3. ARBORI

3.1. INTRODUCERE

Pentru multe aplicații, timpul de acces liniar la informația cuprinsă într-o listă este foarte mare. De asemenea, uneori este necesară folosirea unei descrieri ierarhice pentru modelarea diferitelor fenomene şi/sau obiecte din natură. Prin descrierea ierarhică se înțelege descompunerea unei entități în subentități, fiecare dintre ele putând fi caracterizate printr-un set de atribute sau însuşiri. De asemenea, utilizând o astfel de descriere, se realizeazã şi o ierarhizare a părților unei entități pe unul sau mai multe niveluri.

Organizarea ierarhică este întâlnită în diverse domenii, ca de exemplu: organizarea administrativă a unei țări, planificarea meciurilor în cadrul unui turneu sportiv (Fig.3.1.), evaluarea unor expresii de calcul (Fig.3.2), etc.

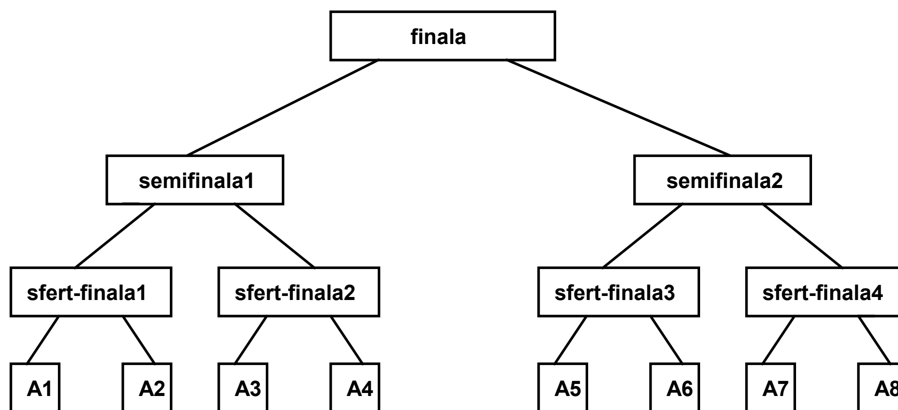

Fig.3.1. Organizarea ierarhică a unei competiţii sportive cu 8 echipe (Ai, i∈{1,2,...,8})

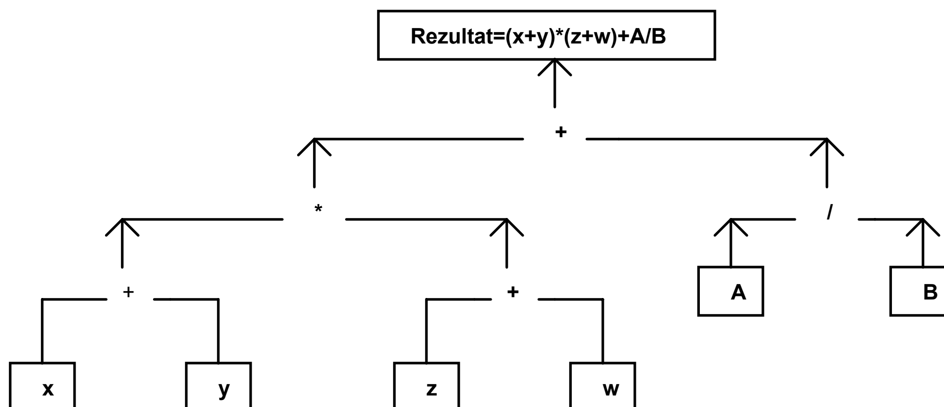

Fig.3.2. Evaluarea expresiei de calcul (x+y)*(z+w)+(A/B)

În figurile de mai sus sunt date exemple de structuri cu organizare ierarhică. Structurile cu organizare ierarhică pot fi reprezentate cu ajutorul arborilor.

3.2. MODELUL ARBORELUI

Definirea arborilor se poate face recursiv. Astfel, un arbore este o colecție de noduri (numărul lor poate fi şi 0). Un arbore constă dintr-un nod numit rădăcină, *r*, şi mai mulți subarbori, T_1, T_2, ... ,T_n, unde *n* este un număr natural (toți aceşti subarbori sunt legați direct la nodul rădăcină). Nodul rădăcină al fiecărui subarbore se numeşte ***nod fiu*** al nodului rădăcină *r*.

Legătura dintre două noduri succesive din arbore se numeşte ***arc***; legătura dintre rădăcină şi un nod oarecare presupune parcurgerea a *m* arce, pe care se află *n=m-1* noduri. Valoarea *n* reprezintă nivelul pe care este poziționat nodul față de rădăcină (nivelul rădăcinii este 1, prin convenție).

Înălțimea unui arbore se poate defini ca valoarea maximă luată de nivelurile nodurilor terminale. În figura 3.3. este prezentat un exemplu de arbore cu înălțimea 3 (este precizată şi înălțimea fiecăruia dintre subarbori).

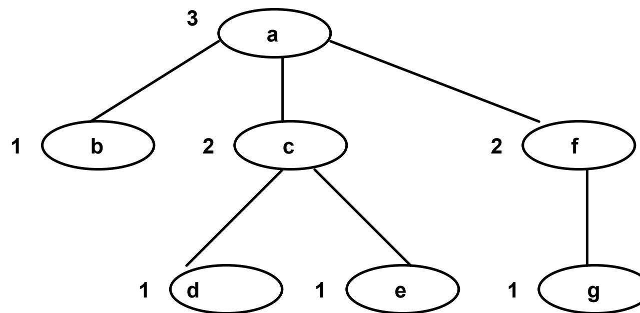

Fig.3.3. Exemplu de arbore cu înălțimea 3

Descendenții direcți ai unui nod, care se mai numesc şi ***fii***, sunt reprezentați de mulțimea nodurilor cu înălțimea mai mică sau egală cu a sa, legate direct de acesta. Numărul de descendenți direcți ai unui nod reprezintã ***ordinul sau gradul nodului***. ***Ordinul sau gradul arborelui*** este valoarea maximă luată de gradul unui nod component al arborelui. Numărul maxim de noduri conținute într-un arbore de ordin *d* şi înălțime *H*, este dat de următoarea relație:

$$N_{Max} = \sum_{k=0}^{H-1} d^k \qquad (1)$$

Arborele binar este un arbore în care fiecare nod are maxim doi fii. Cei doi subarbori corespunzători fiilor unui nod se numesc ***subarbore stâng***, respectiv ***subarbore drept***. Între numărul *N* de noduri al unui arbore binar şi înălțimea sa *H*, există relațiile:

$$H \le N \le 2^H - 1 \qquad (2)$$

$$\log_2 N \le H \le N \qquad (3)$$

Arborele echilibrat este acel arbore pentru care diferenţa între înălţimile subarborilor săi este cel mult 1. Arborii binari echilibraţi se numesc şi ***arbori AVL*** (Adelson-Velsky-Landis).

3.3. IMPLEMENTAREA ARBORILOR

Implementarea arborilor presupune crearea în fiecare nod a unei structuri de date în care să existe câte un pointer/referinţă către fiecare dintre subarbori (Fig.3.4).

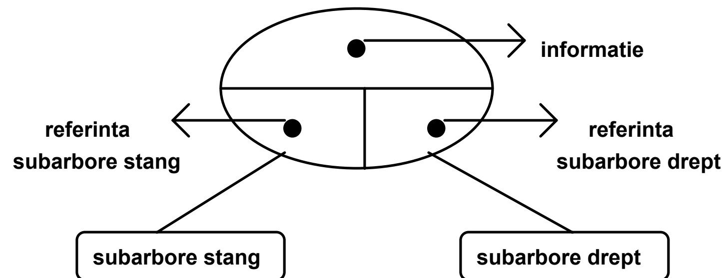

Fig.3.4. Structura de date în nodul unui arbore

Un arbore vid are referinţele rădăcinii *nil* (în Pascal) sau *NULL* (în C/C++).

3.4. OPERAŢII CU ARBORI

Operaţiile fundamentale la care poate fi supus un arbore binar, sunt următoarele:

- operaţii care furnizează informaţii despre un arbore sau un nod propriu:
    - testare dacă arborele este vid sau nu;
    - determinare pointer/referinţă la care se află informaţia dintr-un nod;
    - determinare pointer/referinţă la subarborele stâng aferent unui nod;
    - determinare pointer/referinţă la subarborele drept aferent unui nod;
- operaţii ce modifică un arbore sau un nod:
    - construirea arborelui;
    - modificarea informaţiei din nodul rădăcină;
    - modificarea referinţelor subarborelui stâng, respectiv drept, aferenţi unui nod rădăcină;
    - eliberarea memoriei alocate pentru rădăcina unui arbore;
- operaţii de traversare a unui arbore:
    - traversarea în lăţime (**Ex**.Fig.3.1: finala, semifinala1, semifinala2, ... ,A1, A2, ... ,A8);
    - traversarea în adâncime:

✸ traversarea în preordine (*RSD*): se face în ordinea - rădăcină, stânga, dreapta (**Ex.**Fig.3.2: +*+xy+zw/AB);

✸ traversarea în inordine (*SRD*): se face în ordinea - stânga, rădăcină, dreapta (**Ex.**Fig.3.2: x+y*z+w+A/B);

✸ traversarea în postordine (*SDR*): se face în ordinea - stânga, dreapta, rădăcină (**Ex.**Fig.3.2: xy+zw+*AB/+);

Observaţii:

1. Indiferent de tipul de parcurgere al arborilor, ordinea relativă de tratare a nodurilor terminale este identică (de la stânga la dreapta).

2. Forma de parcurgere care generează forma cea mai apropiată de scrierea uzuală a expresiei de calcul este cea în inordine.

3.5. DESFĂŞURAREA LUCRĂRII

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 corespunzător la găsirea sa.

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

#define NR        10        //numar de noduri

typedef struct Nod
        {
        void *Info;
        struct Nod *Stanga, *Dreapta;
        }NodT, *NodPtrT;

NodPtrT t;
NodPtrT *root;

int TestVid(NodPtrT Radacina)
        {
        return Radacina==NULL;
        }

int AdrInfo(NodPtrT Radacina, void *InfoPtr)
        {
        if (Radacina!=NULL) InfoPtr=(void *)Radacina->Info;
        else InfoPtr=NULL;
        return (Radacina!=NULL && InfoPtr!=NULL);
        }
```

```
int AdrStanga(NodPtrT Radacina, void *InfoPtr)
	{
	if (Radacina!=NULL) InfoPtr=Radacina->Stanga;
	else InfoPtr=NULL;
	return (Radacina!=NULL && InfoPtr!=NULL);
	}

int AdrDreapta(NodPtrT Radacina, void *InfoPtr)
	{
	if (Radacina) InfoPtr=Radacina->Dreapta;
	else InfoPtr=NULL;
	return (Radacina!=NULL && InfoPtr!=NULL);
	}

int ConstrNod(NodPtrT *Radacina, void *Info, void *Stanga, void *Dreapta)
	{
	*Radacina=malloc(sizeof(NodT));
	if (!(Radacina)) return 0;
	(*Radacina)->Info=Info;
	(*Radacina)->Stanga=Stanga;
	(*Radacina)->Dreapta=Dreapta;
	return 1;
	}

int ModificaInfo(NodPtrT Radacina, void *InfoPtr)
	{
	if (Radacina) Radacina->Info=InfoPtr;
	return (Radacina!=NULL);
	}


int ModificaStanga(NodPtrT Radacina, NodPtrT SubArbore)
	{
	if (Radacina) Radacina->Stanga=SubArbore;
	return (Radacina!=NULL);
	}

int ModificaDreapta(NodPtrT Radacina, NodPtrT SubArbore)
	{
	if (Radacina) Radacina->Dreapta=SubArbore;
	return (Radacina!=NULL);
	}

int ElibNod(NodPtrT *Radacina, void** Info, NodPtrT *Stanga, NodPtrT *Dreapta)
	{
	if (!(*Radacina)) return 0;
	else
		{
		*Info=(void*)(*Radacina)->Info;
		*Stanga=(*Radacina)->Stanga;
		*Dreapta=(*Radacina)->Dreapta;
		free(*Radacina);
		*Radacina=NULL;
		return 1;
		}
	}
```

```
int Terminal(NodPtrT Radacina)
	{
	return (Radacina->Stanga==NULL && Radacina->Dreapta==NULL);
	}

int NrNoduri(NodPtrT Radacina)
	{
	if (!Radacina) return 0;
	else return(NrNoduri(Radacina->Stanga)+NrNoduri(Radacina->Dreapta)+1);
	}

int Inaltime(NodPtrT Radacina)
	{
	if (Radacina==NULL) return 0;
	else
		{
		int InaltimeSt, InaltimeDr;
		InaltimeSt=Inaltime(Radacina->Stanga);
		InaltimeDr=Inaltime(Radacina->Dreapta);
		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;
	do
		{
		InfoPtr=&node[NR-NrNoduri];
```

```
            if (ConstrNod(&NodNou, InfoPtr, NULL, NULL))
                AdaugaNod(ArborePtr, NodNou);
            else break;
            } while (--NrNoduri>0);
        }

void PreOrdTrav(NodPtrT Radacina)
        {
        if (Radacina)
            {
            printf("%d\t",*(int*)Radacina->Info);
            PreOrdTrav(Radacina->Stanga);
            PreOrdTrav(Radacina->Dreapta);
            }
        }

void InOrdTrav (NodPtrT Radacina)
        {
        if (Radacina)
            {
            InOrdTrav(Radacina->Stanga);
            printf("%d\t",*(int*)Radacina->Info);
            InOrdTrav(Radacina->Dreapta);
            }
        }

void PostOrdTrav (NodPtrT Radacina)
        {
        if (Radacina)
            {
            PostOrdTrav(Radacina->Stanga);
            PostOrdTrav(Radacina->Dreapta);
            printf("%d\t",*(int*)Radacina->Info);
            }
        }



void Cauta (NodPtrT Radacina,int b)
        {
        int c;
        if (Radacina)
            {
            Cauta(Radacina->Stanga,b);
            c=*(int*)Radacina->Info;
            if(c==b) {t=Radacina;}
            Cauta(Radacina->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 cautat: ");
        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();

}
```