de intrare posibile, numarul de comparatii efectuate fiind acelasi indiferent de lista de intrare. Observati ce numarul de comparatii necesar determinarii celui mai mic element intr-un sir cu n elemente este n-1 si deci relatia de recurenta care ne da acest numar de comparatii efectuate de algoritmul descris mai sus este $C_N = N-1 + C_{N-1}$, daca $N > 1$ si $C_1 = 0$ deci $C_N = \sum_{i=1}^{N-1} i = \frac{1}{2}N(N-1)$ deci complexitatea algoritmul este $O(N^2)$.

Lecția 6. Sortarea prin inserare

Una din cele mai importante metode de sortare este sortarea prin inserare. Aceasta nu necesita memorie auxiliara si este de asemenea o metoda simpla. Ideea principala este urmatoarea: se considera pe rand fiecare element al listei ce trebuie sortata si se introduce intr-o anumita pozitie astfel incat atunci cand elementul i este examinat, toate elementele de pe pozitiile 1, 2, ..., i-1 sunt deja sortate. Deci, daca se presupune ca $K_1 \leq K_2 \leq ... \leq K_{i-1}$ si vrem sa introducem O_i astfel incat secventa primelor i elemente sa fie ordonate crescator, atunci cautam pozitia j, $j \geq 1$ astfel incat $K_j \leq K_i < K_{j+1,}$ incepand de la sfarsitul secventei. Cand pozitia j a fost gasita, intreaga secventa $O_{j+1}, ..., O_{i-1}$ se deplaseaza o pozitie spre dreapta, iar elementul O_i se insereaza in pozitia j+1. Daca nu exista un astfel de j inseamna ca $K_i < K_1 \leq K_2 \leq ... \leq K_{i-1}$ si deci elementul O_i trebuie introdus in prima pozitie deci, intreaga secventa $K_1, ..., K_{i-1}$ se deplaseaza o pozitie spre dreapta, iar elementul O_i se insereaza in pozitia 1. Cazul acesta din urma se reduce la primul daca se considera j=0.

<u>Algoritm de sortare prin inserare:</u> Se dau $O_1, O_2, ..., O_n$ obiecte ce vor rearanjate astfel incat cheile lor $K_1, K_2, ..., K_n$ sa fie ordonate crescator.

```
for i = 2, n
        elem = K_i
        j = i -1
        while j >= 1 and  K_j > K_i
                j = j –1
        endwhile
        pozitie = j +1
        for j= i, pozitie+1, -1
                O_j = O_j-1
        endfor
        K_pozitie = elem
endfor
```

Analiza algoritmului: Intai sa observam ca operatia principala care se efectueaza este comparatia dintre elementele listei, aceasta aparand numai in ciclul while. Sa estimam intai timpul minim de executie. In cel mai bun caz, lista este ordonata crescator, si deci pentru fiecare i=2, ..., n nu se face decat o comparatie $K_{i-1} > K_i$. Deci numarul de comparatii= $\sum_{i=2}^{n} 1 = n-1$ si deci timpul minim este de ordin n. In cel mai rau caz, ciclul while va repeta numarul maxim de comparatii (i-1) pentru fiecare i=2, ..., n, caz ce corespunde unei liste de intrare ordonata strict descrescator. Deci numarul de comparatii= $\sum_{i=2}^{N} i-1 = \frac{1}{2} n(n-1)$ si deci timpul maxim este de ordin n^2. Se poate demonstra ca timpul mediu este de ordin n^2 si deci algoritmul de sortare prin inserare este $O(n^2)$. Deci nici acesta nu este un algoritm rapid. Algoritmul nu necesita memorie auxiliara dar necesita deplasari de blocuri de elemente.

Lecția 7. Mergesort (Sortarea prin interclasare)

"Merge" inseamna a combina una sau mai multe liste ordonate (crescator, sa presupunem) intr-o singura lista ordonata (tot crescator, desigur). Idea algoritmul Mergesort este urmatoarea: Data o lista neordonata, o impartim in doua liste, de dimensiuni egale sau foarte apropriate, ordonam cele doua liste (folosind acelasi algoritm) si apoi efectuam operatia "merge", adica vom combina cele doua liste ordonate intr-una singura, obtinand astfel lista originala ordonata. Este usor de vazut ca algoritmul Mergesort este recursiv. Sa vedem intai algoritmul de efectuare a operatiei de combinare a celor doua liste. Deci presupunem ca avem doua liste A: $A_1 \le A_2 \le ... \le A_N$ si B: $B_1 \le B_2 \le ... \le B_M$ si vrem sa obtinem o lista C cu N+M elemente (toate elementele celor doua liste), astfel incat: $C_1 \le C_2 \le ... \le C_{N+M}$. Deoarece cel mai mic element din lista A este A_1, iar cel mai mic element din lista B este B_1, stim ca cel mai mic element din lista C, C_1, va fi cel mai mic dintre A_1 si B_1. Sa presupunem ca cel mai mic este A_1. Atunci C_2 va fi fie B_1 sau A_2 daca lista A inca mai contine elemente mai mici decat B_1. Pentru a putea hotari ce element urmeaza in lista C, vom folosi doi indici, unul pentru lista A si unul pentru lista B. Elementul cel mai mic este introdus in lista C, dupa care indicele listei din care facea parte acest element minim este incrementat cu 1. Se continua procedeul pana cand una din liste se termina (chiar daca listele au acelasi numar de elemente, una din liste se va termina naintea celeilalte), caz in care elementele ramase in lista cealalta se scriu in ordine in lista C.

Algoritm Merge: Se dau $A_1 \le A_2 \le ... \le A_N$ si B: $B_1 \le B_2 \le ... \le B_M$. Lista C va fi lista nou creata

```
indexA = 1
```

```
indexB = 1
indexC = 1
while indexA ≤ N and indexB ≤ M
        if A_indexA < B_indexB then C_indexC = A_indexA
                                    indexA = indexA +1
                              else  C_indexC = B_indexB
                                    indexB = indexB +1
        indexC = indexC +1
endwhile
while indexA ≤ N
        C_indexC = A_indexA
        indexA = indexA +1
        indexC = indexC +1
endwhile
while indexB ≤ M
        C_indexC = B_indexB
        indexB = indexB +1
        indexC = indexC +1
endwhile
```

Algoritm Mergesort: Se dau $K_1, K_2, ..., K_N$.

```
Mergesort (K, prim, ultim) /* Sorteaza elementele K_prim, ..., K_ultim. Apelarea
                              initiala este pentru prim = 1 si ultim =N*/
if prim < ultim then
            mijloc = ⌊(prim + ultim)/2⌋
            call Mergesort(K, prim, mijloc)
            call Mergesort(K, mijloc +1, ultim)
            call Merge(K_prim ≤ ... ≤ K_mijloc si K_mijloc+1 ≤ ... ≤ K_ultim)⇒ C_prim, ..., C_ultim
            for i = prim, ultim
                    K_i = C_i
            endfor
endif
```

Analiza algoritmului: Operatia principala care se efectueaza este, din nou, comparatia dintre elementele listei. Intai observati ca algoritmul Merge va compara un element al listei A cu un element al listei B pana cand una din liste se termina. Ce se intampla daca toate elementele listei A sunt mai mici decat cele din lista B? Aceasta ar insemna ca fiecare element al lui A va fi comparat cu B (deoarece toate sunt mai mici decat acesta ele vor fi copiate in C iar elementele lui B vor fi apoi copiate), deci numarul de comparatii efectuate va fi N. Similar, daca toate elementele listei B sunt mai mici decat cele din lista A atunci numarul de comparatii efectuate va fi M. Se poate arata ca, in cel mai bun caz, numarul de comparatii va fi min(N, M). Sa consideram acum cazul in care elementele listei A sunt printre elementele listei B, cu alte cuvinte $B_1 \le A_1 \le B_2$, $B_2 \le A_2 \le B_3$, etc. In acest caz numarul comparatiilor este N + M – 1, caz ce corespunde celui mai rau caz. Deci numarul maxim de comparatii este N – M – 1. Acum, cunoscand ordinul de complexitate al algoritmului Merge, putem considera algoritmul Mergesort. Notam cu C^{min} (N) = numarul de comparatii efectuate in cel mai bun caz si cu C^{max} (N)= numarul de comparatii efectuate in cel mai rau caz. Conform analizei anterioare, algoritmul Merge asa cum este aplicat in algoritmul de sortare Mergesort, va efectua un numar minim de comparatii = N/2 si un numar maxim de comparatii = N/2 + N/2 –1 = N –1. Obtinem urmatoarele relatii de recurenta:

$C^{max}(N) = 2\, C^{max}(N/2) + N-1,\ C^{max}(1) = C^{max}(0) = 0$

$C^{min}(N) = 2\, C^{min}(N/2) + N/2,\ C^{min}(1) = C^{min}(0) = 0$

Se poate arata ca $C^{max}(N) = C^{min}(N) = O(N \log N)$.

Deci ordinul de complexitate al algoritmului Mergesort este N log N, si deci mergesort este un algoritm eficient. Dezavantajul acestei metode este faptul ca algoritmul de combinare al celor doua liste ordonate necesita un spatiu de memorie destul de mare.

Probleme propuse

1. O versiune diferita a algoritmului Bubblesort memoreaza locul in care a fost facuta ultima modificare si la pasul urmator nu va analiza elementele sirului care dincolo de acel loc. Scrieti algoritmul acestei noi versiuni si implementati-l. Aceasta versiune noua modifica analiza algoritmului in cel mai rau caz?
2. Implementati algoritmul Shellsort.
3. Implementati algoritmul Heapsort.
4. Implementati algoritmul bucket sort.
5. Implementati oricare trei algoritmi de sortare prezentati in materialul teoretic. Folosind cele trei programe, comparati timpurile de rulare ale lor pentru siruri de elemente generate aleator.

UNITATEA DE ÎNVĂȚARE 5

Algoritmi de căutare

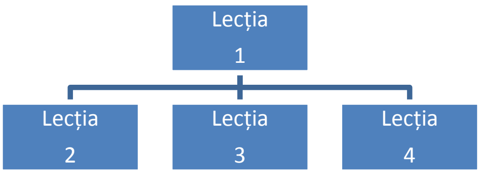
Obiective urmărite:

La sfârşitul parcurgerii acestei UI, studenţii

- vor cunoaste modul de funcționare a principalilor algoritmi de căutare
- vor putea să analizeze algoritmii de căutare studiați
- vor şti să implementeze intr-un limbaj de programare algoritmii de căutare studiați
- vor şti să coreleze cunoştinţele teoretice cu abilitatea de a le aplica în practică
- vor cunoaşte noțiunile de arbori de căutare
- vor şti cum să folosească arborii de căutare pentru implementarea algoritmilor de căutare

Ghid de abordare a studiului:

Timpul mediu necesar pentru parcurgerea si asimilarea unitatii de invatare: 6h.
Lecțiile se vor parcurge în ordinea sugerată de diagramă.



Rezumat

În această UI sunt prezentati principalii algoritmi de căutare: secventiala şi binara. Sunt prezentați arborii binari de căutare ca modalități de structurare a datelor în care se face căutarea.Se analizează cautarea si inserarea in arbori binari de cautare; Fiecare algoritm este prezentat în limbaj pseudocod, este analizată complexitatea algoritmului şi se discută implementarea acestuia într-un limbaj de programare.

Cuvinte cheie

Căutare, căutare secventiala, căutare binara, arbori binari de căutare, cautarea si inserarea in arbori binari de cautare; recursivitate

Lecția 1. Căutare

Ca si in cazul sortarii, presupunem ca avem o colectie de obiecte R_1, R_2, ..., R_N si ca fiecare obiect prezinta o caracteristica pe care o vom numi cheie. Data o valoare K problema este de gasi obiectul din colectie care are cheia K. In general, presupunem ca cele N chei sunt distincte. Cautarea poate fi cu succes, daca a fost localizat obiectul cu cheia K sau fara succes daca nu a fost gasit nici un obiect cu cheia K. In continuare, pentru o intelegere mai buna a algoritmilor de cautare, vom ignora orice alta caracteristica pe care ar putea avea-o obiectele din colectie si vom lua in considerare numai caracteristica cheie.

Lecția 2. Cautare secventiala

Cautarea secventiala presupune cautarea in colectie a elementului dat, element cu element, incepand cu primul si oprindu-ne fie cand elementul a fost gasit, fie cand a fost parcursa intreaga colectie si elementul nu a fost gasit.

Algoritm de cautare secventiala: Se dau K_1, K_2, ..., K_N si K. Se cauta K in lista. Folosim variabila de tip boolean *gasit* care este initial falsa si devine adevarata atunci cand elementul a fost gasit.

```
gasit = false
i = 1
while ( i ≤ N and not gasit)
        if K = Ki then gasit = true
        else i = i+1
        endif
endwhile
if gasit then tipareste "Cautare cu succes"
            K a fost gasit in pozitia i
else tipareste "Cautare fara succes"
endif
```

Analiza algoritmului: Operatia principala care se efectueaza este comparatia dintre elementul cautat si elementele listei. In cazul unei cautari fara succes numarul de comparatii este N. In cazul unei cautari cu succes putem avea urmatoarele clase de valori de intrare:

$K = K_1$ caz in care se efectueaza 1 comparatie

$K = K_2$ caz in care se efectueaza 2 comparatii

......

$K = K_N$ caz in care se efectueaza N comparatii

Deci in medie numarul de comparatii va fi:

$$\frac{1}{N+1}\left(1+2+...+N+N\right)=\frac{N}{2}+1-\frac{1}{N+1}=O(N)$$

Cel mai bun caz: elementul cautat este in prima pozitie si deci numarul minim de comparatii este 1.

Cel mai rau caz daca avem o cautare cu succes: elementul cautat este in ultima pozitie si deci sunt efectuate N comparatii.

Cel mai rau caz daca elementul avem o cautare fara succes: N comparatii.

Algoritm de cautare secventiala intr-un sir ordonat crescator:

Se dau K1 < K2 < ... < KN si K. Se cauta K in lista. Folosim variabila de tip boolean gasit care este initial falsa si devine adevarata atunci cand elementul a fost gasit.

```
gasit = false
i = 1
while ( i ≤ N and not gasit)
        if K = Ki then gasit = true
        else if K < Ki then gasit = false
                    else i = i+1
             endif
        endif
endwhile
if gasit then tipareste "Cautare cu succes"
          K a fost gasit in pozitia i
else tipareste "Cautare fara succes"
endif
```

Analiza algoritmului: Operatia principala care se efectueaza este comparatia dintre elementul cautat si elementele listei.

Cel mai rau caz daca avem o cautare cu succes: elementul cautat este in ultima pozitie si deci sunt efectuate N comparatii.

Cel mai rau caz daca avem o cautare fara succes: $K > K_N$: se efectueaza N comparatii.

Cel mai bun caz daca avem o cautare cu succes: elementul cautat este in prima pozitie si deci numarul minim de comparatii este 1.

Cel mai bun caz daca avem o cautare fara succes: $K < K_1$ si deci numarul minim de comparatii este 1.

In cazul unei cautari cu succes putem avea urmatoarele clase de valori de intrare:

$K = K_1$ caz in care se efectueaza 1 comparatie

$K = K_2$ caz in care se efectueaza 2 comparatii

......

$K = K_N$ caz in care se efectueaza N comparatii.

In cazul unei cautari fara succes putem avea urmatoarele clase de valori de intrare:

$K < K_1$ caz in care se efectueaza 1 comparatie

$K_1 < K < K_2$ caz in care se efectueaza 2 comparatii

$K_2 < K < K_3$ caz in care se efectueaza 3 comparatii

......

$K_{N-1} < K < K_N$ caz in care se efectueaza N comparatii

$K > K_N$ caz in care se efectueaza N comparatii

Deci in medie numarul de comparatii va fi:

$$\frac{1}{2N+1}\left(1+2+...+N+1+2+...+N\right)=\frac{1}{2N+1}N(N+1)=\frac{N}{2}+\frac{1}{4}-\frac{1}{4(2N+1)}=O(N)$$

Lecția 3. Cautare binara

Presupunem ca lista de elemente este ordonata crescator, $K_1 < K_2 < ... < K_N$ si se cauta K in lista. Idea algoritmului de cautare binara este de a testa intai daca elementul K coincide cu elementul din mijloc al listei. Daca da, se obtine o cautare cu succes si algoritmul se termina. In caz contrar, putem aveam unul din urmatoarele cazuri:

i. $K < K_{mijloc}$, caz in care K trebuie cautat printre elementele $K_1, ..., K_{mijloc-1}$,

ii. $K > K_{mijloc}$, caz in care K trebuie cautat printre elementele $K_{mijloc+1}$, ..., K_N.

Procedeul continua pana cand K este gasit sau pana cand K se cauta intr-o lista vida, deci K nu este gasit.

De exemplu, sa presupunem ca avem o lista de numere intregi ordonate crescator:

1	4	9	10	15	21	23	38

si cautam

a) 7 in lista:

Elementul din mijloc este 10 (deoarece numarul de elemente din lista este par, exista de fapt doua elemente de mijloc; in acest caz vom lua drept element de mijloc pe cel cu indicele mai mic) (mijloc = 4). Deoarece 7 < K_4 =10, cautam 7 in prima jumatate a listei adica printre elementele K_1, K_2, K_3.

1	4	9	10	15	21	23	38

Elementul din mijloc al noii secvente din lista este 4 (mijloc = 2). Deoarece 7 > K_2 =3, cautam 7 in secventa din lista formata numai din elementul K_3.

1	4	9	10	15	21	23	38

Elementul din mijloc al noii secvente din lista este evident 9 (mijloc = 3). Deoarece 7 < K_3 = 9, secventa in care 7 trebuie cautat este vida, deci 7 nu a fost gasit si algoritmul se termina.

b) 21 in lista:

Elementul din mijloc este 10 (mijloc = 4). Deoarece 21 > K_4 = 10, cautam 21 in a doua jumatate a listei adica printre elementele K_5, ..., K_8.

1	4	9	10	15	21	23	38

Elementul din mijloc al noii secvente din lista este 21 (mijloc = 6). Deoarece 21 = K_6, cheia data a fost gasita si algoritmul se termina.

Algoritm de cautare binara (varianta iterativa):

Presupunem $K_1 < K_2 < ... < K_N$ si se cauta K in lista. Folosim doi indici *l*, *u* pentru a preciza limitele intre care se cauta cheia K. Initial *l = 1* si *u = N*. Folosim si variabila de tip boolean *gasit* care este initial falsa si devine adevarata atunci cand elementul a fost gasit.

```
l = 1, u =N, gasit = false
while (l ≤ u and not gasit)
        m = ⌊( u + l) / 2⌋
        if K = Km then gasit = true
        else if K < Km then u = m-1
             else l = m+1
             endif
        endif
endwhile
        if gasit then tipareste "Cautare cu succes"
                     K a fost gasit in pozitia m
        else tipareste "Cautare fara succes"
        endif
```

Algoritm de cautare binara (varianta recursiva):

Presupunem K1 < K2 < ... < KN si se cauta K in lista. Folosim doi indici l, u pentru a preciza limitele intre care se cauta cheia K. Initial l = 1 si u = N. Folosim si variabila de tip boolean gasit care este initial falsa si devine adevarata atunci cand elementul a fost gasit.

```
Cautarebinara(K, l, u)
if l ≤ u then
        m = ⌊( u + l) / 2⌋
        if K = Km then gasit = true
                       stop
                 else if K < Km then call Cautarebinara(K, l, m –1)
                          else call Cautarebinara(K, m+1, u)
                      endif
       endif
        else gasit = false
endif
```

Analiza algoritmului: Operatia principala care se efectueaza este comparatia dintre elementul cautat si elementele listei. Se observa ca de fiecare data cand se incearca gasirea elementului se face comparatia cu elementul din mijlocul listei, dupa care, daca nu este gasit se face cautarea intr-o lista de doua ori mai mica. Sa presupunem ca $N = 2^k - 1$, atunci la prima trecere prin lista se compara K cu elementul din mijloc al unei liste cu $2^k - 1$ elemente, la a doua trecere se compara K cu elementul din mijloc al unei liste cu $2^{k-1} - 1$ elemente, s.a.m.d. pana cand, in cel mai rau caz, la a k-a trecere se cauta K intr-o lista cu $2^1-1 =1$ elemente. Deci in cel mai rau caz se efectueaza $k = \log_2(N+1)$ comparatii. Daca N este oarecare, atunci numarul maxim de comparatii este $\lfloor \log_2(N+1) \rfloor$. Se poate arata ca algoritmul de cautare binara este $O(\lfloor \log_2(N+1) \rfloor)$.

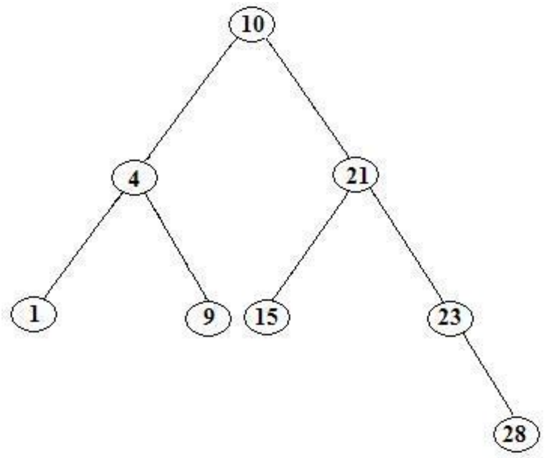
Lecția 4. Arbori binari de cautare

In capitolul precedent, am folosit algoritmul de cautare binara pentru a sorta elementele unei liste. Asa cum am vazut, dupa fiecare iteratie algoritmul reduce numarul de elemente in care se face cautarea la jumatate. Algoritmul este eficient dar structura de date folosita este o lista liniara, in care inserarile si stergerile nu sunt eficiente (mai exact sunt de ordin n). In continuare prezentam arborii binari de cautare in care operatiile de baza sunt proportionale cu inaltimea arborelui, care pt un arbore binar complet cu n noduri este log n.

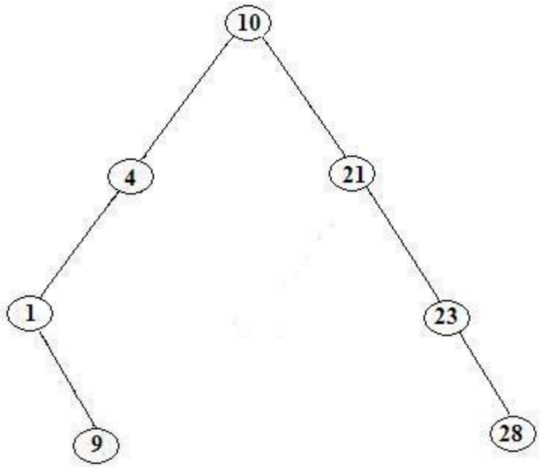
Definitie: Un arbore binar de cautare este un arbore binar in care, daca se presupune ca fiecarui nod ii asociem un element al colectiei de obiecte ce dorim sa o sortam, toate nodurile ce se afla in subarborele stang al oricarui nod dat au o valoare mai mica decat valoarea nodului dat, iar toate nodurile ce se afla in subarborele drept au o valoare mai mare decat valoarea nodului.

Exemple:

1. Arbore care este arbore binar de cautare

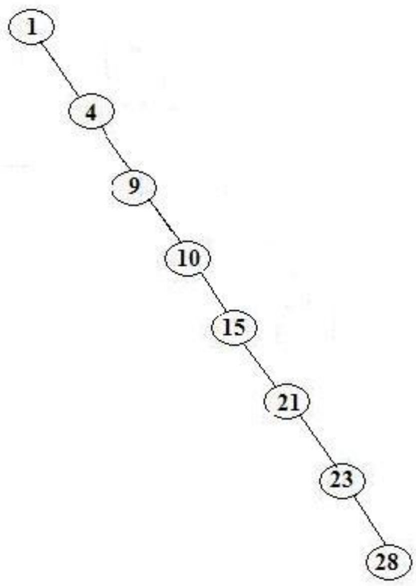


2. Arbore care nu este arbore binar de cautare



4.1 Cautare in arbori binari de cautare

Pentru a cauta o valoare data intr-un arbore binar de cautare incepem comparand valoarea data cu radacina arborelui si mergem apoi in jos la stanga sau la dreapta, depinde cum este valoarea cautata fata de valoarea radacinii. De exemplu, in exemplul 1 de mai sus, pentru a cauta valoarea 9, comparam 9 cu 10 valoarea radacinii, si cum 9 < 10 comparam valoarea cautata cu radacina subarborelui stang adica vom compara 9 si 4 si cum 9>4 se merge si mai jos un nivel si se compara valoarea cautata cu valoarea radacinii subarborelui drept, si observam ca am gasit valoarea cautata. Fiecare comparatie reduce numarul de elemente comparate la jumatate si deci algoritmul este similar cautarii binare intr-o lista liniara. Dar acest lucru se intampla numai cand arborele este <u>echilibrat.</u> De exemplu, pentru un arbore ca cel din figura de mai jos timpul de cautare este proportional cu numarul de elemente ale intregii colectii.



4.2 Inserare si stergere arbori binari de cautare

Operatiile de inserare si stergere trebuie efectuate astfel incat proprietatea de arbore binar de cautare sa se mentina.

Algoritm de inserare (varianta recursiva): Se dau arbore binar de cautare T = pointer la radacina arborelui si v noua valoare ce trebuie inserata in arbore.

```
Insert (T, v)
if  T = NULL then Aloca memorie pentru un nod nou. Returneaza p, un
                  pointer la noul nod.
                  p -> info = v
                  p ->stang = NULL
                  p -> drept = NULL
                  T = p
            else if  v > T -> info then call Insert (T->drept, v)
                else if  v < T -> info then  call Insert (T-> stang, v)
                              else write "valoare deja in arbore"
                                   stop
                     endif
                 endif
endif
```

Algoritm de stergere: Se dau arbore binar de cautare T = pointer la radacina arborelui si nodul N ce trebuie eliminat din arbore. In acest caz exista trei cazuri:

1. N este nod terminal atunci daca se noteaza cu P un pointer la tatal nodului N atunci nodul N este inlocuit cu NULL, adica P->stang = NULL daca N este fiu stang or P->drept = NULL daca N este fiu drept.
2. N are un singur fiu si fie X un pointer la fiul lui N atunci fiul lui N devine fiul tatalui lui N, adica daca se noteaza cu P un pointer la tatal nodului N atunci P->stang = X daca N este fiu stang or P->drept = X daca N este fiu drept.
3. N are 2 fii: algoritmul este:
   Gaseste R cel mai din dreapta nod al subarborelui stang al lui N
   Inlocuieste informatia nodului N cu informatia din nodul R

Sterge nodul R.

Lucru individual

Scrieti un program care

- cauta un element dat intr-un arbore binar de cautare,
- insereaza un element dat intr-un arbore binar de cautare,
- sterge un element dat intr-un arbore binar de cautare.

Probleme propuse

1. Scrieti un program care verifica daca un arbore binar este arbore binar de cautare.
2. Scrieti un program care verifica daca un arbore binar este arbore binar echilibrat.
3. Scrieti un program care calculeaza reuniunea, intersectia si cele doua diferente ale doua multimi date.
4. Implementati algoritmul de cautare prin interpolare.
5. Scrieti un algoritm recursiv de cautare a unui element intr-un sir, examinand ultimul element al sirului.

FORMULAR DE FEEDBACK

În dorinţa de ridicare continuă a standardelor desfăşurării activitatilor dumneavoastra, va rugăm să completaţi acest chestionar şi să-l transmiteţi indrumatorului de an.

Disciplina: Algoritmi si structuri de date
Unitatea de invatare/modulul:___________________
Anul/grupa:___________________________
Tutore:__________________________

Partea I

1. Care dintre subiectele tratate in aceasta unitate/modul consideraţi că este cel mai util şi eficient? Argumentati raspunsul.

2. Ce aplicatii/proiecte din activitatea dumneavoastra doriţi să imbunatatiti/modificaţi/implementaţi în viitor în urma cunoştinţelor acumulate în cadrul acestei unitati de invatare/modul?

3. Ce subiecte consideraţi că au lipsit din acesta unitate de invatare/modul?

4. La care aplicatii practice ati intampinat dificultati in realizare? Care credeti ca este motivul dificultatilor intalnite?

6. Timpul alocat acestui modul a fost suficient?

7. Daca ar fi sa va evaluati, care este nota pe care v-o alocati, pe o scala de la 1-10?. Argumentati.

Partea II. Impresii generale

1. Acest modul a întrunit aşteptările dumneavoastră?

☐ În totalitate ☐ În mare măsură ☐ În mică măsură ☐ Nu

2) Aveţi sugestii care să conducă la creşterea calităţii acestei unitati de invatare/modul?

3) Aveţi propuneri pentru alte unitati de invatare?

Vă mulţumim pentru feedback-ul dumneavoastră!