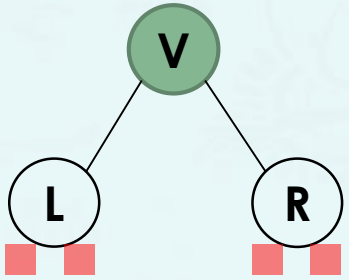
Data Structures
Chapter 5
Tree

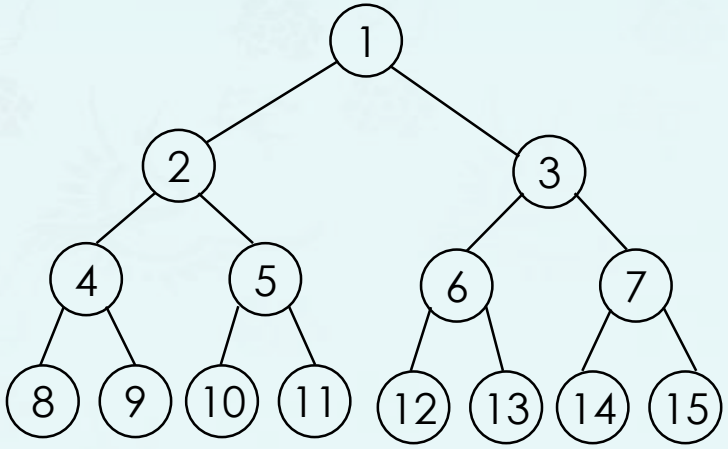
1. introduction
   - Definition and Terminology
2. Binary tree
   - Definition and Properties
   - **Traversal**
   - Coding
3. Binary search tree
4. Tree balancing

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

Binary tree traversals

- **Tree traversal** (known as **tree search**) refers to the process of visiting each node in a tree, exactly once, in a systematic way.
- DFS (Depth-first Search)
  - There are three possible moves if we traverse left before right:
    LVR – inorder
    LRV – postorder
    VLR – preorder
  - They are named according to the position of **V**(the visiting node) with respect to the L and R.
  - These searches are referred to as **depth-first search(DFS)** since the search tree is deepened as much as possible on each child before going to the next sibling.



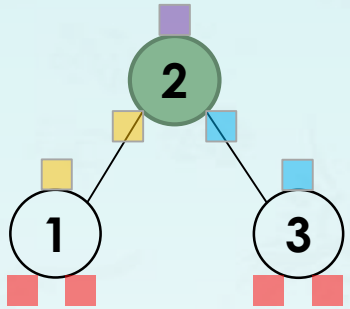


Binary tree traversals

- **Tree traversal** (known as **tree search**) refers to the process of visiting each node in a tree, exactly once, in a systematic way.
- DFS (Depth-first Search)
  - There are three possible moves if we traverse left before right:
    LVR – inorder
    LRV – postorder
    VLR – preorder
  - They are named according to the position of **V**(the visiting node) with respect to the L and R.
  - These searches are referred to as **depth-first search(DFS)** since the search tree is deepened as much as possible on each child before going to the next sibling.
- BFS (Breadth-first search)
  - The **level order traversal** traverses in level-order, where it visit every node one a level before going to a lower level.
  - This search is referred as breadth-first search(BFS), as the search tree broadened as much as possible on each depth before going to the next depth.

Binary tree traversals

- **Example: Inorder traversal(LVR)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Visit root node. (print or save it.)
  - Step 3 – Recursively traverse right subtree.

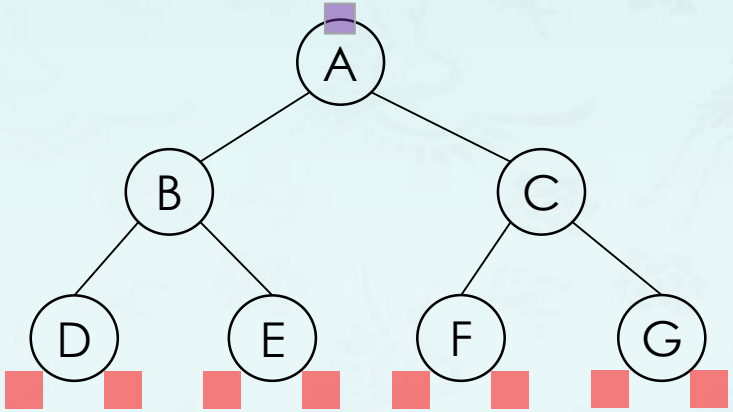



```
void inorder(tree root) {
    if (root == nullptr) return;

    inorder(root->left);      L
    cout << root->key;        V
    inorder(root->right);     R
}
```

Output(LVR) : 1 2 3

Binary tree traversals

- **Example: Inorder traversal(LVR)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Visit root node. (print or save it.)
  - Step 3 – Recursively traverse right subtree.

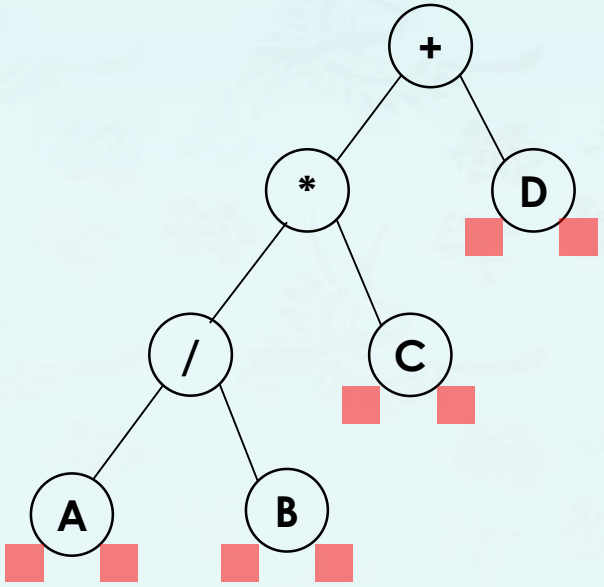



```
void inorder(tree root) {
   if (root == nullptr) return;

   inorder(root->left);      L
   cout << root->key;        V
   inorder(root->right);     R
}
```

Output(LVR) : D B E A F C G

Binary tree traversals

- **Q1: Inorder Traversal(LVR):**
- **Q2: How many times is** inorder( ) invoked for the complete traversal?
- **A2:** Every leaf node must visit (call the function) its left child and right child to make sure they don't have the child. 7 + 4 * 2 = 15

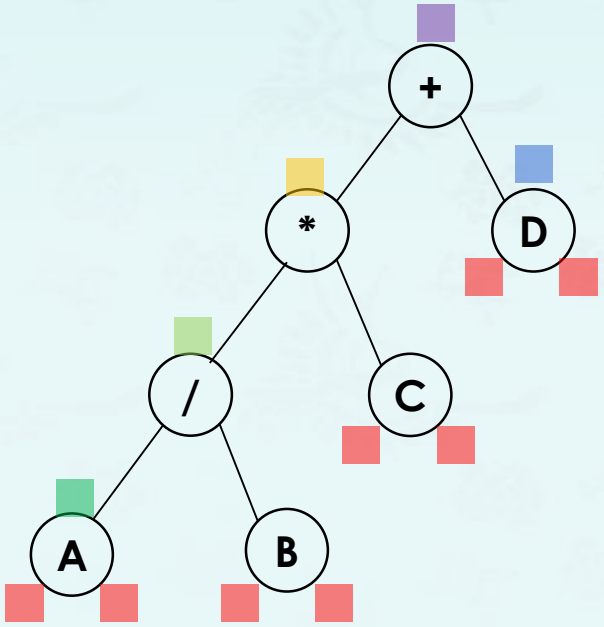



```
void inorder(tree root) {
   if (root == nullptr) return;

   inorder(root->left);          L
   cout << root->key;            V
   inorder(root->right);         R
}
```

Output(LVR) : A / B * C + D

Binary tree traversals

```
void inorder(tree root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```



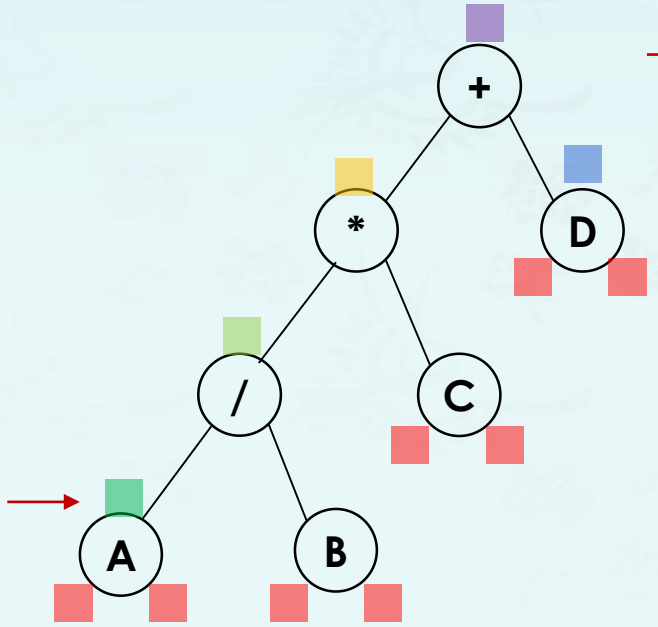
Call of inorder	**root** or **root→key**	Action
1	+	
2	*	
3	/	
4	A	
5	NULL	
4	A	**cout**
6	NULL	
3	/	**cout**
7	B	
8	NULL	
7	B	**cout**
9	NULL	
2	*	**cout**
10	C	

inorder	**root** or **root→key**	Value Action
11	NULL	
10	C	**cout**
12	NULL	
1	+	**cout**
13	D	
14	NULL	
13	D	**cout**
15	NULL	


system stack

Binary tree traversals

```
void inorder(tree root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```

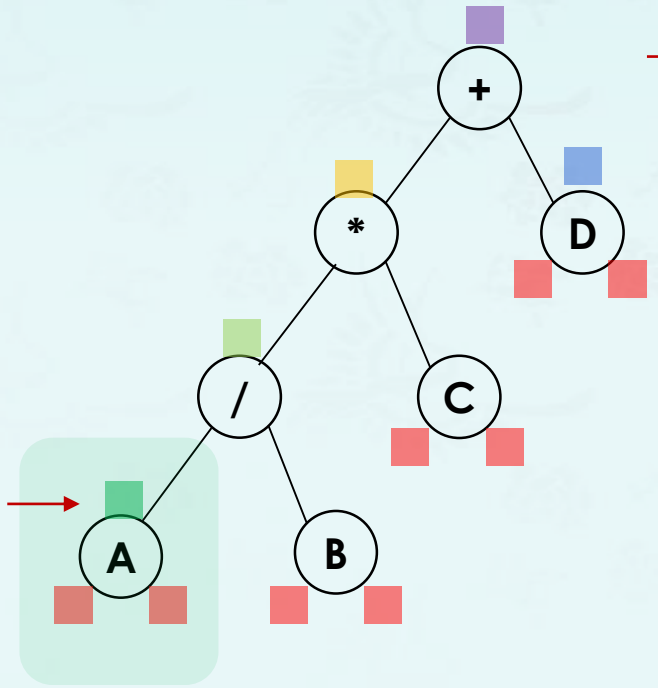


Call of inorder	**root** or **root→key**	Action
1	+	
2	*	
3	/	
4	A	
5	NULL	
4	A	**cout**
6	NULL	
3	/	**cout**
7	B	
8	NULL	
7	B	**cout**
9	NULL	
2	*	**cout**
10	C	

inorder	**root** or **root→key**	Value Action
11	NULL	
10	C	**cout**
12	NULL	
1	+	**cout**
13	D	
14	NULL	
13	D	**cout**
15	NULL	

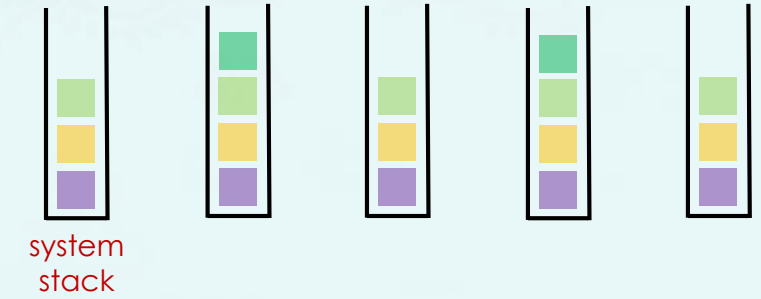
Binary tree traversals

```
void inorder(tree root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```



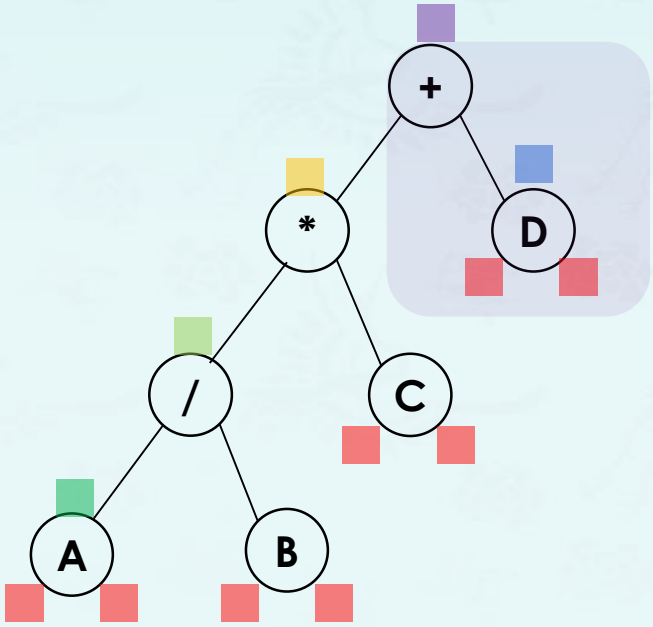
Call of inorder	**root** or **root→key**	Action
1	+	
2	*	
3	/	
4	A	push 4
5	NULL	pop 4 **cout**
4	A	push 4
6	NULL	pop 4, A done **cout**
3	/	
7	B	
8	NULL	
7	B	**cout**
9	NULL	
2	*	**cout**
10	C	

inorder	**root** or **root→key**	Value Action
11	NULL	
10	C	**cout**
12	NULL	
1	+	**cout**
13	D	
14	NULL	
13	D	**cout**
15	NULL	



Binary tree traversals

```
void inorder(tree root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```



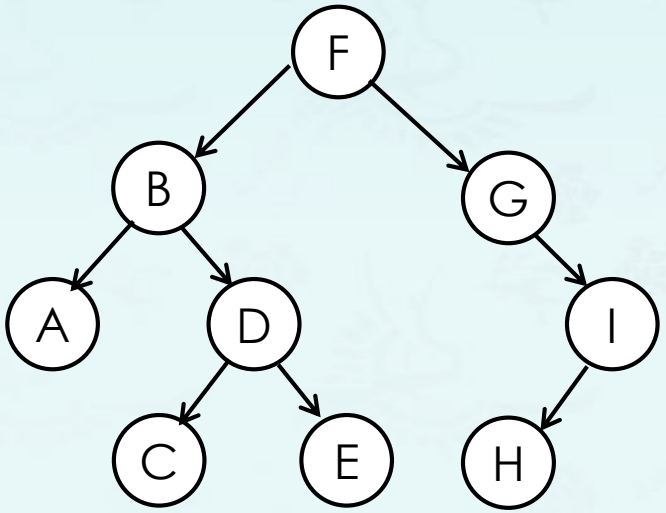
Call of inorder	**root** or **root→key**	Action
1	+	
2	*	
3	/	
4	A	
5	NULL	
4	A	**cout**
6	NULL	
3	/	**cout**
7	B	
8	NULL	
7	B	**cout**
9	NULL	
2	*	**cout**
10	C	

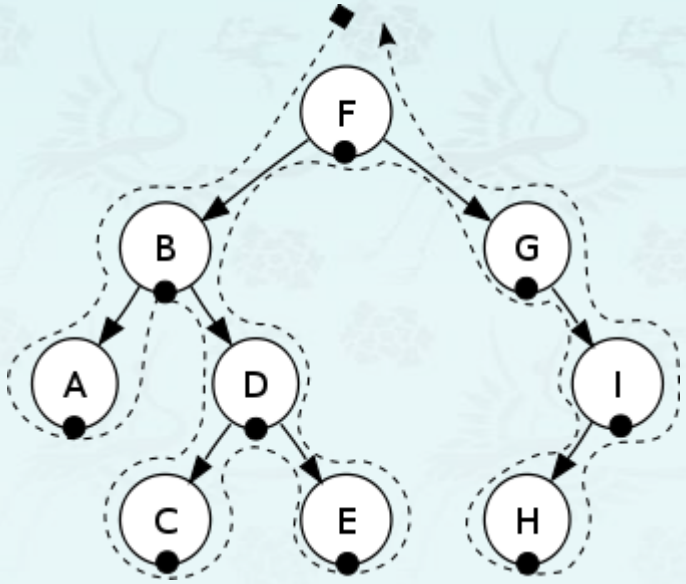
inorder	**root** or **root→key**		Value Action
11	NULL		
10	C		**cout**
12	NULL		
1	+	push 1	**cout**
13	D	push 13	
14	NULL	pop 13	
13	D		**cout**
15	NULL	push 13 pop 13	

+ done

Binary tree traversals

- **Q: Inorder traversal(LVR)**
  - Traverse the left subtree.
  - Visit the root.
  - Traverse the right subtree.



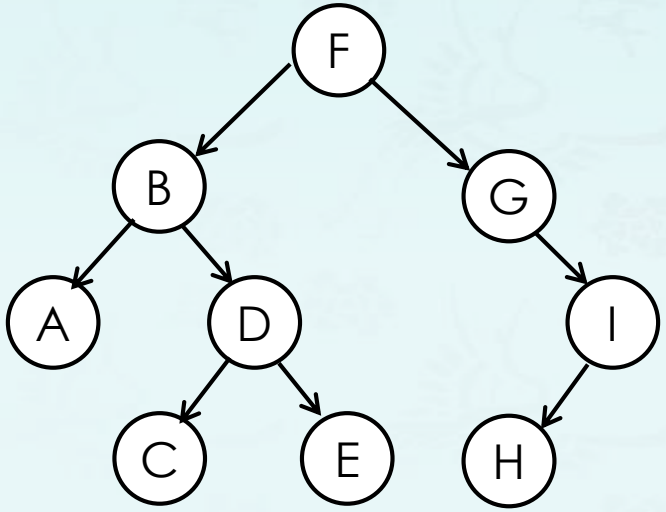


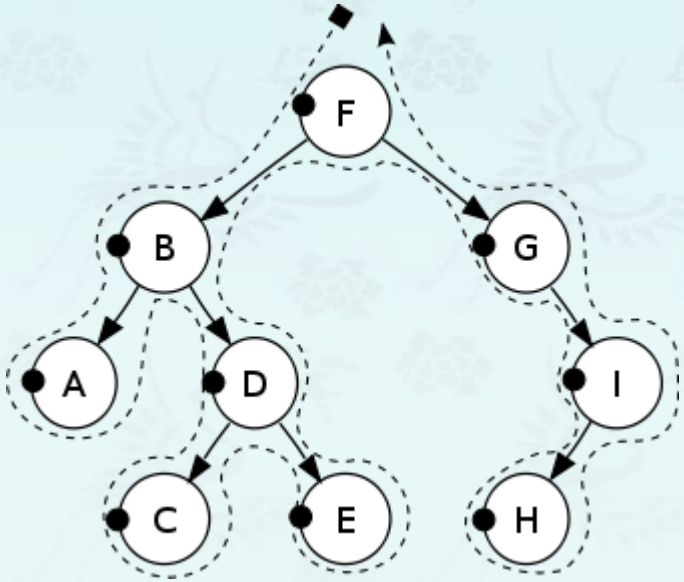

```
void inorder(tree root) {
    if (root == nullptr) return;

    inorder(root->left);
    cout << root->key;
    inorder(root->right);
}
```

Output: A, B, C, D, E, F, G, H, I

Binary tree traversals

- **Q: Preorder traversal(VLR)**
  - Step 1 – Visit root node.
  - Step 2 – Recursively traverse left subtree.
  - Step 3 – Recursively traverse right subtree.



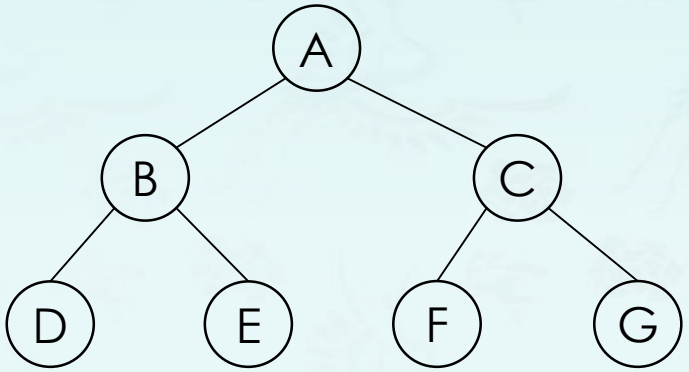



```
void preorder(tree root) {
   if (root == nullptr) return;

   cout << root->key;          V
   preorder(root->left);       L
   preorder(root->right);      R
}
```

Output: (VLR) : F, B, A, D, C, E, G, I, H

Binary tree traversals

- **Q: Preorder traversal(VLR)**
  - Step 1 – Visit root node.
  - Step 2 – Recursively traverse left subtree.
  - Step 3 – Recursively traverse right subtree.

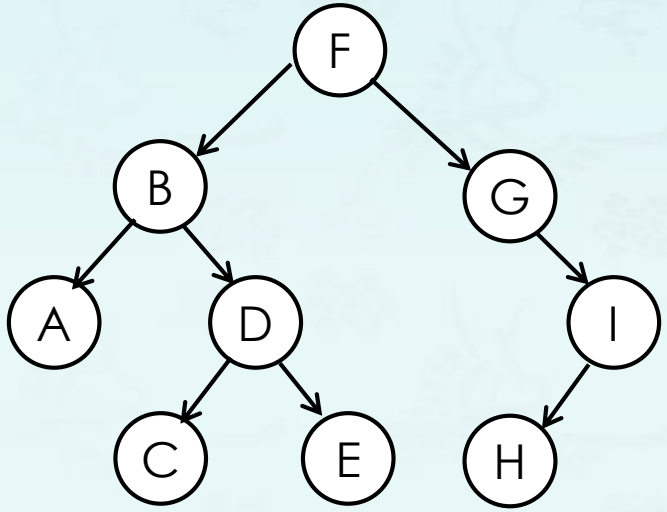


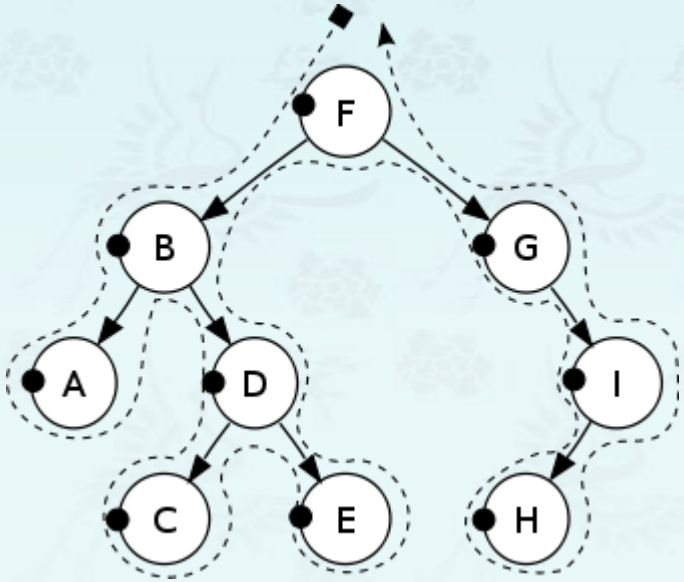

```
void preorder(tree root) {
   if (root == nullptr) return;

   cout << root->key;          V
   preorder(root->left);       L
   preorder(root->right);      R
}
```

Output: (VLR) : A B D E C F G

Binary tree traversals

- **Q: Postorder traversal(LRV)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Recursively traverse right subtree.
  - Step 3 – Visit root node.



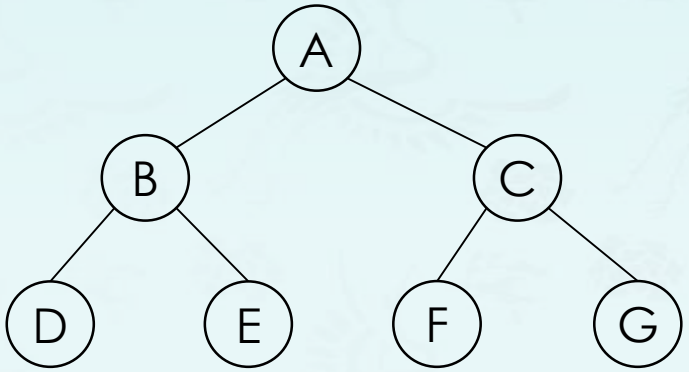



```
void postorder(tree root) {
    if (root == nullptr) return;

    postorder(root->left);          L
    postorder(root->right);         R
    cout << root->key;              V
}
```

Output(LRV): A C E D B H I G F

Binary tree traversals

- **Q: Postorder traversal(LRV)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Recursively traverse right subtree.
  - Step 3 – Visit root node.

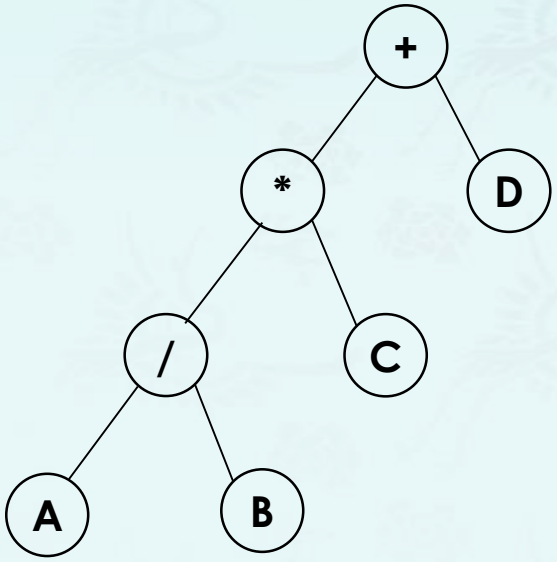



```
void postorder(tree root) {
   if (root == nullptr) return;

   postorder(root->left);
   postorder(root->right);
   cout << root->key;
}
```

Output(LRV): D E B F G C A

Binary tree traversals

- **Q: Postorder traversal(LRV)**
  - Step 1 – Recursively traverse left subtree.
  - Step 2 – Recursively traverse right subtree.
  - Step 3 – Visit root node.

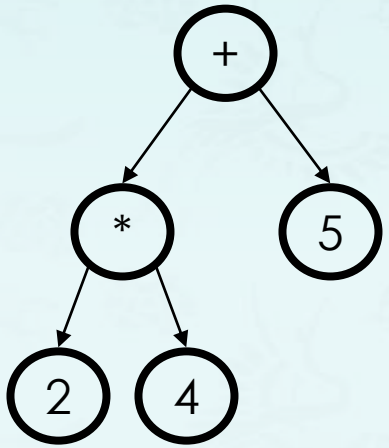



```
void postorder(tree root) {
   if (root == nullptr) return;

   postorder(root->left);
   postorder(root->right);
   cout << root->key;
}
```

Output(LRV): A B / C * D +

Binary tree traversals

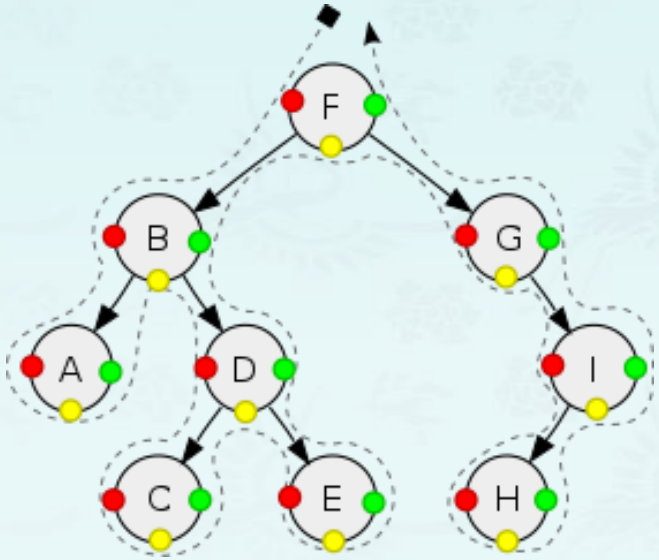
- **Exercise 1:**
  - preorder Traversal(VLR):
  - inorder traversal(LVR) :
  - postorder traversal(LRV):
  - Level Order traversal :

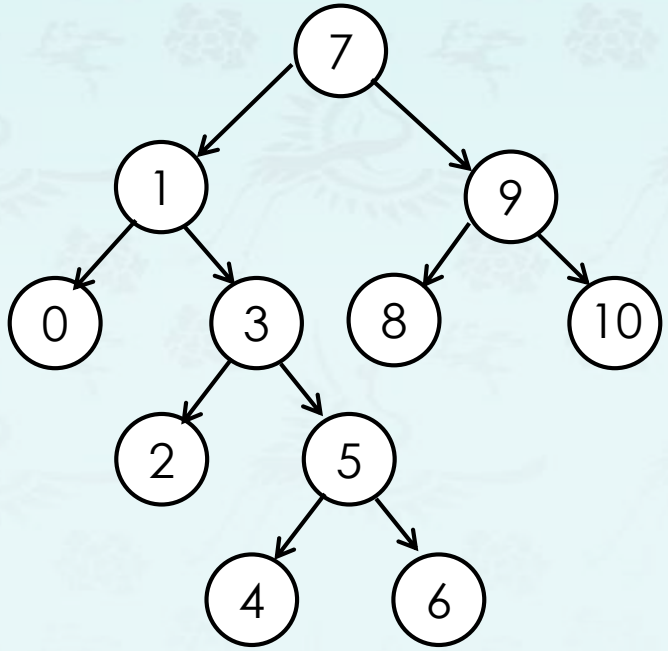


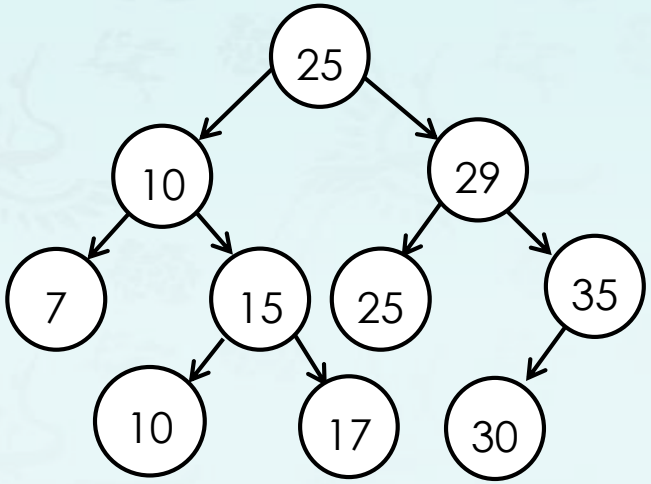
Binary tree traversals

- **Exercise 2:**
  - preorder(VLR)
  - inorder(LVR)
  - postorder(LRV)

F	7	25
A	0	7
A	0	7







Binary tree traversals

- **Observations:**
  1. If you know you need **to explore the roots** before inspecting their leaves, you pick **preorder** because you will encounter the roots (or parents) before their leaves.

  2. If you know you need to **explore the leaves** before their parent nodes, you select **postorder** because you don't waste any time inspecting roots in search for leaves.

  3. If you know that the tree has an inherent sequence in the nodes, and you want to flatten the tree back into its original sequence, than an **inorder** traversal should be used. The tree would be flattened in the same way it was created. A pre-order or post-order traversal might not unwind the tree back into the sequence which was used to create it.

  4. In a binary search tree ordered such that in each node the key is greater than all keys in its left subtree and less than all keys in its right subtree, in-order traversal retrieves the keys in *ascending* sorted order.

Summary &
quaestio quaestio go ? ? ? ?

Data Structures
Chapter 5
Tree

1. introduction
2. **Binary tree**
   - **Definition and Properties**
   - **Traversal**
   - Coding
3. Binary search tree
4. Tree balancing