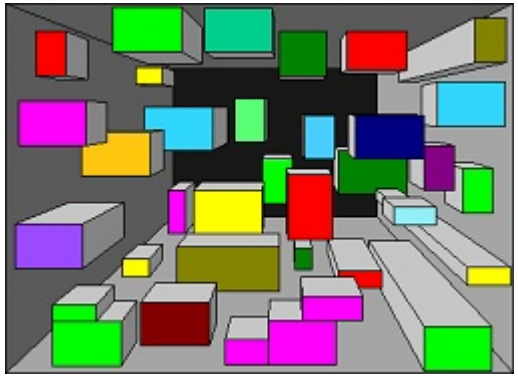
Structuri de Date

Anul universitar 2019 - 2020

Prof. Adina Magda Florea



Cursuri care vă învață să gândiți algoritmic și să programați

- Programarea calculatoarelor – I-1
- **Structuri de date** – I-2
- Analiza Algoritmilor - II - 1
- Programare orientată pe obiecte – II - 1
- Introducere în organizarea calculatoarelor și limbaj de asamblare - II - 1
- Proiectarea algoritmilor - II - 2
- Paradigme de programare – II - 2
- Algoritmi paraleli și distribuiți - III - 1

Conținut curs

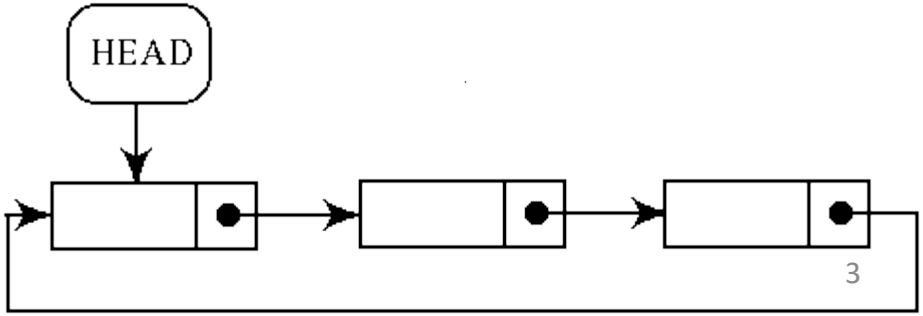
1. Mulțimi
2. Noțiuni elementare de analiza algoritmilor
3. Liste

    Reprezentarea listelor

    Liste simplu înlănțuite

    Liste dublu înlănțuite

    Liste circulare



Conținut curs

3. Stiva și coada

   Stiva

   Coada

   Diferite reprezentări

   Coadă de priorități

Conținut curs

5. Arbori

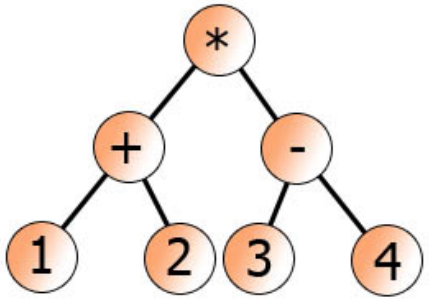
   Arbori binari

   Arbori binari de căutare

   Arbori AVL

   Arbori roșu și negru

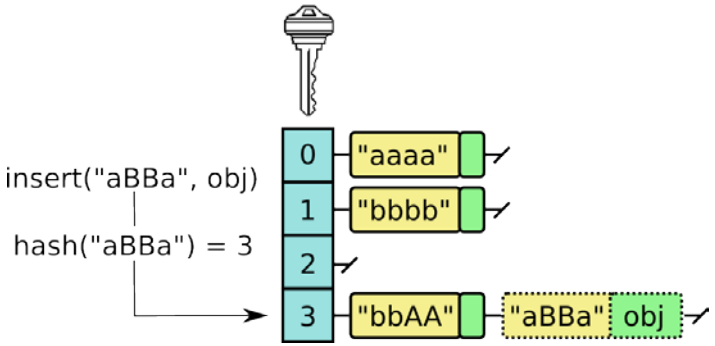
   Heap



((1+2)*(3-4))

Conținut curs

6. Tabele de dispersie



7. Grafuri

   - Reprezentarea grafurilor
   - Parcurgerea grafurilor
   - Algoritmi pe grafur

Bibliografie

- B. Kernighan, D. Ritchie. The C Programming Language, Prentice Hall, 1978
- R. Sedgewick. Algorithms in C Parts 1-4 Fundamental Data Structures, Sorting, Searching –, Addison-Wesley, 1998 (third edition)
- I.Mocanu, E.Kalisz. Structuri de date – variante de implementare in C, Ed.Universitara, 2012
- Th.Cormen, Ch.Leiserson, R.Rivest. Introducere în algoritmi. Ed.Computer Libris Agora, 2000 / Byblos, 2004

Cerințe și notare

- Laborator: 20 puncte
- 3 Teme de casă: 30 puncte
- Lucrare pe parcurs: 10 puncte
- Examen: 40 puncte

TOTAL: 100 puncte

Pentru promovare:

Minim 50% din punctajul pe parcurs (30 puncte)

Minim 50% din punctajul din examen (20 puncte)

Reguli

- Activitatea pe parcurs nu se reface pe parcursul acestui an universitar
- Copiatul va fi penalizat cu strictețe
- Copiat la examen sau la lucrare – examen pierdut și propunere pentru exmatriculare
- Copiat o temă de casă – anulat punctaj tema și mustrare
- Copiat două teme de casă – anulat tot punctajul pe parcurs, examen pierdut și mustrare

Curs Nr. 1

- Structuri de date
- Structuri de date abstracte
- ADT colecție și mulțime
- Structuri de date pentru ADT mulțime
- Implementare ADT mulțime
- Noțiuni elementare de analiza algoritmilor

Rezolvarea problemelor

Pentru rezolvarea unei probleme avem nevoie de:

- o reprezentare a universului problemei
- un set de instrumente pentru a transforma această reprezentare
- un criteriu de succes / terminare

Ce este o Structură de date?

O **structură de date** (SD) este un mod de organizare a datelor în memorie pentru accesul eficient la date

Exemple: vector, matrice, structură, listă înlănțuită

```
int  a[10];
struct point {int x; int y;}
struct key {char *word;  int count;}
struct key keytab[NKEYS];
struct node {int  item;  struct node *next;}
```

Ce este o Structură de Date Abstractă?

- O **structură de date abstractă** (SDA), în engleză **Abstract Data Type** (ADT), este o descrie logică a unui mod de organizare a datelor împreună cu operațiile specifice asociate
- Un ADT permite abstractizarea operațiilor fără a ține cont de detalii (de implementare) cât și încapsularea datelor
- Exemple: mulțime, stivă, coadă, graf

  push(stiva, elem) $\rightarrow$ stiva

  empty(stiva) $\rightarrow$ 1 / 0

Colecții și mulțimi

- O colecție este un grup de elemente de același tip in care pot exista duplicate.
- O mulțime este o colecție ce nu conține duplicate.
- O colecție (respectiv mulțime) este un ADT care poate fi descris prin operațiile de bază care se pot executa asupra acestei colecții
- Operatiile de baza asupra unei colecții pot fi descrise abstract, independent de tipul elementelor componente și de modul de reprezentarea în calculator.
- Pentru implementarea acestor operații avem nevoie de reprezentarea datelor în memorie, deci de o structură de date

Colecții și mulțimi - ADT

- Constructori, care au ca rezultat o colecție nouă sau modifică o colectie existentă
  - initializare () → colectie
  - adaugare (colectie, element) → colectie
  - eliminare (colectie, element) → colectie
  - reuniune (colectie, colectie) → colectie
  - intersectie (colectie, colectie) → colectie
  - diferenta (colectie, colectie) → colectie

Colecții și mulțimi - ADT

- Operații de caracterizare, care furnizează informații despre o colectie
  - vida(colectie) $\rightarrow$ 1 / 0 (da/nu)
  - cardinal (colectie) $\rightarrow$intreg
  - apartine (colectie, element) $\rightarrow$ 1 / 0
  - identice (colectie, colectie) $\rightarrow$ 1 / 0
  - include (colectie, colectie) $\rightarrow$ 1 / 0

Colecții și mulțimi - ADT

- Semantica acestor operații abstracte se poate stabili prin scrierea pseudocodului asociat unei operații

➢ adaugare (colectie, element) → colectie

colectie ← colectie + element

intoarce colectie

➢ adaugare (multime, element) → multime

daca apartine (multime, element) **atunci**

intoarce multime

altfel **intoarce** multime+ element

Mulțimi/colecții - Implementare

Avem nevoie de o structură de date

- Ce alegem?

  - Vector alocat static
  - Vector alocat dinamic
  - Listă înlănțuită

- Trebuie aleasă o reprezentare care să permită prelucrarea cât mai eficientă a elementelor colectiei.
- Daca operațiile de adăugare / eliminare sunt relativ rare, atunci se pot utiliza vectori de elemente. De obicei se poate stabili o limită superioară a numarului de elemente.

Vectorul elementelor din colecție poate fi alocat:

- static, în cazul în care limita superioară este cunoscută din faza de compilare și este valabilă în marea majoritate a cazurilor
- dinamic, în restul situatiilor

Mulțimi/colecții - Implementare

(a) Vector alocat static

```
#define CAP 100
typedef char Item;
typedef struct
   {  Item elem[CAP];
      long dim;
    }  Multime;

Multime  a;
a.elem[0]='a';
a.dim=1;
```

Mulțimi/colecții - Implementare

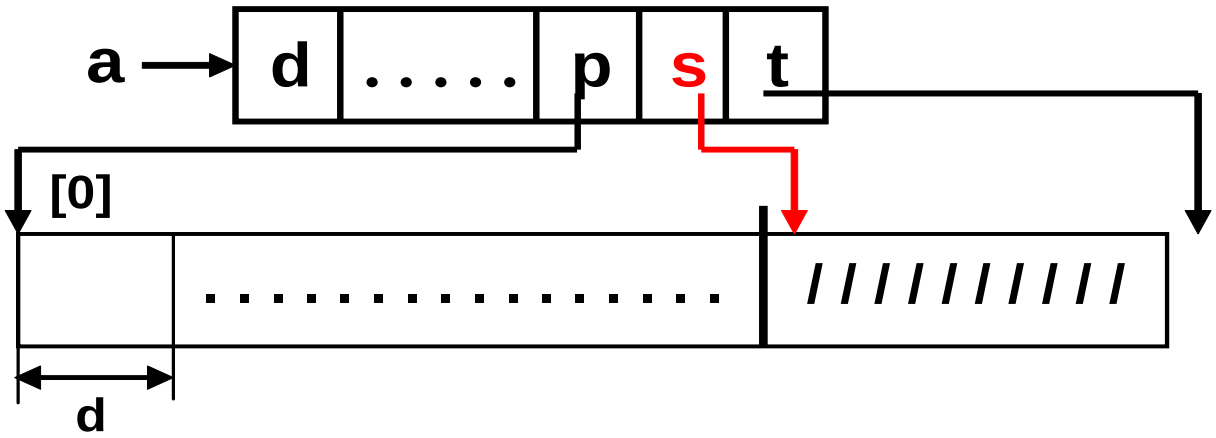
(b) Vector alocat dinamic

```
typedef struct
   {  Item *elem;
      long dim;
      long cap;
   }  Multime;

Multime  b; int  k;   /* citire k */
b.elem =(Item *) malloc(k*sizeof(Item));
b.elem[0]='a';
b.dim=1;
b.cap=k;
```

Mulțimi/colecții - Implementare

(c) Vector alocat dinamic, altă variantă

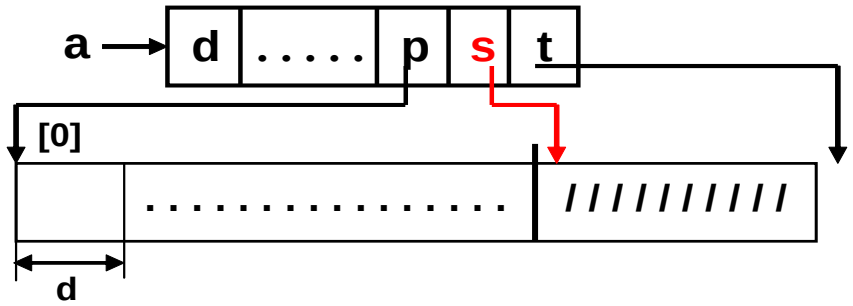



```
typedef  struct
{ size_t d;              /* dimensiune elemente */
……
Item *p, *s, *t;   /* adresa vector,
                      sfarsit zona utila, alocata */
} Multime;
```

Mulțimi/colecții - Implementare

Vector alocat dinamic, altă variantă

```
typedef  struct
{ size_t d;
  …….
  Item *p, *s, *t}  Multime;
```

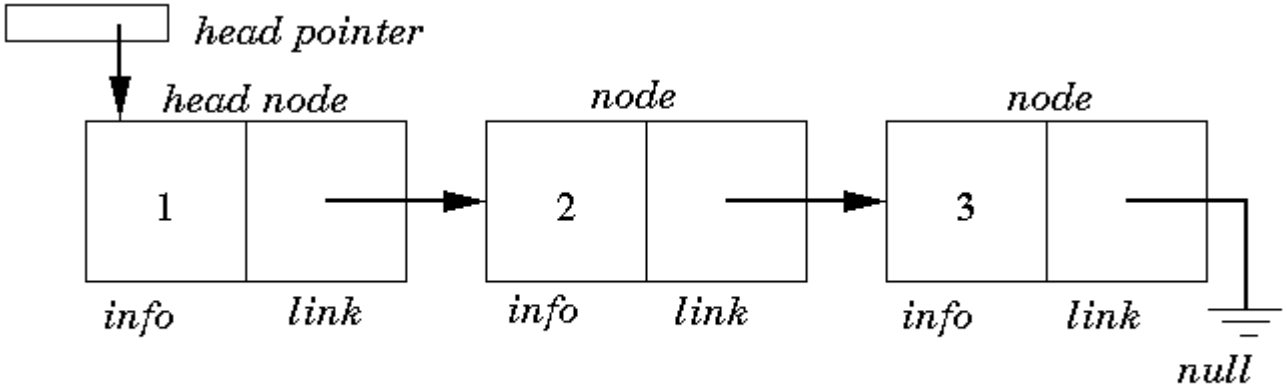


Multime *InitMultime(long n)

```
{  Multime *a;  size_t dim_elem;
   a = (Multime *) malloc(sizeof(Multime));
   a->d = dim_elem = sizeof(Item);
   a->p = (Item *) malloc(n*dim_elem);
   a->s = a->p;
   a->t = (Item *)(a->p)+ n*dim_elem;
    return a;
}
```

Mulțimi/colecții - Implementare

(d) Lista înlănțuită – cursul viitor



Căutare în colecții/mulțimi

- Căutarea se realizează pe baza unui criteriu specific
- Metoda de căutare depinde de modul de memorare a elementelor:
  - elemente nesortate

    → căutare secvențiala, cu oprire la
    - sfârșitul colecției ➔ esec
    - îndeplinirea criteriului ➔ succes
  - elemente sortate conform criteriului de căutare

    → căutare secvențiala, cu oprire la
    - sfârșitul colecției sau întalnire succesor ➔ esec
    - îndeplinirea criteriului ➔ succes

    → căutare binară

Căutare în colecții/mulțimi

Algoritm Cautare_Secventiala (cheie) **intoarce** succes/pozitie sau esec

```
{ initializari
  cat timp mai exista elemente de analizat repeta
      daca elementul curent = cheie
      atunci intoarce succes sau pozitie
      altfel avans la urmatorul element;
  intoarce esec
}
```

ADT apartine (colectie, element) $\rightarrow$ 1 / 0

Căutare în colecții/mulțimi

Algoritm Cautare_Secventiala_Sortat (cheie) **intoarce** succes/pozitie sau esec

```
{ initializari
  cat timp mai exista elemente de analizat repeta
      daca elementul curent = cheie
      atunci intoarce succes sau pozitie
      altfel
          daca elementul curent > cheie
          atunci intoarce esec
          altfel avans la urmatorul element;
  intoarce esec
}
```

Căutare în colecții/mulțimi

Algoritm Cautare_Binara_Sortat (cheie) **intoarce** succes/pozitie sau esec

{ initializari inf, sup

cat timp mai exista elemente de analizat (inf<=sup)

repeta

determina *m* – mijlocul zonei

daca elementul de pe pozitia *m* = cheie

atunci intoarce succes sau pozitie

altfel

daca elementul de pe pozitia *m* < cheie

atunci sup ← *m*

altfel inf ← *m*

intoarce esec

}

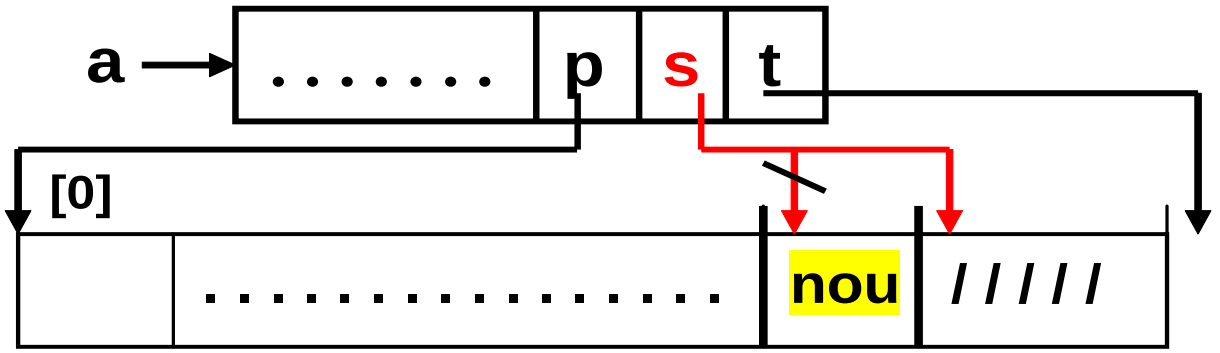
Adăugare element

- Adăugarea unui element se poate realiza numai dacă există spațiu suficient

  (**`a->s < a->t`**).

- Modul în care se realizează adăugarea depinde de modul de memorare a elementelor:

  a. elemente nesortate → adăugare la sfârșit

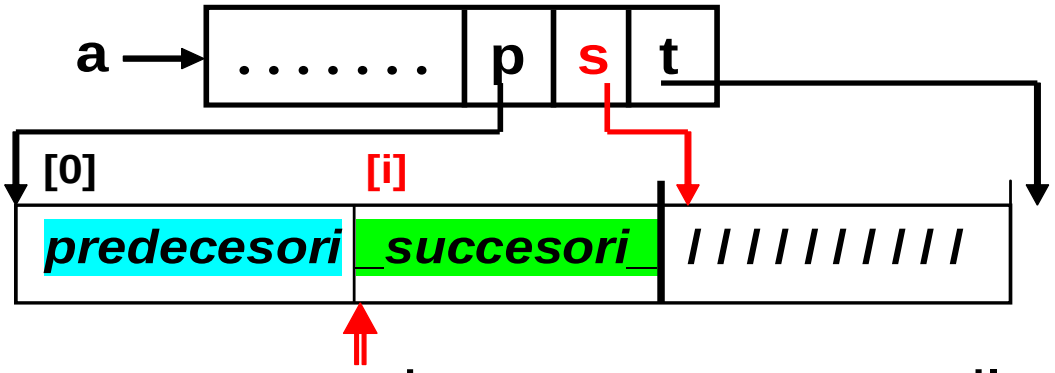


Adăugare element

b. elemente sortate → inserare în interiorul vectorului, între predecesori și succesori, realizată în 3 etape:

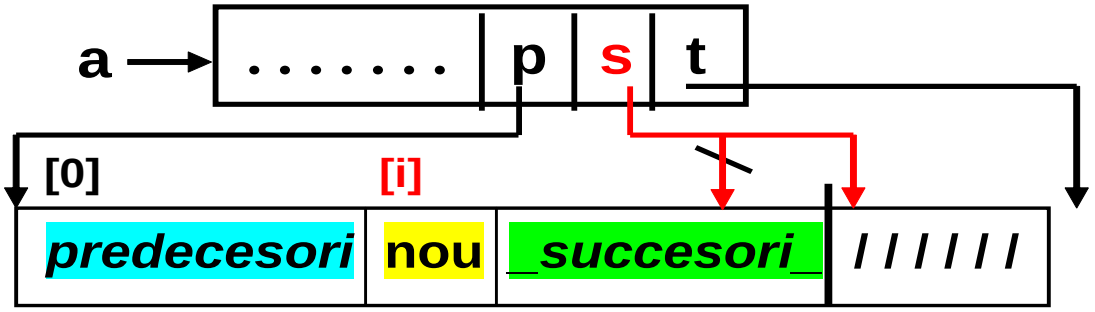
1. căutarea locului de inserare

Se pot face simultan



2. deplasarea spre dreapta a succesorilor

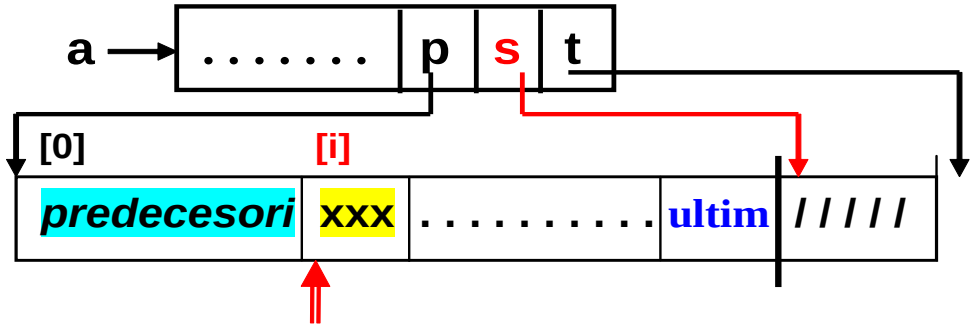
3. inserarea propriu-zisă

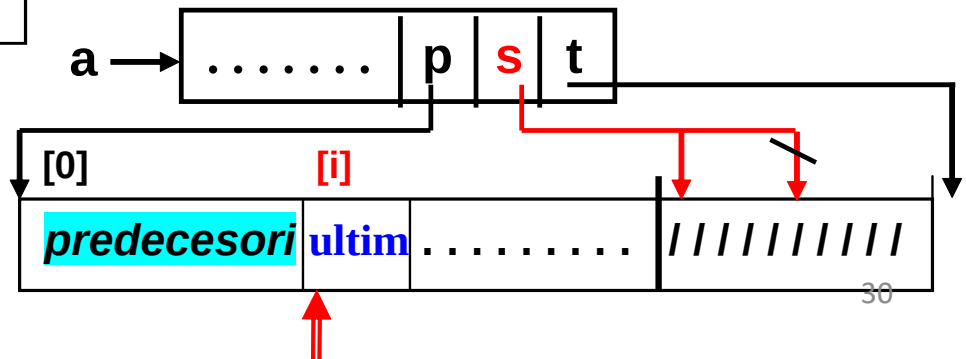


Eliminare element

- Eliminarea unui element se realizează în două etape:
- 1. localizare (căutare)
- 2. eliminare, dacă a fost găsit
- Modul în care se realizează eliminarea depinde de modul de memorare a elementelor:

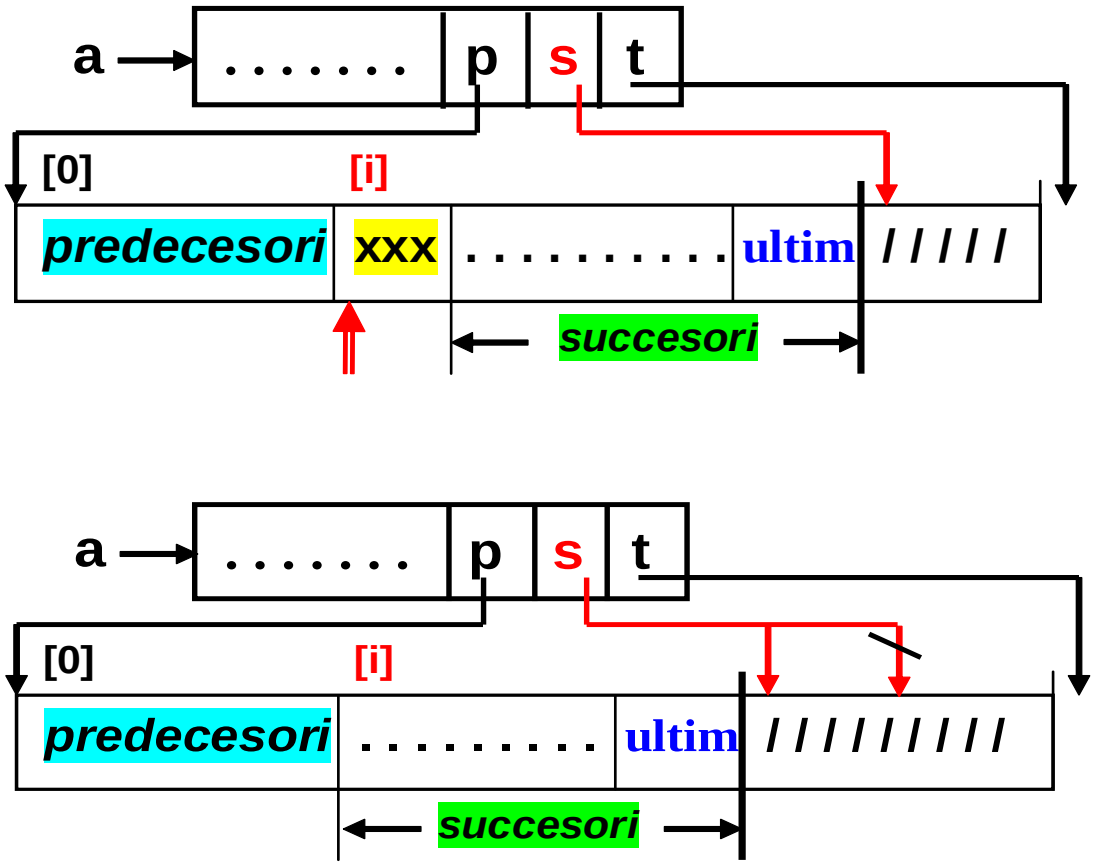
a. elemente nesortate → înlocuire prin ultimul element





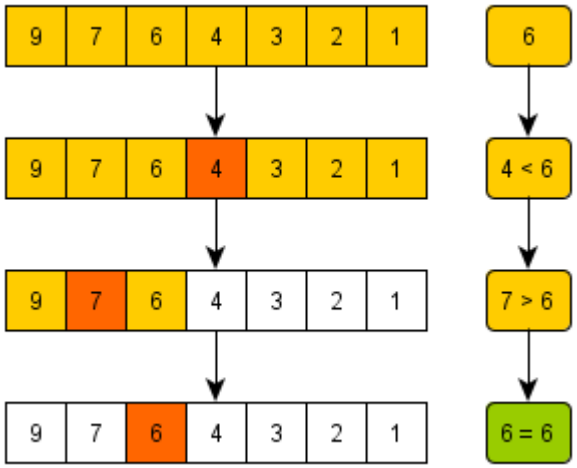
Eliminare element

b. elemente sortate → deplasare spre stânga a succesorilor



Mulţimi sortate

- De ce este mai bine să avem elementele sortate?
- De ce s-a inventat căutarea binară?

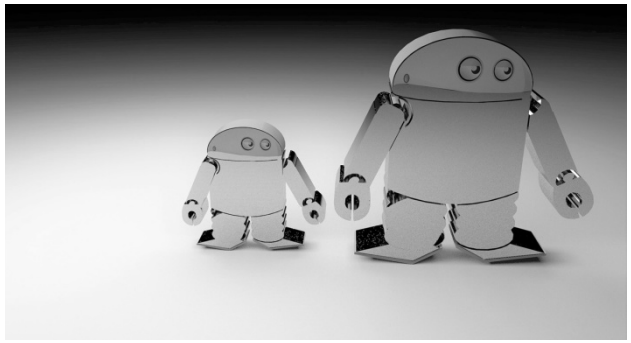


Noțiuni elementare de analiza algoritmilor

- Eficiența algoritmilor
- Considerăm atât operațiile care se execută cât și datele pe care se rulează
- Datele de intrare: cele reale, aleatoare, perverse
- Comparare a 2 algoritmi – analiza algoritmilor
- Identificarea operațiilor abstracte pe care se bazează algoritmul, separând astfel analiza de implementare
- Timpul de rulare – performanța algoritmului și performanța calculatorului

Noțiuni elementare de analiza algoritmilor

- Cazul mediu (average case)
- Cazul cel mai defavorabil (worst case)
- Cazul cel mai defavorabil poate fi rar întâlnit în practică (sau niciodată)
- Ambele ne pot da o indicație asupra performanțelor algoritmilor



Funcții de creștere

- N – un parametru al datelor de intrare care afectează timpul de rulare (running time)

- Cei mai mulți algoritmi discutați la curs vor avea timpul de execuție proporțional cu una din următoarele funcții

1. Cele mai multe instrucțiuni sunt executate o dată sau numai de câteva ori – timp constant
2. logN – probleme descompuse în subprobleme – timp logaritmic

Funcții de creștere

3. N - O prelucrare se efectuează pe fiecare element de intrare – timp liniar
4. N*logN – algoritmul rezolvă problema prin descompunere în subprobleme, rezolvă subproblemele și combină apoi intrarea
5. N^2 – algoritmul prelucrează toate perechile formate cu datele de intrare – timp cuadratic
6. N^3 - algoritmul prelucrează toate tripletele formate cu datele de intrare – timp cubic
7. 2^N – timp exponențial

Timpul de rulare

Timpul de rulare a unui algoritm va fi de obicei una din aceste funcții înmulțită cu o constantă plus o altă constantă

N ~ 1 milion

Op/sec	N	N*lnN	N^2
10^6	sec	sec	sapt
10^9	instant	instant	ore
10^{12}	instant	instant	sec

N ~ 1 miliard

Op/sec	N	N*lnN	N^2
10^6	ore	ore	nic
10^9	sec	sec	decade
10^{12}	instant	instant	sapt

Notația Big-Oh (O mare)

O funcție **g(N)** se spune că este de ordinul O(f(N)) dacă există constantele $\mathbf{c_0}$ și $\mathbf{N_0}$ astfel încât

$$\mathbf{g(N) < c_0 * f(N)} \text{ pentru } \forall \mathbf{N > N_0}$$

$\mathbf{c_0}$ si $\mathbf{N_0}$ se referă la detalii de implementare

De obicei dorim un algoritm cu o funcție de creștere cât mai mică

Totuși, în anumite cazuri putem prefera, de ex, un algoritm cu N^2 fată de unul cu logN dacă valorile tipice ale lui N sunt tot timpul sub N_0

Evaluare algoritmi

Algoritm Cautare_Secventiala

```
int search(int a[], int v, int l, int r)
{  int i;
   for (i=l; i<=r; i++)
          if(v==a[i]) return i;
    return -1;
}
```

- Căutarea secvențială examinează N numere pt căutarea fără succes (worst case) și aproximativ N/2 numere pentru căutarea cu succes (average case)
- **O(N)**

Evaluare algoritmi

Algoritm Cautare_Binara

```
int search(int a[], int v, int l, int r)
{  while (r >= l)
        { int m = (l+r)/2;
          if (v==a[m]) return m;
          if (v<a[m]) r=m-1;
          else l=m+1;
        }
   return -1;
}
```

- Căutarea binară nu examinează mai mult de logN+1 numere (worst case)
- **O(logN)**

Evaluare algoritmi

Algoritm Sortare Bubble

```
void Bubble(int a[], int N)
{  int i, j, t;
   for(i = N; i >= 1; i--)
       for(j = 2; j <= i; j++)
            if( a[j-1] > a[j] )
             { t=a[j-1]; a[j-1]=a[j]; a[j]=t; }
}
```

- **$O(N^2)$** cazul mediu și cel mai defavorabil

Evaluare algoritmi

- **Algoritm sortare Quicksort** - Folosește paradigma **“divide-and-conquer”**
- **Divide:** daca S (șirul de sortat) are cel puțin două elemente, selectează un element **x** din S – **pivot** (tipic ultimul). Împarte elementele din S în 3 părți:
  - Elemente **<** decât **x** (L)
  - Elemente **=** cu **x** (E)
  - Elemente **>** ca **x** (G)
- **Conquer:** sortează L și G
- **Combine:** Reface S punând în S, în ordine: L, E și G
- **$O(N^2)$** cazul cel mai defavorabil
- **O(NlogN)** cazul mediu