## Laborator 4 – Arbori

### 4.1 Probleme rezolvate

Un arbore reprezintă o colecție de noduri relaționate între ele în mod ierarhic. Fiecare nod are asociată o anumită informație, de exemplu informații despre angajații unei companii (vezi Figura 1), arborele reprezentând în acest caz ierarhia de organizare a companiei.

În funcție de poziția lor în arbore, nodurile pot fi de mai multe tipuri:

- **părintele unui nod**: nodul care se află imediat deasupra nodului în cauză;
- **copilul unui nod**: nodul care urmează imediat sub acesta;
- **nod rădăcină**: nod unic în arbore, care nu are nici un părinte (primul nod);
- **noduri frunză**: nod care nu are nici un copil (ultimele noduri).

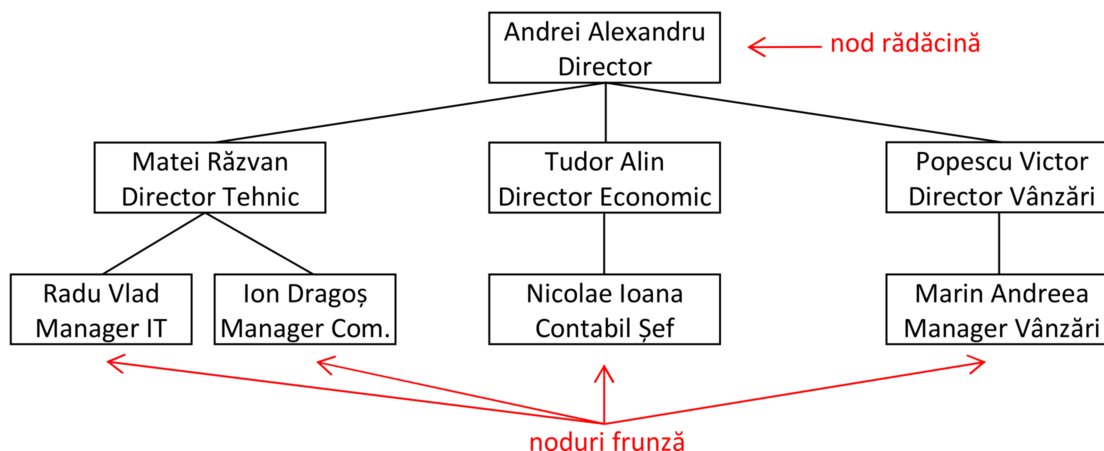


Figura 1 – Exemplu de structură arborescentă.

În exemplul de mai sus, *Andrei Alexandru* este rădăcina arborelui, deoarece nu are nici un părinte. Copiii săi direcți sunt: *Matei Răzvan*, *Tudor Alin* și *Popescu Victor*, care au la rândul lor alți copii. Aceștia din urmă sunt frunze ale arborelui, deoarece nu mai au alți copii.

Orice arbore are următoarele proprietăți:

- Există o singură rădacină;
- Toate nodurile au exact un singur părinte, excepție făcând nodul rădăcină;
- Nu există cicluri, ceea ce înseamnă că, pornind de la un nod oarecare nu se poate parcurge un anumit traseu, astfel încât să se ajungă înapoi la nodul de plecare.

Un caz particular de arbori, sunt **arborii binari**. Aceştia au proprietatea că fiecare nod are maxim 2 copii: fie copilul din stânga şi copilul din dreapta, fie un singur copil, fie niciunul.

Fiecare nod al unui arbore binar reţine informaţia propriu-zisă, precum şi adresele de memorie a copiilor din stânga şi din dreapta, dacă aceştia există (vezi Figura 2). Structura unui astfel de nod poate fi de tipul următor:

```
struct NOD
{
	int informatie;
	struct NOD *adresa_copil_dreapta;
	struct NOD *adresa_copil_stanga;
}
```

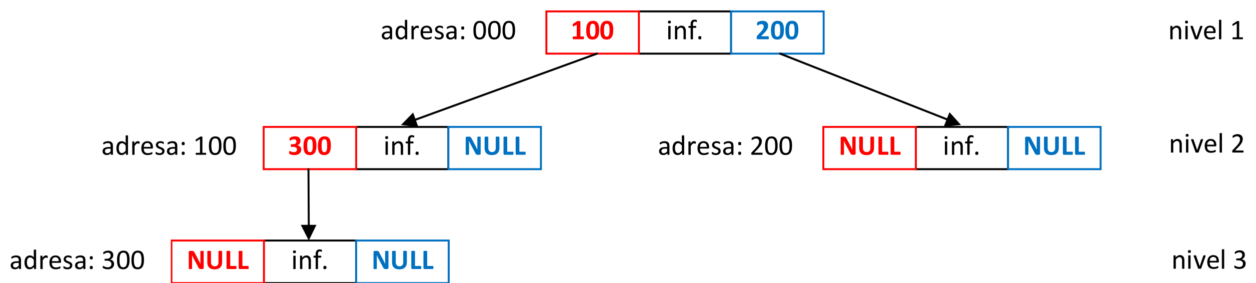


Figura 2 – Exemplu de arbore binar

**Arborele binar de căutare** este un arbore binar particular care stochează suplimentar informaţiilor existente o valoare numerică cu rol de ordonare. Acesta are proprietatea că, pentru oricare nod *n*, fiecare dintre descendenţii din subarborele din stânga are valoarea numerică mai mică decât a nodului *n*, iar fiecare din descendenţii din subarborele din dreapta va avea valoarea informaţiei mai mare sau egală. Un exemplu de arbore binar de căutare este ilustrat în Figura 3.

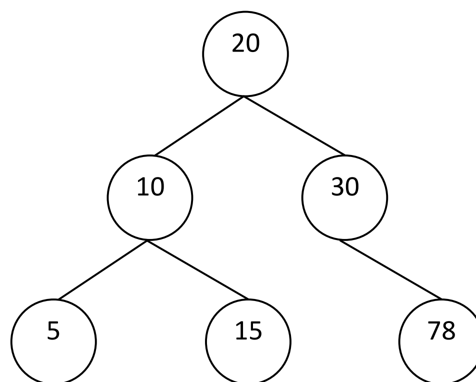


Figura 3 – Exemplu de arbore binar de căutare

În cuprinsul acestui laborator se va implementa un arbore binar de căutare, împreună cu operațiile uzuale ce se pot realiza cu acesta, precizând că pentru oricare alt tip de arbore implementările se realizează în mod similar.

Operațiile uzuale sunt:

- inserarea unui nod în arbore;
- parcurgerea arborelui;
- ștergerea unui subarbore.

P4.1 Să se realizeze un program ce permite implementarea unui arbore binar de căutare, precum și a operațiilor uzuale cu acesta. Programul permite afișarea pe ecran a unui meniu cu următoarele operații posibile:

- [1] Citirea unei valori de la tastatură și inserarea acesteia în arbore;
- [2] Afișarea arborelui în preordine;
- [3] Afișarea arborelui în inordine;
- [4] Afișarea arborelui în postordine;
- [5] Ștergerea unui subarbore, descendent dintr-un nod specificat;
- [6] Căutarea unui nod în arbore;
- [0] Ieșire din program.

Fiecare opțiune din meniu va fi implementată folosind funcții.

Rezolvare:

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

/* definire structura arbore */
struct NOD
{
 int x;
 struct NOD *NOD_stanga;
 struct NOD *NOD_dreapta;
};

/* functie creare nod nou */
struct NOD *creare_nod(int x)
{
 struct NOD *nod_nou;

 /* alocare memorie nod*/
 nod_nou=(struct NOD *)malloc(sizeof(struct NOD));
```

```
 if (nod_nou==NULL)
 {
  printf("Eroare: Memoria nu a putut fi alocata! \n");
  return NULL;
 }

 /* initializare informatii */
 nod_nou->x=x;
 nod_nou->NOD_stanga=NULL;
 nod_nou->NOD_dreapta=NULL;

 return nod_nou;
}

/* inserare nod in arbore */
struct NOD *inserare_nod(struct NOD *prim, int x)
{
 struct NOD *nod_nou, *nod_curent, *nod_parinte;

 nod_nou=creare_nod(x);

 if (prim==NULL)
 {
  /* arborele este vid */
  prim=nod_nou;
  printf("A fost adaugat primul nod: %d. \n", prim->x);
  return prim;
 }
 else
 {
  /* pozitionare in arbore pe parintele nodului nou */
  nod_curent=prim;

  while (nod_curent!=NULL)
  {
   nod_parinte=nod_curent;

   if (x<nod_curent->x)   /* parcurgere spre stanga */
    nod_curent=nod_curent->NOD_stanga;
   else                   /* parcurgere spre dreapta */
    nod_curent=nod_curent->NOD_dreapta;
  }

  /* creare legatura nod parinte cu nodul nou */
  if (x<nod_parinte->x)
  {
   /* se insereaza la stanga nodului parinte */
   nod_parinte->NOD_stanga=nod_nou;
   printf("Nodul %d a fost inserat la stanga nodului %d. \n",
          x, nod_parinte->x);
  }
```

```
  else
  {
   /* se insereaza la dreapta nodului parinte */
   nod_parinte->NOD_dreapta=nod_nou;
   printf("Nodul %d a fost inserat la dreapta nodului %d.\n",
            x, nod_parinte->x);
  }

 return prim;
 }
}

/* parcurgere arbore in preordine */
void afisare_preordine(struct NOD *prim)
{
 if (prim!=NULL)
 {
  /* parcurgere radacina, stanga, dreapta */
  printf("%d \n", prim->x);
  afisare_preordine(prim->NOD_stanga);
  afisare_preordine(prim->NOD_dreapta);
 }
}

/* parcurgere arbore in inordine */
void afisare_inordine(struct NOD *prim)
{
 if (prim!=NULL)
 {
  /* parcurgere stanga, radacina, dreapta */
  afisare_inordine(prim->NOD_stanga);
  printf("%d \n", prim->x);
  afisare_inordine(prim->NOD_dreapta);
 }
}

/* parcurgere arbore in postordine */
void afisare_postordine(struct NOD *prim)
{
 if (prim!=NULL)
 {
  /* parcurgere stanga, dreapta, radacina */
  afisare_postordine(prim->NOD_stanga);
  afisare_postordine(prim->NOD_dreapta);
  printf("%d \n", prim->x);
 }
}
```

```
/* stergerea unui arbore sau subarbore */
struct NOD *stergere_arbore(struct NOD *tmp)
{
 if (tmp!=NULL)
 {
  stergere_arbore(tmp->NOD_stanga);
  stergere_arbore(tmp->NOD_dreapta);
  free(tmp);
 }

 return NULL;
}

/* cautarea unui nod dorit */
struct NOD *cauta_nod(struct NOD *tmp, int x)
{
 if (tmp!=NULL)
 {
  if (x==tmp->x)
  {
   printf("Nodul a fost gasit. \n");
   return tmp;
  }
  else if (x<tmp->x)
   return cauta_nod(tmp->NOD_stanga, x);
  else
   return cauta_nod(tmp->NOD_dreapta, x);
 }
 else
 {
  printf("Nodul dorit nu exista in arbore.\n");
  return NULL;
 }
}

int main()
{
 struct NOD *prim=NULL, *nod_gasit;
 char operatie;
 int x;

 printf("MENIU: \n");
 printf("[1] Inserare nod in arbore \n");
 printf("[2] Afisare arbore preordine \n");
 printf("[3] Afisare arbore inordine \n");
 printf("[4] Afisare arbore postordine \n");
 printf("[5] Stergere arbore \n");
 printf("[6] Cautare nod in arbore \n");
 printf("[0] Iesire din program \n");
```

```
do
{
 printf("\nIntroduceti operatie: ");
 operatie=getche();

 printf("\n");
 switch (operatie)
 {
  case '1':
   printf("#Inserare nod in arbore# \n");
   printf("Introduceti valoarea nodului care va fi inserat: ");
   scanf("%d", &x);
   prim=inserare_nod(prim, x);
   break;

  case '2':
   printf("#Afisare arbore preordine# \n");
   if (prim==NULL)
    printf("Atentie: Arborele este gol.");
   else
    afisare_preordine(prim);
   break;

  case '3':
   printf("#Afisare arbore inordine# \n");
   if (prim==NULL)
    printf("Atentie: Arborele este gol.");
   else
    afisare_inordine(prim);
   break;

  case '4':
   printf("Afisare arbore postordine: \n");
   if (prim==NULL)
    printf("Atentie: Arborele este gol.");
   else
   afisare_postordine(prim);
   break;

  case '5':
   printf("#Stergere arbore# \n");
   if (prim==NULL)
    printf("Atentie: Arborele este gol.");
   else
   {
    printf("Introduceti valoarea nodul al carui arbore va fi sters: ");
    scanf("%d", &x);
    nod_gasit=cauta_nod(prim, x);
    if (nod_gasit!=NULL)
     {
      nod_gasit->NOD_stanga=
      stergere_arbore(nod_gasit->NOD_stanga);
```

```
      nod_gasit->NOD_dreapta=
      stergere_arbore(nod_gasit->NOD_dreapta);
      printf("Arborele a fost sters. \n");
     }
    }
    break;

   case '6':
    printf("#Cautare nod in arbore# \n");
    if (prim==NULL)
     printf("Atentie: Arborele este gol.");
    else
    {
     printf("Introduceti valoarea nodului: ");
     scanf("%d", &x);
     cauta_nod(prim, x);
    }
    break;

   case '0':
    printf("Iesire din program \n");
    stergere_arbore(prim);
    system("PAUSE");
    return 0;
    break;

   default:
    printf("Operatie invalida \n");
   }
 } while(1);
}
```

Discuţie:

- Acest program implementează un arbore binar de căutare, ale cărui noduri conțin ca informație numere întregi. Fiecare nod al arborelui este de tipul `struct NOD`. Nodurile relaționează între ele prin intermediul pointerilor `struct NOD *NOD_stanga` și `struct NOD *NOD_dreapta`, care stochează adresele de memorie ale copiilor acestora;
- Funcția `struct NOD *creare_nod(int x)` este o funcție generală, prin intermediul căreia se creează și se alocă memorie pentru un nod nou. Conținutul acestuia este inițializat cu informația primită ca parametru de intrare (`int x`), iar locațiile copiilor sunt inițializate cu NULL, deoarece în prima fază acest nod nu este inserat în arbore. Adresa nodului nou creat este returnată printr-un pointer la structura NOD;
- Inserarea unui nod în arbore se face cu ajutorul funcției `struct NOD* inserare_nod(struct NOD *prim, int x)`. Aceasta primește ca parametru de intrare adresa de memorie a primului nod (rădăcina arborelui), dacă aceasta există, și valoarea

nodului ce se dorește a fi inserat. Dacă nodul rădăcină nu există, acesta se creează prin apelarea funcției `creare_nod()` și se returnează adresa lui. Dacă nodul rădăcină există, nodul nou creat se inserează în arbore pe poziția corespunzătoare, în funcție de valoarea acestuia. Pentru găsirea poziției, este necesară o parcurgere a arborelui pornind de la nodul rădăcină și vizitând nodurile, până când se ajunge la un nod frunză. Decizia pentru vizitarea copilului unui nod curent se face în funcție de valoarea nodului ce se dorește a fi inserat, astfel: dacă valoarea nodului de inserat este mai mică decât valoarea nodului curent, atunci se va vizita copilul nodului curent din stânga, iar dacă nu, se va vizita copilul din dreapta. După luarea acestei decizii, nodul curent devine nodul vizitat, și procesul se reia. Parcurgerea acestor noduri s-a implementat folosind structura repetitivă `while`. Atunci când se ajunge la un nod frunză (nu are nici un copil), nodul creat se inserează în arbore prin inițializarea uneia dintre cele 2 adrese de legătură a acestuia cu locația nodului creat: `nod_parinte->NOD_stanga=nod_nou` sau `nod_parinte->NOD_dreapta=nod_nou`. Decizia de alegere a acestei adrese (stânga sau dreapta) se face după aceeași regulă de comparație definită mai sus. Astfel, nodul frunză devine nod părinte pentru noul nod, iar noul nod devine nod frunza în arbore. În final se returnează adresa nodului rădăcină, care este necesară doar pentru cazul în care arborele este gol, în celelalte cazuri ea rămânând neschimbată;

- Afișarea arborelui se realizează folosind cele trei tipuri de parcurgere a unui arbore: *preordine*, *inordine* și *postordine*. Aceste denumiri corespund modului în care se vizitează rădăcina:

  - *preordine*: se vizitează mai întâi rădăcina, copilul (copiii) din stânga, iar apoi copilul (copiii) din dreapta;
  - *inordine*: se vizitează mai întâi copilul (copiii) din stânga, apoi rădăcina și copilul (copiii) din dreapta;
  - *postordine*: se vizitează copilul (copiii) din stânga, apoi copilul (copiii) din dreapta și apoi rădăcina.

  Fiecare tip de afișare este implementat într-o funcție separată, ce primește ca parametru de intrare adresa de memorie a nodului rădăcină: `void afisare_preordine(struct NOD *prim)`, `void afisare_inordine(struct NOD *prim)`, `void afisare_postordine(struct NOD *prim)`. Funcțiile sunt recursive, astfel încât este important să înțelegem ce se întâmplă pe unul din niveluri (de exemplu pe primul), pe restul procedându-se identic;

- Căutarea unui nod în arbore se realizează cu ajutorul funcției recursive `struct NOD* cauta_nod(struct NOD *tmp, int x)` ce primește ca parametru de intrare adresa de memorie a nodului rădăcină și valoarea nodului ce se dorește a fi căutat. Funcția se autoapelează, vizitând nodurile în funcție de valoarea nodului căutat, după aceeași regulă de comparație definită în cazul inserării unui nod. Condiția de oprire este îndeplinită fie atunci

când nodul este găsit, caz în care se returnează adresa acestuia, fie când s-a ajuns la un nod frunza și nodul nu a fost găsit, caz în care se returnează NULL;

- Funcția `struct NOD* stergere_arbore(struct NOD *tmp)` permite ștergerea arborelui descendent al unui nod specificat. Pornind de la adresa acestui nod, se vizitează și se șterge recursiv fiecare copil, ștergându-se și legătura acestuia cu părintele. Acest lucru se realizează prin atribuirea valorii NULL returnate de funcție în iterația precedentă către părinte. Pentru a găsi adresa de memorie a nodului specificat este necesară apelarea funcției `cauta_nod()`, lucru ce se realizează în funcția `main()`, imediat înainte de ștergerea propriu-zisă;
- Meniul programului este implementat în funcția `main()`, prin intermediul căruia utilizatorul poate apela funcțiile definite mai sus.

## 4.2 Probleme propuse

1. Să se modifice corespunzător programul anterior astfel încât să includă în meniu și posibilitatea de calcul a numărului de frunze din arbore.
2. Să se modifice corespunzător programul anterior astfel încât să includă în meniu și posibilitatea de calcul a numărului maxim de niveluri din arbore.
3. Implementați operațiile de afișare, ștergere și căutare a nodurilor unui arbore folosind funcții nerecursive.
4. Să se modifice corespunzător programul anterior astfel încât să includă în meniu și posibilitatea ștergerii complete a arborelui (pornind de la radacină).