



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY  
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

# DATA STRUCTURES AND ALGORITHMS

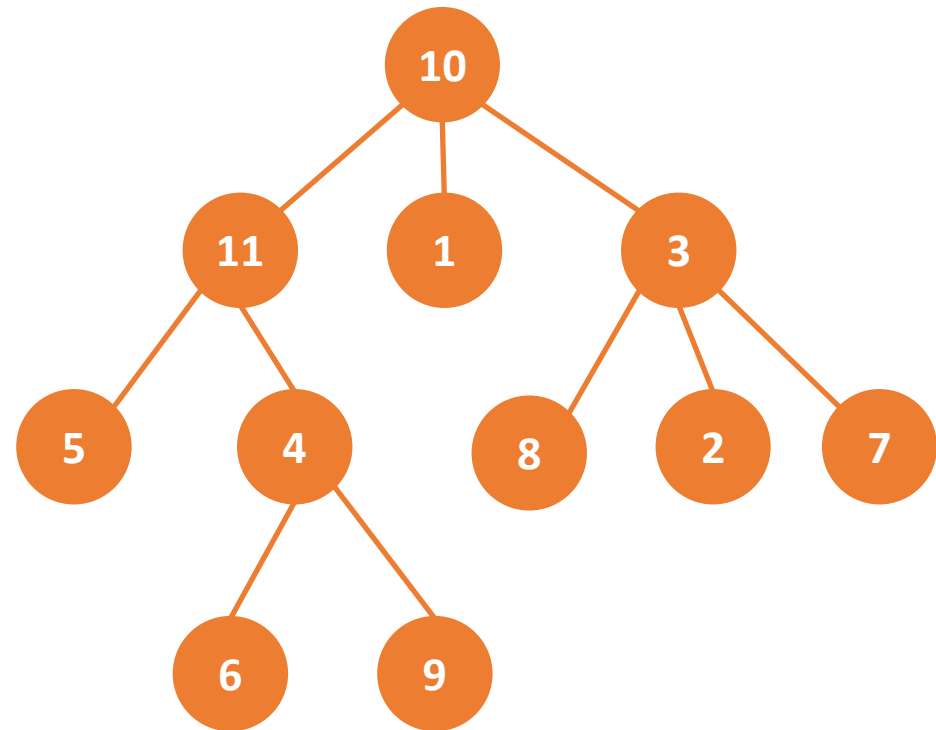
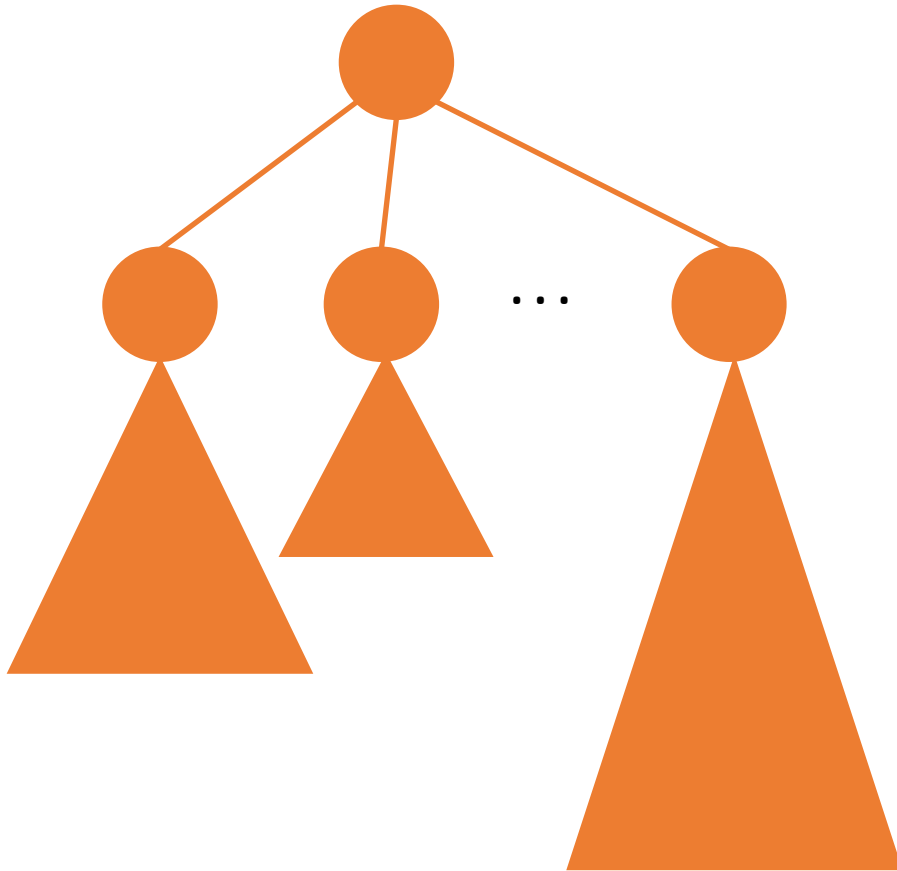
## Trees

# Content

---

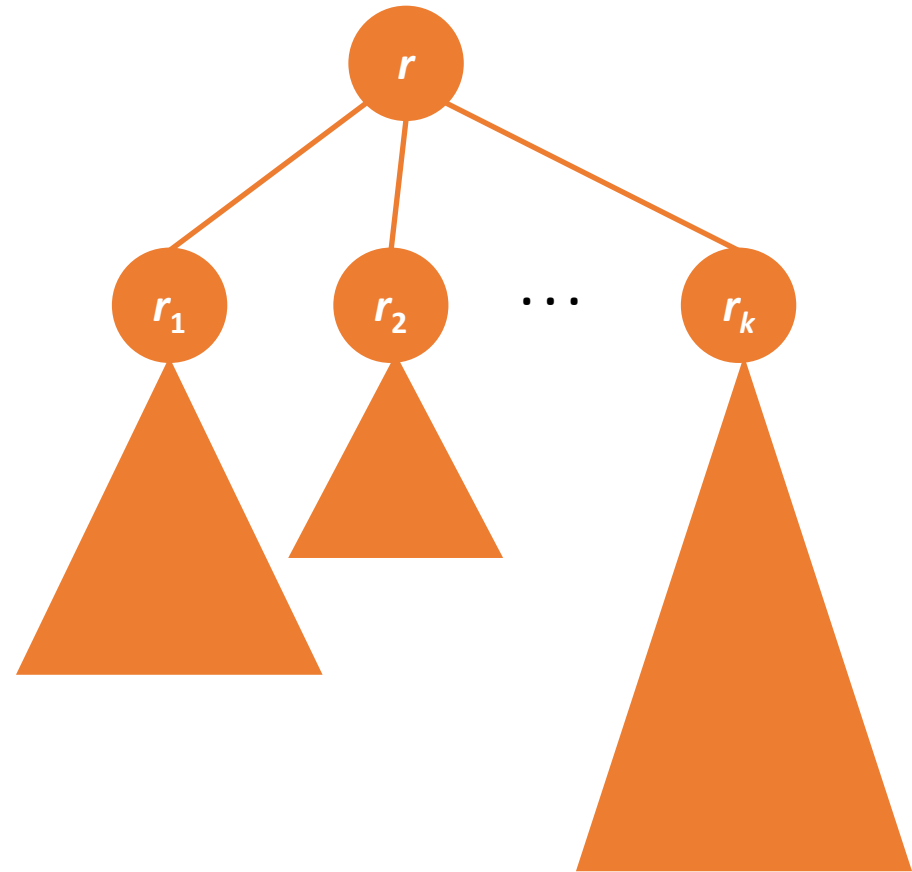
- Definition
- Terminology
- Traversal
- Data structures
- Operations

# Tree: definition



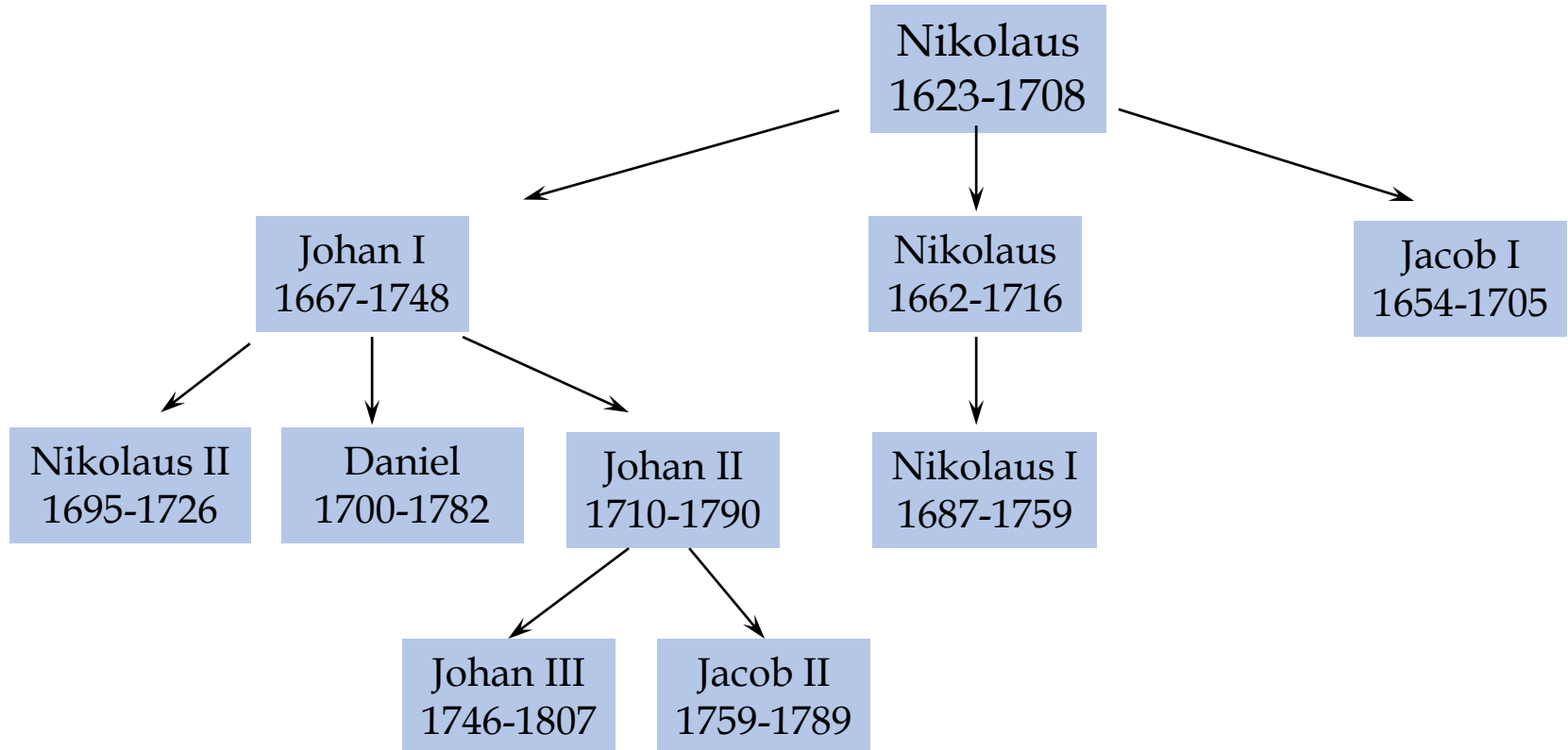
# Tree: definition

- Store objects in a hierarchical structure
- Recursive definition
  - Base case:  $r$  is a node,  $T$  is a tree containing only one node  $r$  which is also the root of  $T$
  - Recursion:
    - Suppose  $T_1, T_2, \dots, T_k$  are trees rooted at  $r_1, r_2, \dots, r_k$
    - Given a node  $r$
    - Make  $r_1, r_2, \dots, r_k$  children of  $r$  creating a new tree  $T$



# Applications

- Family tree of mathematicians of the Bernoulli family



# Applications

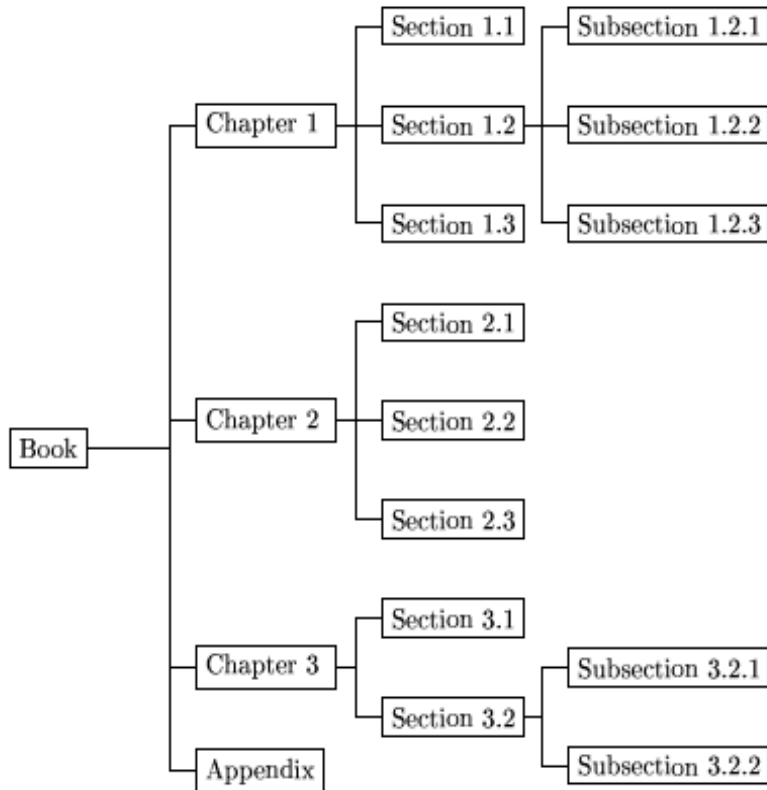
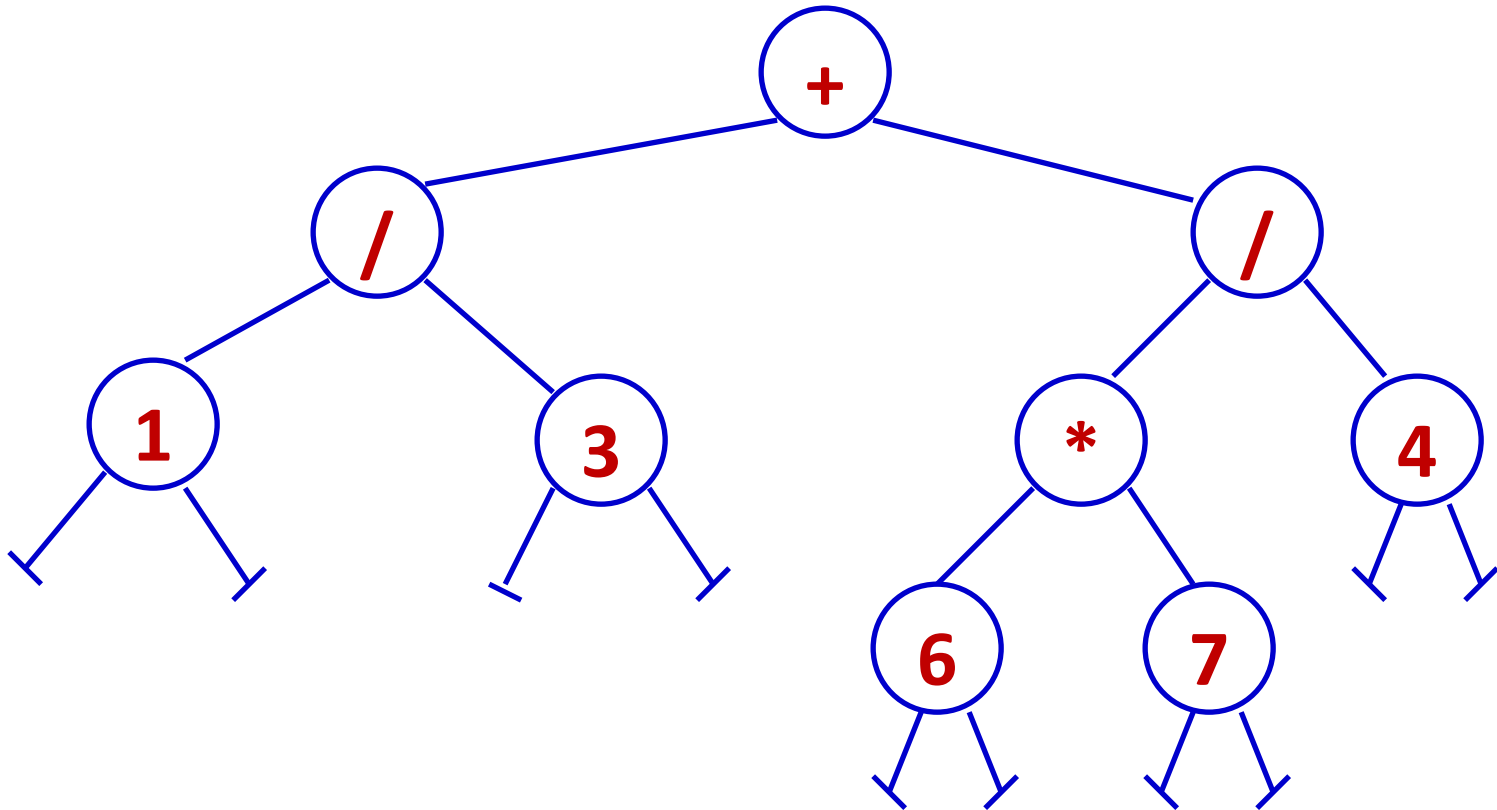


Table of contents



Folders organization

# Applications: expression tree



$$1/3 + 6*7/4$$

# Terminology

---

- Path: sequence of nodes  $x_1, x_2, \dots, x_q$  where  $x_i$  is the parent of  $x_{i+1}$ ,  $i = 1, 2, \dots, q-1$ . Length of the path is  $q-1$
- Leaves: do not have children
- Internal nodes: have children
- Sibling: 2 nodes  $u$  and  $v$  are sibling if they have the same parent
- Ancestors: node  $u$  is an ancestor of  $v$  if there is a path from  $u$  to  $v$
- Descendants: node  $u$  is a descendant of  $v$  if  $v$  is an ancestor of  $u$
- Height: the height of a node is the length of the longest path from that node to some leaf plus 1
- Depth: the depth of a node  $v$  is the length of the unique path from that node to the root plus 1



# Terminology

- **Root:** do not have parent (example: node A)
- B, C, D are siblings;
- **Internal nodes:** A, B, C, F
- **Leaves:** E, I, J, K, G, H, D

Con của B:

- E, F

Cha của E:

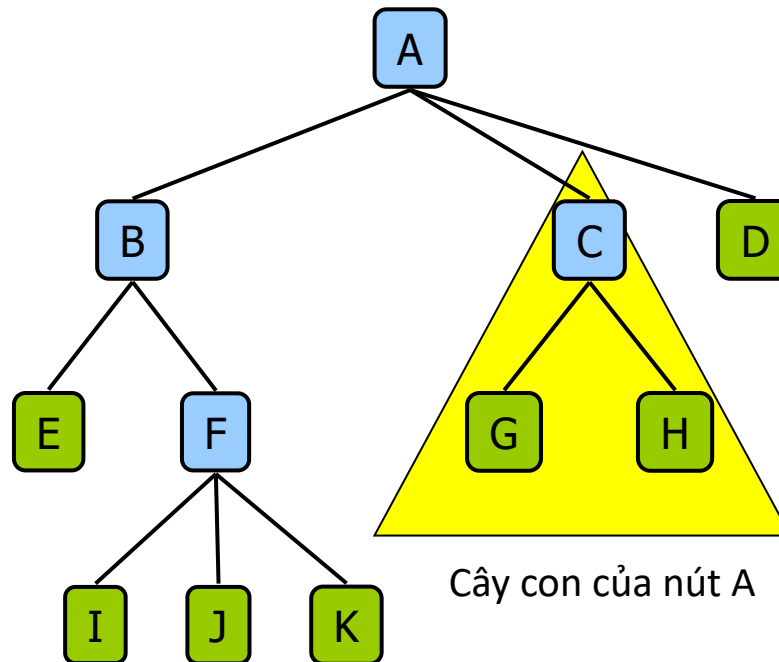
- B

Tổ tiên của F:

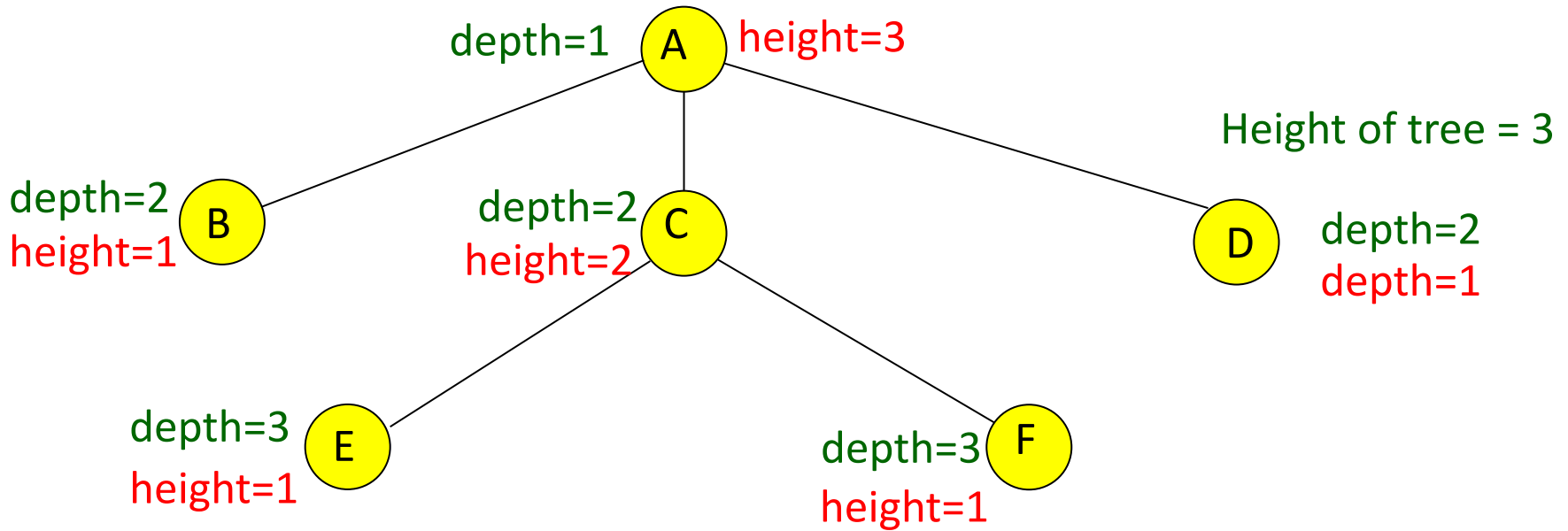
- B, A

Hậu duệ của B:

- E, F, I, J, K



# Terminology



# Traversal

---

- Visit nodes of a tree in some order
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal
- Consider a tree  $T$ 
  - root  $r$
  - subtrees  $T_1$  (root  $r_1$ ),  $T_2$  (root  $r_2$ ), ...,  $T_k$  (root  $r_k$ ) from left to right

# Traversal

- Pre-order traversal
  - Visit the root
  - Traverse  $T_1$  in the pre-order
  - Traverse  $T_2$  in the pre-order
  - ...
  - Traverse  $T_k$  in the pre-order

```
preOrder(r){  
    if(r = NULL) return;  
    visit(r);  
    for each p =  $r_1, r_2, \dots, r_k$  {  
        preOrder(p);  
    }  
}
```

# Traversal

- In-order traversal
  - Traverse  $T_1$  in the in-order
  - Visit the root  $r$
  - Traverse  $T_2$  in the in-order
  - ...
  - Traverse  $T_k$  in the in-order

```
inOrder(r){  
    if(r = NULL) return;  
    inOrder(r1);  
    visit(r);  
    for each p = r2, ..., rk {  
        inOrder(p);  
    }  
}
```

# Traversal

- Post-order traversal
  - Traverse  $T_1$  in the post-order
  - Traverse  $T_2$  in the post-order
  - ...
  - Traverse  $T_k$  in the post-order
  - Visit the root  $r$

```
postOrder(r){  
    if(r = NULL) return;  
    for each p =  $r_1, r_2, \dots, r_k$  {  
        postOrder(p);  
    }  
    visit(r);  
}
```

# Data structures

---

- Array:
  - Suppose nodes are numbered  $1, 2, \dots, n$
  - $a[1..n]$  in which  $a[i]$  is the parent of  $i$
  - Implementation of many operations on the tree might be too complicated
- Pointer: Each node has two pointers
  - leftMostChild: a pointer to the left-most child
  - rightSibling: a pointer to the right-sibling node

# Data structures

---

```
struct Node{  
    int id; // identifier of the node  
    Node* leftMostChild;// pointer to the left-most child  
    Node* rightSibling;// pointer to the right-sibling  
};  
Node* root;// pointer to the root of the tree
```



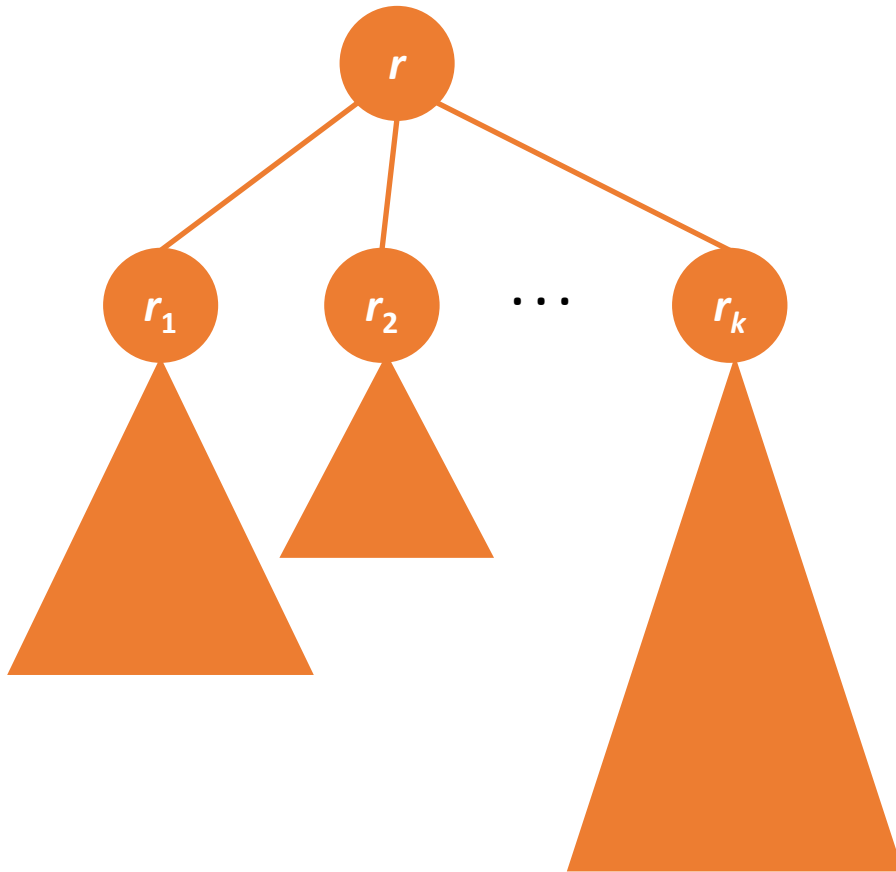
# Operations

---

- $\text{find}(r, \text{id})$ : return the node having identifier  $\text{id}$  on the tree rooted at  $r$
- $\text{insert}(r, p, \text{id})$ : create a node having identifier  $\text{id}$ , insert that node to the end of the children list of  $p$  on the tree rooted at  $r$
- $\text{height}(r, p)$ : return the height of node  $p$  on the tree rooted at  $r$
- $\text{depth}(r, p)$ : return the depth of the node  $p$  on the tree rooted at  $r$
- $\text{parent}(r, p)$ : return the parent of  $p$  on the tree rooted at  $r$
- $\text{count}(r)$ : return the number of nodes of the tree rooted at  $r$
- $\text{countLeaves}(r)$ : return the number of leaves of the tree rooted at  $r$

# Operations

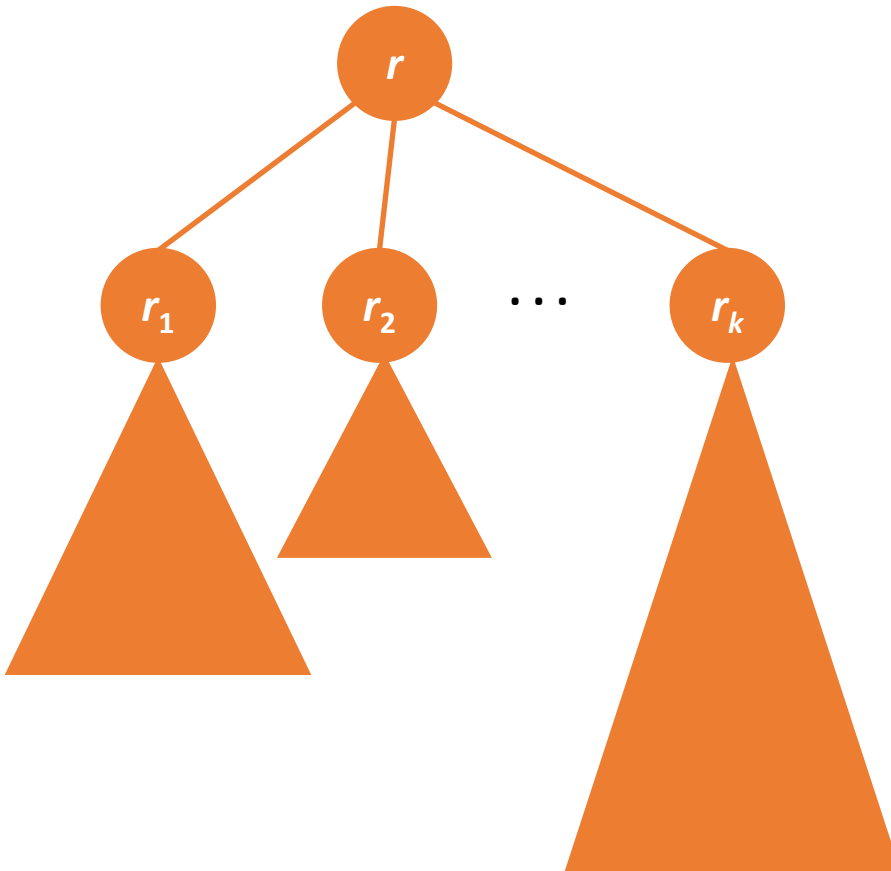
- Find a node given the identifier



```
Node* find(Node* r, int v){
    if(r == NULL) return NULL;
    if(r->id == v) return r;
    Node* p = r->leftMostChild;
    while(p != NULL){
        Node* h = find(p,v);
        if(h != NULL) return h;
        p = p->rightSibling;
    }
    return NULL;
}
```

# Operations

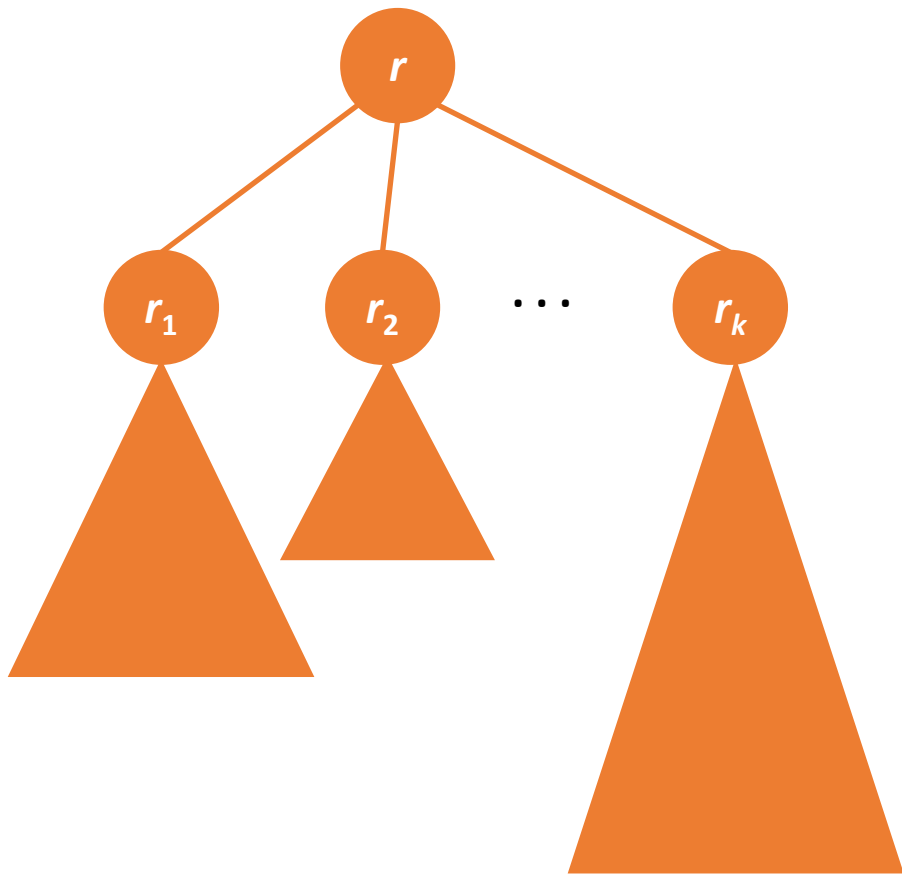
- Pre-order traversal



```
void preOrder(Node* r){  
    if(r == NULL) return;  
    printf("%d ",r->id);  
    Node* p = r->leftMostChild;  
    while(p != NULL){  
        preOrder(p);  
        p = p->rightSibling;  
    }  
}
```

# Operations

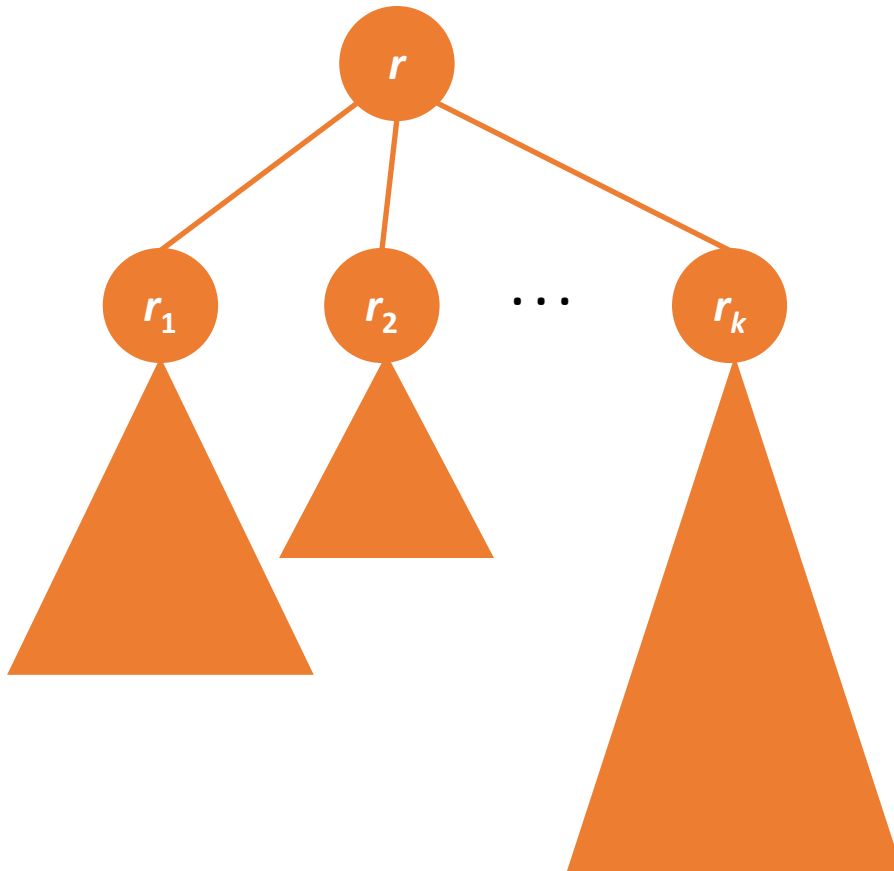
- In-order traversal



```
void inOrder(Node* r){  
    if(r == NULL) return;  
    Node* p = r->leftMostChild;  
    inOrder(p);  
    printf("%d ",r->id);  
    if(p != NULL)  
        p = p->rightSibling;  
    while(p != NULL){  
        inOrder(p);  
        p = p->rightSibling;  
    }  
}
```

# Operations

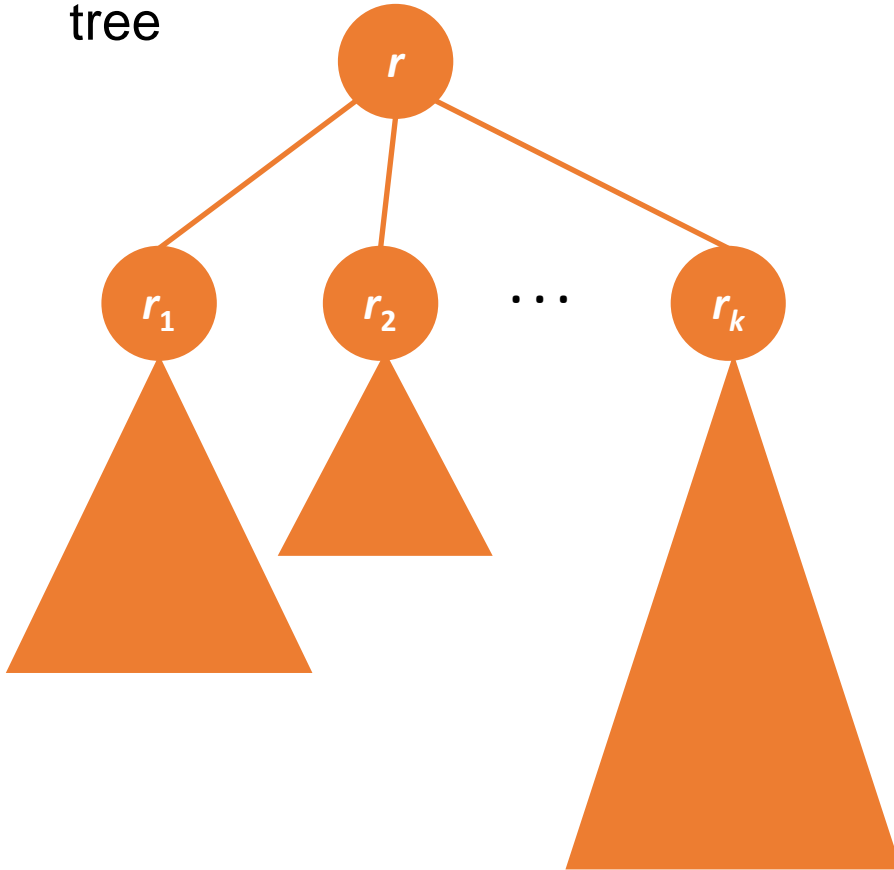
- Post-order traversal



```
void postOrder(Node* r){  
    if(r == NULL) return;  
    Node* p = r->leftMostChild;  
    while(p != NULL){  
        postOrder(p);  
        p = p->rightSibling;  
    }  
    printf("%d ",r->id);  
}
```

# Operations

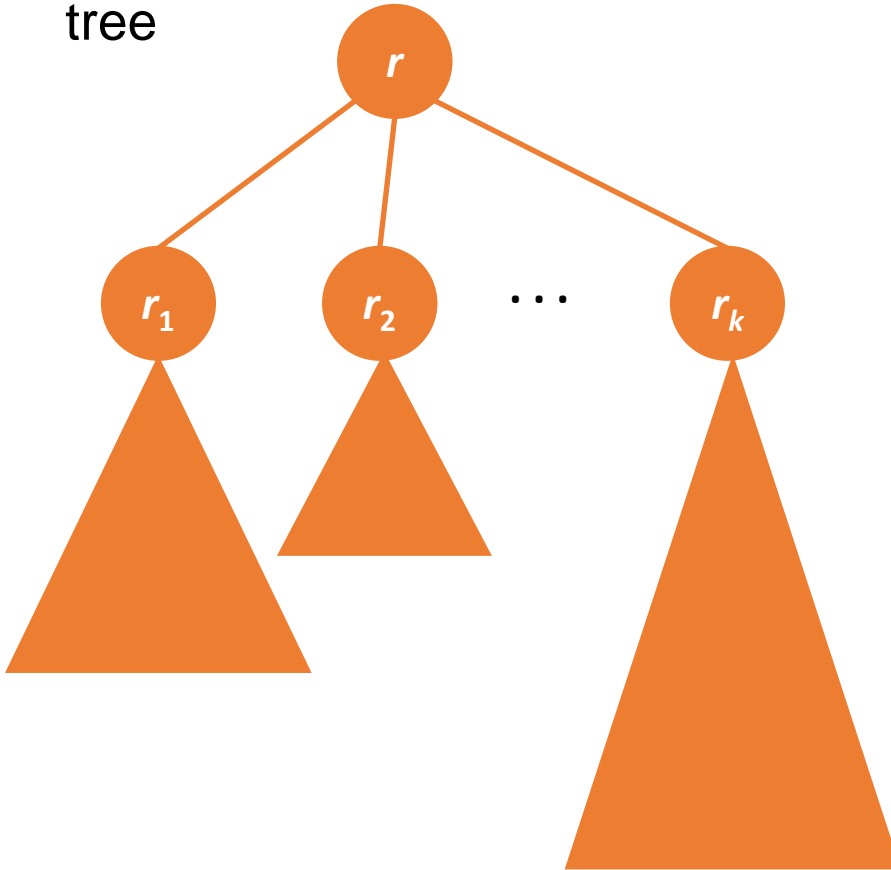
- Count the number of nodes of a tree



```
int count(Node* r){  
    if(r == NULL) return 0;  
    int s = 1;  
    Node* p = r->leftMostChild;  
    while(p != NULL){  
        s += count(p);  
        p = p->rightSibling;  
    }  
    return s;  
}
```

# Operations

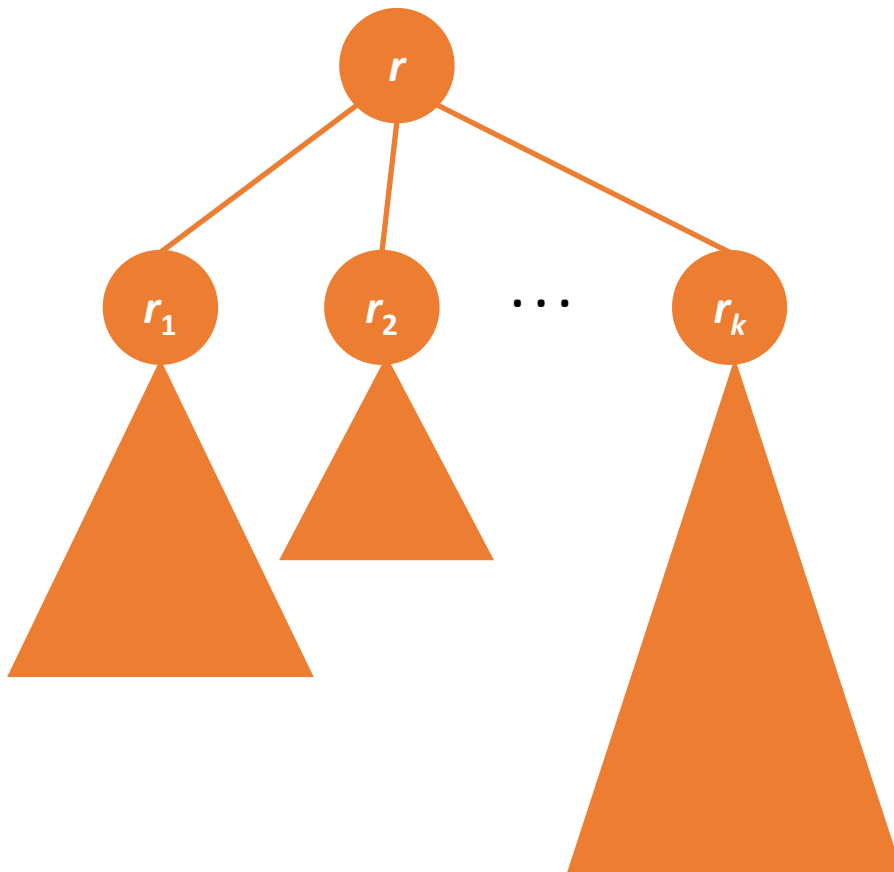
- Count the number of leaves of a tree



```
int countLeaves(Node* r){  
    if(r == NULL) return 0;  
    int s = 0;  
    Node* p = r->leftMostChild;  
    if(p == NULL) s = 1;  
    while(p != NULL){  
        s += countLeaves(p);  
        p = p->rightSibling;  
    }  
    return s;  
}
```

# Operations

- Compute the height of a node

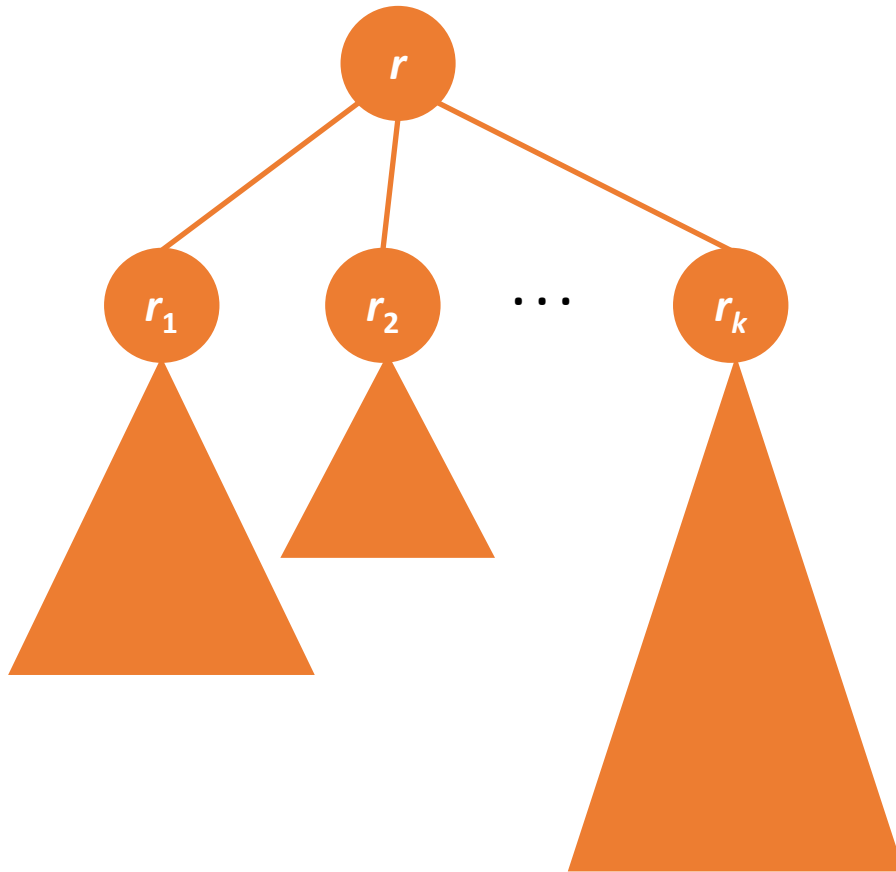


```
int height(Node* p){  
    if(p == NULL) return 0;  
    int maxh = 0;  
    Node* q = p->leftMostChild;  
    while(q != NULL){  
        int h = height(q);  
        if(h > maxh) maxh = h;  
        q = q->rightSibling;  
    }  
    return maxh + 1;  
}
```



# Operations

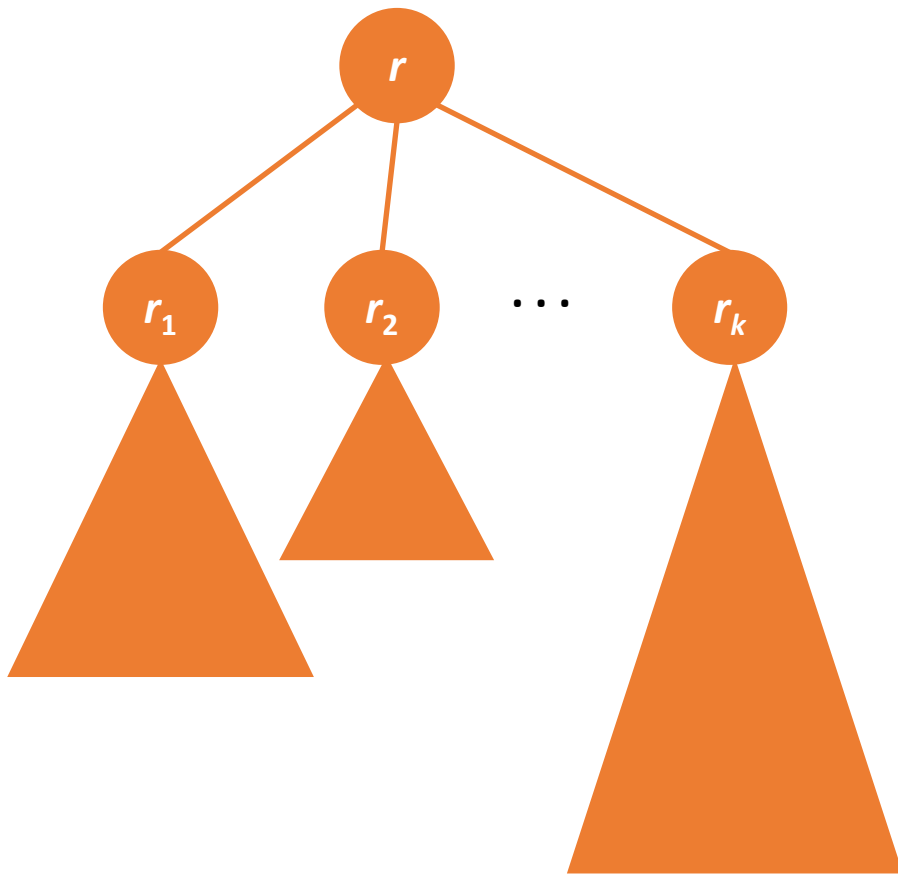
- Compute the depth of a node



```
int depth(Node* r, int v, int d){  
    // d la do sau cua nut r  
    if(r == NULL) return -1;  
    if(r->id == v) return d;  
    Node* p = r->leftMostChild;  
    while(p != NULL){  
        if(p->id == v) return d+1;  
        int dv = depth(p,v,d+1);  
        if(dv > 0) return dv;  
        p = p->rightSibling;  
    }  
    return -1;  
}  
  
int depth(Node* r, int v){  
    return depth(r,v,1);  
}
```

# Operations

- Find the parent of a node



```
Node* parent(Node* p, Node* r){  
    if(r == NULL) return NULL;  
    Node* q = r->leftMostChild;  
    while(q != NULL){  
        if(p == q) return r;  
        Node* pp = parent(p, q);  
        if(pp != NULL) return pp;  
        q = q->rightSibling;  
    }  
    return NULL;  
}
```

# Binary trees

---

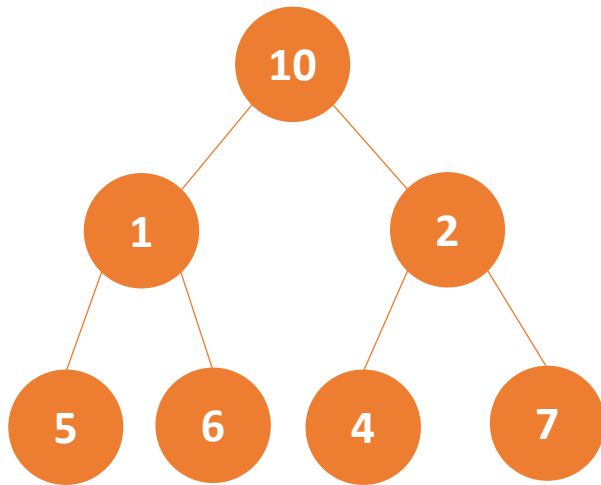
- Each node has at most two children
- Distinct between the left child and the right child

```
struct BNode{  
    int id;  
    BNode* leftChild; // pointer to the left child  
    BNode* rightChild; // pointer fo the right child  
};
```

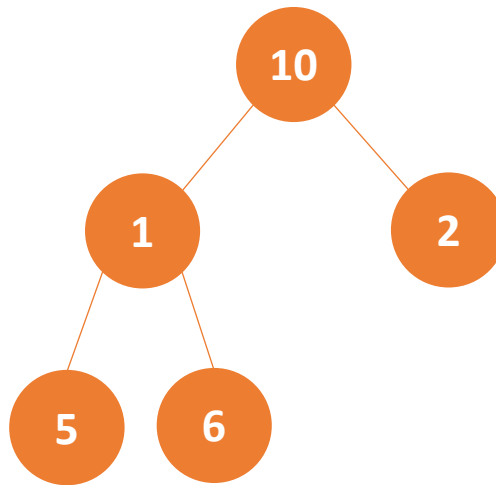
- `leftChild = NULL`: current node does not have the left child
- `rightChild = NULL`: current node does not have the right child

# Binary trees

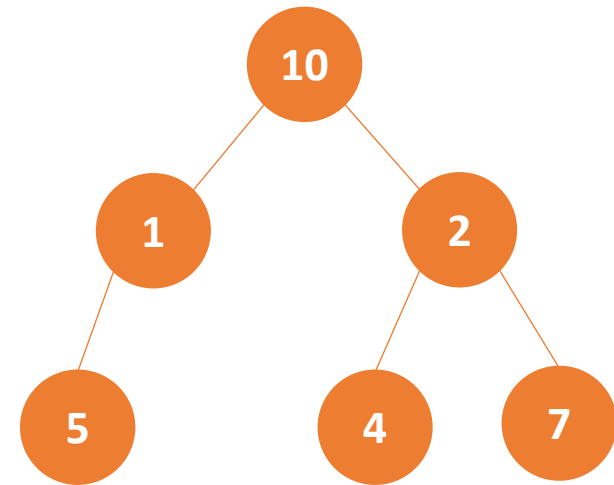
- Classification



Perfect tree



Complete tree



Balanced tree

# Operations on a binary tree

```
void preOrder(BNode* r) {  
    if(r == NULL) return;  
    printf("%d ",r->id);  
    preOrder(r->leftChild);  
    preOrder(r->rightChild);  
}
```

```
void inOrder(BNode* r) {  
    if(r == NULL) return;  
    inOrder(r->leftChild);  
    printf("%d ",r->id);  
    inOrder(r->rightChild);  
}
```

# Operations on a binary tree

```
void postOrder(BNode* r) {  
    if(r == NULL) return;  
    postOrder(r->leftChild);  
    postOrder(r->rightChild);  
    printf("%d ",r->id);  
}
```

```
int count(BNode* r) {  
    if(r == NULL) return 0;  
    return 1 + count(r->leftChild) +  
            count(r->rightChild);  
}
```

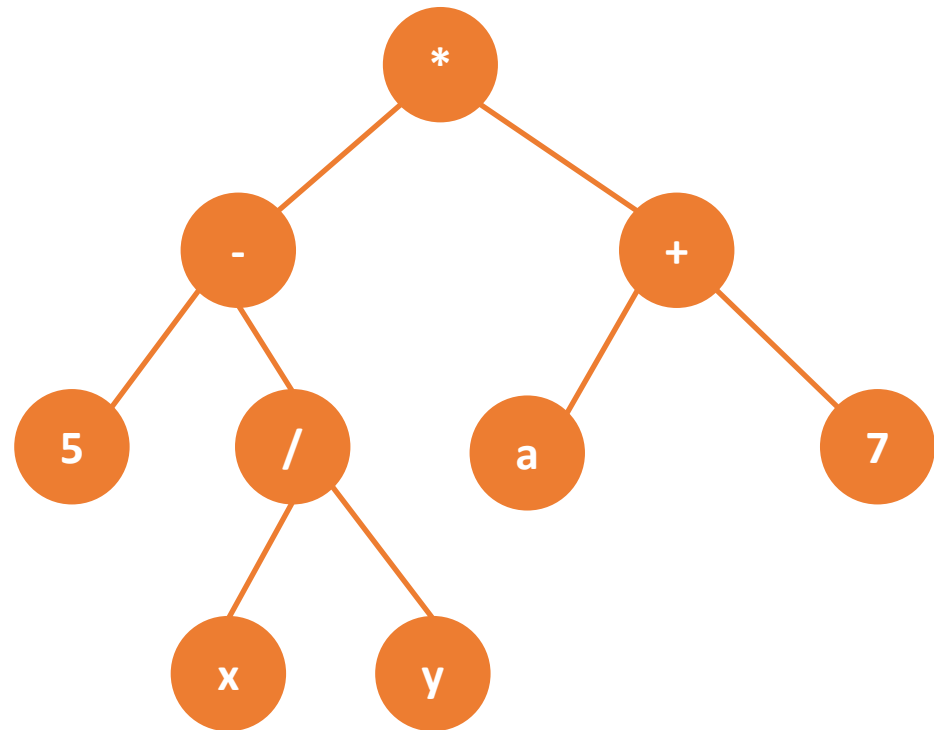
# Expression trees

- Binary tree
  - Internal node are math operations
  - Leaves are operands (variables, constants)
- Infix expression: sequence of elements visited by the in-order traversal:

$$(5 - x/y) * (a + 7)$$

- Postfix expression: sequence of elements visited by the post-order traversal:

$$5 \ x \ y \ / \ - \ a \ 7 \ + \ *$$



# Evaluation of a postfix expression

---

- Initialize a stack  $S$
- Scan elements of the postfix from left to right
- If meet an operand, then push it into the stack  $S$
- If meet an operator ***op***, then pop 2 operands  $A$  and  $B$  out of  $S$ , perform  $C = B \text{ ***op*** } A$ , and then push  $C$  into  $S$
- Termination: the element stays in the stack  $S$  is the value of the given postfix expression