SDA - Seminar 7 - AB

- **Conținut**: Materialul curent exemplifică rezolvarea mai multor probleme cu **Arbori Binari** (AB).

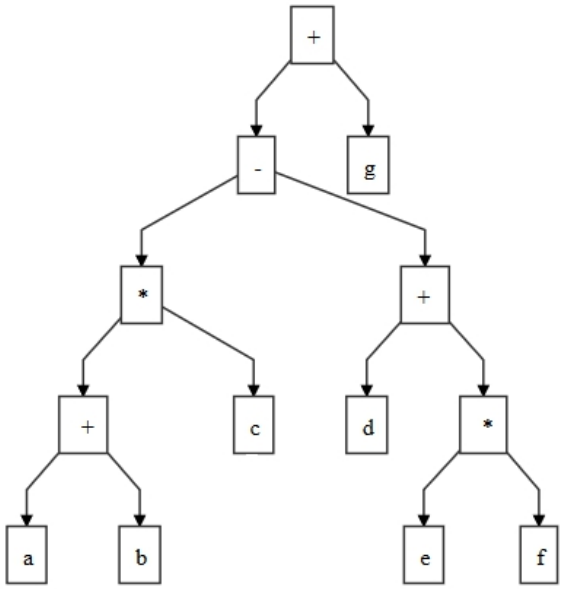
1. **Să se construiască arborele binar asociat unei expresii aritmetice conținând operatorii aritmetici +, -, *, /, pornind de la forma poloneză post-fixată.**

Considerând expresia aritmetică (a + b) * c – (d + e * f) + g, care este forma ei poloneză postfixată?

✓ Forma poloneză postfixată (FPP) a expresiei este: a b + c * d e f * + - g +

Observație: În forma **post**fixată, operatorii apar **după** operanzi.

Care este arborele binar asociat expresiei exemplificate?



Justificare:

Considerând expresia (a + b)* c – (d + e * f) + g, observăm că:

- Se efectuează o sumă (=> **+** în rădăcină, pe nivelul 0) între o diferență (=> **-** ca fiu stâng pentru rădăcină) și g (=> **g** ca fiu drept pentru rădăcină)
- Diferența se efectuează între un produs (=> ***** ca fiu stâng pentru -) și o sumă (=> **+** ca fiu drept pentru -)

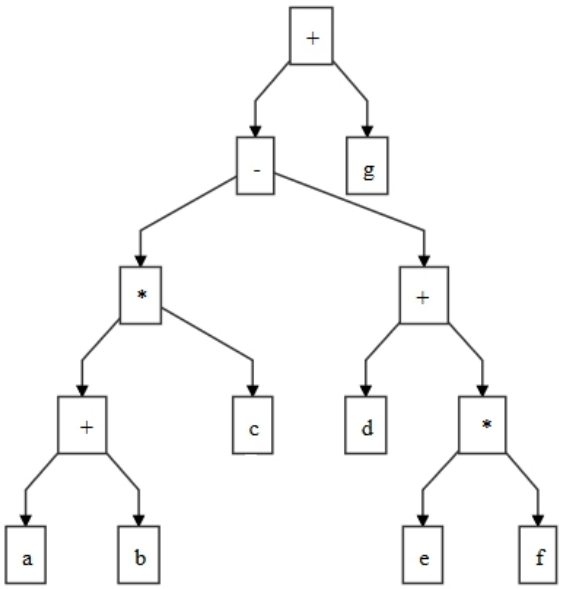
- Produsul se efectuează între o sumă (=> **+** ca fiu stâng pentru * de pe nivelul 2) și c (=> **c** ca fiu drept pentru * de pe nivelul 2), iar suma se efectuează între d (=> **d** ca fiu stâng pentru + de pe nivelul 2) și un produs (=> ***** ca fiu drept pentru + de pe nivel 2)
- Suma se efectuează între a (=> **a** ca fiu stâng pentru + de pe nivelul 3) și b (=> **b** ca fiu drept pentru + de pe nivelul 3), iar produsul se efectuează între e (=> **e** ca fiu stâng pentru * de pe nivelul 3) și f (=> **f** ca fiu drept pentru * de pe nivelul 3)

Observații: S-a considerat rădăcina ca fiind pe nivelul 0.

Dacă s-ar fi considerat operatorul **-** în rădăcină ar fi fost necesară transcrierea expresiei ca (a + b) * c - [(d + e * f) - g].

Ce secvență se obține prin parcurgerea în postordine a arborelui obținut?

Observație: Parcurgerea în **post**ordine presupune parcurgerea rădăcinii **după** parcurgerea celor (cel mult) doi fii (stâng și drept).

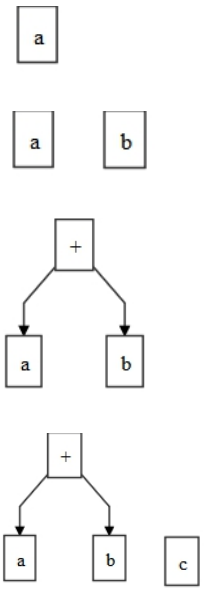


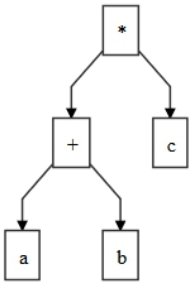
- ✓ Parcurgând arborele în postordine, se obține secvența: a b + c * d e f * + - g +, deci întocmai forma postfixată a expresiei.

Ideea algoritmului:

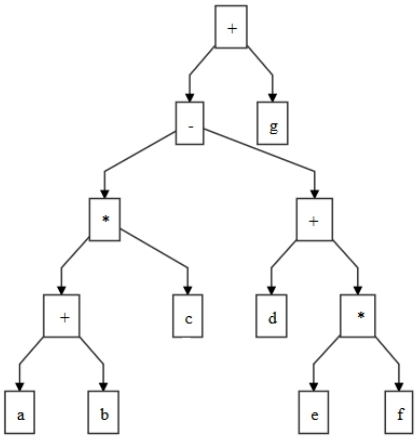
- Folosind o **stivă** care conține adresele nodurilor din arbore, construim arborele de jos în sus, astfel:
  - Se parcurge expresia în FPP
    a. Dacă întâlnim un operand, îl adăugăm în stivă
    b. Dacă întâlnim un operator
       i. Scoatem ultimul element din stivă, care va deveni fiul **drept**
       ii. Scoatem penultimul element din stivă, care va deveni fiul stâng
       iii. Creăm un nod având ca informație utilă operatorul curent, iar ca descendenți elementele obținute la pașii i. și ii.
       iv. Adăugăm nodul creat în stivă
  - (Pointerul la) rădăcina arborelui va fi ultimul (și singurul) element rămas în stivă după parcurgerea expresiei în FPP

Cum va "evolua" conținutul stivei odată cu parcurgerea expresiei în FPP și aplicarea algoritmului descris anterior?





… … ….



Nod:
e:TElement
st, dr: ↑Nod

AB:
răd: ↑Nod

❖ Stiva folosită va avea elemente de tip ↑Nod și vom folosi operațiile stivei:

- creează
- adaugă
- Șterge

```
Subalgoritm creează (Epost, arb):
     creează (s) //creăm o stivă, folosindu-ne de constructorul din
     interfața TAD Stivă
     pentru fiecare e din Epost execută //parcurgem expresia și
          alocă (nou) //pentru fiecare element, creăm un nou nod
          [nou].e ← e  //având-l ca informație utilă
          dacă e este operand atunci //dacă este operand, înseamnă că
va fi frunză în AB, deci nu are fii
               [nou].st ← NIL
```

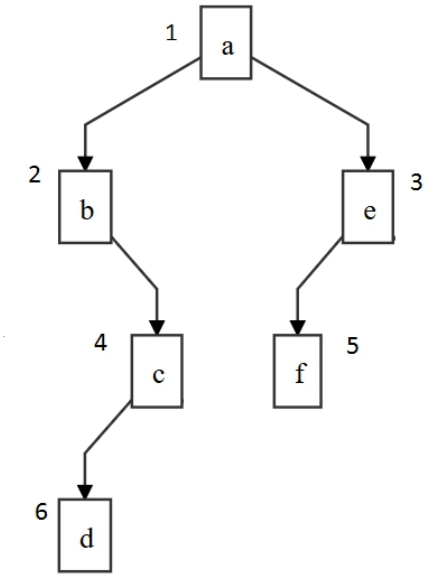
```
                [nou].dr ← NIL
            altfel  //altfel îi extragem fiii din stivă
                șterge(s, p1)
                șterge(s, p2)
                [nou].st ← p2 //stabilim legăturile cu aceștia
                [nou].dr ← p1
            sf_dacă
            adaugă (s, nou) //și îl adăugăm în stivă
    sf_pentru
    șterge(s, p) //în final, dacă expresia este corectă, stiva
    arb.răd ← p  //va conține doar pointerul la nodul rădăcină
sf_subalgoritm
```

- **Observație:** Dacă dorim să folosim interfața AB în implementare, stiva va conține elemente de tipul AB și se vor folosi operații din interfața AB, algoritmul rescriindu-se astfel:

```
Subalgoritm creează (Epost, arb):
      creează (s)
      pentru fiecare e din Epost execută
            dacă e este operand atunci
                creeazăFrunză(ab, e)
            altfel
                șterge(s, p1)
                șterge(s, p2)
                creeazăArbore(ab, p2, e, p1)
            sf_dacă
                adaugă(s, ab)
      sf_pentru
      șterge(s, arb)
sf_subalgoritm
```

- **Observație:** Specificațiile operațiilor din interfața AB se găsesc în cursul 11.

2. **Să se genereze tabelul corespunzător arborelui, numerotându-se nodurile pe nivele, de la stânga la dreapta, precum în exemplul următor.**



	1	2	3
	Informație utilă	**Index fiu stâng**	**Index fiu drept**
1	a	2	3
2	b	0	4
3	e	5	0
4	c	6	0
5	f	0	0
6	d	0	0

- **Observație**: Inexistența unui fiu stâng / drept se marchează cu 0.

  - Împărțim soluția în 2 funcții: *numerotare*, *parcurgere*
  - Pentru numerotare, presupunem că tipul Nod are un câmp nr: Întreg, în care vom reține numărul asociat nodului. Astfel, reprezentarea devine:

Nod:
e:TElement
st, dr: ↑Nod
nr: Întreg

AB:
răd: ↑Nod

✓ Pentru a parcurge AB pe nivele (pe lățime) vom folosi o coadă în care vom reține nodurile

```
Subalgoritm numerotare (arb,k:
    k ← 0 //inițializăm numărul nodurilor din arbore
    creează(c) //creăm o coadă
```

```
    dacă arb.răd ≠ NIL atunci //în cazul în care arborele este nevid
        adaugă(c, arb.răd) //începem prin a îi adăuga rădăcina în
coadă
        k ← 1 //incrementăm k
        [arb.răd].nr ← k //și numerotăm nodul rădăcină
    sf_dacă
    câttimp (¬ vidă (c)) execută //cât timp mai avem elemente în
coadă
        șterge (c, p) //ștergem elementul cel mai devreme adăugat în
coadă
        dacă ([p].st ≠ NIL) atunci //verificăm dacă are fiu stâng
          k ← k + 1 //în caz afirmativ,
          [[p].st].nr ← k //îl numerotăm
          adauga(c, [p].st) //și îl adăugăm în coadă
        sf_dacă
        dacă ([p].dr ≠ NIL) atunci //procedăm similar
     k ← k + 1  //pentru fiul drept
          [[p].dr].nr ← k
          adauga(c, [p].dr)
       sf_dacă
    sf_câttimp
sf_subalgoritm



subalgoritm parcurgere(p, T):

     dacă (p ≠ NIL) atunci  //dacă arborele referit de p este nevid
        T[[p].nr, 1] ← [p].e //completăm prima coloană a rândului
corespunzător nodului rădăcină cu informația utilă din acesta
        dacă ([p].st ≠ NIL) atunci //în cazul în care nodul
rădăcină are fiu stâng
            T[[p].nr, 2] ← [[p].st].nr //completăm cea de-a doua
coloană a rândului corespunzător cu numărul asociat fiului stâng
        altfel
            T[[p].nr, 2] ← 0 //altfel, completăm cea de-a doua
coloană a rândului corespunzător cu 0 indicând, astfel, inexistența
fiului stâng
        sf_dacă
  dacă ([p].dr ≠ NIL) atunci //în cazul în care nodul rădăcină are
fiu drept
            T[[p].nr, 3] ← [[p].dr].nr //completăm cea de-a treia
coloană a rândului corespunzător cu numărul asociat fiului drept
        altfel
            T[[p].nr, 3] ← 0 //altfel, completăm cea de-a treia
coloană a rândului corespunzător cu 0 indicând, astfel, inexistența
fiului drept
        sf_dacă
```

```
        parcurgere([p].st,   T)//apelăm,   recursiv,   subalgoritmul
pentru parcurgerea fiului stâng  (OBS: dacă nu exista, nu va fi
respectata condiția de continuitate verificată la intrarea în
subalgoritmul recursiv)
        parcurgere([p].dr,   T)  //apelăm,  recursiv,  subalgoritmul
pentru parcurgerea fiului drept
    sf_dacă
sf_subalgoritm
```

- **Observație:** În cele ce urmează, descriem în Pseudocod subalgoritmul principal care ulterior numerotării, creează un tabel de dimensiune corespunzătoare și apelează subalgoritmul recursiv de parcurgere.

```
subalgoritm principal(arb, T, k) este:
    numerotare(arb, k)
    @creăm T cu k linii și 3 coloane
    parcurgere(arb.răd, T)
sf_subalgoritm
```

Cum putem adapta soluția în cazul în care nu ne dorim existența unui câmp *nr* în reprezentarea nodurilor AB?

✓ Dacă dorim rezolvarea problemei printr-un singur subalgoritm (de numerotare + parcurgere), putem pune în coadă perechi <nod:↑Nod, nr:Întreg (număr asociat nodului)>.

✓ Dacă dorim descompunerea soluției în 2 subalgoritmi, subalgoritmul *numerotare* poate să creeze un Dictionar cu elemente <nod:↑Nod, nr:Întreg)> sau <arb:AB, nr:Întreg>, care să fie transmis subalgotimului *parcurgere*.

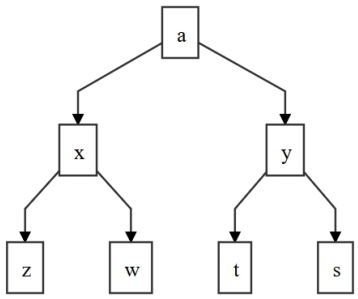
Cum putem adapta soluția în cazul în care nu ne dorim ca subalgoritmul *parcurgerea* să fie recursiv?

✓ Putem folosi o coadă sau o stivă pentru parcurgere, având în vedere că, odată numerotate nodurile conform ordinii de parcurgere în lățime / pe nivele, ordinea de parcurgere pentru completarea tabelului devine irelevantă.

3. **Se dă arborele genealogic al unei persoane, incluzând strămoșii până la generația a *n*-a, arborescența stângă reprezentând linia maternă, iar cea dreaptă linia paternă. Presupunem că rădăcina este de gen feminin.**

a. Să se afișeze toate persoanele de sex feminine (presupunând că rădăcina este de gen feminin)

Care este răspunsul așteptat pentru arborele genealogic exemplificat ulterior?



✓ **a, x, z, t**

Semnificație:

- Mama domnișoarei *a* este doamna *x*, iar tatăl ei este domnul *y*
- Bunica domnișoarei *a,* din partea mamei, este doamna *z,* iar bunicul din partea mamei este domnul *w*
- Bunica domnișoarei *a,* din partea tatălui, este doamna *t,* iar bunicul din partea tatălui este domnul *s*

b. Să se afișeze toți strămoșii de gradul k (se consideră gradul rădăcinii ca fiind 0)

Care este răspunsul așteptat pentru arborele genealogic exemplificat anterior, în cazul în care k=2?

✓ **z, w, t, s**

✓ a) Se parcurge arborele, folosind o coadă (sau o stivă) și se tipăresc, pe lângă rădăcină, doar fii stângi. De pildă, subalgoritmul poate fi descris astfel:

```
Subalgoritm genFeminin(arb):
creeaza(c) //creăm o coadă
dacă arb.răd ≠ NIL atunci //în cazul în care arborele este nevid
        adaugă (c, arb.răd) //începem parcurgerea prin a îi
 adăuga rădăcina în coadă
        scrie arb.răd//și tipărim informația utilă din rădăcină
    sf_dacă
```

```
    câttimp ¬vidă(c) execută //cât timp mai avem noduri de
parcurs
        șterge(c, p) //îl ștergem pe cel introdus cel mai
devreme în coadă
        dacă ([p].st ≠ NIL) atunci //iar dacă are fiu stâng,
deci de gen feminin,
            adaugă(c, [p].st)//îl adăugăm în coadă și
            scrie [[p].st].e //îi tipărim informația utilă
        sf_dacă
        dacă ([p].dr ≠ NIL) atunci //dacă are fiu drept, deci
de gen masculin,
            adaugă(c, [p].dr) //doar îl adăugăm în coadă
(deoarece printre strămoșii acestuia putem găsi persoane de gen
feminin)
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

- ✓ b) În cele ce urmează, descriem în Pseudocod o varianta recursivă a subalgoritmului *nivelk* care returnează într-un vector *v* strămoșii de grad *k* dintr-un arbore genealogic *arb*, folosind doar interfața AB, deci fără a intra în detalii de reprezentare.

```
Subalgoritm nivelk(arb, k, v) este
   dacă ¬vid(arb) atunci //dacă (sub)arborele curent este nevid,
     dacă k = 0 atunci //și dacă rădăcina lui este pe nivelul k
în arborele inițial, deci, prin decrementare, k a ajuns 0
          adaugă(v, element(arb)) //adăugam elementul /
informația utilă din aceasta în vector
     altfel //altfel, înseamnă că nu am ajuns încă la nivelul k,
deci este nevoie să mai "coborâm" în arbore
          dacă ¬vid(stâng (arb)) atunci //așadar, dacă avem
subarbore stâng, reaplicăm, recursiv, subalgoritmul pe acesta,
decrementând nivelul
               nivel(stâng (arb), k-1, v)
          sf_dacă
          dacă ¬vid(drept (arb)) atunci //analog pentru
subarborele drept
               nivel(drept (arb), k-1, v)
          sf_dacă
     sf_dacă
  sf_dacă
sf_subalgoritm
```

- **Observație:** Dacă ne dorim tipărirea elementelor din vector (desigur, s-ar fi putut face în algoritmul anterior), descriem, în cele ce urmează, subalgoritmul principal care, ulterior inițializării vectorului rezultat, apelează subalgoritmul recursiv anterior, iar apoi afișează conținutul vectorului rezultat.

```
Subalgoritm strămoși (arb, k, v) este:
    creeaza (v) // inițializăm un vector vid
    nivelk (arb, k, v)
    pentru i ← 1, dim(v) execută //tipărim conținutul
vectorului rezultat
        scrie element(v, i)
    sf_pentru
sf_subalgoritm
```

Cum putem rezolva cerința b) printr-un algoritm nerecursiv?

✓ Folosindu-ne de o coadă (sau stivă) în care să adăugăm perechi <p:↑Nod, nivel:Întreg> sau <arb:AB, nivel:Întreg>.

4. **Să se creeze un arbore pe nivele. Se dau informațiile astfel:**
   Rădăcina: 1
   Descendenții lui 1: 2, 5
   Descendenții lui 2: 0, 3
   Descendenții lui 5: 6, 0
   Descendenții lui 3: 4, 0
   Descendenții lui 6: 0, 0
   Descendenții lui 4: 0, 0
   Presupunem că marcăm cu 0 inexistența unui nod.

✓ Folosindu-ne de o **coadă**, descriem în continuare un subalgoritm `creeazăNivele`, reprezentând o soluție pentru această problemă. Acesta folosește următoarea funcție auxiliară:

```
Funcția creeazăNod€:
    @returnează un pointer la un Nod care conține e ca
informație utilă și având cei doi fii NIL
Sf_functie

Subalgoritm creeazăNivele(arb):
    scrie ”Rădăcina:”
```

```
    citește e //citim informația din rădăcina,
    dacă e ≠ 0 atunci //iar dacă aceasta există, deci arb. e nevid
        arb.răd ← creeazăNod(e)//o înglobăm într-un nod
        creează(c) //și creăm o coadă
        adaugă(c, arb.răd) //în care o adăugăm
        câttimp ¬ vidă(c) execută //cât timp mai avem noduri pentru
care să interogăm descendenții
            șterge(c, p) //eliminăm nodul cel mai devreme introdus
în coadă
            scrie ”Descendenții lui ” [p].e //cerem descendenții
lui
            citește e1, e2 //îi citim (ca informații)
            dacă e1 ≠ 0 atunci //dacă există fiu stâng
                p1 ← creeazăNod (e1) //îl înglobăm într-un nod
                [p].st ← p1 //și stabilim legătura de de la părinte
la acesta
                adaugă(c, p1) //urmând să îl adăugăm în coadă
            sf_dacă
          dacă e2 ≠ 0 atunci //similar pentru descendentul drept
                p2 ← creeazăNod (e2)
                [p].dr ← p2
                adaugă(c, p2)
            sf_dacă
        sf_câttimp
    altfel
       arb.răd ← NIL  //altfel, deci dacă arborele este vid, marcăm
(pointerul la) nodul rădăcină cu NIL
    sf_dacă
sf_subalgoritm
```