Structuri de Date
Laboratorul 6: Arbori Binari de Căutare

Dan Novischi, Mihai Nan

5 aprilie 2021

1. Introducere

Scopul acestui laborator îl reprezintă implementarea unui arbore binar de căutare. Obiectivul este acela de a vă familiariza cu lucrul cu arbori binari și de a face o comparație cu structurile ordonate definite în laboratoarele anterioare (mulțimi și liste ordonate).

Laboratorul are 3 cerințe care urmăresc:

- înțelegerea interfeței de lucru cu arborii binari de căutare (BST);
- implementarea funcțiilor aferente interfeței de lucru cu BST;
- implementarea unei funcții care determină cel mai apropiat strămoș comun pentru două noduri având cheile `value1` și `value2`.

2. Arbori binari de căutare

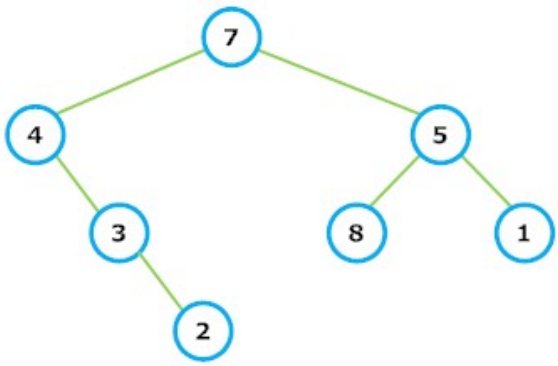


Figura 1: Arbore Binar Simplu

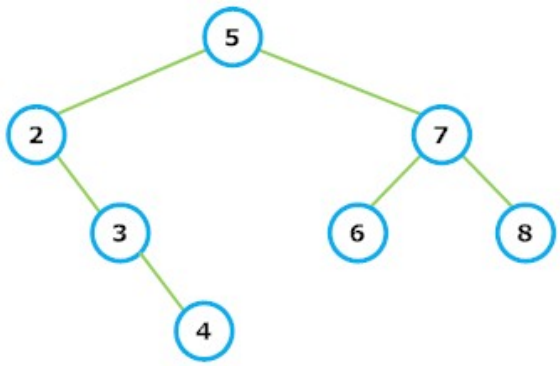


Figura 2: Arbore Binar de Căutare

Arborii sunt structuri de date ierarhice ale căror elemente se numesc **noduri** (similar cu nodurile listelor definite în laboratoarele anterioare). Nodurile stochează valorile datelor de interes (într-un câmp denumit **elem**) și conțin legături aferente pentru organizarea acestora sub formă ierarhică. Astfel, un arbore binar (vezi Figura 1) este o structură de date dinamică

ale cărui noduri pot avea cel mult două legături ierarhice.

Vocabular:

- Cel mai de sus nod din ierarhie se numește *rădăcina* arborelui. În Figura 1 rădăcina este dată de nodul cu $elem = 7$, respectiv în Figura 2 de nodul cu $elem = 5$.
- Dacă un nod x este ierarhic imediat sub alt nod y, atunci x se numește copil al nodului y, iar y se numește părinte al nodului x. În Figura 2 nodul cu $elem = 6$ este copil al nodului cu $elem = 7$. La rândul lui, nodul cu $elem = 7$ este părinte pentru nodurile cu $elem = 6$ și $elem = 8$.
- Nodurile care nu au copii se numesc frunze. În Figura 2, frunzele sunt nodurile cu elementele 4, 6 și 8.

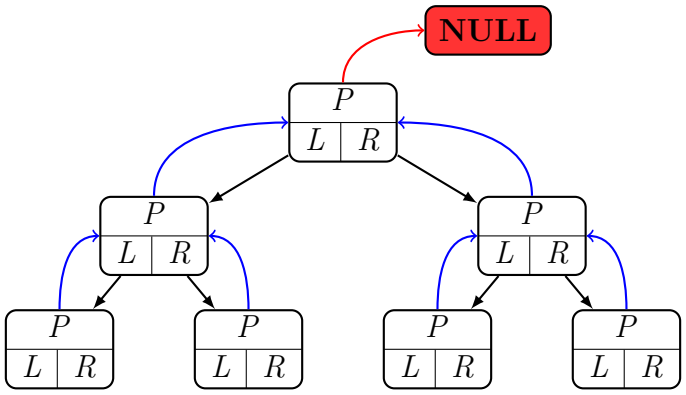
Un **arbore binar de căutare** este un arbore binar ale cărui noduri respectă următoarele relații de ordine:

```
node->left->elem < node->elem; // dacă node->left != NULL
node->right->elem > node->elem; // dacă node->right != NULL
```

Spre exemplu, arborele din Figura 2 este un arbore binar de căutare, deoarece toate elementele acestuia respectă relațiile de ordine de mai sus. Astfel, avem următoarele definiții pentru un abore și nodurile acestuia:

```
1  typedef struct node {
2      Item value;
3      struct node *left;
4      struct node *right;
5      struct node *parent;
6  } TreeNode, *Tree;
```

Reprezentarea grafică a variantei în care pentru fiecare nod păstrăm și un pointer la părintele nodului este următoarea:



Cerința 1 (1p) Analizați antetele funcțiilor definite în fișierul `tree.h` și parcurgeți descrierile lor.

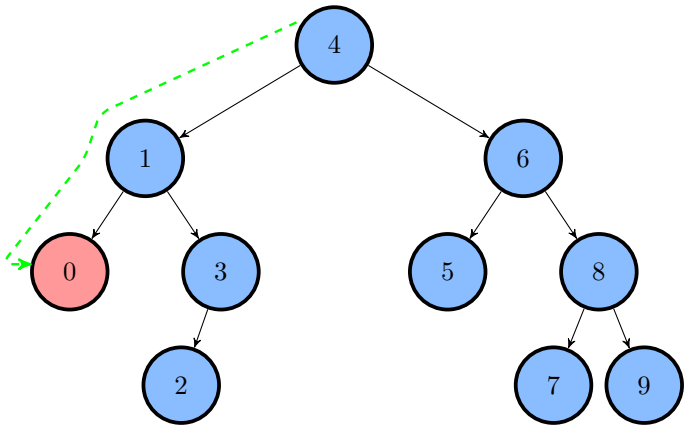
- `createTree`: Primește ca parametru un pointer la un nod ce reprezintă părintele și o valoare și returnează un pointer la un nod frunză ce va conține valoarea primită ca argument și va avea o legătură către părintele primit ca parametru.
- `init`: Similar cu `createTree`, dar este de tip `void`, iar rezultatul o să fie furnizat pe baza primului parametru de tip pointer la `Tree`.
- `isEmpty`: Verifică dacă un arbore este gol sau nu (1 – dacă este arbore vid sau 0 – dacă nu este arbore vid).
- `insert`: Primește un arbore și un `elem`. Funcția inserează valoarea în arbore, dacă ea nu există deja, ținând cont de proprietățile unui arbore binar de căutare. Pentru fiecare nod va trebui să realizați corect legătura cu părintele.
- `contains`: Primește un arbore și un `elem`. Funcția verifică dacă valoarea există în arbore (1 – dacă valoarea există în arbore sau 0 – dacă valoarea nu există în arbore).
- `minimum`: Funcție care determină nodul ce conține elementul minim dintr-un arbore binar de căutare. Funcția va întoarce `NULL` pentru arborele vid.
- `maximum`: Funcție care determină nodul ce conține elementul maxim dintr-un arbore binar de căutare. Funcția va întoarce `NULL` pentru arborele vid.
- `delete`: Funcție care primește ca argumente rădăcina unui arbore și o valoare și șterge nodul care conține respectiva valoare din arbore. Funcția va întoarce rădăcina arborelui rezultat după ștergere.
- `successor`: Funcție ce primește ca parametru rădăcina unui arbore și o valoare existentă în arbore și întoarce succesorul nodului ce conține valoarea indicată.
- `predecessor`: Funcție ce primește ca parametru rădăcina unui arbore și o valoare existentă în arbore și întoarce predecesorul nodului ce conține valoarea indicată.
- `destroyTree`: Funcție care dealocă întreaga memorie alocată pentru un arbore binar. Pointerul la rădăcina arborelui primit ca parametru va pointa către `NULL` după ce se va apela funcția

Cerința 2 (7p) Implementați funcțiile de mai sus în urmatoarea ordine: `init` pe baza funcției `createTree`, `insert`, `isEmpty`, `contains`, `minimum` și `maximum`, `successor`, `predecessor`, `delete`, `destroyTree`.

Indicații:

Determinarea elementului minim

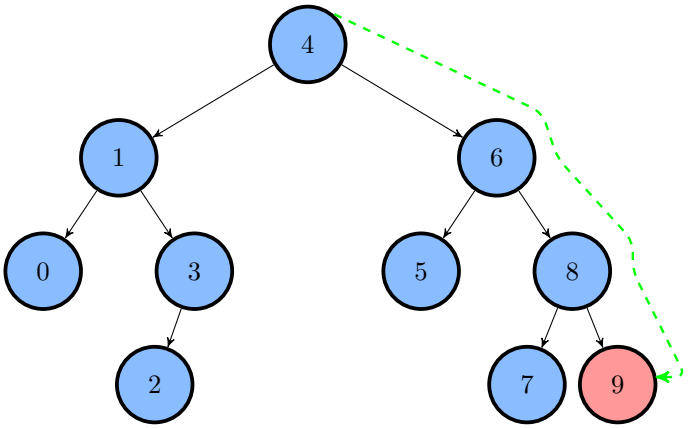
Pentru determinarea valorii minime, ne deplasăm în fiul stâng al rădăcinii. De acolo, în fiul stâng al acelui fiu ș.a.m.d., până când ajungem la un nod care nu mai are fiu stâng. Acest nod conține valoarea minimă din arbore. (Vezi fig. de mai jos)



Algorithm 1 FindMinimum

```
1: procedure FindMinimum(root)
2:     if root ≠ NULL then
3:         while root → left ≠ NULL do
4:             root ← root → left
5:         return root → value
```

Determinarea elementului maxim



Pentru determinarea nodului cu valoarea maximă procedăm similar, ca în cazul determinării minimului, avansând mereu spre fiul drept, până când găsim un nod care nu mai are niciun fiu drept. Acest nod va conține valoarea maximă. (Vezi fig. de mai sus)

Algorithm 2 FindMaximum

```
1: procedure FindMaximum(root)
2:     if root ≠ NULL then
3:         while root → right ≠ NULL do
4:             root ← root → right
5:         return root → value
```

Ștergerea unui nod dintr-un arbore binar de căutare

Operația de ștergere a unui nod presupune următoarele etape: se caută nodul care va fi șters, iar după ce am găsit nodul apar trei cazuri pe care le analizăm separat. Cele trei cazuri posibile sunt următoarele:

I. nodul care urmează să fie șters este o frunză (nu are fii);

II. nodul are un singur fiu;

III. nodul are doi fii.

Algorithm 3 Delete

```
1:  procedure DELETE(root, x)
2:      if root = NULL then
3:          print(Nodul nu a fost gasit)
4:          return root
5:      if root → value > value then
6:          root → left ← delete(root → left, x)
7:      else if root → value < value then
8:          root → right ← delete(root → right, x)
9:      else                                          ▷ Am găsit nodul căutat
10:         if root → left ≠ NULL and root → right ≠ NULL then
11:                                                   ▷ Nodul are 2 fii - cazul III
12:             tmp ← findMinimum(root → right)
13:             root → value ← tmp → value
14:             root → right ← delete(root → right, tmp → value)
15:         else                          ▷ Nodul are un fiu sau este frunză - cazurile I și II
16:             tmp ← root
17:             if root → left ≠ NULL then
18:                 root ← root → left
19:             else
20:                 root ← root → right
21:             free(tmp)
22:     return root
```

Succesorul în inordine

Succesorul unui nod, la traversarea în inordine, se defineşte ca:

1. elementul minim din sub-arborele drept al nodului, dacă nodul are sub-arbore drept.

2. dacă nodul nu are sub-arbore drept, el va fi element maxim într-un sub-arbore. Părintele rădăcinii acestui sub-arbore este nodul succesor.

Predecesorul în inordine

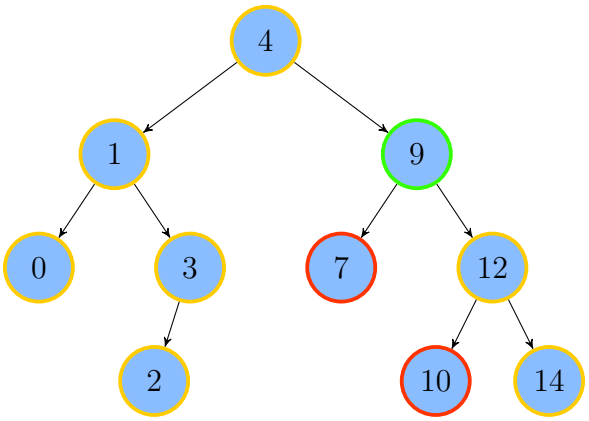
Similar, predecesorul unui nod, la traversarea în inordine, se definește ca:

1. elementul maxim din sub-arborele stâng al nodului, dacă nodul are sub-arbore stâng.
2. dacă nodul nu are sub-arbore stâng, el va fi element minim într-un sub-arbore. Părintele rădăcinii acestui sub-arbore este nodul predecesor.

Cerința 3 (2p) Implementația funcția `lowestCommonAncestor` care determină cel mai apropiat strămoș comun pentru două noduri având cheile `value1` și `value2`.

```
Tree lowestCommonAncestor(Tree root, Item value1, Item value2);
```

Pentru nodurile cu valorile 7 și 10, din arborele de mai jos, avem cel mai apropiat strămoș comun nodul cu valoarea 9.



3. Informații utile

Pentru toate cerințele, putem testa și individual funcțiile implementate. Pentru acest lucru, va trebui să rulăm executabilul `testBST` utilizând argumente în linia de comandă:

- 1 – pentru a testa `insert` și `init` (1 punct);
- 2 – pentru a testa `isEmpty` (0.25 puncte);
- 3 – pentru a testa `contains` (0.25 puncte);
- 4 – pentru a testa `minimum` și `maximum` (1 punct);
- 5 – pentru a testa `successor` (1 punct);
- 6 – pentru a testa `predecessor` (1 punct);
- 7 – pentru a testa `delete` (2 puncte);
- 8 – pentru a testa `lca` (2 puncte).

Observație: Punctajul pentru `destroyTree` se va acorda manual dacă nu există erori sau pierderi de memorie atunci când rulăm cu `valgrind`!

Exemple de testări:

- Testarea funcției `insert`

```
$ ./testBST 1
insert  - Toate testele au trecut!
Toate testele selectate au trecut!
Punctajul obtinut este: 2.00
Teste rulate: 1
```

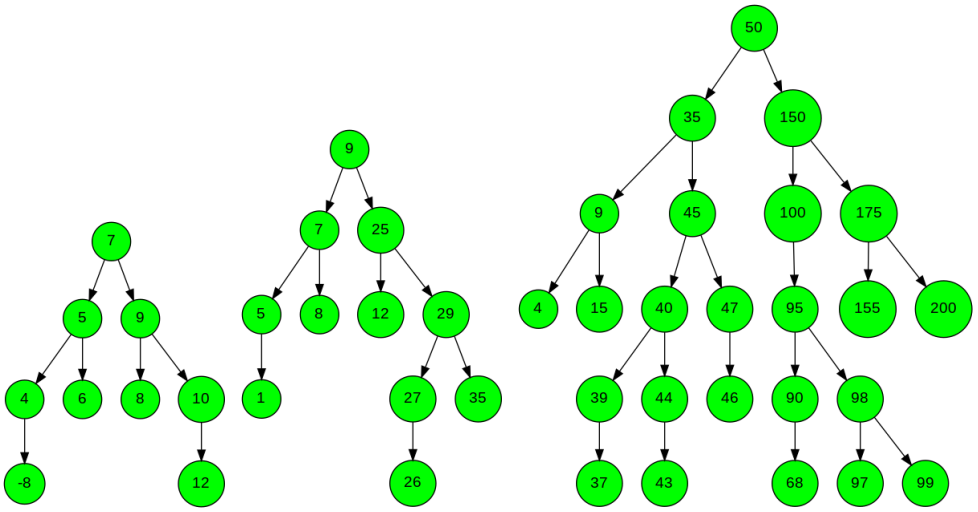
Pentru a fi mai ușor de verificat dacă este corectă implementarea, este implementată o funcție care desenează arborele. Această funcție utilizează utilitarul `graphViz`.

Pentru a putea instala acest utilitar pe Linux, putem folosi comanda:

```
sudo apt-get install graphviz
```

Pentru a putea instala acest utilitar pe Windows, putem descărcă executabilul de la adresa: https://graphviz.org/download/.

Exemple de imagini generate:



- Testarea tuturor funcțiilor folosind valgrind:

```
$ make test
valgrind ./TestBST
==32759== Memcheck, a memory error detector
==32759== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==32759== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==32759== Command: ./TestBST
==32759==
insert      - Toate testele au trecut!
isEmpty     - Toate testele au trecut!
contains    - Toate testele au trecut!
minmax      - Toate testele au trecut!
successor   - Toate testele au trecut!
predecessor - Toate testele au trecut!
delete      - Toate testele au trecut!
lca         - Toate testele au trecut!
Toate testele au trecut! Felicitari!
Punctajul obtinut este: 9.50
Teste rulate: 8
==32759==
==32759== HEAP SUMMARY:
==32759==     in use at exit: 0 bytes in 0 blocks
==32759==   total heap usage: 389 allocs, 389 frees, 95,952 bytes allocated
==32759==
==32759== All heap blocks were freed -- no leaks are possible
==32759==
==32759== For counts of detected and suppressed errors, rerun with: -v
==32759== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```