Tema „**Analiza prelucrării structurilor de date arborescente** ”

Sarcina și obiectivele:

- de studiat și însușit materialul teoretic pentru evidențierea esențialului prelucrării structurilor de date cu **arbori** în elaborarea modelelor soluției, analizând exemplele din text;
- de analizat principiile și modurile de prelucrare, principalele operații cu structuri de date arborescente, dotând cu comentariile de vigoare
- exemplele din 1.3. (Aplicații de *arbori binari* în C), 2. și exemplele 2.7 de derulat, obținând rezultatele de testare, verificare și expuse în raport.
- să se rezolve problemele din compartimentul “5. PROBLEME PROPUSE spre rezolvare” pentru care să se elaboreze scenariile succinte de soluționare, algoritmii funcțiilor principale, organigramele SD și programele cu **arbori și fișiere** (declarări și procesări).
- analizați, comentați, afișați organigramele și toate componentele.
- să se analizeze și să se evidențieze tehnica modelării și programării eficiente pentru diverse compartimente ale diferitor situații cu diverse argumentări și modele de structuri abstracte, incluzând fișiere cu teste de verificare, explicații de rigoare și vizualizări.
- În raport să fie expuse toate activitățile și dificultățile întâlnite pe parcursul efectuării ll.

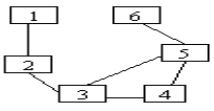
Considerații teoretice:

1. **Structura arborescentă** este o structură de date de tip arbore datele fiind înmagazinate în noduri, nodul de baza e rădăcină. Un arbore reprezintă un "graf" particular, în care substructurile legate la orice nod sunt disjuncte și în care exista un nod numit rădăcină din care e accesibil orice nod prin traversarea unui număr finit de arce. Conform teoriei grafurilor, un **arbore** este un graf neorientat, conex și fără cicluri. Arborii reprezintă grafurile cele mai simple ca structură din clasa grafurilor conexe, ei fiind cel mai frecvent utilizați în practică.

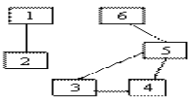
Pentru un graf arbitrar G cu n vârfuri si m muchii sunt echivalente următoarele afirmații:

1) G este arbore;
2) G este un graf conex si m = n - 1;
3) G este un graf aciclic si m = n - 1;
4) oricare doua vârfuri distincte (diferite) ale lui G sunt unite printr-un lanț simplu care este unic;
5) G este un graf aciclic cu proprietatea ca, daca o pereche oarecare de vârfuri neadiacente vor fi unite cu o muchie, atunci graful obținut va conține exact un ciclu.

Notă: Graf conex = graf neorientat **G=(V,E)** in care pentru orice pereche de noduri **(v,w)** exista un lanț care le unește.

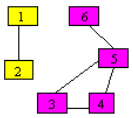


graf conex



nu este graf conex

Componenta conexa = subgraf al grafului de referință, maximal in raport cu proprietatea de conexitate (intre oricare doua vârfuri exista lanț);



graful nu este conex. Are 2 componente conexe: 1, 2 si 3, 4, 5, 6

Se numește **Arbore cu rădăcină =** graf neorientat conex fără cicluri in care unul din noduri este desemnat ca rădăcină. Nodurile pot fi așezate pe niveluri începând cu rădăcină care este plasata pe nivelul **1**.

Rădăcina = Nod special care generează așezarea unui arbore pe niveluri; Aceasta operație se efectuează in funcție de lungimea lanțurilor prin care celelalte noduri sunt legate de rădăcină.

Descendent = într-un arbore cu rădăcină nodul **y** este descendentul nodului **x** daca este situat pe un nivel mai mare decât nivelul lui **x** si exista un lanț care le unește si nu trece prin rădăcină.

Descendent direct / fiu = într-un arbore cu rădăcină nodul **y** este fiul (descendentul direct) nodului **x** daca este situat pe nivelul imediat următor nivelului lui **x** si exista muchie intre **x** si **y**.

Ascendent = într-un arbore cu rădăcină nodul **x** este ascendentul nodului **y** daca este situat pe un nivel mai mic decât nivelul lui **y** si exista un lanț care le unește si nu trece prin rădăcină.

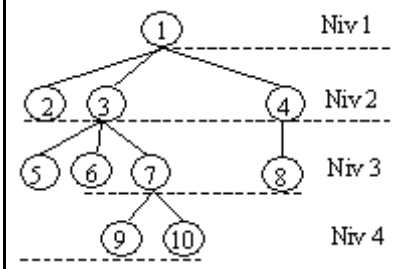
Ascendent direct / părinte = într-un arbore cu rădăcină nodul x este părintele (ascendentul direct) nodului y daca este situat pe nivelul imediat superior (cu număr de ordine mai mic) nivelului lui y si exista muchie intre x si y.

Frați = într-un arbore cu rădăcină nodul **x** este fratele nodului **y** daca au același părinte.

Frunza = într-un arbore cu rădăcină nodul x este frunza daca nu are nici un descendent direct

Cum un arbore este un caz particular de graf neorientat înseamnă ca poate fi reprezentat ca un graf. De aici rezulta ca pentru reprezentarea unui arbore se pot utiliza: Metode specifice grafurilor; Metode specifice arborilor

Într-o structură de tip arbore, elementele sunt structurate pe nivele ; pe primul nivel, numit nivel 0, există un singur element numit ***rădăcină***, de care sunt legate mai multe elemente numite ***fii*** care formează nivelul 1; de acestea sunt legate elementele de pe nivelul 2 ș. a. m. d. (vezi figura).



Arbore cu rădăcină

- Nodul **1** este rădăcină.
- Nodurile **5, 6, 7** sunt fii nodului **3**.
- Nodul **7** este părintele nodurilor 9 si **10**;
- Nodul **9** este descendentul lui **3**
- Nodul **3** este ascendentul lui **10**
- Nodurile **8, 9** si **10** sunt frunze
- Nodurile **5, 6** si **7** sunt frați.

Deci un ***arbore*** este compus din elementele numite ***noduri*** sau ***vârfuri*** și legăturile dintre acestea. Un nod situat pe un anumit nivel este ***nod tată*** pentru nodurile legate de el, situate pe nivelul următor, acestea reprezentând ***fiii*** săi. Fiecare nod are un singur tată, cu excepția rădăcinii care nu are tată. Nodurile fără fii se numesc ***noduri terminale*** sau ***frunze***. Termenii ' nod tată', 'fiu' sau 'frate' sunt preluați de la arborii genealogici, cu care arborii se aseamănă foarte mult.

Nodul 1 fiind rădăcină, atunci vectorul de tip tata este:

0	1	1	1	3	3	3	4	7	7

Legătura de tip TATA se mai numește si legătura cu referințe ascendente.

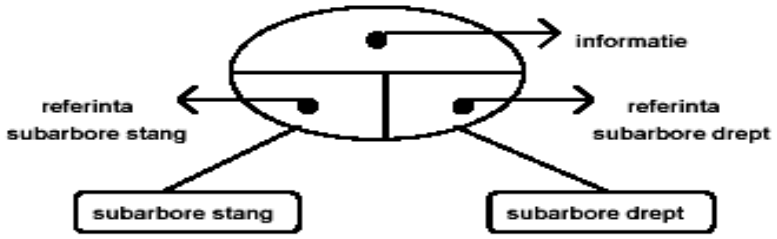
Legătura de tip TATA este determinata si de modul in care am ales nodul rădăcină. Spre exemplu daca vom considera nodul 4 ca fiind rădăcină vom obține o alta soluție.

Arborii, ca structuri dinamice de date, au extrem de multe aplicații în informatică. Deosebit de utilizată în aplicații este structura de tip ***arbore binar***. Un ***arbore binar*** este un arbore în care fiecare nod are cel mult doi fii, ***fiul stâng*** și ***fiul drept*** (fiul stâng este legat în stânga tatălui și cel drept în dreapta).

Dacă în figură se elimină rădăcina și legăturile ei, se obțin doi arbori binari care se numesc ***subarborii stâng*** și ***drept*** ai arborelui inițial. Arborele binar este, deci, o structură recursivă de date. Un arbore binar nevid fie se reduce la rădăcină, fie cuprinde rădăcina și, cel mult, doi subarbori binari. Un arbore binar se poate implementa foarte ușor cu ajutorul adreselor de înlănțuire, fiecare element cuprinzând, în afară de informația propriu-zisă asociată nodului, adresa fiului stâng și adresa fiului drept, acestea exprimând legăturile existente între noduri.

Daca nodul A conține o referință la un nod B, atunci A e ascendentul lui B sau B e descendentul lui A. Considerând nodul rădăcină situat la nivelul 0 descendenții nodului rădăcină vor fi situați pe nivelul 1, iar descendenții acestora pe nivelul 2 ș.a.m.d. Numărul de arce traversat de la nodul rădăcină până la un nod K definește lungimea căii acelui nod. Ca structură de date recursivă un arbore se definește recursiv astfel: el poate fi vid sau poate consta dintr-un nod conținând referințe la arborele disadjuncti.

Implementarea arborilor presupune crearea în fiecare nod a unei structuri de date în care să existe câte un pointer/referință către fiecare dintre subarbori. Structura de date în nodul unui arbore:



Există multe tipuri de arbori. Un arbore vid are referințele rădăcinii *nil* (în Pascal) sau *NULL* (în C/C++).

Un arbore binar e un arbore orientat, în care fiecare vârf are maximum 2 descendenți. Un arbore binar în care orice vârf are 1 sau 2 descendenți se numește arbore binar complet.

Arbore binar echilibrat e acel pentru care fiecare nod a căruia numărul de noduri din subarborele stâng diferă de cel din subarborele drept cu cel mult 1. Ca exemplu poate servi arborele genealogic.

Un arbore binar poate fi definit recursiv ca fiind o mulțime (colecție) de noduri vidă sau formată din nodul *Rădăcină, Subarbore_stâng și Subarbore_drept*. La reprezentarea în memorie a unui arbore binar, pe lângă informația propriu-zisă se vor memora în fiecare nod și adresele de legătură spre nodurile succesoare în felul următor:

Un mod de reprezentare a arborilor e acel a expresiilor cu paranteze incluse, ce se construiește astfel: expresia începe cu rădăcină arborelui, iar fiecare vârf, ce are descendenți e urmat de expresiile atașate subarborilor, ce au ca rădăcini descendenții vârfurilor, despărțite între ele prin virgula și cuprinse în paranteze.

A(B(C,D(H)),K(L(N),M(E)))

Exista 3 metode principale de parcurgere a unui arbore și anume:

1) preordine ;
2) inordine ;
3) postordine.

Traversarea în preordine e acea, în care mai întâi se prelucrează informația din nod și apoi se parcurge prin arbore.

Traversarea în inordine e acea, în care secvență acțiunilor e următoarea: parcurgerea subarborelui stâng, vizitarea nodului rădăcină și apoi parcurgerea subarborelui drept.

Traversarea în postordine cu parcurgerea subarborelui stâng, subarborelui drept și apoi vizitând rădăcină.

Structura arborescenta **tree** - structura arborescenta de date utilizata frecvent în sisteme, ce implica memorarea unei mari cantități de date sortate pe baza unor chei compuse în vederea unor operații de căutare.

1.1. Analiza definițiilor matematice și proprietăților arborelui

Definiție. Se numește *arbore* cuplul format din **V** si **E : T= (V,E)** cu **V** — o mulțime de noduri si $\mathbf{E \subseteq VxV}$ — o mulțime de arce, cu proprietățile:

1) $\exists$ nodul $\mathbf{r \in V}$ (nodul rădăcină) astfel încât $\forall \mathbf{j \in V, (j, r) \notin E}$

2) $\forall \mathbf{x \in V\{r\}}$, $\exists \mathbf{y \in V}$ unic , astfel încât **(y, x) ∈ E** (Cu alte cuvinte, pentru toate nodurile rădăcină, ∃ un singur arc ce intra în nodul respectiv)

3) $\forall \mathbf{y \in V}$, $\exists$ un drum $\mathbf{\{ r = x_0, x_1, x_2, ... ,x_n = y\}}$, **cu $x_i \in V$ si $(x_i, x_{i+1}) \in E$** (Sau arborele trebuie sa fie un graf conex: nu exista noduri izolate sau grupuri de noduri izolate).

Proprietate a arborelui. Daca **T, T= (V, E)** este un arbore si $\mathbf{r \in V}$ este rădăcină arborelui, atunci mulțimea **T\{r} = (V', E')**, cu **V' = V -{r}** si **E' = E -{ (r, x)/(r, x) ∈ E }** poate fi partiționată astfel încât sa avem mai mulți arbori, a căror reuniune sa fie **T\{r}**, si oricare ar fi doi arbori intersectați, sa dea mulțimea vida:

$$\mathbf{T\{r\} = T1 \cup T2 \cup ... \cup Tk}\ ,\ \mathbf{Ti \cap Tj = \varnothing}\ .$$

Definiții:

1) Daca avem (x, y) ∈ E , ⇒ x — *predecesorul* lui y (*tata*), y — *succesorul* lui x (*fiu*)

2) Fie **E(x) = { y, (x, y) ∈ E }** mulțimea succesorilor lui **x**.

Definim ***gradul lui x***: **degree(x) = │ E(x) │** = numărul de succesori

Definim ***gradul arborelui*** : **degree(T) = max{ degree(x)}, x ∈ V** unde **y** se mai numește ***nod terminal***, sau ***frunza,*** daca **degree(y) = 0**, adică daca nu are descendenți.

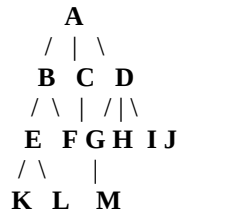
Strămoșii unui nod sunt așa-numiții **ancestors(x)** : **ancestors(x) = {r = x_0, x_1, x_2, ..., x_k}** cu proprietatea ca

$(x_i, x_{i+1}) \in E$, i= {0,k - 1} si **$(x_k, x) \in E$**

Nivelul unui nod : **level(x) = │ ancestors(x) │ + 1**

Adâncimea arborelui : **depth(T) = max { level (x)** pentru **x ∈ V**

Exemplu:



predecesor(E) = B
succesor(C) = G
E(D) = {H, I, J}
degree(D) = 3
degree(B) = 2
degree(F) = 0

degree(T) = 3
ancestors(L) = *{A, B, E}*
level(L) = 4 , *level(B)* = 2 , *level(A)* = 1
depth(T) = 4

1.2. Reprezentarea arborilor prin organigrame

1.2. 1.*Reprezentarea prin liste generalizate*

Se considera ca nodurile terminale sunt elemente atomice, iar nodurile de grad ≥ 1 sunt subliste. Deci, fie arborele de mai sus scris sub forma : A(B (E (K, L), F), C (G), D (H (M), I, J)) cu reprezentarea:

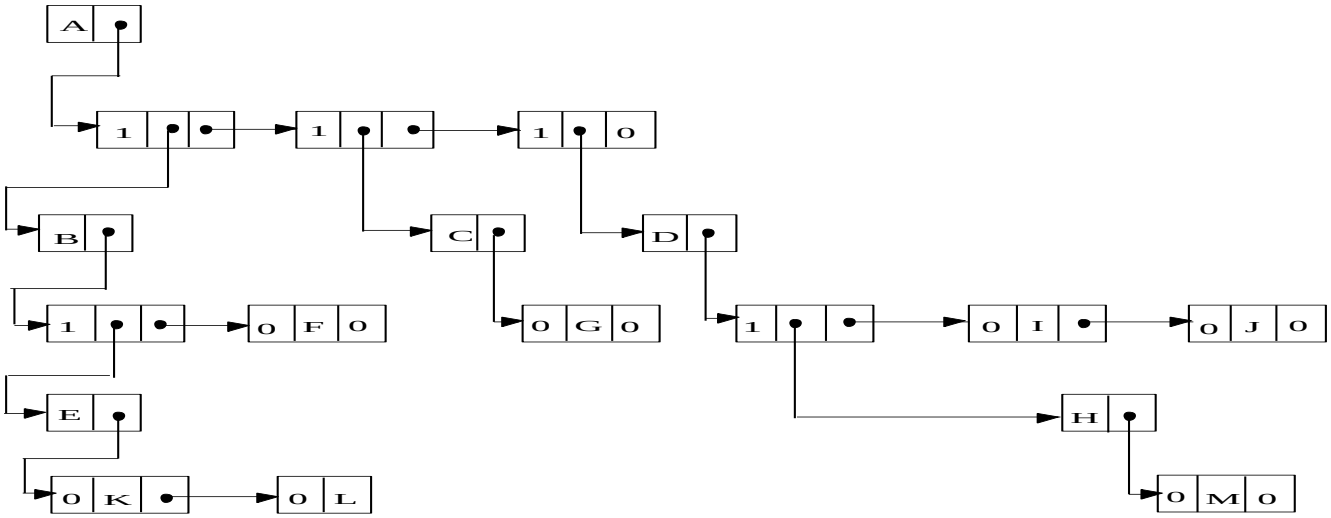
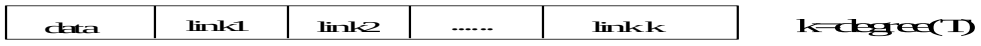


Fig.1 Organigrama arborelui

*1.2.2.*Reprezentarea prin structuri înlănțuite



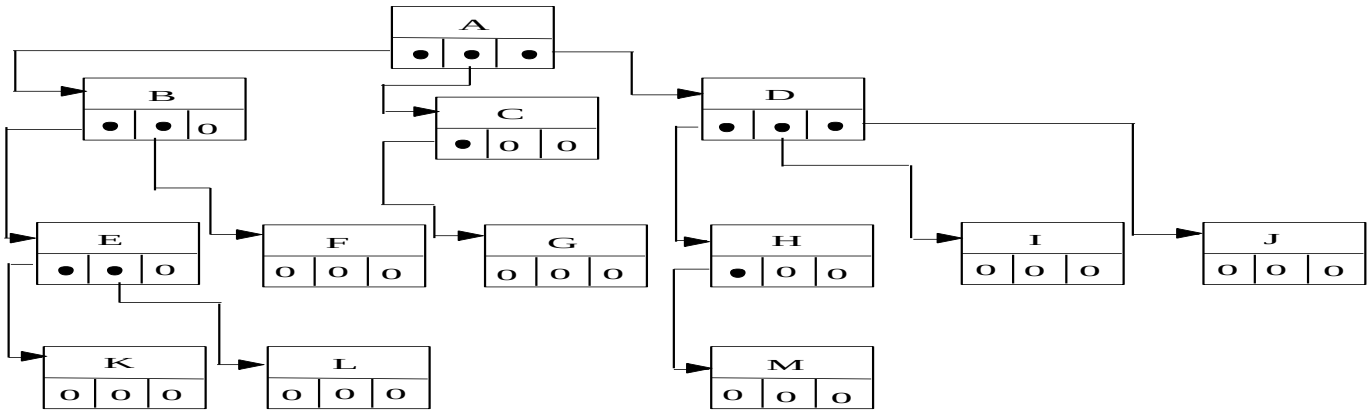


Fig.2 Organigrama arborelui *înlănțuit*

Aceasta reprezentare are calitatea ca, atunci când contează ordinea descendenților, ea poate surprinde structura diferita. De exemplu:

structura **x** este diferita de structura **x**

/ | \ / | \

vid y vid **y vid vid**

Metodele de reprezentare expuse permit identificarea legăturilor nod-descendent (succesor). Exista aplicații în care este nevoie de legătura nod-predecesor. Aşadar, pare utila reprezentarea arborelui sub forma nodului (data, parent):

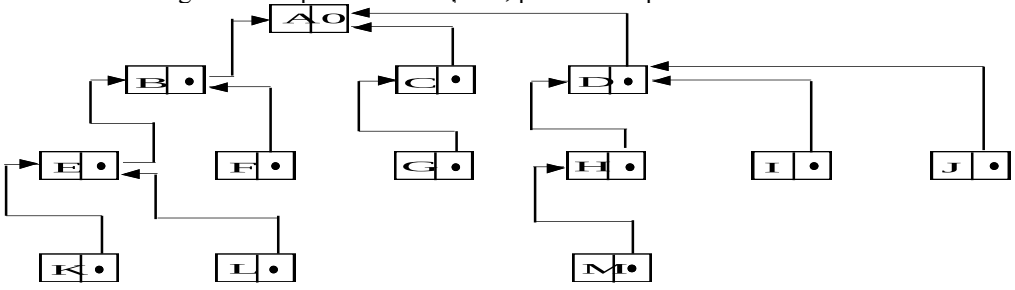
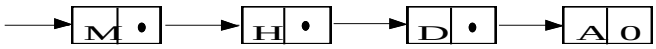


Fig.3 Organigrama arborelui pentru identificarea legăturilor nod-descendent

Având adresa unui nod, se găsesc toți predecesorii, obținându-se o *lista înlantuita:*
(*Reprezentarea TATA*):



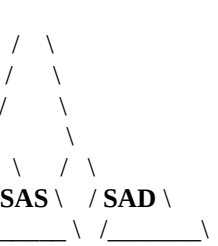
1.2.2.1 Reprezentarea arborilor binari. Cum am menționat mai sus un arbore binar este un arbore de grad maxim 2. În cazul acestor arbori, se pot defini aplicații, instrumente în plus de operare. Arborii binari pot avea deci gradele 2, 1, 0:

```
   A       A   A
  / \     /   / \
 B   C   B   B   C
/ \   \ /   /\ /\
D  E  F C   D E F G
```

Observație: Arborele **A** este diferit de **A**

```
  / \       / \
 B  vid    vid  B
```

Dacă mulțimea de elemente este nevidă, arborele binar se divide în două submulțimi: *subarborele drept* și *cel de stânga*. Arborele binar este ordonat, deoarece în fiecare nod subarborele stâng se consideră că precede subarborele drept. Un nod al unui arbore binar poate să aibă numai un descendent: subarborele drept sau subarborele stâng.



SAS — subarborele stâng (binar) SAD — subarborele drept (binar)

Definiții

1) Se numește *arbore binar strict* arborele pentru care oricare ar fi un nod $\mathbf{x \in V \Rightarrow degree(x) = 2}$, sau $\mathbf{degree(x) = 0}$. ***Exemplu:***

```
   a
  / \
 b   c
/ \ / \
d  e f  g
 / \
h  i
```

2) Se numește *arbore binar complet* un arbore binar strict pentru care $\forall$ **y** cu:

$$\mathbf{degree(y) = 0 \ (frunza) \Rightarrow level(y) = depth(T)}$$

Cu alte cuvinte, nodurile terminale aparțin ultimului nivel din arbore. ***Exemplu:***

```
     a
   /   \
  b     c
 / \   / \
d   e f   g
/\  /\ / \ /\
h i j k l m n o
```

1.2.2.2 Relații între numărul de noduri si structura unui arbore binar

Lema 1. Numărul maxim de noduri de pe nivelul **i** al unui arbore binar este egal cu $\mathbf{2^{i-1}}$.

Demonstrația se face prin inducție:

La nivelul **1** avem $\mathbf{2^0 = 1}$ **nod = rad (rădăcină A)**. Presupunem conform metodei **P(n)**: pe nivelul **n** avem $\mathbf{2^{n-1}}$ noduri. Demonstram pentru **P(n+1)**: se observa ca toate nodurile de pe nivelul **n+1** sunt noduri descendente de pe nivelul **n**. Notând **niv(i)** numărul de noduri de pe nivelul **i**,

$\mathbf{\Rightarrow niv(n+1) \le 2 \cdot niv(n) \le 2 \cdot 2^{n-1} = 2^n}$.

Lema 2. Numărul maxim de noduri ale arborelui binar de adâncime **h** este egal cu $\mathbf{2^h - 1}$.

Demonstrație:

Numărul total de noduri este egal cu: 

(progresie geometrica)

Observație: Numărul maxim de noduri pentru arborele binar se atinge în situația unui arbore binar complet.

$\mathbf{2^h - 1}$ = numărul de noduri în arborele binar complet de adâncime **h**

Lema 3. Notam cu:

n_2 — numărul de noduri de grad 2 din arborele binar;

n_1 — numărul de noduri de grad 1 din arborele binar;

n_0 — numărul de noduri terminale (frunze) din arborele binar;

În orice arbore binar, $\mathbf{n_0 = n_2 + 1}$ (nu depinde de $\mathbf{n_1}$).

Demonstrație: fie $\mathbf{n = n_0 + n_1 + n_2}$ (numărul total de noduri); conform definiției, fiecare nod are un singur predecesor $\Rightarrow$ numărul de muchii $|\mathbf{E}| = \mathbf{n - 1}$. Același număr de muchii $|\mathbf{E}| = \mathbf{2\,n_2 + n_1}$.

Deci, $\mathbf{n - 1 = 2n_2 + n_1}$, înlocuind, $\mathbf{n_0 + n_1 + n_2 - 1 = 2n_2 + n_1 \Rightarrow n_0 = n_2 + 1}$ ceea ce trebuia de demonstrat.

Rezulta ca într-o expresie numărul de operatori binari si unari este egal cu numărul de operanzi **+ 1**.

Lemele se folosesc pentru calcule de complexitate.

1.2.2.3 Operații asupra arborilor binari

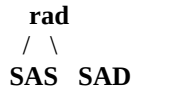
Operații curente:

- selecția câmpului de date dintr-un nod si selecția descendenților;
- inserarea unui nod;
- ștergerea unui nod.

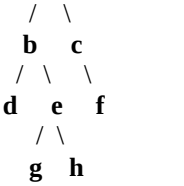
Traversarea arborilor binari (A.B.) Traversarea consta în "vizitarea" tuturor nodurilor unui arbore într-un scop anume, de exemplu, listare, testarea unei condiții pentru fiecare nod, sau alta prelucrare. O traversare realizează o ordonare a nodurilor arborelui (un nod se prelucrează o singura data).

Strategii de traversare:

- traversare în *preordine*: prelucrare în ordinea: rad, SAS, SAD;
- traversare în *inordine*: prelucrare în ordinea: SAS, rad, SAD;
- traversare în *postordine*: prelucrare în ordinea: SAS, SAD, rad.



Exemplu de traversare:



- preordine : A B D F G H C F
- inordine : D B G E H A C F
- postordine : D G H E B F C A

Funcții de parcurgere (in pseudocod)

Facem notațiile:

p — pointer la un nod

lchild(p) — pointer la succesorul stâng (p → stg)

rchild(p) — pointer la succesorul drept (p → drt)

data(p) — informația memorata în nodul respectiv (p → data)

În C/C++ avem:

```
struct Nod{
        Atom data;
        Nod* stg;
        Nod* dr;
        };
Nod* p;
```

Procedurile de parcurgere sunt:

```
preorder(t)
{
    if(t==0) return
        else  ¯ print (data(t))          // vizitarea unui nod
              | preorder (lchild(t))     // parcurgerea subarborilor
              _ preorder (rchild(t))
}

    inorder(t)
    {
        if(t ≠ 0) ¯ inorder (lchild(t))
                  | print (data(t))
                  _ inorder (rchild(t))
    }
```

```
postorder(t)
{
    if(t ≠ 0) ⌐ postorder (lchild(t))
              | postorder (rchild(t))
              |_ print(data(t))
}
```

1.2.2.4 Binarizarea arborilor oarecare

Lema 1. Daca **T** este un arbore de grad **k** cu noduri de dimensiuni egale (**k** pointeri în fiecare nod), arborele având **n** noduri reprezentarea va conține **n· (k - 1) + 1** pointeri cu valoare zero (nuli).

Demonstrație: Numărul total de pointeri utilizați în reprezentare este **n·k.** Numărul total de pointeri nenuli este egal cu numărul de arce ⇒ **n·k - (n - 1) = n (k - 1) + 1**

raportul este maxim pentru **k = 2**.

Rezulta ca cea mai eficienta reprezentare (în structura înlantuita) este reprezentarea în arbori binari.

Arborele binar de căutare (BST). Arborele binar de căutare reprezintă o soluție eficienta de implementare a structurii de date numite "dicționar". Vom considera o mulțime "atomi". Pentru fiecare element din aceasta mulțime avem: ∀ **a** ∈ atomi, este definita o funcție numita *cheie de căutare*: **key(a) ∈ k** cu proprietatea ca doi atomi distincți au chei diferite de căutare: $\mathbf{a_1 \neq a_2 \Rightarrow key(a_1) \neq key(a_2)}$.

Exemplu: (abreviere, definiție) ("**BST**","Binary Search Tree")

("LIFO","Last In First Out") **key(a) = a.abreviere**

Un *dicționar* este o colecție **S** de atomi pentru care se definesc operațiile:

- insert(S,a) — inserează atomul **a** în **S** daca nu exista deja;
- delete(S,k) — șterge atomul cu cheia **k** din **S** daca exista;
- search(S,k) — caută atomul cu cheia **k** în **S** si-l returnează sau determina daca nu este.

O soluție imediata ar fi reținerea elementelor din **S** într-o lista înlantuita, iar operațiile vor avea complexitatea *O(n).*

Tabelele Hashing. Acestea sunt o alta soluție pentru a retine elementele din **S**. Complexitatea pentru arborele binar de căutare în cazurile:

- cel mai defavorabil: *O(n)*;
- mediu: *O(log n)*.

Un arbore binar de căutare este un arbore **T** ale cărui noduri sunt etichetate cu atomii conținuți la un moment dat în dicționar.

T = (V, E) , **|V|** = **n**. (**n** atomi în dicționar)

Considerând **r ∈ V** (rădăcină arborelui), **Ts** — subarborele stâng al rădăcinii si **Td** — subarborele drept al rădăcinii, atunci structura acestui arbore este definita de următoarele proprietăți:

1) ∀ un nod **x ∈ Ts** atunci **key(data(x)) < key(data(r))**;

2) ∀ **x ∈ Td** atunci **key(data(x)) > key(data(r))**;

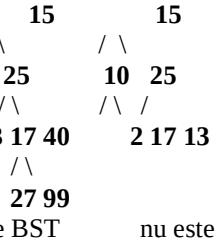
3) **Ts** si **Td** sunt BST.

Observații:

1) Consideram ca pe mulțimea **k** este definita o relație de ordine (de exemplu lexico-grafica);

2) Pentru oricare nod din BST toate nodurile din subarborele stâng sunt mai mici decât rădăcină si toate nodurile din subarborele drept sunt mai mari decât rădăcină.

Exemple:



este BST

nu este BST.

3) Inordine: vizitează nodurile în ordine crescătoare a cheilor: 2 7 9 13 15 17 25 27 40 99

Funcții:

1) ***Search***:

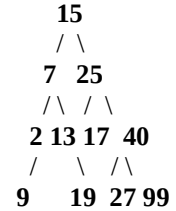
```
search(rad,k)          // rad — pointer la rădăcină arborelui
{                      // k — cheia de căutare a arborelui căutat
    if (rad = 0) then return NULL
    else
        if key (data (rad)) > k then
            return search (lchild (rad))
        else
            if key (data (rad)) < k then
```

```
                    return search (rchild (rad))
            else            return rad
}
```

2) ***Insert:***

Se va crea un nod în arbore care va fi plasat la un nou nod terminal. Poziția în care trebuie plasat acesta este unic determinata în funcție de valoarea cheii de căutare.

Exemplu: vom insera 19 în arborele nostru:




```
insert(rad,a)   // rad — referință la pointerul la rădăcină arborelui
{   if (rad= 0) then rad= make_nod(a)
        else if key (data (rad)) > key(a) then insert(lchild(rad),a)
        else if key (data(rad)) < key(a)then
        insert (rchild (rad),a)
}
```

Funcția *make_nod* creează un nou nod:

```
make_nod(a)
{   p= get_sp()     // alocare de memorie pentru un nod nou
    data(p)= a
    lchild(p)= 0
    rchild(p)= 0
    return(p)
}
```

Observație:

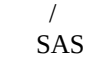
- La inserarea unui atom deja existent în arbore, funcția *insert* nu modifica structura arborelui. Exista probleme în care este utila contorizarea numărului de inserări a unui atom în arbore.
- Funcția *insert* poate returna pointer la rădăcină făcând apeluri de forma **p= insert(p,a).**

3) ***Delete:***

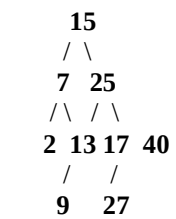
```
delete(rad,k)   // rad — referință la pointer la rădăcină
{                   // k — cheia de căutare a atomului care trebuie șters de noi
    if rad = 0 then return      // nodul cu cheia k nu se afla în arbore
            else if key(data(rad)) > k then     delete(lchild(rad),k)
    else if key(data(rad)) < k then delete(rchild(rad),k) else delete_root(rad)
}
```

Ștergerea rădăcinii unui BST.:

1) rad ⇒ arbore vid
2) a) rad sau b) rad ⇒ a) SAS sau b) SAD




```
delete_root(rad)        // rad — referință la pointer la rădăcină
{   if lchild(rad)=0 then
        ⌐ p= rchild(rad)
        | ret_sp(rad)
        |_ rad= p
    else if rchild(rad)= 0 then
        ⌐ p= lchild(rad)
        | ret_sp(rad)
        |_ rad= p
    else    ⌐ p= remove_greatest(lchild(rad))
        | lchild(p)= lchild(rad)
        | rchild(p)= rchild(rad)
        | ret_sp(rad)
        |_ rad= p
}
```



/ \
26 33

Detașarea din structura arborelui BST a celui mai mare nod — (remove_greatest):

Pentru a găsi cel mai mare nod dintr-un arbore binar de căutare, se înaintează în adâncime pe ramura dreapta până se găsește primul nod care nu are descendent dreapta. Acesta va fi cel mai mare.

Vom trata aceasta procedura recursiv:

Caz1: rad ⇒ se returnează pointer la rădăcină si arborele rezultat va fi vid.

Caz2: rad ⇒ se returnează pointer la rădăcină si arborele rezultat va fi format doar din SAS — subarborele stâng (SAS).

Caz3: rad ⇒ funcția returnează pointer la cel mai mare nod din SAD, iar rezultatul va fi SAS — arborele care este format din rădăcină, SAS și SAD — cel mai mare nod.

```
remove_greatest(rad) //rad -referință la pointer la  = rădăcină: un pointer la rădăcină poate fi modificat de către funcție
{   if rchild (rad)= 0 then     |¯ p= rad
                                |  rad= lchild (rad)
                                |_ return(p)
    else    return (remove_greatest (rchild(rad)))
}
```

Observație: Funcția *remove_greatest* modifica arborele indicat de parametru, în sensul eliminării nodului cel mai mare, si întoarce pointer la nodul eliminat.

Demonstrația eficientei (complexitate)

Complexitatea tuturor funcțiilor scrise depinde de adâncimea arborelui. În cazul cel mai defavorabil, fiecare funcție parcurge lanțul cel mai lung din arbore. Funcția de căutare are, în acest caz, complexitatea *O(n).*

Structura arborelui BST este determinata de ordinea inserării.

De exemplu, ordinea 15 13 12 11 este alta decât 15 12 11 13 .

Studiem un caz de complexitate medie:

Crearea unui BST pornind de la secvență de atomi ($a_1\ a_2$ ... a_n)

```
gen_BST         (va fi în programul principal)
| rad= 0
| for i= 1 to n
|       insert (rad, ai)
```

Calculam complexitatea medie a generării BST:

Complexitatea în cazul cel mai defavorabil este: $\sum_{i=1}^{n}(i)=O(n^2)$

Notam cu **T(k)** — numărul de comparații mediu pentru crearea unui BST pornind de la o secvență de **k** elemente la intrare. Ordinea celor **k** elemente se considera aleatoare.

Pentru problema **T(n)** avem de creat secvență **($a_1\ a_2$ ... a_n)** cu observația ca **a_1** este rădăcină arborelui. Ca rezultat, în urma primei operații de inserare pe care o facem, rezulta:

a1
/ \
a_1 a_i
(a_i<a1) (a_i>a1)

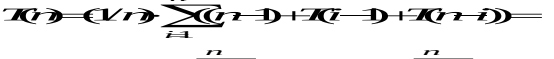
Nu vom considera numărarea operațiilor în ordinea în care apar ele, ci consideram numărul de operații globale. După ce am inserat **a_1**, pentru inserarea fiecărui element în SAS sau SAD a lui **a_1**, se face o comparație cu **a_1**. Deci:

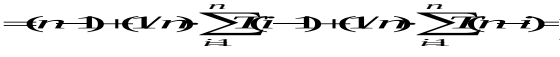
T(n)= (n - 1) + val.med.SAS + val.med.SAD

val.med.SAS = valoarea medie a numărului de comparații necesar pentru a construi subarborele stâng SAS
val.med.SAD = valoarea medie a numărului de comparații necesar pentru a construi subarborele drept SAD

Notam: **$T_i(n)$** = numărul mediu de comparații necesar pentru construirea unui BST cu **n** noduri atunci când prima valoare inserata (**a_1**) este mai mare decât **i-1** dintre cele **n** valori de inserat. Putem scrie:

Atunci:



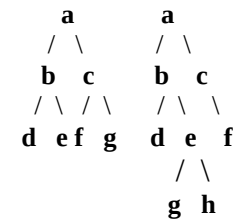


Deci:

Complexitatea acestei funcții este: *O(n·ln n)* (vezi curs 5 — complexitatea medie a algoritmului Quick-Sort)

1.2.2.5 Arbori binari de căutare dinamic echilibrați (AVL).

Definiție Un arbore binar este echilibrat daca si numai daca, pentru fiecare nod din arbore, diferența dintre adâncimile SAS si SAD în modul este ≤ 1. ***Exemple:***

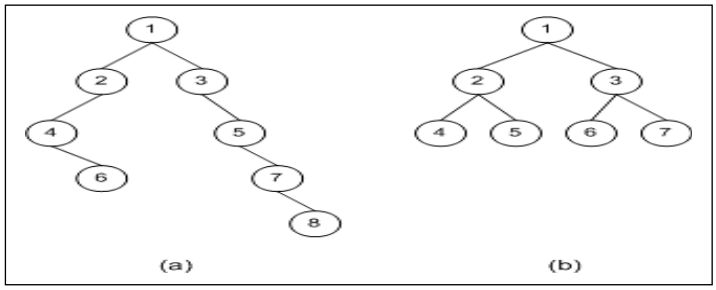


arbore binar complet echilibrat — arbore binar echilibrat

Adâncimea unui arbore echilibrat cu **n** noduri este *O(ln n)*.

Se completează operațiile *insert* si *delete* cu niște prelucrări care sa păstreze proprietățile de arbore binar echilibrat pentru arborele binar de căutare. Arborele binar echilibrat este un BST echilibrat, proprietatea de echilibrare fiind conservata de *insert* si *delete*. Efortul, în plus, pentru completarea operațiilor *insert* si *delete* nu schimba complexitatea arborelui binar echilibrat.

1.3. Aplicații de *arbori binari* în C. Exemplu de arbori binari:



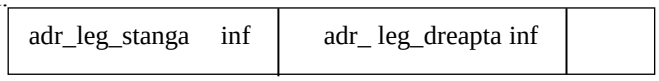
Arborele din figura (a) are 8 noduri, nodul 1 fiind rădăcina. Acesta are ca și copil stânga nodul nr.2, iar ca și copil dreapta nodul nr.3. La rândul său nodul nr.2 nu are decât un singur copil (stânga), și anume nodul nr 4.

Deci un nod dintr-un arbore binar poate avea 2 copii (stânga și dreapta), un singur copil (doar stânga sau doar dreapta) sau nici unul (exemplu nodul 8).

Nodurile care nu au nici un copil se numesc **noduri frunză**.

Nodurile care au 1 sau 2 copii se numesc **noduri interne**.

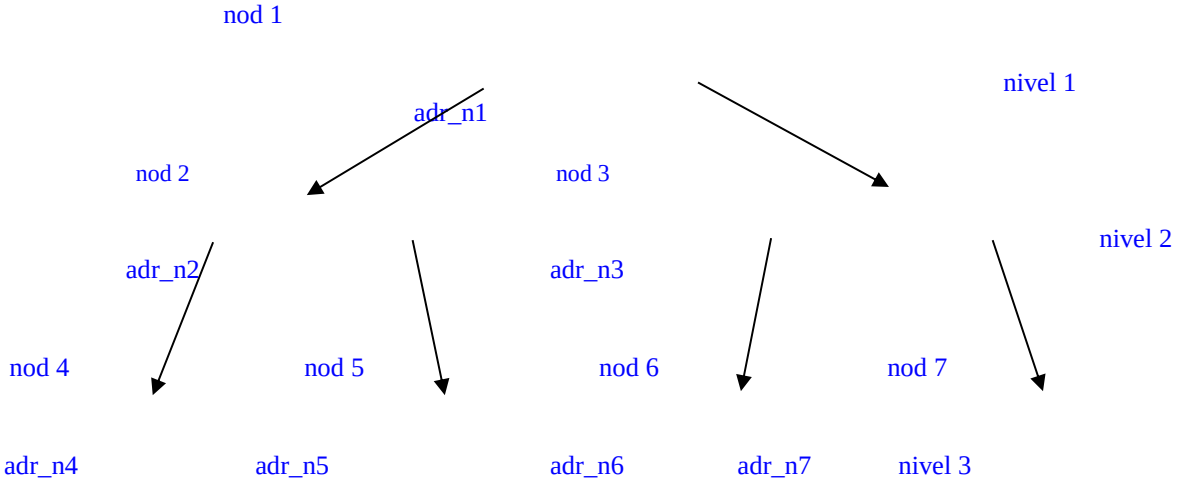
Pentru reținerea informației în calculator se va folosi alocarea dinamică. Deoarece pentru fiecare nod în parte este necesar să se rețină, pe lângă informația utilă, și legăturile cu nodurile copii (adresele acestora), ne putem imagina următoarea structura a unui nod:




```
struct Nod
{  int inf;
   Nod *adr_leg_stanga;
   Nod *adr_leg_dreapta;  };
Nod *primul_nod;
```

Unde **adr_leg_stanga** reprezintă adresa nodului copil din stânga, **inf** reprezintă câmpul cu informație utilă, iar **adr_leg_dreapta** este legătura cu copilul din dreapta.

Arborelui binar din figura (b) de mai sus, va avea următoarea structura interna in memoria calculatorului :



Pe nivelul 1 vom avea un singur nod, nodul rădăcină. Componenta **adr_leg_stanga** va fi egală cu adresa **adr_n2**, a nodul din stânga de pe nivelul 2, iar componenta **adr_leg_dreapta** va fi egală cu adresa **adr_n3**, a nodului din dreapta de pe nivelul 2.

Componentele **adr_leg_stanga** și **adr_leg_dreapta** ale nodului din stânga de pe nivelul 2, vor fi egale cu **adr_n4**, respectiv **adr_n5**.

Componentele **adr_leg_stanga** și **adr_leg_dreapta** ale nodului din dreapta de pe nivelul 2, vor fi egale cu **adr_n6**, respectiv **adr_n7**.

Toate nodurile din stânga, respectiv dreapta de pe nivelul 3 nu mai are nici un copil, descendent, și de aceea componentele lor, **adr_leg_stanga** și **adr_leg_dreapta** vor avea valoarea **NULL**, ca și la listele dinamice.

Alt exemplu, mai general, un nod al unui arbore binar poate fi o structură care poate fi definită în felul următor:

```
typedef struct tnod
 { int nr,   int f;                    //declarații
                 struct tnod *st;      // este pointerul spre subarborele stâng al //nodului curent
                 struct tnod *dr;      // este pointerul spre subarborele drept al //nodului curent
} TNOD;
```

Asupra arborilor binari pot fi definite următoarele operații:

- *afișarea componentelor informaționale ale nodului,*
- *specificarea criteriului de determinare a poziției în care să se insereze în arbore nodul curent;*
- *determinarea echivalenței a doi arbori;*
- inserarea unui nod terminal într-un arbore binar;
- accesarea unui nod al arborelui,
- parcurgerea unui arbore;
- ștergerea unui arbore.

Afișarea componentelor informaționale ale nodului se poate de efectuat prin funcția:

```
void prelucrare (TNOD *p)
{printf("numărul = %d apariții= %d \n", p->nr, p->f);}
```

Criteriul de determinare a poziției, în care să se insereze în arbore nodul curent, se definește de funcția:

```
int criteriu(TNOD *p, *q)
        { if (q->nr < p -> nr )
        return –1; // inserarea nodului curent în subarborele stâng al nodului spre care indică pointerul p
 if (q->nr > p-> nr )
 return 1;  // inserarea nodului curent în subarborele drept al nodului spre care indică  pointerul p
}
```

Inserarea unui nod terminal într-un arbore binar poate fi efectuată prin următoarea funcție:

```
TNOD* insert_nod()
{ TNOD *parb, *p, *q;
  int n=sizeof(TNOD);
 if (parb ==0) { parb=p; return p; }
int i; q=parb;
for(;;)   if ((i=criteriu(q,p)) <0) {q->st=p;  return p; }   else  {  q=q->st;  continue;  }
if (i>0)      if (q->dr ==0)       {q->dr=p;  return p;}       else  {q=q->dr; continue; }
return eq(q,p); }
}
if(p==0) { printf("eroare: memorie insuficientă\n"); exit(1);}   elibnod(p);   return 0;  }
```

Accesarea unui nod al unui arbore poate fi realizată prin următoarea funcție:

```
TNOD * caută (TNOD *p)
{TNOD *parb, *q;
if (parb==0) return 0;
int i;
for (q=parb;q;)
  if ((i=criteriu(q,parb))==0) return q;
  else   if(I<0)  q=q->st;
           else    q=q->dr;
return 0; }
```

Parcurgerea unui arbore poate fi efectuată în trei modalități: *în preordine; în inordine; în postordine.*

Parcurgerea în preordine *presupune accesul la rădăcină și apoi parcurgerea celor doi subarbori ai săi: mai întâi subarborele stâng, apoi cel drept.*

```
void preord (TNOD *p)
{ if (p!=0)  {  prelucrare(p); preord(p->st); preord(p->dr); }  }
```

Parcurgerea în inordine presupune parcurgerea mai întâi a subarborelui stâng, apoi accesul la rădăcină și în continuare se parcurge subarborele drept.

```
void inord (TNOD *p)
{  if (p!=0)   {inord(p->st);  prelucrare(p); inord(p->dr);}          }
```

Parcurgerea în postordine *presupune parcurgerea mai întâi a subarborelui stâng, apoi a arborelui drept și, în final, accesul la rădăcina arborelui.*

```
void postord (TNOD *p)
{ if (p!=0)    {  postord(p->st);  postord(p->dr);  prelucrare(p);  }}
```

Ștergerea unui arbore poate fi efectuată de următoarea funcție:

```
void elib_nod(TNOD *p)
{  delete(p);  }
    void sterge_arbore (TNOD *p)
    { if (p!=0)  {  postord(p->st);  postord(p->dr);  elibnod(p);  }
}
```

AVL-urile sunt arbori de căutare echilibrați care au complexitate **O(lg n)** pe operațiile de inserare, ștergere si căutare.

2. Principalele operații asupra arborilor binari de căutare (inserare, căutare, ștergere, traversare).

Arborii binari de căutare sunt des folosiți pentru memorarea și regăsirea rapidă a unor informații, pe baza unei chei. Fiecare nod al arborelui trebuie să conțină o cheie distinctă.

Structura unui nod al unui arbore binar de căutare este următoarea:

```
typedef struct tip_nod {  tip cheie;
                          informații_utile;
                          struct tip_nod *stg, *dr;
                        } TIP_NOD;
```

În cele ce urmează, rădăcina arborelui se consideră ca o variabilă globală:

```
TIP_NOD *rad;
```

Structura arborelui de căutare depinde de ordinea de inserare a nodurilor.

2.1. Inserarea într-un arbore binar de căutare.

Construcția unui arbore binar de căutare se face prin inserarea a câte unui nod de cheie key. Algoritmul de inserare este următorul:

a) Dacă arborele este vid, se creează un nou nod care este rădăcina, cheia având valoarea key, iar subarborii stâng și drept fiind vizi.

b) Dacă cheia rădăcinii este egală cu key atunci inserarea nu se poate face întrucât există deja un nod cu această cheie.

c) Dacă cheia key este mai mică decât cheia rădăcinii, se reia algoritmul pentru subarborele stâng (pasul a).

d) Dacă cheia key este mai mare decât cheia rădăcinii, se reia algoritmul pentru subarborele drept (pasul a).

Funcția nerecursivă de inserare va avea următorul algoritm:

```
void inserare_nerecursivă (int key)
    {TIP_NOD *p, *q;
     int n;     /* construcție nod p*/
     n=sizeof (TIP_NOD);          p=(TIP_NOD*)malloc(n);          p->cheie=key;
     /* introducere informație utilă în nodul p */
     p->stg=0; p->dr=0; /* este nod frunză */
     if (rad==0) { /* arborele este vid */
                    rad=p;   return;        }
     /* arborele nefiind vid se caută nodul tată pentru nodul p */
     q=rad; /* rad este rădăcina arborelui variabilă globală */
     for ( ; ; )  { if (key<q->cheie) { /* căutarea se face în subarborele stâng */
                    if (q->stg==0) { /* inserare */
                                     q->stg=p; return;  }  else q=q->stg;  }
          else if (key>q->cheie)  { /* căutarea se face în subarborele drept */
                                if (q->dr==0) { /* inserare */
                                                q->dr=p; return;   } else q=q->dr;  }
               else { /* cheie dublă */     /* scriere mesaj */
                    free (p);     return;     }
                  }
     }
```

2.2 Căutarea unui nod de cheie dată *key* într-un arbore binar de căutare.

Căutarea într-un arbore binar de căutare a unui nod de cheie dată se face după un algoritm asemănător cu cel de inserare.

Numărul de căutări optim ar fi dacă arborele de căutare ar fi total echilibrat (numărul de comparații maxim ar fi $log_2 n$ – unde n este numărul total de noduri).

Cazul cel mai defavorabil în ceea ce privește căutarea este atunci când inserarea se face pentru nodurile având cheile ordonate crescător sau descrescător. În acest caz, arborele degenerează într-o listă.

Algoritmul de căutare este redat prin funcția următoare:

```
TIP_NOD *căutare (TIP_NOD *rad, int key)
/* funcția returnează adresa nodului în caz de găsire sau 0 în caz că nu există un nod de cheia key */
{          TIP_NOD *p;
if (rad==0) return 0; /* arborele este vid  dacă arborele nu este vid, căutarea începe din rădăcina rad */
p=rad;
while (p!=0)   {   if (p->cheie==key) return p;      /* s-a găsit */
                        else if (key<p->cheie) p=p->stg; /*căutarea se face în subarb.stâng */
                             else p=p->dr; /* căutarea se face în subarborele drept */
```

```
            }
        return 0; /* nu există nod de cheie key */
    }
```

Apelul de căutare este:

p=căutare (rad, key); rad fiind pointerul spre rădăcina arborelui.

2.3 Ștergerea unui nod de cheie dată într-un arbore binar de căutare

În cazul ștergerii unui nod, arborele trebuie să-și păstreze structura de arbore de căutare.

La ștergerea unui nod de cheie dată intervin următoarele cazuri:

a) Nodul de șters este un nod frunză. În acest caz, în nodul tată, adresa nodului fiu de șters (stâng sau drept) devine zero.

b) Nodul de șters este un nod cu un singur descendent. În acest caz, în nodul tată, adresa nodului fiu de șters se înlocuiește cu adresa descendentului nodului fiu de șters.

c) Nodul de șters este un nod cu doi descendenți. În acest caz, nodul de șters se înlocuiește cu nodul cel mai din stânga al subarborelui drept sau cu nodul cel mai din dreapta al subarborelui stâng.

Algoritmul de ștergere a unui nod conține următoarele etape:

- căutarea nodului de cheie key și a nodului tată corespunzător;
- determinarea cazului în care se situează nodul de șters.

2.4 Ștergerea unui arbore binar de căutare.

Ștergerea unui arbore binar de căutare constă în parcurgerea în postordine a arborelui și ștergerea nod cu nod, conform funcției următoare:

```
void stergere_arbore(TIP_NOD *rad)
{  if (rad !=0) {
                    stergere_arbore (rad->stg);
                    stergere_arbore (rad->dr);
                    free (rad);
                  }
 }
```

2.5 Traversarea unui arbore binar de căutare

Ca orice arbore binar, un arbore binar de căutare poate fi traversat în cele trei moduri: în preordine, în inordine și în postordine conform funcțiilor de mai jos:

```
void preordine (TIP_NOD *p)
{ if (p!=0) {
              extragere informație din nodul p;
              preordine (p->stg);
              preordine (p->dr);
            }    }
            void inordine (TIP_NOD *p)
            {
                if (p!=0) {
                            inordine (p->stg);
                            extragere informație din p;
                            inordine (p->dr);
                          }     }
void postordine (TIP_NOD *p)
{
    if (p!=0) {
                postordine (p->stg);
                postordine (p->dr);
                extragere informație din nodul p;
              } }
                Apelul acestor funcții se va face astfel:
                        preordine(rad);
                        inordine(rad);
                        postordine(rad);
```

2.6 Arbori binari de căutare optimali

Lungimea drumului de căutare a unui nod cu cheia *x*, într-un arbore binar de căutare, este nivelul h_i al nodului în care se află cheia căutată, în caz de succes sau 1 plus nivelul ultimului nod întâlnit pe drumul căutării fără succes.

Fie $S = \{ c_1, c_2, \ldots, c_n \}$ mulțimea cheilor ce conduc la căutarea cu succes ($c_1 < c_2 < \ldots < c_n$).

Fie p_i probabilitatea căutării cheii c_i ($i=1,2,\ldots,n$).

Dacă notăm cu C, mulțimea cheilor posibile, atunci C $\setminus$ S reprezintă mulțimea cheilor ce conduce la căutarea fără succes. Această mulțime o partiționăm în submulțimile:

k_0 – mulțimea cheilor mai mici ca c_1;

k_n – mulțimea cheilor mai mari ca c_n;

k_i (i=1,2,...,n) – mulțimea cheilor în intervalul (c_i, c_{i+1}).

Fie q_i probabilitatea căutării unei chei din mulțimea k_i.

Căutarea pentru orice cheie din k_i se face pe același drum; lungimea drumului de căutare va fi h'_i.

Notăm cu L costul arborelui, care reprezintă lungimea medie de căutare:

cu condiția:

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1 \qquad L = \sum_{i=1}^{n} p_i h'_i + \sum_{j=0}^{n} q_j$$

Se numește arbore optimal, un arbore binar de căutare care pentru anumite valori p_i, q_i date realizează un cost minim.

Arborii optimali de căutare nu sunt supuși inserărilor și eliminărilor.

Din punct de vedere al minimizării funcției L, în loc de p_i și q_i se pot folosi frecvențele apariției căutărilor respective în cazul unor date de test.

Se notează cu A_{ij} arborele optimal construit cu nodurile c_{i+1}, c_{i+2}, ..., c_j.

Greutatea arborelui A_{ij} este:

$$q_{ij} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

care se poate calcula astfel:

Rezultă că costul arborelui optimal A_{ij} se va putea calcula astfel:

$$q_{ii} = q_i \qquad \text{pentru } i = 0, n$$

$$q_{ij} = q_{i,j-1} + p_j + q_j \qquad \text{pentru } 0 \le i < j \le n$$

$$c_{ii} = q_{ii} \qquad \text{pentru } 0 \le i \le n$$

Fie r_{ij} valoarea lui k pentru care se obține minimul din relația lui c_{ij}. Nodul cu cheia $c[r_{ij}]$ va fi rădăcina subarborelui optimal A_{ij}, iar subarborii săi vor fi $A_{i,k-1}$ și A_{kj}.

Calculul valorilor matricei C este de ordinul $O(n^3)$. S-a demonstrat că se poate reduce ordinul timpului de calcul la $O(n^2)$.

Construirea se face cu ajutorul funcției următoare:

```
TIP_NOD *constr_arbore_optimal(int i, int j)
{
   int n;
   TIP_NOD *p;
   if(i==j) p=0;
   else {
          n=sizeof (TIP_NOD);
          p=(TIP_NOD*)malloc(n);
          p->stg=constr_arbore_optimal(i, r[i][j]-1);
          p->stg=constr_arbore_optimal[r[i][j]];
          p->dr=constr_arbore_optimal(r[i][j], j);
         }
   return p;
 }
```

2.7 Exemple-model pentru analiza principiilor de prelucrare și rezultatele execuției.

1. Primul program prezintă toate funcțiile descrise în lucrare asupra unui arbore de căutare. Un nod conține drept informație utilă numai cheia, care este un număr întreg.
2. Al doilea program conține funcțiile principale asupra unui arbore binar de căutare optimal.
3. **Exemplul nr.3** construiește un arbore binar cu cele N valori, traversează și afișează arborele, prin metodele: preordine, inordine, postordine. De asemenea, prin intermediul programului, se va căuta în arbore o valoare dată, afișând un mesaj traversarea în preordine (*RSD*): se face în ordinea - rădăcină, traversarea în inordine (*SRD*): se face în ordinea - stânga, traversarea în postordine (*SDR*): se face în ordinea – stânga.

Analizați! Comentați afișați organigramele și toate componentele! Îmbunătățiți !!!!!!!!!!

2.7.1. Exemplul nr.1 (arbori de căutare)

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
typedef struct tip_nod{
                        int cheie;/*informație */
                        struct tip_nod *stg,*dr;
                       } TIP_NOD;
TIP_NOD *rad;
    void preordine(TIP_NOD *p, int nivel)
    {
     int i;
```

```
    if (p!=0){
          for(i=0;i<=nivel;i++) printf(" ");
          printf("%2d\n",p->cheie);
          preordine(p->stg,nivel+1);
          preordine(p->dr,nivel+1);
        }
    }
        void inordine(TIP_NOD *p, int nivel)
        {
         int i;
         if (p!=0){
                  inordine(p->stg,nivel+1);
                  for(i=0;i<=nivel;i++) printf(" ");
                  printf("%2d\n",p->cheie);
                  inordine(p->dr,nivel+1);
             }
        }
                 void postordine(TIP_NOD *p, int nivel)
                 {
                  int i;
                  if (p!=0){
                           postordine(p->stg,nivel+1);
                           postordine(p->dr,nivel+1);
                           for(i=0;i<=nivel;i++) printf(" ");
                           printf("%2d\n",p->cheie);
                       }
                 }
void inserare(int key)
{
 TIP_NOD *p,*q;
 int n;
 n=sizeof(TIP_NOD);
 p=(TIP_NOD *)malloc(n);
 p->cheie=key;
 p->stg=0;p->dr=0;
 if(rad==0){
          rad=p;
          return;
      }
 q=rad;
 for(;;)
   {
          if (key < q->cheie){
                                    if(q->stg==0){
                                             q->stg=p;
                                              return;
                                      }
                        else q=q->stg;
                                 }
          else if (key > q->cheie) {
                                      if(q->dr == 0) {
                                                          q->dr=p;
                                                          return;
                                                    }
                                      else q=q->dr;
                                     }
              else {  /* chei  egale */
                        printf("\n Exista un nod de cheie = %d\n",key);
                                  /* eventuala prelucrare a nodului */
                        free(p);
                        return;
                      }
 }
   }
TIP_NOD *inserare_rec(TIP_NOD *rad,int key)
{
 TIP_NOD *p;
```

```
 int n;
 if (rad==0){
            n=sizeof(TIP_NOD);
            p=(TIP_NOD *)malloc(n);
            p->cheie=key;p->stg=0;p->dr=0;
            return p;
        }
 else {
      if(key < rad->cheie) rad->stg=inserare_rec(rad->stg,key);
    else {
          if(key > rad->cheie) rad->dr=inserare_rec(rad->dr,key);
            else { /* cheie dubla */
                        printf("\n Exista un nod de cheie=%d\n",key);
                }
        }
   };
  return rad;
}

TIP_NOD * căutare(TIP_NOD *rad, int key)
{
   TIP_NOD *p;

   if(rad==0) return 0;/*arborele este vid */
   p=rad;
   while(p != 0)
            {
            if(p->cheie == key) return p;/* s-a găsit nodul */
            else if(key < p->cheie) p=p->stg;
                else p=p->dr;
            }
   return 0; /* nu exista nod de cheie key */
  }
TIP_NOD *stergere_nod(TIP_NOD *rad,int key)
{
TIP_NOD *p,*tata_p;/* p este nodul de șters, iar tata_p este tatăl lui */
TIP_NOD *nod_inlocuire,*tata_nod_inlocuire;/*nodul care îl va înlocui pe p si tatăl sau */
 int directie; /*stg=1;dr=2*/
 if(rad==0) return 0; /*arborele este vid */
 p=rad; tata_p=0;
 /* căutare nod cu cheia key */
 while((p!=0)&&(p->cheie!=key))
    {
            if (key<p->cheie){    /*căutare in stânga */
                                        tata_p=p;
                                        p=p->stg;
                                        directie=1;
                                    }
             else {  /*căutare in dreapta */
                        tata_p=p;
                        p=p->dr;
                        directie=2;
                       }
     }
   if(p==0){
            printf("\n NU EXISTA NOD CU CHEIA=%d\n",key);
            return rad;
       }
   /* s-a găsit nodul p de cheie key */
   if(p->stg==0) nod_inlocuire=p->dr; /* nodul de șters p nu are fiu sting */
         else if(p->dr==0) nod_inlocuire=p->stg;                /*nodul de șters p nu are fiu drept*/
    else { /* nodul de șters p are fiu stâng si fiu drept */
             tata_nod_inlocuire=p;
             nod_inlocuire=p->dr;    /* se caută in subarborele drept*/
             while(nod_inlocuire->stg!=0)
               {
                        tata_nod_inlocuire=nod_inlocuire;
```

```
                    nod_inlocuire=nod_inlocuire->stg;
                }
              if(tata_nod_inlocuire!=p)
                    {
                     tata_nod_inlocuire->stg=nod_inlocuire->dr;
                     nod_inlocuire->dr=p->dr;
                    }
               nod_inlocuire->stg=p->stg;
             }
   free(p);
   printf("\nNodul de cheie=%d a fost șters!\n",key);
   if(tata_p==0) return nod_inlocuire; /*s-a șters chiar rădăcină inițială */
   else {
     if (directie==1) tata_p->stg=nod_inlocuire;
     else tata_p->dr=nod_inlocuire;
     return rad;
      }
}
void stergere_arbore(TIP_NOD *rad)
{
 if(rad!=0) {
              stergere_arbore(rad->stg);
          stergere_arbore(rad->dr);
              free(rad);
      }
}
            void main(void)
            {
             TIP_NOD *p;
             int i, n,key;
             char ch;
             printf("ALEGETI Inserare recursiva r/R sau nerecursiva alt caracter");
             scanf("%c",&ch);
             printf("\nNumarul total de noduri=");
             scanf("%d",&n);
             rad=0;
             for(i=1;i<=n;i++)
               {
                  printf("\nCheia nodului=");
                  scanf("%d",&key);
                  if((ch=='R')||(ch=='r')) rad=inserare_rec(rad,key);
                  else inserare(key);
                }
             printf("\nVIZITAREA IN PREORDINE\n");
             preordine(rad,0);
             getch();
             printf("\nVIZITAREA IN INORDINE\n");
             inordine(rad,0);
             getch();
             printf("VIZITAREA IN POSTORDINE\n");
             postordine(rad,0);
             getch();
             fflush(stdin);
             printf("\n Doriți sa căutați un nod DA=D/d Nu= alt caracter :");
             scanf("%c",&ch);
             while((ch=='D')||(ch=='d'))
                        {
                         printf("Cheia nodului căutat=");
                         scanf("%d",&key);
                         p=căutare(rad,key);
                        if(p!=0) printf("Nodul exista si are adresa p\n");
                        else printf("Nu exista un nod de cheie data\n");
                        fflush(stdin);
                        printf("\n Doriți sa căutați un nod DA=D/d Nu= alt caracter : ");
                        scanf("%c",&ch);
                        }
             fflush(stdin);
```

```
printf("\n Doriți sa ștergeți un nod DA=D/d Nu= alt caracter :");
scanf("%c",&ch);
while((ch=='D')||(ch=='d'))
        {
         printf("Cheia nodului de șters=");
         scanf("%d",&key);
         rad=stergere_nod(rad,key);
         inordine(rad,0);
         fflush(stdin);
         printf("\n Doriți sa ștergeți un nod DA=D/d Nu= alt caracter : ");
         scanf("%c",&ch);
        }
printf("ștergeți arborele creat ? da=d/D nu=alt caracter ");
fflush(stdin);
scanf("%c",&ch);
if((ch=='D')||(ch=='d')) {    stergere_arbore(rad);        rad=0;
                              printf("\nARBORELE ESTE STERS!!\n");     }
getch(); }
```

2.7.2. Exemplul nr.2 (arbori optimali)

```
        #include <stdio.h>
        #include <conio.h>
        #include <alloc.h>
        #define nmax 25
        typedef struct tip_nod {   char cheie;   tip_nod *stg,*dr;  } TIP_NOD;
        char chei[nmax]; /* cheile c1,c2,...,cn */
        int p[nmax];/* frecventa de căutare a cheilor */
        int q[nmax]; /* frecventa de căutare intre chei */
        int r[nmax][nmax]; /* rădăcinile subarborilor optimali */
void calcul(int nr,float *dr_med)   {            /* determina structura arborelui */
         int c[nmax][nmax];/* costul subarborilor optimali */
         int g[nmax][nmax]; /* greutatea arborilor */
         int i,j,k,m,l;
         int x,min;
         /* calculul matricei greutate */
         for(i=0;i<=nr;i++)
                {
                        g[i][i]=q[i];
                        for(j=i+1;j<=nr;j++)
                                g[i][j]=g[i][j-1]+p[j]+q[j];
                }
         /* calculul matricei c */
         for(i=0;i<=nr;i++)
                c[i][i]=g[i][i];
         for(i=0;i<=nr-1;i++)
                {
                        j=i+1;
                        c[i][j]=c[i][i]+c[j][j]+g[i][j];
                        r[i][j]=j;
                }
         /*calcul c[i][l+i] */
         for(l=2;l<=nr;l++)
                for(i=0;i<=nr-l;i++)
                        {
                     min=32000;
                     for(k=i+1;k<=l+i;k++)
                           {
                            x=c[i][k-1]+c[k][l+i];
                            if(x<min) {
                                        min=x;
                                        m=k;
                                            }
                           }
                      c[i][l+i]=min+g[i][l+i];
                      r[i][l+i]=m;
                         }
                printf("\nMATRICEA G\n");
```

```
                for(i=0;i<=nr;i++)
                        { for(j=i;j<=nr;j++)
                          printf("%d ",g[i][j]);
                    printf("\n");
                        }
                getch();
                printf("\nMATRICEA C\n");
                for(i=0;i<=nr;i++)
                        { for(j=i;j<=nr;j++)
                          printf("%d ",c[i][j]);
                    printf("\n");
                        }
                getch();
            printf("\nMATRICEA R\n");
                for(i=0;i<=nr;i++)
                        { for(j=i;j<=nr;j++)
                          printf("%d ",r[i][j]);
                    printf("\n");
                        }
                getch();
                printf("c[0][nr.]=%ld g[0][nr.]=%ld\n",c[0][nr.],g[0][nr.]);
                getch();
                *dr_med=c[0][nr.]/(float)g[0][nr.];
}
     TIP_NOD* constr_arbore(int i,int j)
      /* construirea arborelui optimal */
     {          int n;     TIP_NOD *p;
      if (i==j) p=0;
      else {                        n=sizeof(TIP_NOD);
                                    p=(TIP_NOD*)malloc(n);
                                    p->stg=constr_arbore(i,r[i][j]-1);
                                    p->cheie=chei[r[i][j]];
                                    p->dr=constr_arbore(r[i][j],j);
                }
      return p;          }
     void inordine(TIP_NOD *p,int nivel)
{    /* Afişare in inordine a arborelui */
      int i;
      if(p!=0){  inordine(p->stg,nivel+1);
                 for(i=0;i<=nivel;i++)    printf("  ");
                 printf("%c\n",p->cheie);   inordine(p->dr,nivel+1);  }
 }
 void main(void)
 {     TIP_NOD *rădăcină;       int i;
       int n; /*n este numărul cheilor */
       float drum_mediu;
       printf("\nNumarul cheilor=");        scanf("%d",&n);
       /*Citirea cheilor si a frecventelor de căutare a lor*/
       for(i=1;i<=n;i++) { printf("Cheia[%d]=",i);  chei[i]=getche();
          printf(" frecventa=");  scanf("%d",&p[i]);    }
       /*Citirea frecventelor de căutare intre chei */
       for(i=0;i<=n;i++)   {      printf("q[%d]=",i);   scanf("%d",&q[i]);  }
        calcul(n,&drum_mediu);
        printf("Drumul mediu=%6f\n",drum_mediu);   getch();
        rădăcină=constr_arbore(0,n);  inordine(radacina,0);              getch();
      }
```

2.7.3. **Exemplul nr.3** Să se realizeze un program care citeşte de la tastatură N numere întregi cuprinse în domeniul [1...NMAX], construieşte un arbore binar cu cele N valori, traversează şi afişează arborele, prin metodele: preordine, inordine, postordine. De asemenea, prin intermediul programului, se va căuta în arbore o valoare dată, afişând un mesaj traversarea în preordine (*RSD*): se face în ordinea - rădăcină, traversarea în inordine (*SRD*): se face în ordinea - stânga, traversarea în postordine (*SDR*): se face în ordinea - stânga,

1. Indiferent de tipul de parcurgere al arborilor -- ordinea relativă de tratare
2. Forma de parcurgere care generează forma cea mai apropiată de scrierea uzuală.
3. De asemenea, prin intermediul programului, se va căuta în arbore o valoare dată, afişând un mesaj corespunzător la găsirea sa.

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

```
#include <alloc.h>
#include <stdlib.h>
#define NR 10 //număr de noduri
        typedef struct Nod
        { void *Info;
        struct Nod *Stanga, *Dreapta; }NodT, *NodPtrT;  NodPtrT t; NodPtrT *root;
int TestVid(NodPtrT Rădăcina) {return Rădăcina==NULL; }
int AdrInfo(NodPtrT Rădăcina, void *InfoPtr)
{ if (Rădăcina!=NULL) InfoPtr=(void *)Rădăcina->Info; else InfoPtr=NULL; return (Rădăcina!=NULL &&
InfoPtr!=NULL); }
        int AdrStanga(NodPtrT Rădăcina, void *InfoPtr)       {
        if (Rădăcina!=NULL) InfoPtr=Rădăcina->Stanga;     else InfoPtr=NULL;
        return (Rădăcina!=NULL && InfoPtr!=NULL);       }
int AdrDreapta(NodPtrT Rădăcina, void *InfoPtr)     {
if (Rădăcina) InfoPtr=Rădăcina->Dreapta;    else InfoPtr=NULL;
return (Rădăcina!=NULL && InfoPtr!=NULL);       }
        int ConstrNod(NodPtrT *Rădăcina, void *Info, void *Stanga, void *Dreapta)     {
        *Rădăcina=malloc(sizeof(NodT));
        if (!(Rădăcina)) return 0;
        (*Rădăcina)->Info=Info;
        (*Rădăcina)->Stanga=Stanga;
        (*Rădăcina)->Dreapta=Dreapta;
        return 1; }
int ModificaInfo(NodPtrT Rădăcina, void *InfoPtr) {
if (Rădăcina) Rădăcina->Info=InfoPtr;
return (Rădăcina!=NULL);  }
        int ModificaStanga(NodPtrT Rădăcina, NodPtrT SubArbore)  {
        if (Rădăcina) Rădăcina->Stanga=SubArbore; return (Rădăcina!=NULL); }
int ModificaDreapta(NodPtrT Rădăcina, NodPtrT SubArbore)
{if (Rădăcina) Rădăcina->Dreapta=SubArbore; return (Rădăcina!=NULL); }
int ElibNod(NodPtrT *Rădăcina, void** Info, NodPtrT *Stanga, NodPtrT *Dreapta)
{if (!(*Rădăcina)) return 0; else
{*Info=(void*)(*Rădăcina)->Info; *Stanga=(*Rădăcina)->Stanga;
*Dreapta=(*Rădăcina)->Dreapta; free(*Rădăcina); *Rădăcina=NULL; return 1; } }
int Terminal(NodPtrT Rădăcina)
{return (Rădăcina->Stanga==NULL && Rădăcina->Dreapta==NULL); }
int NrNoduri(NodPtrT Rădăcina)
{if (!Rădăcina) return 0;
else return(NrNoduri(Radacina->Stanga)+NrNoduri(Radacina->Dreapta)+1);
return (InaltimeSt>=InaltimeDr ? InaltimeSt : InaltimeDr)+1;
AdaugaNod(&((*ArborePtr)->Stanga), Nod);
AdaugaNod(&((*ArborePtr)->Dreapta), Nod);  }
int Inaltime(NodPtrT Rădăcina)
{if (Rădăcina==NULL) return 0; else
{ int InaltimeSt, InaltimeDr;
InaltimeSt=Inaltime(Rădăcina->Stanga);
InaltimeDr=Inaltime(Rădăcina->Dreapta);
return (InaltimeSt>=InaltimeDr ? InaltimeSt : InaltimeDr)+1;
return (Rădăcina->Stanga==NULL && Rădăcina->Dreapta==NULL);
else return(NrNoduri(Radacina->Stanga)+NrNoduri(Radacina->Dreapta)+1);
return (InaltimeSt>=InaltimeDr ? InaltimeSt : InaltimeDr)+1; } }
void AdaugaNod(NodPtrT *ArborePtr, NodPtrT Nod) {
if (*ArborePtr==NULL) { printf("R\t"); *ArborePtr=Nod; printf("%#x\t",Nod);
printf("%d\n",*(int*)Nod->Info); } else
{ switch (random(2))
{case 0: printf("s"); AdaugaNod(&((*ArborePtr)->Stanga), Nod); break;
case 1: printf("d"); AdaugaNod(&((*ArborePtr)->Dreapta), Nod); break; } } }
void ConstrArbore(NodPtrT *ArborePtr, int NrNoduri,int node[NR])
void *InfoPtr;
NodPtrT NodNou; { InfoPtr=&node[NR-NrNoduri];
if (ConstrNod(&NodNou, InfoPtr, NULL, NULL)) AdaugaNod(ArborePtr, NodNou);
else break;  } while (--NrNoduri>0); }
void PreOrdTrav(NodPtrT Rădăcina) {
if (Rădăcina) { printf("%d\t",*(int*)Rădăcina->Info); PreOrdTrav(Rădăcina->Stanga);
PreOrdTrav(Rădăcina->Dreapta); } }
void InOrdTrav (NodPtrT Rădăcina) { if (Rădăcina)
{ InOrdTrav(Rădăcina->Stanga);
```

```
printf("%d\t",*(int*)Rădăcina->Info);
InOrdTrav(Rădăcina->Dreapta); } }
void PostOrdTrav (NodPtrT Rădăcina) { if (Rădăcina)
{ PostOrdTrav(Rădăcina->Stanga); PostOrdTrav(Rădăcina->Dreapta);
printf("%d\t",*(int*)Rădăcina->Info); } }
void Cauta (NodPtrT Radacina,int b) { int c; if (Rădăcina)
{ Cauta(Rădăcina->Stanga,b); c=*(int*)Rădăcina->Info; if(c==b) {t=Rădăcina;}
Cauta(Rădăcina->Dreapta,b); } }
void main(void) { int c; int i; int a[NR]; int n; i=0; c=0; n=NR; clrscr();
while (i<NR) { printf("Nod nou: "); scanf("%d",&a[i]); i++; }
ConstrArbore(root,n,a);
///*
printf("TRAVERSARE IN ORDINE — LDR\n"); InOrdTrav(*root);
//*/
/*
printf("TRAVERSARE IN PREORDINE — DLR\n"); PreOrdTrav(*root);
*/
/*
printf("TRAVERSARE IN POSTORDINE — LRD\n"); PostOrdTrav(*root);
*/
printf("Nodul căutat: "); scanf("%d",&c); t=NULL; Cauta(*root,c);
if(t!=NULL) {printf("%#x\t",t); printf("%d\n",*(int*)t->Info);}
else printf("Nod inexistent"); getch();
}
```

2.7.4. <u>Exemplul nr. 4. Analizați programul și modificați-l, utilizând numai pointeri.</u>

```
#include <stdio.h>
#include <conio.h>
#define max 10
#define infinit 1000
int muchie[max+1][3];          int m,n,i,cost=0;  int arb1,arb2;     int TATA[max+1];
    int APART(int v)  { while (TATA[v]>0) v=TATA[v];          return v;          }
void REUN(int a, int b)  {  int suma=TATA[a]+TATA[b];
if (TATA[a]<TATA[b])  { TATA[a]=suma; TATA[b]=a; }
else { TATA[b]=suma; TATA[a]=b; }   }
    void main()  {  clrscr(); printf("Algoritmul lui KRUSKAL\n");
        printf("\nIntroduceti numărul de noduri: ");  scanf("%d",&n);
        printf("\nIntroduceti numărul de muchii: ");  scanf("%d",&m);
        printf("\nIntroduceti muchiile in ordinea crescătoare a costurilor\n");
        for (i=1; i<=m; i++) { printf("\nMuchia nr. %d:\n",i); printf("Prima extremitate = ");
         scanf("%d",&muchie[i][1]);  printf("A doua extremitate = ");  scanf("%d",&muchie[i][2]);
        printf("Costul muchiei = "); scanf("%d",&muchie[i][0]);  }
        for (i=1; i<=n; i++) TATA[i]=-1;  TATA[1]=0;
        printf("\n\nArborele de cost minim este format din muchiile:\n");
        for (i=1; i<=m; i++) { arb1=APART(muchie[i][1]);  arb2=APART(muchie[i][2]);
        if (arb1!=arb2)  { REUN(arb1, arb2);  cost+=muchie[i][0];
        printf("%d-%d\n",muchie[i][1],muchie[i][2]);  }  }  printf("\nCost arbore = %d\n",cost);  getch();  }
```

3. MASIVELE – STRUCTURI DE DATE OMOGENE ȘI CONTIGUE

3.1. Masive unidimensionale (vectori). Sunt puține programele în care nu apar definite masive unidimensionale. Problemele de ordonare a șirurilor, de calcul a indicatorilor statistici medie și dispersie, programele pentru găsirea elementului minim și multe altele presupun stocarea valorilor numerice ale șirurilor în zona de memorie care în mod folcloric le numim vectori. Ceea ce de fapt se recunoaște sub numele de vector este în realitate o structură de date omogene.

Prin definirea: int x[10]; se specifică:

x – este o dată compusă din 10 elemente;
elementul – este de tip întreg;
primul element al structurii este x [0]
al doilea element este x [1]
. . . . . .
al zecelea element al structurii este x [9].

$$\lg(x..) = \sum_{i=1}^{n} \lg \quad (x[i]\,;\text{int}).$$

Dacă funcția: lg(a;b) se definește pentru tipurile standard prin

lg (x;b) = lg(.;b) = k

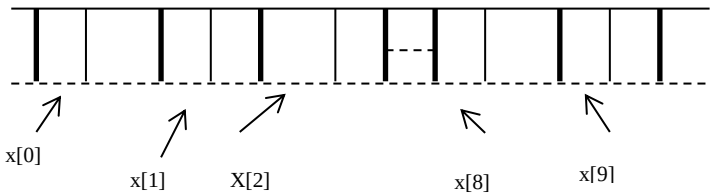
unde, k – e lungimea standard alocată la compilare pentru o dată de tipul b și

$\lg(x_0, x_1, x_2 \ldots .x_n;b) = \lg(x_0;b) + \lg(x_1;b) + \ldots + \lg(xn;b) = \lg(.;b) + \lg(.;b) + \ldots + \lg(.;b) = k_b + k_b + \ldots k_b = n^* k_b$

rezultă că:

lg(x,int) = 10*2 bytes

Modul grafic de reprezentare a alocării memoriei pentru elementele vectorului x, conduce la:



Pentru că elementele sunt de același tip.

lg(x[0], .) = lg(x[1], .) = . . . = lg(x[9], .)

Dacă notăm:

α = adr(x[i])

β = adr (x[i+1])

$\alpha - \beta$ = lg (x[i],.)

Se observă că :

dist (x[i], x[i+1]) = lg(x[i], int) = 2

întrucât elementele ocupă o zonă de memorie contiguă.

Se definesc funcțiile:

succ() – care desemnează succesorul unui element într-un șir;

pred() – care desemnează predecesorul unui element într-un șir astfel:

succ(x[i]) = x [i+1]

pred(x[i]) = x [i-1]

Aceste funcții au prioritățile:

succ(pred(x[i])) = succ(x[i-1]) = x[i]

pred(succ(x[i])) = pred(x[i+1]) = x[i]

Deci, funcțiile succ() și pred() sunt una inversă celeilalte.

Pentru extremitățile vectorului

pred(x[0]) = ϕ

succ(x[9]) = ϕ

succ (ϕ) = pred (ϕ) = ϕ

Direct, se observă că:

$succ^0$ (x[i]) = x [i]

$pred^0$ (x[i]) = x [i]

$succ^m$ (x[i]) = x[i+m]

$pred^n$ (x [i]) = x[i-n]

$succ^m$ ($pred^n$ (x[i])) = x[i-n+m]

$pred^n$ ($succ^m$ (x[i])) = x[i+m –n]

Dacă elementul x[i] este considerat reper (baza), adresele celorlalte elemente se calculează folosind deplasarea față de elementul reper.

Funcția deplasare: depl(a;b) va permite calculul deplasării lui a față de b.

Astfel.

depl(x[i+k],x[i]) = (i+k –i)*lg (.,int)=k*lg (.,int)

depl (x[i],x [i+k] = -k*lg(.,int)

sau depl(x[i+k],x[i] = adr(x[i+k]) – adr(x[i]).

Dacă x reprezintă structura de date omogene în totalitatea ei, iar x[0], x[1], ... ,x[9] sunt părțile care o compun, din reprezentarea grafică rezultă că:

adr(x) = adr(x[1]).

Se definește: adr(a+b) = adr(a) + (b –1)*lg(.,int),

unde:

a este numele datei structurate;

b poziția elementului a cărui adresă se dorește a fi calculată;

adr(a+b) = adr(b+a)

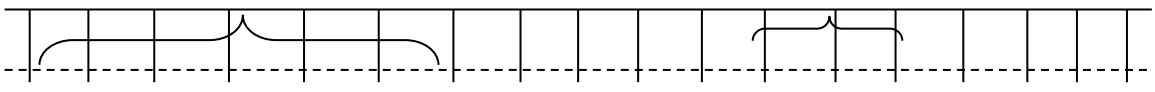
Din calcule de adrese rezultă că:

adr(x[n]) – adr(x[1])

. . . . . + 1 = n

lg(x[1])

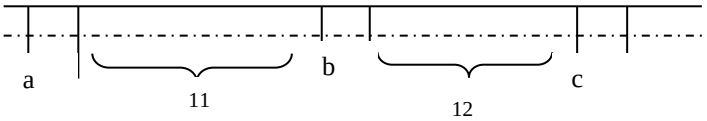
Contiguitatea este pusă în evidență prin definirea funcției de distanță DST(a,b), care indică numărul de bytes neaparținători ai datelor a și b care separă data a de data b.



(a,b) = 5

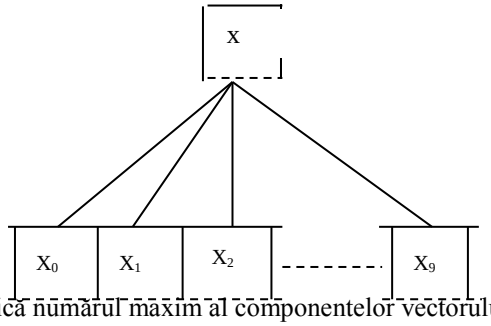
În raport cu modul de definire a funcției DST ( ), observăm că

DST(x[i] .x[i+1]) = 0 i є {0,2,. . . . . 9}
DST(x[i] .x[i]) = 0
DST(a,a) = 0
DST (a,b) > = 0



DST (a,b) ‹ ` ,c) + DST (c,b)
11 <= 11 + 12 + lg (b) + 12
lg(b) + 2*12 >= 0
DST (a,b) = DST (b,a)

Reprezentat ca structură arborescentă, vectorul are modelul grafic:



În programe, se specifică numărul maxim al componentelor vectorului, avându-se grija ca în problemele ce se rezolvă, numărul efectiv să nu depășească dimensiunea declarată.

De exemplu, în secvența:

```
. . . . . .
  int x[10];
  int n;
. . . . . .
cin<<n;
for( i= 0; i<n ;i++)
        x[i] = 0;
```

identificăm următoarele inexactități:
pentru faptul că n nu rezultă a fi inițializată cu o valoare cuprinsă între 0 și 9, există posibilitatea ca în cazul n=15, adresa calculată să fie adr(x+15) = adr(x) + (15-1)*lg(x,int)
și să cuprindă unui cuvânt ce urmează cu mult mai departe de componenta x[9], cuvânt al cărui conținut devine zero.

Prin parametrii compilării, se controlează expresiile indiciale, așa fel încât să fie incluse în intervalul definit pentru variație, specificat în calificatorul tip[e_1. .e_2], evitându-se distrugerea necontrolată a operanzilor adiacenți masivului unidimensional.

Exemplul dat arată că expresia indicială, aparține intervalului [0,9] N, însă limbajul C/C++ permite definirea unor limite într-o formă mult mai flexibilă.

De exemplu, definirea:

```
int y[11];
```

permite referirea elementelor:

……,y [-7], y[-6], ... y[0], y[1], y[2], y[3]
adr(y) = adr(y[-7])
DST(y[-7], y[-6]) = [-6-(7)-1]*lg(. . int) = 0

și acest vector rezultă că este contiguu.

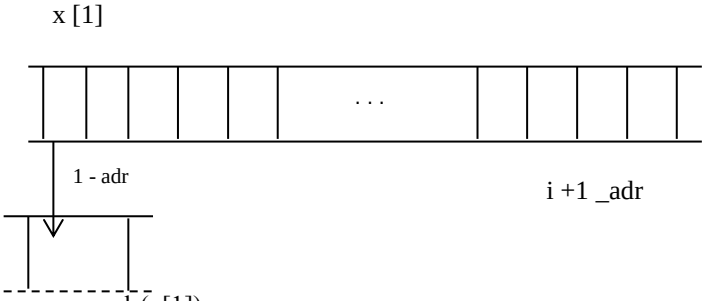
Numărul de componente este dat de:

$$\frac{adr(y[3]) - adr(y[-7])}{\lg(..\text{int } eger)} + 1 = 11$$

adr(y+4) = adr(y[-7]) +[4-(-7)]*lg(y, int) = = adr(y[-7]) + 11*lg(y, int)
succ(y[-5]) = = y [-4]
pred(y[-1]) = = y[-2]
depl(y[-7+2], y[-7]) = = 2*lg(y, int)
Deplasarea lui y[-6] față de elementul y[2]:
depl(y[-6], y[2]) = = depl(y[2-8], y[2]) = (2-8-2)*lg(y, int) = = -8*lg(y, int)
depl(y[2], y[-6]) = = [2-(-6)]*lg (y, int) = = 8*lg(y, int)

Se observă că: depl(a,b) = = - depl (b,a)
Sau depl (a, b) + depl(b,a) = = 0

Proprietățile masivelor unidimensionale, conduc la ideea stocării într-o zonă de memorie a adresei unuia dintre termeni și prin adăugarea sau scăderea unei rații egale cu lungimea unui element, se procedează la baleierea spre dreapta sau spre stânga a celorlalte elemente.



α : = adr(x[1])
α : = α + i*lg(x[1], int)
conduce la situația în care cont (α) ≡ adr (x[i+1])
Datorită contiguității memoriei și a regulilor de regăsire a elementelor, masivul unidimensional nu conține în mod distinct informații privind poziția elementelor sale. Cunoscând numele masivului, referirea elementelor se realizează specificând acest nume și poziția elementului cuprins între paranteze drepte sub forma unei expresii indiciale.

Expresia indicială are un tip oarecare, însă înainte de efectuarea calculului de adresa are loc conversia spre întreg, care este atribuită funcției int( ). Deci funcția int ( ), *realizează rotunjirea în minus a valorii expresiei indiciale.
Astfel, pentru definirea: int z[100]; unde indicele pleacă de la 6,

adresa elementului z [e^a + cos (b) / $\sqrt{c}$]
se calculează astfel :

adr (z [e^a + cos (b) / $\sqrt{c}$]) = adr (z[6]) + (int (e^a + cos (b) / $\sqrt{c}$)) *
* lg (z[6] , int)

Funcțiile de validare a adreselor se vor defini astfel :

$$fp(a) = \begin{cases} \text{TRUE, dacă adr(a)} \in [A_i, A_f] \\ \text{FALSE, în caz contrar} \end{cases}$$

$$gm(a) = \begin{cases} \text{TRUE, dacă a} \in [B_i, B_f] \\ \text{FALSE, în caz contrar} \end{cases}$$

unde:
fp( ) este funcția de validare adrese în raport cu programul ;
bm( ) este funcția de validare a adreselor în raport cu un masiv m;
A_i este adresa se început a programului ;
A_f este adresa se sfârșit a programului ;
B_i este a dresa de început a masivului m ;
B_f este adresa de sfârșit a masivului m.

Dacă programul P este format din instrucțiunile I_1, I_2, …, I_{100} atunci :
adr (I_1) = A_i
adr (I_{100}) = A_f.

Dacă în programul P este definit masivul : int x[10];
atunci :
adr (x[0]) = B_i
adr (x[99]) = B_f.

Vom spune că este corect folosită expresia indicială e^a + cos (b) / $\sqrt{c}$ dacă :

g(x [e^a + cos (b) / $\sqrt{c}$]) = TRUE
sau dacă :

int (e^a + cos (b) / $\sqrt{c}$) * (lg (z[6] , int) < B_f - B_i

Dacă se ia în considerare că: f(x [e^a + cos (b) / $\sqrt{c}$]) = TRUE pot apare situații în care să fie modificate alte zone decât cele asociate masivului unidimensional.
Dacă se iau în considerare reprezentări simplificate, pentru localizarea unui element x[i] al unui masiv unidimensional, se folosește formula cunoscută : x[0] + x[i – 0] * lungime_element

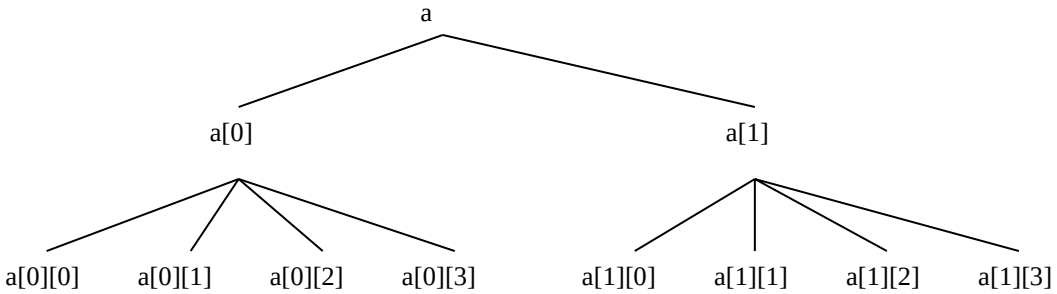
Atunci când se construiesc programe în care se definesc masive unidimensionale, alegerea tipului, alegerea limitei inferioare și a limitei superioare pentru variație, depind de contextul problemei dar și de formulele identificate prin inducție matematică pentru explorare, folosind structuri repetitive.
De fiecare dată, trebuie avută grijă ca numărul de componente ce rezultă la definire să fie acoperitor pentru problemele ce se rezolvă.
Pentru stabilirea numărului maxim de componente ale vectorilor ce sunt definiți, se consideră:
L – lungimea în baiți a disponibilului de memorie;
L_p – lungimea în baiți a necesarului de memorie pentru program (instrucțiuni executabile și alte definiri);
M – numărul de masive unidimensionale de tip T_i, având același număr de componente care apar în program;
x – numărul maxim de componente ale unui masiv.

$x = int((L – L_p) / (N * lg(. ;T_i)))$.

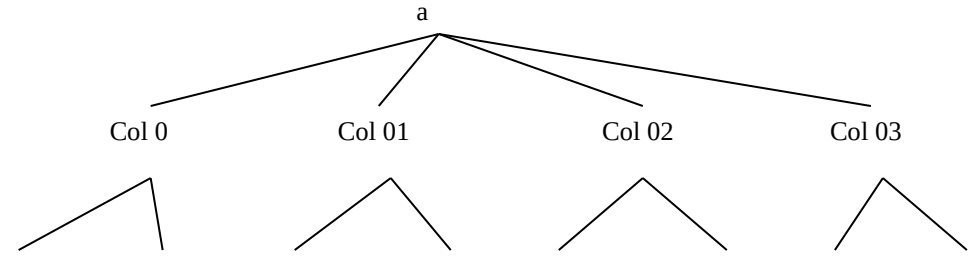
În cazul în care cele N masive au dimensiuni variabile $d_1, d_2, \ldots, d_N$, condiția de utilizare corecta a resursei memoriei este ca :

$d_1 + d_2 + \ldots + d_N \leq int((L – L_p) / lg(. ;T_i))$.

3.2 Masive bidimensionale (matrice). Este aproape sigur că în activitatea de programare, matriceale ocupă ca importanță un loc deosebit. Și tot atât este de adevărat că lucrul corect cu matrice oferă rezultate rapide, cum la fel de adevărat este că utilizarea defectuoasă reduce considerabil șansa de a obține rezultate corecte.
Singura modalitate de a realiza programe corecte utilizând matrice, este cunoașterea mecanismelor de implementare a acestora în diferite limbaje de programare, precum și studierea proprietăților ce decurg din ele.
Pornind de la ideea că o matrice este formată din linii, iar liniile sunt formate din elemente, se ajunge la modelul grafic al matricei, care apare sub forma unei arborescențe organizată pe trei nivele.
Pentru definirea : int a[2][4];
rezultă că matricea a este formată din 2 linii; fiecare linie având câte 4 componente.
Modelul grafic pentru matricea a este :



Modelul grafic presupune existența a trei nivele:
la nivelul cel mai înalt se află întregul, matricea a;
la nivelul imediat următor se află primele părți în care se descompun matricea și anume: liniile acesteia a[0] și a[1];
la nivelul al treilea se află elementele grupate pentru fiecare linie în parte.
O altă definiție a matricei conduce la descompunerea în coloane și a coloanelor în elemente.



reprezentând dispunerea contiguă în aml iilor ur azuri se ontinua structu leilalte, ɔresce ctiv, a lemente anelor un baza ontinu escenț celeilal spectiv, a coloanelor una în continuarea celeilalte, regăsind ambele situații ca modalități de „liniarizare a matricelor”.

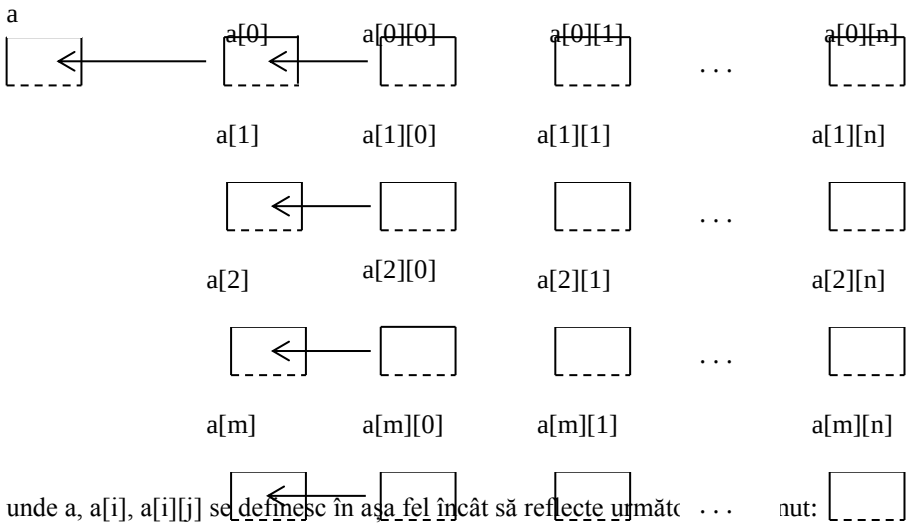
Dacă se consideră o matrice A, având m linii și n coloane, elementele fiind de tipul T_i, adresa elementului a[i][j] se calculează fie după formula :
adr(a[i][j]) = adr(a[0][0]) + ((i – 0) * n + j) * lg(a , T_i) ,
dacă liniarizarea se face „linie cu linie”, fie după formula :
adr(a[i][j]) = adr(a[0][0]) + ((j – 0) * m + i) * lg(a , T_i) ,
dacă liniarizarea se face „coloană după coloană”.
Intuitiv, o matrice poate fi privită luând în considerare numai primul nivel de descompunere, ca vector de linii sau ca vector de coloane.

Fiecare linie sau coloană, prin descompunerea la nivelul următor, este privită ca vector de elemente. Deci, în final o matrice poate fi privită ca vector de vectori.

Definirea :
int a[2][4];
pune în evidență exact acest lucru.
Din această descriere rezultă că :
adr(a) = adr(a[0]) = adr(a[0][0])
adr(a[i]) = adr(a[i][0])
Dacă dispunerea elementelor este linie cu linie,
DST(a[i][n] , a[i+1][0]) = 0.
Dacă dispunerea elementelor este coloană după coloană,
DST(a[m][j] , a[0][j+1]) = 0
Deplasarea elementelor a_{ij} față de elementul a_{kh} , se calculează după relația :
depl (a[i][j] . a[k][h]) = adr(a[k][h]) – adr(a[i][j]) = adr (a[0][0]) + ((k-0)n + h) * *lg(a, T_i) – (adr(a[0][i])+
+ ((i-0)n + j) *lg(a, T_i) = lg(a, T_i)*[(k-0)n – (i-0)n + h – j] = lg(a, T_i)*[(k-i)n + h – j]
Numărul de elemente al matricei n se obține

$$\frac{adr(a[m][n]) - adr(a[0][0])}{lg(a, T_i)} + 1 = m * n$$

În cazul dispunerii linie de linie,
succ(a[i][n]) = a[i+1][0]
pred(a[i][0]) = a [i-1][n]
succ (a[i]) = a[i+1][0]
pred (a[i]) = a[i-1][0]
succ(a[i+k]) = succ (a[i])
pred (a[i]) = pred (a[i+k])
În mod asemănător,
adr(a+i) = adr(a[0]) + (i-0)*lg(a[0])
adr (a[i] +j) = adr(a[i]) + (j-0)* lg (a[0][0])
lg(a[i]) = lg(a[i][0]) + . . .+lg(a[i][n]) sau lg(a[i]) = n*lg(a[i][0])
deci: adr(a+i) = adr(a[0]) + (i-0)*n*lg(a[i][0])
Dacă privim matricele ca vectori de vectori, în mod corespunzător se identifică forma grafică:



unde a, a[i], a[i][j] se definesc în așa fel încât să reflecte următo . . . nut:
a[i] = adr(a[i][1])
i = 0,2, ...m
a = adr (a[0]) .
Definirea unei matrice ca având m linii și n coloane, deci în totalul m*n elemente și explorarea elementelor prin includerea unui alt număr de linii și coloane conduce la o localizare a elementelor însoțită de un rezultat eronat, a cărui proveniență e necesar a fi interpretată și reconstituită.
Se consideră matricea:

$$\begin{Vmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 7 & 1 & 1 & 4 & 2 \end{Vmatrix}$$

stocată în:

int a[5][5];

definită și inițializată în programul principal, iar într-o procedură se consideră că avem definirea:

int b[4][4]:

Procedura realizează însumarea elementelor diagonalelor matricei b. Întrucât transmiterea parametrilor se face prin adresă, punerea în corespondență este:

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{20}	a_{21}
1	2	3	4	5	6	7	8	9	10	11	12
b_{00}	b_{01}	b_{02}	b_{03}	b_{10}	b_{11}	b_{12}	b_{13}	b_{20}	b_{21}	b_{22}	b_{23}
a_{22}	a_{23}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}
13	14	15	16	17	18	19	20	7	1	1	4	2
b_{30}	b_{31}	b_{32}	b_{33}								

Prin specificații se cere $Sa = \sum_{i=0}^{4} a[i,1].$ iar prin procedură se obține:

$$Sb = \sum_{i=0}^{3} b[i,1]$$

În loc de $S_a = 42$ se obține $S_b = 34$.

În cazul în care structura repetitivă a procedurii efectuează corect numărul de repetări.

for (i = 0; i < 5; i + +)

$S_1 = S + b[i][1];$

i = 0 localizează termenul:

adr (b[0][0]) = adr(a[0][0])

cont (b[0][0]) = cont (a[0][0]) = 1

i = 1

adr(b[1][1]) = adr (b[1][1]) + (1-0)*4* lg (int) + 1*lg (int) = adr (b[0][0]) +6*lg (int)

cont (b[1][1]) = 6

i = 2

adr (b[2][2]) = adr (b [0][0]) + (2-0) * 4* lg (int) + 2*lg (int) = adr (b[0][0]) + 11*lg (int)

cont (b[2][2]) = 11

i = 3

adr (b[3][3]) = adr (b [0][0]) + 16*lg (int)

cont (b [3][3]) = 16

i = 4

adr (b[4]) = adr (b[0][0} + (4-0) 4* lg*(int) + 4*lg (int) = adr (b[0][0]) + 21* lg (int)

cont (b [4][4]) = 7

Valoarea obținută S = 1 + 6 + 11 + 16 + 7 = 41

Cunoașterea modului de adresare și a modului de transmitere a parametrilor, permite interpretarea cu rigurozitate a oricărui rezultat.

Memorarea unei matrice numai ca formă liniarizată, neînsoțită de informația dedusă din structura arborescentă, implică efectuarea calculului de adresa ori de câte ori este accesat un element al matricei.

În cazul în care nodului rădăcină a, i se rezervă o zonă de memorie, iar nodurilor a [0], a[1], . . . ,a[m] li se asociază, de asemenea, zona ce se inițializează adecvat, regăsirea elementelor se efectuează folosind calcule de deplasări.

De la relația:

adr (a[i][j] = adr (a[0][0]) + [(i – 0)*n + j]*lg (int)

se ajunge la: adr (a [i][j]) = adr (a [i]) + depl(a [i][j], a [i][0])

În cazul în care matricea este densă sau încărcată sau are puține elemente nule sau este “full”, dar are elemente ce conțin valori particulare, se efectuează o tratare diferențiată.

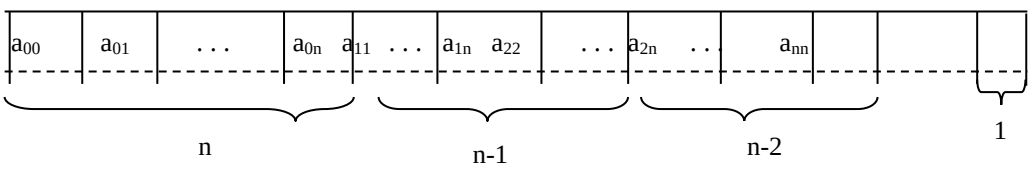
Matricele simetrice, au proprietatea:

cont (a [i][j]) = cont (a [j][i]).

ceea ce conduce la ideea memorării numai a elementelor de pe diagonala principală și de sub aceasta sau de deasupra ei.

Pentru o matrice simetrică, a[n][n] cu n*n elemente., impune memorarea a n* (n+1) /2 dintre acestea.

Dacă sunt memorate elementele a[i][j], j >= i, i = 0,1,2. . . . n, j = i, i +1, . . .n, astfel:



adr(a [i][j]) = adr ([0][0]) + [(n-0) + (n-1) + . . . (n-i+1) + (j-0)]*lg (int)
adr (a [i][j]) = adr (a [0][0]) + f (i, j, n) * lg (int)
f (i, j, n) reprezintă funcția de calcul a deplasării față de elementul a [i][0] a elementului a [i][j] : funcția f (i, j, n) se deduce prin inducție.

Matricea cu linii sau coloane identice, sau cu elemente identice, permit o astfel de memorare care conduce la o economisire a spațiului de memorie.

Astfel, pentru matricea:

$$\begin{Vmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{Vmatrix}$$

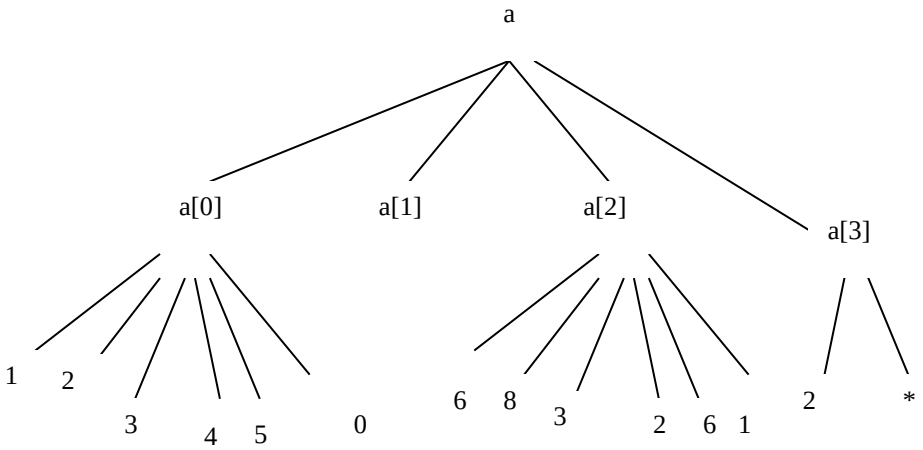
se impun definirile:

- nume matrice
- număr linii
- număr coloane
- valoarea elementului repetat (1)

În cazul în care matricea are conținutul:

$$\begin{Vmatrix} 1 & 2 & 3 & 4 & 5 & 0 \\ 1 & 2 & 3 & 4 & 5 & 0 \\ 6 & 8 & 3 & 2 & 6 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \end{Vmatrix}$$

aceasta are reprezentarea:



Matricele diagonale sunt matrıcele în care elementele componente apar sub forma:

$$A = \begin{Vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{Vmatrix}$$

În acest caz, forma de memorie adecvată, în condițiile în care în urma prelucrării matricei A, aceasta nu-și schimbă structura, este vectorul.

Un alt tip matrice, cu modalități specifice de prelucrare, este matricea tridiagonală, care are forma:

$$T = \begin{Vmatrix} 1 & 4 & 0 & 0 & 0 & 0 \\ 2 & 8 & 1 & 0 & 0 & 0 \\ 0 & 2 & 7 & 3 & 0 & 0 \\ 0 & 0 & 6 & 5 & 9 & 0 \\ 0 & 0 & 0 & 0 & 8 & 1 \end{Vmatrix}$$

Elementele nenule se memorează într-un masiv unidimensional, iar accesarea lor se efectuează după formula:

adr (t [i][j]) = adr (t [0][0]) + [2* (i –0) + j – 0]* lg (int)
unde, i = 0, 1. . . .n și j =0,1 pentru i =1

j = i – 1, i, i + 1 pentru 1 < i < n

j = n – 1, n pentru i = n

Matricea triunghiulară are elementele nenule fie deasupra, fie sub diagonala principală, iar stocarea în memorie este asemănătoare matricei simetrice.

Pentru matricea triunghiulară A cu elementele nenule sub diagonala principală, adresa elementului a[i,j] este:

adr (a [i][j]) = adr (a [0][0]) + [i* (i –0) /2 + (j –0)] * lg (int)

Construirea masivului unidimensional se realizează în secvența:

```
k = 0;
for ( i = 0 ; i < n ; i + + )
        for ( j = i ; j < n ; j + + )     {          k = k + 1;          s [k] = T [i][j];     };
```

dacă elementele nenule sunt deasupra diagonalei principale sau în secvența:

```
k = -n* (n+1) /2;
 for ( i = n-1 ; i >= 0 ; i-- )
        for ( j = i ; j >= 0 ; j-- ) {    s [k] = T [i,j];      k = k – 1; };
```

dacă elementele nenule sunt sub diagonala principală.

În toate cazurile, se urmărește atât reducerea lungimii zonei de memorie ocupată, cât și creșterea vitezei de acces la componente. Pentru aceasta, mai întâi matricele sunt memorate în extenso, ca mai apoi după analiza componentelor să se deducă dacă matricele sunt simetrice sau dacă au linii sau coloane constante, cazuri în care se fac alocări adecvate.

Operațiile cu matrice se efectuează cu algoritmi care iau în considerare forma efectivă în care s-a făcut memorarea, existând posibilitatea trecerii rezultatului la un alt tip de stocare în memorie.

De exemplu, dacă A este o matrice triunghiulară cu elemente nenule sub diagonala principală și B este o matrice triunghiulară cu aceeași dimensiune ca matricea A, dar cu elementele nenule deasupra diagonalei principale, în cazul adunării celor două matrice, rezultă o matrice cu toate elementele nenule., ce este stocată după regulile acestui tip.

4.1 MATRICE RARE. Pentru a deduce dacă o matrice este sau nu rară, se definește gradul de umplere al unei matrice:

$$G = \frac{k}{m * n} * \quad 100$$

unde:

k – numărul elementelor nenule;
m -numărul de linii al matricei;
n – numărul de coloane al matricei.

În cazul în care k < 0.3*m*n, se consideră că matricea este rară.

Problema matricelor rare comportă două abordări:

- abordarea statică, în care alocarea memoriei se efectuează în faza de compilare, ceea ce presupune ca programatorul să cunoască cu o precizie, bună numărul maxim al elementelor nenule:
- abordarea dinamică, în care alocarea se efectuează în timpul execuției, caz în care nu este necesară informația asupra numărului de elemente nenule; această abordare este dezvoltată în partea destinată listelor.

Memorarea elementelor matricei rare, presupune memorarea indicelui liniei, a indicelui coloanei și, respectiv, valoarea nenulă a elementului. Se consideră matricea:

$$A = \begin{Vmatrix} 0 & 0 & 6 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 2 & 0 & 0 \end{Vmatrix}$$

Gradul de umplere al matricei A cu numărul de linii m=4, numărul de coloane n = 5 și numărul elementelor nenule k = 5 este:

$$G = \frac{5}{5 * 4} = 0.25$$

Se definesc 3 vectori astfel:

lin [ ] – memorează poziția liniei ce conține elemente nenule;
col [ ] - memorează poziția coloanei ce conține elemente nenule;
val [ ] – memorează valoarea nenulă a elementelor.

Vectorii se inițializează cu valorile:

LIN	COL	VAL
1	3	6
2	1	7
3	4	9
4	2	8
4	3	2

Pentru efectuarea calculelor cu matrice rare definite în acest fel, un rol important îl au vectorii LIN, COL, iar pentru matricele rare rezultat, trebuie definiți vectori cu număr de componente care să asigure și stocarea noilor elemente ce apar.

Astfel, pentru adunarea matricelor rare definite prin:

LIN_a	COL_ a	VAL_a
1	1	-4
2	2	7
4	4	8

și

LIN_b	COL_b	VAL_b
1	1	4
2	2	-7
3	2	8
4	1	5
4	3	6

rezultatul final se stochează în vectorii:

LIN_C	COL_C	VAL_C
1	1	0
2	2	0
3	2	8
4	1	5
4	3	6
4	4	8
?	?	?
?	?	?

Vectorii LIN_C, COL_C și VAL_C au un număr de componente definite, egal cu:

DIM (LIN_a) + DIM (LIN_b)

unde, DIM( ) este funcția de extragere a dimensiunii unui masiv unidimensional.

DIM : J -> N

Astfel, dacă:

```
int a[n-m];
DIM (a) = n-m+1
```

Cum aici este abordată problematica matricelor rare, în mod natural, se produce glisarea elementelor nenule, obținându-se în final:

LIN_C	COL_C	VAL_C
3	2	8
4	1	5
4	3	6
4	4	8
?	?	?
?	?	?
?	?	?
?	?	?
?	?	?

Prin secvențe de program adecvate, se face diferența între definirea unui masiv bidimensional și componentele inițializate ale acestora, cu care se operează pentru rezolvarea unei probleme concrete.

De exemplu, vectorii LIN_a și LIN_b au 3, respectiv 5 componente în utilizare, dar la definire au rezervate zone de memorie ce corespund pentru câte 10 elemente. Rezultă că vectorul LIN_C trebuie definit cu 20 componente încât să preia și cazul în care elementele celor două matrice rare au poziții disjuncte.

În cazul operațiilor de înmulțire sau inversare, este posibil ca matricele rezultat să nu mai îndeplinească cerința de matrice.

În acest scop, se efectuează calculele cu matrice rezultat complet definite și numai după efectuarea calculelor se analizează gradul de umplere și dacă acesta este redus, se trece la reprezentarea matricei complete ca matrice rară.

Funcțiile full() și rar(), au rolul de a efectua trecerea la matricea completă, respectiv la matricea rară.

Funcția full() conține secvența:

```
for( i = 0 ; i < n ; i++)
   a [LIN _a[i]][COL_a[i]] = val_a[i];
```

iar funcția rar() conține secvența:

```
k =1;
```

```
for( i = 0 ; i < m ; i++)
  for( j = 0 ; j < n ; j++)
    if a[i][j] != 0
          {
            LIN_a[k] = i;
            COL_a[k] = j;
            val_a[k] = a[I][j];
            k = k + i;
          };
```

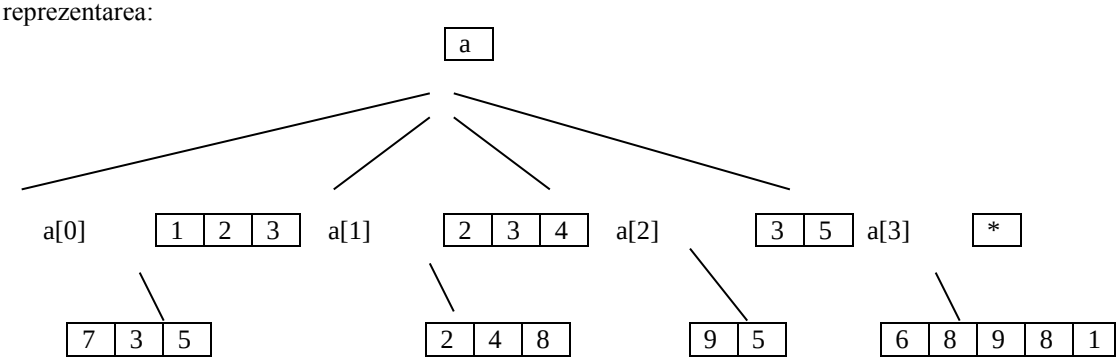
Funcțiile full() și rar() sunt una inversa celeilalte.

În cazul în care gradul de umplere nu este suficient de mic încât matricea să fie considerată rară, pentru memorare se utilizează o structură arborescentă care conține pe nivelul al doilea pozițiile elementelor nule, iar pe nivelul al treilea valorile.

Astfel matricei:

$$a = \begin{Vmatrix} 7 & 3 & 5 & 0 & 0 \\ 0 & 2 & 4 & 8 & 0 \\ 0 & 0 & 9 & 0 & 5 \\ 6 & 8 & 9 & 8 & 1 \end{Vmatrix}$$

îi corespunde reprezentarea:



Se elaborează convenții asupra modului de stabilire a lungimii vectorului de poziții, fie prin indicarea la început a numărului de componente inițializate, fie prin definirea unui simbol terminal.

De asemenea. în cazul considerat s-a adoptat convenția ca liniile complete să fie marcate cu '*', fără a mai specifica pozițiile elementelor nenule, care sunt de fapt termenii unei progresii aritmetice.

Liniarizarea masivelor bidimensionale conduce la ideea suprapunerii acestora peste vectori. Deci, punând în corespondență elementele unei matrice cu elementele unui vector, se pune problema transformării algoritmilor, în așa fel încât operând cu elementele vectorilor să se obțină rezultate corecte pentru calcule matriciale.

Astfel, considerând matricea:

$$a = \begin{Vmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{Vmatrix}$$

prin punerea în corespondență cu elementele vectorului b, să se obțină operând cu acesta din urmă, interschimbul între două coloane oarecare k si j ale matricei.

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{10}	a_{11}		a_{20}	a_{21}	a_{22}	a_{23}	a_{24}
1	2	3	4	5	6	7	...	11	12	13	14	15
b_0	b_1	b_2	b_3	b_4	b_5	b_6		b_{10}	b_{11}	b_{12}	b_{13}	b_{14}

Dacă matricea are M linii și N coloane

adr(a[i][j]) = adr(a[0][0]) + ((i-0)*N+j)*1g(int)

Din modul în care se efectuează punerea în corespondență a matricei a[ ] cu vectorul b[], rezultă:

adr(b[0]) = adr(a[0][0])

deci

adr(a[i][j]) = adr(b[0])+((i-0)*N+j)*lg(int) = adr(b[(i-0)*N+j])

Secvența de înlocuire a coloanelor:

```
for( i = 0 ; i < M ; i++)
  {
c = a[i][j];
a[i][j] = a[i][k];
```

```
a[i][k] = c;
        }
```
este înlocuită prin secvența:
```
        for( i = 0 ; i < M ; i++)
                {
                  c = b[(i-0)*N+j];
                  b [(i-0)*N+j] = b[(i-0)*N+k];
                  b[(i-0)*N+k] = c;
                }
```
Transformarea algoritmilor de lucru cu masive bidimensionale în algoritmi de lucru cu masive unidimensionale, este benefică, ne mai impunându-se cerința de transmitere ca parametru a dimensiunii efective a numărului de linii (dacă liniarizarea se face pe coloane), sau a numărului de coloane. (dacă liniarizarea se face pe linii).
În cazul matricelor rare, aceeași problemă revine la interschimbarea valorilor de pe coloana a treia dintre elementele corespondente ale coloanelor k și j cu posibilitatea inserării unor perechi și respectiv ștergerii altora.
Pentru generalizare, un masiv n-dimensional rar, este reprezentat prin n+1 vectori, fiecare permițând identificarea coordonatelor elementului nenul, iar ultimul stocând valoarea acestuia.
În cazul în care se construieşte o matrice booleana ce se asociază matricei rare, odată cu comprimarea acesteia, se dispun elementele nenule într-un vector. Punerea în corespondență a elementelor vectorului are loc odată cu decomprimarea matricei booleene și analizarea acesteia.
De exemplu, matricei :

$$A = \begin{Vmatrix} 8 & 0 & 0 & 0 & 4 & 0 \\ 3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 7 & 0 \\ 5 & 6 & 9 & 0 & 0 & 0 \end{Vmatrix}$$

se asociază matricea booleană:

$$B = \begin{Vmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{Vmatrix}$$

care prin compactare, ocupă primii 3 baiți ai unei descrieri, urmați de componentele vectorului:
C = (8, 4, 3, 3, 1, 7, 5, 6, 9).
Compactarea este procedeul care asociază fiecărei cifre din forma liniarizată a matricei B, un bit.

4.2. Masive multidimensionale Utilizarea masivelor multidimensionale este impusă de tendința unificării grupărilor de date omogene, după diferite criterii într-o singură mulțime, cu posibilitatea reconstituirii apartenenței elementelor la fiecare grupă.
Dacă într-o secție sunt 10 muncitori, iar întreprinderea are 6 secții și se memorează salariile lunare ale acestora pentru 7 ani consecutivi, intuim stocarea volumului impresionant de salarii folosind un masiv cu 4 dimensiuni, definit prin:

```
int salarii [10][6][12][7];
```

iar elementul
salarii [i][j][k][h] identifică salariul muncitorului i, din secția k, obținut în luna k a anului h.
Obținerea unui nivel mediu
salariu[.][j][k][h]
salariu[.][.][k][h]
salariu[.] [.] [.] [h]
salariu [.][.][.][.]
sau a oricărei variante de grad de cuprindere a elementelor colectivității, este o problemă de:
- definire a masivelor multidimensionale, cu număr de dimensiuni mai redus cu o unitate, cu două unități, cu trei și în final cu patru și apoi, inițializarea lor;
- efectuarea calculelor în cadrul unei secvențe de structuri repetitive incluse.

De exemplu, pentru calculele mediilor specificate mai sus se definesc variabilele:

```
float mjkh_salariu[6][12][7];
float mmkh_salariu[12][7];
float mmmh_salariu[7];
float mmmm_salariu;
```

Variabilele de control sunt definite pe domeniile:
$i \in [0,9] \cap N$ $k \in [0,11] \cap N$
$j \in [0, 5] \cap N$ $h \in [0, 6] \cap N$

Problema inițializării se simplifică atunci când masivele multidimensionale se pun în corespondență cu un masiv unidimensional, acesta din urmă fiind inițializat în cadrul unei singure structuri repetitive, în care variabila de control are ca domeniu :

$[1, 10*6*12*7] \cap N$

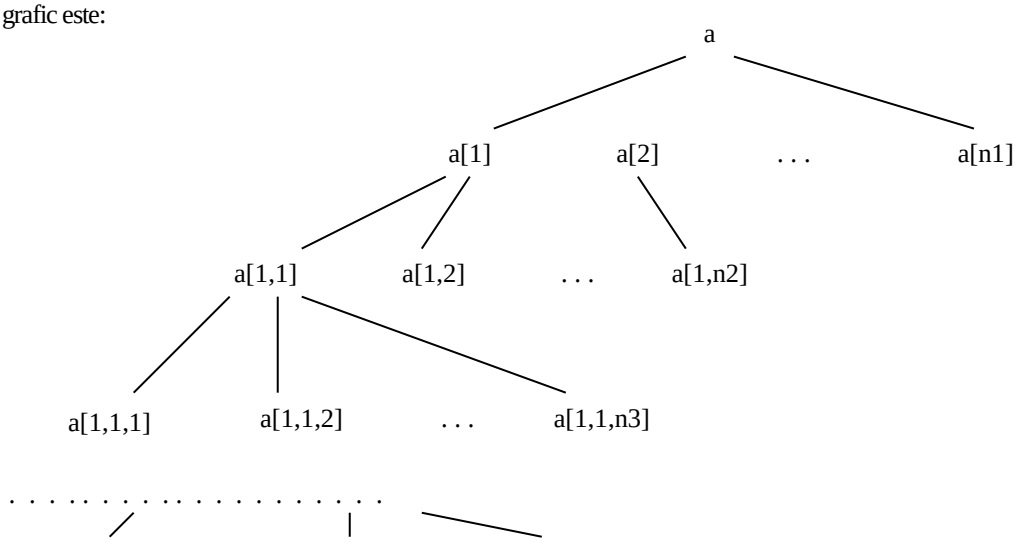
Modelul grafic asociat masivului multidimensional rămâne în continuare structura arborescentă, cu deosebirea că numărul de nivele este mai mare și depinde de numărul de dimensiuni.

Dacă la variabila unidimensională numărul nivelelor este 2, la variabila bidimensională numărul nivelelor este 3, se deduce că la o variabila p_dimensională, numărul nivelelor este p+1.

Dacă se consideră masivul a, p_dimensional, având pentru fiecare dimensiune definirea:

A[n1][n2][…][np],

modelul grafic este:



Localizarea unui element oarecare din masivul p_dimensional de tipul T_i se efectuează după formula:

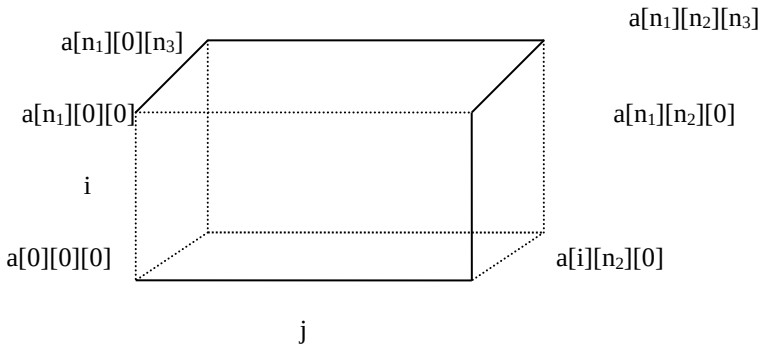
$adr(a[i_1][i_2][\ \ \ ][i_p]) = adr(a[l][l][\ \ \ ][0]) +$

$+ f(i_1,i_2,...,i_p; n_1,n_2,....,n_p)*lg(T_i)$

Spre exemplificare, pentru p = 3 se obține:

$adr(a[i][j][k]) = adr(a[0][0][0]) + ((i-0)*n_2*n_3 + (j-0)*n_2+k)*1g(int)$

Dacă reprezentarea în memorie corespunde paralelipipedului:



Și masivele multidimensionale sunt tratate ca matrice rare, numărul vectorilor "informaționali" care conțin indicii pentru corecta localizare a valorilor nenule, sunt în număr egal cu numărul dimensiunilor masivului de bază. Asocierea arborescenței și a posibilității de adresare, respectă aceleași reguli ca în cazul masivului bidimensional.

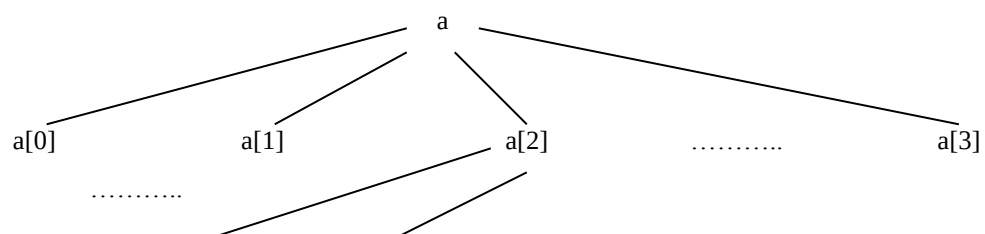
Când se fac stocări de informații referitoare la un grup de elemente, este corectă abordarea în care se specifică o serie de informații precum:

poziția în memorie a primului element;
poziția în memorie a ultimului element;
numărul de elemente sau poziția elementelor în substructura reală a masivului.

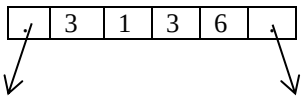
De exemplu, dacă într-un masiv tridimensional, în secțiunea corespunzătoare lui i = 3, avem pe linia a 2-a elementele:

(l 0 2 0 0 4)

asociem arborescența :



Variabila a[2][1] are o structură convenabil definită, care să poată memora adresele primei şi respectiv ultimei componente de pe nivelul imediat inferior, precum şi poziţiile elementelor nenule din linie.
Este convenabilă şi definirea:



în care, prima informaţie de după adresa primei componente indică numărul componentelor memorate la nivelul inferior.
Este de dorit ca multitudinea de informaţii suplimentare care vin să descrie structura, fie să menţină un grad de ocupare a memoriei cât mai redus, fie să permită un acces mult mai rapid la informaţiile dintr-un masiv multidimensional.
Modul de a pune problema masivelor ca structuri de date omogene, prin folosirea de exemple, pentru programatori nu reprezintă dificultate în înlocuirea valorilor particulare cu parametrii care conduc la formule generale pentru fiecare clasă de probleme în parte.

4.3. Ştergerea "naivă". *Eliminarea unei frunze dintr-un arbore este o problemă trivială şi se rezumă la ştergerea efectivă a frunzei din arbore. Pentru un nod intern, abordarea cea mai simplă, dar şi cea mai ineficientă este următoarea: Fie un nod n cu o cheie k ce se doreşte a fi ştearsă din arbore. Eliminarea poate fi reprezentată grafic în felul următor (Figura 1):*

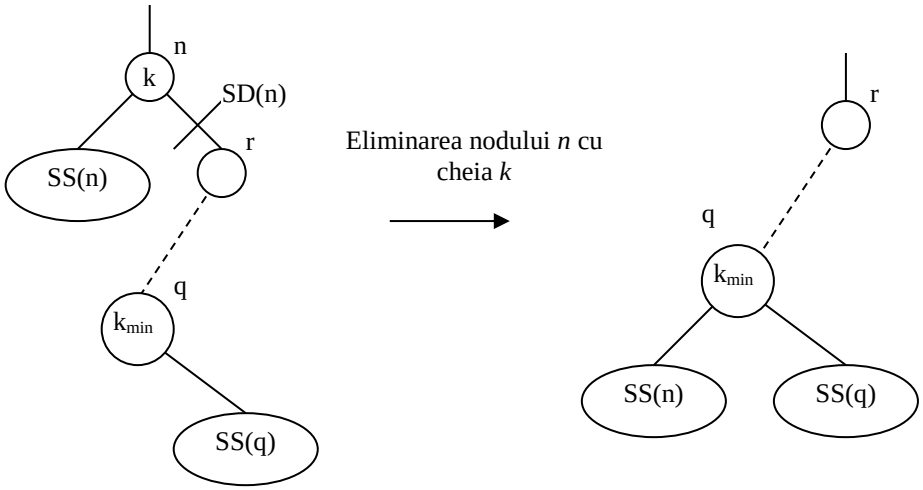


Figura 1 Eliminarea "naiva" stânga dintr-un arbore binar de căutare

Să considerăm exemplul prezentat în Figura 2.

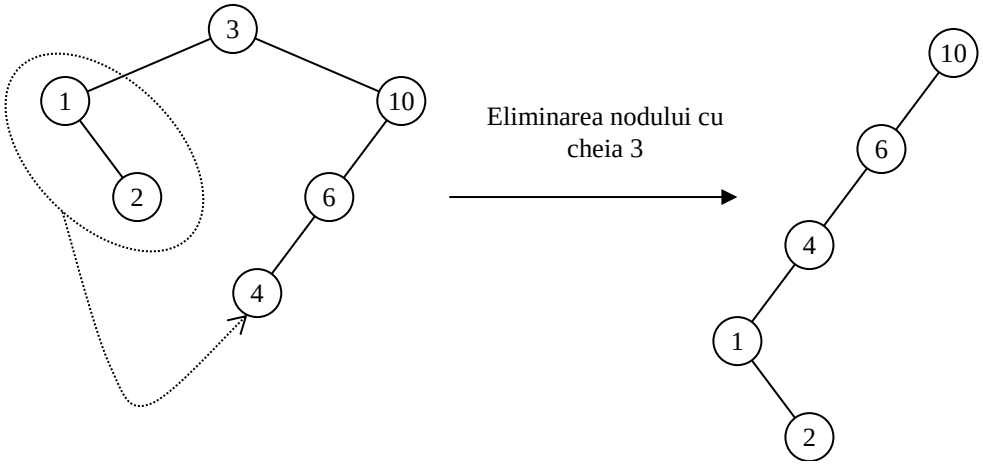


Figura 2 Eliminarea nodului cu cheia 3 duce la degenerarea arborelui

Se constată că oricât de echilibrat ar fi un arbore, la ştergerea unui nod înălţimea unui subarbore va creşte, iar după o serie de eliminări succesive el va degenera într-o listă.
De remarcat că se poate aplica şi metoda în oglindă (simetrică), numită *eliminarea "naivă" dreapta* dintr-un arbore binar de căutare.

Eliminarea Hibbard

Definim:

- $h^{max}(SS(n))$ lungimea drumului de la rădăcina SS(n) la nodul cu k_{max} din SS(n)
- $h^{min}(SD(n))$ lungimea drumului de la rădăcina SD(n) la nodul cu k_{min} din SD(n)

Cu aceste notaţii, reprezentarea grafică a eliminării Hibbard stânga a nodului *n* ce conţine cheia *k* este prezentată în Figura 3.

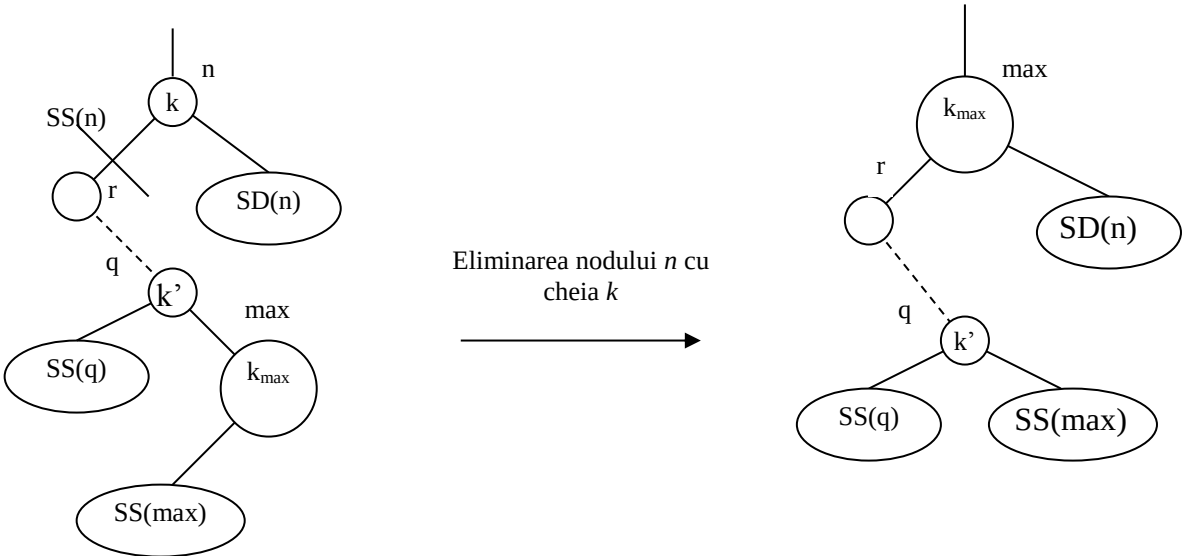


Figura 3 Eliminarea Hibbard stânga dintr-un arbore binar de căutare

Este evident că se poate aplica şi metoda de eliminarea în oglindă (simetrică), constând în aducerea nodului cu cheie minimă din subarborele drept al nodului *n* cu cheia *k* de eliminat.
O abordare eficientă a problemei presupune determinarea înălţimilor $h^{max}(SS(n))$ şi $h^{min}(SD(n))$ definite anterior şi aplicarea metodei *stânga* respectiv *dreapta* în funcţie de maximum dintre înălţimile celor doi subarbori (stâng şi drept). Vom numi metoda Hibbard "inteligentă" această abordare a eliminării unui nod.

Reluând exemplul anterior se constată că $h^{max}<h^{min}$ ceea ce duce la aplicarea *eliminării Hibbard dreapta* şi la scăderea înălţimii totale a arborelui (vezi Figura 4). O urmare evidentă a faptului că înălţimea arborelui scade este aceea că viitoarele căutări în arbore se vor face mai rapid.

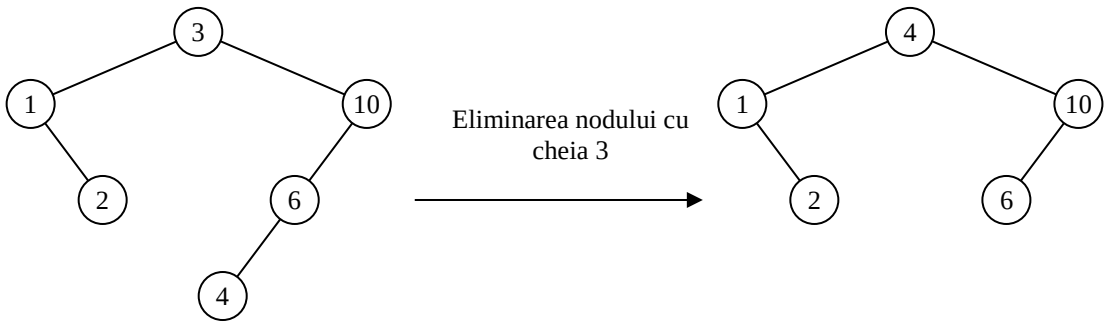


Figura 4 Eliminarea nodului cu cheia 3 duce la scăderea înălţimii arborelui şi deci la o optimizare a căutărilor viitoare

5.PROBLEME PROPUSE spre rezolvare

5.1. Să se scrie programul care realizează următoarele:

- **creare arbore binar de căutare;**
- **-parcurgere arbore;**

- **-înserare nod în arbore;**

-verificarea dacă e arbore binar complet sau nu;

5.2. Să se scrie programul care realizează următoarele

- **creare arbore binar de căutare;**
- **-ștergere nod în arbore;**
- **-determinarea înălțimii arborelui;**
- **-determinarea greutății arborelui;**

5.3 Să se scrie programul care realizează funcțiile modurilor posibile de stocare a matricelor rare ca structuri arborescente, apoi de înmulți cu un vector, folosind numai elementele nenule ale matricei rare. Rezultatul de comparat cu calculul tradițional.

5.4 Arbori binari pentru expresii aritmetice. Orice expresie aritmetica se poate reprezenta printr-un arbore binar in care nodurile interne conțin operatori iar nodurile terminale (frunze)
conțin operanzi. Exemplu de expresie cu operanzi întregi: (18 + ((15+35)/(30-5)))

Arborele corespunzător:

```
      +
    /   \
   18    /
        / \
       +   -
      / \ / \
     15 35 30 5
```

Fiecare nod de arbore va conține ca date un număr întreg. O expresie complet parantezată ("fully parenthesized") conține paranteze in jurul fiecărei subexpresii cu doi operanzi. Exemple:
(22+33) , (((10+20)+30)+40) , ((600/2)+(2*300)) , ((10*30)+(80/(60-40)))

5.5 Sa se scrie o funcție pentru afișarea unui arbore binar ca o expresie cu paranteze in jurul fiecărei (sub)expresii (ca varianta de afișare infixata a unui arbore binar). Exemplu de afișare: (22+(33/11))

Exemplu de utilizare:

```
int main () {
 tnod* r=build();
 infix(r);
 system("pause");
}
```

5.6. Funcția următoare calculează rezultatul unei expresii complet parantezate
cu operanzi si rezultate intermediare numere întregi :

```
int eval (char* & p) {                  // p = adresa sir cu expresie
 int a,b,x; char op;
 if (isdigit(*p)){                      // daca este o cifra
  x= (int) strtod(p,&p);   // conversie in binar
  return x;
 }
 if (*p=='(') {        // daca început subexpresie
  a= eval (++p);       // calcul operand 1
  op= *p++;                             // retine operator
  b=eval (p);         // calcul operand 2
  p++;                                  // peste ')' de la sfârșitul expresiei
  return calc(op,a,b);    // rezultat subexpresie
 }
}
```

5.7 Sa se scrie o funcție care construiește un arbore binar dintr-o expresie infixata cu paranteze la fiecare subexpresie, modificând funcția anterioara.

```
tnod* build (char* & p);  // p = adresa sir cu expresie
```

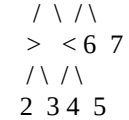
Pentru verificare se va afișa arborele cu funcțiile "infix" si "prefix".
Orice expresie logica sau de relație se poate reprezenta printr-un arbore binar in care nodurile interne conțin operatori iar nodurile terminale (frunze) conțin operanzi. Pentru simplificare vom considera operatori de relație si logici de un
singur caracter (așa cum sunt in limbajul BASIC):

<, > ,
= (in loc de ==), # (in loc de !=),
& (in loc de &&), | (in loc de ||)

Exemplu de expresie logica: (2>3|4<5)&(6#7) echivalent cu expresia din C: (2>3 || 4<5) && (6 != 7)

Arborele corespunzător:

```
      &
    /   \
   |    #
```



Fiecare nod de arbore va conține un întreg, care reprezintă fie codul unui operator, fie valoarea unui operand (intre 0 si 9).

5.8 O expresie complet parantezata ("fully parenthesized") conține paranteze in jurul fiecărei subexpresii cu doi operanzi. Exemple:
(2<3) , (((1&0)|1)&1) , ((2>3)|(4<5)) , ((1#2)|(1&(6=4)))

5.9. Sa se definească tipul "tnod" pentru un nod de arbore binar si o funcție pentru crearea unui nod de arbore binar cu date primite ca argumente: tnod* make (int x, tnod* left, tnod* right);

5.10. Funcția următoare construiește un arbore pentru expresia (1&(1=2))

```
tnod* build () {
 tnod * r, *f1, *f2;
 f1=make(1,0,0);
 f2=make(2,0,0);
 r=make('=',f1,f2);
 r=make('&',f1,r);
 return r;
}
```

5.11. Sa se scrie o funcție pentru afișarea unui arbore binar ca o expresie cu paranteze in jurul fiecărei (sub)expresii (ca varianta de afișare infixata a unui arbore binar). Exemplu de afișare: (1&(1=2))

Exemplu de utilizare:

```
int main () {
 tnod* r=build();
 infix(r);
 system("pause");
}
```

5.12. Sa se scrie o funcție care construiește un arbore binar dintr-o expresie infixata cu paranteze la fiecare subexpresie, modificând funcția anterioara. Se vor modifica si operațiile cu stiva pentru o stiva de pointeri "void*".

```
tnod* build (char* p);  // p = adresa sir expresie
```

Pentru verificare se va folosi funcția "main" de mai sus, cu apelarea funcțiilor de afișare infixata, prefixata si de evaluare pe arbore.

5. 13. Structura unui nod de arbore 2-4 cu valori întregi este definita astfel:

```
#define M 3         // nr maxim de valori pe nod
typedef struct bnod {
 int n;          // număr de valori dintr-un nod
 int val[M];       // vector de valori din nod
 struct bnod* leg[M+1];  // vector de pointeri la nodurile fii
} bnod;
```

Valorile din subarborele cu rădăcină leg[i] sunt mai mici decât val[i] iar valorile din subarborele cu rădăcină leg[i+1] sunt mai mari decât val[i].

5. 14. Sa se scrie funcțiile:

```
void set (bnod* p, int n,int v[],bnod* lnk[]); // introduce date in nodul p
bnod* make (int n,int v[],bnod* lnk[]); // creare nod cu valorile primite
void write (bnod* p);  // afișare conținut nod p (pointeri si valori)
```

Funcția "make" aloca memorie si folosește pe "set" pentru inițializare nod. Legăturile nefolosite dintr-un nod se vor face zero (NULL), in funcția "set".

5.15 Funcție "build" construiește un arbore 2-4 asemănător cu cel din exemplu:

```
bnod* build () {
 int i,v[3];
 bnod*a[4]={0};
 int trei[]={15,25,45,55,75,85};   // valori pe niv. 3
 bnod* t[6],* d[3];   // adrese succesori noduri pe niv. 3 si 2
 int doi[]={20,50,80};          // valori pe niv. 2
 // noduri pe niv. 3
 for (i=0;i<6;i++){
  v[0]=trei[i]; t[i]=make(1,v,a);
 }
  // noduri pe niv.2
 for (i=0;i<3;i++){
  v[0]= doi[i];
  a[0]=t[2*i]; a[1]=t[2*i+1];
  d[i]=make(1,v,a);
 }
  // nod pe niv.1 (rădăcină)
 v[0]=30; v[1]=70;
```

```
 for (i=0;i<3;i++)
  a[i]=d[i];
 return make (2,v,a);
}
```
Pentru verificare se adaugă apelul funcției "write" in funcția "make" astfel ca sa se afișeze fiecare nod creat (pe o linie separata).

5.16. Funcția "split" sparge un nod plin p in doua noduri si o valoare care se va duce pe nivelul superior:

```
void split (int x, bnod* p, int & med, bnod* & nou) {
 puts("split");
 int i,j, m,a[M+1],b[M]; bnod* al[M+1],*bl[M]={0};
 // copiază x si valorile din p in a
 i=j=0; m=M+1;
 if (x> p->val[M-1]) {a[M]=x; al[M]=0;m--;}  // daca x mai mare ca toate
 while (j<m)                // copiază din nodul p in a si al
  if (x > p->val[i]) {a[j]=p->val[i]; al[j]=p->leg[i];i++;j++;}
  else {a[j]=x;al[j]=0;j++;x=INT_MAX;}
 m = (M+1)/2;     // m = indice median intre valorile din a
 med=a[m];      // med este valoarea care se duce in sus
  // muta m valori din a in b
 for (j = 0; j < M-m; j++) {
  b[j] = a[j+m+1];
  bl[j] = al[j+m+1];
 }
 set(p,m,a,al);                          // modifica conținut nod p
 nou=make(M-m,b,bl);                             // creare nod nou
}
```
Sa se verifice funcțiile "add" si "split" prin adăugarea la arborele creat anterior a valorilor: 82,60,13,91,83,81

5.17. Sa se scrie o funcție pentru determinarea numărului de noduri dintr-un arbore B cu rădăcină r:

```
    int size (bnod* r);
```
Sa se adauge apelul funcției "size" in ciclul de adăugare valori din programul anterior.

5.18. Sa se scrie o funcție care caută nodul ce conține o valoare data x sau nodul frunza care va conține valoarea căutată dar negăsită:

```
bnod* find (bnod* r, int x, bnod* & pp);
```
Rezultatul funcției este adresa nodului care va conține x sau NULL daca x exista in arbore; "pp" este adresa nodului părinte al nodului găsit.
Căutarea se oprește la un nod frunza (care are leg[0]==0) sau la un nod care conține pe x.
Sa se verifice prin afișarea primei valori din nodurile găsit de "find" si "pp". Se vor caută valorile 21, 33, 41, 65.

5.19Functia următoare face adăugarea si corecția necesara pentru arbori AVL:

```
void addFix ( tnod* & r, int x ) { // adaugă x la arbore cu rădăcină r
if( r == NULL ) {                  // daca arbore vid
 r = make(x);       // creare nod frunza cu înălțimea 1
 return;   }
if (x==r->val)                     // daca x exista deja in arbore
  return;         // atunci nu se mai adaugă
if( x < r->val ) {      // daca x mai mic
  addFix ( r->st,x );    // atunci se adaugă la stânga
  if( ht(r->st) - ht(r->dr) > 1 )       // daca r dezechilibrat stânga
    if( x < r->st->val ) // daca x e in fiul stânga
     rotR( r );              // rotație simpla la dreapta
    else         // daca x e in fiul dreapta
     rotLR( r );    // rotație dubla stânga-dreapta
  }
  else {                    //daca x mai mare
   addFix ( r->dr,x );  // atunci se adaugă la dreapta
   if( ht(r->dr) - ht(r->st) > 1 ) // daca r dezechilibrat dreapta
    if( x > r->dr->val ) // daca x e in fiul dreapta
      rotL( r );    // rotație simpla la stânga
    else         // daca x e in fiul stânga
      rotRL( r );    // rotație dubla dreapta-stânga
  }
r->h = max(ht(r->st), ht(r->dr)) + 1;  // recalculare înălțime nod r
}
```
5.20De analizat problema și soluția în C cu structuri și pointeri. Se dau n orașe și se cunoaște distanta dintre oricare doua orașe. Un distribuito
de carte caută sa-si facă un depozit in unul dintre aceste orașe. Se cere sa se găsească traseul optim de la depozit către celelalte orașe
astfel încât distanta totala pe care o va parcurge pentru a distribui in toate celelalte n-1 orașe sa fie minima. Sa se precizeze care ar f

orașul in care sa se afle depozitul pentru ca toate celelalte orașe sa fie ușor accesibile {din acel centru de depozitare sa se poată pleca sper cat mai multe alte orașe}. Datele se citesc dintr-un fișier astfel:

-pe prima linie numărul de orașe

-pe următoarele linii traseele existente astfel: orașul din care pleacă, orașul in care ajunge si distanta in km a drumului.

{Deoarece vor exista foarte multe trasee algoritmul lui Prim este mai bun. Fiind un graf neorientat se poate pleca cu vârful 1. Pentru a afla care ar fi orașul optim vedem în arbore care este nodul cu cei mai mulți fii si refacem vectorul tata.}

5.21. Construiți modelul analitic pentru matricea rară reprezentată prin trei vectori. 2. Scrieți funcția pentru compararea a două matrice rare. 3. Scrieți funcția pentru interschimbul a două frunze dintr-un arbore ternar. 4. Scrieți procedura care verifică dacă elementele unei liste duble sunt deja sortate crescător sau descrescător. 5. Scrieți funcțiile de conversie a unui masiv bidimensional și ale unei liste simple în masiv unidimensional, justificați diferențele. 6. Calculați gradul de utilizare aferent arborilor binari care permit traversarea în toate direcțiile.

5.22 Se dă un graf conex. Se cere împărțirea acestuia în m arbori parțiali de cost minim fiecare cu p vârfuri. Să se afișeze acești arbori.

5.23 Să se scrie programul care realizează funcțiile modurilor posibile de stocare a matricelor rare tridiagonale ca structuri arborescente apoi de inversat, folosind numai elementele nenule ale matricei rare. Rezultatul de comparat cu calculul tradițional.

5.24. Scrieți funcția de normalizare a unei matrice. 8. Alegeți domeniul de definire pentru matricea cu 4 linii și patru coloane ai căror termeni sunt termenii șirului Fibonacci. 9. Enumerați în ordinea importanței și justificați 3 argumente în ordinea importanței, prin care justificați alegerea structurilor de date din proiectul individual.

5.25. Scrieți funcția care verifică faptul că toți subarborii unui arbore binar sunt simetrici.

5.26. Scrieți funcția pentru inserarea unui nod într-un arbore binar.

5.27. Scrieți funcțiile pentru modurile de ștergere a unui nod într-un arbore binar. De descris structurile prin organigrama arborelui și de efectuat în program toate operațiile necesare pentru a evidenția principalele proprietăți al arborelui.

5.28. Scrieți funcția pentru traversarea a unui arbore binar.

5.29. Definiți nodul unui arbore B scriind instrucțiunile si funcțiile corespunzătoare.

5.30. Scrieți funcția care creează un arbore binar echilibrat dintr-un vector sortat și de efectuat în program toate operațiile necesare pentru a evidenția principalele proprietăți al arborelui.

5.31. Să se scrie programul care realizează: - crearea arborelui binar de căutare; - parcurgere arbore; - inserare nod în arbore.

5.32. Să se scrie programul care realizează: - crearea arborelui binar de căutare; - ștergere nod din arbore; - determinarea înălțimii arborelui; - verificarea dacă este arbore binar complet sau nu.

5.33. Să se scrie funcția care realizează copierea unui arbore și apoi să se afișeze nodurile acestuia.

5.34. Să se construiască un arbore binar cu elemente citite dintr-un vector, să se scrie o funcție care returnează adresa unui anumit nod al arborelui, nod de care se va lega un alt arbore construit cu elemente citite tot dintr-un vector.

5.35. Să se construiască un arbore binar, să se scrie două funcții care calculează suma elementelor din subarborele stâng respectiv suma elementelor din subarborele drept.

5.36. Să se construiască un arbore binar în care informația utilă este alcătuită dintr-o valoare întreagă și un pointer la o listă simplu înlănțuită.

5.37. Să se scrie funcția de construire arbore, tipărire arbore și de căutare a unui nod din arbore cu o valoare dată.

5.38. Scrieți și apelați funcția de copiere a unui arbore binar.

5.39. Să se scrie programul care realizează funcțiile modurilor posibile de stocare a matricelor rare, format din cinci diagonale ca structuri arborescente, apoi de înmulțit cu un vector, folosind numai elementele nenule ale matricei rare. Rezultatul de comparat cu calculul tradițional.

5.40. Analizați structurile arborescente ale matricelor rare si elaborați programul în care se utilizează diferite cazuri, afișând arborele si structurile matricelor. Construiți principalele organigrame.

Bibliografie

1. Limbajul de programare C". Brian W.Kernighan. Dennis M.Ritchie.
3. Liviu Negrescu. ”*Limbajul de programare C și C++”* V.1-4. Buc. 1999
4. Knuth, D. E. - "Arta programării calculatoarelor, vol. 1: Algoritmi fundamentali", Ed. Teora, 1999.
5. Knuth, D. E. - "Arta programării calculatoarelor, vol. 2: Algoritmi seminumerici", Ed. Teora, 2000.
6. Knuth, D. E. - "Arta programării calculatoarelor, vol. 3: Sortare si căutare", Ed. Teora, 2001.
7. Bacivarov, A.; Nastac, I. - "Limbajul C. Îndrumar de laborator", Tipografia UPB, București, 1997.
8. Bates, J; Tompkins, T. - "Utilizare C++", Ed. Teora 2001.
9. Andonie, R.; Gabarcea, I. - "Algoritmi fundamentali. O perspectiva C++", Ed. Libris, 1995.
10. Help din Turbo C ++IDE ,versiunea 3.0. 1999
11. CORMEN, T. - LEISERSON, CH. - RIVEST, R. : Introducere in algoritmi, Editura Computer Libris. Agora, Cluj-Napoca, 2000.
12. Conspectele la PC 2009 și SDA-2010
13. L. Negrescu. Limbajele C si C++ pentru începători. Vol 1 si 2.Ed.Microinformatica.Cluj-N.,1994 si reeditata in 1996 ,1998,2000.

14 L. Livovschi, H. Georgescu. Analiza si sinteza algoritmilor. Ed. Enciclopedică, București,, 1986.
15 I. Ignat, C. Ignat. Structuri de date si algoritmi. Îndrumător de laborator. U.T.Pres, 2001.
16 Informatica. Îndrumar metodic pentru lucrările de laborator. Marin Șt. Chișinău 2003. UTM