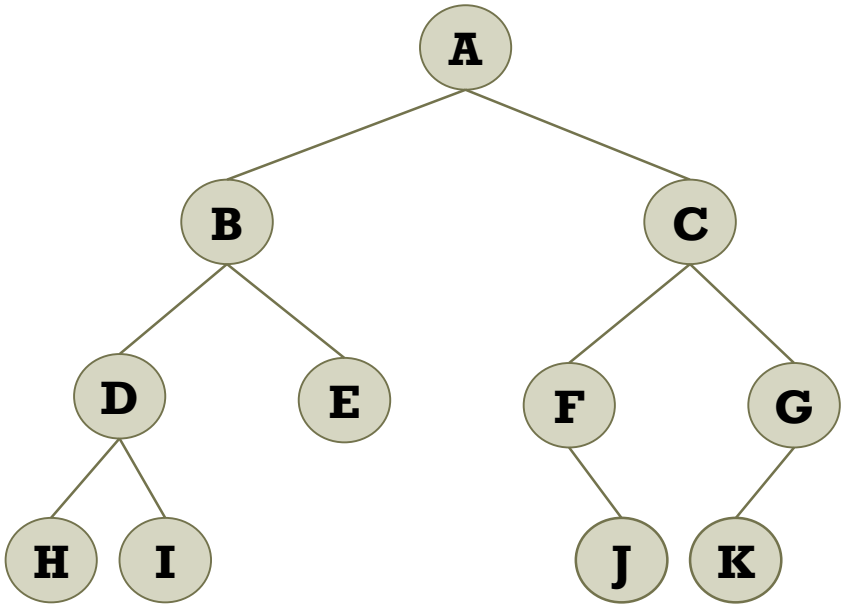
Binary Tree Traversal

Traversal of Binary Tree

- Traversal
  - The process of visiting each node in a tree

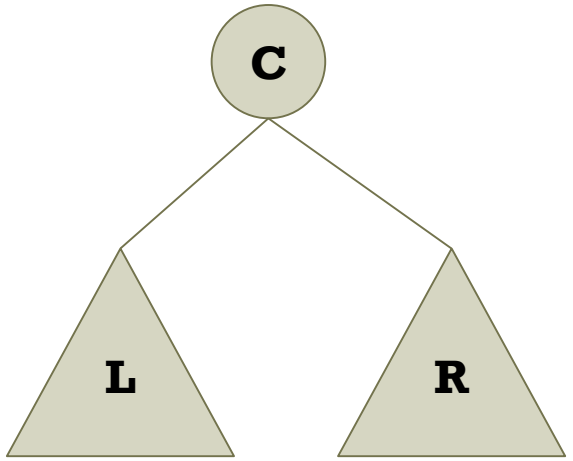
- Why the traversal necessary?
  - Checking whether insertions/deletions work well.
  - Searching a specific node.

- How to visit all nodes once?



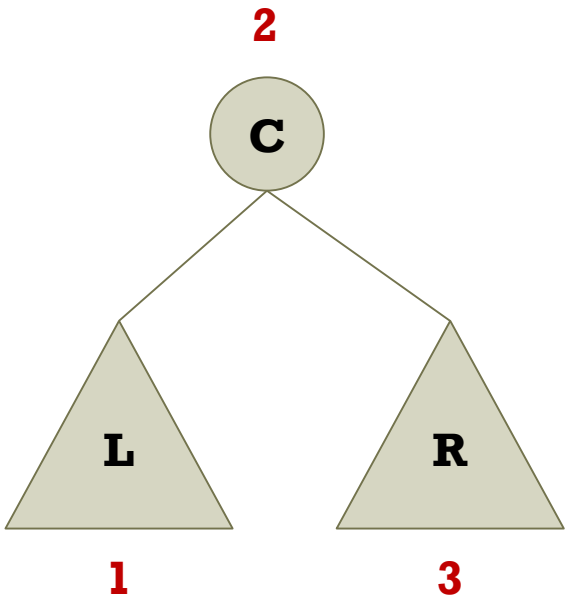
Traversal of Binary Tree

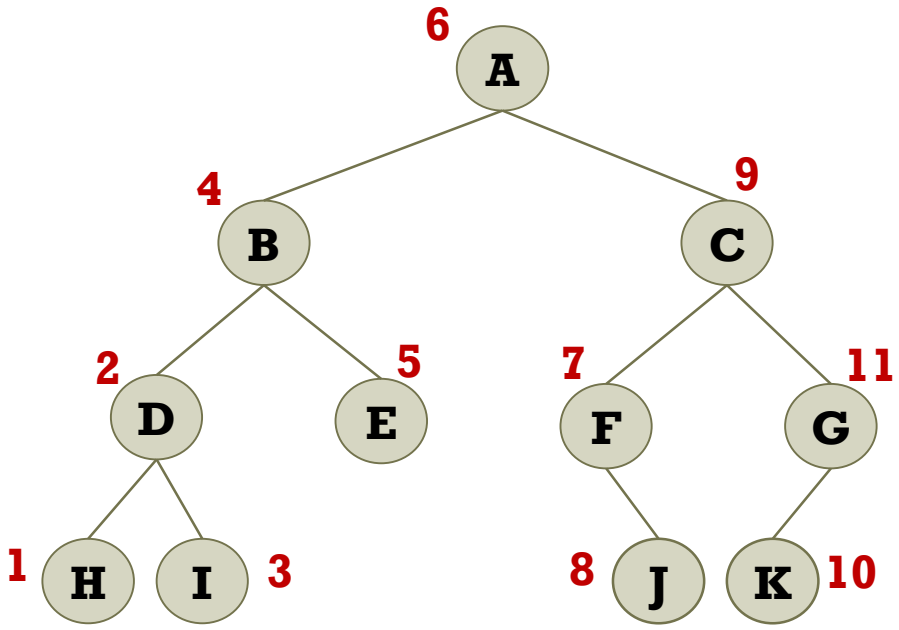
- Traversal methods
  - **Inorder traversal: LCR**
    - Visiting a left subtree, a root node, and a right subtree
  - **Preorder traversal: CLR**
    - Visiting the root node node before subtrees
  - **Postorder traversal: LRC**
    - Visiting subtrees before visiting the root node
  - **Level order traversal**



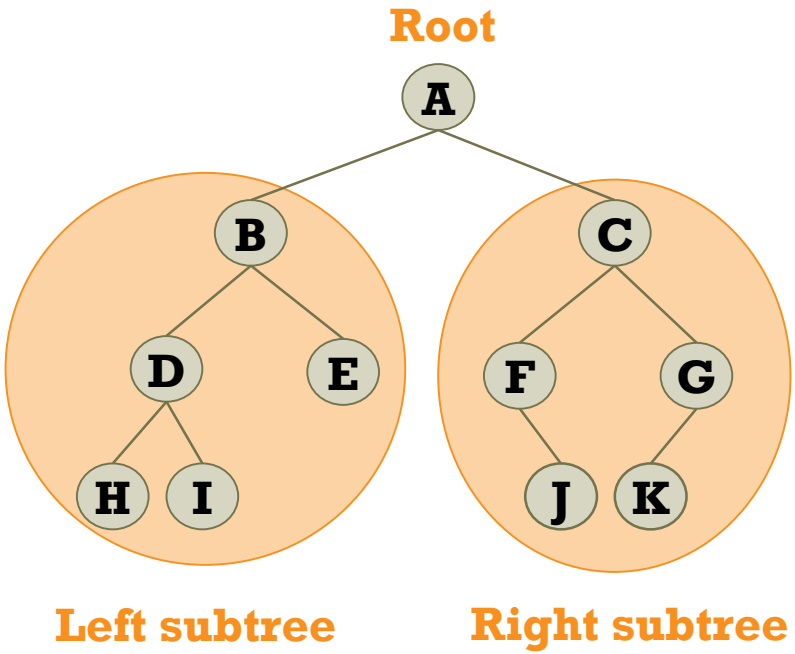
Inorder Traversal (LCR)

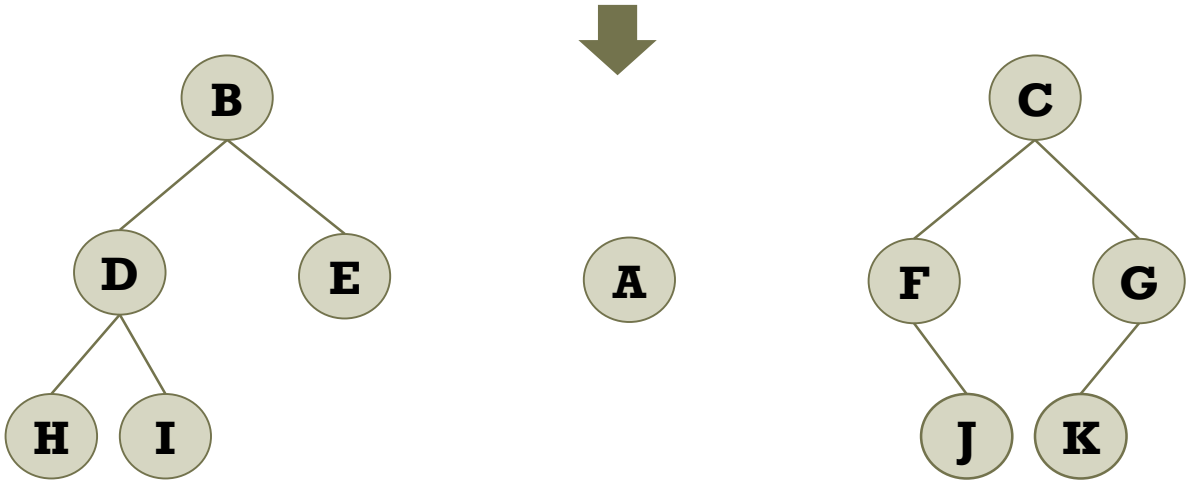
- Algorithm: LCR
  - Step 1: Visiting a left subtree
  - Step 2: Visiting the root node
  - Step 3: Visiting a right subtree



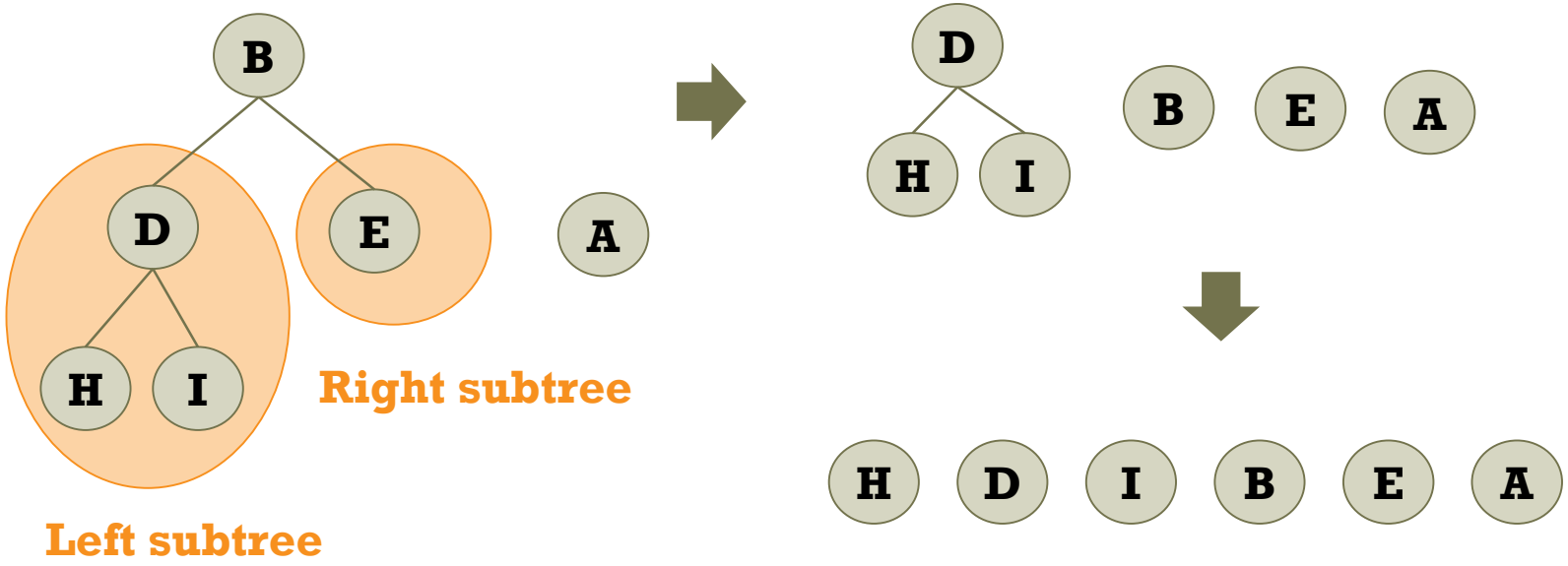


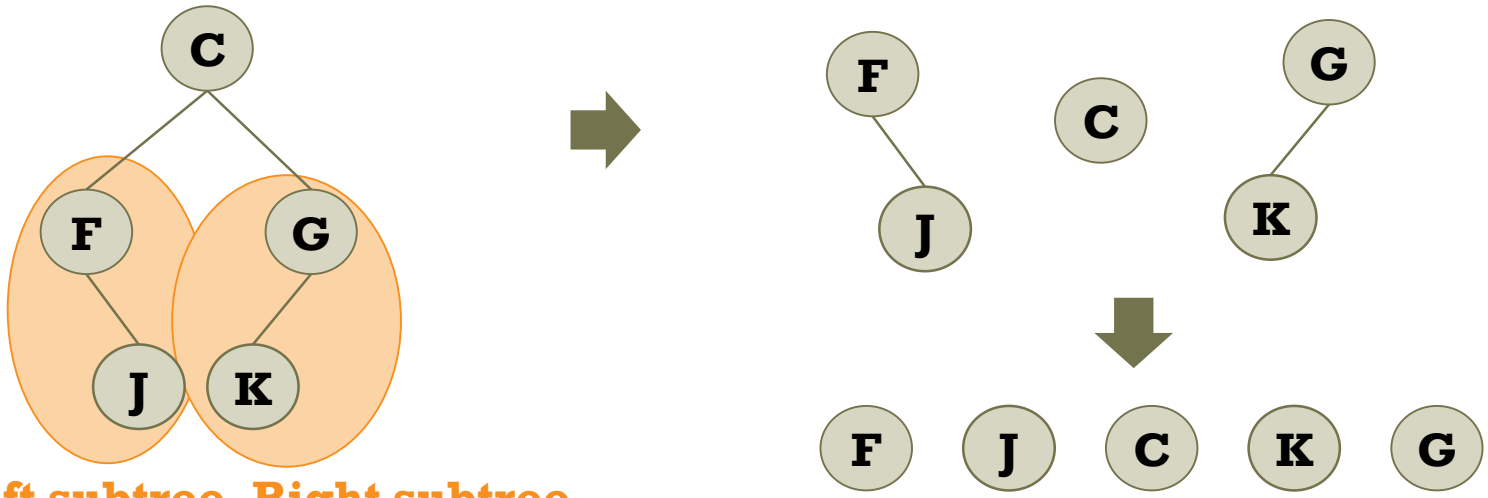
Inorder Traversal (LCR)





Inorder Traversal (LCR)

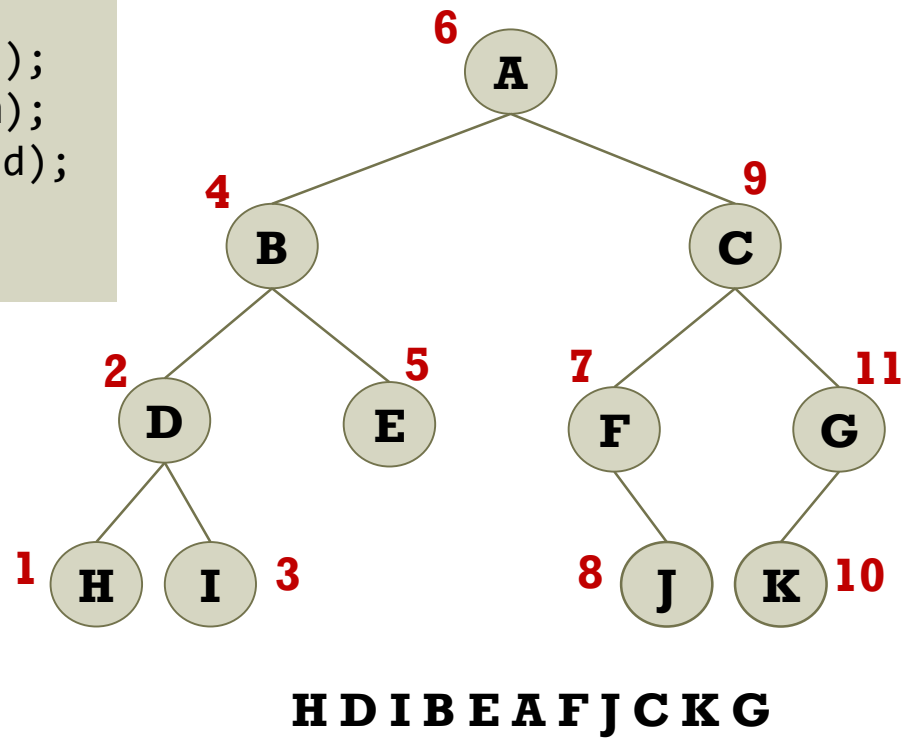




Inorder Traversal (LCR)

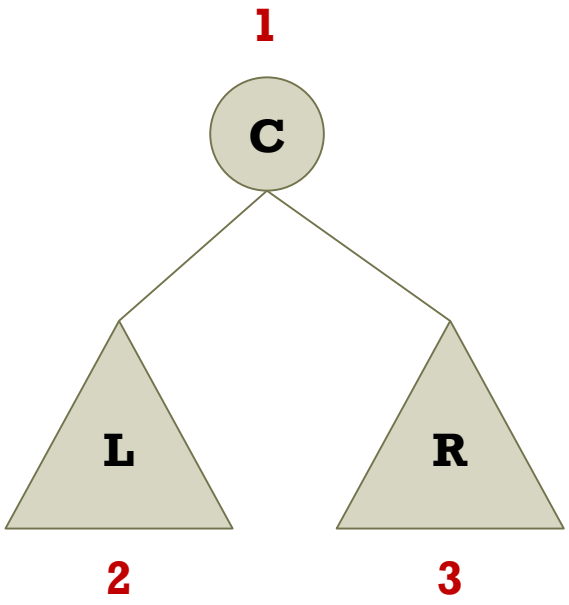
- Algorithm

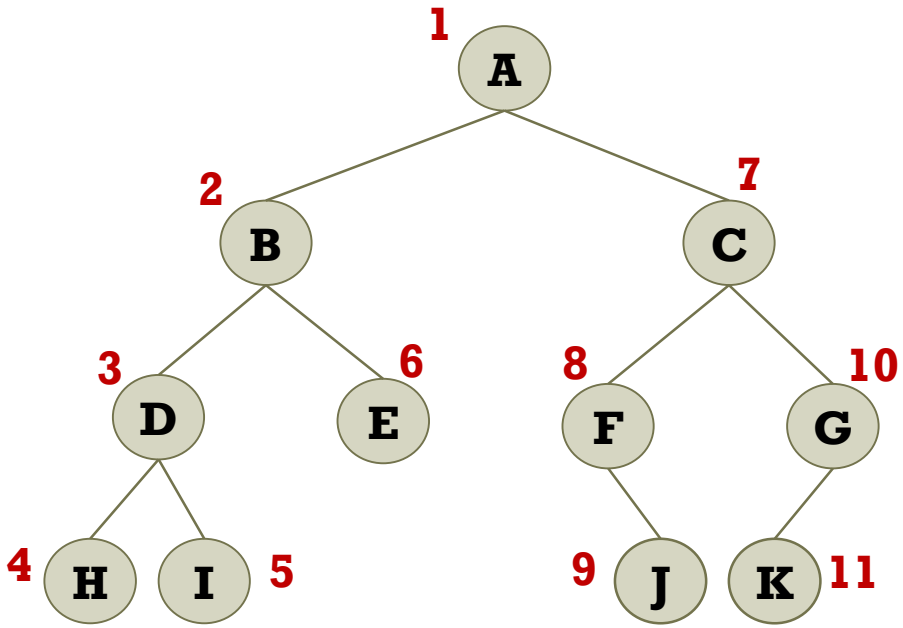
```
void Inorder(BTreeNode* root)
{
    if (root != NULL)
    {
        Inorder(root->left_child);
        printf("%d ", root->item);
        Inorder(root->right_child);
    }
}
```



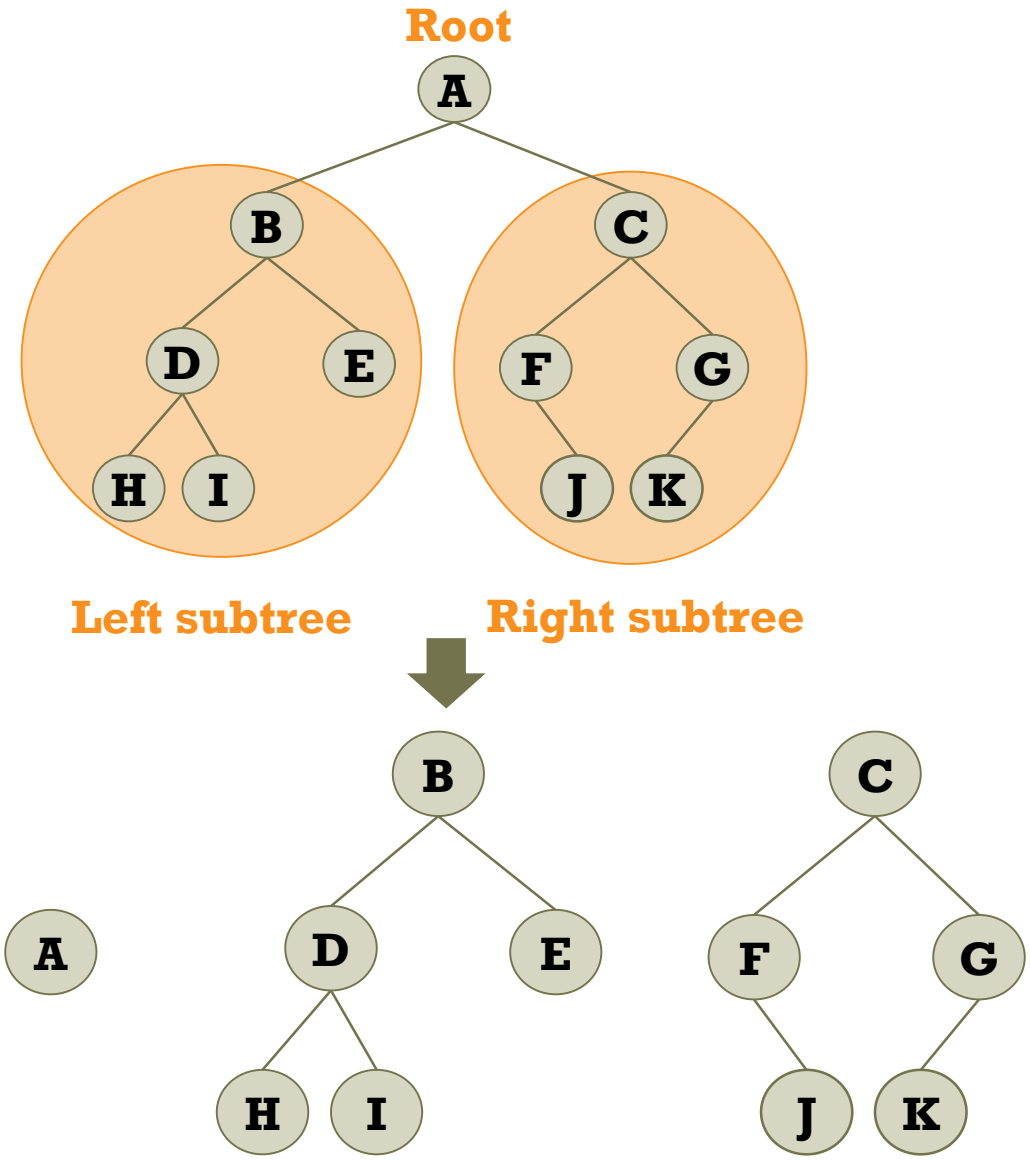
Preorder Traversal (CLR)

- Algorithm: CLR
  - Step 1: Visiting the root node
  - Step 2: Visiting a left subtree
  - Step 3: Visiting a right subtree

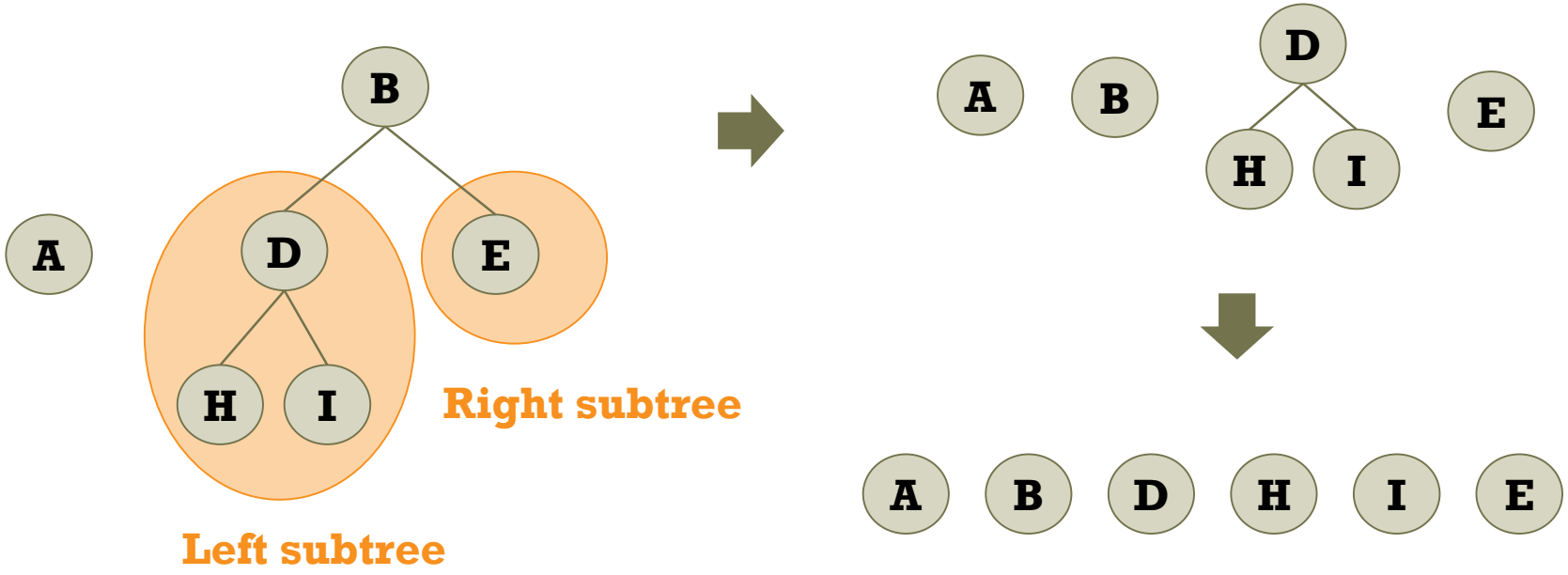


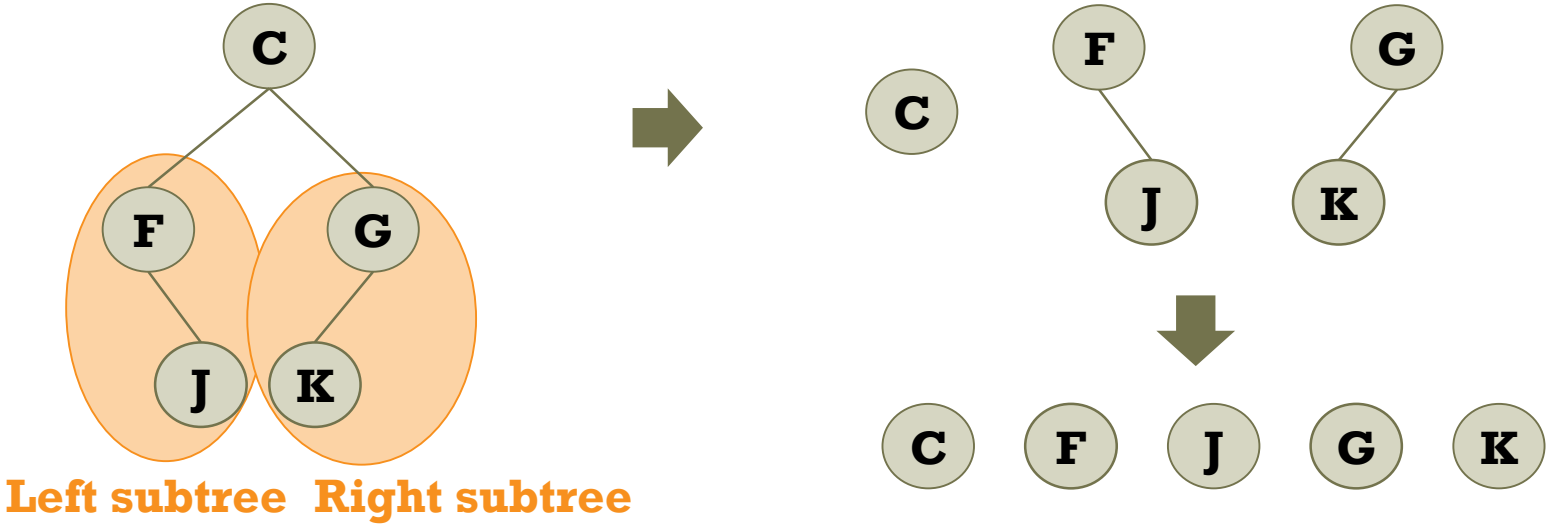


Preorder Traversal (CLR)



Preorder Traversal (CLR)

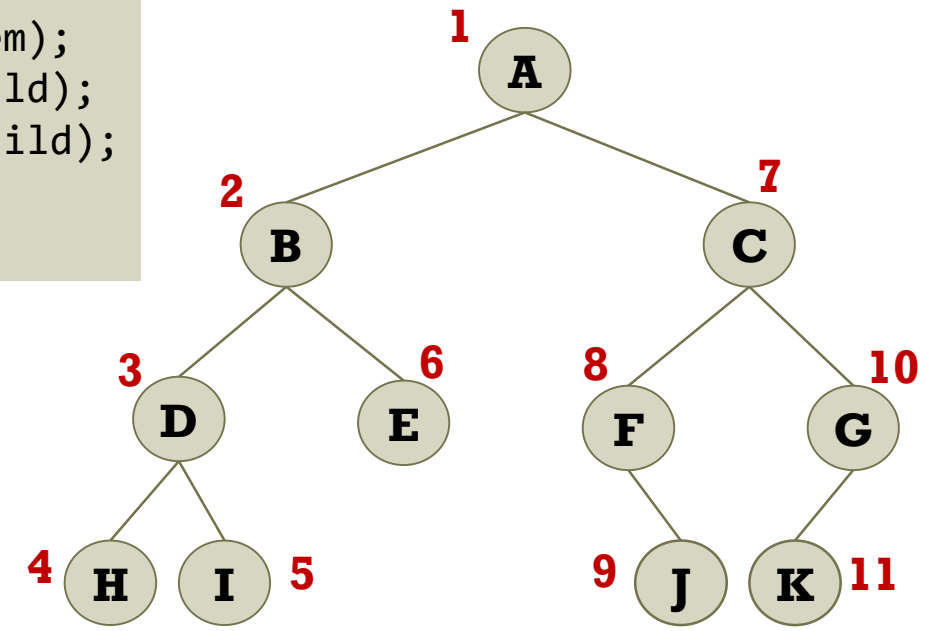




Preorder Traversal (CLR)

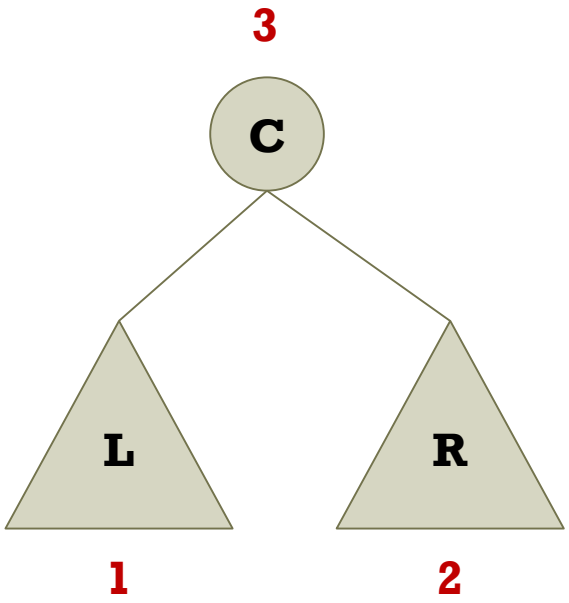
- Algorithm: CLR

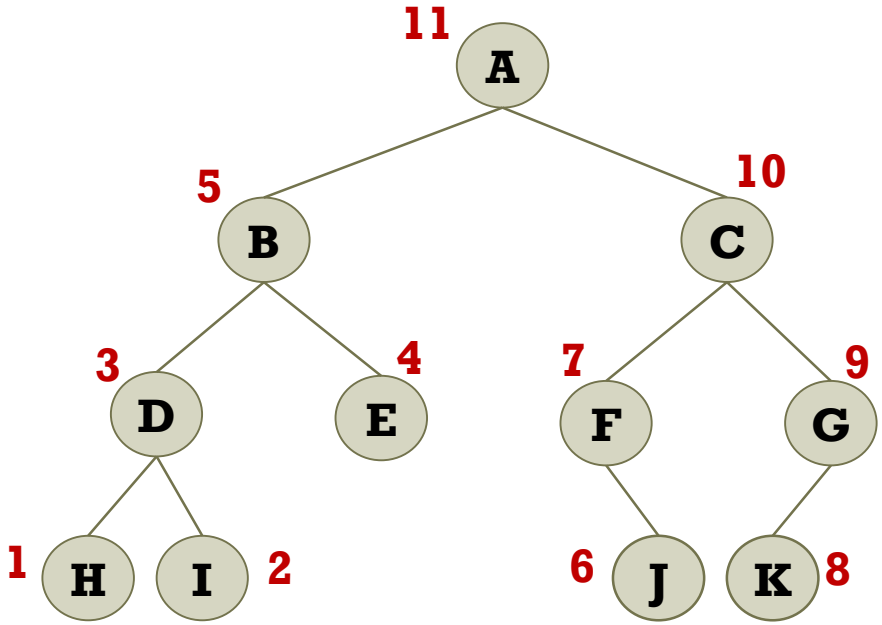
```
void Preorder(BTreeNode* root)
{
    if (root != NULL)
    {
        printf("%d ", root->item);
        Preorder(root->left_child);
        Preorder(root->right_child);
    }
}
```



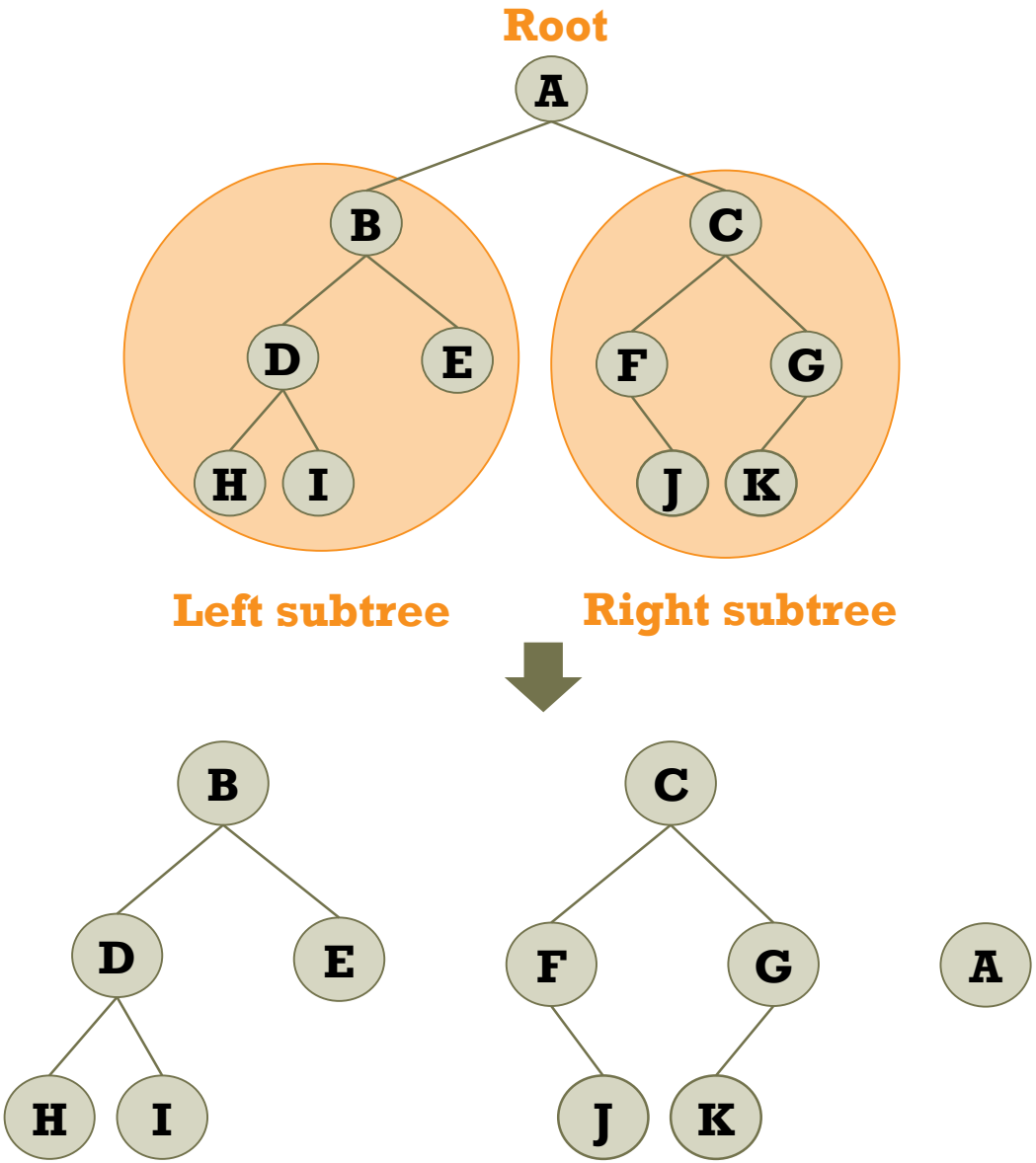
Postorder Traversal (LRC)

- Algorithm: LRC
  - Step 1: Visiting a left subtree
  - Step 2: Visiting a right subtree
  - Step 3: Visiting the root node

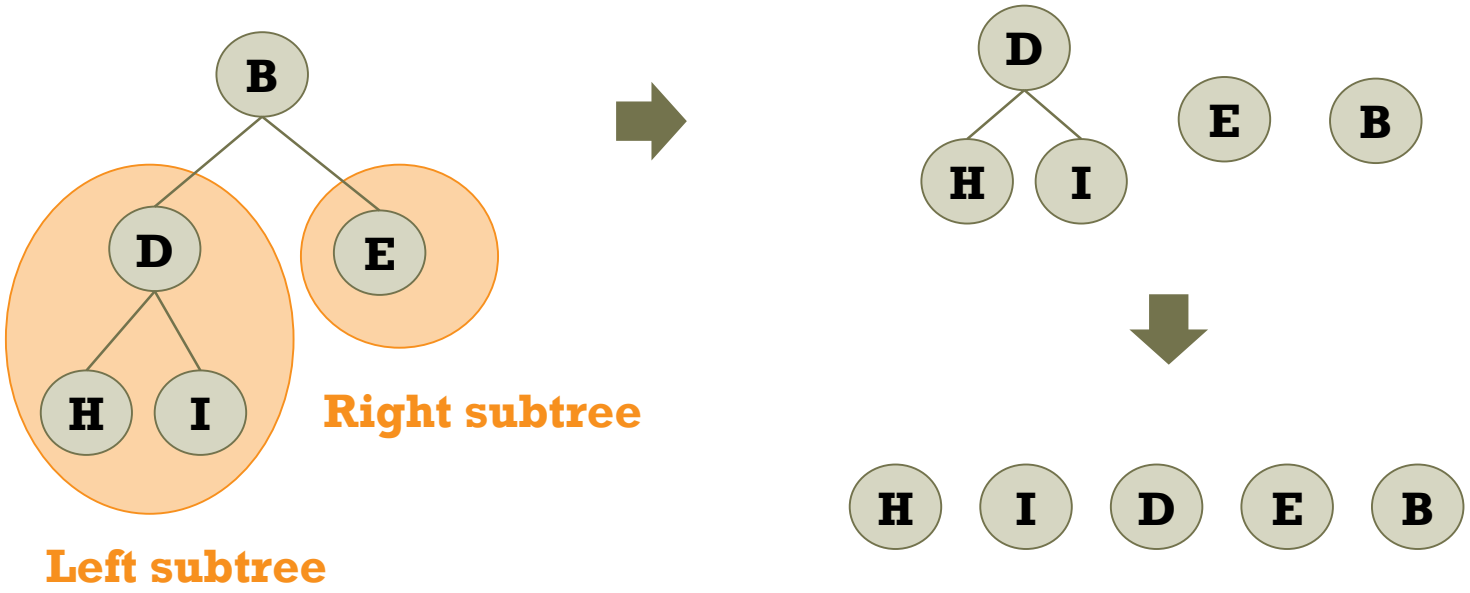


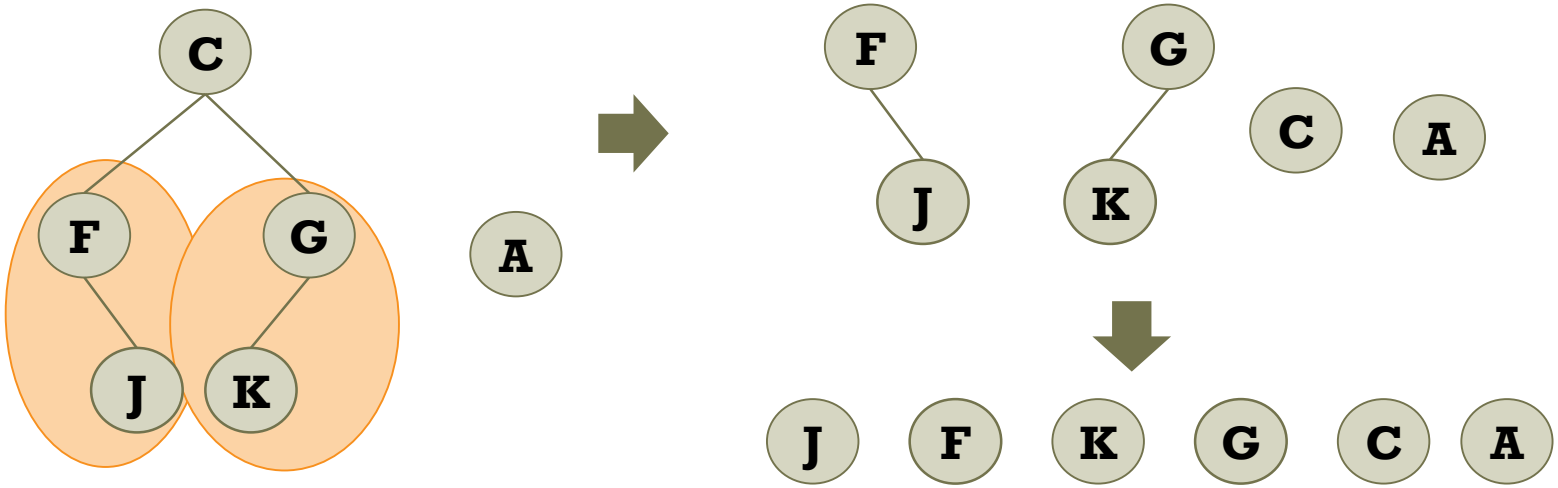


Postorder Traversal (LRC)



Postorder Traversal (LRC)

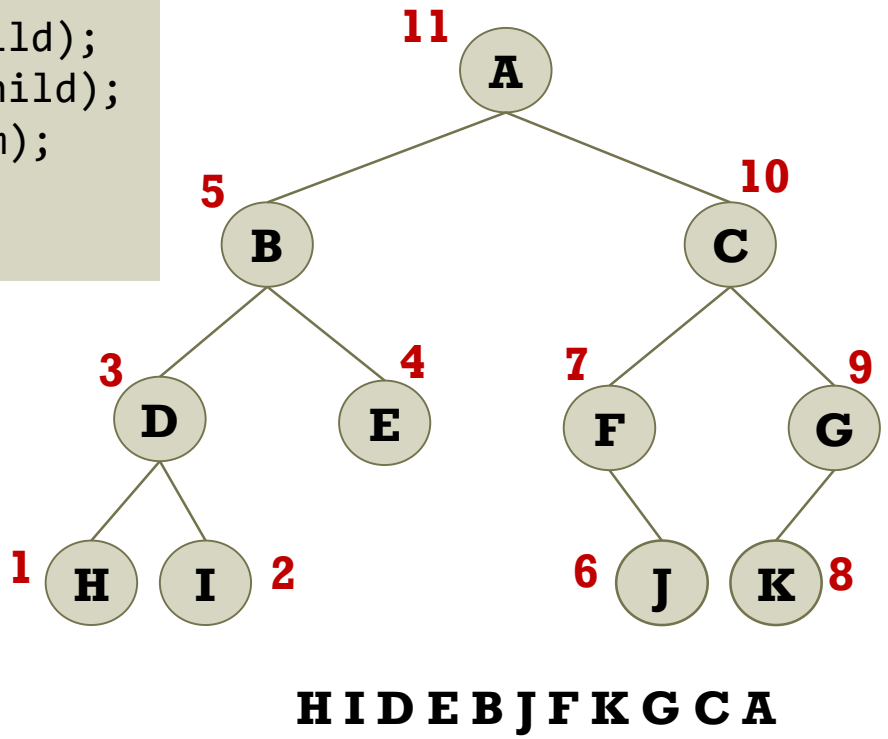




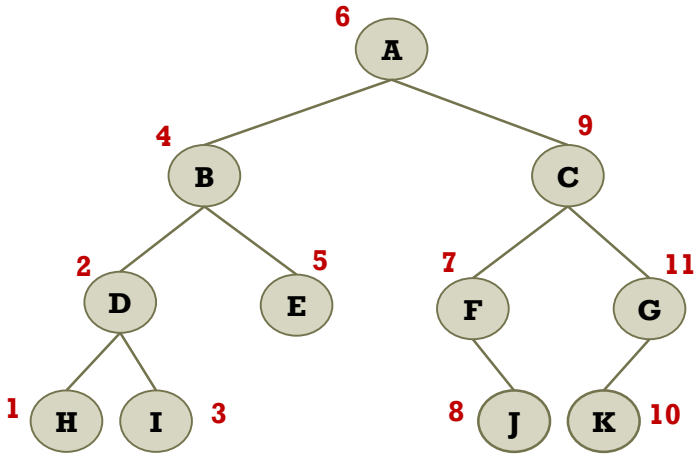
Postorder Traversal (LRC)

- Algorithm: LRC

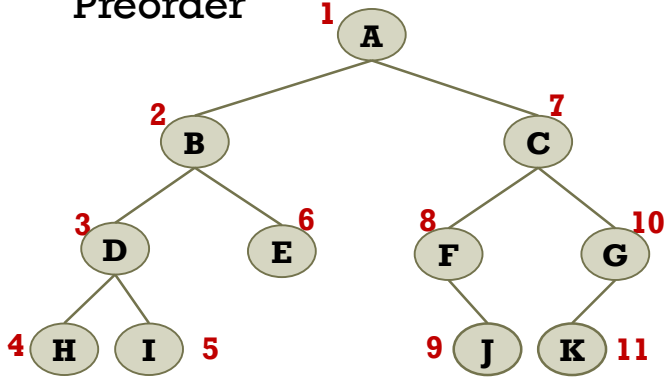
```
void Postorder(BTreeNode* root)
{
    if (root != NULL)
    {
        Postorder(root->left_child);
        Postorder(root->right_child);
        printf("%d ", root->item);
    }
}
```



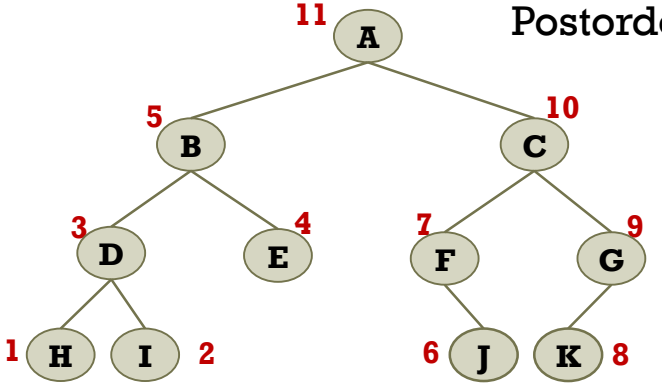
Inorder



Preorder

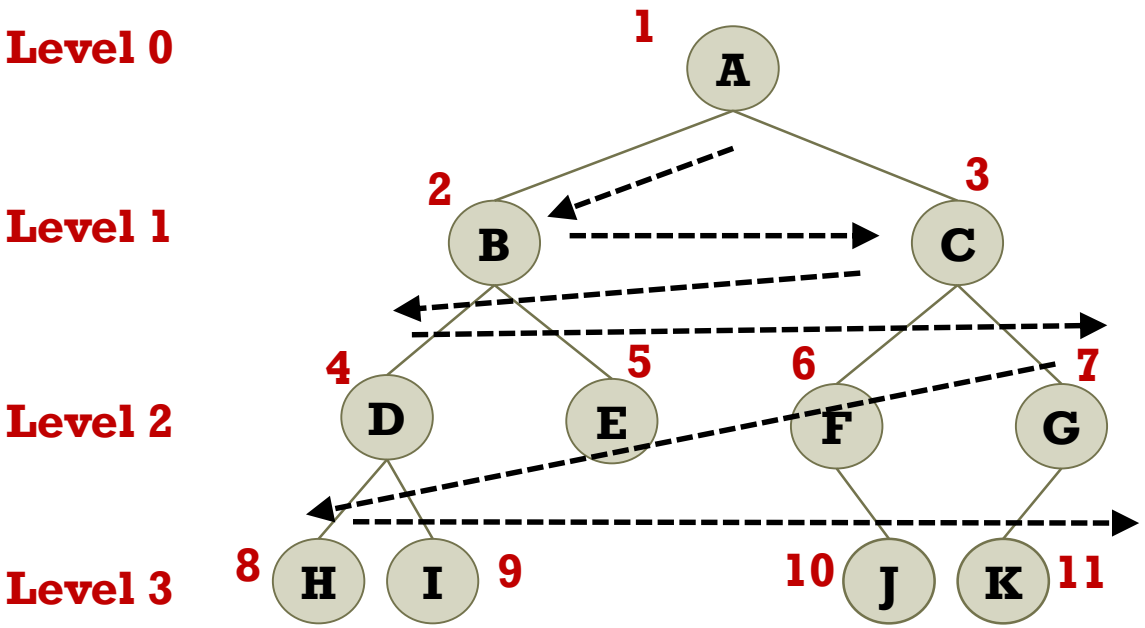


Postorder



Level Order Traversal

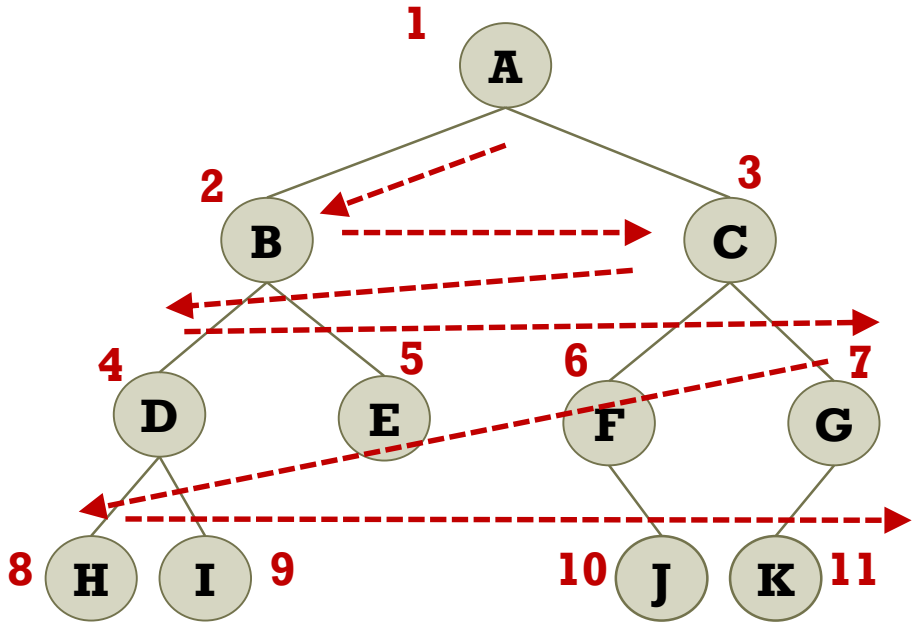
- Algorithm
  - For each level, visit nodes from the left to right direction.



A B C D E F G H I J K

Level Order Traversal

- Algorithm
  - Traverse a tree by using a queue (FIFO)
    - When dequeuing a node, enqueue its children from the left to the right direction.



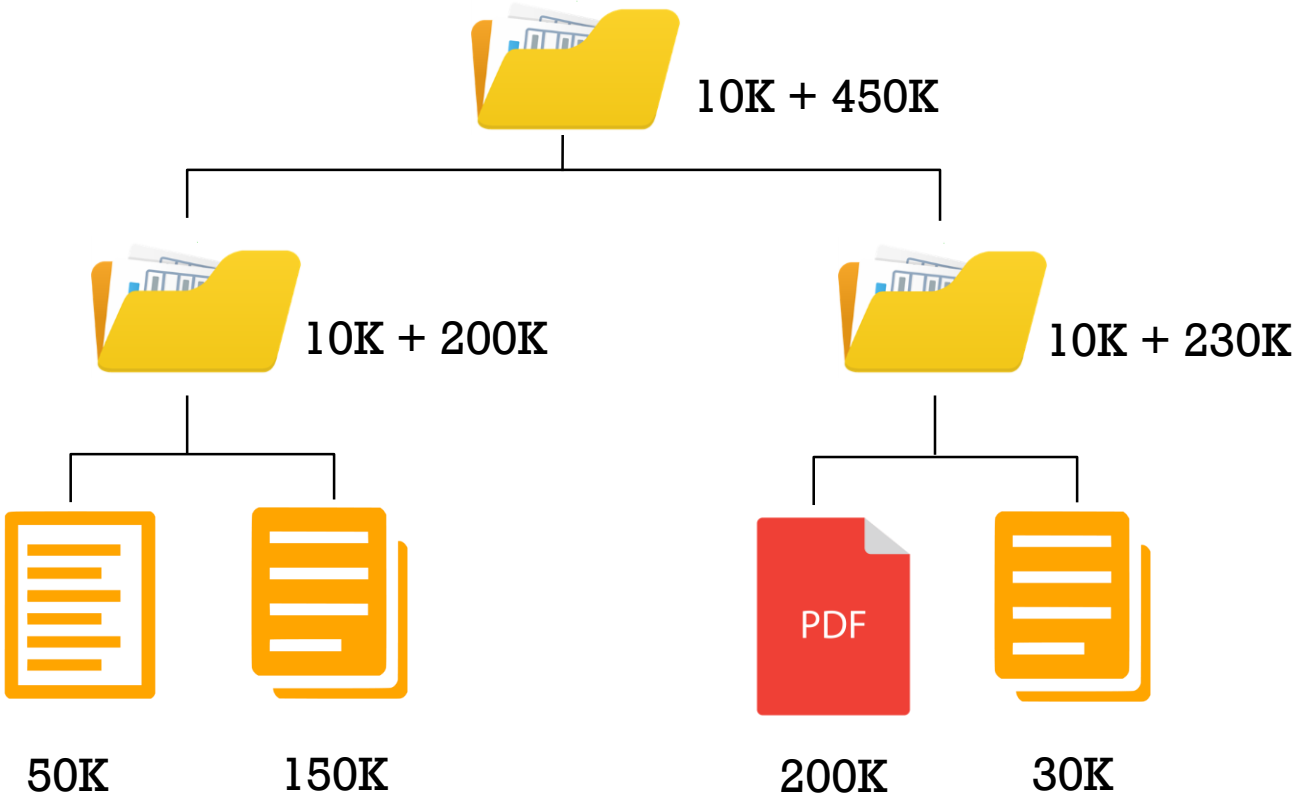
EnQueue A	A				
EnQueue B, C	B	C			
EnQueue D, E	C	D	E		
EnQueue F, G	D	E	F	G	
EnQueue H, I	E	F	G	H	I
EnQueue -	F	G	H	I	
EnQueue J	G	H	I	J	
EnQueue K	H	I	J	K	

Level Order Traversal

```
void Levelorder(BTreeNode* root)
{
    Queue queue;
    if (root == NULL) return;

    InitQueue(&queue);
    EnQueue(&queue, root);
    while (!IsEmpty(&queue))
    {
        root = Peek(&queue);
        DeQueue(&queue);

        printf("%d ", root->item);
        if (root->left_child != NULL)
            EnQueue(&queue, root->left_child);
        if (root->right_child != NULL)
            EnQueue(&queue, root->right_child);
    }
}
```

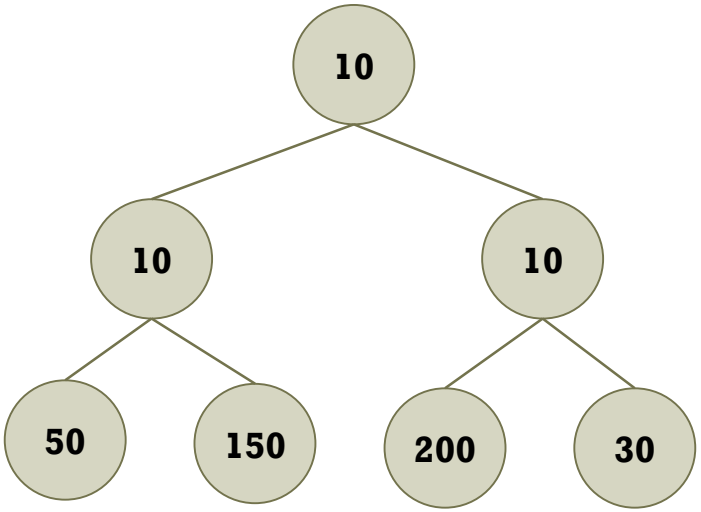
Calculating Directory Size

- How to accumulate directory size?
  - Each file has different size, and each directory has 10K.



Calculating Directory Size

```
// Make use of postorder traversal.
int CalDirectorySize(BTreeNode *root)
{
    int left_size, right_size;
    if (root == NULL) return 0;
    else {
        left_size = CalDirectorySize(root->left_child);
        right_size = CalDirectorySize(root->right_child);
        return (root->item + left_size + right_size);
    }
}
```

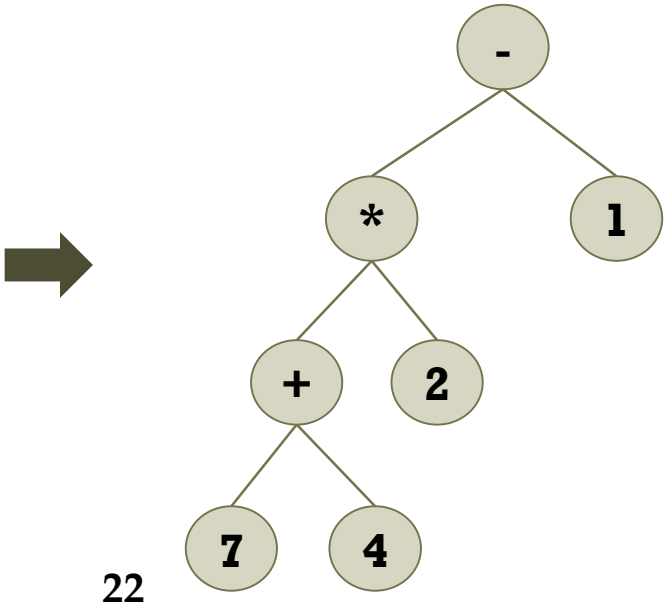


Binary Expression Tree

- Infix notation: X + Y
  - Operators are written in-between their operands.
  - Need extra information to make the order of evaluation of the operators clear.

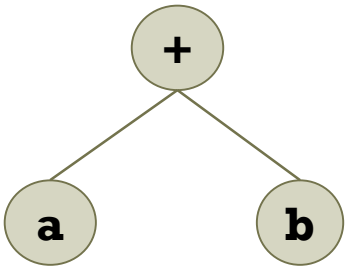
- Example: (7 + 4) * 2 – 1
  - The expression tree is easier to understand than infix notation.

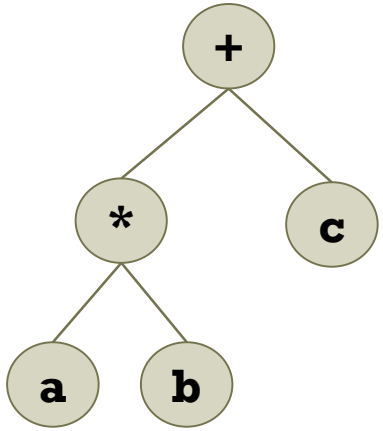
(7 + 4) * 2 - 1

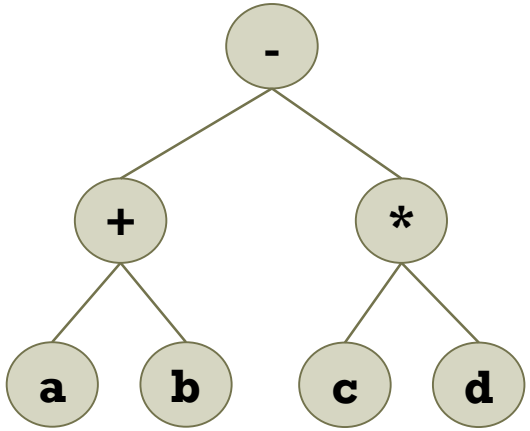


Binary Expression Tree

- Definition
  - Representing an expression as a tree.
    - Non-leaf node: operator, leaf node: operand



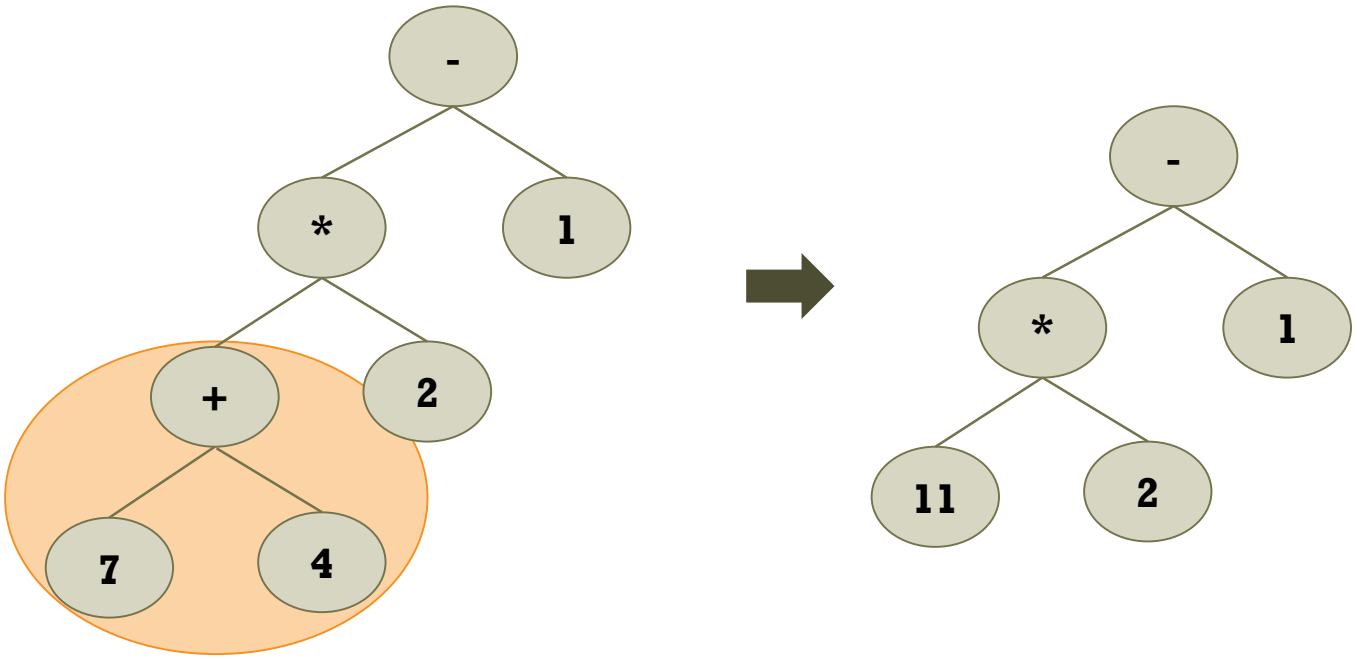




	a+b	a*b+c	a+b-c*d
Infix	a+b	a*b+c	a+b-c*d
Prefix	+ab	+*abc	-+ab*cd
Postfix	ab+	ab*c+	ab+cd*-

Calculating Expression Tree

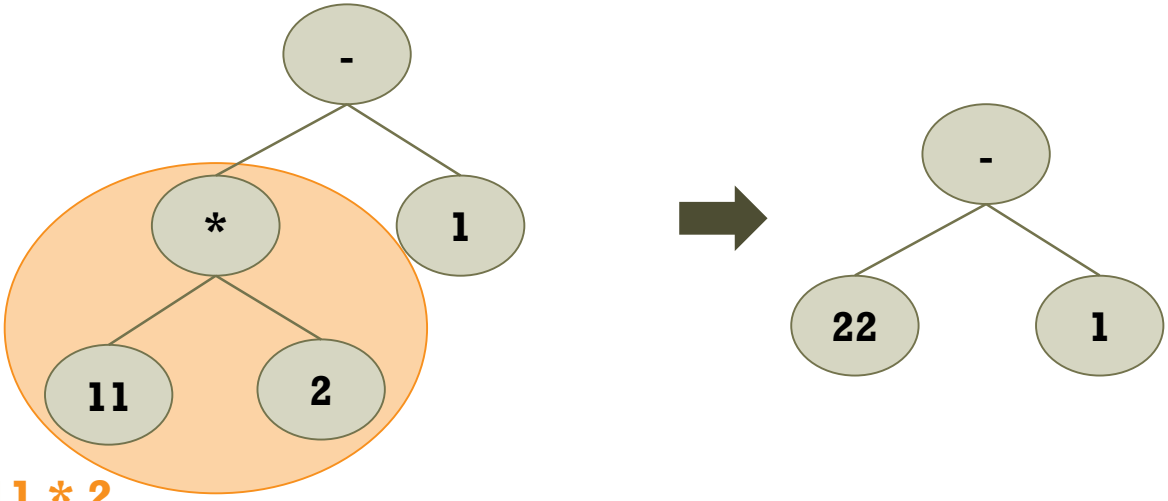
- Calculate 7 + 4 and update the node.



Calculating 7 + 4

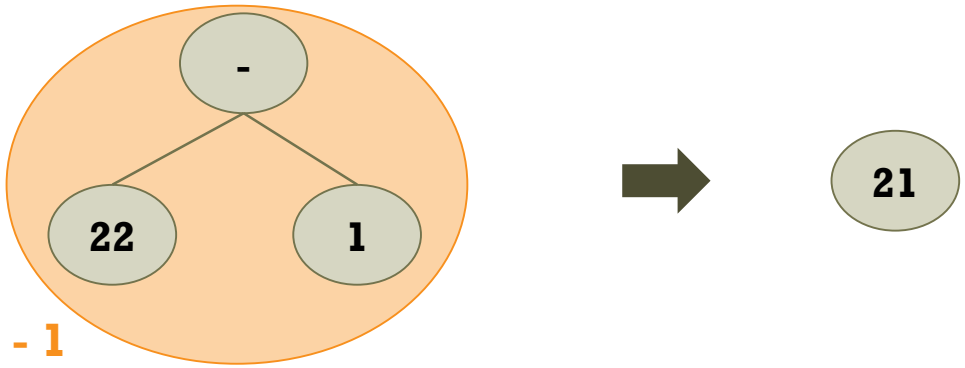
Calculating Expression Tree

- Calculate 11 * 2 and update the node.



Calculating 11 * 2

- Calculate 22 - 1 and update the node.



Calculating 22 - 1

Calculating Expression Tree

```
int CalculateExpTree(BTreeNode * root)
{
    int op1, op2;
    if (root == NULL) return 0;
    if (root->left_child == NULL && root->right_child == NULL)
        return root->item;

    op1 = CalculateExpTree(root->left_child);
    op2 = CalculateExpTree(root->right_child);

    switch (root->item)
    {
        case '+':   return op1 + op2;
        case '-':   return op1 - op2;
        case '*':   return op1 * op2;
        case '/':   return op1 / op2;
    }
    return 0;
}
```

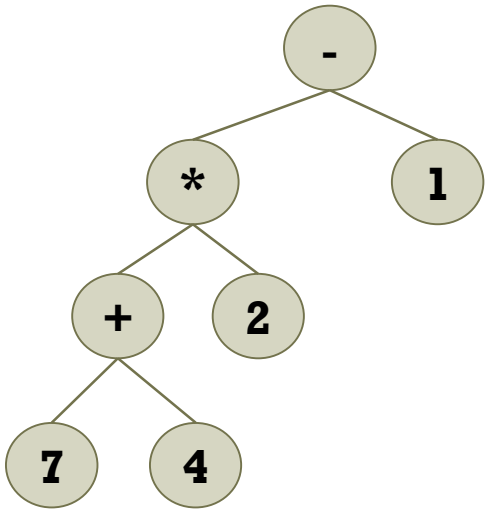
Building Expression Tree

- Overall procedure

Infix notation → **Postfix notation** → **Expression tree**

(7 + 4) * 2 - 1

7 4 + 2 * 1 -



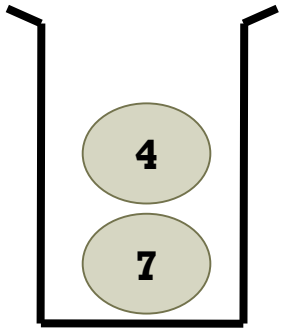
- Infix notation → postfix notation
  - Use a stack (See Lecture Note 3).

- Postfix notation → expression tree
  - Build a tree incrementally using a stack.

Building Expression Tree

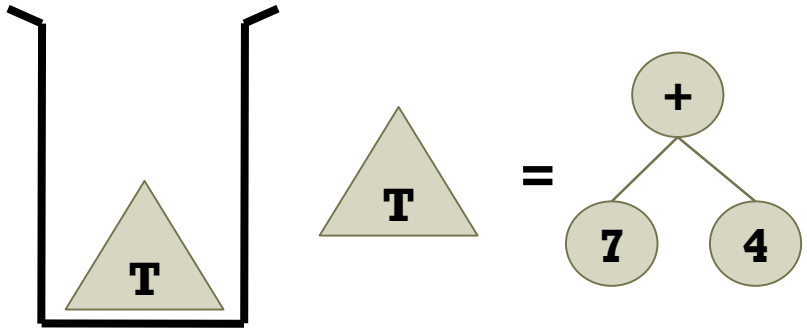
- Push nodes from operands until finding a operator.

```
7 4 + 2 * 1 -
  ^
```



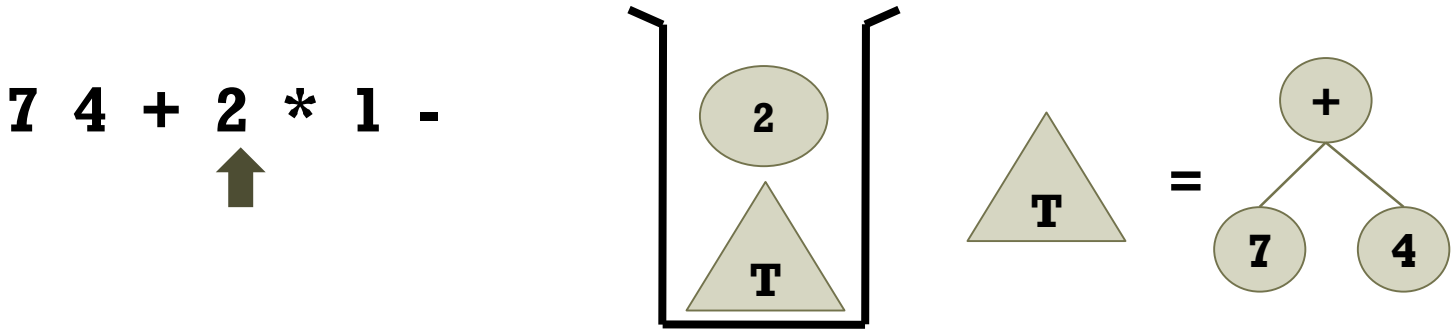
- Pop two nodes and push an expression tree.

```
7 4 + 2 * 1 -
    ^
```

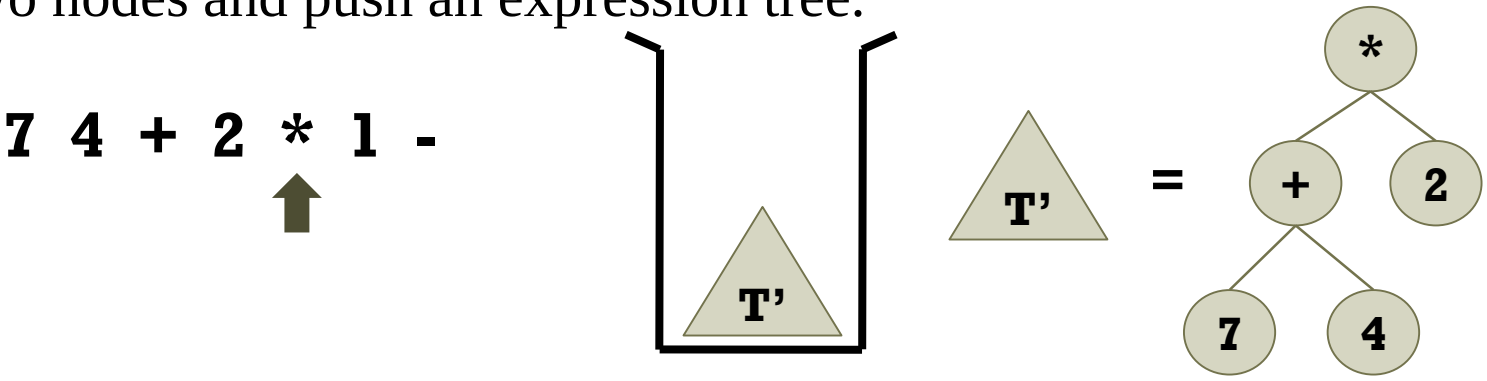


Building Expression Tree

- Push nodes from operands until finding a operator.

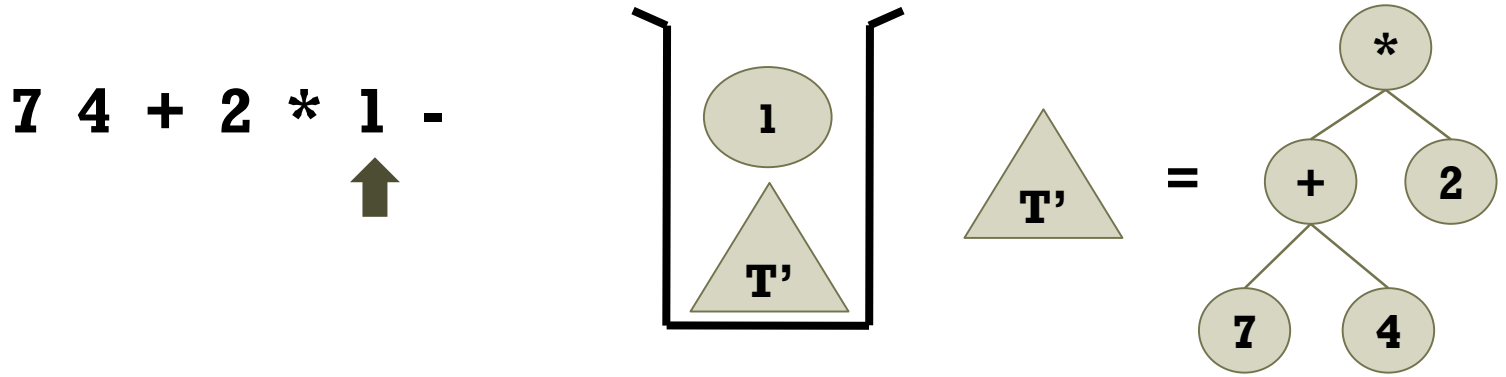


- Pop two nodes and push an expression tree.

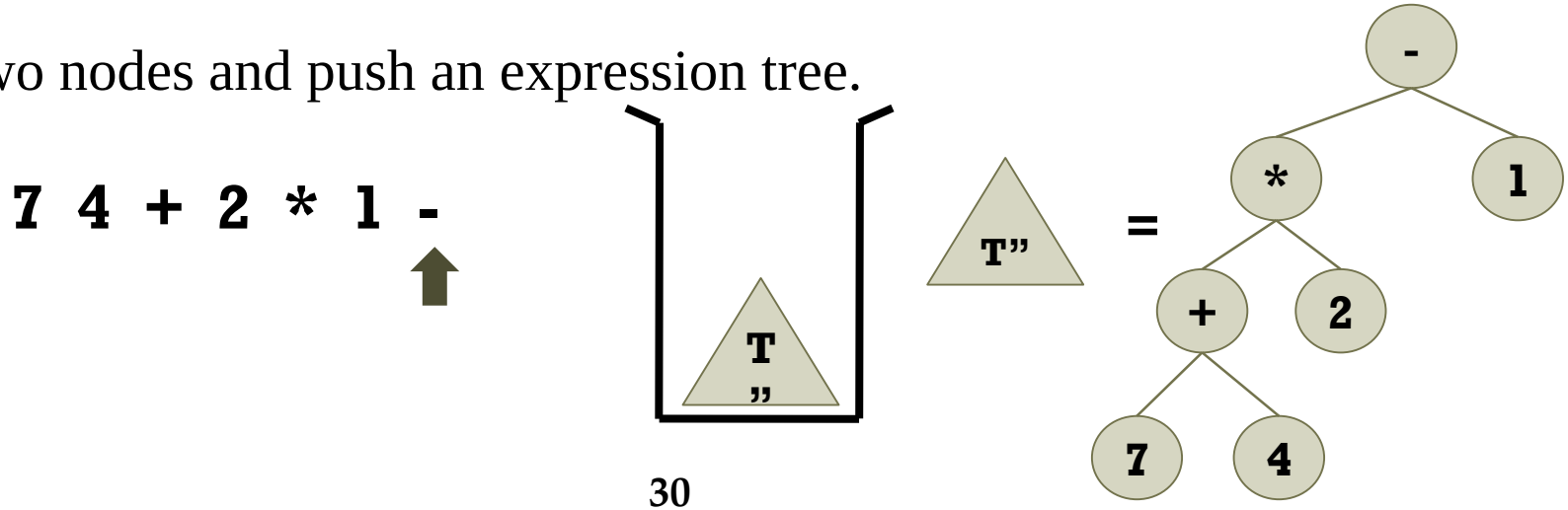


Building Expression Tree

- Push nodes from operands until finding a operator.



- Pop two nodes and push an expression tree.

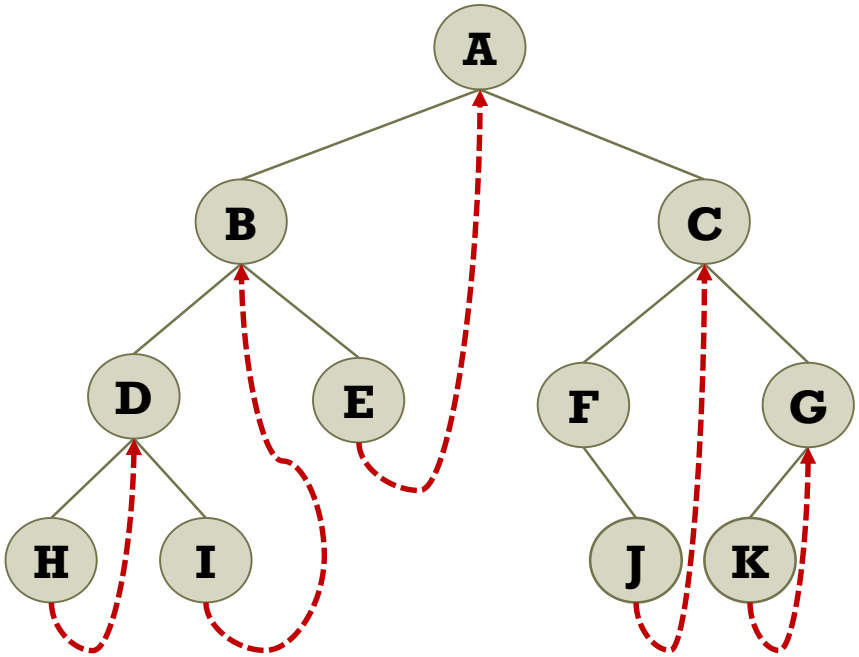


Building Expression Tree

```
BTreeNode * MakeExpTree(char* exp, int len)
{
    Stack stack;
    BTreeNode * node, *right_node, *left_node;

    InitStack(&stack);
    for (int i = 0; i < len; i++) {
        if ('0' <= exp[i] && exp[i] <= '9')
            node = CreateNode(exp[i]);
        else {
            right_node = Peek(&stack), Pop(&stack);
            left_node = Peek(&stack), Pop(&stack);

            node = CreateNode(exp[i]);
            CreateRightSubtree(node, right_node);
            CreateLeftSubtree(node, left_node);
        }
        Push(&stack, node);
    }
    return Peek(&stack);
}
```

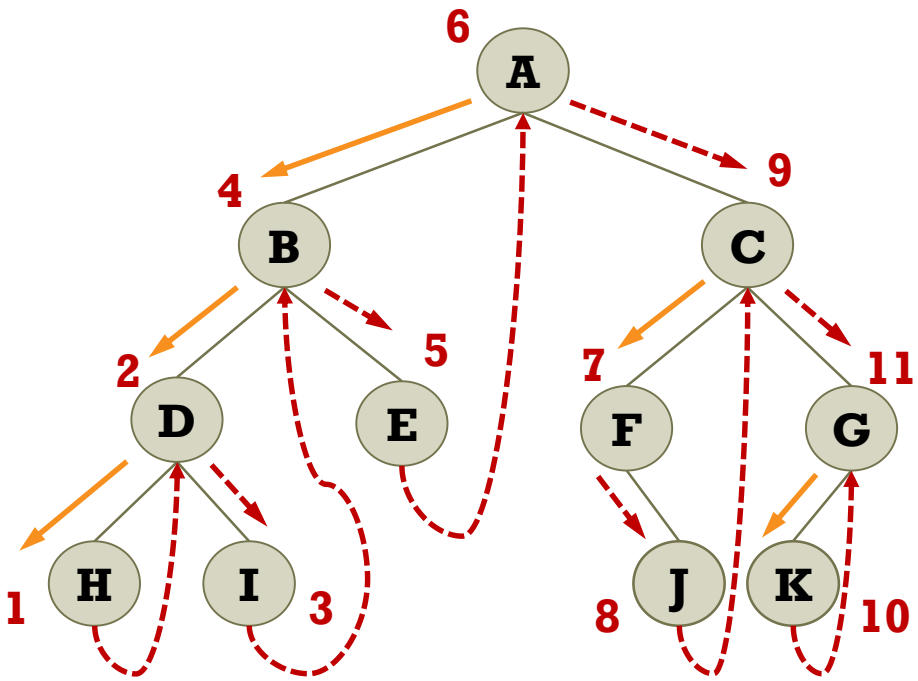
Threaded Binary Tree

- Variation of binary tree
  - Modify a binary tree to cheaply find its inorder successor.
    - Have special threading links (dashed arrows).
  - It can be faster traversal than the recursive version.



Threaded Binary Tree

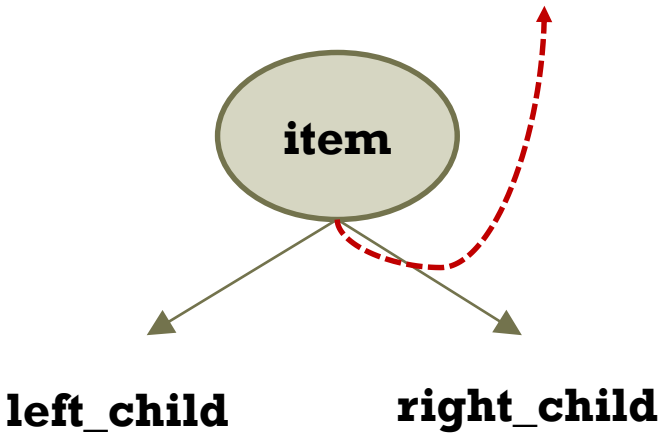
- Inorder traversal
  - Traverse a tree until finding the leftmost node.
    - Solid arrows mean left-side movements.
  - Traverse a tree by a threaded link or a right-side child pointer.
    - Dashed arrows mean right-side movements.



Node in Threaded Binary Tree

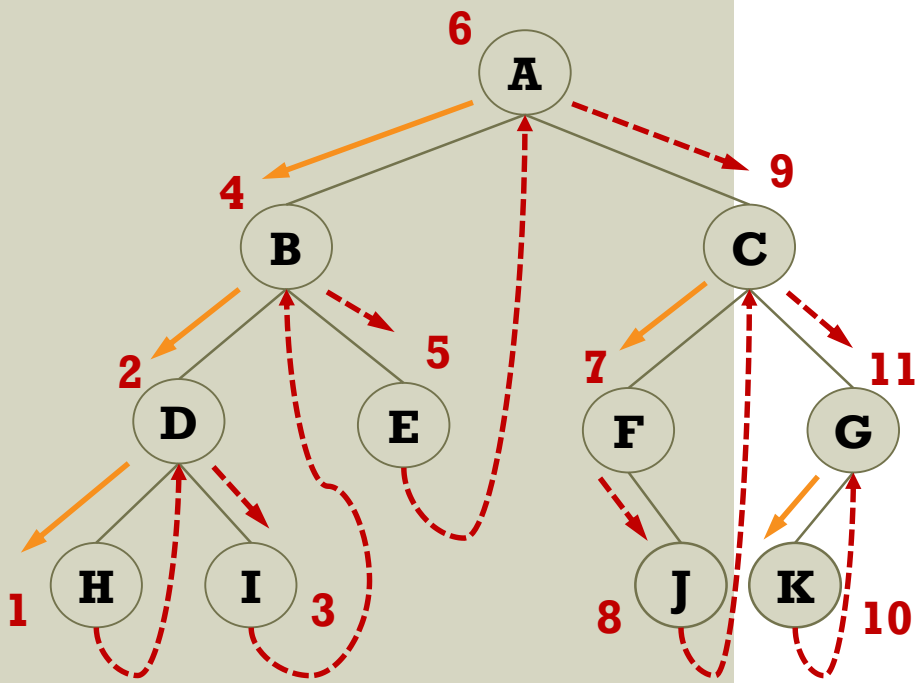
- Node representation in threaded binary tree
  - If no right child node exists, the right_child pointer of the node refers to its inorder successor.
    - This node indicates a threaded node.

```
typedef int BData;

typedef struct _bTreeNode
{
    BData item;
    struct _bTreeNode * left_child;
    struct _bTreeNode * right_child;
    bool isTheaded;
} BTreeNode;
```



Inorder in Threaded Binary Tree

```
BTreeNode* leftMost(BTreeNode* node)
{
    if (node == NULL) return NULL;
    while (node->left_child != NULL)
        node = node->left_child;

    return node;
}
```




```
void inorder(BTreeNode* node)
{
    BTreeNode* cur = leftmost(node);
    while (cur != NULL) {
        printf("%d ", cur->item);
        // If the node is a thread node, go to its inorder successor.
        if (cur->isTheaded)
            cur = cur->right_child;
        else // Go to the leftmost child in a right subtree.
            cur = leftmost(cur->right_child);
    }
}
```