Structuri de date: ARBORI

Organizarea liniară de tip listă este adecvată pentru aplicațiile în care datele (elementele din listă) formează o mulțime omogenă si deci se află pe acelasi nivel. În multe aplicații, este strict necesară *organizarea ierarhic*ă pentru studiul si rezolvarea problemelor.

Exemple:

- planificarea meciurilor într-un turneu sportiv de tip cupă (fig.1);
- organizarea ierarhică a fisierelor în memoria calculatorului;
- structura de conducere a unei întreprinderi, minister etc.;
- organizarea administrativă a unei țări.

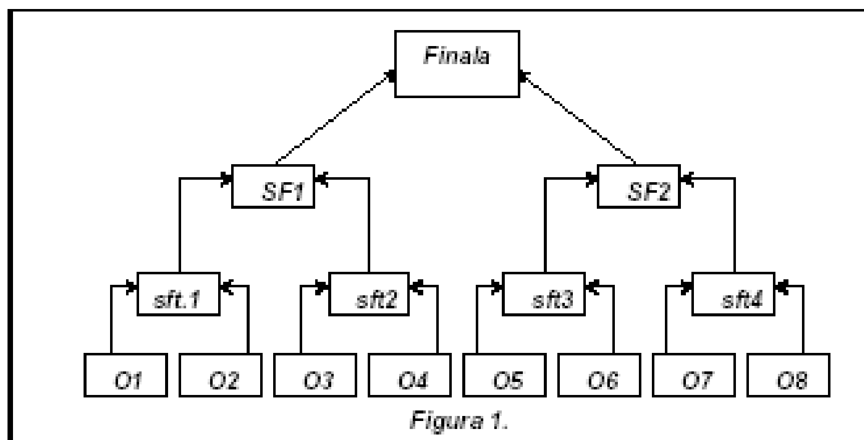


Figura 1.

În unele aplicații, implementarea structurilor ierarhice în programele de calculator este singura cale de rezolvare.

Un ***arbore*** este o structură ramificată formată dintr-un nod A (rădăcina) si un număr finit de arbori (subarbori) ai lui A.

- orice nod din arbore este rădăcină pentru un subarbore iar orice arbore poate deveni subarbore;
- doi subarbori pot fi în relație *de incluziune*, când un subarbore este inclus în celălalt sau *de excluziune*, când nu au noduri comune.

Definiții:

nod = punct în care se întâlnesc doi subarbori;
nivel = numărul de noduri parcurse de la rădăcină până la un nod;
*rădăcin*ă = nivel 1;
descendent = primul nod al unui subarbore;
nod terminal = nod fără descendenți;
înălțimea unui nod = numărul maxim de niveluri;
arbore binar = arbore în care toate nodurile au 2 descendenți;

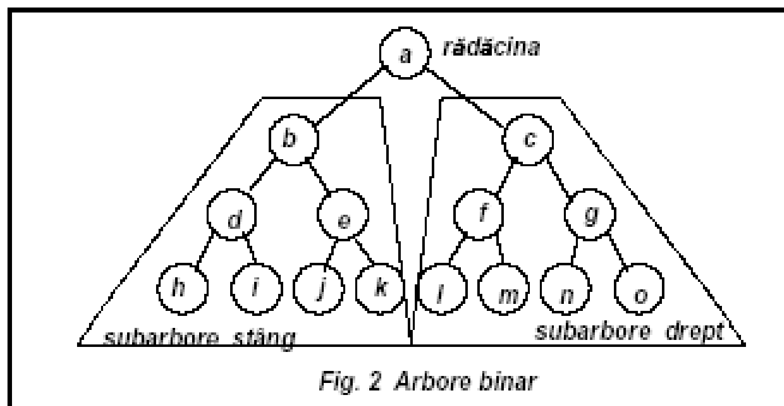


Fig. 2 Arbore binar

Orice arbore binar are doi subarbori: stâng si drept. Arborii binari pot fi arbori de căutare si arbori de selecție; ei se caracterizează prin faptul că fiecare nod areo*"cheie"* reprezentată printr-o informație specifică de identificare a nodului. Cheia permite alegerea unuia din cei doi subarbori în funcție de o decizie tip *"mai mic"*, *"mai mare", etc.*

Un arbore este *echilibrat* dacă pentru orice subarbore diferența dintre înalțimile subarborilor săi este cel mult 1.

1. Parcurgerea arborilor

Prin parcurgerea arborelui înțelegem inspectarea (vizitarea) fiecărui nod si prelucrarea informației specifice lui. Pentru un arbore dat, corespunzător unei anumite aplicații, se impune o anumită ordine de parcurgere.

În programe se utilizează algoritmi de parcurgere sistematică a arborilor implementați sub formă de proceduri. Procedurile eficiente de parcurgere generează liste dinamice cu nodurile ce urmează a fi examinate, care sunt reactualizate după examinarea fiecărui nod; când lista devine vidă operația de parcurgere se încheie (au fost examinate toate nodurile). Posibilitățile de parcurgere sunt:

1. ***În lățime*** , examinând nodurile pe niveluri, în acelasi sens.

2. ***În adâncime***: de la rădăcină spre nodurile terminale sau invers;

În cazul parcurgerii *în l*ăț*ime* algoritmul conține operațiile:

- se examinează nodul rădăcină si se formează lista (coada) descen- denților săi într-o anumită ordine (exemplu: de la stânga la dreapta) ;

- se examinează primul nod din coada de asteptare si se adaugă la coadă descendenții săi în ordinea stabilită;

- se repetă operația precedentă până când coada devine vidă.

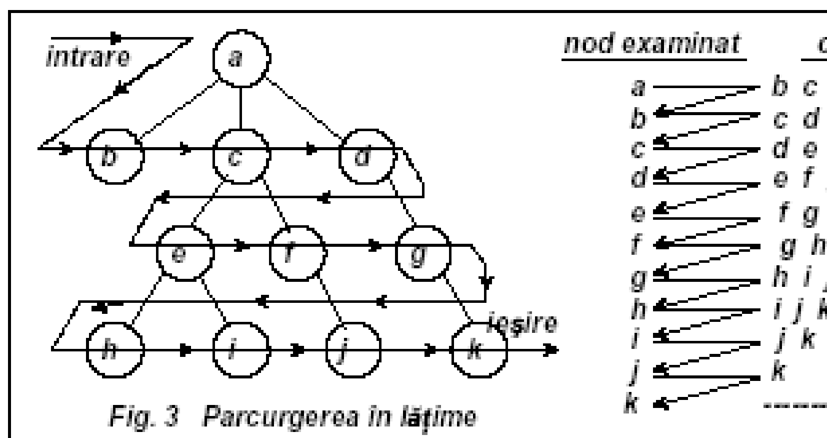


Fig. 3 Parcurgerea în lățime

În cazul parcurgerii în *adâncime*, algoritmul conține operațiile:

- se examinează nodul rădăcină si se formează lista (stiva) descen- denților săi într-o anumită ordine (exemplu: de la stânga la dreapta) ;

- se examinează primul nod din stiva de asteptare si se introduc în stivă descendenții nodului curent până la cei terminali (întregul subarbore);

- se repetă operația precedentă până când stiva devine vidă.

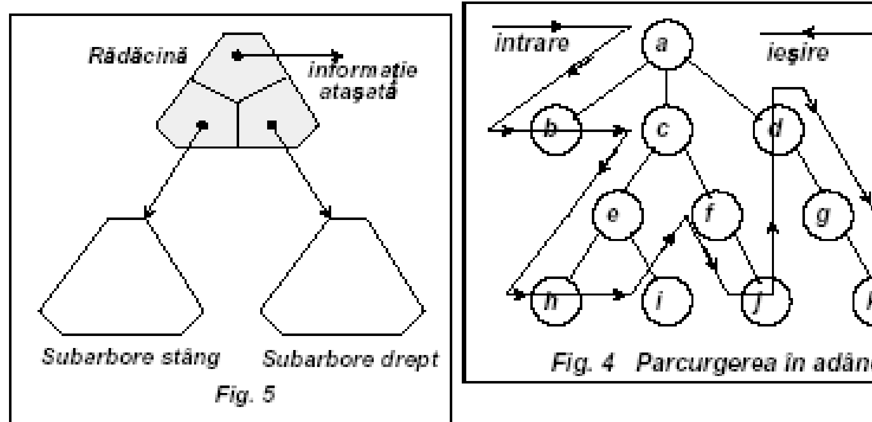


Fig. 5

Fig. 4 Parcurgerea în adânc

2. Implementarea arborilor binari

Un arbore binar are în general 3 componente: nodul rădăcină, subarbore stâng, subarbore drept.

Cazul limită este un arbore vid, reprezentat printr-o constantă simbolică "*ArboreVid*". Tipul informației atasate nodurilor dintr-un arbore este specificat în fiecare aplicație.

De aceea vom considera că informația din fiecare nod este adresată indirect prin intermediul unui pointer (fig. 5). Cei doi subarbori sunt de asemenea adresați indirect, fie prin pointeri (fig. 5), fie prin intermediul indicilor de tablou în cazul implementării statice.

Operațiile fundamentale specifice arborilor binari sunt:

a) Operații care furnizează informații despre un arbore:

- testarea existenței unui arbore (dacă nu există, este vid);
- furnizarea adresei către un subarbore;
- evaluarea numărului de noduri;

b) Operații care modifică structura unui arbore:
- inserarea unui nod;
- stergerea unui nod;
- modificarea informaț iei dintr-un nod;
- stergerea arborelui (eliberarea spațiului de memorie).

Observații:

1. Operația de căutare nu este inclusă în operațiile fundamentale pentru că aceasta poate avea diferite variante, în funcție de tipul arborelui prelucrat.

2. Arborii sunt utilizați (în mod frecvent) pentru a memora informații care se modifică. De aceea, pentru reprezentarea lor se preferă soluția dinamică, bazată pe utilizarea pointerilor. În implementarea arborilor se utilizează o metodă asemănătoare cu cea de la liste, prin construirea unui fisier cu operațiile fundamentale si includerea acestuia în programele de aplicații.

3. În afara operațiilor prezentate anterior, pot fi definite şi altele, necesare în diferite aplicații.

Deoarece numărul de legături ale unui nod este cunoscut (cel mult 2), se utilizează doi pointeri pentru nodurile din stânga si din dreapta, adică rădăcinile celor doi subarbori. Un subarbore vid, va fi reprezentat prin pointerul NULL.

Presupunând că s-a definit un tip de date DATA pentru câmpul de informaț ie, următoarea structură C defineste tipurile de date NODE si BTREE (referință la NODE):

```
struct node {
DATA d;
struct node *left;
struct node *right;
typedef struct node NODE, *BTREE;
```

O primă funcție este cea care construieste un nod.

```
BTREE new_node(DATA d)
{
BTREE t=(BTREE) malloc(sizeof(NOD));
if(t==NULL) err_exit("new_node: Eroare de
alocare");
t->d=d;
t->left=t->right=NULL;
return t;
}
```

Este utilă în aplicații o funcție constructor de nod, care primeste atât câmpul de date cât si cele două câmpuri de legătură:

```
BTREE init_node(DATA d, BTREE left, BTREE right)
{
BTREE t=new_node(d);
t->left=left;
t->right=right;
return t;
}
```

Parcurgerea arborilor binari se face în trei moduri :

- În ***pre**ordine* (**R**SD - rădăcină, stânga, dreapta), adică se vizitează rădăcina, subarborele stâng si apoi subarborele drept; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelasi algoritm - RSD.
- În ***in**ordine* (S**R**D - stânga, rădăcină, dreapta), adică se parcurge subarborele stâng, se vizitează rădăcina si apoi subarborele drept; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelasi algoritm - SRD.
- În ***post**ordine* (SD**R** - stânga, dreapta, rădăcină), adică se parcurge subarborele drept, cel stâng si apoi se vizitează rădăcina; cei doi subarbori vor fi tratați ca arbori în procesul de parcurgere, aplicând acelasi algoritm - SDR.

Pentru arborele binar din fig. 2, aplicând metodele de parcurgere de mai sus, rezultă următoarele siruri de noduri:

- preordine: **`a b d h i e j k c f l m g n o`** ;
- inordine: **`h d i b j e k a l f m c n g o`** ;
- postordine: **`h i d j e k b l m f n o g c a`** ;

Definițiile celor trei metode de parcurgere sunt recursive, adică se aplică la fel pentru orice subarbore; acest fapt sugerează o implementare recursivă a funcțiilor de parcurgere. Considerăm o funcție, visit() care examinează informația dintr-un nod, cu prototipul:

```
void visit(BTREE);
```

Cele trei metode de parcurgere, se implementează în mod natural, astfel:

```
void par_rsd(BTREE t)
{
if(t!=NULL) {
visit(t);
par_rsd(t->left);
par_rsd(t->right);
}
}
```

```
void par_srd(BTREE t)
{
if(t!=NULL) {
par_srd(t->left);
visit(t);
par_srd(t->right);
}
}
```

```
void par_sdr(BTREE t)
{
if(t!=NULL) {
par_sdr(t->left);
par_sdr(t->right);
visit(t);
}
}
```

Pentru afisarea nodurilor parcurse (prin nume nod), sub formă indentată, se poate folosi o funcție **print_tree()**, recursivă, care scrie un număr variabil de spații, înainte de afisarea unui nod, în funcție de nivelul nodului respectiv.

```
void print_t(BTREE t, int n)// functie ajutatoare
{
int i;
for(i=0; i<n; i++)
printf(" ");
if(t!=NULL) {
print_node(t->d); // functie de afisare nod
print_t(t->left, n+1);
print_t(t->right, n+1);
}
else
printf("Arbore vid!");
}
void print_tree(BTREE t)
{
print_t(t, 0);
putchar('\n');
}
```

3. Arbori de căutare

Arborii de căutare sunt arbori binari în care există o relație de ordine pe mulțimea elementelor de tip DATA, acestea fiind câmpurile de date din noduri. Categoria arborilor de căutare este caracterizată de următoarele proprietăți esențiale:

- câmpurile de date din noduri conțin valori distincte;
- un arbore vid este, prin convenție, arbore de căutare;
- rădăcina unui arbore de căutare conține o valoare mai mare decât toate valorile din subarborele stâng si mai mică decât toate valorile din subarborele drept;
- subarborii stâng si drept sunt arbori de căutare.

O importantă proprietate a arborilor de căutare, care decurge din definiție, este aceea că parcurgerea în inordine produce o listă de elemente sortate crescător, în raport cu câmpul DATA.

Această proprietate este utilizată de algoritmii de sortare internă.

A doua proprietate importantă a arborilor de căutare, care dă chiar denumirea lor, este posibilitatea implementării unui algoritm eficient de căutare a unui element, pe baza valorii din câmpul DATA: se compară elementul căutat cu data din rădăcină; apar trei posibilități:

- acestea coincid – elementul a fost găsit;
- elementul căutat este mai mic decât data din rădăcină – atunci căutarea continuă în subarborele stâng;
- elementul căutat este mai mare decât data din rădăcină - atunci căutarea continuă în subarborele drept;

Presupunem că s-a definit o funcție de comparație (care este dependentă de tipul DATA, deci de aplicație):

```
int cmp_data(DATA a, DATA b);
```

care returnează o valoare negativă dacă **a<b**, zero dacă **a=b** sau o valoare pozitivă, dacă **a>b**.

Funcția de căutare, în variantă recursivă, se poate scrie astfel:

```
BTREE cauta (BTREE t, DATA x)
{
int y;
if(t==NULL || (y=cmp(x, t-.d))==0)
return t;
t=(y<0) ? t->left: t->right;
return cauta(t,x);
}
```

Funcția **cauta()** primeste un pointer **t** la rădăcina arborelui şi o valoare **x** şi returnează pointerul la nodul găsit (care conține **x**) sau **NULL** dacă valoarea **x** nu există în arborele **t**.

Dacă arborele de căutare este echilibrat (adică fiecare subarbore are aproximativ acelasi număr de noduri în stânga si dreapta), numărul de comparații necesare pentru a găsi o anumită valoare este de ordinul lui , unde n este numărul de noduri. $\log_2(n)$.

Dacă arborele de căutare este neechilibrat, atunci căutarea este asemănătoare celei dintr-o listă simplu înlănțuită, caz în care numărul de comparații necesare, poate ajunge la n.

Petru a elimina restricția ca nodurile să aibă elemente distincte, se modifică tipul **DATA**, introducând, pe lângă valoarea propriu-zisă, un întreg care reprezintă numărul de repetări ale valorii respective.

Definim tipul informației din nod ca fiind **KEY** iar tipul **DATA** devine:

```
typedef struct {KEY key; int count;} DATA;
```

Definiția tipurilor **NODE** si **BTREE** nu se schimbă dar funcția de construcție a unui nod se modifică, primind acum o valoare de tip **KEY** si inițializând cu 1 câmpul count al nodului creat.

Inserarea unui element într-un arbore de căutare

Din definiția arborelui de căutare, rezultă că adăugarea unui nod se face în funcție de valoarea câmpului de informație; pentru adăugarea unui nod, este necesar asadar să se determine locul în care se inserează. Se fac deci două operații: căutarea locului şi inserarea propriu-zisă.

Algoritmul de bază utilizat în construcția arborilor de căutare este numit *căutare si inserare*.

Se dă arborele t si un element x. Dacă x este în t, se incrementează câmpul count corespunzător valorii x; dacă nu, x este inserat în t într-un nod terminal, astfel încât arborele să rămână *de căutare*.

Considerăm că s-a definit o funcție de comparare, **cmp_key()** care returnează o valoare negativă, zero sau o valoare pozitivă, după cum relația dintre argumente este **a<b, a=b, a>b**.

```
int cmp_key(KEY a, KEY b);
```

O implementare recursivă a algoritmului de căuare si inserare este:

```
BTREE cauta_si_insr(BTREE t, KEY x)
{
int c;
```

```
if(t==NULL) return new_node(x);
if((c=cmp_key(x, (t->d).key))<0)
t->left=cauta_si_insr(t->left, x);
else
if(c>0)
t->right=cauta_si_insr(t->right, x);
else (t->d).count++;
return t;
}
```

Funcția primeşte pointerul t la rădăcina arborelui si cheia x. Dacă s-a ajuns la un nod terminal sau arborele este vid, rezultă că x nu este în arbore si ca urmare se construieste un nod nou cu valoarea x.

Dacă **(t!=NULL)**, se aplică algoritmul de căutare, apelând recursiv funcția cu subarborele stâng, respectiv drept. Dacă se identifică valoarea x în arbore, atunci se incrementează câmpul count al nodului în care s-a găsit x. În final se returnează pointerul la arborele modificat.

3.1. Sortarea unui sir, bazată pe arbore de căutare

Se construieste un arbore de căutare cu elementele sirului de date, considerate ca elemente ale tabloului tab[]; apoi se parcurge în inordine arborele creat returnând elementele sale, în ordine, în acelasi tablou.

```
void sort(KEY tab[], int n)
{
BTREE t=NULL;
int i
for(i=0; i<n; i++)
t=cauta_si_insr(t, tab[i]);
index=0;
parcurge(t, tab);
delete(t);
}
```

Funcția parcurge() preia elementele din arborele t si le plasează în tabloul v[]:

```
static int index;
void parcurge(BTREE t, KEY v[])
{
int j;
if(t!=NULL) {
parcurge(t->left, v);
for(j=0; j<(t->d).count; j++)
v[index++]=(t->d).key;
parcurge(t->right, v);
}
}
```

Se parcurge t în inordine, copiind în v cheia din rădăcină, de atâtea ori cât este valoarea lui count. Este esențială declararea indicelui de tablou, index, în clasa **static external** şi inițializarea lui cu 0, înainte de a apela funcția **parcurge()** în cadrul

funcției **sort()**; se asigură astfel incrementarea sa corectă pe parcursul apelurilor recursive ale funcției **parcurge()**.

Funcția **delete()** (utilizată în **sort()**), eliberează spațiul de memorie alocat arborelui t, după copierea datelor sortate, fiind si un exemplu de parcurgere în postordine:

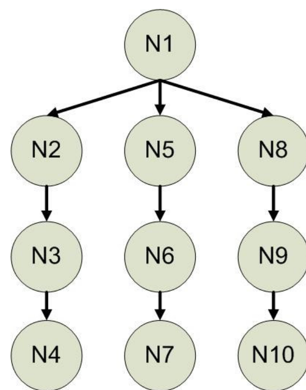
```
void delete(BTREE t)
{
if(t!=NULL) {
delete(t->left);
delete(t->right);
free(t);
}
}
```

Apelul funcției free() trebuie să fie după apelurile recursive, deoarece, după execuția lui free(t), variabila t nu mai există.

Acest algoritm de sortare, combinat cu implementarea arborelui prin alocare secvențială si anume chiar în tabloul care trebuie sortat, duce la o metodă eficientă de sortare internă.

TEMA:

Se citesc 10 numere de la tastatură N1, N2, ... N10, cu care se formează un arbore cu următoarea formă:



Să se mute apoi numerele într-un arbore binar, utilizând orice metodă de parcurgere se doreşte pentru arborele inițial. Apoi să se parcurgă arborele binar creat pentru a afişa numerele în ordine inversă.