PA - Arbori

Daniel Chiș - 2021, UPB, ACS, An I, Seria AC

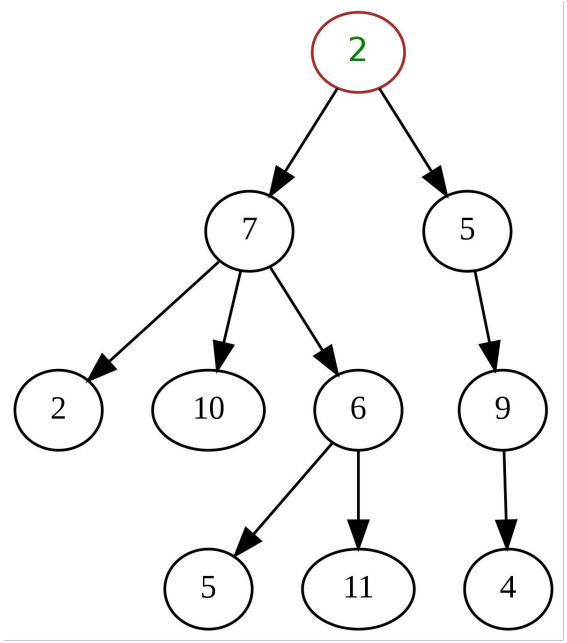
Arbori

Arbori - Trees

Arborii sunt o structură de date realizată cu ajutorul unor noduri folosită pentru a stoca date.

Orice arbore are un nod rădăcină (root node) de la care se începe construcția arborelui. Root are zero sau mai multe noduri copii (child nodes). Fiecare nod copil are la rândul său zero sau mai mulți copii.

Un arbore nu poate să aibă cicluri (altfel s-ar transforma într-un graf).

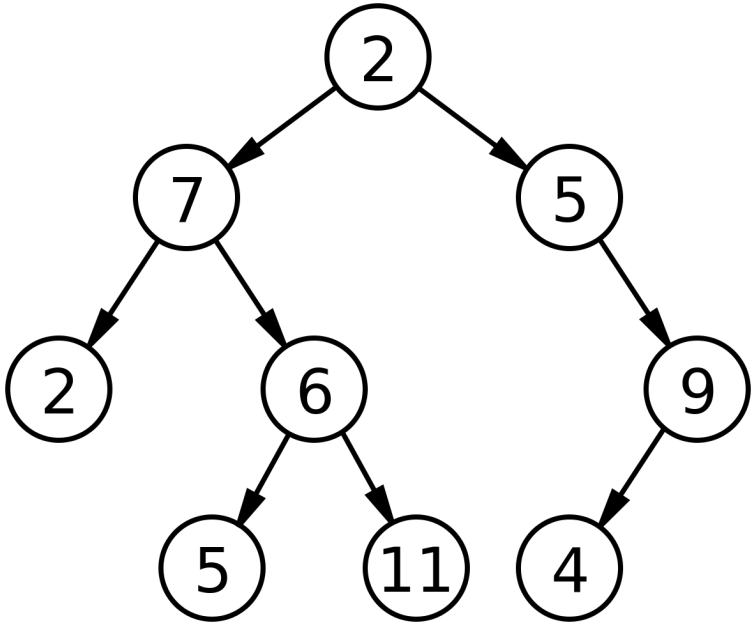


Arbori binari - Binary Trees

Arborii binari sunt o variație de arbori. Aceștia au condiția ca fiecare nod să aibă maxim doi copii.

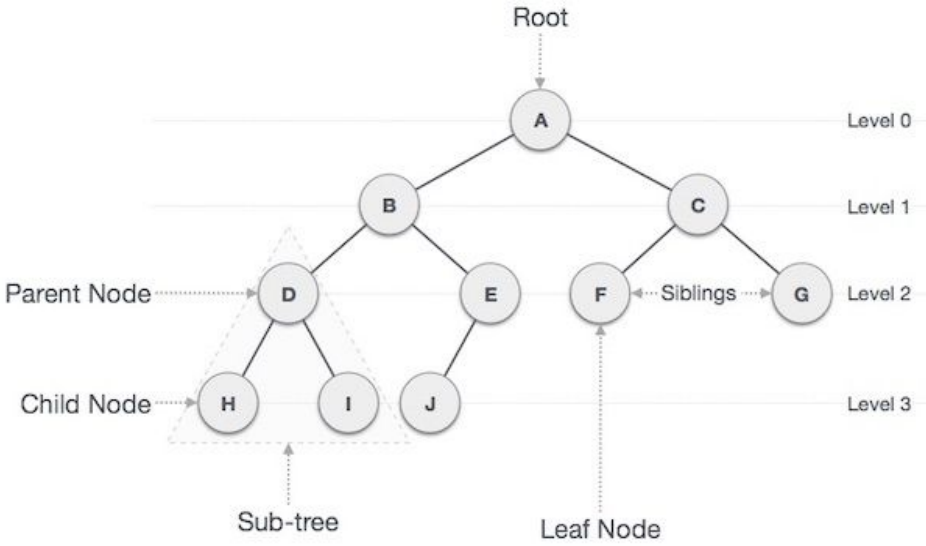
Operația de search este rapidă pentru arborii binari (ca și într-o matrice sortată) și operațiile de insert și delete sunt rapide (ca într-o listă înlănțuită).

```
1  struct node {
2     int data;
3     struct node *leftChild;
4     struct node *rightChild;
5  };
```



Terminologie Arbori

- Root - rădăcină, nod primar
- Parent - părinte, nod care are copii, orice nod are un părinte mai puțin root
- Child - nod care este legat la un altul în jos
- Leaf - frunză, nod care nu are copii
- Subtree - configurație realizată de copiii unui nod
- Visiting - a verifica valoarea unui nod
- Traversing - a trece printr-o succesiune de noduri într-o ordine anume
- Levels - nivel de adâncime al arborelui



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

struct node {
    int data;

    struct node *leftChild;
    struct node *rightChild;
};

struct node *root = NULL;
```

Definire noduri

Insert arbore

```
13 void insert(int data) {
14    struct node *tempNode = (struct node*) malloc(sizeof(struct node));
15    struct node *current;
16    struct node *parent;
17
18    tempNode->data = data;
19    tempNode->leftChild = NULL;
20    tempNode->rightChild = NULL;
21
22    //arborele este gol
23    if(root == NULL) {
24       root = tempNode;
25    } else {
26       current = root;
27       parent = NULL;
28
29       while(1) {
30          parent = current;
31
32          //subarbore stanga
33          if(data < parent->data) {
34             current = current->leftChild;
35
36             //insert la stanga
37             if(current == NULL) {
38                parent->leftChild = tempNode;
39                return;
40             }
41          }  //subarbore dreapta
42          else {
43             current = current->rightChild;
44
45             //insert la dreapta
46             if(current == NULL) {
47                parent->rightChild = tempNode;
48                return;
49             }
50          }
51       }
52    }
53 }
```

```
55 struct node* search(int data) {
56    struct node *current = root;
57    printf("Visiting elements: ");
58 
59    while(current->data != data) {
60       if(current != NULL)
61          printf("%d ",current->data);
62 
63       //subarbore stanga
64       if(current->data > data) {
65          current = current->leftChild;
66       }
67       //subarbore dreapta
68       else {
69          current = current->rightChild;
70       }
71 
72       //nu exista
73       if(current == NULL) {
74          return NULL;
75       }
76    }
77 
78    return current;
79 }
```

Cautare arbore

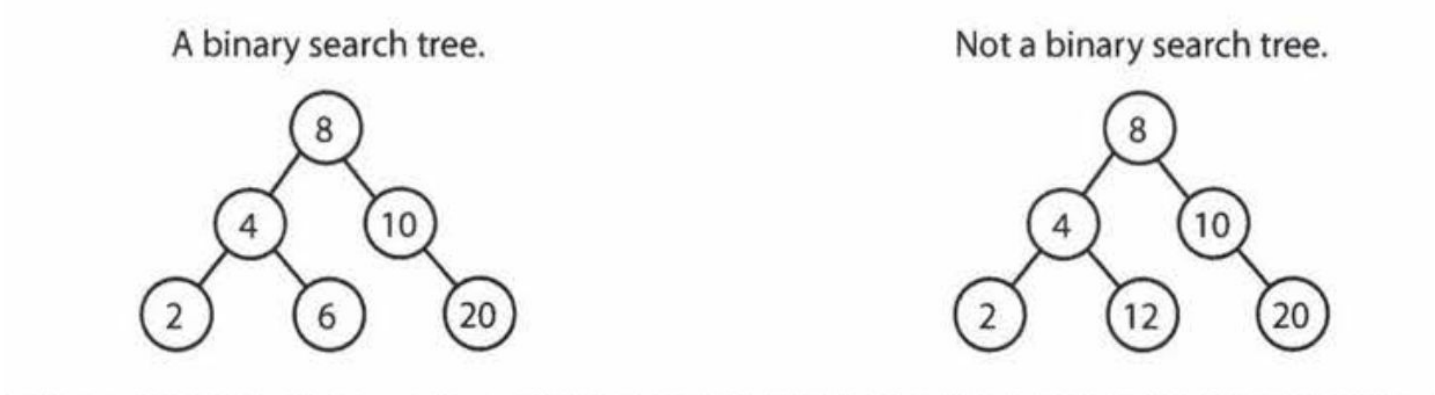
```
82  int main() {
83     int i;
84     int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
85  
86     for(i = 0; i < 7; i++)
87        insert(array[i]);
88  
89     i = 31;
90     struct node * temp = search(i);
91  
92     if(temp != NULL) {
93        printf("[%d] Gasit", temp->data);
94        printf("\n");
95     }else {
96        printf("[ x ] Nu exista (%d).\n", i);
97     }
98  
99     i = 15;
100    temp = search(i);
101 
102    if(temp != NULL) {
103       printf("[%d] Gasit", temp->data);
104       printf("\n");
105    }else {
106       printf("[ x ] Nu exsita (%d).\n", i);
107    }
108 
109    return 0;
110 }
```

Operatii Main arbore

Binary Tree vs Binary Search Tree

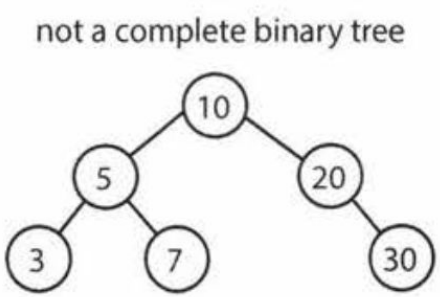
Binary Search Tree reprezintă o variație de Binary Tree, în care fiecare nod se subscrie unei operații de ordonare specifice, și anume:

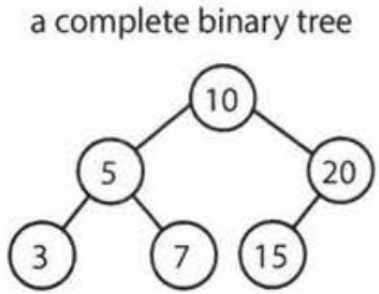
toți descendenții din stânga <= n < toți descendenții din dreapta



Complete Binary Tree

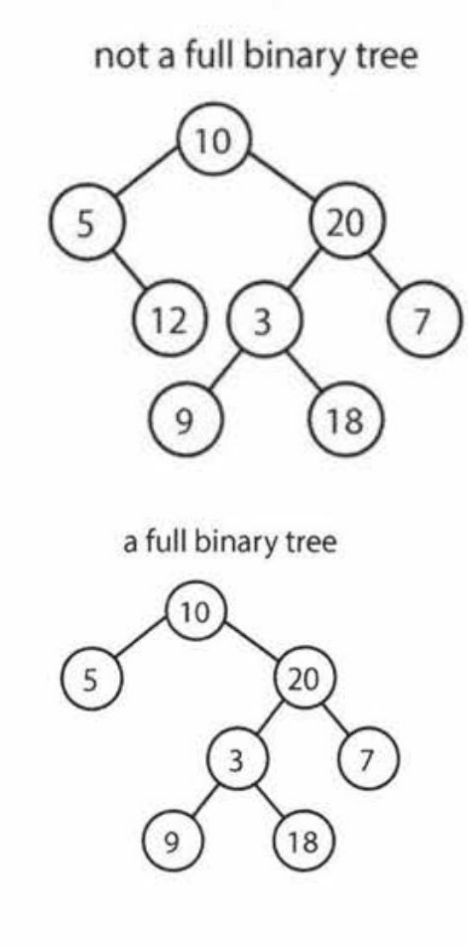
Binary tree în care fiecare level este completat. Excepția poate fi la ultimul nivel unde trebuie completat de la stânga la dreapta.





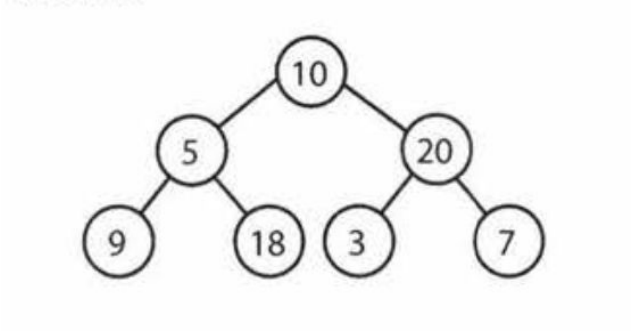
Full Binary Tree

Binary tree în care fiecare nod are fie un zero copii fie 2, niciodată doar unul.



Perfect Binary Tree

Binary tree în care ultimul nivel este completat completat.



Parcurgere arbori

Parcurgerea arborilor, reprezintă procesul de vizitare a tuturor nodurilor din arbore. Deoarece toate nodurile sunt conectate prin link-uri începem de la rădăcină. Există trei tipuri de a parcurge un arbore:

1. In-order
2. Pre-order
3. Post-order

In-order

1. Parcurgere subarborele din partea stângă
2. Se vizitează nodul
3. Se trece la subarborele din partea dreaptă

Parcurgere In-order în Binary Search Tree va aduce nodurile sortate.

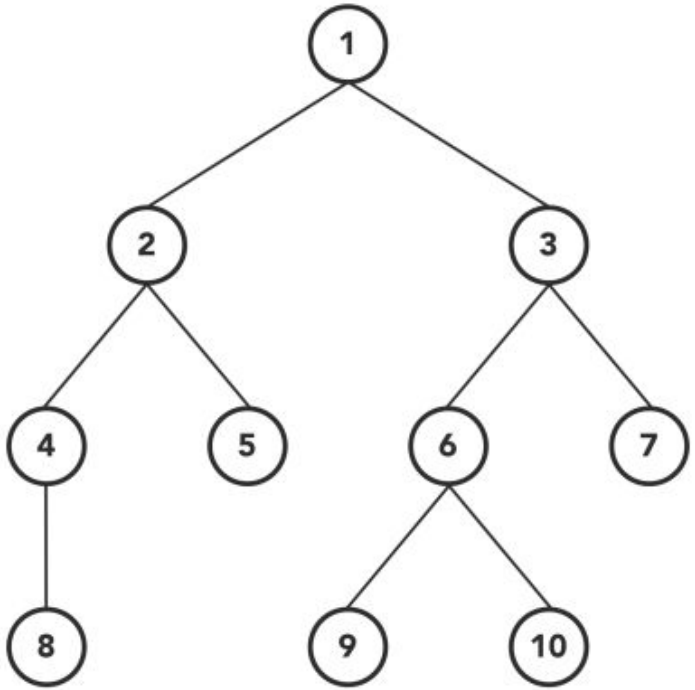
```
1  void inOrderTraversal(TreeNode node){
2      if(node != null){
3          inOrderTraversal(node.left);
4          visit(node);
5          inOrderTraversal(node.right)
6      }
7  }
```

Pre-order

1. Se vizitează nodul
2. Se merge la subarborele din stânga
3. Se merge la subarborele din dreapta

Root e primul vizitat mereu.

```
1  void preOrderTraversal(TreeNode node){
2      if(node != null){
3          visit(node);
4          preOrderTraversal(node.left);
5          preOrderTraversal(node.right)
6      }
7  }
```

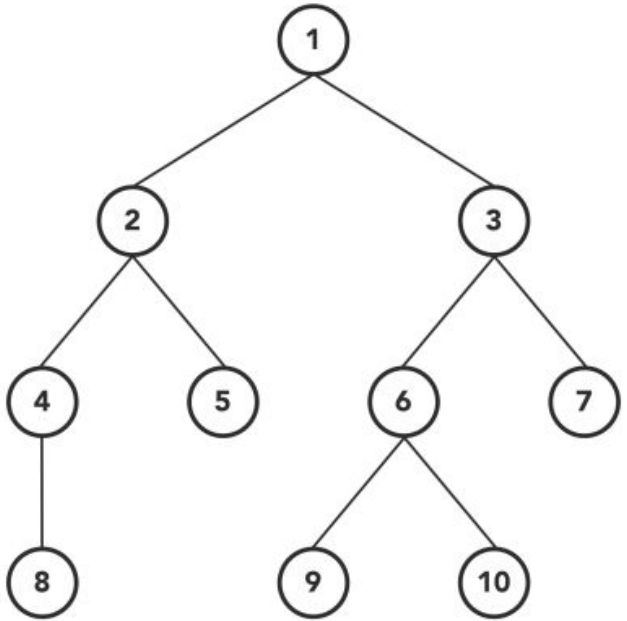


Post-order

1. Se merge la subarborele din stânga
2. Se merge la subarborele din dreapta
3. Se vizitează nodul

Root e ultimul vizitat mereu.

```
1  void postOrderTraversal(TreeNode node){
2      if(node != null){
3          postOrderTraversal(node.left);
4          postOrderTraversal(node.right)
5          visit(node);
6      }
7  }
```



Exerciții

Exerciții

1. Implementați o funcție ca să verificați că un binary tree este echilibrat (doi subarbori ai oricărui nod nu au o diferență de nivel mai mare de 1). **3p**
2. Implementați o funcție prin care să verificați că un binary tree este binary search tree. **3p**
3. Implementați o funcție care să găsească ce mai apropiat strămoș comun al două noduri dintr-un arbore binar. **3p**

Fiecare arbore trebuie și traversat și afișat cu una dintre cele trei tehnici prezentate în laborator. (câte o tehnică pentru fiecare problemă, toate trei trebuie folosite).

Exerciții FIIR

Creați un arbore de minim 4 nivele. Mai intai desenați-l pe o foaie sau pe calculator. După care trebuie traversat și afișat cu cele trei tehnici prezentate în laborator.

Metoda simpla de creat arbori: https://www.geeksforgeeks.org/binary-tree-set-1-introduction/