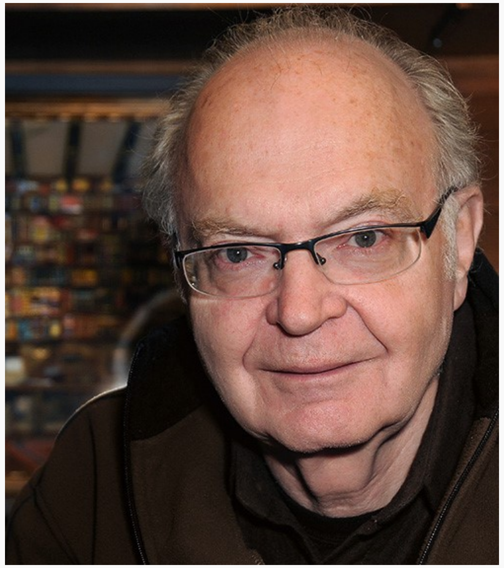
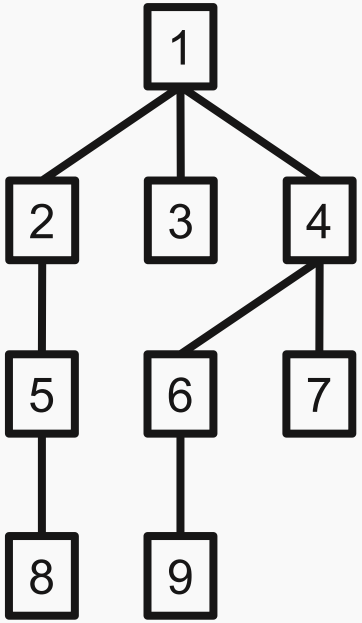
Structuri de date și algoritmi
Arbori - 1

Lect. Dr. Ing. Cristian Chilipirea – cristian.chilipirea@mta.ro

Arbore. Definiție TAoCP Knuth:





Set T de noduri

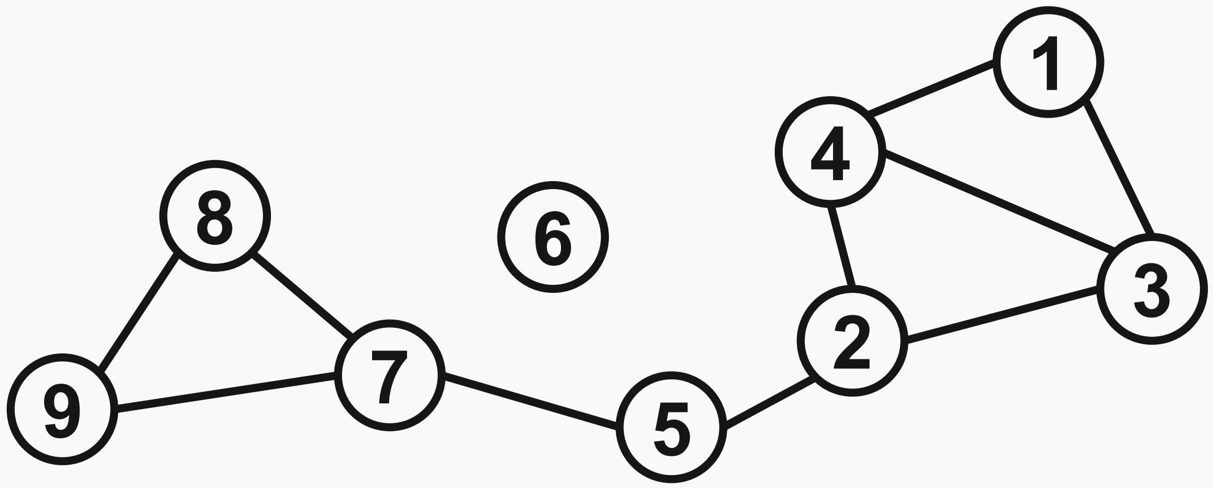
a) Există un nod numit rădăcină $root(T)$

b) Celelalte noduri sunt partiționate în $m \geq 0$ seturi **disjuncte** $T_1, T_2, \ldots, T_m$. Fiecare astfel de set este un arbore. Astfel arborii $T_1, T_2, \ldots, T_m$ sunt subarbori ai rădăcinii.

Graf. Definiție:

Graf $G = (V, E)$. Unde:

- V – setul de noduri – Vertex
- E – setul de muchii – Edges
  - $E \subseteq \{(\mathrm{x}, \mathrm{y}) | (\mathrm{x}, \mathrm{y}) \in V^2 \wedge \mathrm{x} \neq \mathrm{y}\}$



Arbore vs Graf

Un arbore este un graf

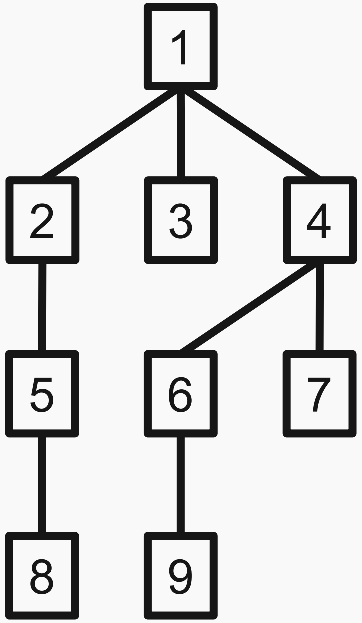
- neorientat - orice muchie poate fi parcursă în ambele direcții
- connex - există o cale, parcurgând muchiile, de la un nod la oricare altul
- fără cicluri - nu există nici o cale de forma

$$(v_0 \to v_1 \to \cdots \to v_n \to v_0)$$

Arbore - stocare

```c
typedef struct treeNode {
    void* value;
    int numChildren;
    struct treeNode** children;
    // struct treeNode *parent;
}treeNode;
```

Vector de pointeri



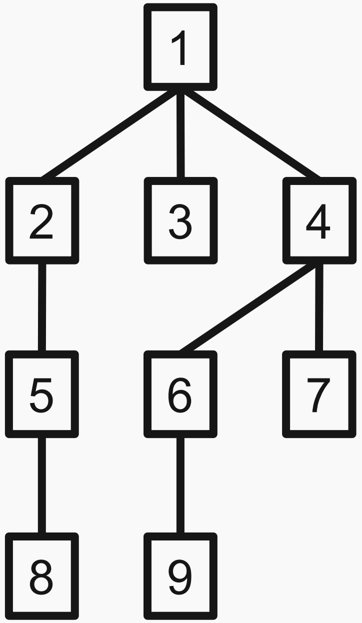
Listă conexiuni

1	1	1	2	4	4	5	6
2	3	4	5	6	7	8	9

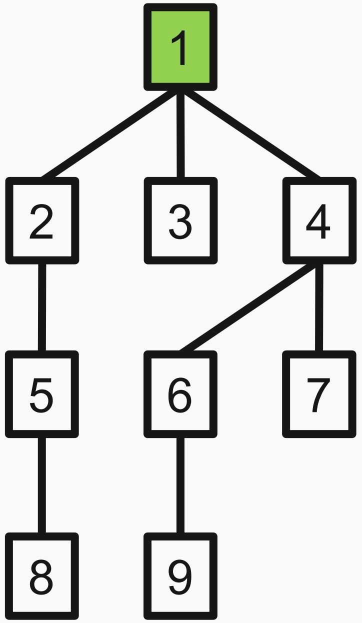
Vector părinți

1	2	3	4	5	6	7	8	9
-	1	1	1	2	4	4	5	6

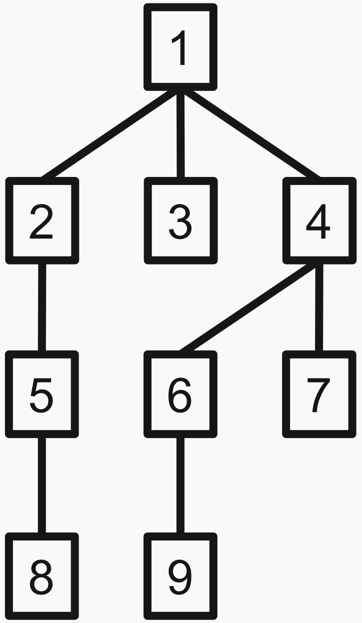
Ca orice alt graf:
ex. Matrice adiacență



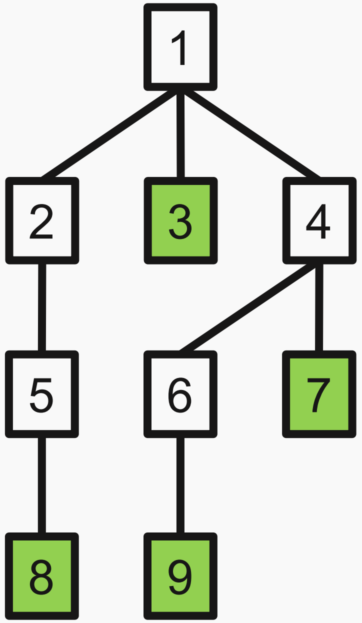
Care este rădăcina arborelui?



Care este rădăcina arborelui?



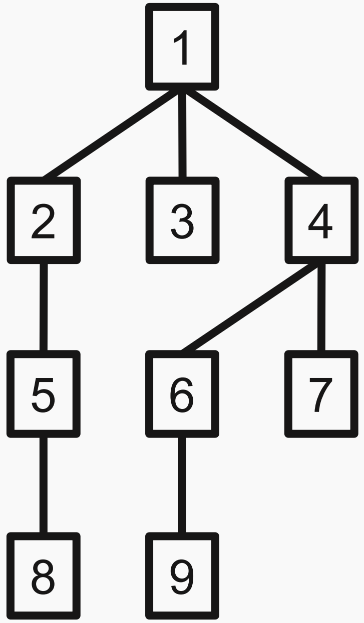
Care sunt frunzele arborelui?



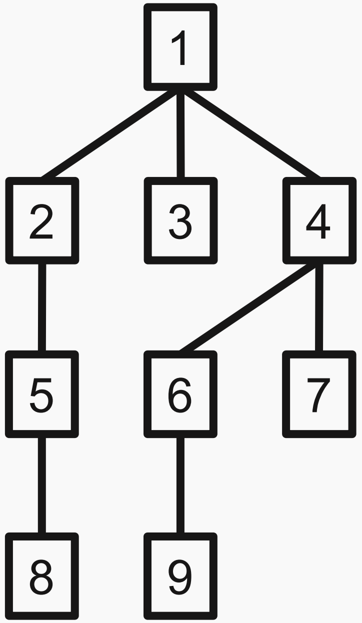
Care sunt frunzele arborelui?

Mărimi arbore

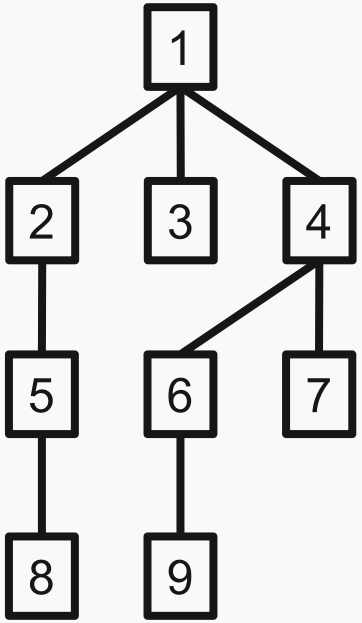
- **Height (Înălțime)** – mărimea celei mai lungi căi de la nodul rădăcină la un nod frunză.
- **Width (Lățime)** – numărul de noduri după un nivel.
- **Degree of a node (gradul unui nod)** – număr copii nod.
- **Degree of tree (gradul arborelui)** – numărul maxim de copii al unui nod din arbore.



Afișarea unui arbore fără recursivitate

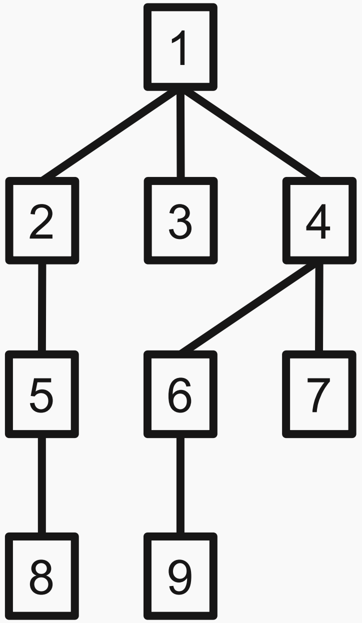


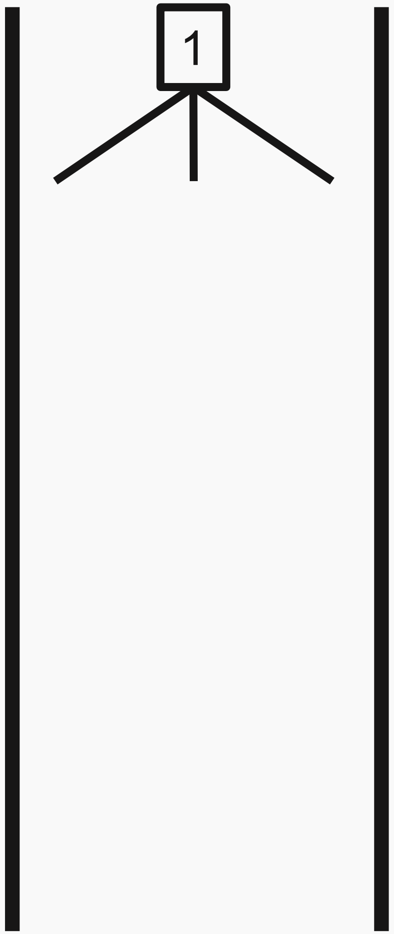
Afișarea unui arbore folosind o coadă



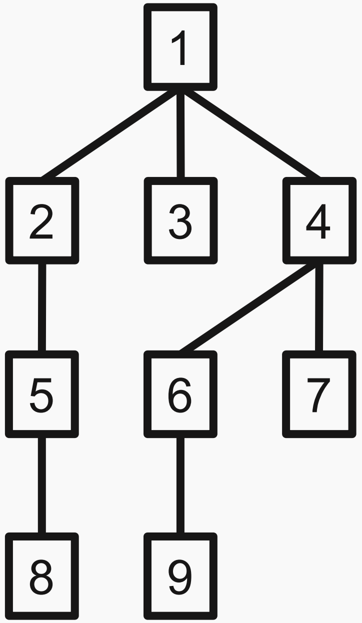
- Adăugăm rădăcina în coadă
- Până ce coada este goală
  - Scoatem un nod din coadă
  - Adăugăm toți copiii în coadă

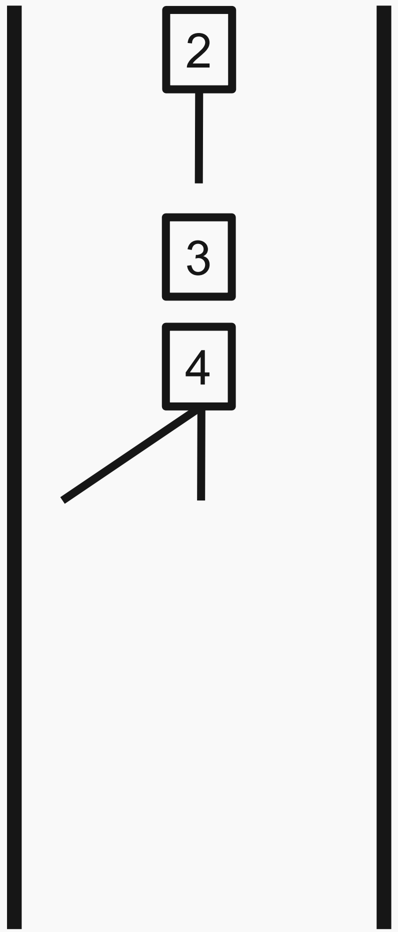
Afișarea unui arbore folosind o coadă





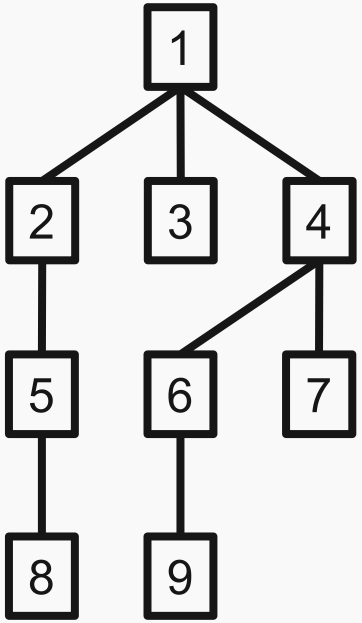
Afișarea unui arbore folosind o coadă

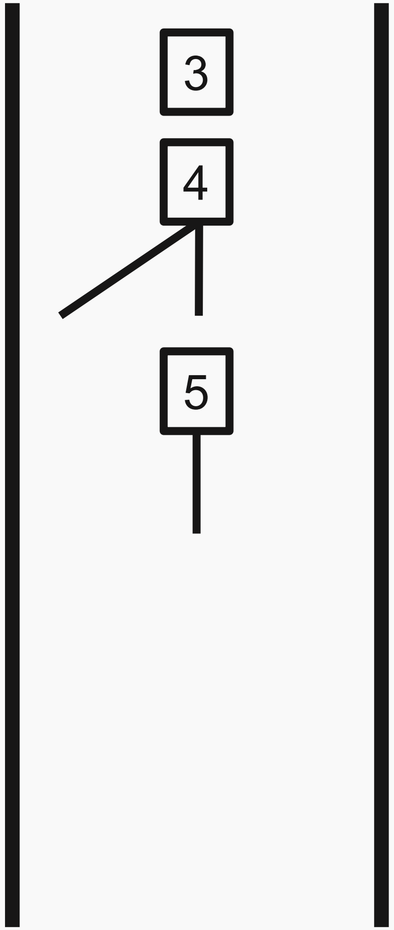




Afișare: 1

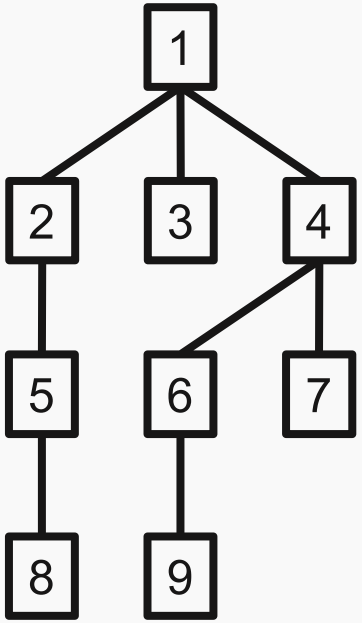
Afișarea unui arbore folosind o coadă

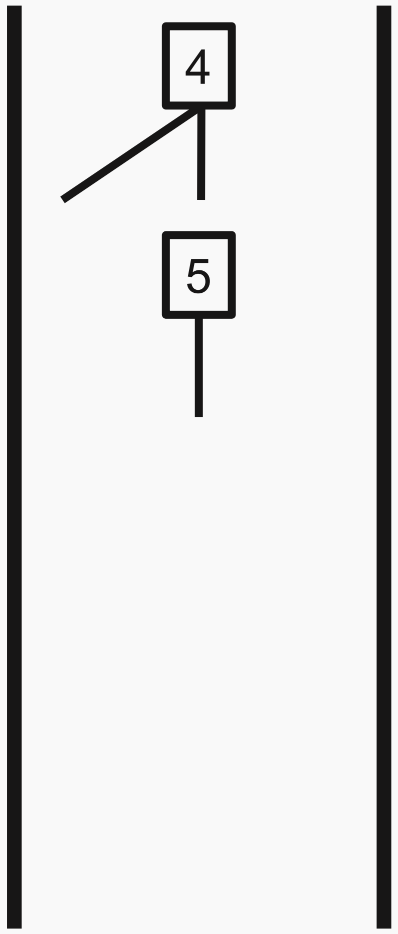




Afișare: 1 2

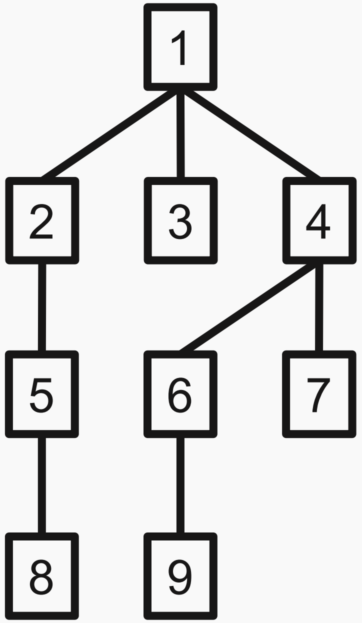
Afișarea unui arbore folosind o coadă

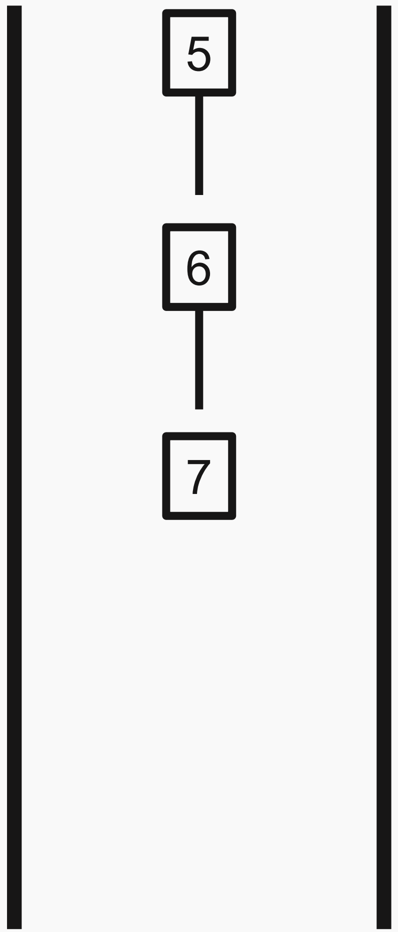




Afișare: | 1 | 2 | 3 |

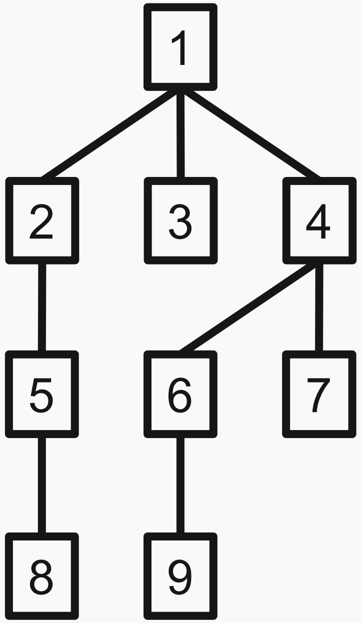
Afișarea unui arbore folosind o coadă

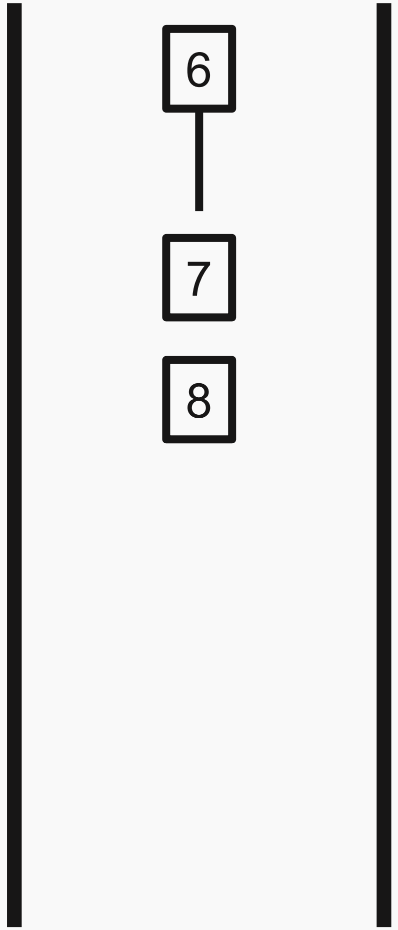




Afișare: | 1 | 2 | 3 | 4 |

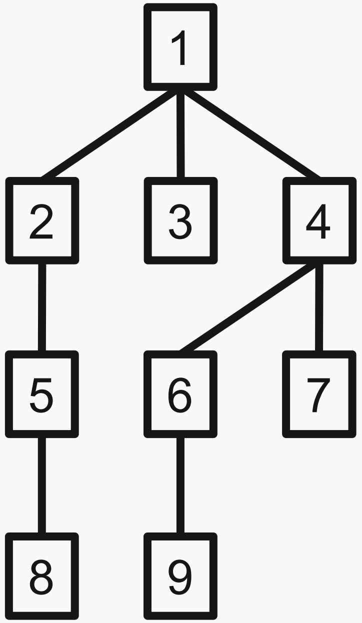
Afișarea unui arbore folosind o coadă

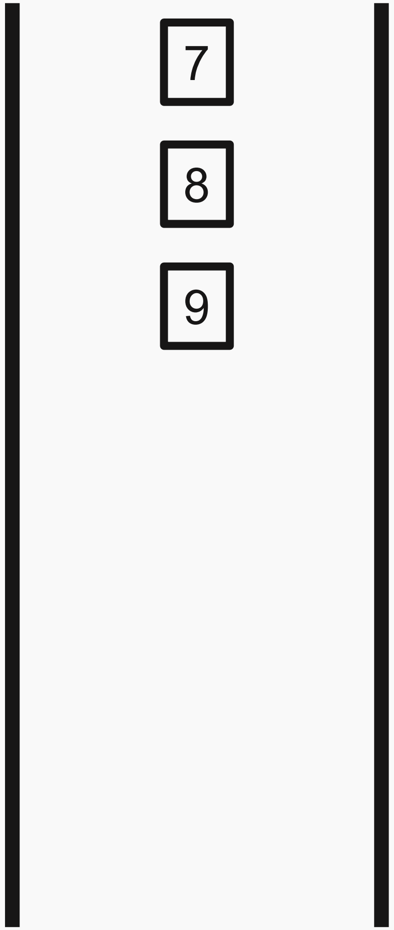




Afișare: | 1 | 2 | 3 | 4 | 5 |

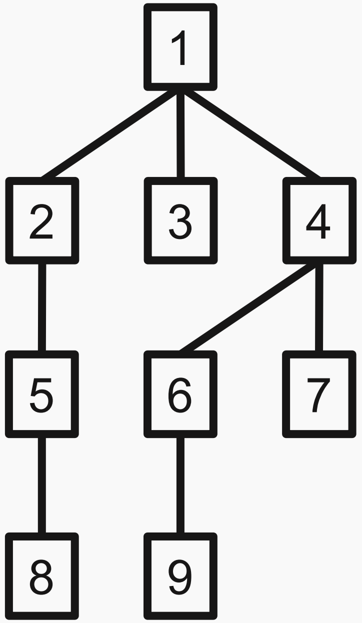
Afișarea unui arbore folosind o coadă

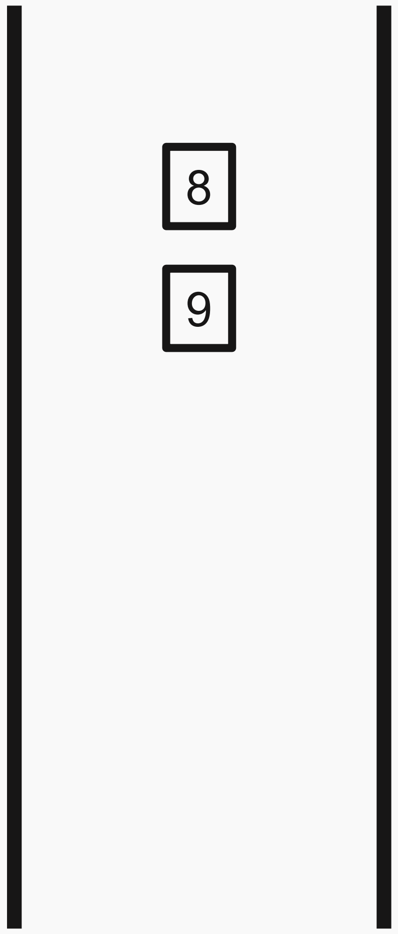




Afișare: | 1 | 2 | 3 | 4 | 5 | 6 |

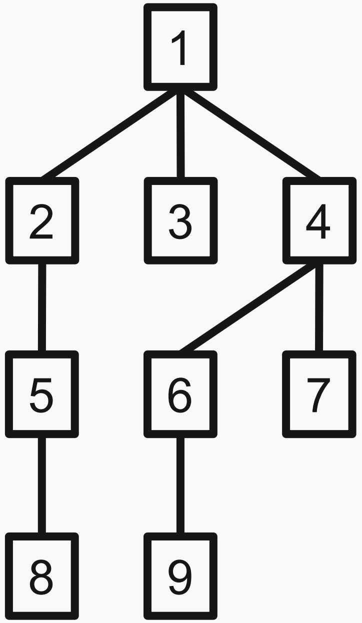
Afișarea unui arbore folosind o coadă

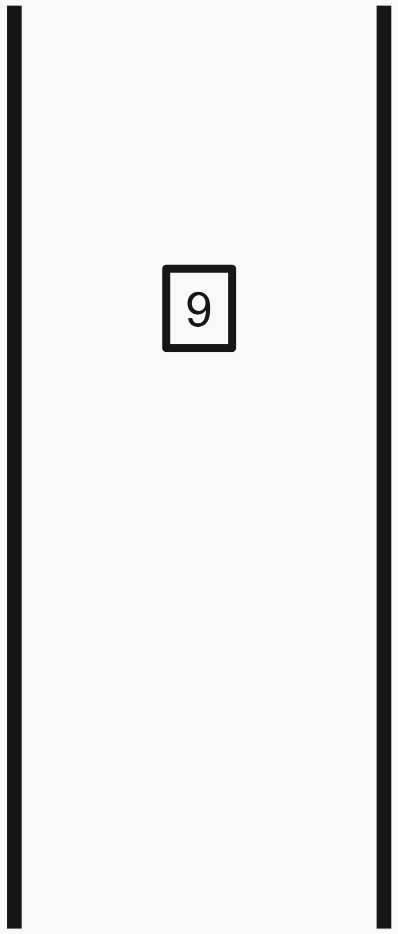




Afișare: | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Afișarea unui arbore folosind o coadă

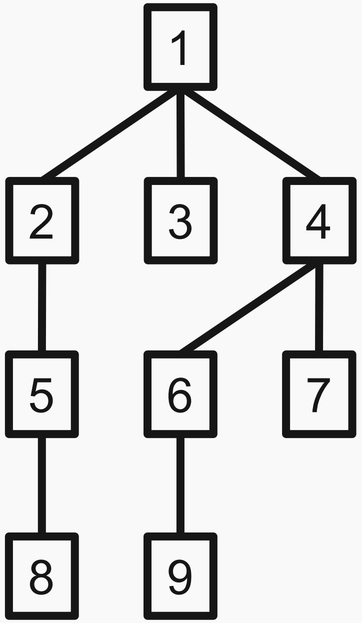




Afișare:

1	2	3	4	5	6	7	8

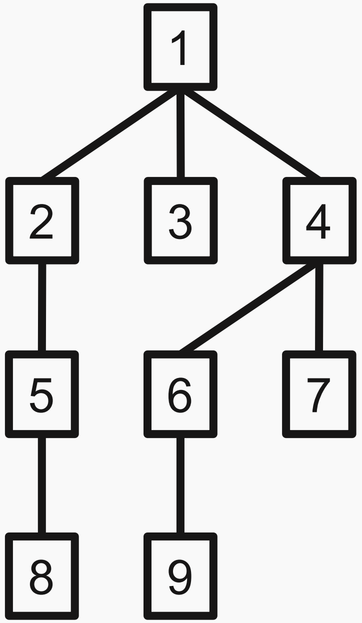
Afișarea unui arbore folosind o coadă



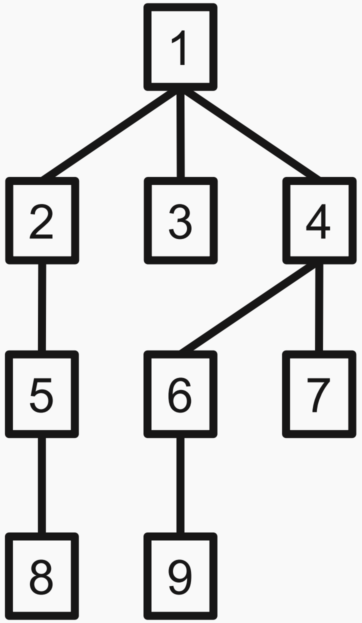
Afișare:

1	2	3	4	5	6	7	8	9

Afișarea unui arbore folosind o stivă



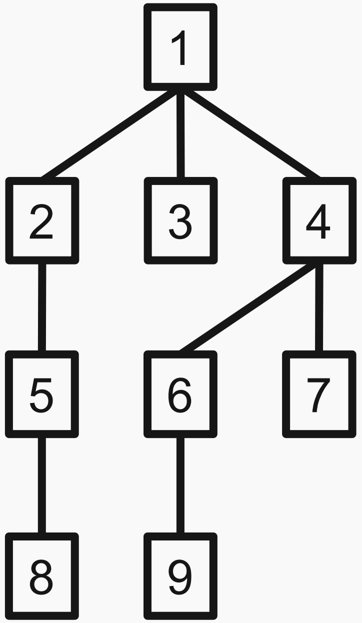
Afișarea unui arbore folosind o stivă

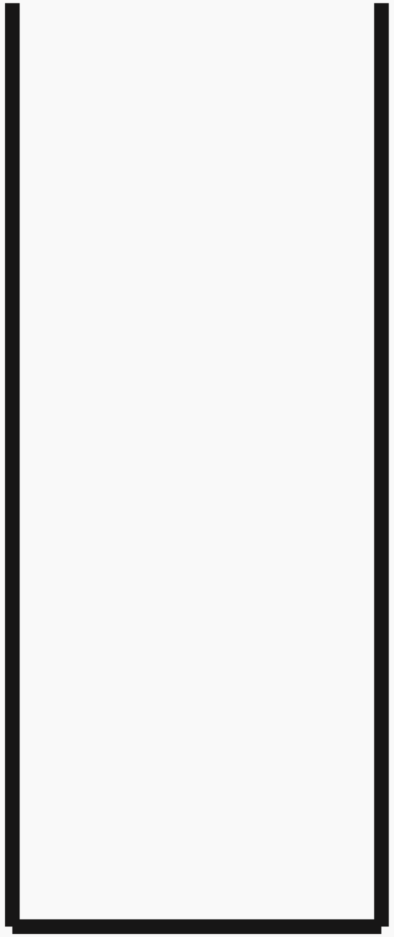


- Adăugăm rădăcina în stivă
- Până ce stiva este goală
  - Scoatem un nod din stivă
  - Adăugăm toți copiii în stivă

Cum va fi afișat arborele?

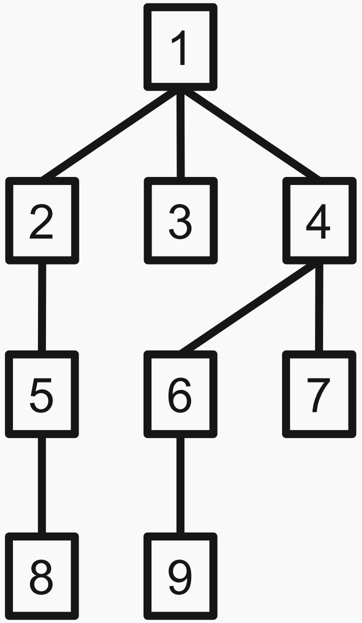
Afișarea unui arbore folosind o stivă

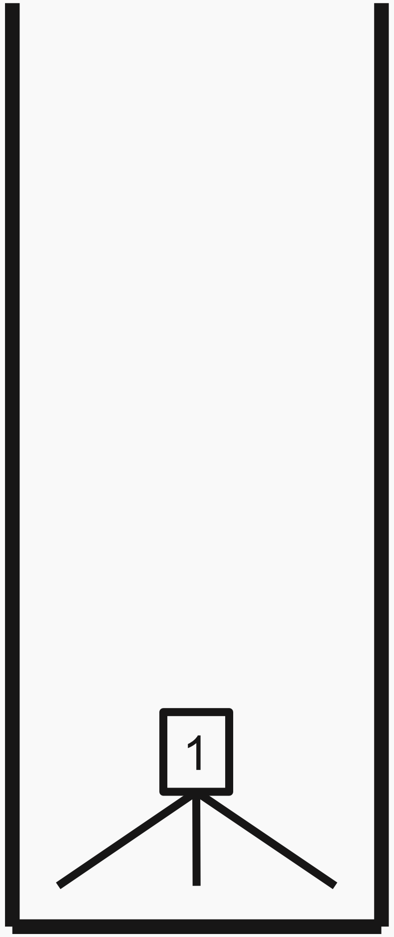




Afișare:

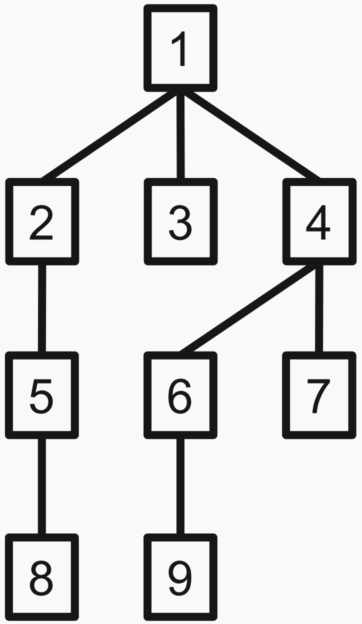
Afișarea unui arbore folosind o stivă

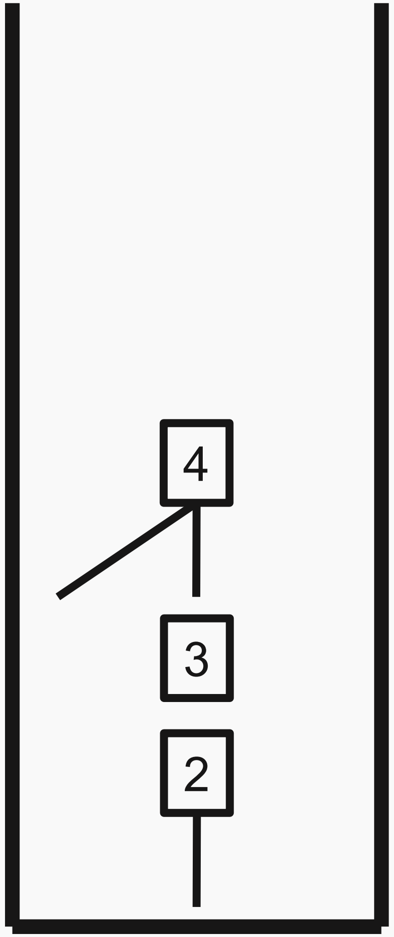




Afișare:

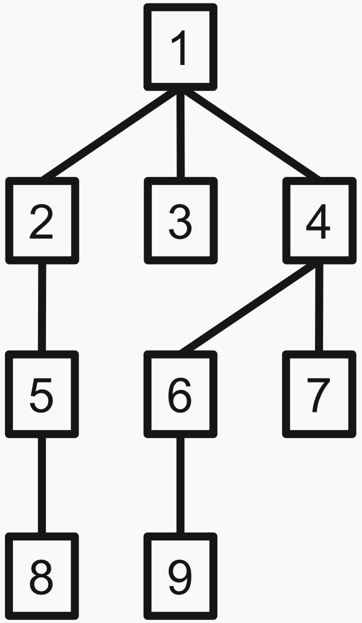
Afișarea unui arbore folosind o stivă

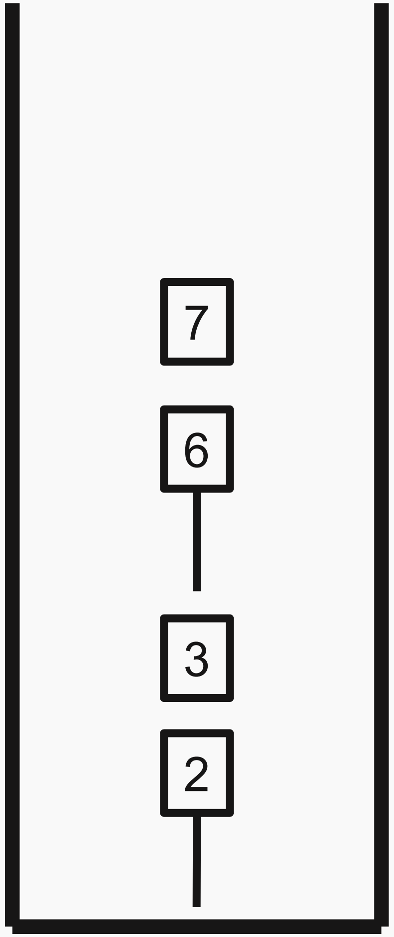




Afișare: 1

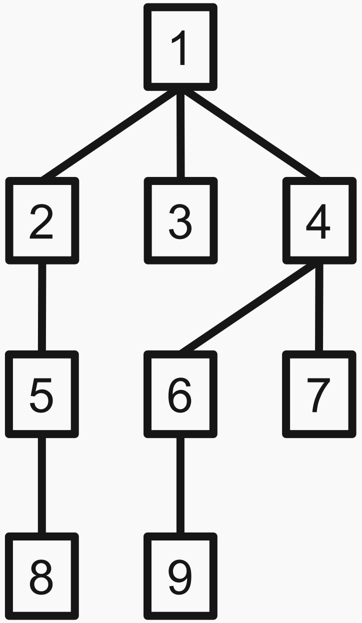
Afișarea unui arbore folosind o stivă

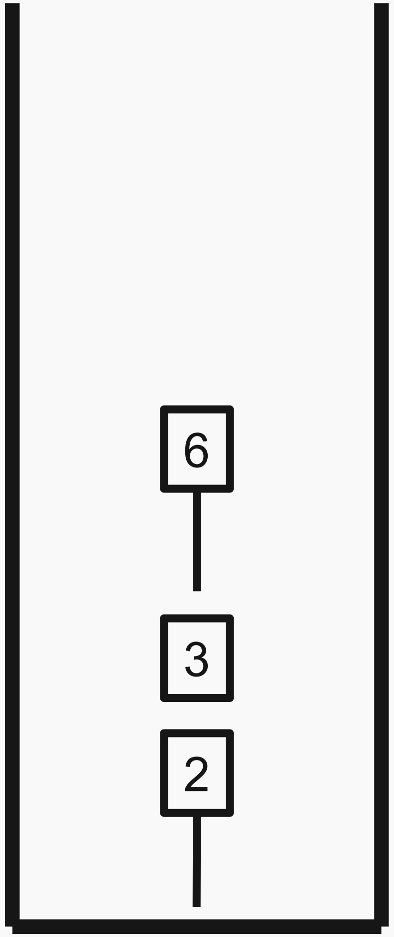




Afișare: 1 4

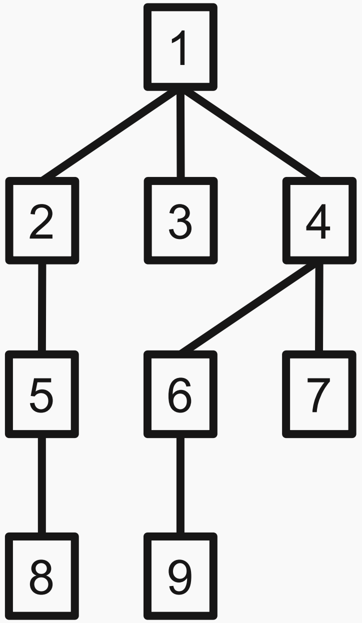
Afișarea unui arbore folosind o stivă

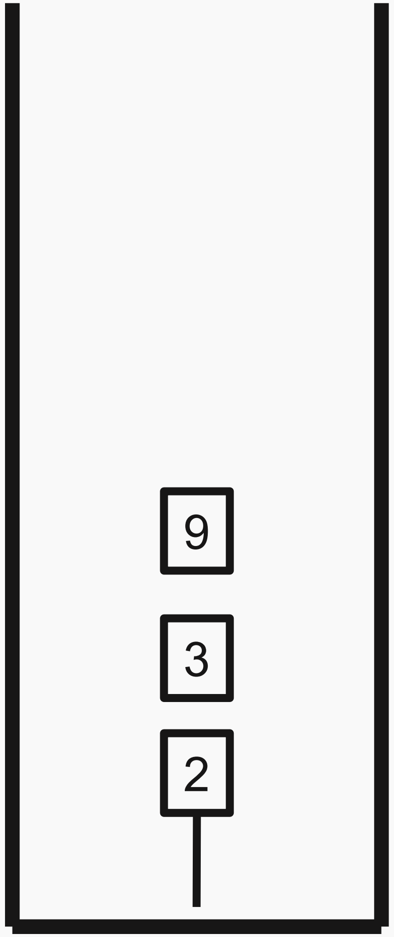




Afișare: 1 4 7

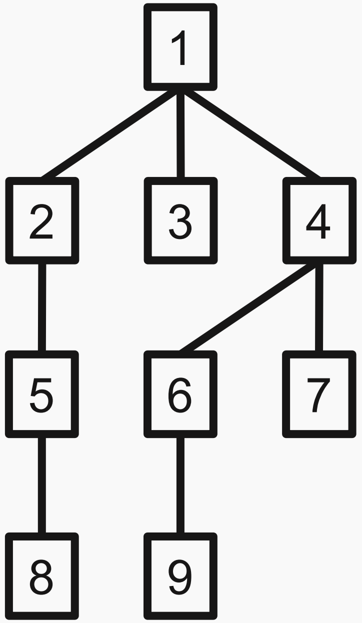
Afișarea unui arbore folosind o stivă

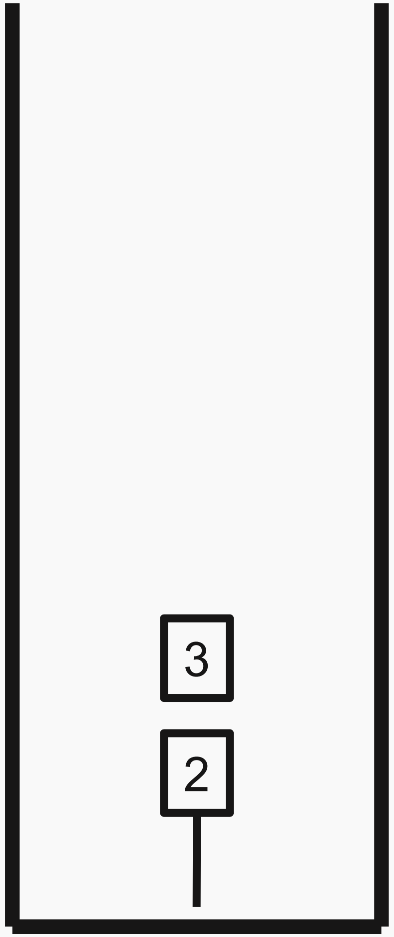




Afișare: | 1 | 4 | 7 | 6 |

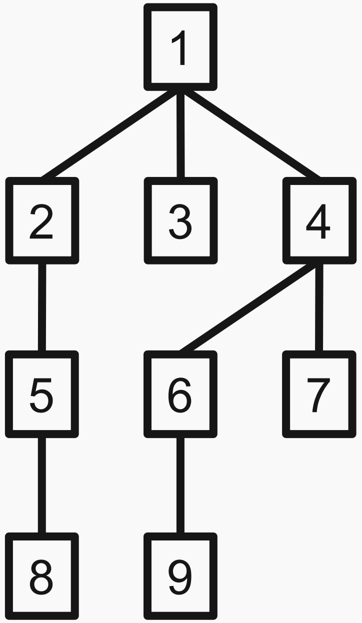
Afișarea unui arbore folosind o stivă

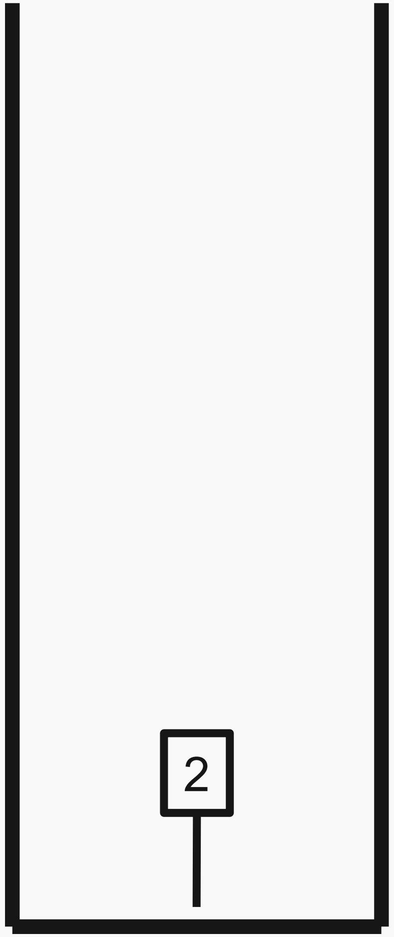




Afișare: | 1 | 4 | 7 | 6 | 9 |

Afișarea unui arbore folosind o stivă

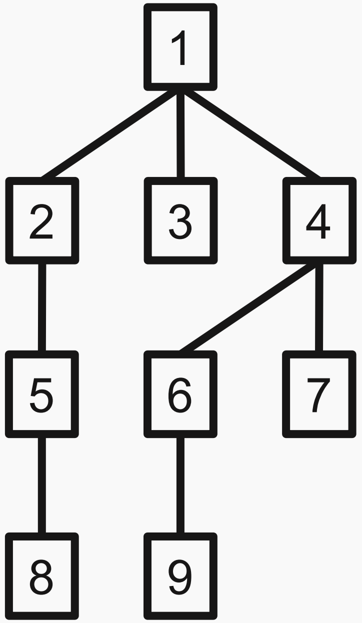


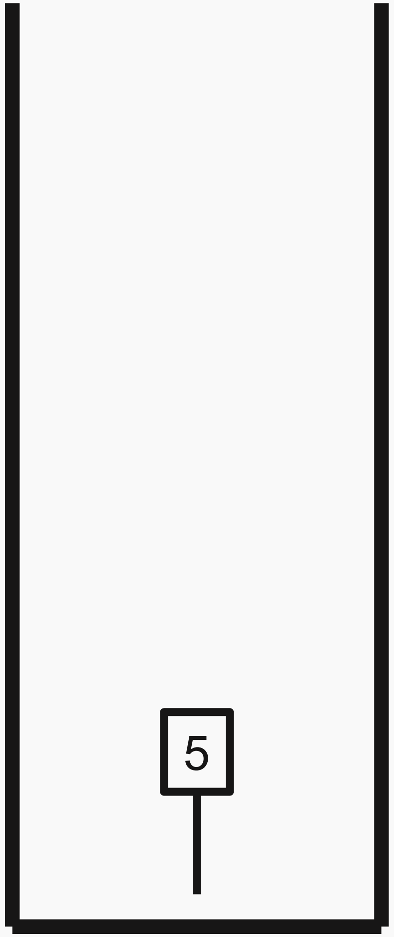


Afișare:

1	4	7	6	9	3

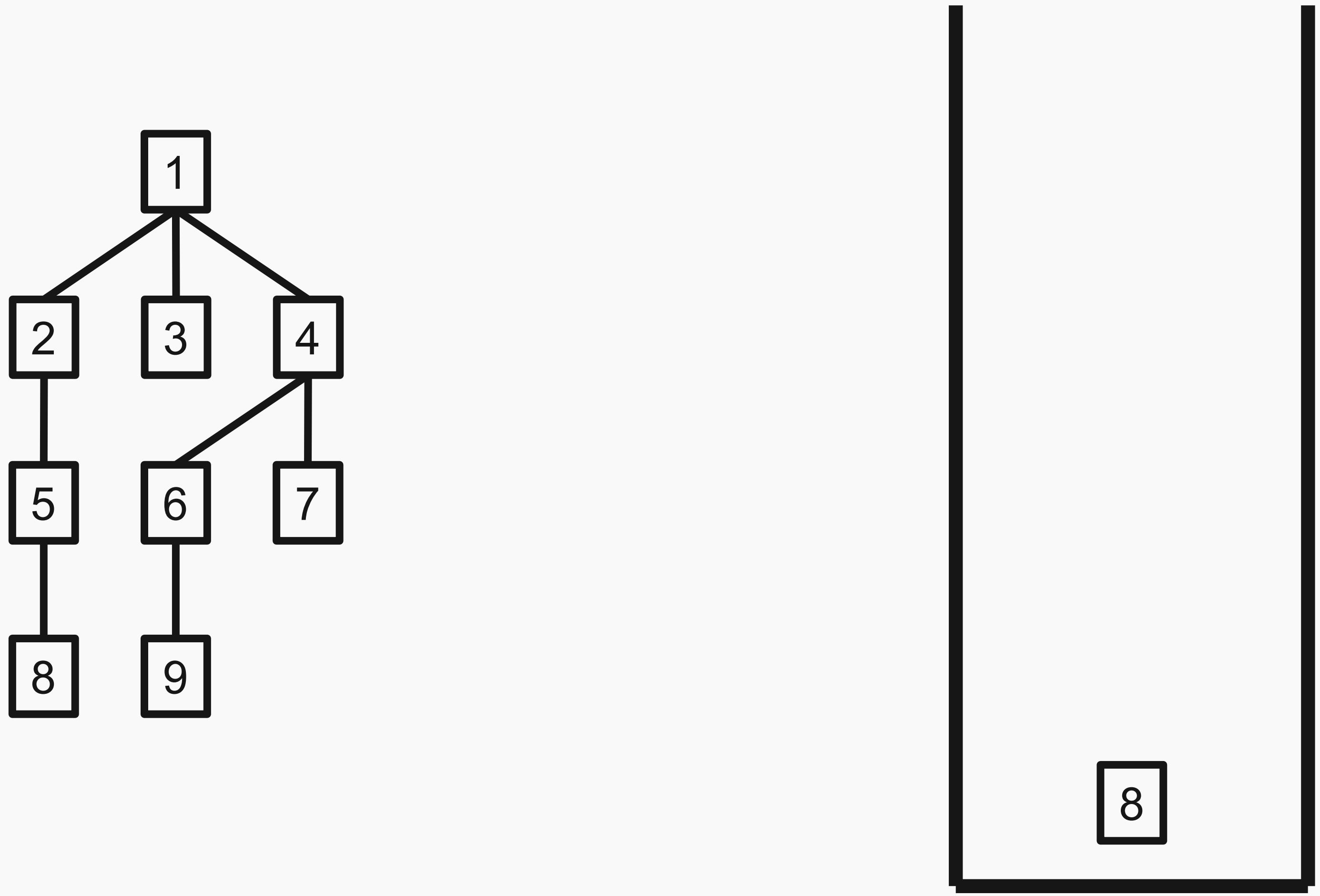
Afișarea unui arbore folosind o stivă





Afișare: | 1 | 4 | 7 | 6 | 9 | 3 | 2 |

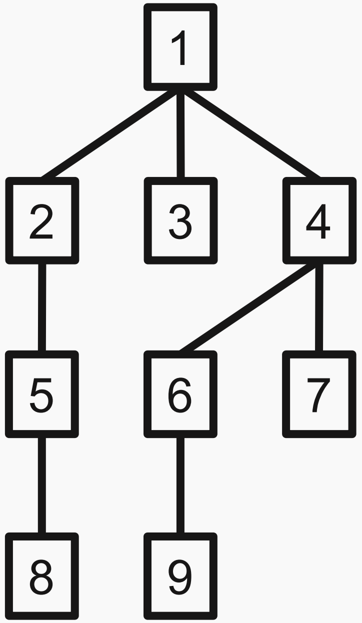
Afișarea unui arbore folosind o stivă

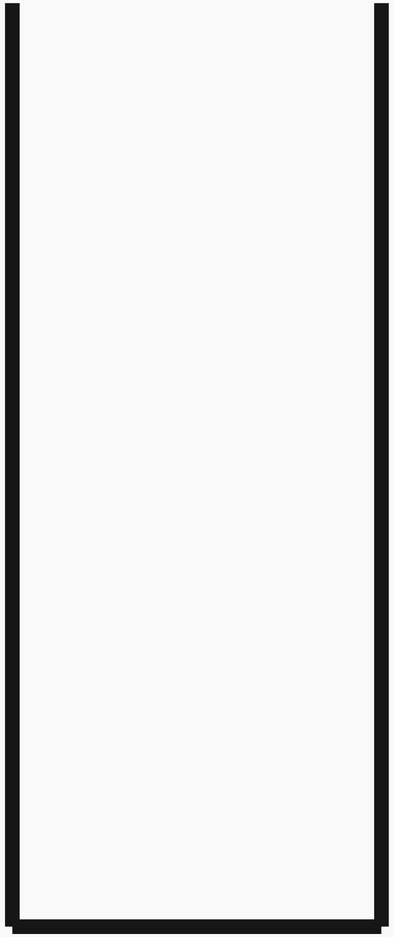


Afișare:

1	4	7	6	9	3	2	5

Afișarea unui arbore folosind o stivă





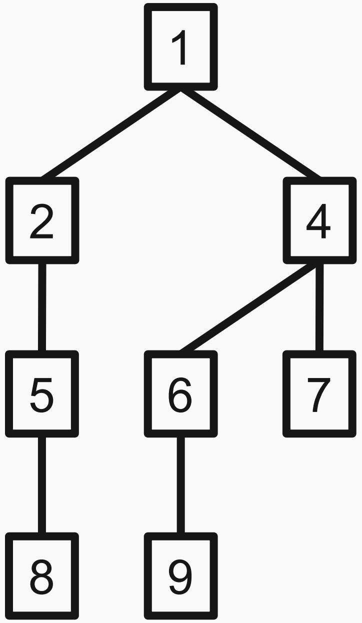
Afișare:

1	4	7	6	9	3	2	5	8

Arbori binari

- Fiecare nod are **maxim** doi copii.

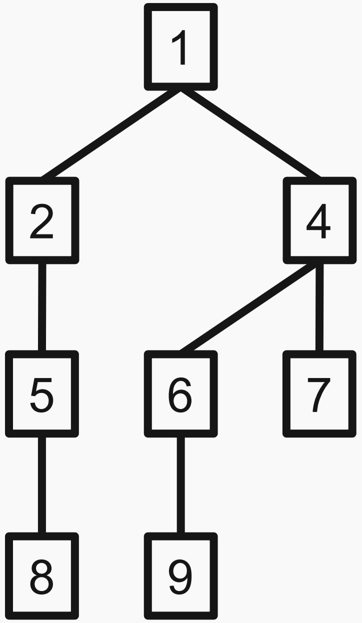
```
typedef struct binaryTreeNode
{
    void *value;
    struct binaryTreeNode *childLeft;
    struct binaryTreeNode *childRight;
    // struct binaryTreeNode *parent;
} binaryTreeNode;
```



Parcurgere arbori binari

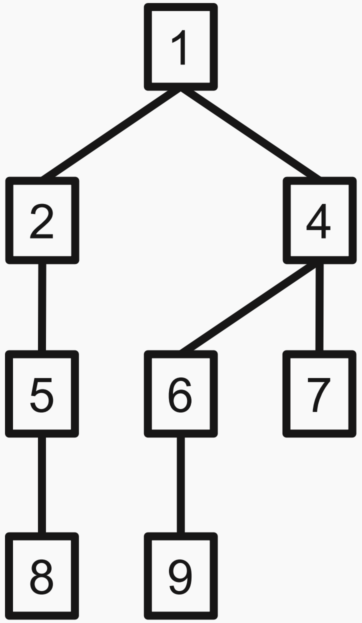
Recursiv:

- inorder (left, root, right)
- preorder (root, left, right)
- postorder (left, right, root)



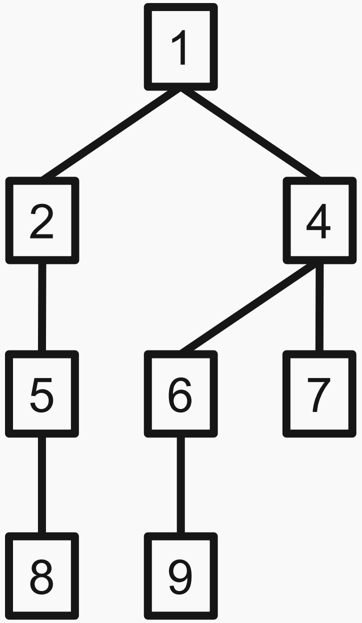
Parcurgere arbori binari - inorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childRight);
}
```



Parcurgere arbori binari - inorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childRight);
}
```

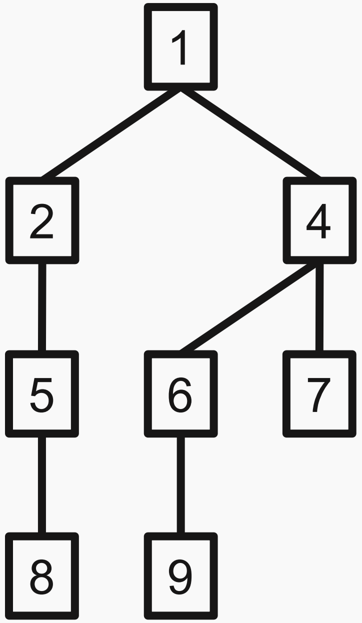


Afișare:

8	5	2	1	9	6	4	7

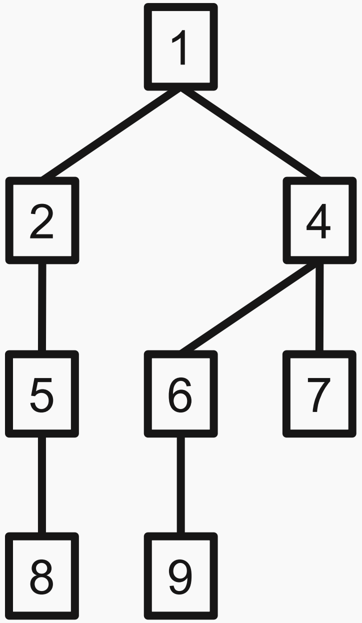
Parcurgere arbori binari - preorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childLeft);
    printBinaryTree(bTreeNode->childRight);
}
```



Parcurgere arbori binari - preorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childLeft);
    printBinaryTree(bTreeNode->childRight);
}
```

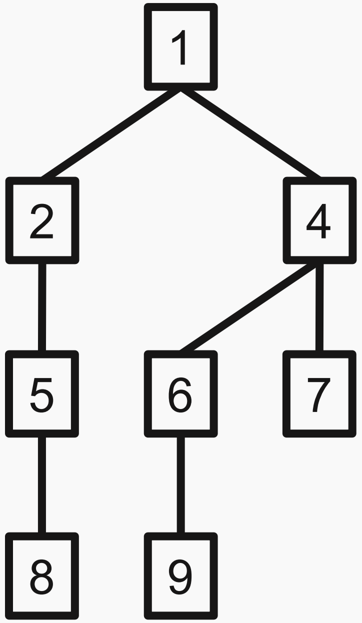


Afișare:

1	2	5	8	4	6	9	7

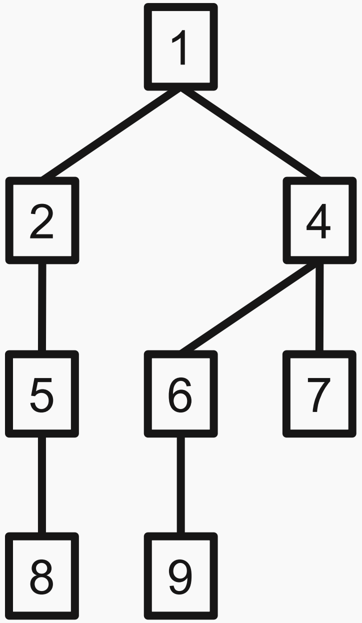
Parcurgere arbori binari - postorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBinaryTree(bTreeNode->childRight);
    printBTData(bTreeNode->value);
}
```



Parcurgere arbori binari - postorder

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBinaryTree(bTreeNode->childRight);
    printBTData(bTreeNode->value);
}
```



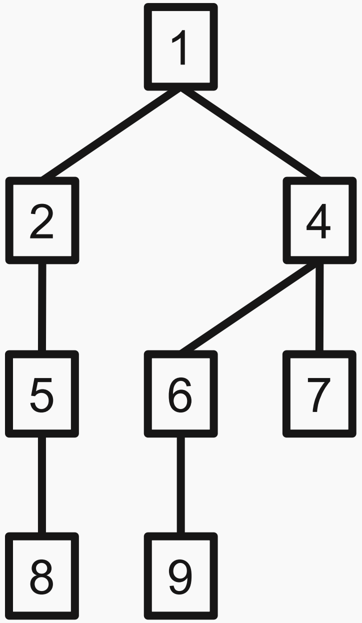
Afișare:

8	5	2	9	6	7	4	1

Înălțime arbori binari - Recursiv

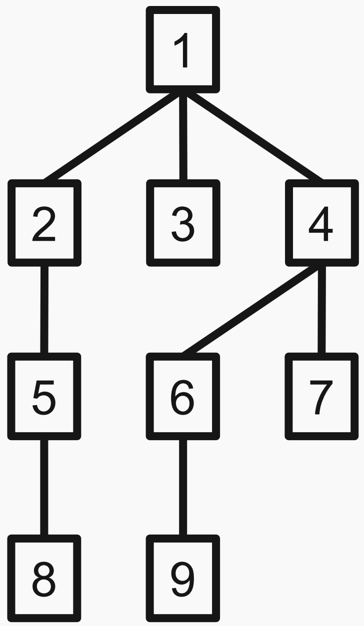
```
int getBinaryTreeHeight(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return 0;

    int hL = getBinaryTreeHeight(bTreeNode->childLeft);
    int hR = getBinaryTreeHeight(bTreeNode->childRight);
    return fmax(hL, hR) + 1;
}
```



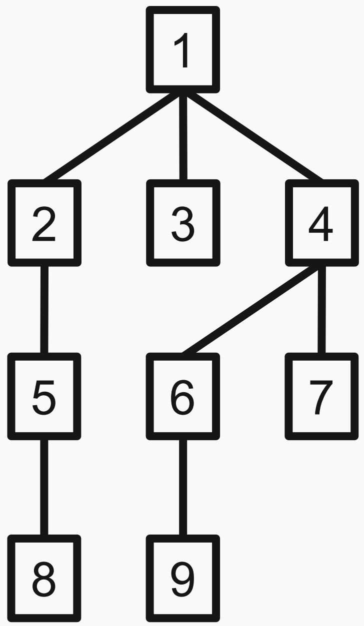
Parcurgere Arbore - Recursiv

```
void printTree(treeNode *treeNode)
{
    if (treeNode == NULL)
        return;
    printTreeData(treeNode->value);
    for (int i = 0; i < treeNode->numChildren; i++)
        printTree(treeNode->children[i]);
}
```



Parcurgere Arbore - Recursiv

```
void printTree(treeNode *treeNode)
{
    if (treeNode == NULL)
        return;
    printTreeData(treeNode->value);
    for (int i = 0; i < treeNode->numChildren; i++)
        printTree(treeNode->children[i]);
}
```



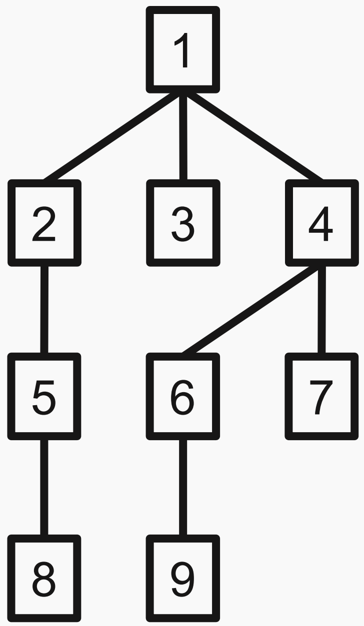
Afișare:

1	2	5	8	3	4	6	9	7

Înălțime arbori - Recursiv

```
int getTreeHeight(treeNode *treeNode)
{
    if (treeNode == NULL)
        return 0;

    int maxH = 0;
    for (int i = 0; i < treeNode->numChildren; i++)
    {
        int h = getTreeHeight(treeNode->children[i]);
        maxH = fmax(maxH, h);
    }
    return maxH + 1;
}
```



Căutare binară

- Avem un vector sortat.
- Se verifică dacă un element este în vector și pe ce poziție se află.
- Dorim o variantă mai rapidă decât liniar (să nu verificăm toate elementele din vector).

Căutare binară

- Comparăm elementul căutat cu cel din mijloc.
  - Dacă este mai mic, căutăm recursiv în partea stângă.
  - Dacă este mai mare căutăm recursiv în partea dreaptă.

Căutare binară

Căutăm $\boxed{3}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Căutare binară

Căutăm | 3 |

Între pozițiile | 0 | 15 |

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

$3 < 7$

Căutare binară

Căutăm $\boxed{3}$

Între pozițiile $\boxed{0}\boxed{7}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 5

Căutare binară

Căutăm $\boxed{3}$

Între pozițiile $\boxed{0}\boxed{3}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 > 2

Căutare binară

Căutăm $\boxed{3}$

Între pozițiile | 3 | 3 |

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 = 3

end

Complexitate ?

Căutare binară

Căutăm 3

Între pozițiile 3 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

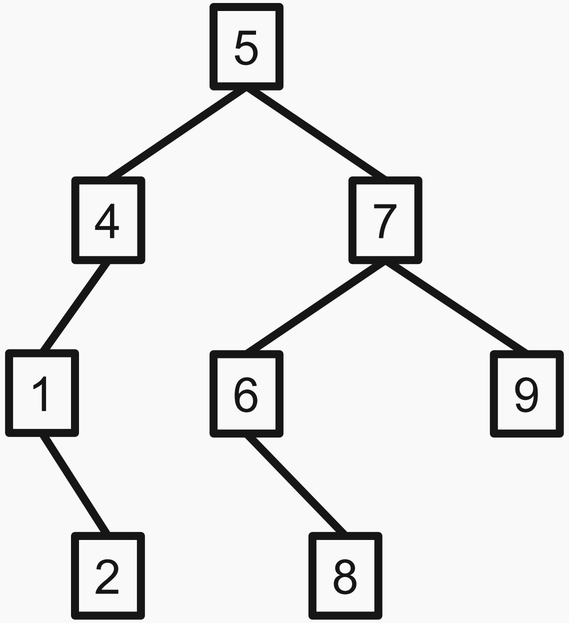
3 = 3

end

Complexitate $O(log_2(N))$

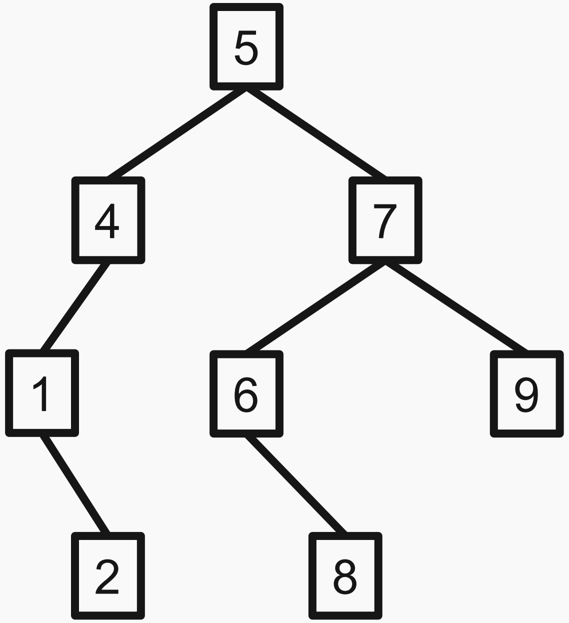
Arbori binari de căutare

- Fiecare nod reprezintă un număr.
- Pentru oricare nod:
  - Toate elementele de pe subarborele **stâng** sunt mai **mici** decât acesta.
  - Toate elementele de pe subarborele **drept** sunt mai **mari** decât acesta.



Arbori binari de căutare

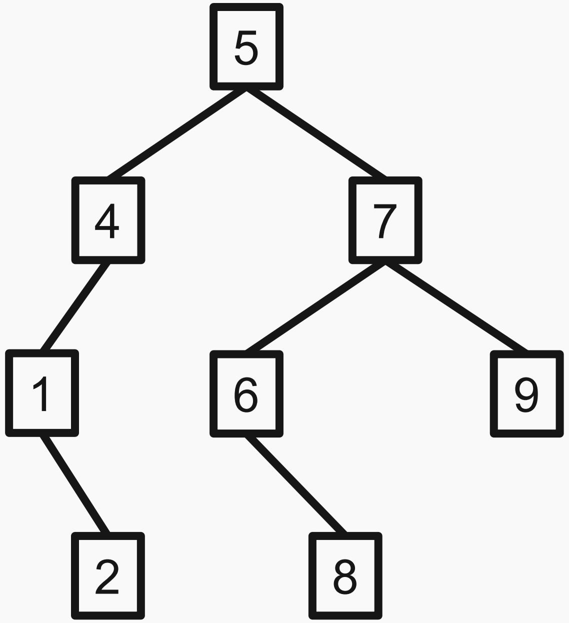
```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childRight);
}
```



Afișare inorder:

Arbori binari de căutare

```
void printBinaryTree(binaryTreeNode *bTreeNode)
{
    if (bTreeNode == NULL)
        return;
    printBinaryTree(bTreeNode->childLeft);
    printBTData(bTreeNode->value);
    printBinaryTree(bTreeNode->childRight);
}
```

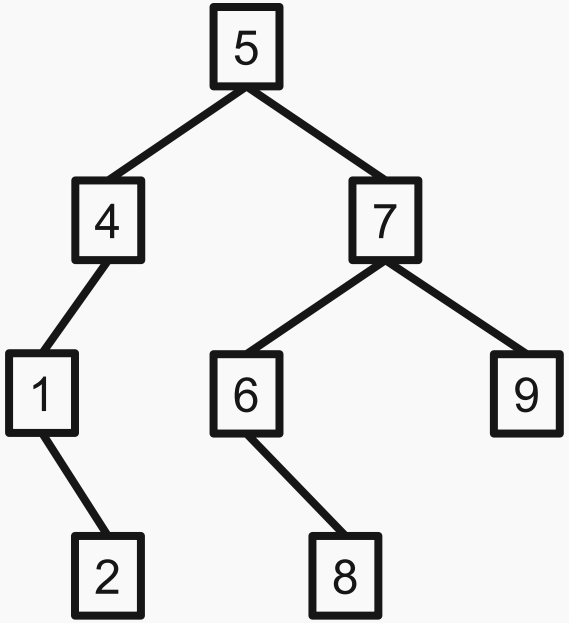


Afișare inorder:

1	2	4	5	6	7	8	9

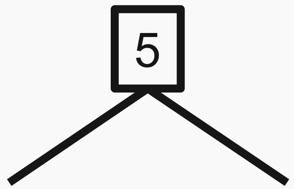
Arbori binari de căutare – Căutarea

```c
int bstSearch(binaryTreeNode *bstNode, int needle) {
    if (bstNode == NULL)
        return 0;
    if (bstNode->value == needle)
        return 1;
    if (bstNode->value < needle)
        return bstSearch(bstNode->childRight, needle);
    if (bstNode->value > needle)
        return bstSearch(bstNode->childLeft, needle);
}
```



Arbori binari de căutare – Creare nod nou

```
binaryTreeNode *createBSTNode(int value)
{
    binaryTreeNode *newBSTNode = (binaryTreeNode *)malloc(sizeof(binaryTreeNode));
    if (newBSTNode == NULL)
        return NULL;
    newBSTNode->value = value;
    newBSTNode->childLeft = NULL;
    newBSTNode->childRight = NULL;
    return newBSTNode;
}
```

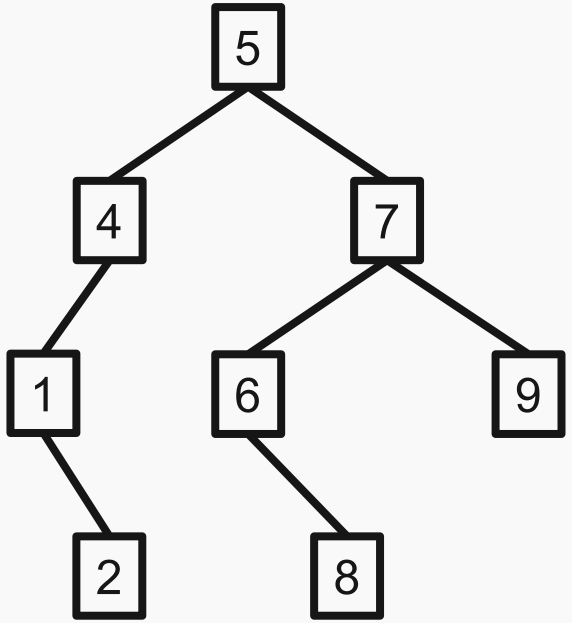


Arbori binari de căutare – Inserare nod

```
binaryTreeNode *bstInsert(binaryTreeNode *bstNode, binaryTreeNode *toInsert)
{
    if (bstNode == NULL)
        return toInsert;

    if (bstNode->value < toInsert->value)
        bstNode->childRight = bstInsert(bstNode->childRight, toInsert);
    else if (bstNode->value >= toInsert->value)
        bstNode->childLeft = bstInsert(bstNode->childLeft, toInsert);

    return bstNode;
}
```




```
rootBST = bstInsert(rootBST, createBSTNode(3));
```

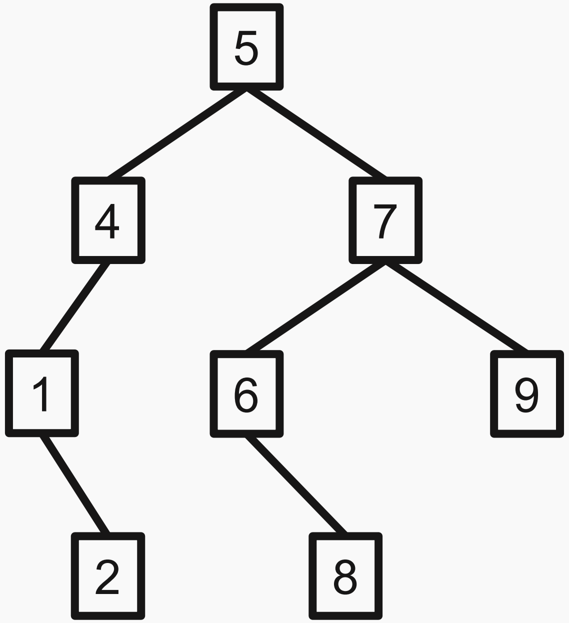
Arbori binary de căutare - Ștergere nod

```
binaryTreeNode *bstRemove(binaryTreeNode *bstNode, int value)
{
    if (bstNode == NULL)
        return NULL;

    if (bstNode->value == value) {
        binaryTreeNode *aux = bstInsert(bstNode->childLeft, bstNode->childRight);
        free(bstNode);
        return aux;
    }

    if (bstNode->value < value)
        bstNode->childRight = bstRemove(bstNode->childRight, value);
    else if (bstNode->value > value)
        bstNode->childLeft = bstRemove(bstNode->childLeft, value);

    return bstNode;
}
```

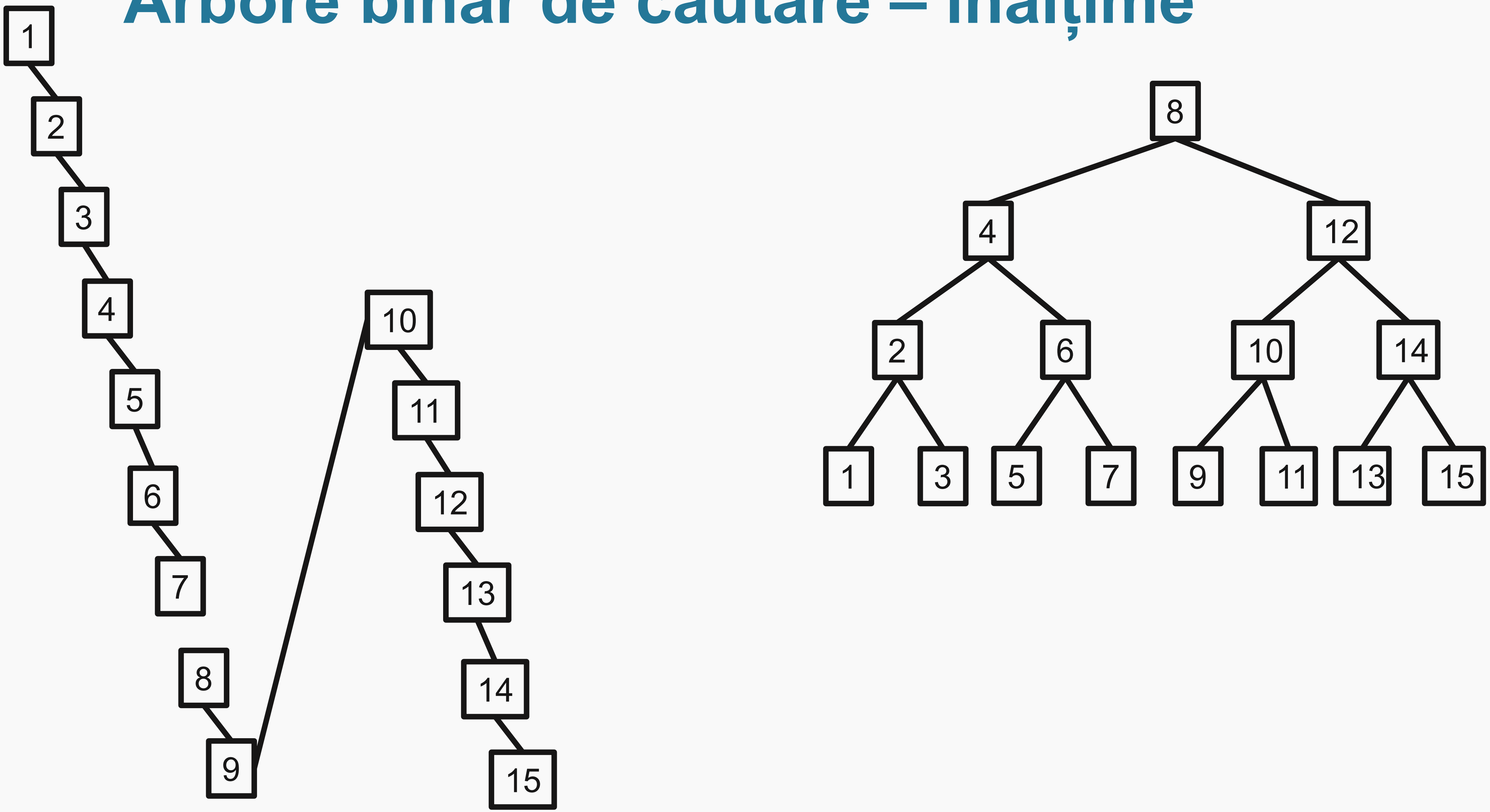


Complexitate

$$\Omega(1) \leq insert|search|delete \leq O(h)$$

h este înălțimea arborelui

Arbore binar de căutare – înălțime



Arbore binar de căutare – înălțime minimă

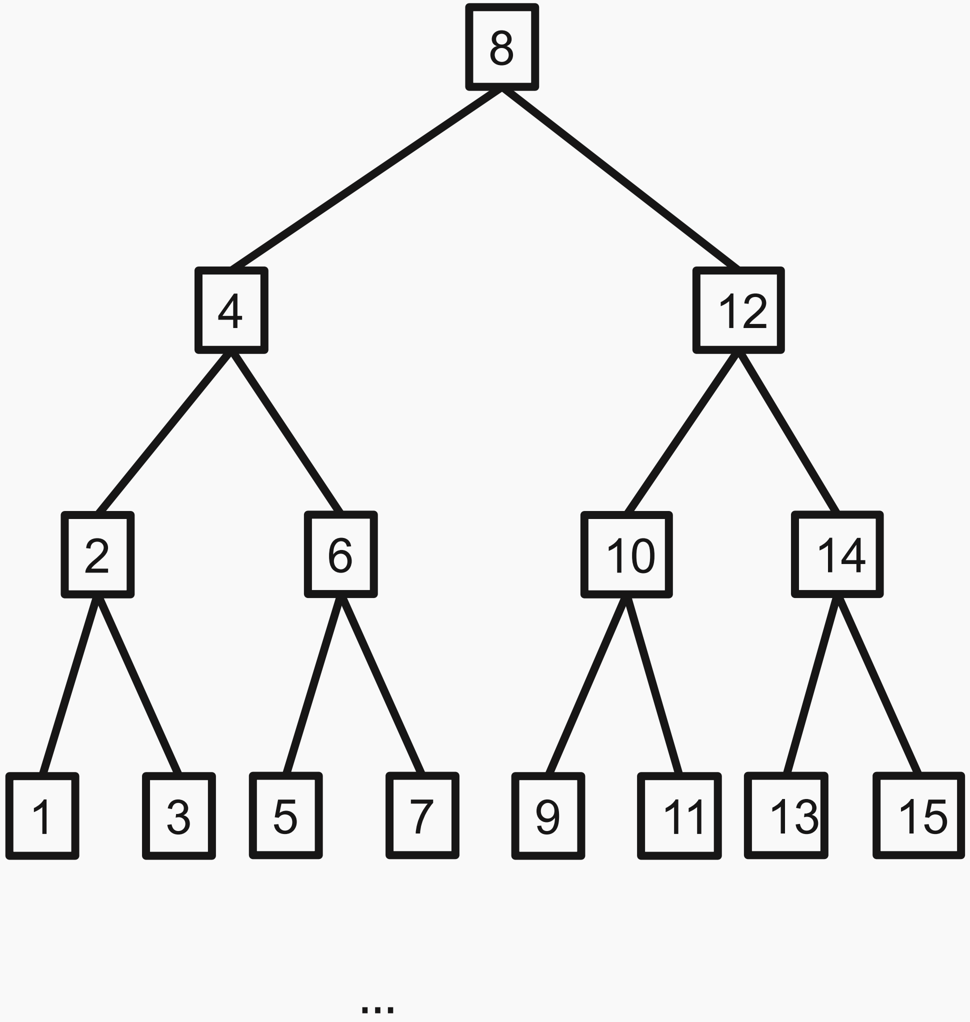
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



Arbore binar de căutare – înălțime minimă

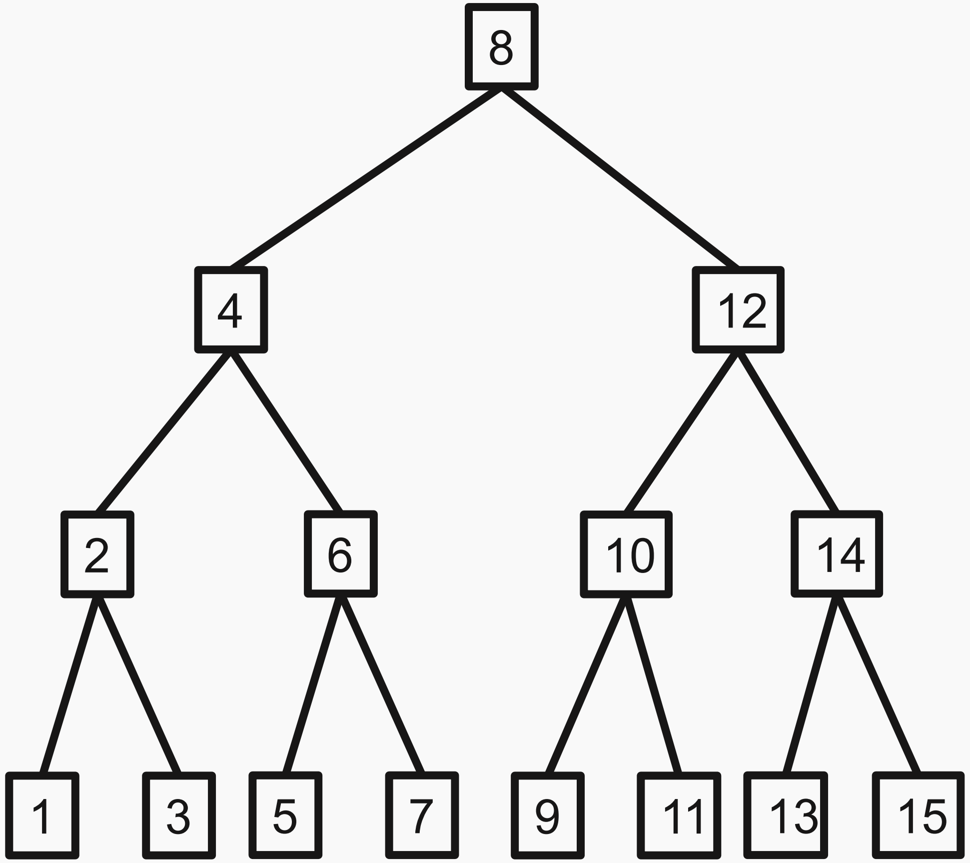
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



...

$$N = 1 + 2 + 4 + 8 + \cdots + ?$$
$$= 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}$$

Arbore binar de căutare – înălțime minimă

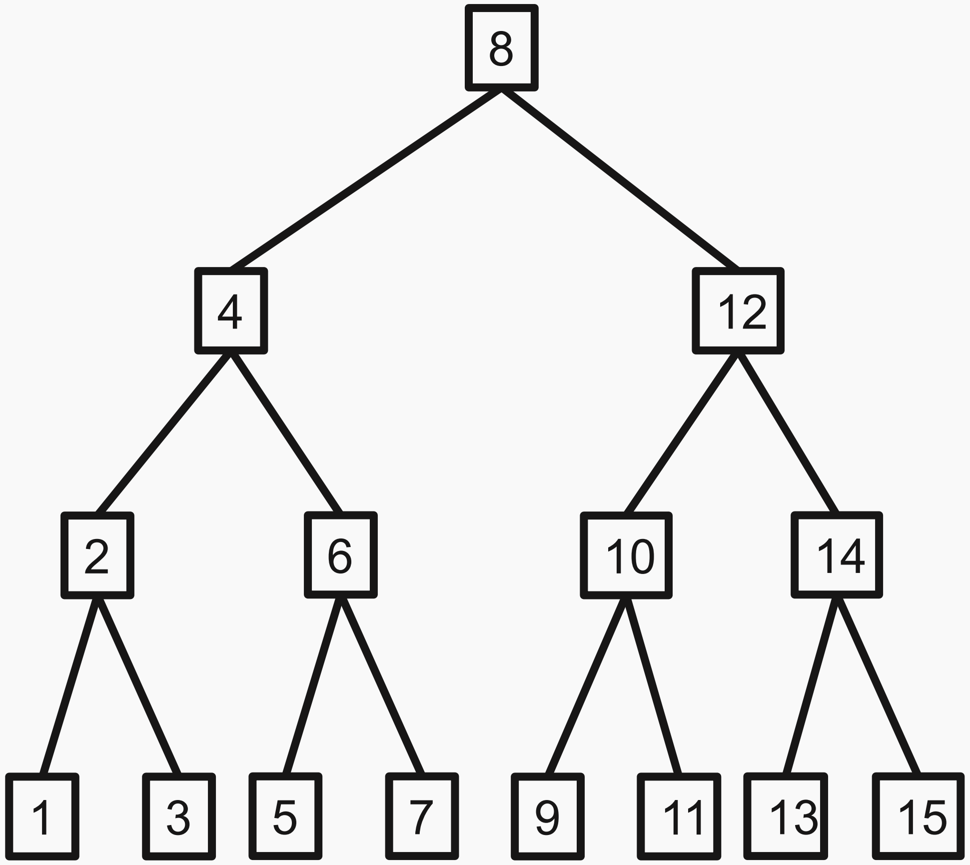
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



...

$$N = 1 + 2 + 4 + 8 + \cdots + ?$$
$$= 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}$$
$$= 0000\ 0000\ 0000\ 0001 +$$
$$0000\ 0000\ 0000\ 0010 +$$
$$0000\ 0000\ 0000\ 0100 + \cdots$$

...

Arbore binar de căutare – înălțime minimă

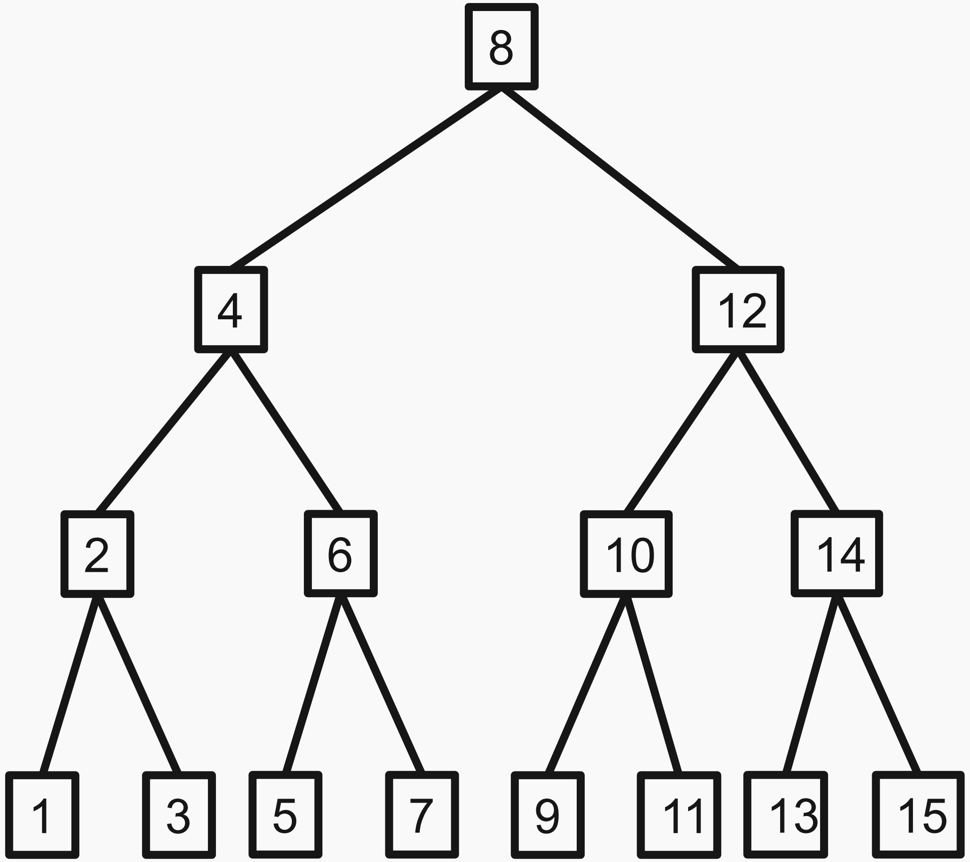
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



...

$$N = 1 + 2 + 4 + 8 + \cdots + ?$$
$$= 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1}$$
$$= 0000\ 0000\ 0000\ 0001 +$$
$$0000\ 0000\ 0000\ 0010 +$$
$$0000\ 0000\ 0000\ 0100 + \cdots$$
$$\ldots$$
$$= 1111\ 1111\ 1111\ 1111\ (h \text{ biți})$$

Arbore binar de căutare – înălțime minimă

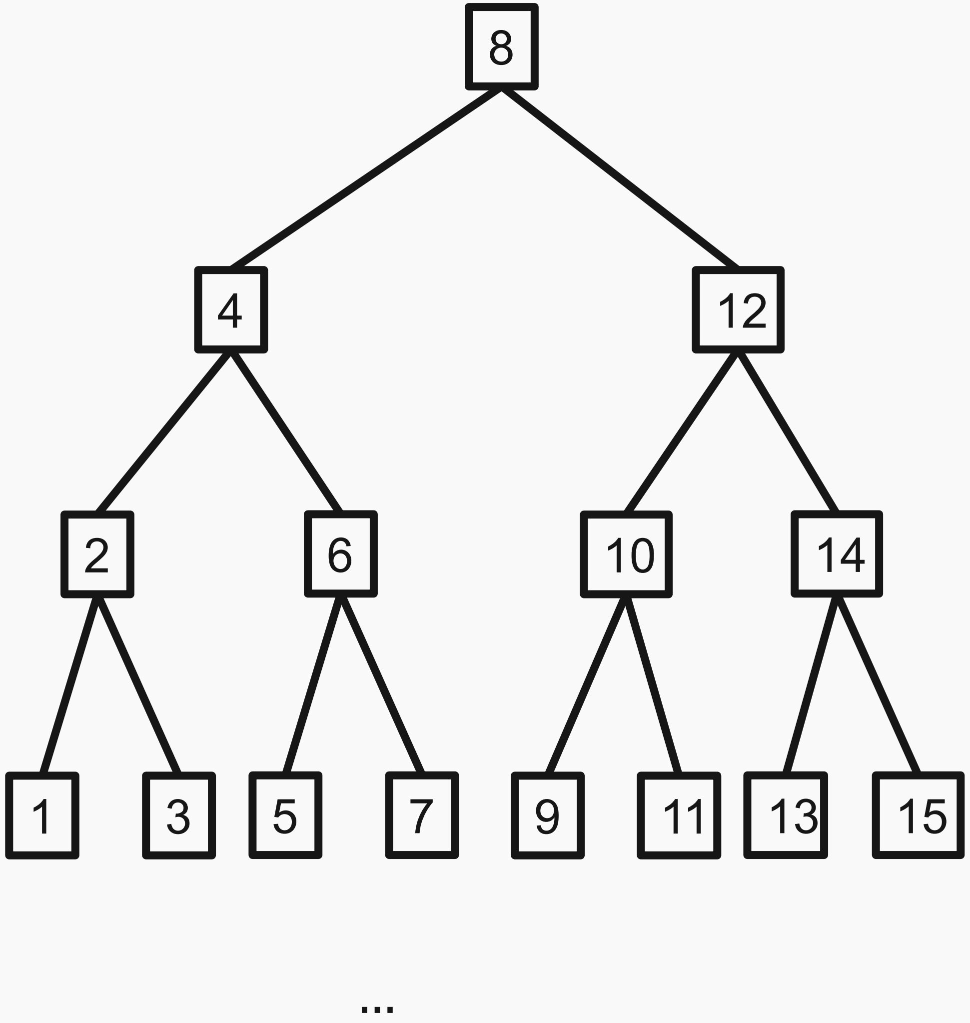
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



$$
\begin{aligned}
N &= 1 + 2 + 4 + 8 + \cdots + ? \\
&= 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} \\
&= 0000\ 0000\ 0000\ 0001 + \\
&\quad 0000\ 0000\ 0000\ 0010 + \\
&\quad 0000\ 0000\ 0000\ 0100 + \cdots \\
&\ \ldots \\
&= 1111\ 1111\ 1111\ 1111\ (h \text{ biți}) \\
&= 2^h - 1
\end{aligned}
$$

Arbore binar de căutare – înălțime minimă

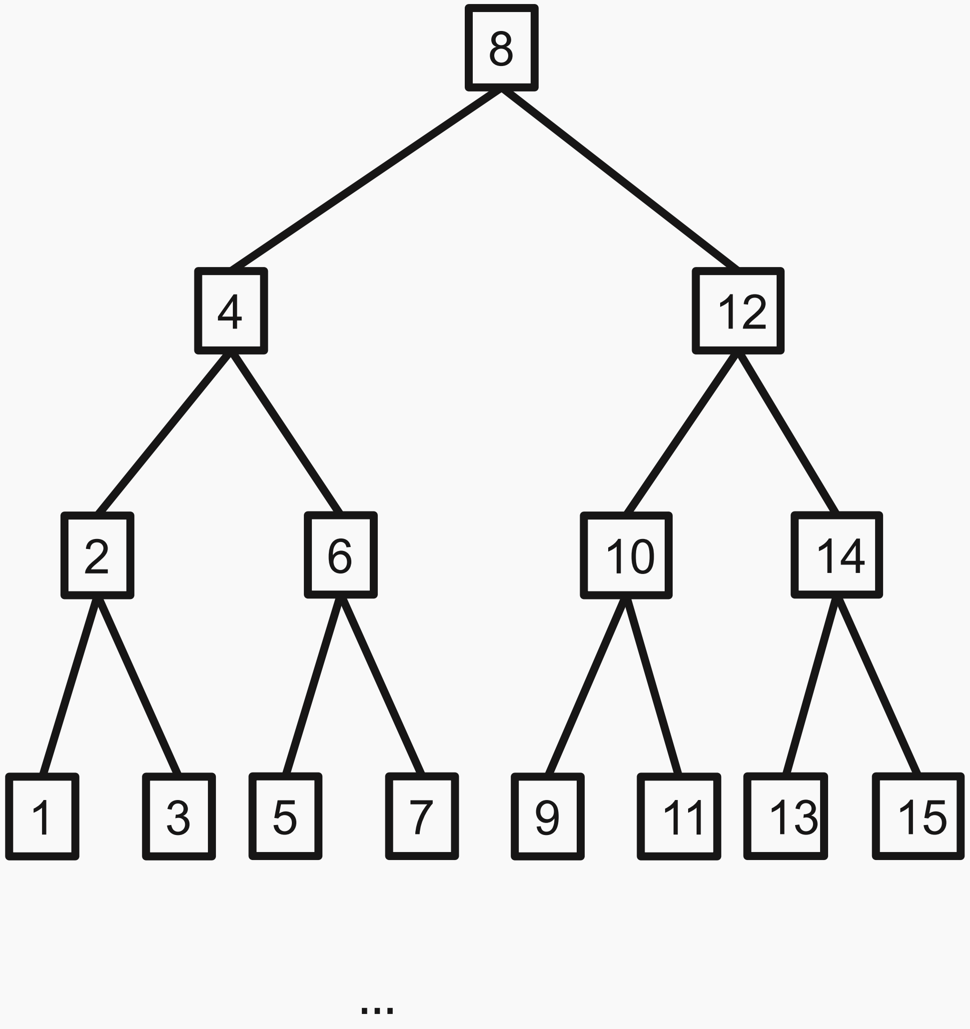
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



$$N = 2^h - 1$$
$$N + 1 = 2^h$$

$$h = \log_2(N + 1)$$

Dacă arborele este echilibrat

Arbore binar de căutare – înălțime minimă

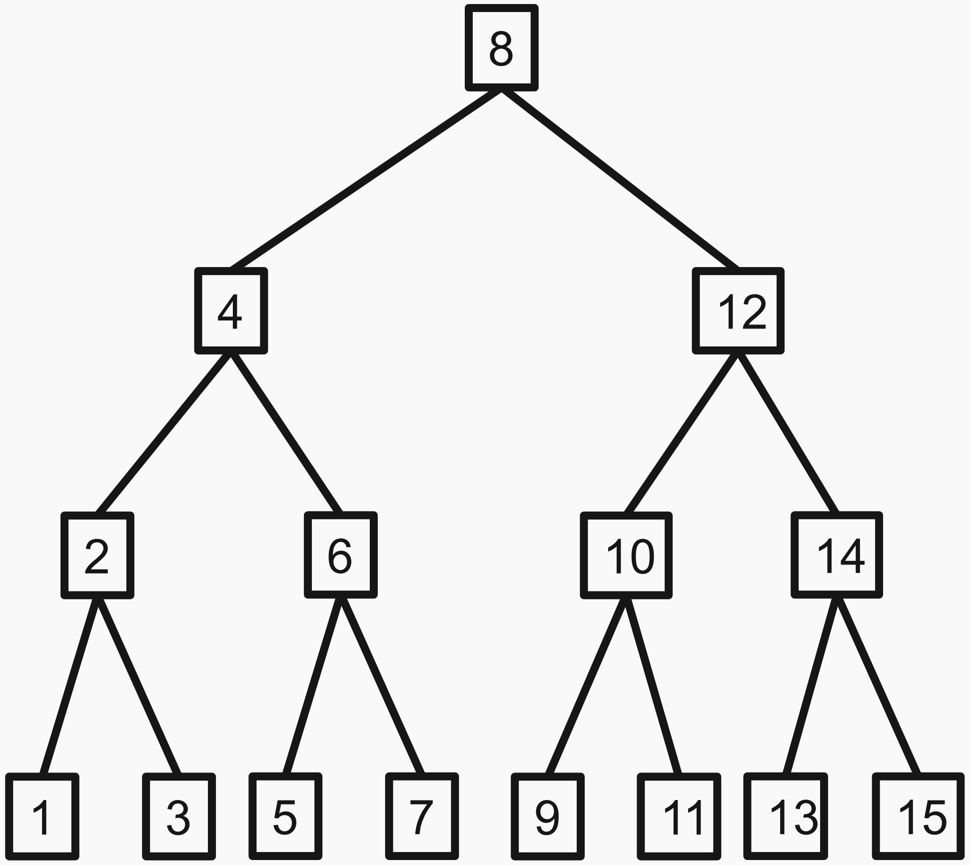
$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^{h-1} = ?$



...

$$N = 2^h - 1$$
$$N + 1 = 2^h$$

$$\boxed{h = \log_2(N)}$$

Complexitate

	Vector	Listă	Arbore binar căutare echilibrat
Complexitate acces	O(1)	O(N)	O(log2(N))
Complexitate inserție	O(N)	O(N)	O(log2(N))
Complexitate inserție capete	O(N)/O(1)	O(1)	O(log2(N))
Complexitate ștergere	O(N)	O(N)	O(log2(N))
Complexitate ștergere capete	O(N)/O(1)	O(1)	O(log2(N))