

Structuri de Date

Anul universitar 2019-2020

Prof. Adina Magda Florea



Curs Nr. 6

- Căutare
- Operații de bază cu structuri de căutare
- Reprezentarea structurilor de căutare
- Arbori binari de căutare

1. Căutare

- Există un număr imens de aplicații în care avem nevoie să căutăm informații
 - Căutare clienți într-o bază de date
 - Căutare filme preferate
 - Căutare cuvinte în pagini Web
 - Căutare calculator preferat pe un site etc.
- Avem nevoie de o structură care să permită căutarea elementelor după o **cheie de căutare**
- Baze de date
- Tabele de simboluri
- Dicționare
- **ADT Structură de căutare**

ADT Structură de căutare

- Structură de date abstracte **Structură de căutare**
- Poate conține numai **cheia de căutare** sau o **structură care să conțină cheia de căutare**

Ionescu	George	341	C4
Popescu	Mihai	342	C3
Georgescu	Ion	341	C4

2. Operații de bază cu SC

- Initializare
- Test SC vidă
- Inserare element
- Căutare element
- Căutare și inserare
- Ștergere element
- Vizitare elemente
- Copiere SC
- Distrugere SC



3. Reprezentare SC

- Anumiți algoritmi nu implică o **ordonare a cheilor**, alții necesită ordonarea cheilor
- Unele aplicații cer **chei unice** (distincte), altele permit chei duplicate
- Pentru **chei duplicate**
 - Structură de date primară \Rightarrow conține numai elemente cu chei distincte și menține o listă a elementelor cu aceeași cheie; la o căutare întoarce toate elementele cu aceeași cheie
 - Structura de date primară conține chei duplicate și la o căutare se întoarce primul găsit
 - Există și alte posibilități (de ex crearea de chei unice)

Reprezentare SC

- Reprezentare vectorială \Rightarrow **vector de elemente**
 - Sortate
 - Nesortate
- Reprezentare cu **liste înlanțuite**
 - Sortate
 - Nesortate
- Dacă valorile cheilor sunt întregi pozitivi mai mici ca M și elementele au chei distincte, atunci SC se poate implementa ca un **vector indexat după chei**
- Căutare prin **adresare directă**

Adresare directă

```
typedef struct client
{   int cheie;
    char *nume;} Item;

static Item *sc;
static int M = 100;
static Item NULLItem = {0,NULL};
void scInit(int M)
{   int i;
    sc = malloc (M * sizeof(Item));
    for (i=0; i<M; i++) sc[i]=NULLItem;
}
```


Adresare directă

```
int scInsert(Item elem)
{
    int ch = elem.cheie;
    if(ch < 0 || ch > M) return -1;
    /* nu mai este spatiu de inserare */
    if (sc[ch].cheie != NULLItem.cheie)
        return 0;
    /*elementul este deja */
    sc[ch] = elem; return 1;
    /* insereaza element */
}
```

```
Item scSearch(int cheie)
{
    return sc[cheie]; }
```

Reprezentare SC

Vector nesortat sau listă nesortată – inserare rapidă dar căutarea durează

Vector sortat sau listă sortată – căutare rapidă dar inserarea durează

- **Arbori binari de căutare** – ideea căutării binare combinată cu flexibilitatea listelor înlănțuite



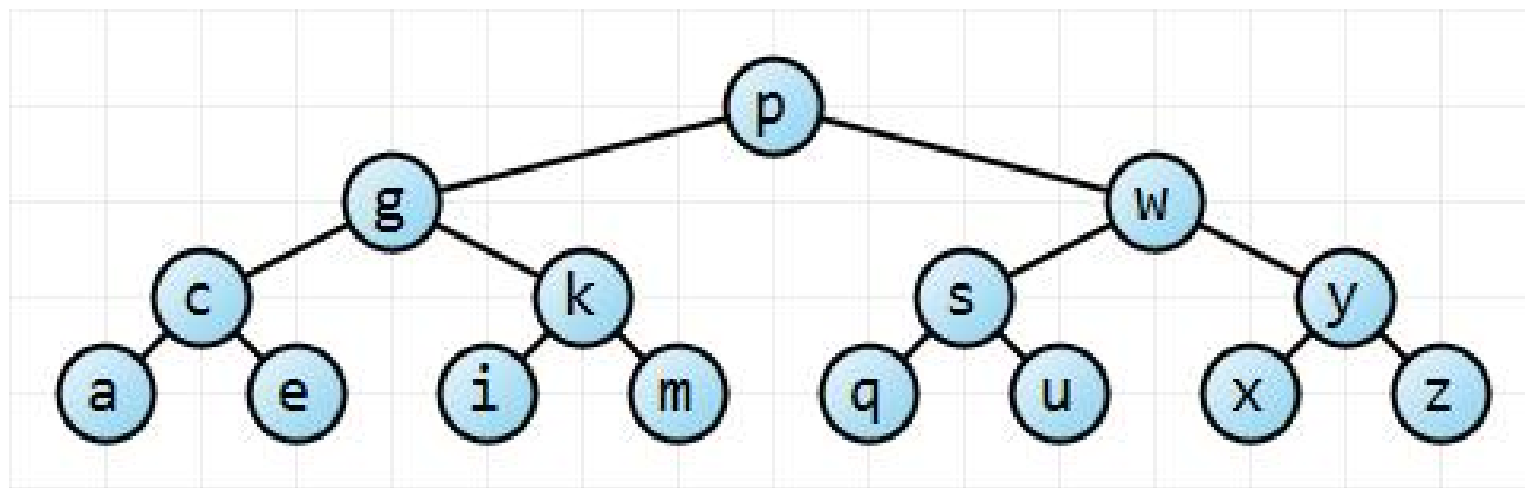
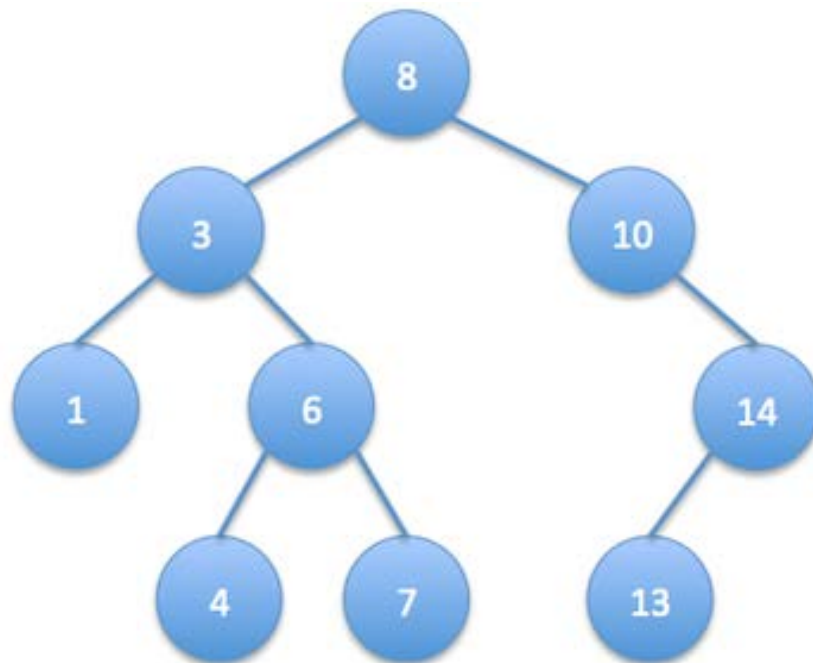
Performanțele depind de reprezentare

	Inserare (caz defavorabil)	Căutare (caz defavorabil)	Inserare (caz mediu)	Căutare și găsit (caz mediu)	Căutare și eșec (caz mediu)
Vector indexat după chei	1	1	1	1	1
Vector sortat	N	N	N/2	N/2	N/2
Listă sortată	N	N	N/2	N/2	N/2
Vector nesortat	1	N	1	N/2	N
Căutare binară	N	lgN	N/2	lgN	lgN
Arbori binari de căutare	N	N	lgN	lgN	lgN
Arbori echilibrați	lgN	lgN	lgN	lgN	lgN
Tabele de dispersie	1	N* (f puțin probabil)	1	1	1

N – numărul de elemente

4. Arbori binari de căutare

- Combină eficiența căutării binare cu flexibilitatea structurilor înlănțuite într-o aceeași structură de date
- **Definiție.** Un **arbore binar de căutare** (BST) este un arbore binar care are o **cheie asociată fiecărui nod** cu proprietatea că pentru fiecare **nod intern**:
 - cheia din nod este **mai mare** (sau egală) decât **orice cheie din subarborele stâng**
 - cheia din nod este **mai mică** (sau egală) decât **orice cheie din subarborele drept**



4.1 Reprezentare arbori binari de căutare

Aceeași cu cea a arborilor binari

```
typedef int Item;
```

```
typedef struct node {
```

```
    Item elem;
```

```
    struct node *lt, *rt; } TreeNode, *TTree;
```

sau

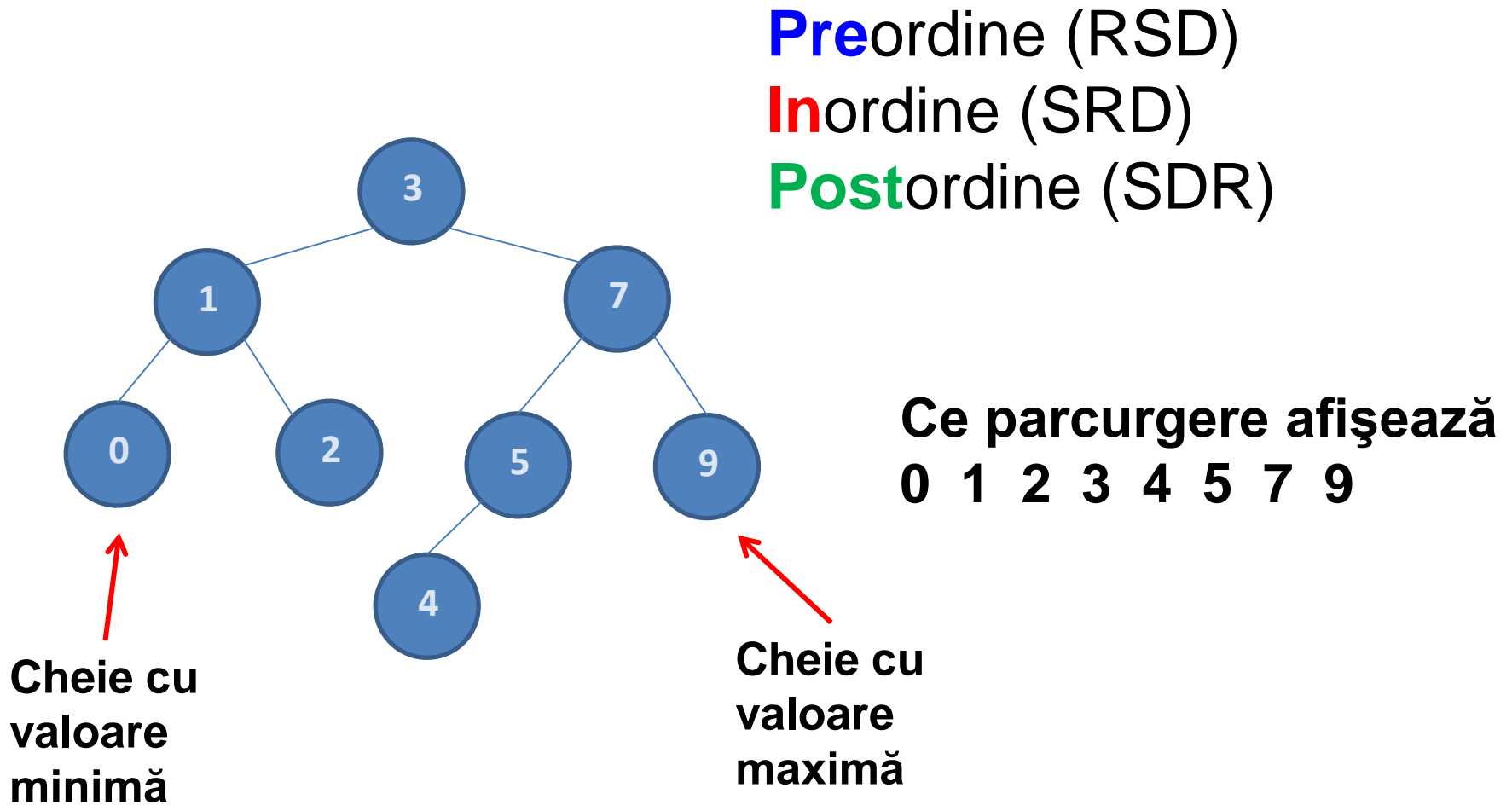
```
typedef struct node *TLink;
```

```
typedef struct node {
```

```
    Item elem;
```

```
    TLink lt, rt; } TreeNode;
```

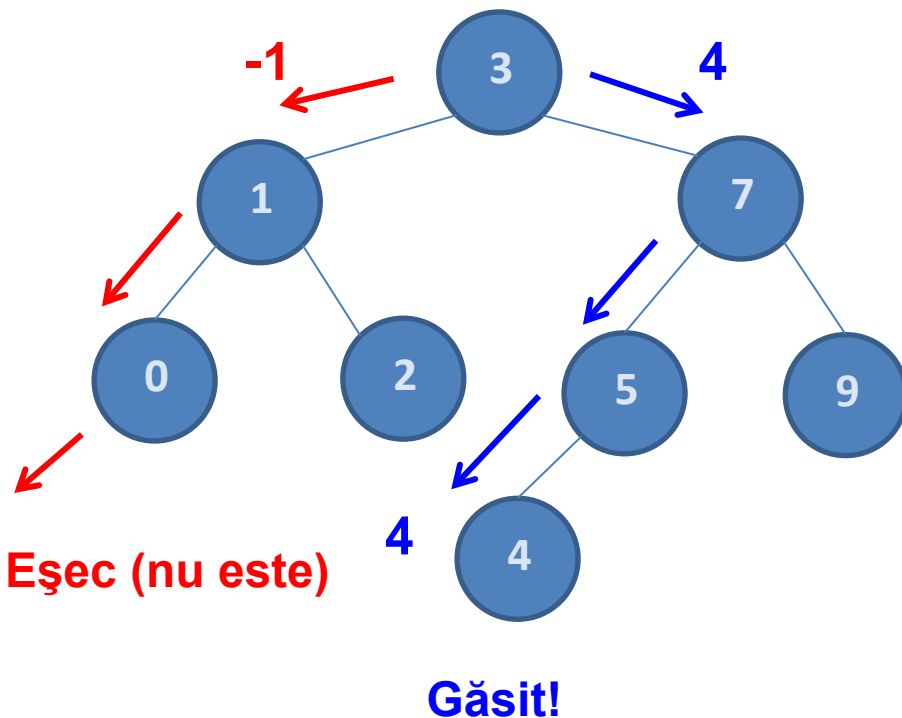
4.2 Parcurgere Arbore Binar de Căutare (ABC)



4.3 Căutare element într-un ABC

Căutare 4

Căutare -1



Algoritm Cauta(Arb, cheie)

intoarce pointer la nod

daca Arb ese vid **intoarce** null

daca cheie = elem din nod curent
atunci intoarce nod (Gasit!)

daca cheie < elem din nod curent
atunci Cauta(s_stg, cheie)
altfel Cauta(s_dr, cheie)

Căutare element într-un ABC

```
TLink BTSearch(TLink t, Item cheie)
{
    if(t == NULL) return NULL;
    if (t->elem == cheie) return t;
    if(cheie < t->elem)
        return BTSearch(t->lt,cheie);
    return BTSearch(t->rt,cheie);
}
```

N.B. Se poate și varianta nerecursivă

4.4 Găsire element maxim și minim

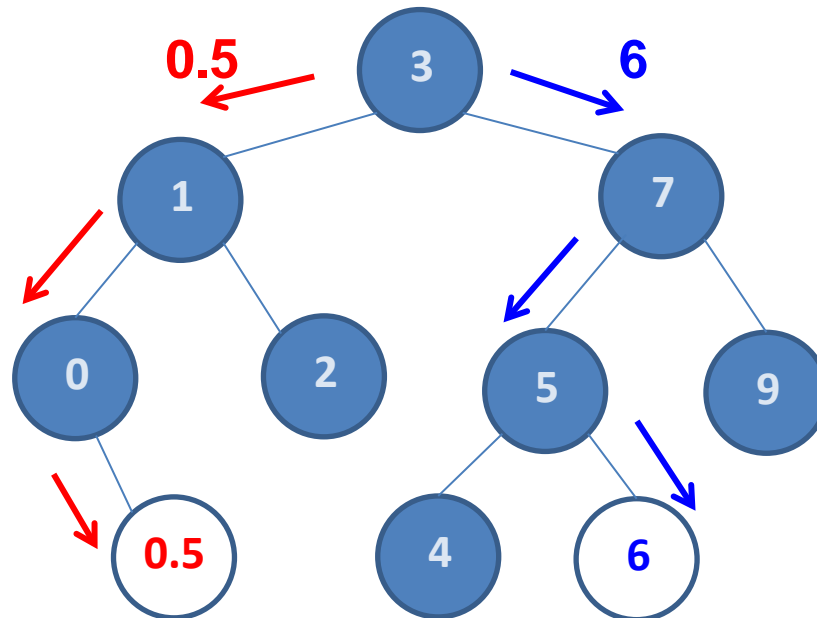
```
TLink findMax(TLink t)
    /* varianta nerecursiva */
{
    if (t!=NULL)
        while(t->rt != NULL) t=t->rt;
    return t;
}
```

```
TLink findMin(TLink t)
    /* varianta recursiva */
{
    if(t == NULL) return NULL;
    else
        if (t->lt == NULL) return t;
        return(findMin(t->lt));
}
```

4.5 Inserare element într-un ABC

Inserare 6

Inserare 0.5



Inserare element într-un ABC

Algoritm BTInsert(arbore, cheie) **intoarce** pointer la arb

daca arbore vid **atunci**

construieste un arbore cu 1 nod cu cheie

intoarce nod

daca cheie < element din nod

atunci s_stg al nod =

BTInsert(s_stg al nod, cheie)

altfel s_dr al nod =

BTInsert(s_dr al nod, cheie)

intoarce nod

sfarsit

Inserare element într-un ABC

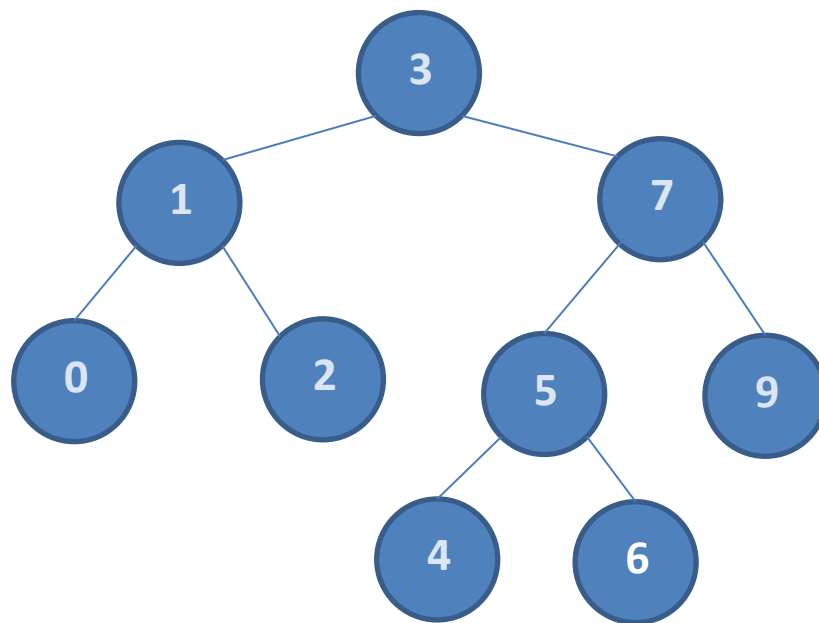
```
TLink BTInsert(TLink t, Item cheie)
{
    if(t == NULL)
        return BuildNode(cheie, NULL, NULL);
    if(cheie < t->elem)
        t->lt = BTInsert(t->lt, cheie);
    else t->rt = BTInsert(t->rt, cheie);
    return t;
}
```

N.B. Se poate și varianta nerecursivă.

Obs referitoare la tratarea cheilor multiple

4.6 Construire ABC prin inserări repetate

Pt **ordinea de inserare** a cheilor: 3, 1, 7, 2, 5, 6, 0, 4, 9, 5
rezultă arborele binar de cautare de mai jos



Dar pt. ordinea de inserare a cheilor: 0, 1, 2, 3, 4, 5, 6, 7, 9
ce ABC rezultă?

Construire ABC prin inserări repetate

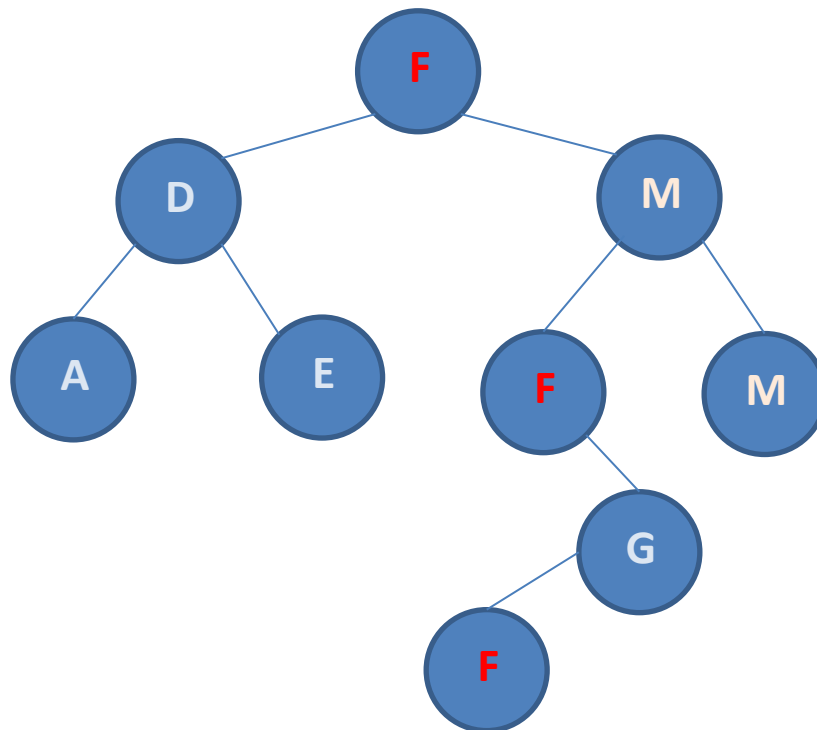
Ne dorim arbori cu **înălțimi cât mai mici** pentru același număr de chei

Numărul de căutări este proporțional cu nivelul arborelui / cu înălțimea acestuia

	Inserare (caz defavorabil)	Căutare (caz defavorabil)	Inserare (caz mediu)	Căutare și găsit (caz mediu)	Căutare și eșec (caz mediu)
Arbori binari de căutare	N	N	$\lg N$	$\lg N$	$\lg N$
Arbori echilibrați	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$

Construire ABC prin inserări repetate

- ❑ Inserarea **cheilor repetate** poate conduce la arbori foarte înalți
- ❑ În plus, aceste chei nu sunt întotdeauna grupate în arbore
- ❑ Ordinea de inserare: F, D, M, F, M, A, E, G, F conduce la:

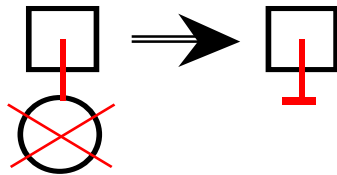


4.7 Eliminare element dintr-un ABC

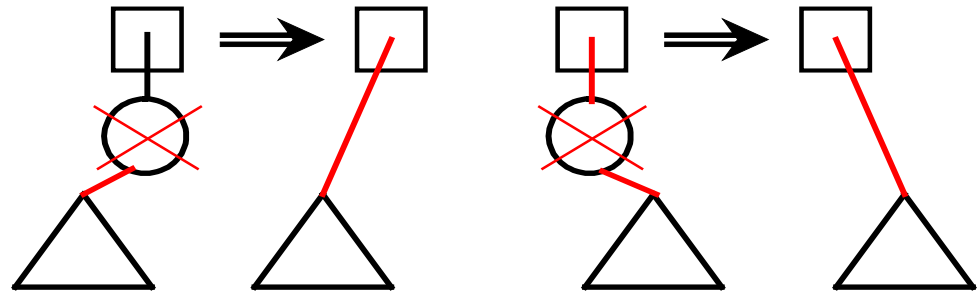
- Eliminarea unui element (nod) trebuie să se facă astfel încât să se respecte condiția de ABC după eliminare
- Există mai multe cazuri care trebuie tratate:
 - Eliminare frunză
 - Eliminare nod de ordinul 1 (are un singur subarbore)
 - Eliminare nod de ordinul 2 (are ambii subarbori)
- Poate rezulta o dezechilibrare
- Pentru nodurile de ordinul 2: eliminare cu pivot care nu dezechilibrează arborele

Eliminare element dintr-un ABC

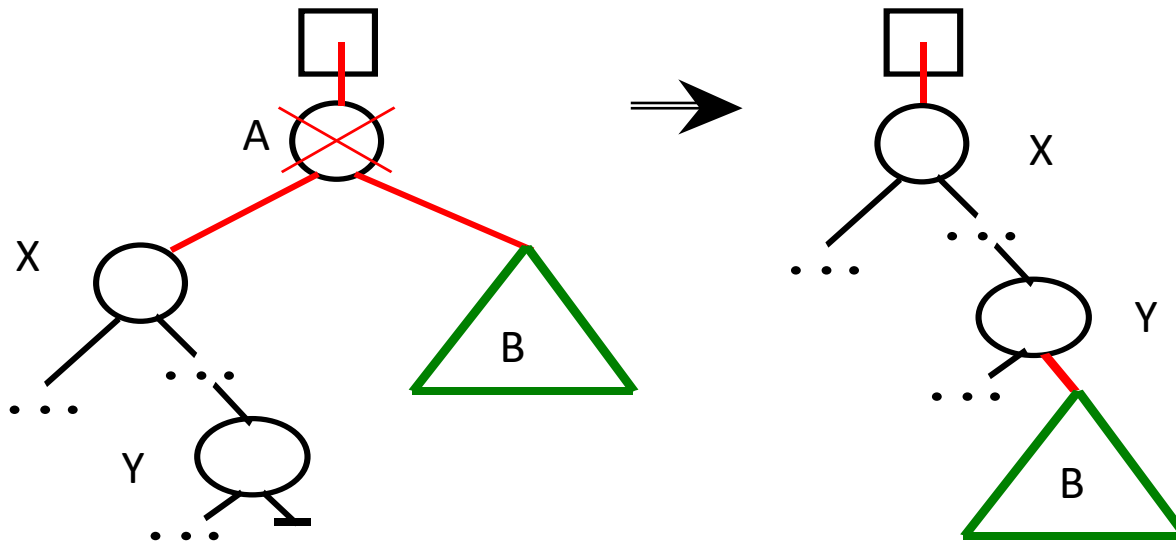
- frunza



- de ordin 1



- de ordin 2 – eliminare (posibila dezechilibrare)



Eliminare element – fct ajutatoare constructie

Algoritm BuildTree (s_stang, s_drept)

intoarce pointer la noul nod radacina

daca s_stang = null **atunci intoarce** s_drept

daca s_drept = null **atunci intoarce** s_stang

arb = s_stang

cat timp arb nu este vid **repeta**

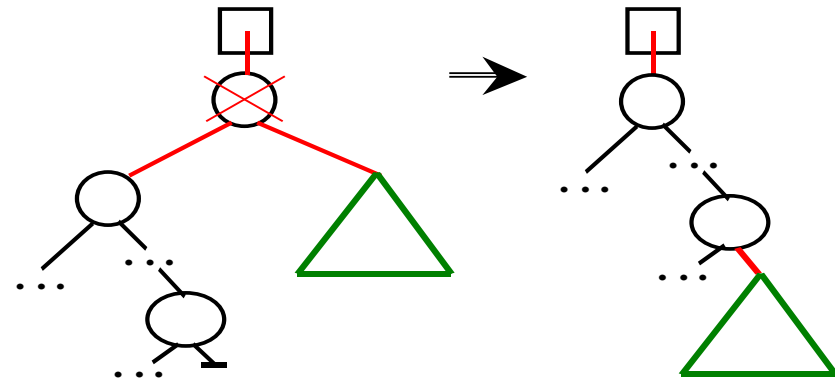
 parinte = arb

 arb = s_dr al arb

 s_dr al parinte = s_drept

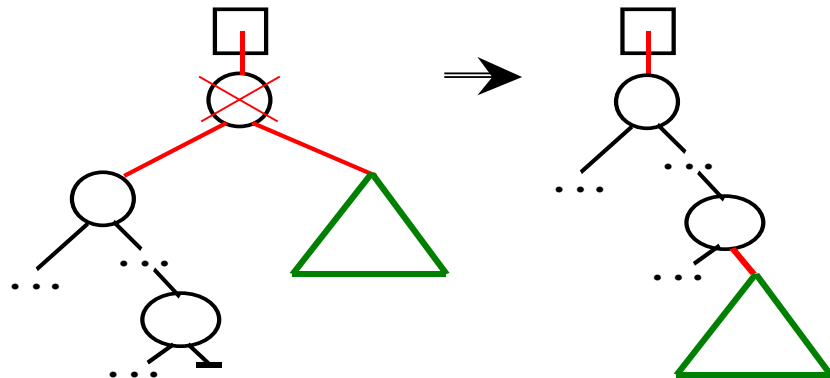
intoarce s_stang

sfarsit



Eliminare element – fct ajutatoare constructie

```
TLink BuildTree(TLink ltree, TLink rtree)
{
    TLink t=ltree, parent;
    if (ltree == NULL) return rtree;
    if (rtree == NULL) return ltree;
    while(t != NULL)
        { parent = t; t = t->rt; }
    parent->rt = rtree;
    return ltree;
}
```



Eliminare element

Algoritm BDelete(arbore, cheie) **intoarece** pointer la arb

parinte = NULL

daca arbore vid **atunci intoarce null** /* arbore vid */

/* caut nod cu cheie si tin minte parintele lui */

cat timp arbore nevid **repeta**

daca cheie = element nod curent

atunci break /* am gasit nodul de eliminat*/

 parinte = nod curent

daca cheie < element nod curent

atunci arbore = subarbore stang

altfel arbore = subarbore drept

daca arbore vid **atunci intoarce null**

/* nu am gasit cheia */

Eliminare element

```
daca parinte = NULL / *cheia gasita este in radacina */  
atunci x= BuildTree(subarbore stang, subarbore drept)  
    elibereaza spatiu nod, intoarce x  
daca cheia este frunza atunci x = null /*cheia frunza */  
    /* cheia nici radacina nici frunza */  
altfel x = BuildTree(subarbore stang, subarbore drept)  
daca elem din s_dr al parinte = cheie /* afla unde */  
atunci s_dr al parinte = x  
altfel s_stg al parinte =x  
    elibereaza spatiu nod  
intoarce arbore  
sfarsit
```

Eliminare element

```
TLink BDelete(TLink t, Item cheie)
{
    TLink parent=NULL, p=t, x;
    if (p == NULL) {printf("arb vid\n");
                    return NULL;}
    while(p != NULL)
    {
        if(p->elem == cheie) break;
        parent = p;
        if(cheie < p->elem) p = p->lt;
        else p = p->rt;
    }
    if (p ==NULL) { printf("cheia nu este");
                  return NULL;}
```

Eliminare element

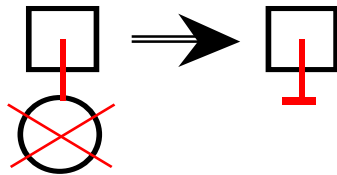
```
if(parent ==NULL) /*cheia gasita este in
                    radacina */
{ x = BuildTree(p->lt,p->rt); free(p);
                                return x;}

if(p->lt == NULL && p->rt == NULL)
    x = NULL; /*cheia este o frunza */
else x = BuildTree(p->lt, p->rt);

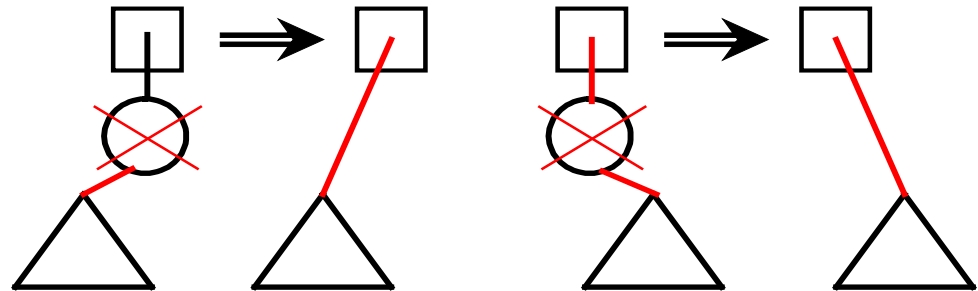
if (parent->rt->elem == cheie)
    parent->rt = x;
else parent->lt = x;
free(p);
return t;
}
```


Eliminare element dintr-un ABC

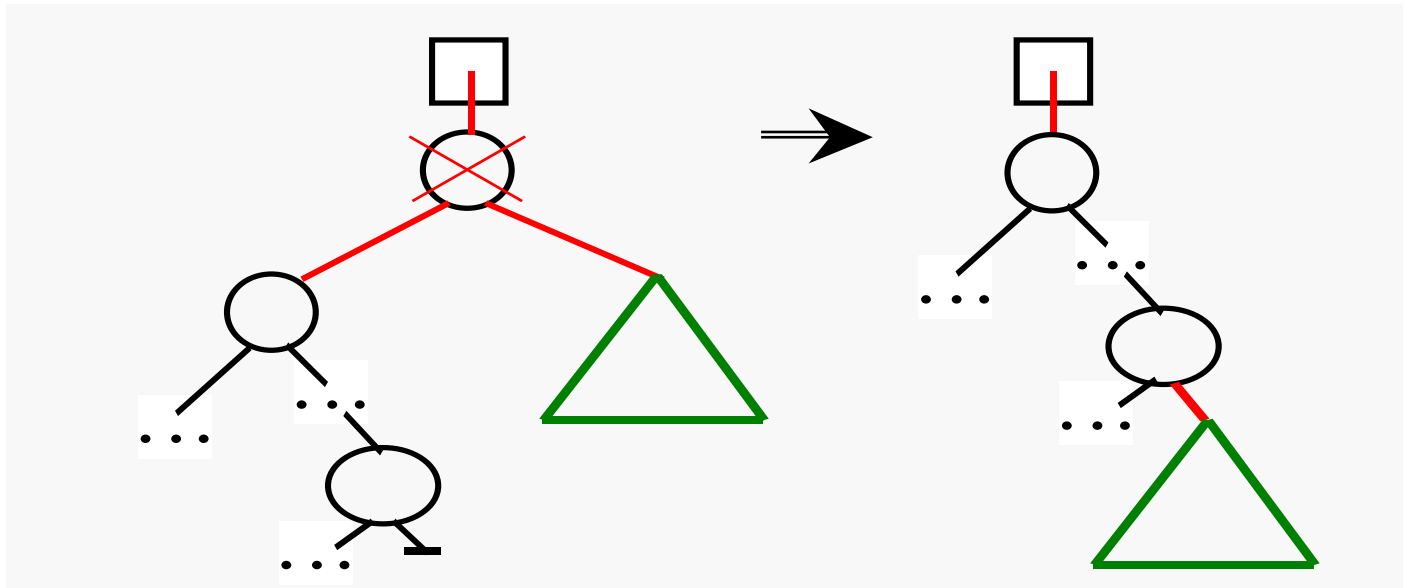
- frunza



- de ordin 1

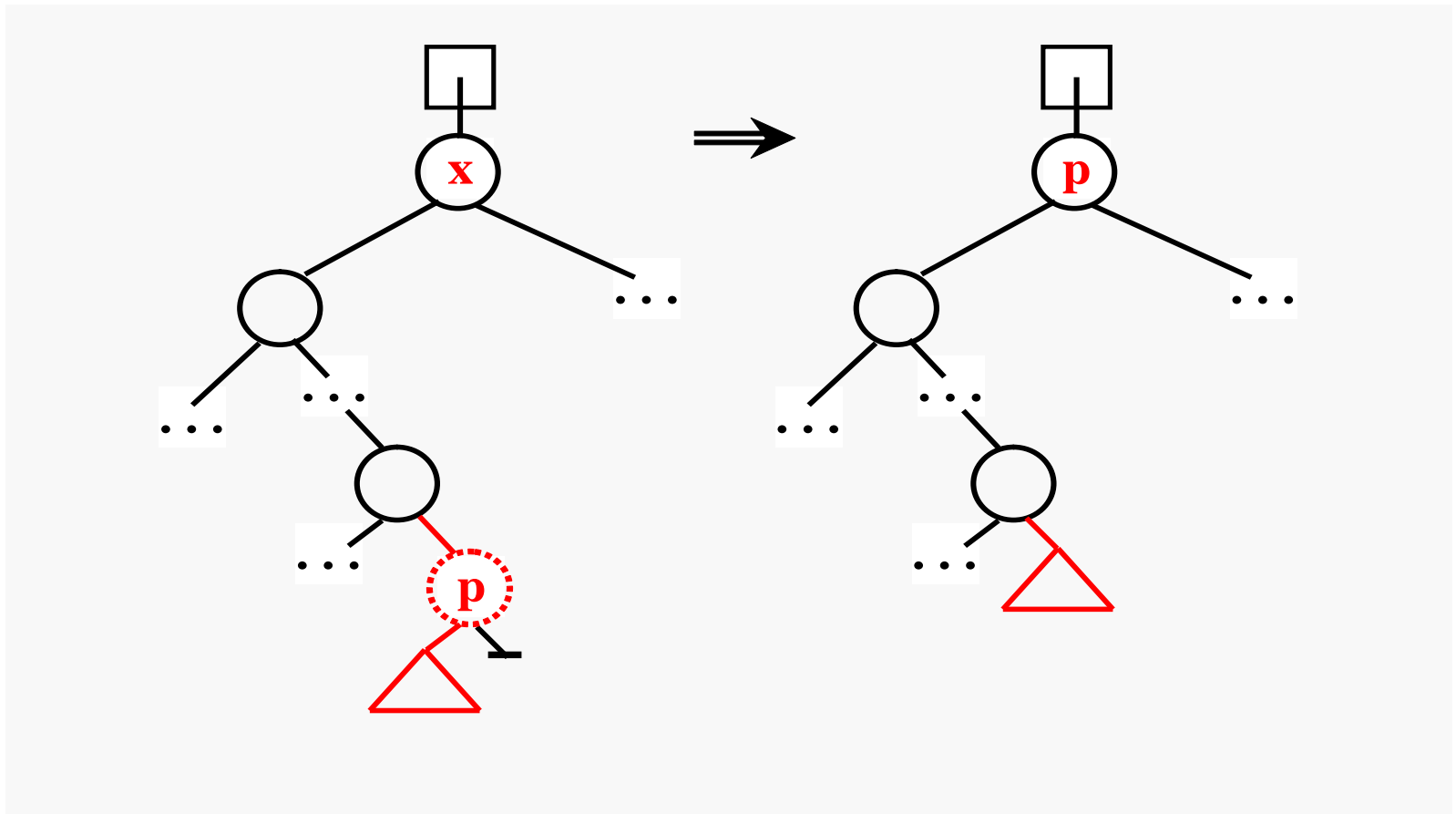


- de ordin 2 – eliminare cu posibila dezechilibrare



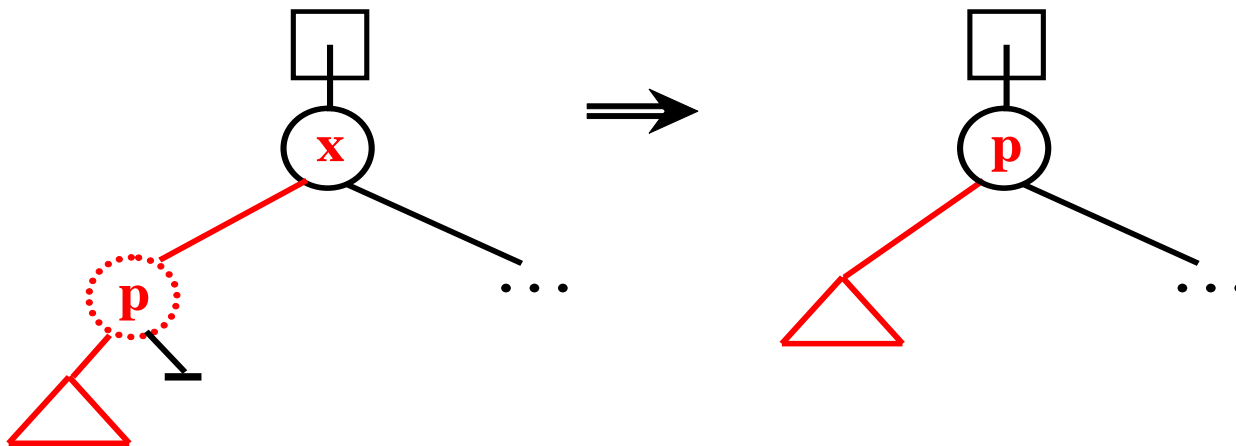
Eliminare element cu pivot

- de ordin 2 – eliminare cu pivot stanga, cazul general



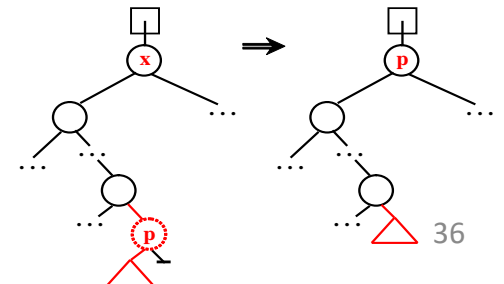
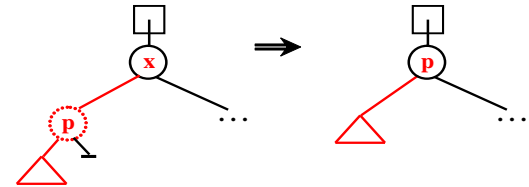
Eliminare element cu pivot

- de ordin 2 – eliminare cu pivot stanga, caz particular



Eliminare element cu pivot - fct ajutatoare constructie

```
TLink BuildTreePivot(TLink ltree, TLink rtree)
{
    TLink t=ltree, parent;
    if (ltree==NULL) return rtree;
    if (rtree==NULL) return ltree;
    if(ltree->rt==NULL)
        {ltree->rt=rtree; return ltree;}
    /* aici ltree si pivot sunt aceeasi */
    while(t->rt != NULL) /* cu t caut pivot */
    { parent=t; t=t->rt; }
    /* la iesirea din while am gasit pivot punctat de t */
    parent->rt=t->lt; t->lt=ltree; t->rt=rtree;
    return t;
}
```



Eliminare element cu pivot – altă variantă

```
TLink BDeleteR(TLink t, Item cheie)
{
    TLink temp, child;
    if(t==NULL) {printf("not found \n"); return NULL;}
    if(cheie < t->elem)
        t->lt = BDeleteR(t->lt, cheie);
    else
        if (cheie > t->elem)
            t->rt = BDeleteR(t->rt, cheie);
        else
            /* am gasit elementul de eliminat */
}
```

Eliminare element cu pivot – altă variantă

```
else          /* am gasit elementul de eliminat */
    if (t->lt && t->rt)          /* 2 copii */
    { temp = findMin(t->rt);    /* gaseste min */
      t->elem = temp->elem;
      /* si inlocuieste cheia curenta cu cheia minima */
      t->rt = BTDeleteR(t->rt, t->elem);
      /* sterge cheia minima */
    }
    else /* un singur copil */
    {
        temp=t;
        if(t->lt == NULL) child = t->rt;
        if(t->rt == NULL) child = t->lt;
        free(temp);
        return child;
    }
return t;
}
```


4.8 Ce căutăm?

- De obicei prelucrăm structuri mai complicate decât întregi sau caractere

```
typedef struct client {  
    int    id_client;  
    char  *nume;  
    char  *adresa;  
    char  *nr_tel; } Item;
```

```
typedef struct node *TLink;  
typedef struct node {  
    Item elem;  
    TLink lt, rt;} TreeNode;
```


Ce căutăm?

```
typedef struct client {  
    int id_client;   
    char *nume;  
    char *adresa;  
    char *nr_tel; } Item;  
#define Key(pers) pers.id_client
```

Sau


```
int Key(Item pers)  
{return pers.id_client;}
```


Cheia de căutare

```
typedef struct client {  
    int id_client;   
    char *nume;  
    char *adresa;  
    char *nr_tel; } Item;  
#define Key(pers) pers.id_client
```

```
TLink BTSearch(TLink t, int cheie)  
{  
    if(t == NULL) return NULL;  
    if (Key(t->elem) == cheie) return t;  
    if(cheie < Key(t->elem))  
        return BTSearch(t->lt,cheie);  
    return BTSearch(t->rt,cheie);  
}
```

Cheia de căutare

```
typedef struct client {
    char *nume; 
    char *adresa;
    char *nr_tel; } Item;

#define Key(pers) pers.nume

/* definim functiile equal, less - cu strcmp de exemplu */

TLink BTSearch(TLink t, int cheie)
{
    if(t == NULL) return NULL;
    if (equal(cheie,Key(t->elem)) return t;
    if(less(cheie,Key(t->elem)))
        return BTSearch(t->lt,cheie);
    return BTSearch(t->rt,cheie);
}
```

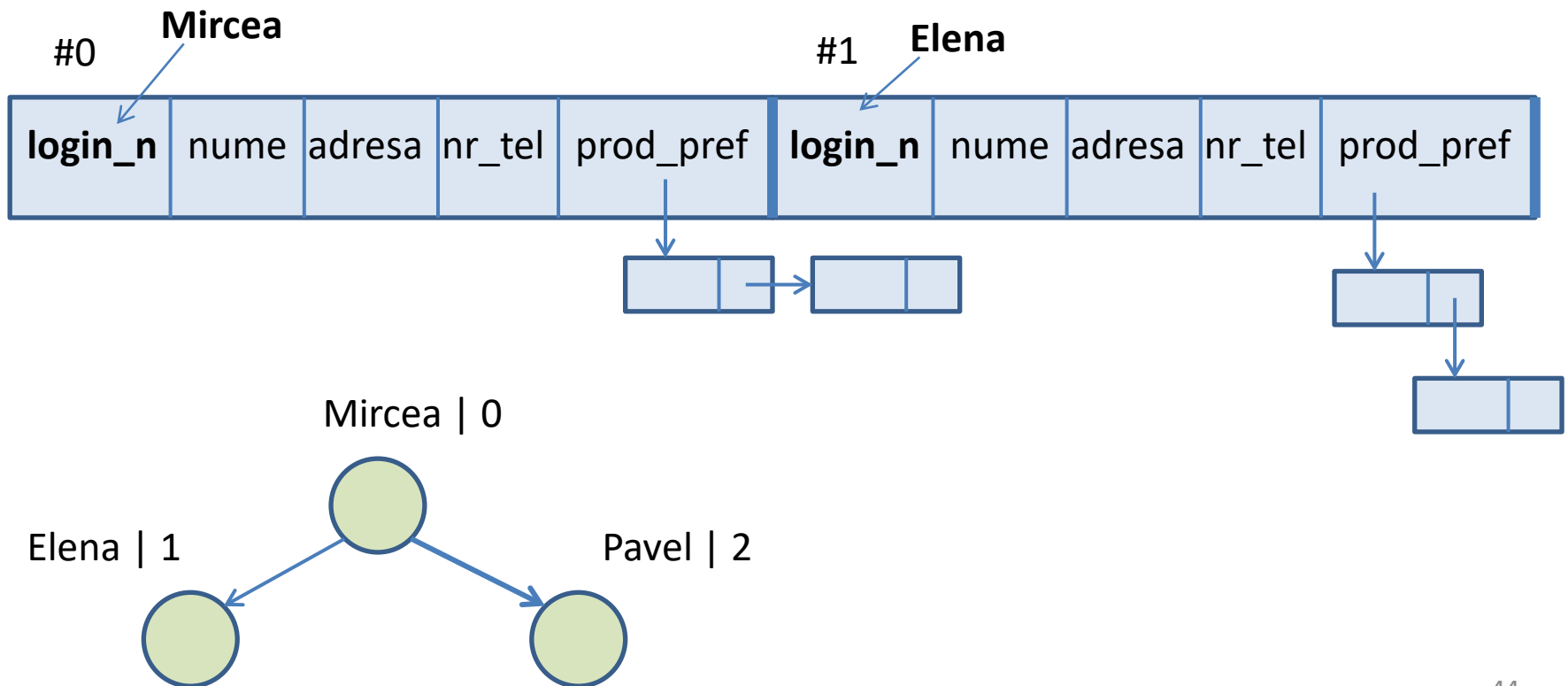
4.9 Căutare bazată pe index

- Pentru multe aplicații dorim o structură de date care să ne ajute la căutare dar care să nu ne oblige să mutăm efectiv elementele
- Utilă și în cazul elementelor de dimensiune mare care ar trebui alocate

```
typedef struct client {  
    char login_n[10];  
    char *nume;  
    char *adresa;  
    char *nr_tel;  
    TList produse_pref} Item;
```

Căutare bazată pe index

- Elementele care conțin informație sunt organizate într-o structură separată, chiar statică, și folosim indexul acestora ca elemente din nodurile arborelui





Căutare bazată pe index

- Caut cuvinte într-un text organizând textul într-un arbore binar pe baza index-ului literei de început a unui cuvânt

0 **j**os în vraful de foi ude ..
4 în vraful de foi ude, prin lăstari ...
7 **v**raful de foi ude, prin lăstari și vrejuri ...
14 **d**e foi ude, prin lăstari și vrejuri ...
17 **f**oi ude, prin lăstari și vrejuri crude ...
21 **u**de, prin lăstari și vrejuri crude s-ar putea ...
25 **p**rin lăstari și vrejuri crude s-ar putea să dau ...
30 **l**ăstari și vrejuri crude s-ar putea să dau de el ...
38 **ș**i vrejuri crude s-ar putea să dau de el: