Unit-II

Trees

Unit II	Trees	(08 Hours)

Tree- basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree. Huffman Tree (Concept and Use), Binary Search Tree (BST), BST operations, Threaded binary search tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary search tree, in order traversal of in-order threaded binary search tree.

TREES

- Trees are one of the important non- Linear data structure.
- A tree is a Multilevel data structure that represent a hierarchical relationship between the Set of individual elements called nodes.
- Each tree structure starts with a node Which is called the root node of the Tree.

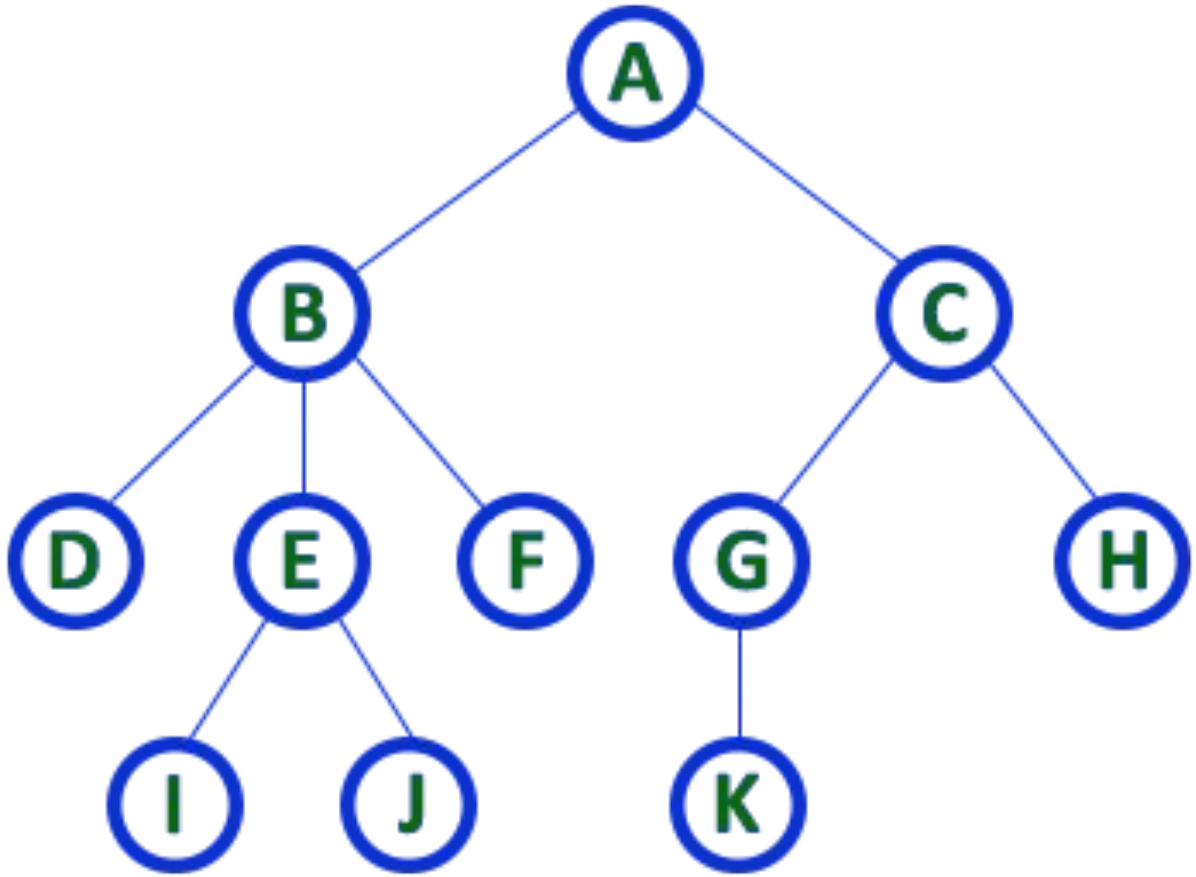
Basic terminology of Trees

1) A tree consists of a finite set of elements, called "nodes", and a finite set of **directed** lines, called "branches", that connect the nodes.

2) The number of branches associated with a node is called the "degree of the node".

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

Example: TREE

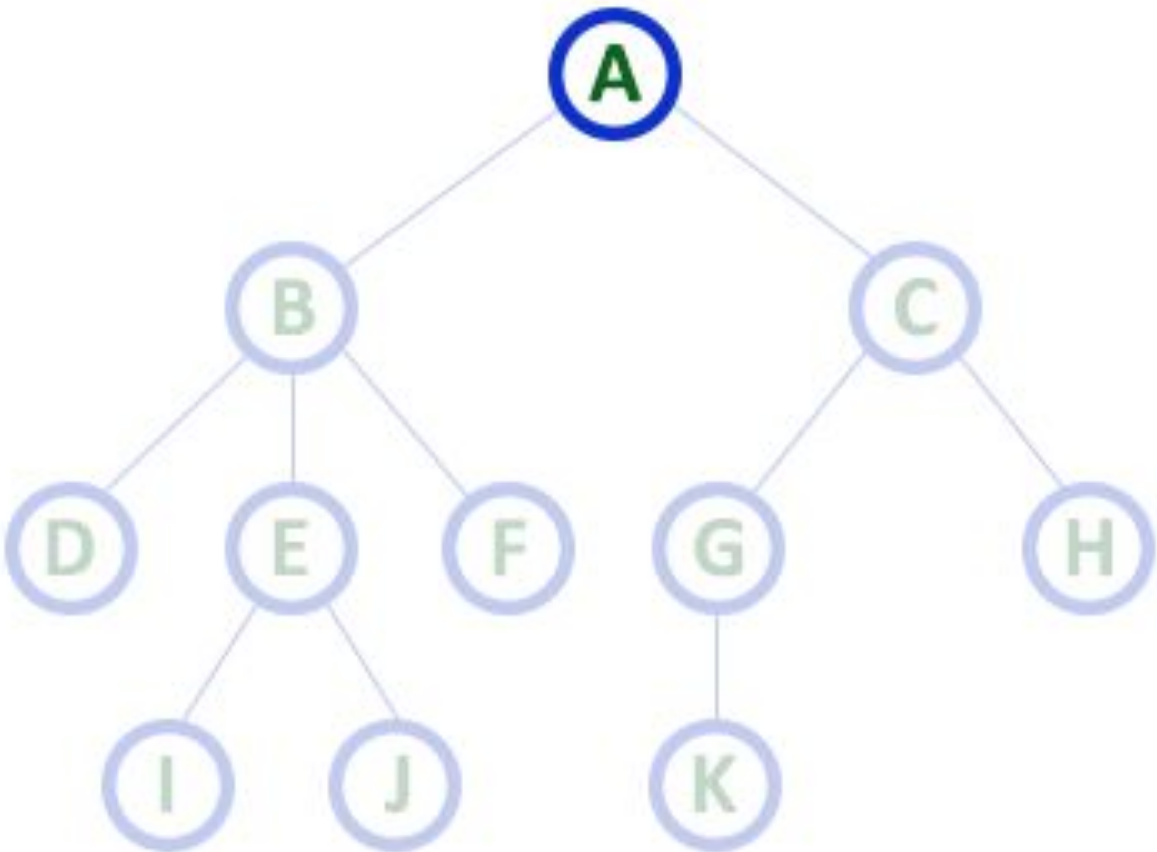


TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

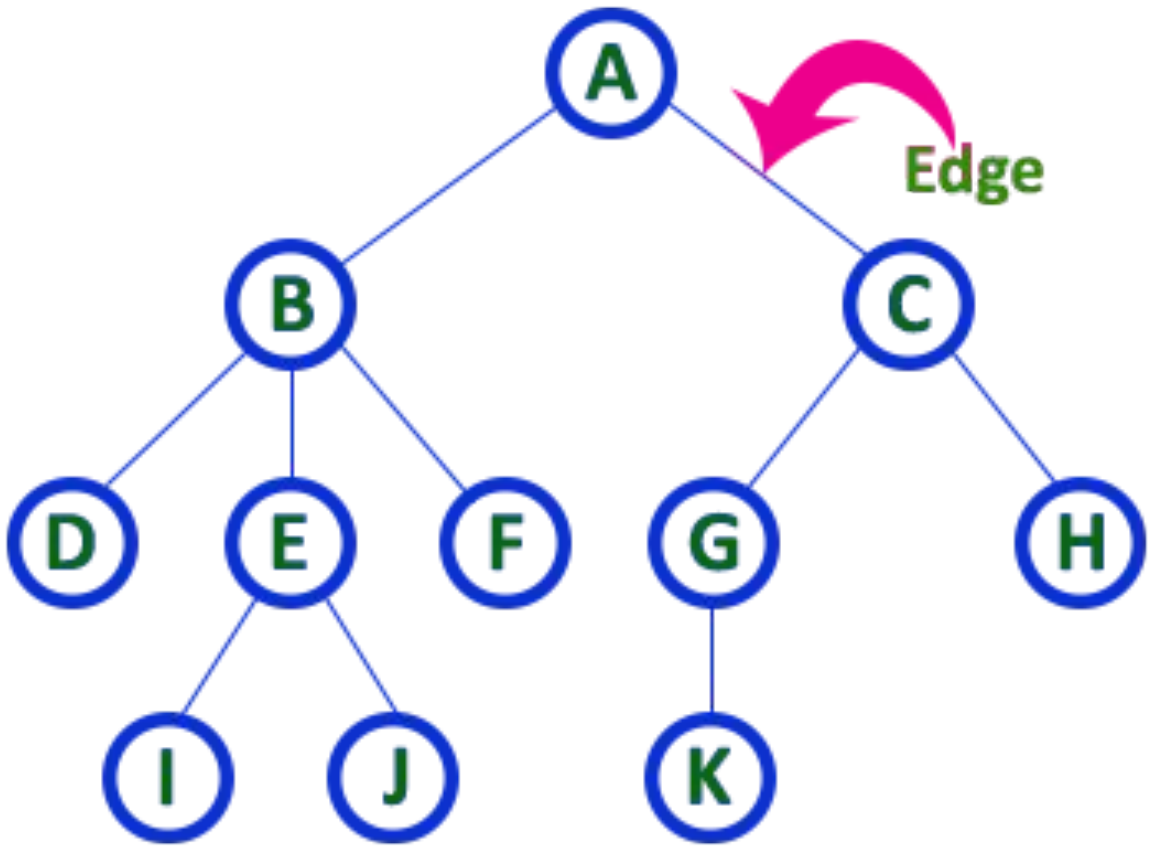
1. Root



Here ‘A’ is the ‘root’ node

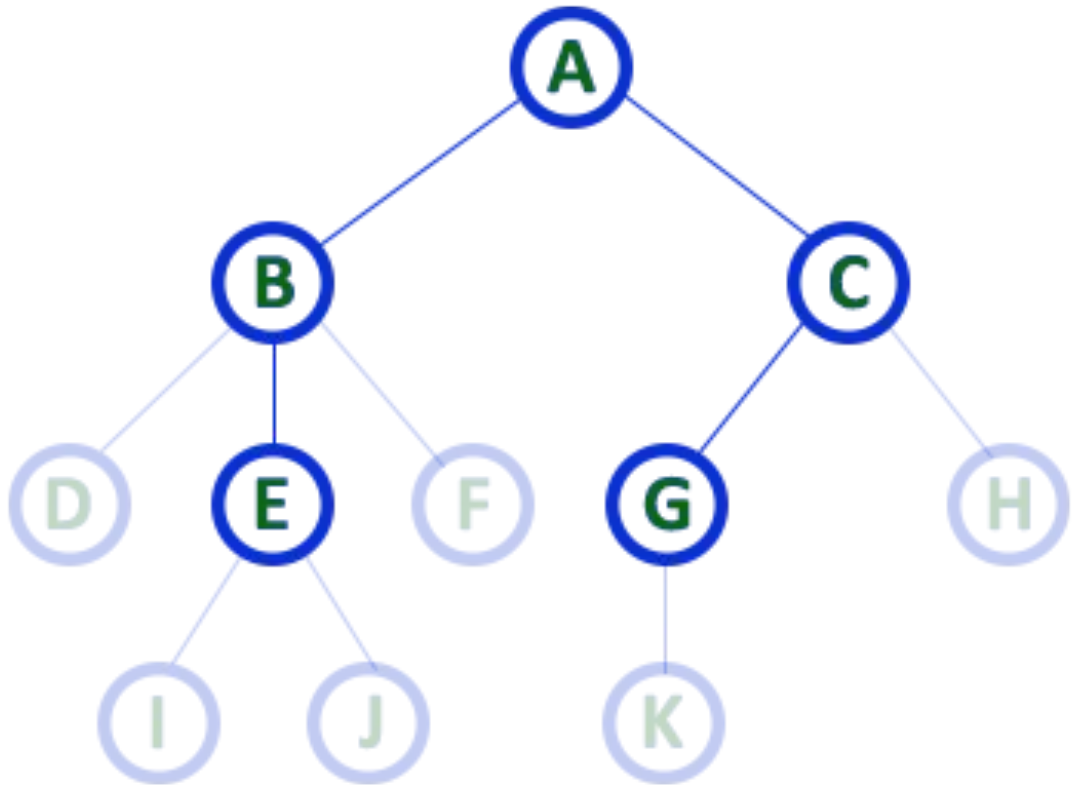
- In any tree the first node is called as ROOT node

2. Edge



- In any tree, 'Edge' is a connecting link between two nodes.

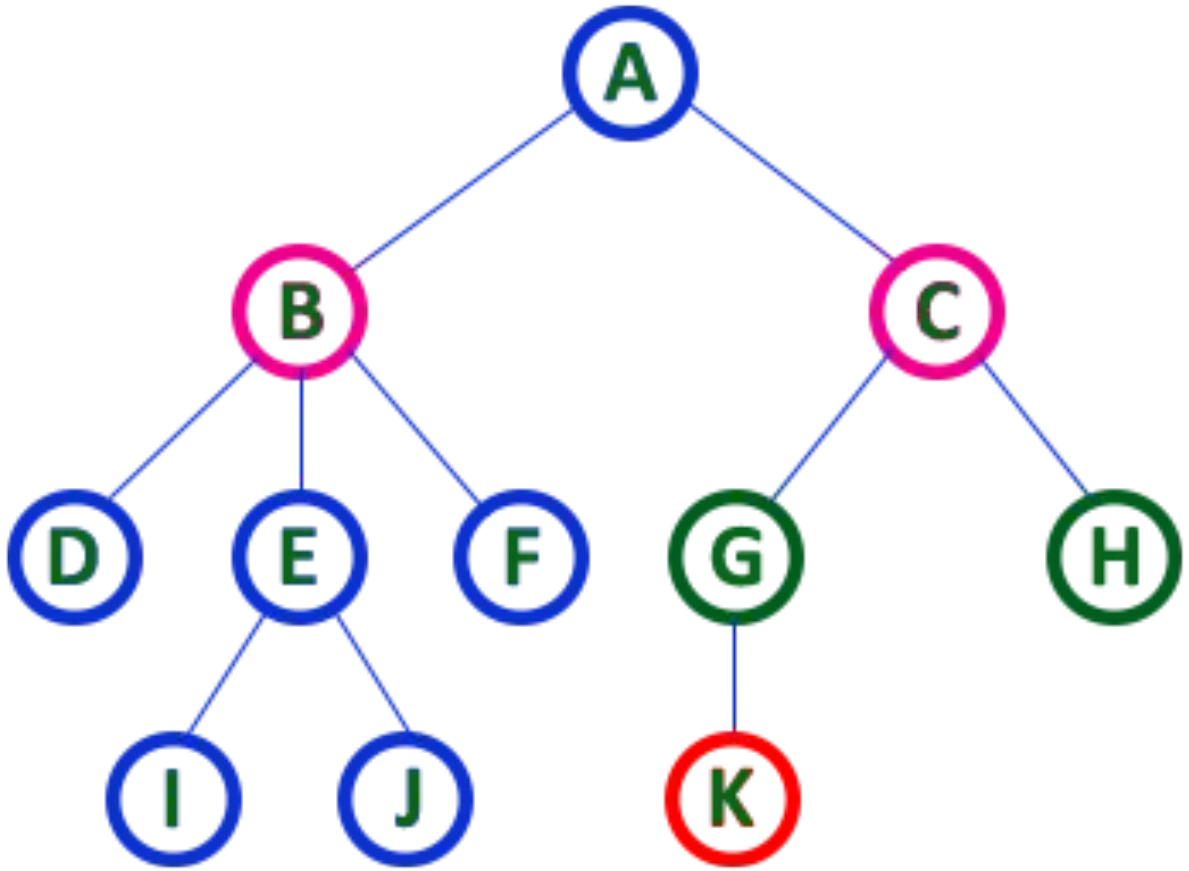
3. Parent



Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child



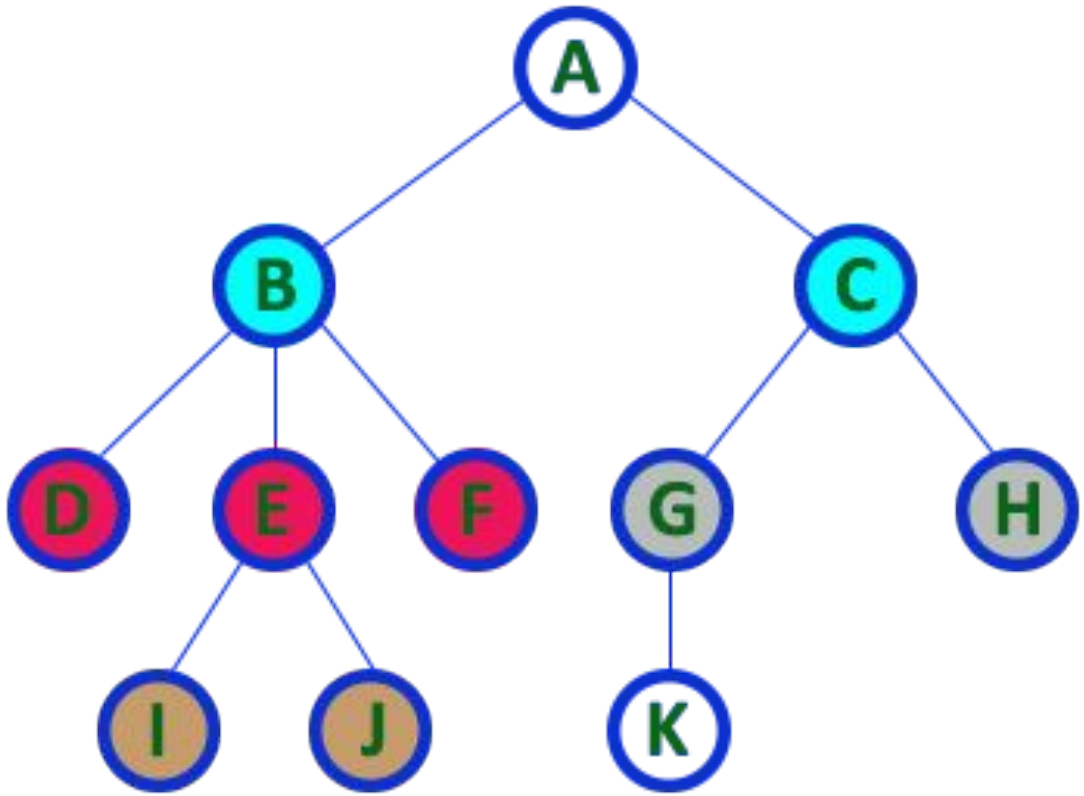
Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

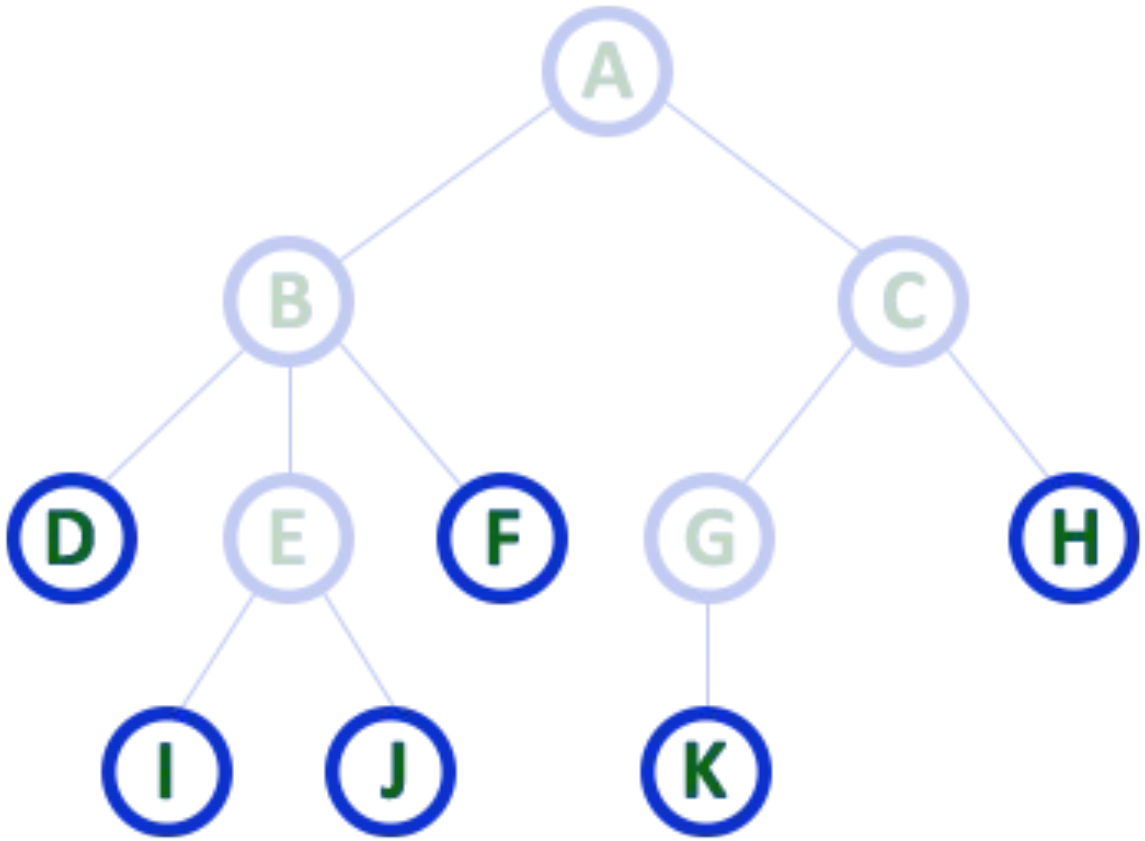
5. Siblings



Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

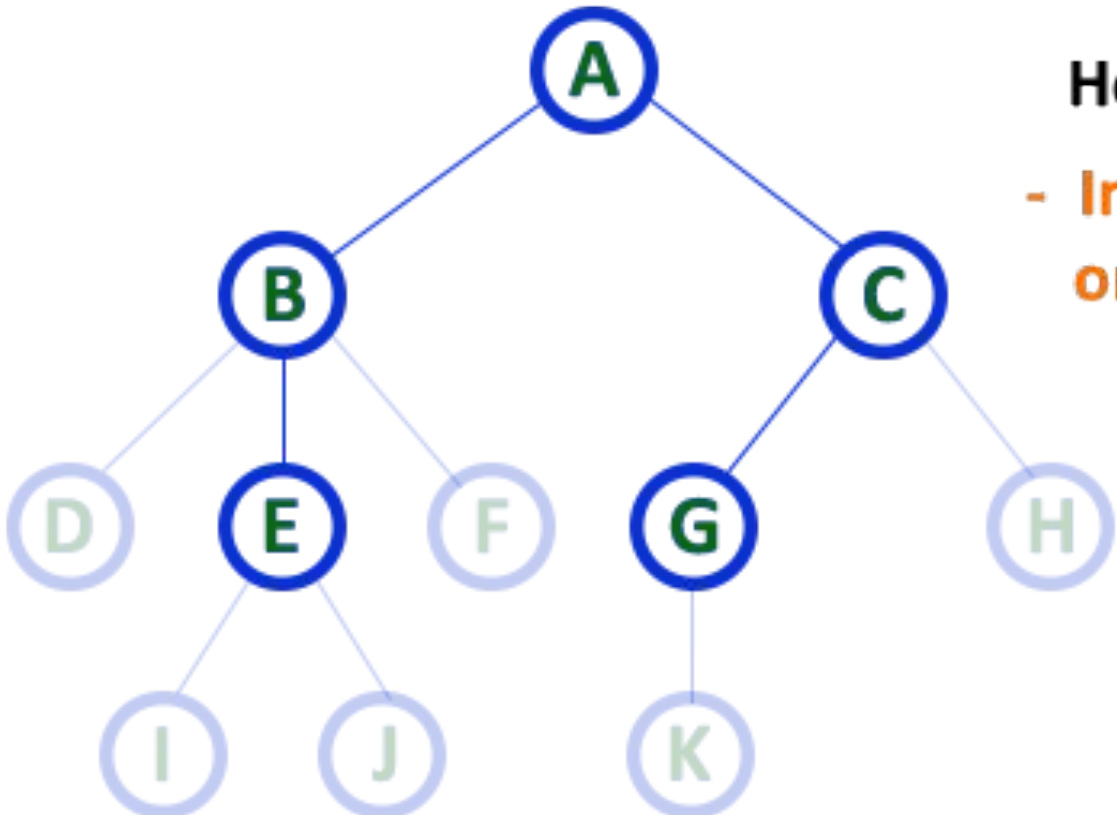
6. Leaf



Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

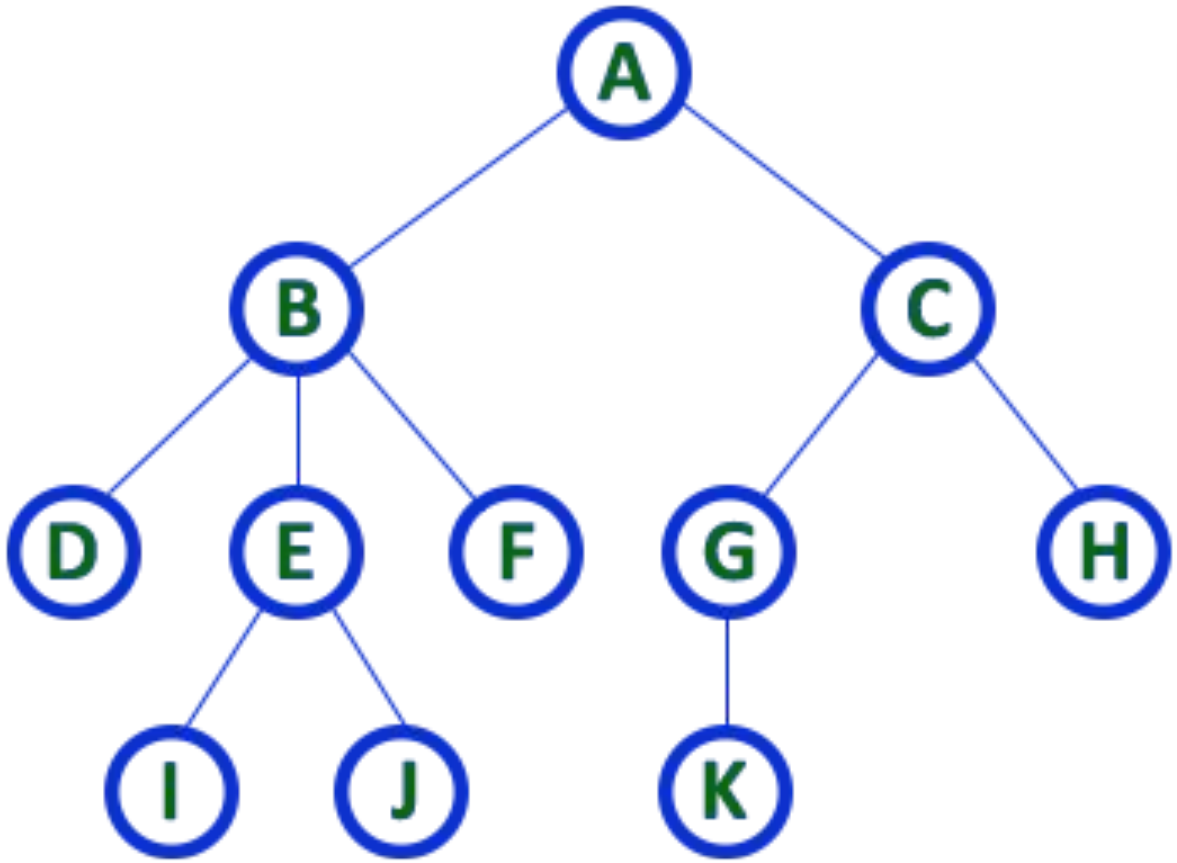


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node

- Every non-leaf node is called as 'Internal' node

8. Degree



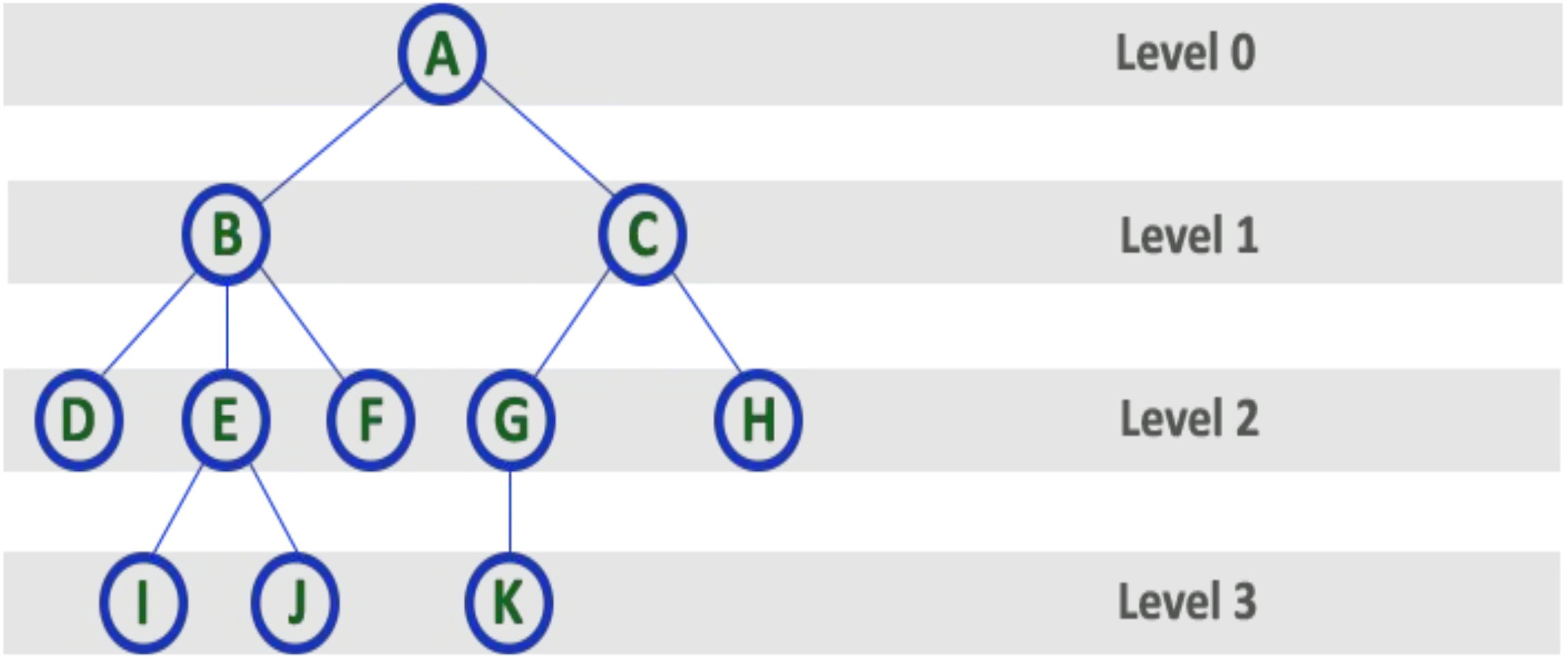
Here Degree of B is 3

Here Degree of A is 2

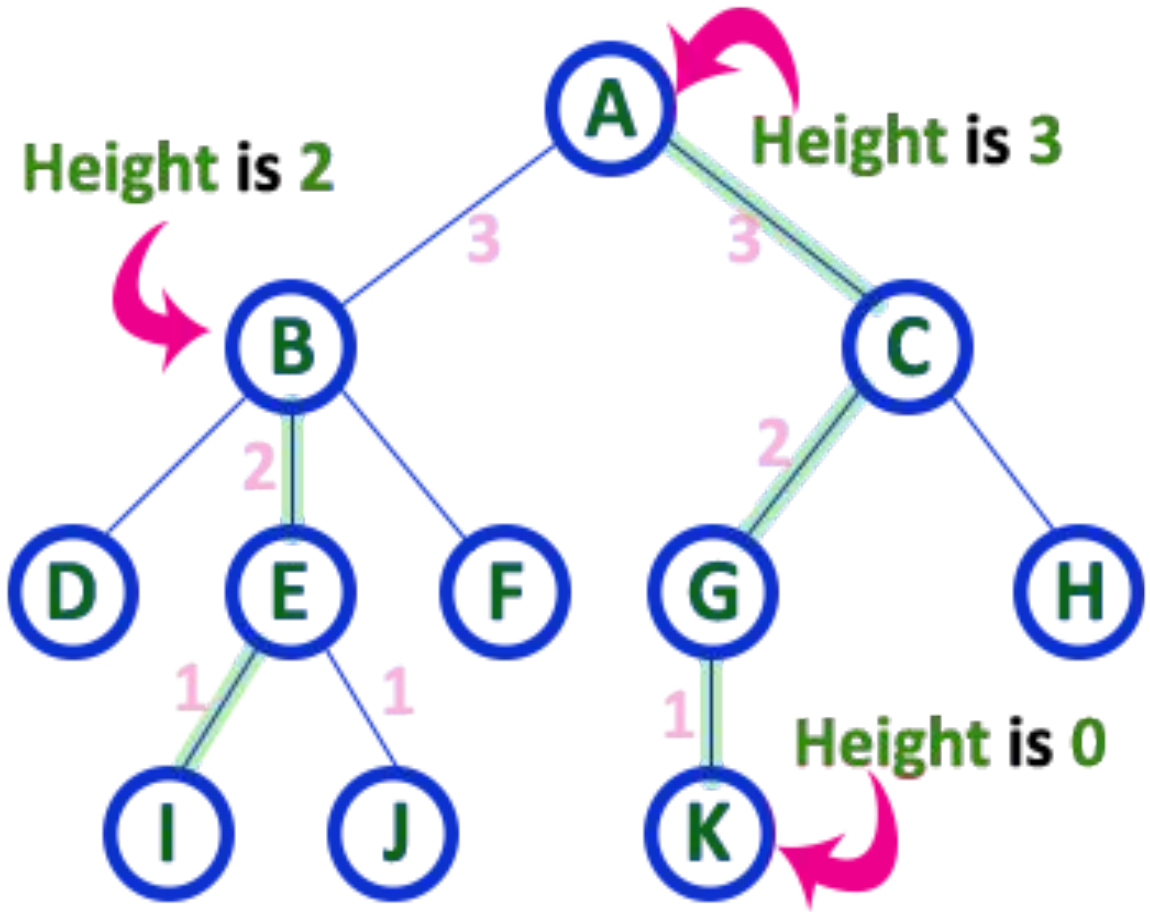
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

9. Level



10. Height

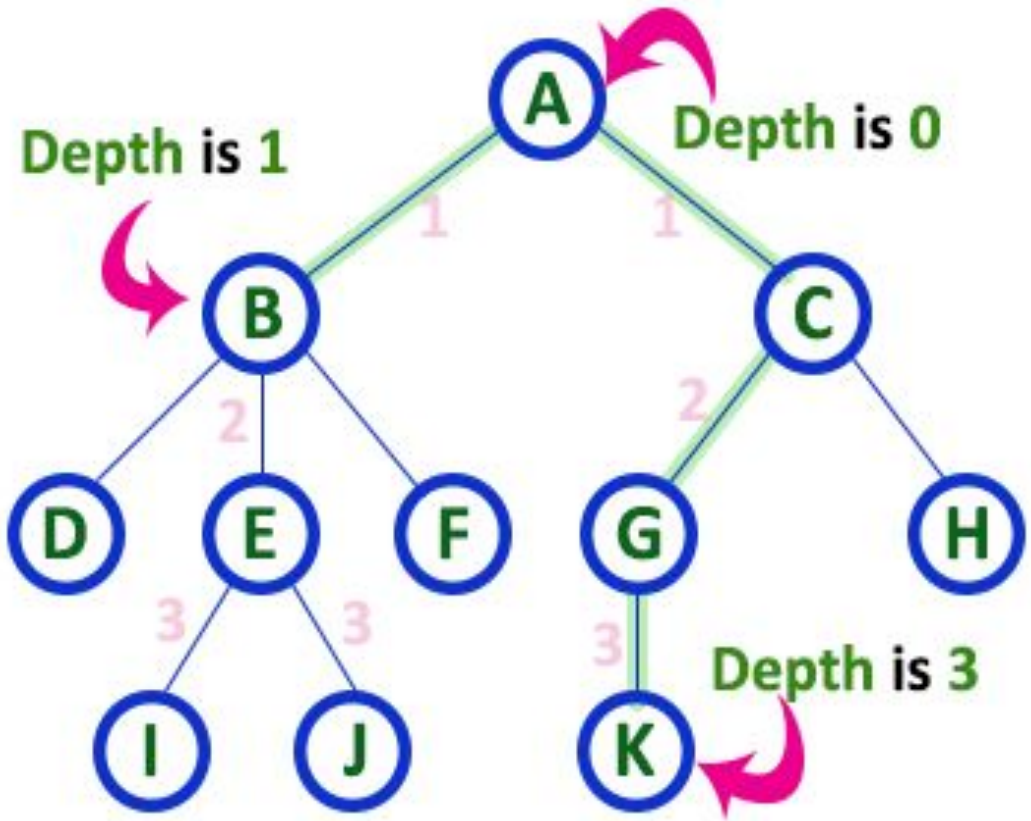


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.

- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

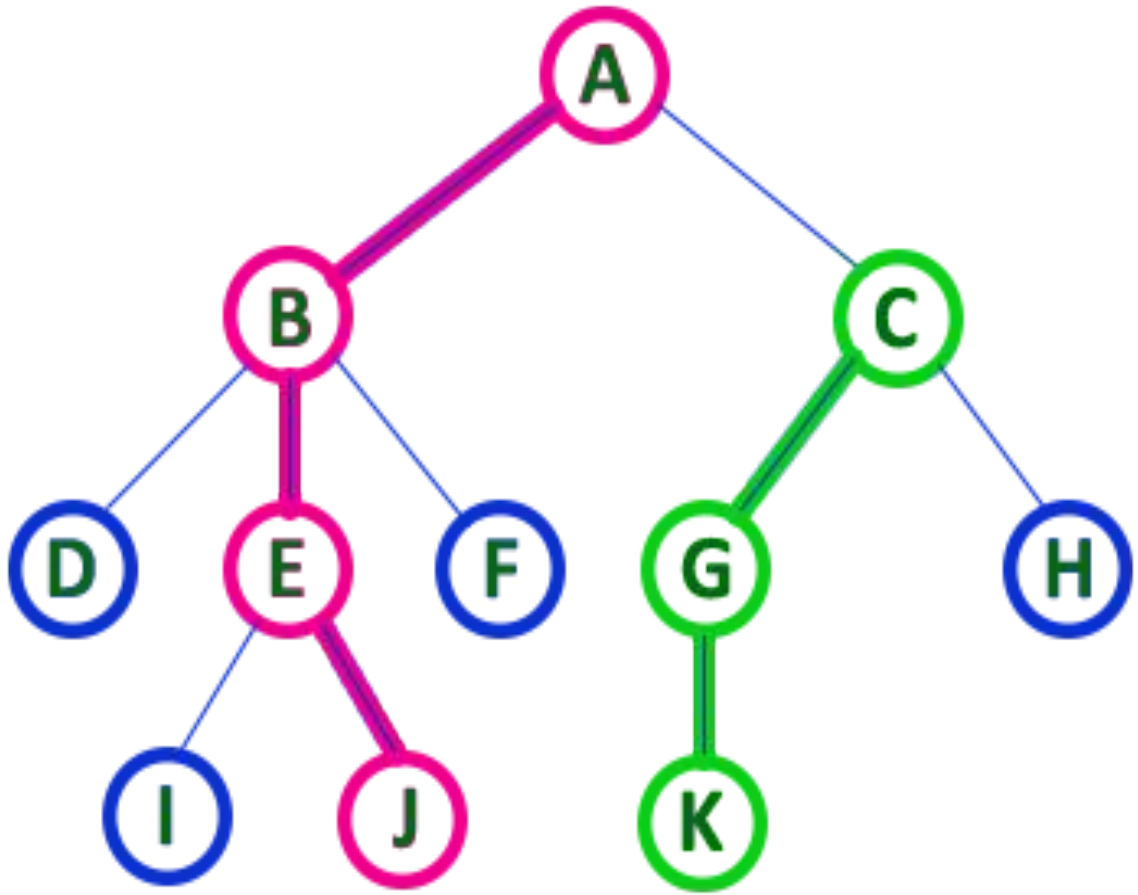


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.

- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

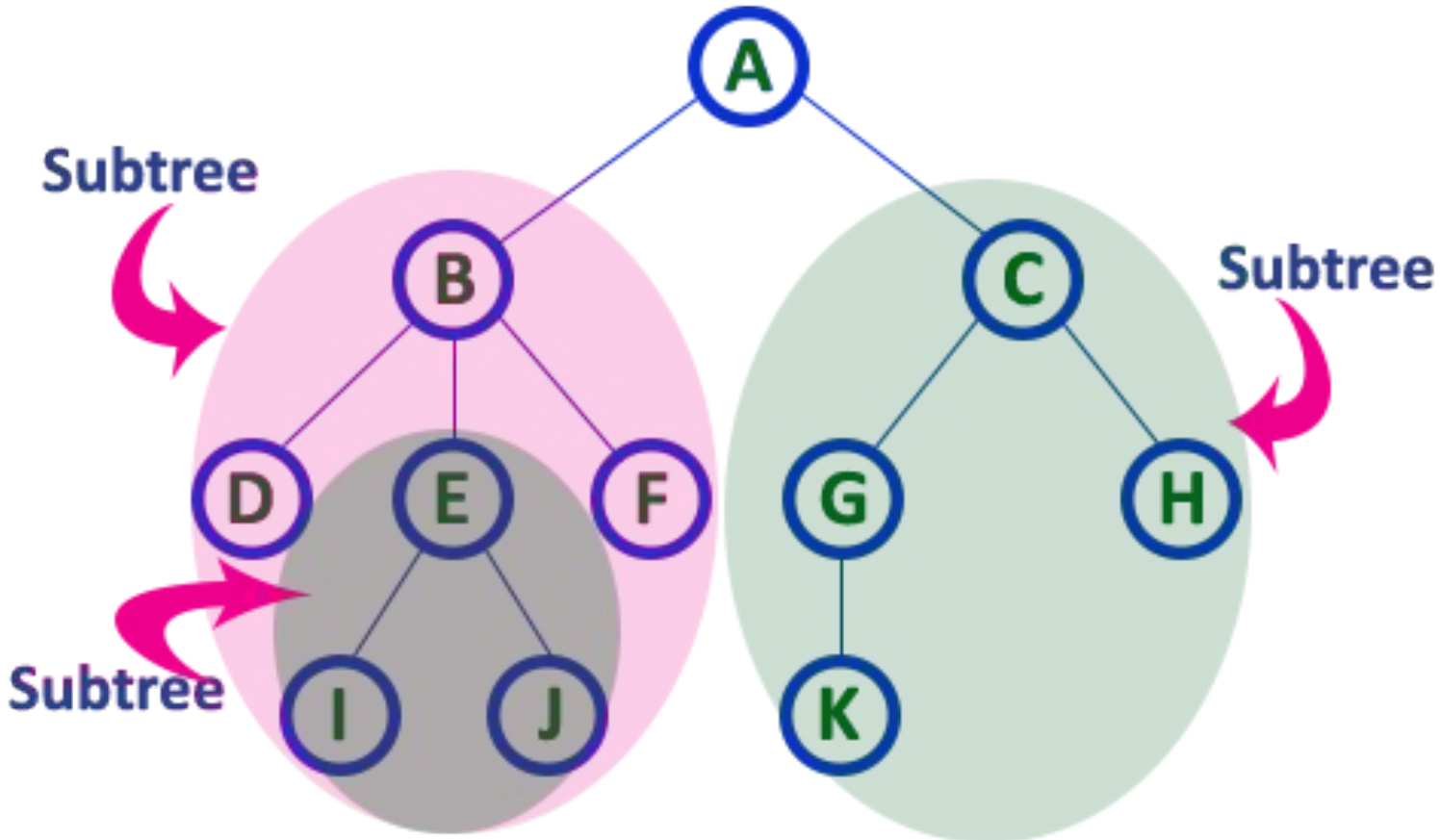
Here, 'Path' between A & J is

A - B - E - J

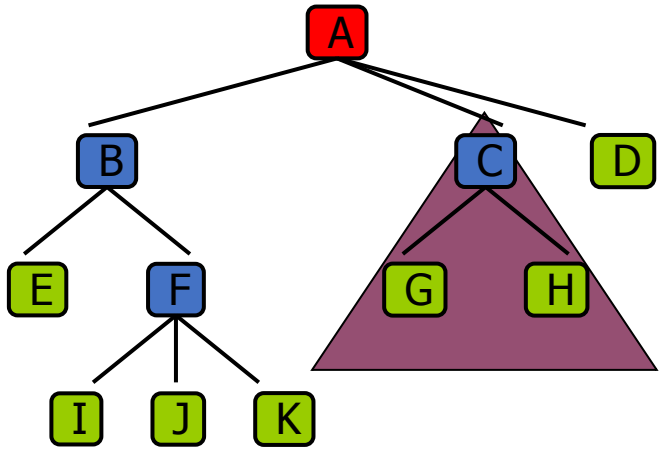
Here, 'Path' between C & K is

C - G - K

13. Sub Tree

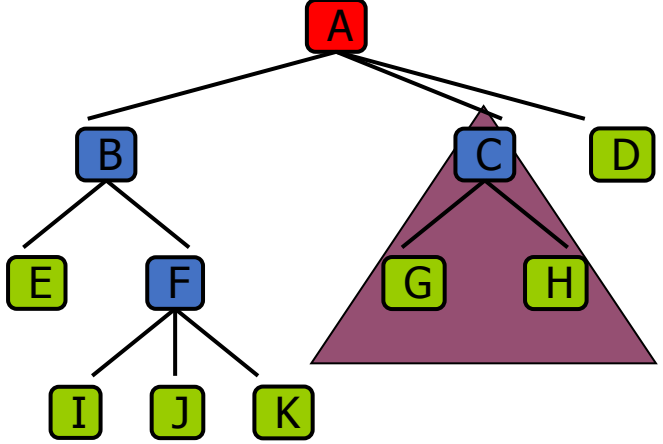


Tree terminology



- **Root**: Node **without parent** (A).The **top node** in a tree.
- **Child :**A node directly connected to another node when moving away from the root.
- **Parent :**The converse notion of a child.
- **Siblings**: **Nodes share the same parent**
- **Internal node(Branch node)**: node with at least one child (A, B, C, F)
- **External node** (leaf node): node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc. A node reachable by repeated proceeding from child to parent.
- **Descendant of a node**: child, grandchild, grand-grandchild, etc A node reachable by repeated proceeding from parent to child. Also known as sub child.

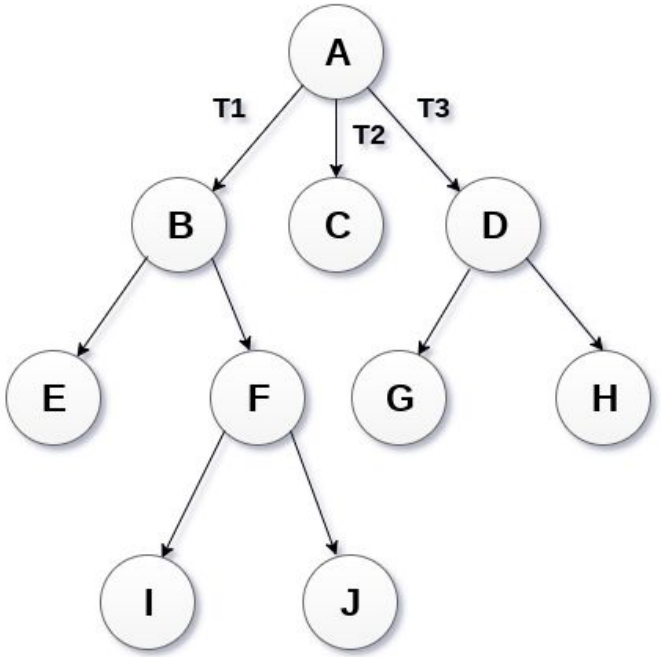
Tree terminology



- **Depth of a node**: number of ancestors. The depth of a node is the number of edges from the tree's root node to the node.
- **Height of node:** The height of a node is the number of edges on the longest path between that node and a leaf.
- **Height of a tree**: maximum depth of any node (3).The height of a tree is the height of its root node
- **Degree of a node**: the number of its children. A leaf is necessarily degree zero.
- **Degree of a tree**: the maximum number of its node.
- **Subtree**: tree consisting of a node and its descendants.
- **Edge:** The connection between one node and another.
- **Path:** A sequence of nodes and edges connecting a node with a descendant.
- **Forest:** A forest is a set of $n \geq 0$ disjoint trees.

General tree

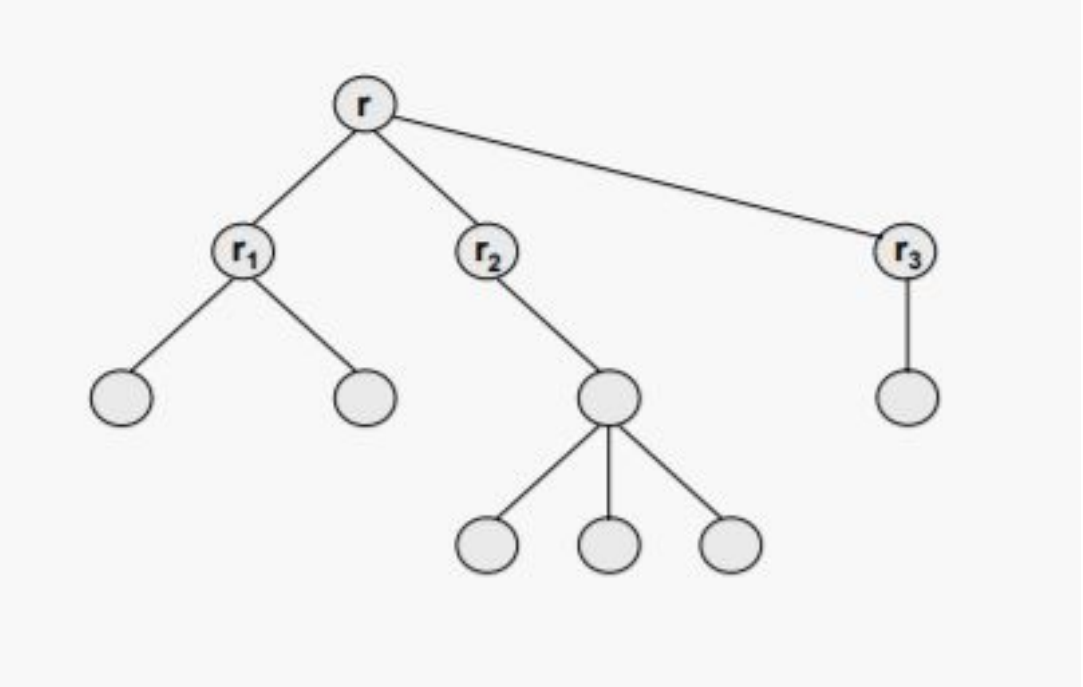
- A node may contain any number of sub-trees.



General Tree

General tree and its representation

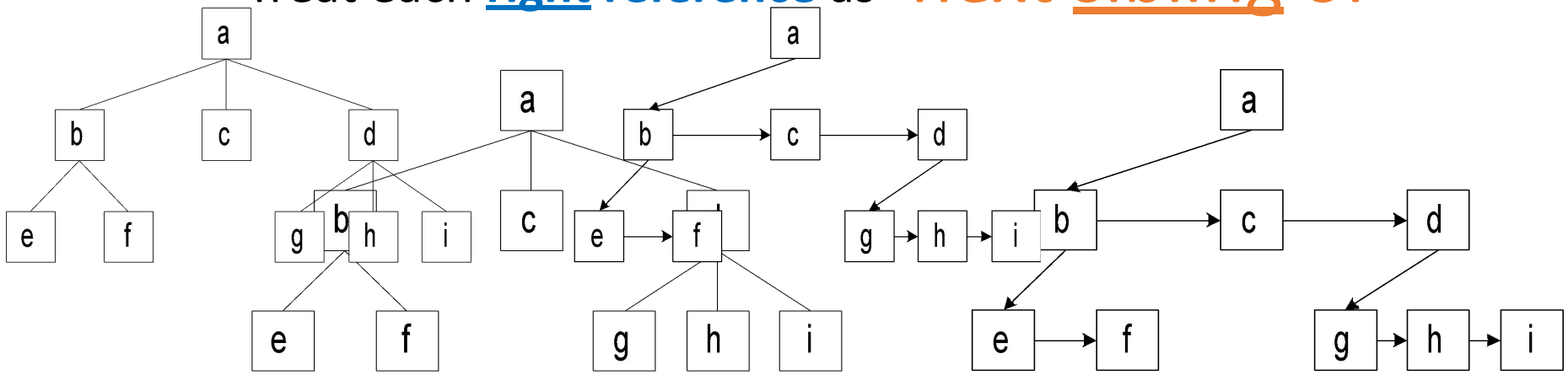
- General Trees A general tree T is a finite set of one or more nodes such that there is one designated node r, called the root of T, and the remaining nodes are partitioned into n ≥ 0 disjoint subsets T1, T2, …, Tn, each of which is a tree, and whose roots r1, r2, …, rn, respectively, are children of r



General Trees Binary Tree

Any tree can be transformed into binary tree by **left child-right sibling** representation

- How to store general tree in binary tree
  - Treat each **left reference** as "first child of"
  - Treat each **right reference** as "next sibling of"

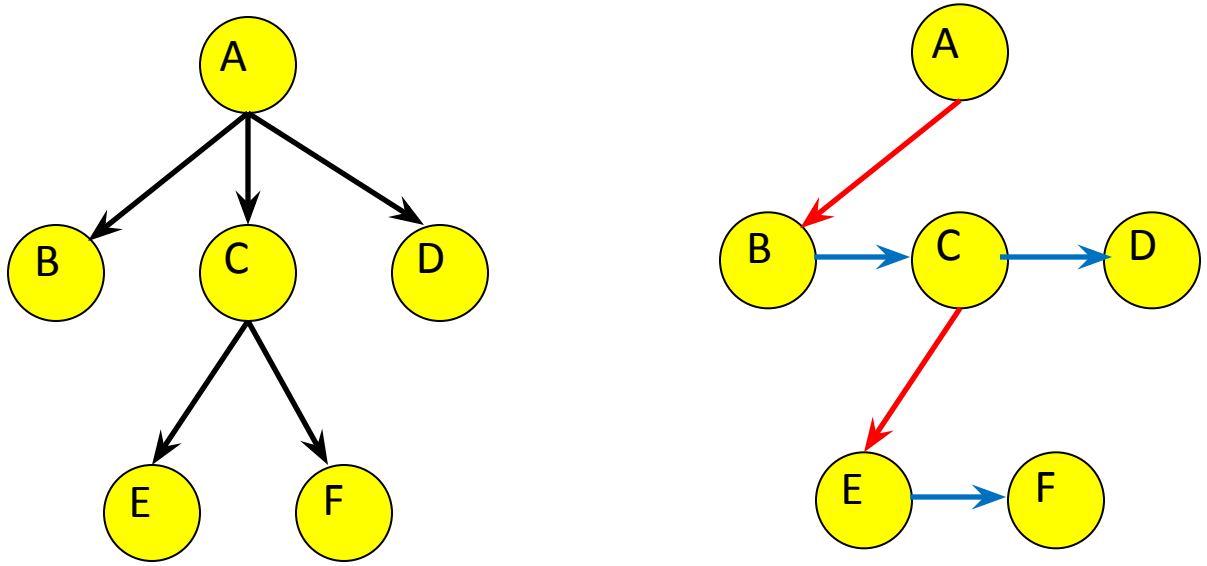


Original general tree

Convert to binary tree

1st Child/Next Sibling Representation

- Each node has 2 pointers: one to its first child and one to next sibling



Introduction

- Representation Of Trees
  - List Representation
    - we can write of Figure 5.2 as a list in which each of the subtrees is also a list

      (A (B (E (K, L), F), C (G), D (H (M), I, J)))
    - The root comes first,
      followed by a list of sub-trees

Figure 5.3: Possible list representation for trees

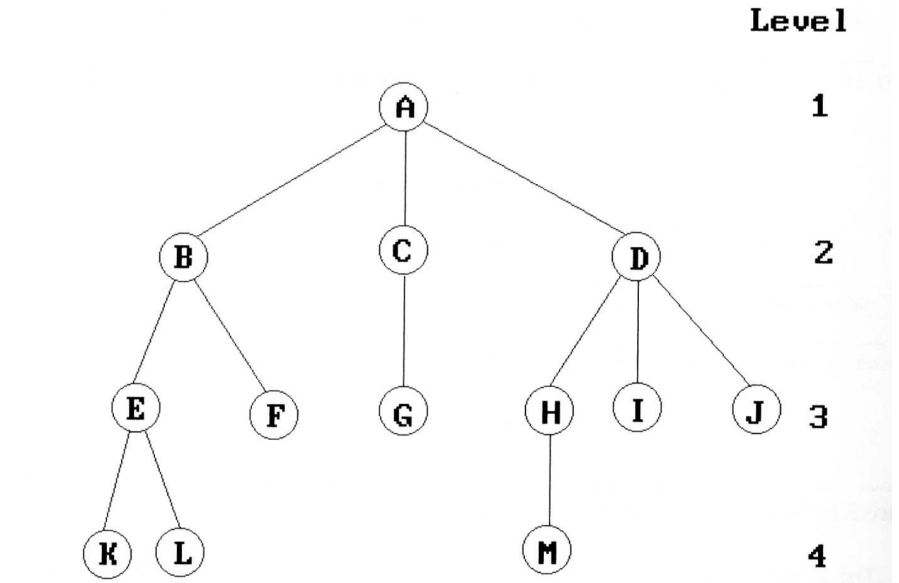


Figure 5.2: A sample tree

Introduction

- Representation Of Trees (cont'd)
  - Left Child-Right Sibling Representation

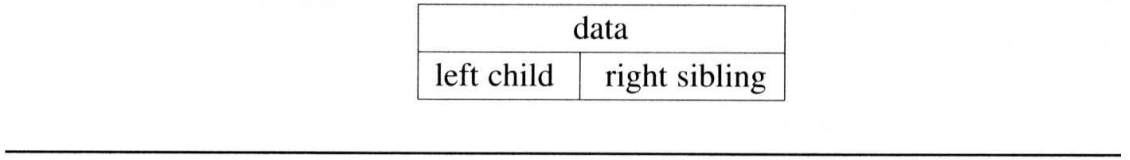


Figure 5.4: Left child-right sibling node structure

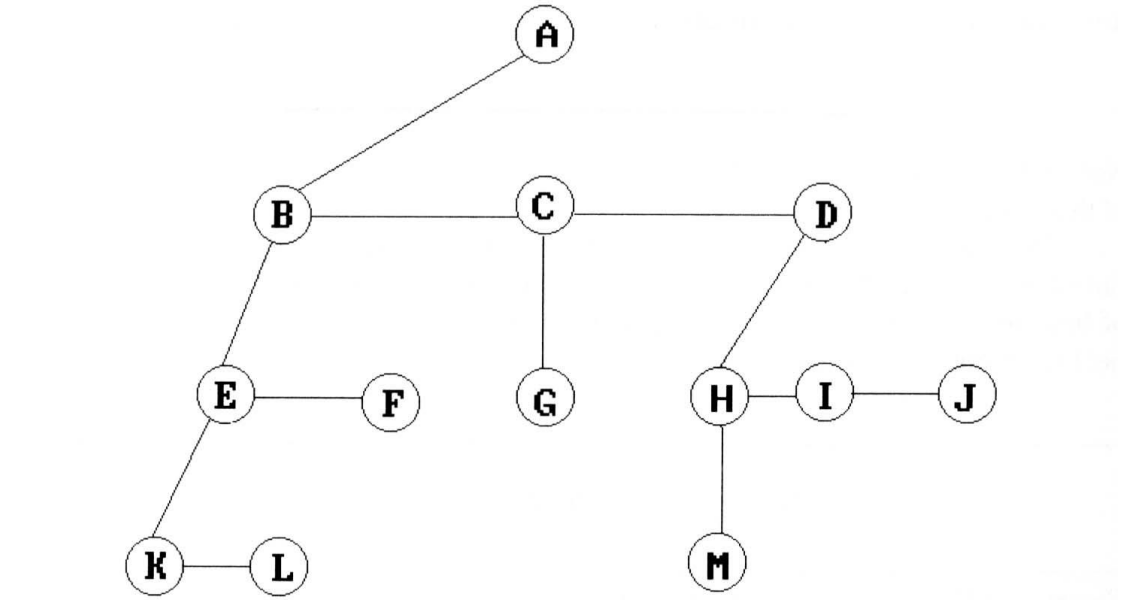


Figure 5.5: Left child-right sibling representation of a tree

Introduction

- Representation Of Trees (cont'd)
  - Representation As A Degree Two Tree

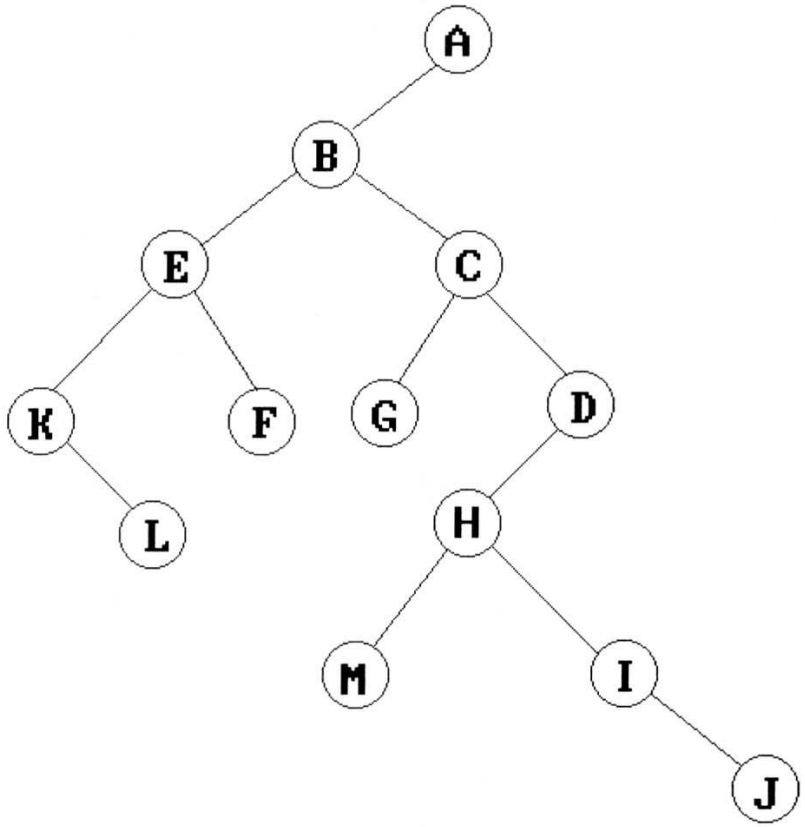


Figure 5.6: Left child-right child tree representation of a tree

Binary Trees

- Binary trees are characterized by the fact that any node can have at most two branches

- **Definition** (recursive):
  - A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree

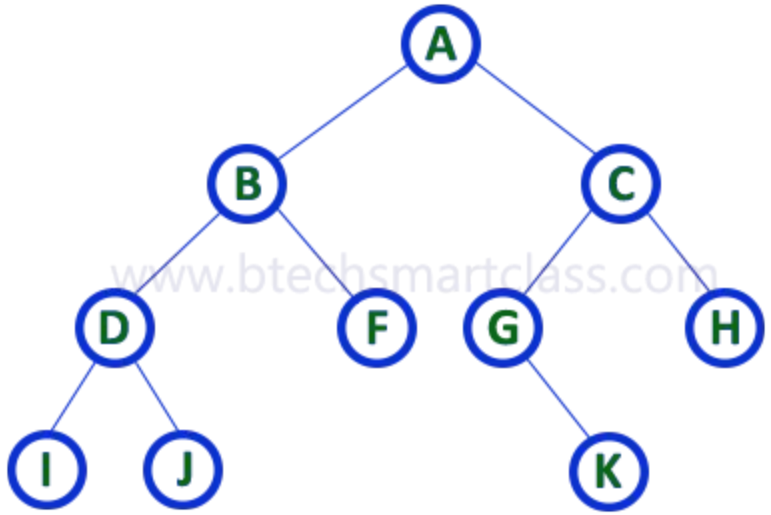
- Thus the left subtree and the right subtree are distinguished

Representation using sequential and linked organization

Binary Tree Representation

1. Sequential representation using arrays
2. List representation using Linked list

Sequential representation



Here d=3

Required array Size = 15

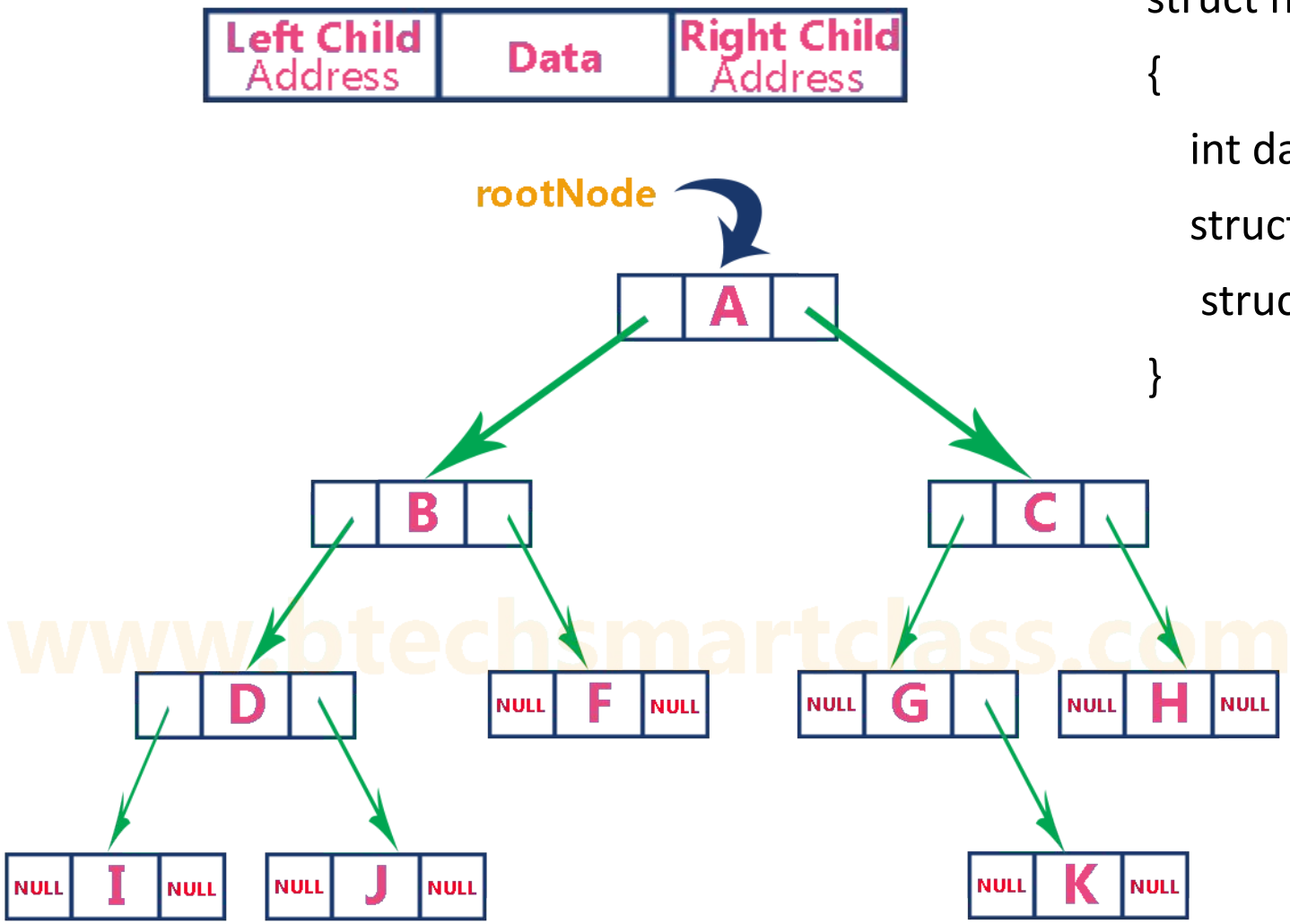
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-

- To represent a binary tree of depth **'d'** using array representation, we need one dimensional array with a maximum size of $\mathbf{2^{d+1} - 1}$.

Sequential representation

- Advantages:
  - Direct access to all nodes (Random access)
- Disadvantages:
  - Height of tree should be known
  - Memory may be wasted
  - Insertion and deletion of a node is difficult

List representation

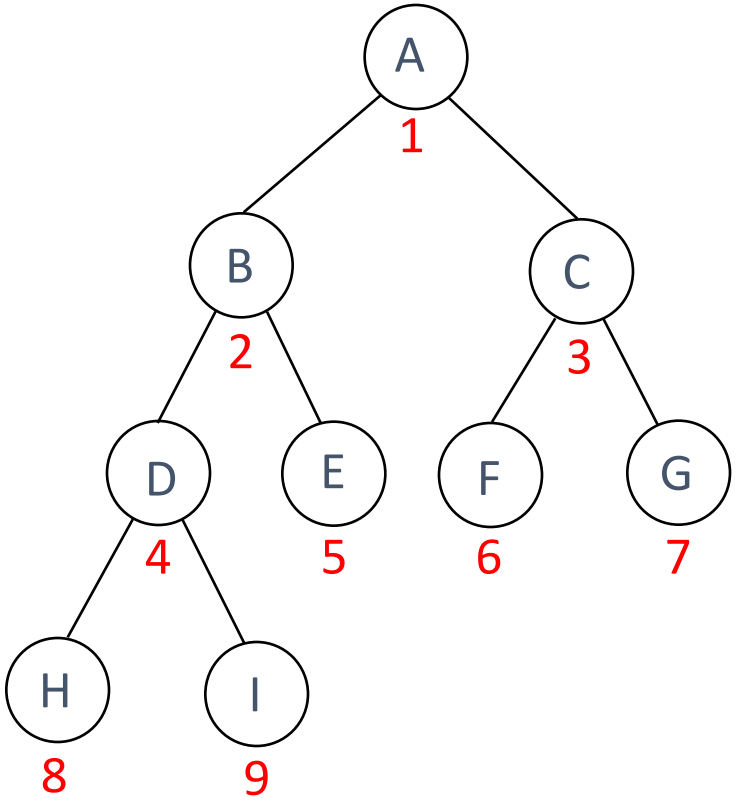



```
struct node
{
    int data;
    struct node *left;
     struct node *rifgt;
}
```

List representation

- Advantages:
  - Height of tree need not be known
  - No memory wastage
  - Insertion and deletion of a node is done without affecting other nodes
- Disadvantages:
  - Direct access to node is difficult
  - Additional memory required for storing address of left and right node

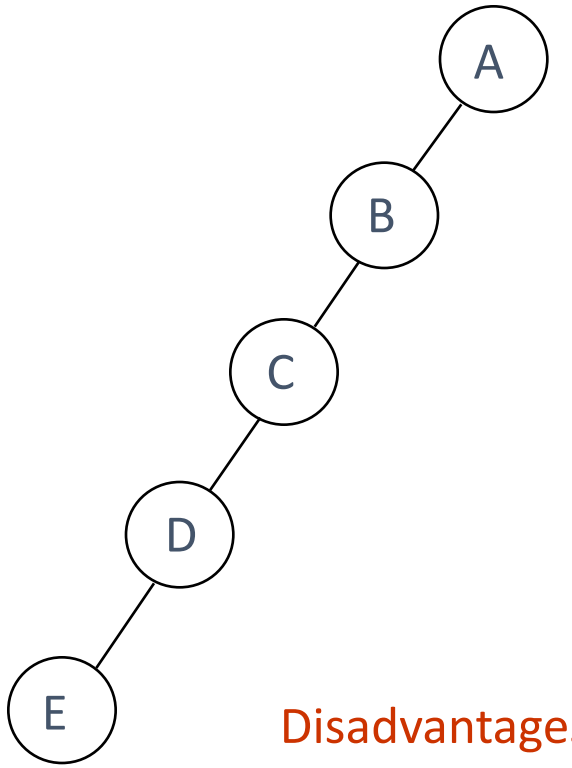
Array Representation



(in computer)

[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Array Representation: Example



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

Disadvantages:

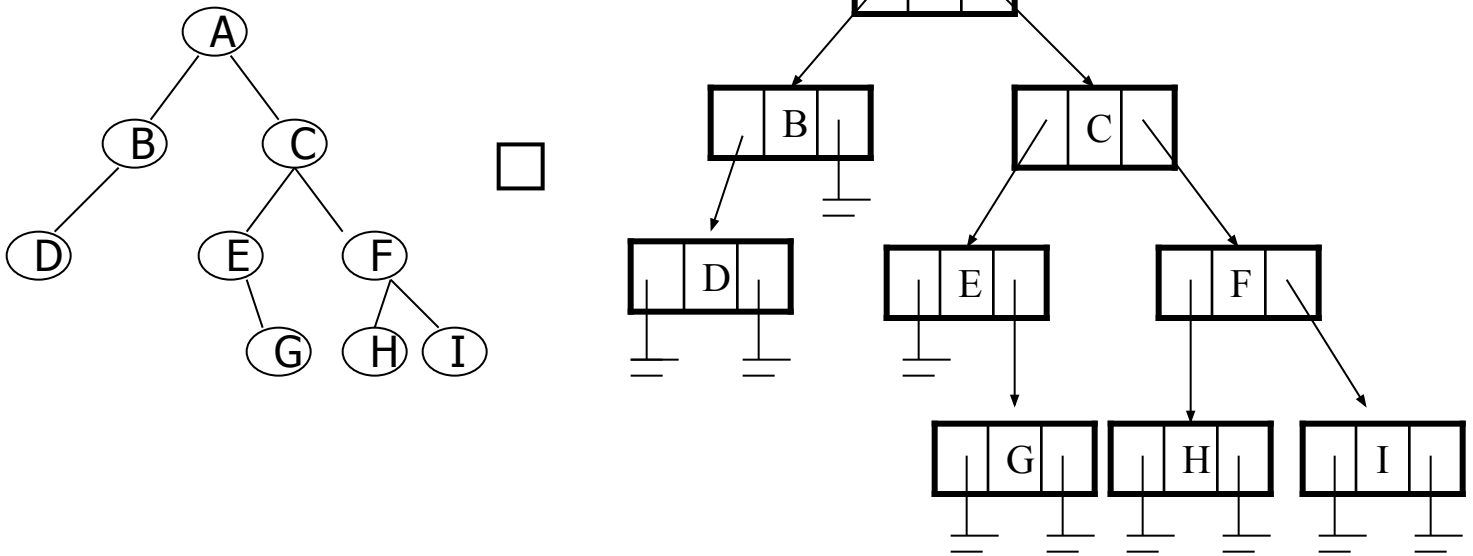
(1) waste space

(2) insertion/deletion problem

Linked Representation

- Node structure:

```
typedef struct node *tree_pointer;
typedef struct node {
      int data;
      tree_pointer left_child, right_child;
};
```



Binary tree- properties

- **A binary tree is a finite set of nodes that is either empty or consist a root node and two disjoint binary trees called the left subtree and the right subtree.**
- In other words, **a binary tree is a non-linear data structure in which each node has maximum of two child nodes. The tree connections can be called as branches**.

Binary Trees

- Properties of binary trees
  - **Lemma 5.1 [*Maximum number of nodes*]:**

1. The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
2. The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.

  - **Lemma 5.2 [*Relation between number of leaf nodes and degree-2 nodes*]:**

    For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Binary Trees

- Definition:
  - A *full binary tree* of depth *k* is a binary tree of death *k* having 2^k-1 nodes, k ≥ 0.
  - A binary tree with n nodes and depth k is complete *iff* its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.
  - From Lemma 5.1, the height of a complete binary tree with *n* nodes is $\lceil \log_2(n+1) \rceil$

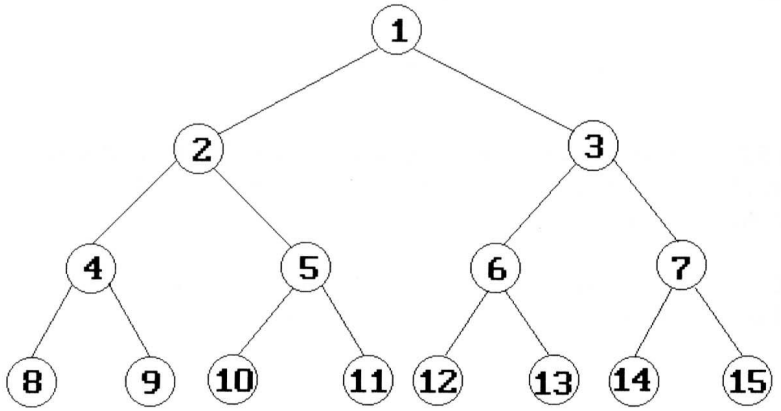
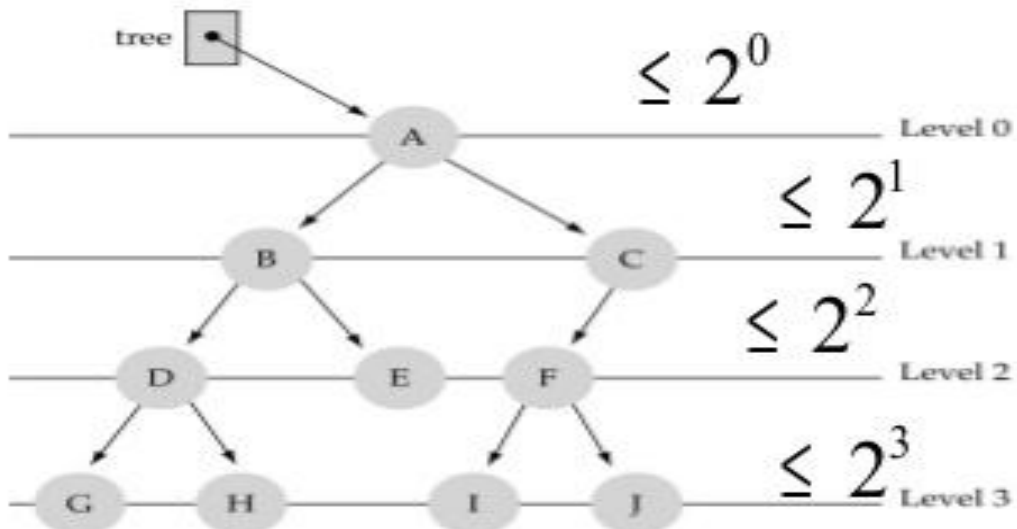


Figure 5.10: Full binary tree of depth 4 with sequential node numbers

What is the max number of nodes at any level l ?

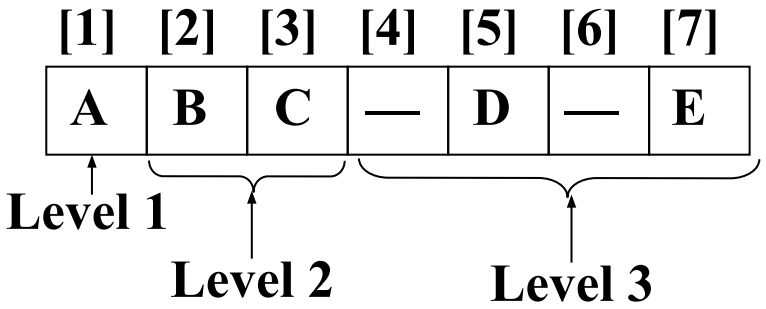
The maximum number of nodes at any level l is less than or equal to 2^l where l=0, 1, 2, 3,..... L-1

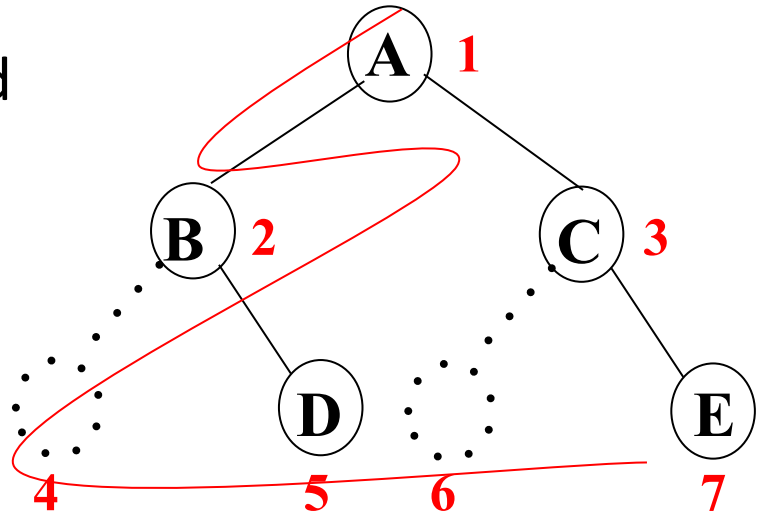


Binary Trees

- Binary tree representations (using array)
  - **Lemma 5.3:** If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have

1. *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
   If $i = 1$, i is at the root and has no parent.
2. *LeftChild(i)* is at $2i$ if $2i \le n$.
   If $2i > n$, then i has no left child.
3. *RightChild(i)* is at $2i+1$ if $2i+1 \le n$.
   If $2i+1 > n$, then i has no left child





Binary Trees

- Binary tree representations (using array)
  - Waste spaces: in the worst case, a skewed tree of depth *k* requires 2^k-1 spaces. Of these, only *k* spaces will be occupied
  - Insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes

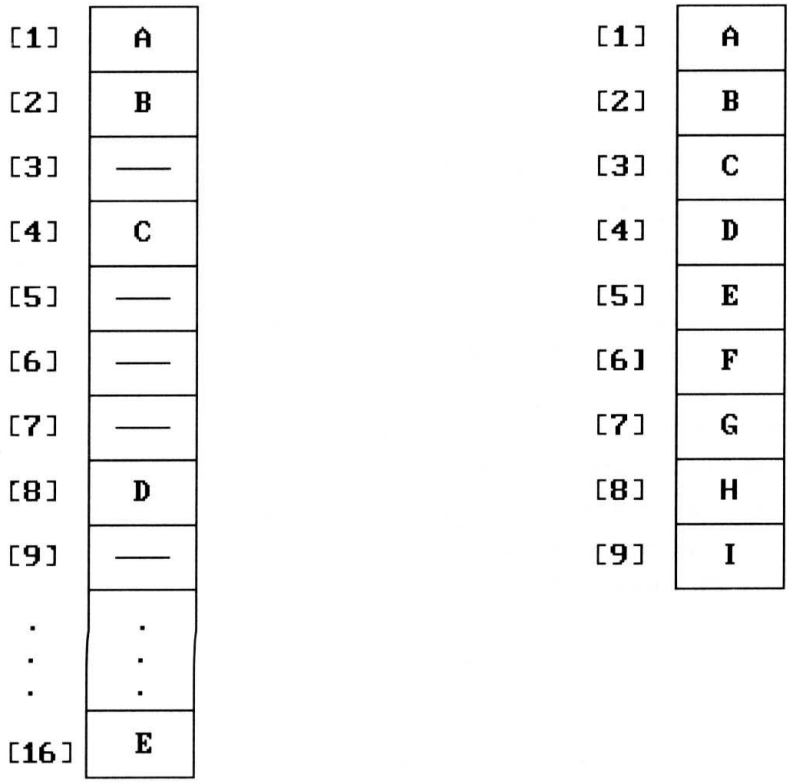


Figure 5.11: Array representation of binary trees of Figure 5.9

Binary Trees

- Binary tree representations (using link)

```
typedef struct node *tree_pointer;
typedef struct node {
        int data;
        tree_pointer left_child, right_child;
        };
```

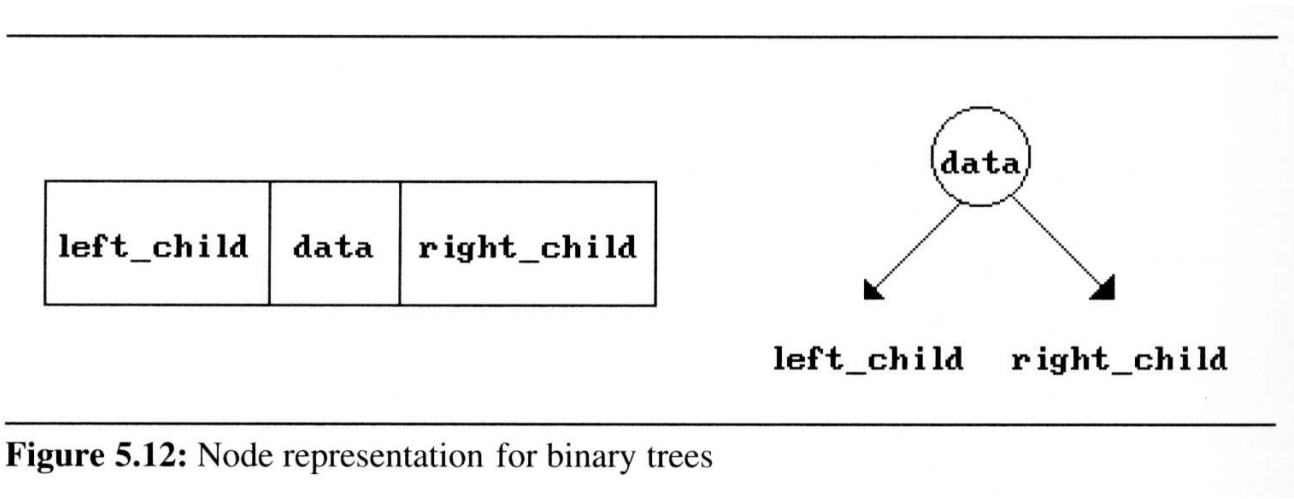


Figure 5.12: Node representation for binary trees

Binary Trees

- Binary tree representations (using link)

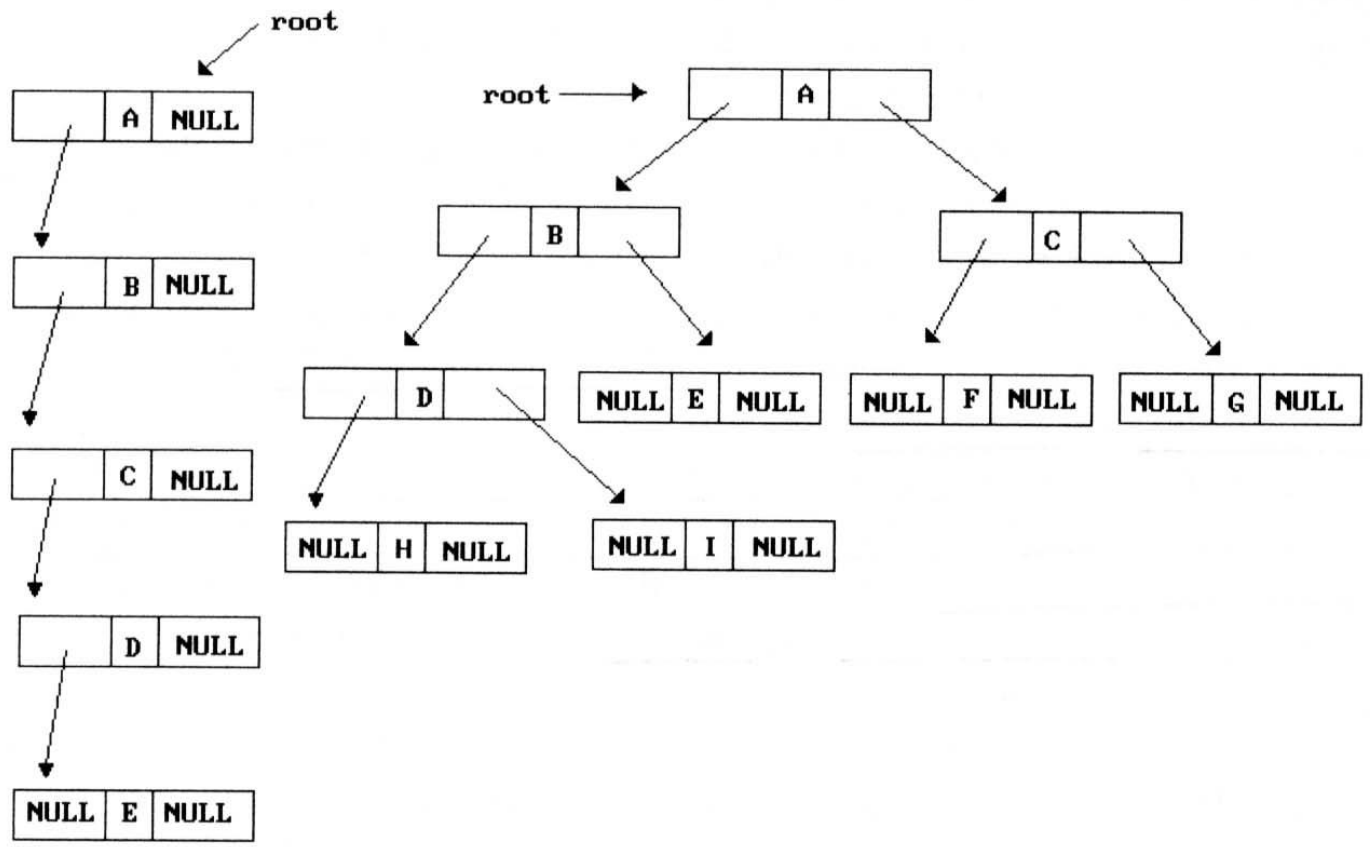
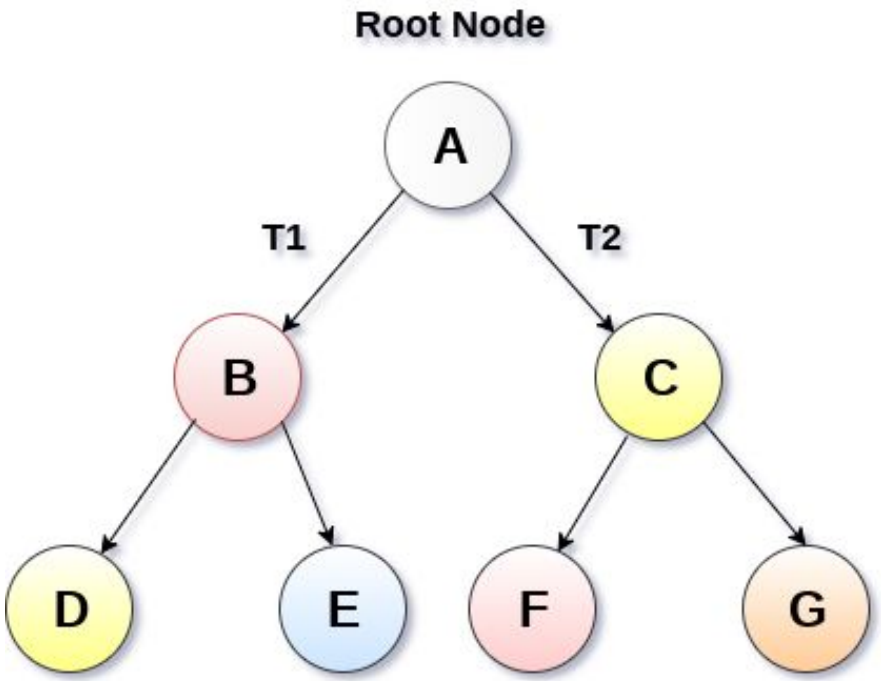


Figure 5.13: Linked representation for the binary trees of Figure 5.9

Binary Tree

- A tree in which every node can have a maximum of two children is called as Binary Tree.



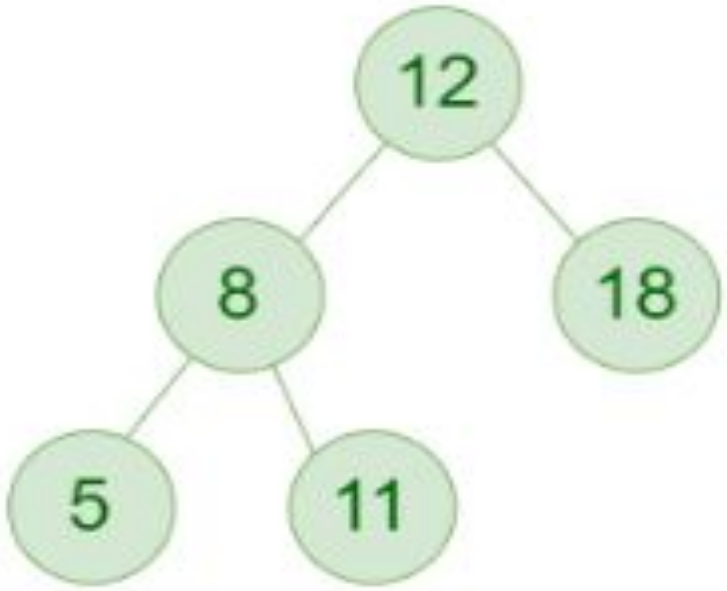
Binary Tree

Types of binary tree:

1. Full Binary Tree
2. Complete Binary Tree
3. Skewed Binary Tree

Full Binary Tree

A binary tree with either zero or two child nodes for each node (does not have any nodes that have only one child node)



Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.

- Complete binary tree is also called as **Perfect Binary Tree**.

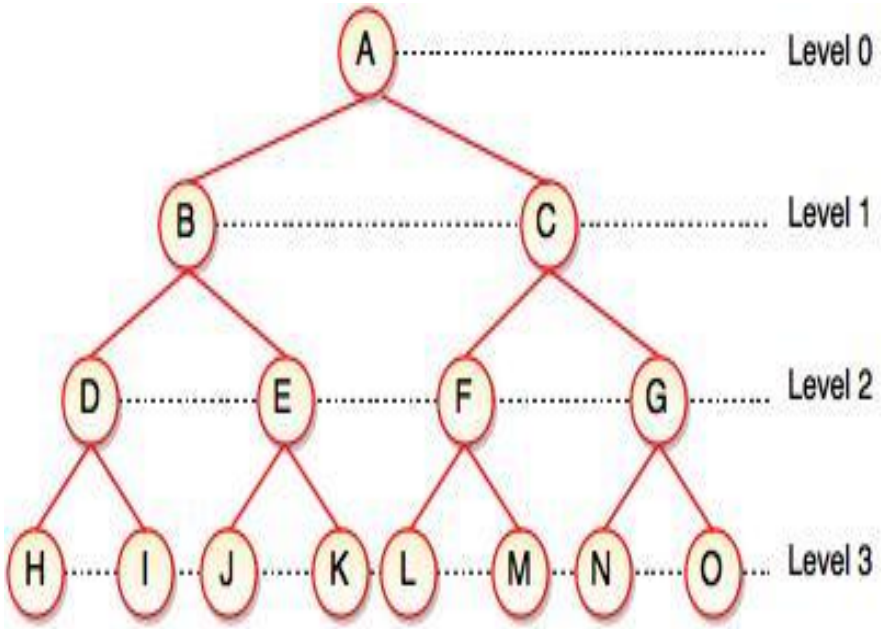


Fig. Complete Binary Tree

- A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree. Complete binary tree is also called as **Perfect Binary Tree**

Number of nodes = $2^{d+1} - 1$
Number of leaf nodes = 2^{d}
Where, d – Depth of the tree

For example, at Level 2, there must be $2^2 = 4$ nodes and at Level 3 there must be $2^3 = 8$ nodes.

Skewed Trees

•If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.

Binary tree has only left sub trees
- Left Skewed Trees

Binary tree has only right sub trees
- Right Skewed Trees

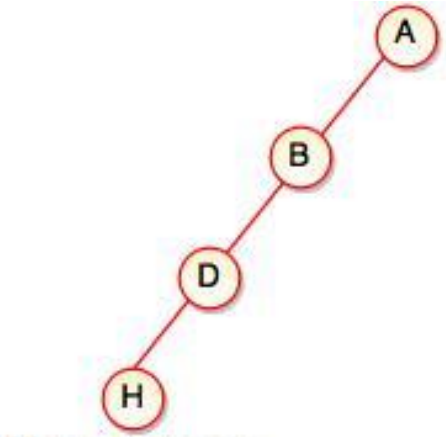


Fig. Left Skewed Binary Tree

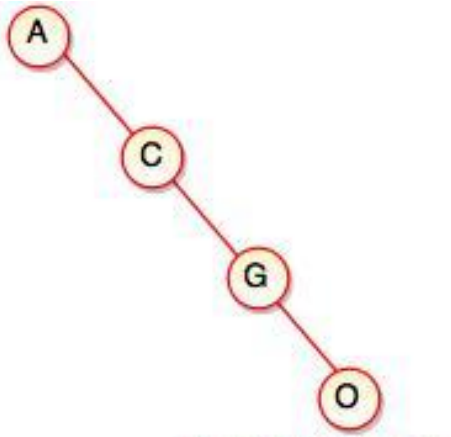


Fig. Right Skewed Binary Tree

•In a left skewed tree, most of the nodes have the left child without corresponding right child.

•In a right skewed tree, most of the nodes have the right child without corresponding left child.

Converting tree to binary tree

We can convert the given general tree into equivalent binary tree as follows -

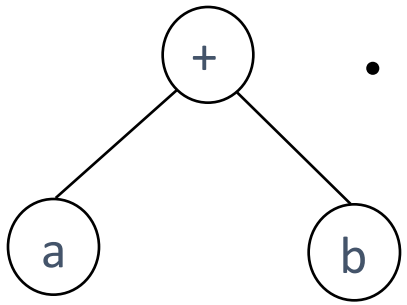
i) The root of general tree becomes the root of the binary tree.

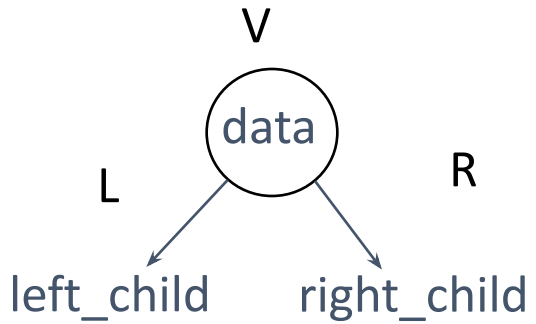
ii) Find the first child node of the node attach it as a left child to the current node in binary tree.

iii) The right sibling can be attached as a right child of that node.

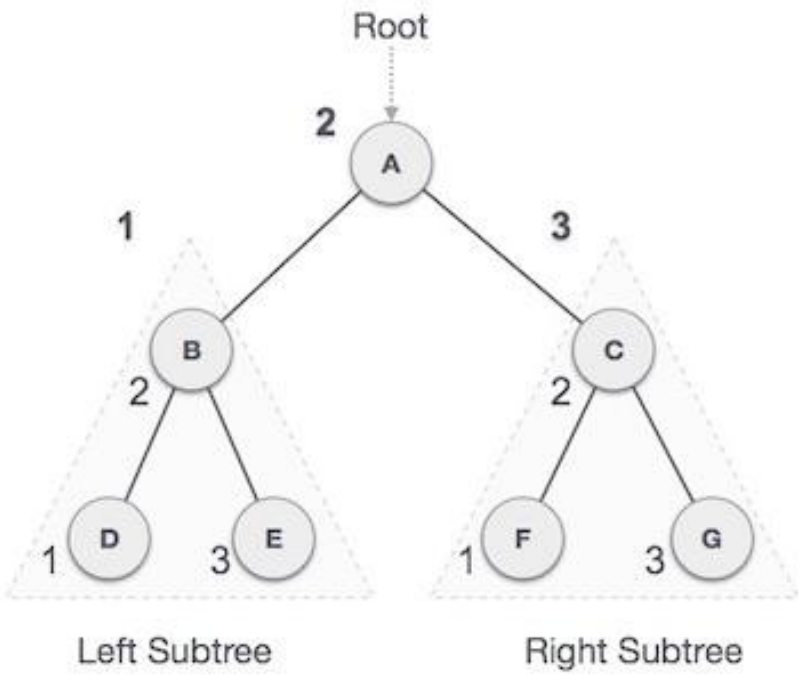
Binary Tree Traversals

- **Binary Tree Traversals: Combinations**
  - Let L, V, and R stand for moving left, visiting the node, and moving right.
  - There are six possible combinations of traversal:
    LVR, LRV, VLR, VRL, RVL, RLV
  - Adopt convention that we traverse left before right, only 3 traversals remain
    - LVR: inorder, a+b
    - LRV: postorder, ab+
    - VLR: preorder, +ab





In-order Traversal Algorithm



D → B → E → A → F → C → G

```
Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.
```

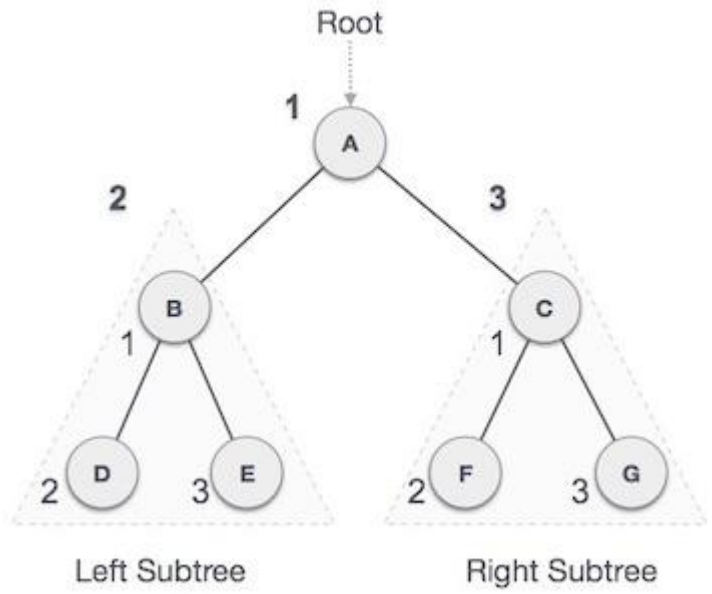
Algorithm

- **Step 1:** Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** INORDER(TREE -> LEFT)
- **Step 3:** Write TREE -> DATA
- **Step 4:** INORDER(TREE -> RIGHT)

[END OF LOOP]

- **Step 5:** END

Pre-order Traversal Algorithm



A → B → D → E → C → F → G

```
Until all nodes are traversed –
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.
```

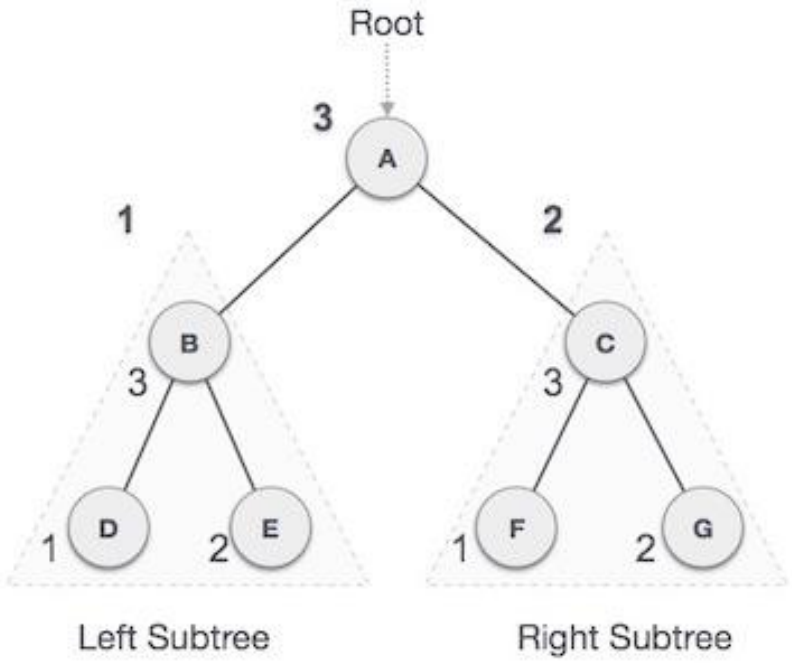
Algorithm

- **Step 1:** Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** Write TREE -> DATA
- **Step 3:** PREORDER(TREE -> LEFT)
- **Step 4:** PREORDER(TREE -> RIGHT)

[END OF LOOP]

- **Step 5:** END

Post-order Traversal Algorithm




```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Recursively traverse right subtree.
Step 3 – Visit root node.
```

Algorithm

- **Step 1**: Repeat Steps 2 to 4 while TREE != NULL
- **Step 2:** POSTORDER(TREE -> LEFT)
- **Step 3:** POSTORDER(TREE -> RIGHT)
- **Step 4:** Write TREE -> DATA

[END OF LOOP]

- **Step 5:** END

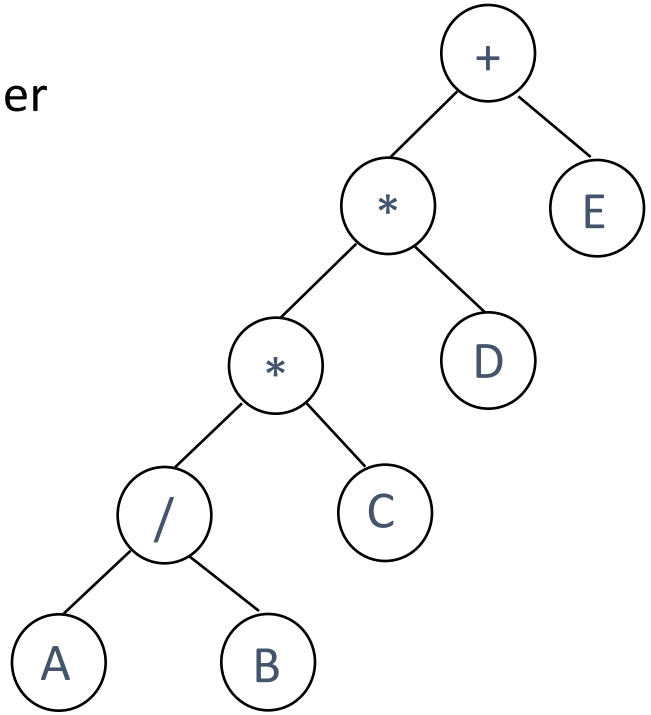
D → E → B → F → G → C → A

Binary Tree Traversal: Inorder

- **Inorder traversal**
  - Traverse the left subtree in inorder
  - visit the root
  - Traverse the right subtree in inorder

(recursive version)

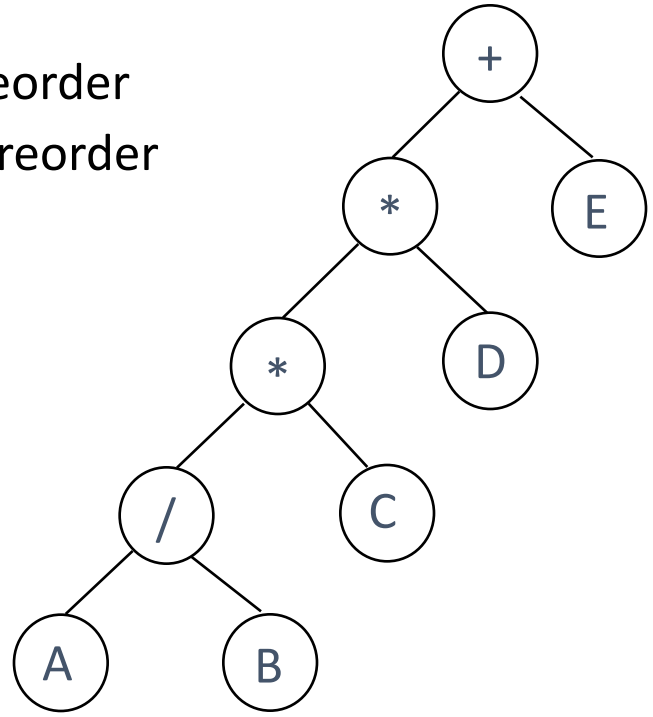
```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
     if (ptr) {
          inorder(ptr->left_child);
          printf("%d", ptr->data);
          indorder(ptr->right_child);
     }
}
```



A / B * C * D + E

Binary Tree Traversal: Preorder

- **Preorder traversal**
  - visit the root
  - Traverse the left subtree in preorder
  - Traverse the right subtree in preorder



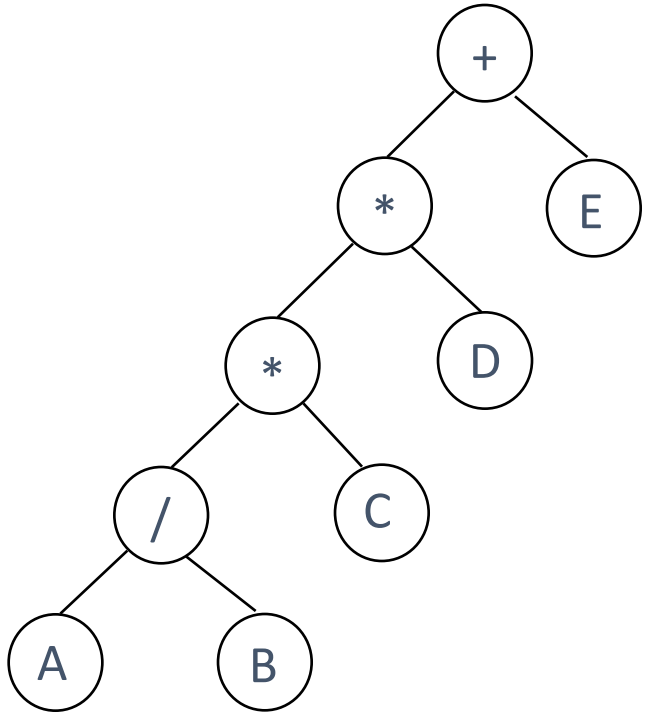
(recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
     if (ptr) {
          printf("%d", ptr->data);
          preorder(ptr->left_child);
          predorder(ptr->right_child);
     }
}
```

+ * * / A B C D E

Binary Tree Traversal: Postorder

- **Postorder traversal**
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the root



A B / C * D * E +

(recursive version)

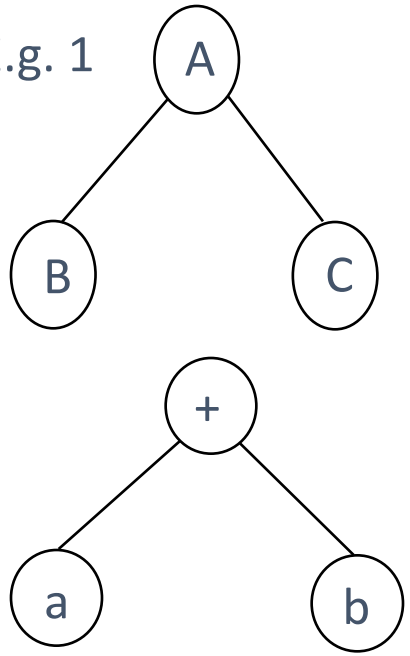
```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
     if (ptr) {
          postorder(ptr->left_child);
          postdorder(ptr->right_child);
          printf("%d", ptr->data);
     }
}
```

Binary Tree Traversal: Example

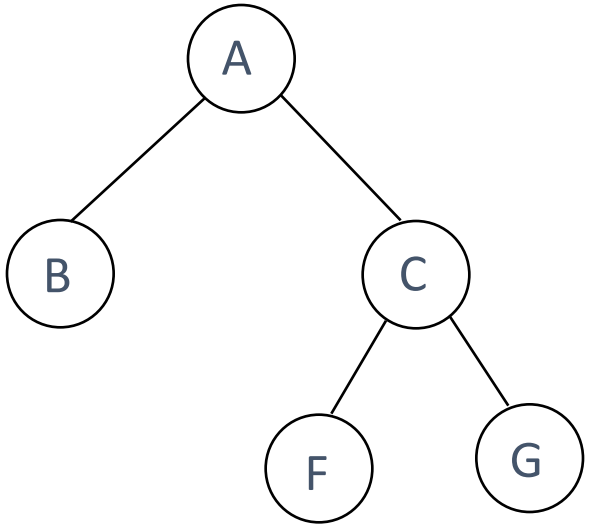
- **Traversing (walking through) a tree**
  - Inorder traversal BAC
  - Preorder traversal ABC
  - Postorder traversal BCA

a+b infix
+ab prefix
ab+ postfix

E.g. 1



E.g. 2



Inorder traversal BA____

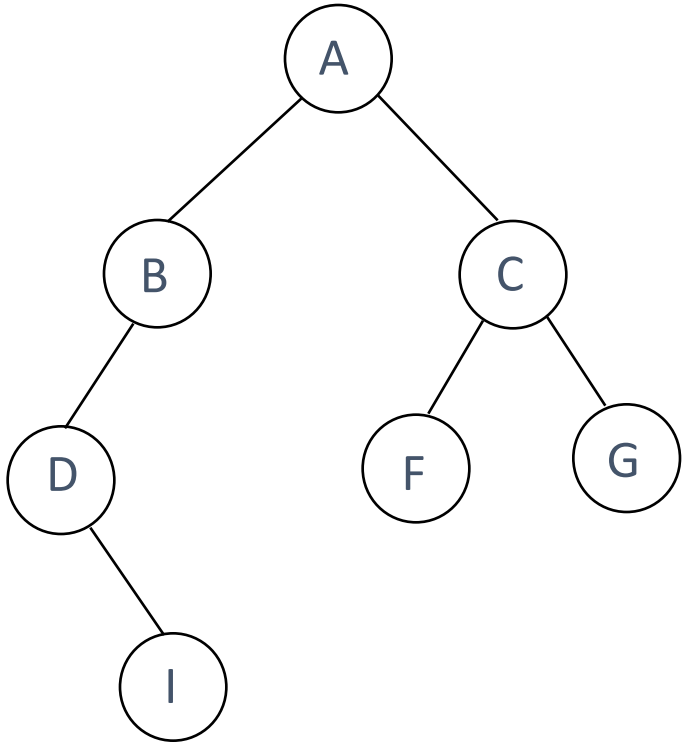
Preorder traversal ABC____

Postorder traversal _____

Binary Tree Traversal: Exercise

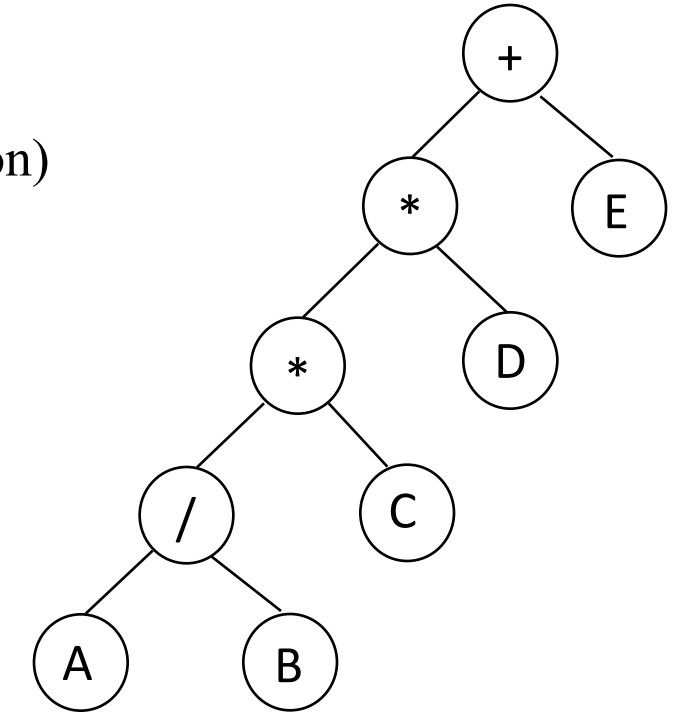
E.g. 3

- inorder traversal DIBA____
- preorder traversal ABDI____
- postorder traversal _____

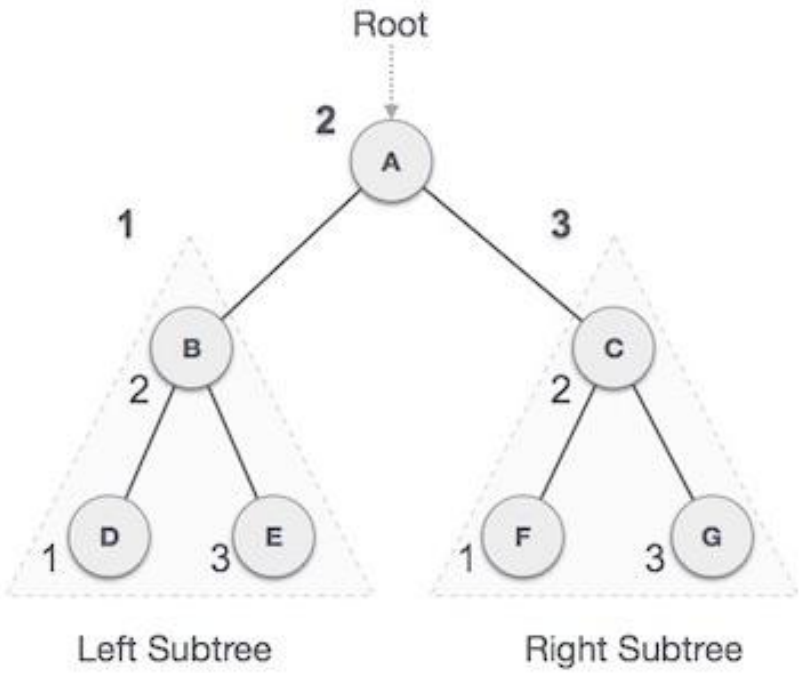


Arithmetic Expression using Binary Tree

- Arithmetic Expression: A / B * C * D + E
  - Inorder traversal (infix expression)
    A / B * C * D + E
  - Preorder traversal (prefix expression)
    + * * / A B C D E
  - Postorder traversal (postfix expression)
    A B / C * D * E +
  - Level order traversal
    + * E * D / C A B



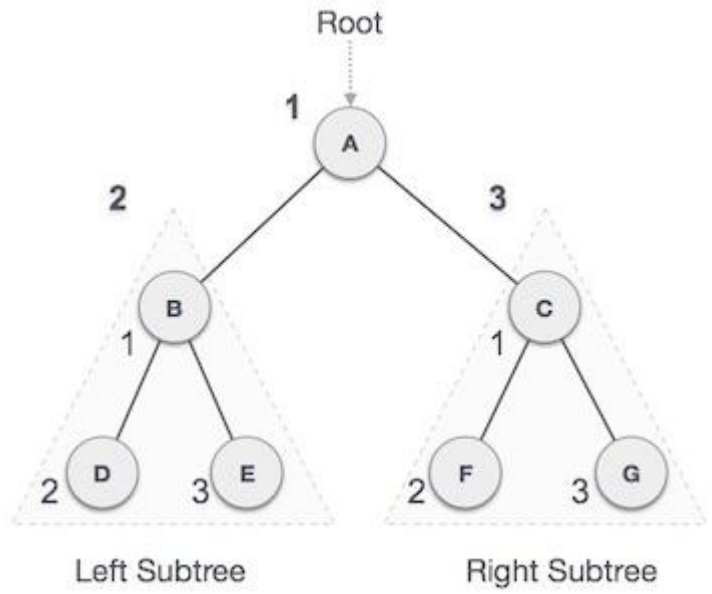
In-order Traversal Algorithm(Without Recursion using stack)



D → B → E → A → F → C → G

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

Pre-order Traversal Algorithm(Without Recursion)

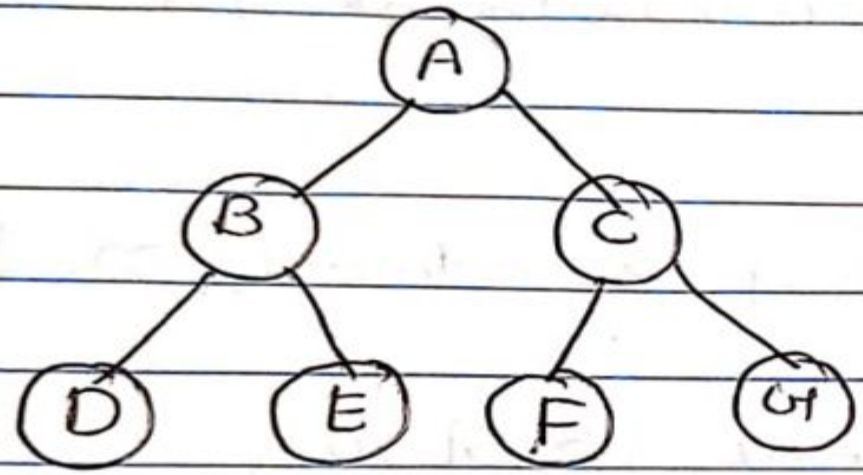


A → B → D → E → C → F → G

1. *Create an empty stack nodeStack and push root node to stack.*
2. *Do the following while nodeStack is not empty.*
    1. *Pop an item from the stack and print it.*
    2. *Push right child of a popped item to stack*
    3. *Push left child of a popped item to stack*

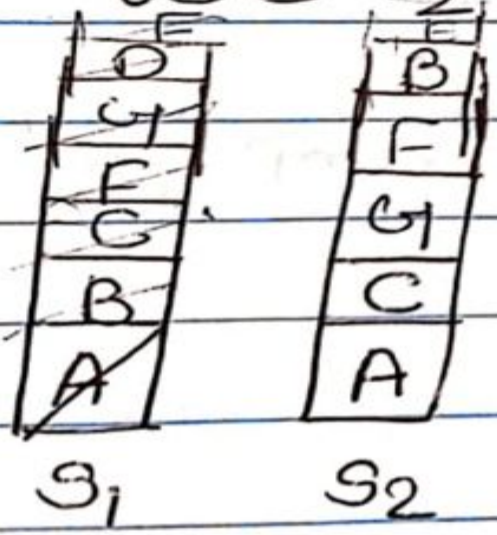
The right child is pushed before the left child to make sure that the left

Non-recursive Postorder (L R Ro)



postorder
D E B F G C A

- Use 2 stacks



D E B F G C A

```
Postorder ( )
 {
 Stack S1 , S2 ;
  node *T = root ;
    S1 . push(T);
while ( ! S1 . empty() )
    { T = S1 . POP() )
      S2 . push(T);
    if ( T -> left != NULL)
        S1 . push( T -> left),
    if ( T -> right != NULL)
      S1 . push ( T -> right) ;
    } while ( ! S2 . empty() )
        { root = S2 . POP() ;
          cout << root -> data ;
    }
```

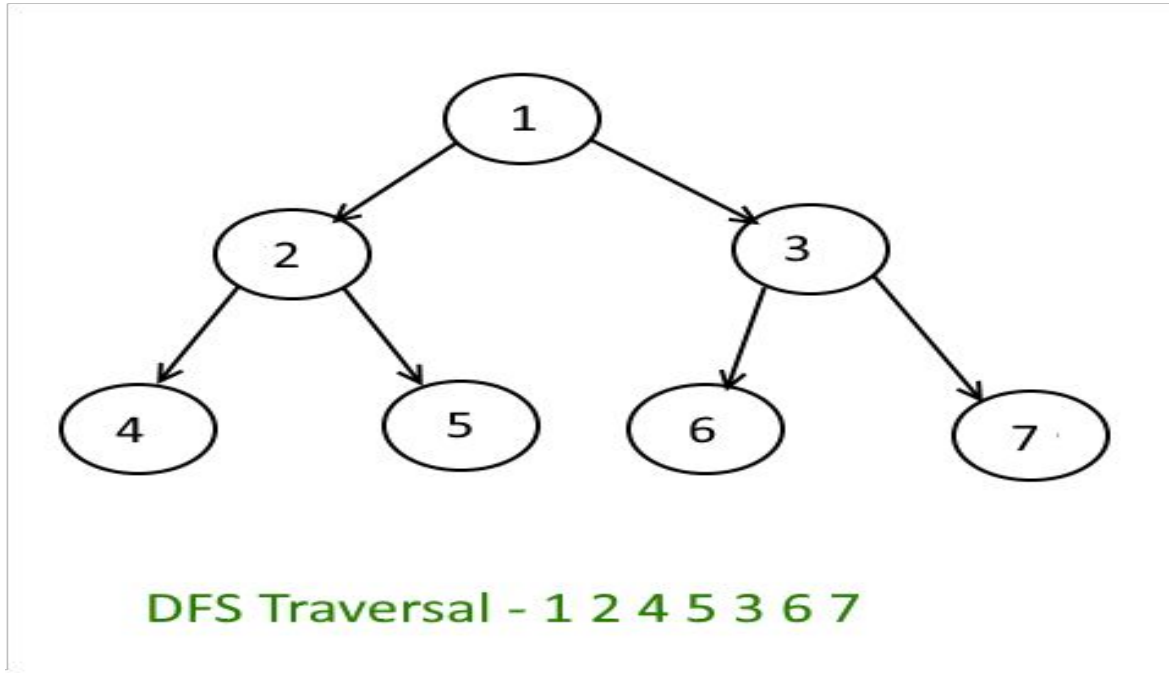
DFS (Depth First Search)Tree Traversal Algorithm

- It is a tree traversal algorithm that traverses the structure to its deepest node

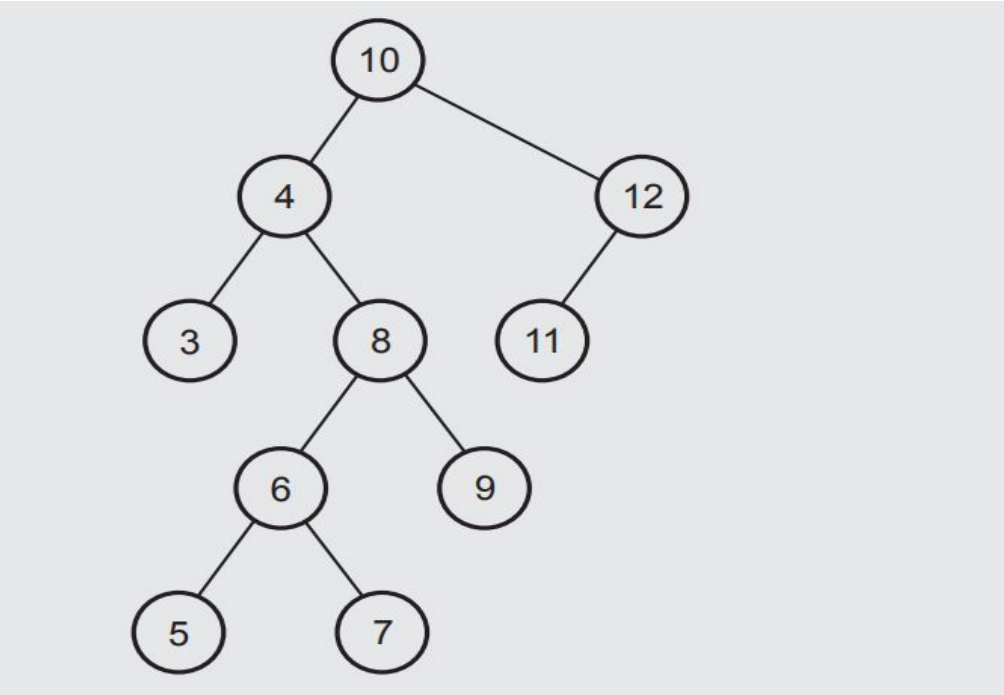
Algorithm:

Step 1 :First add the add root to the Stack.
Step 2: Pop out an element from Stack and add its right and left children to stack.
Step 3:Pop out an element and print it and add its children.
Step 4 :Repeat the above two steps until the Stack id empty.

DFS (Depth First Search)Tree Traversal Algorithm

- 

DFS (Depth First Search)Tree Traversal Algorithm



DFS Traversal : 10 4 3 8 6 5 7 9 12 11

BFS (Breadth First Search) Tree Traversal Algorithm

- It is a tree traversal algorithm that is also known as Level Order Tree Traversal.
- In this traversal we will traverse the tree row by row i.e. 1st row, then 2nd row

BFS (Breadth First Search) Tree Traversal Algorithm

Algorithm :

Step 1 : Visit the root node . Insert it in Queue from the rear end.

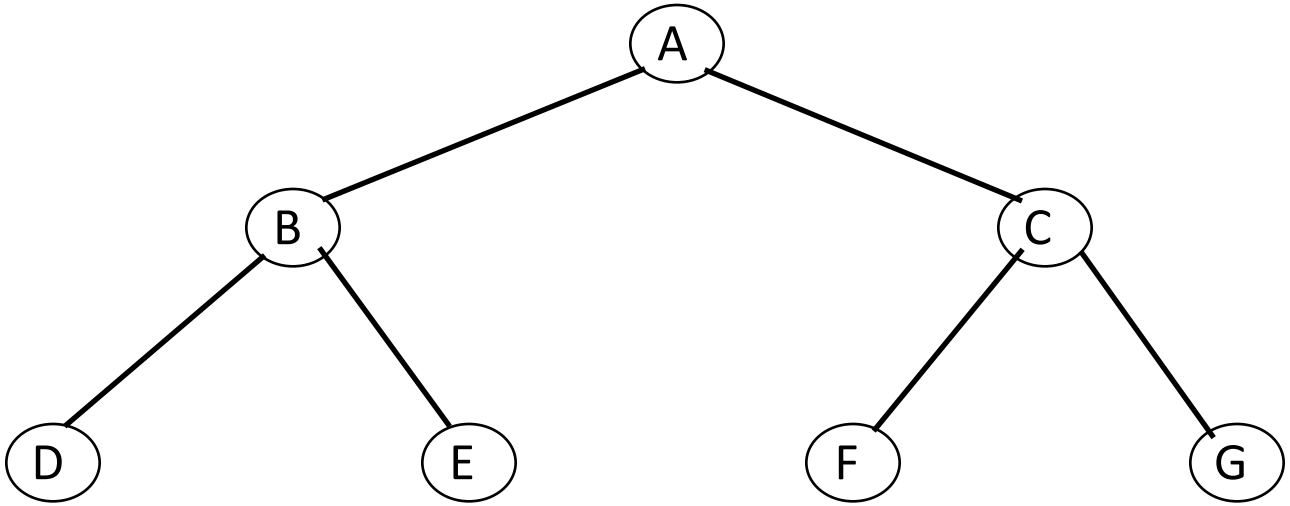
Step 2 : Delete the node from front end of the queue and display it as output.

Step 3 : If its left child is not null insert the left child in the queue.

Step 4 : If Its right child is not null, insert the right child in the queue.

Step 5 : Repeat step 2 to Step 4 until queue is not empty and all nodes are visited.

Breadth-First Search

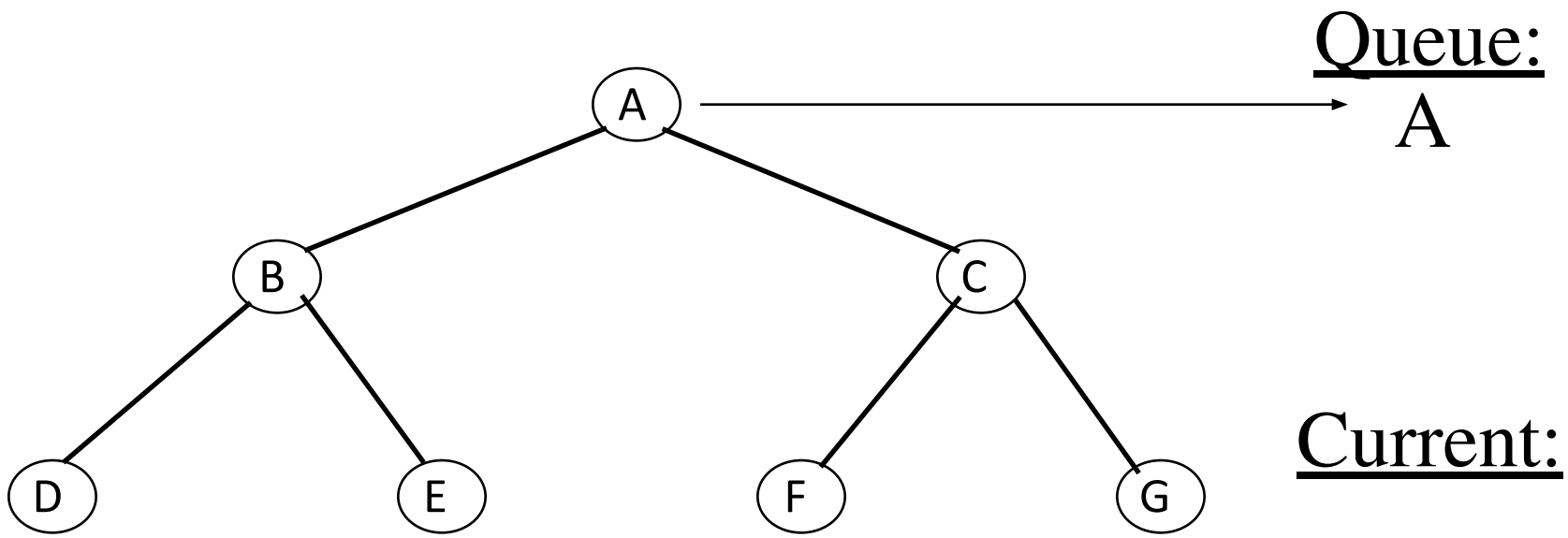


Queue:

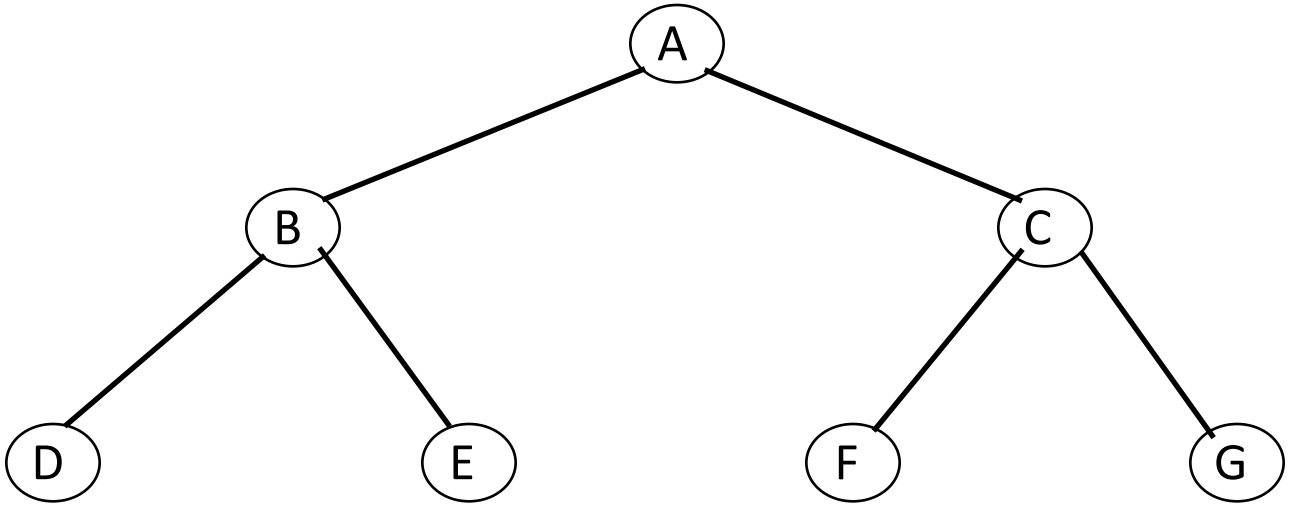
Current:

A B C D E F G

Breadth-First Search



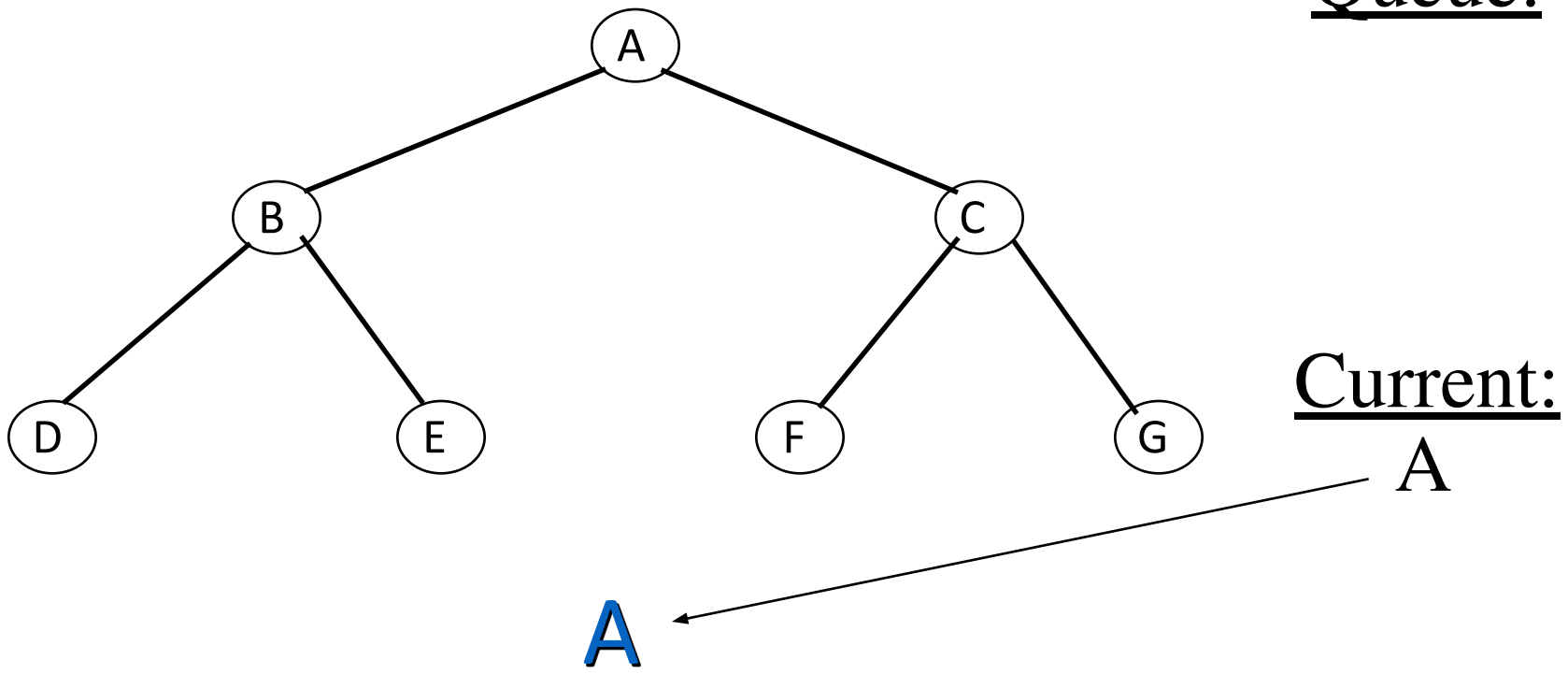
Breadth-First Search



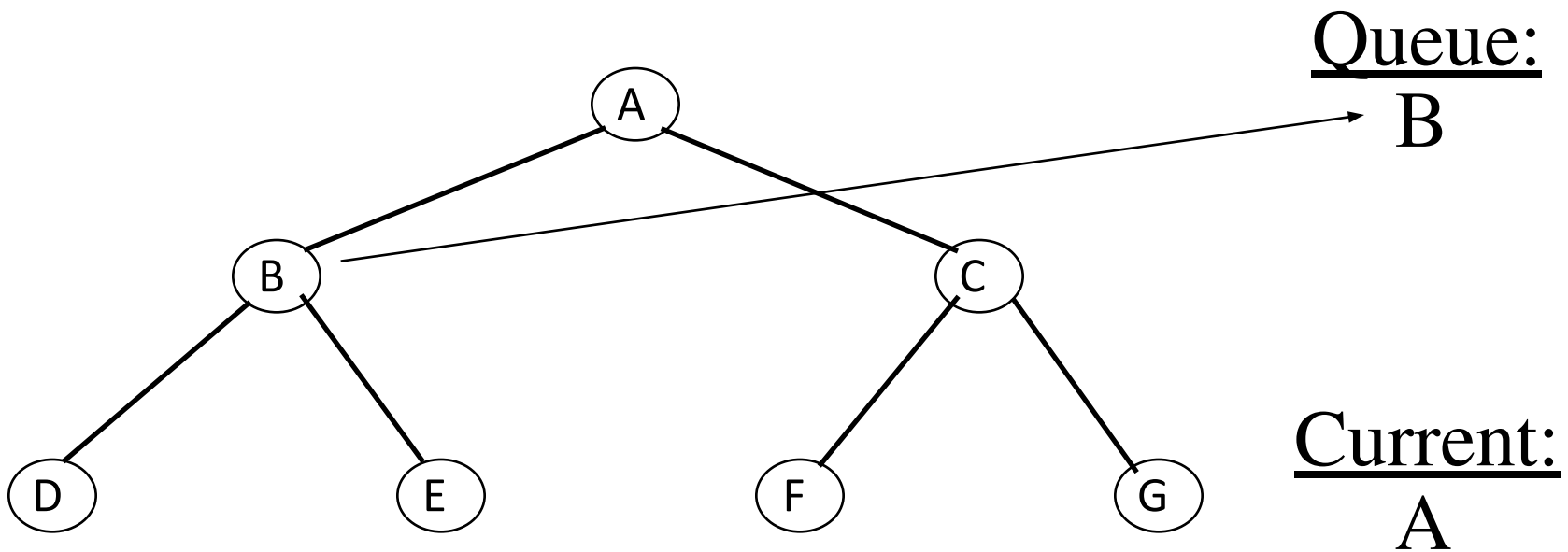
Queue:
A

Current:
A

Breadth-First Search

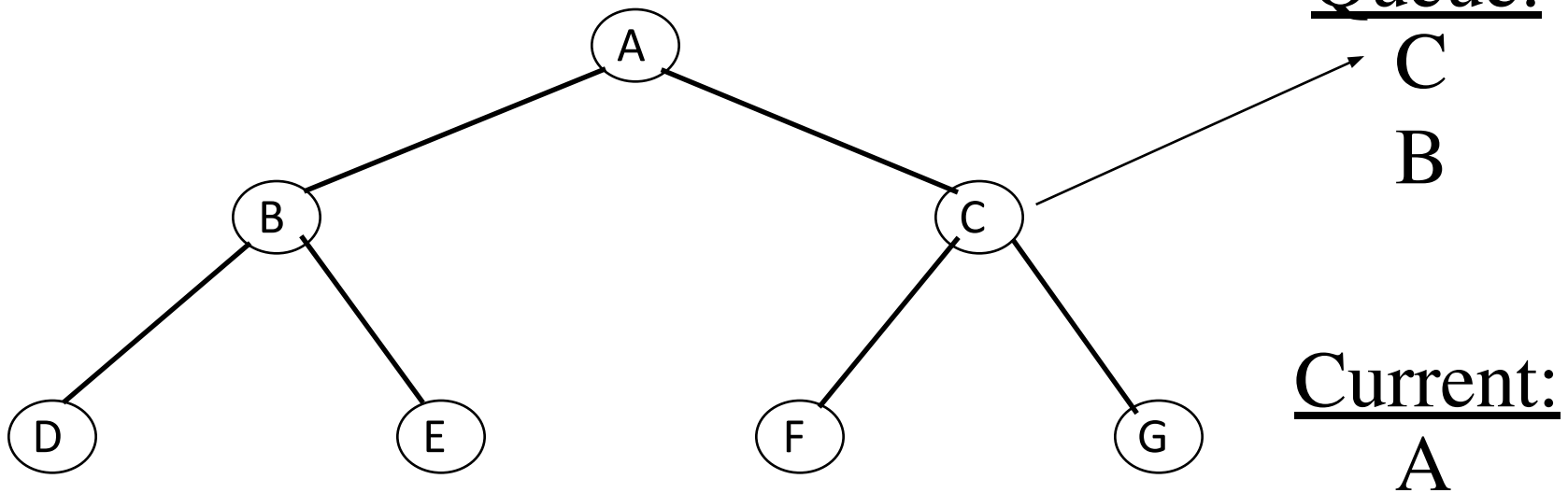


Breadth-First Search



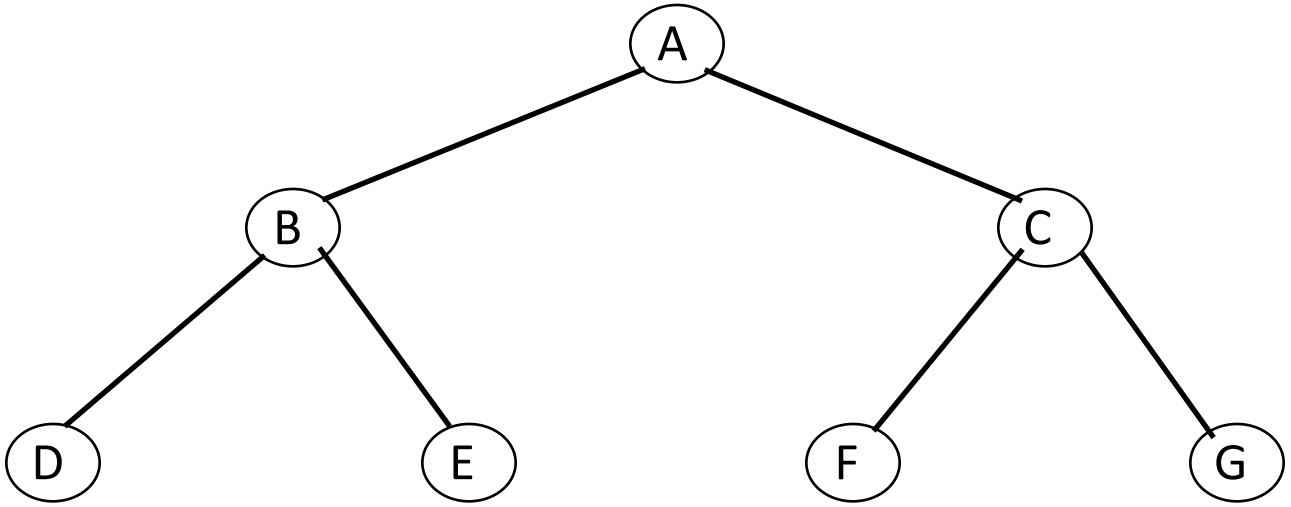
A

Breadth-First Search



A

Breadth-First Search

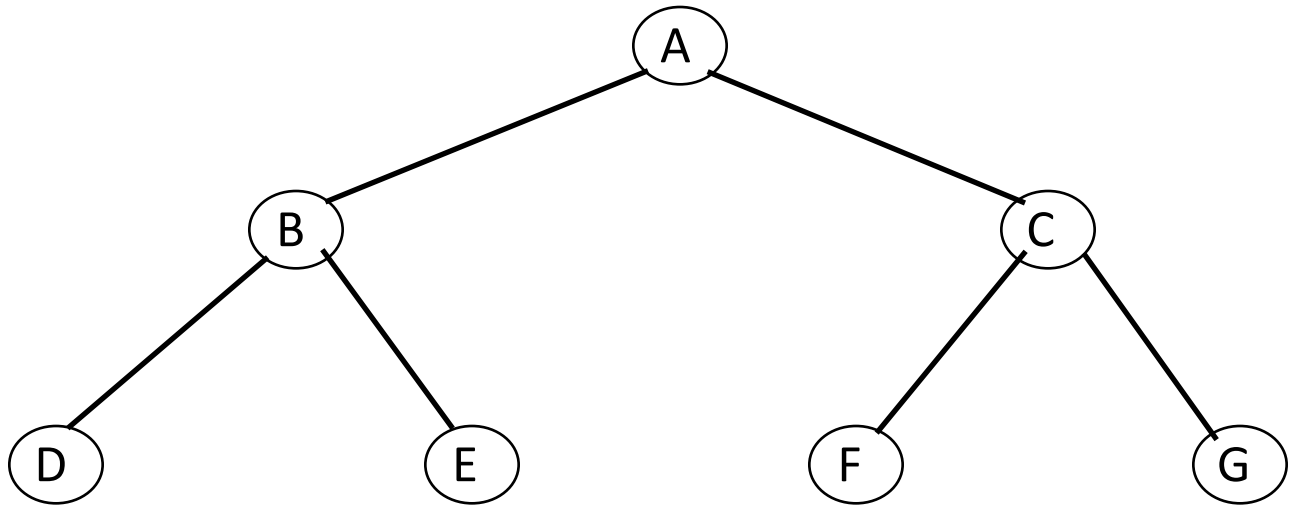


Queue:
C
B

Current:
B

A

Breadth-First Search

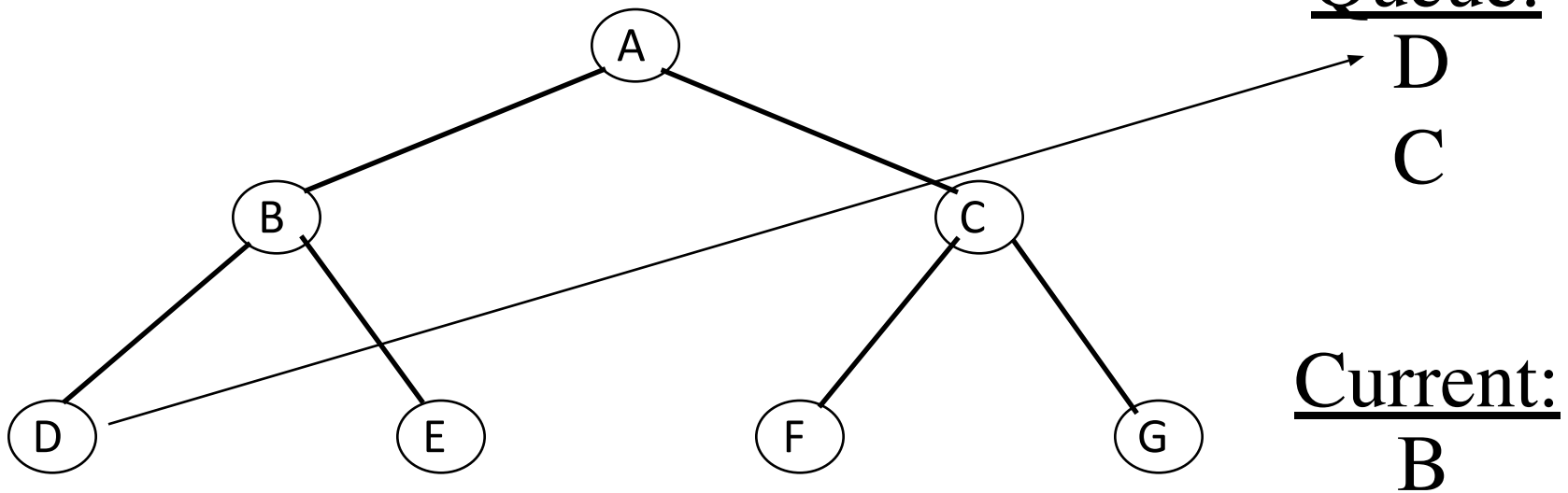


Queue:
C

Current:
B

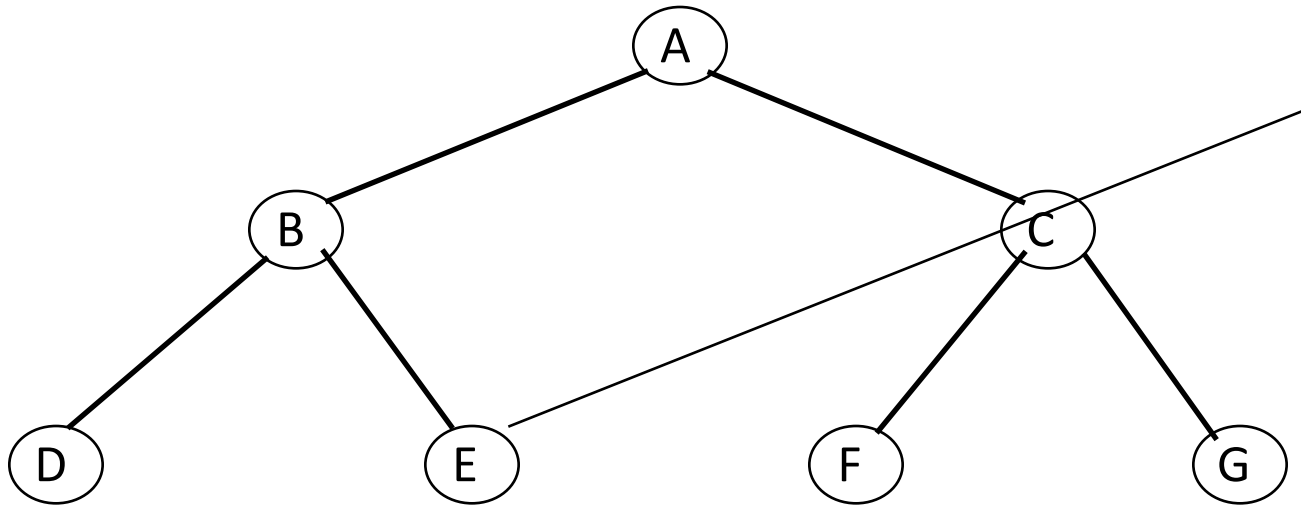
A B

Breadth-First Search



A B

Breadth-First Search

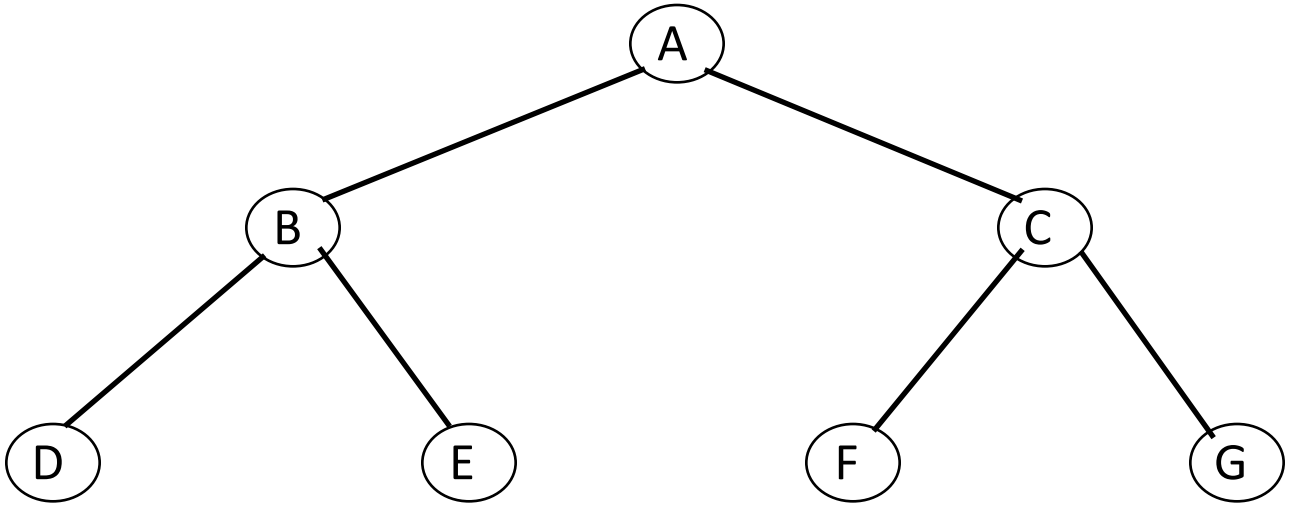


Queue:
E
D
C

Current:
B

A B

Breadth-First Search

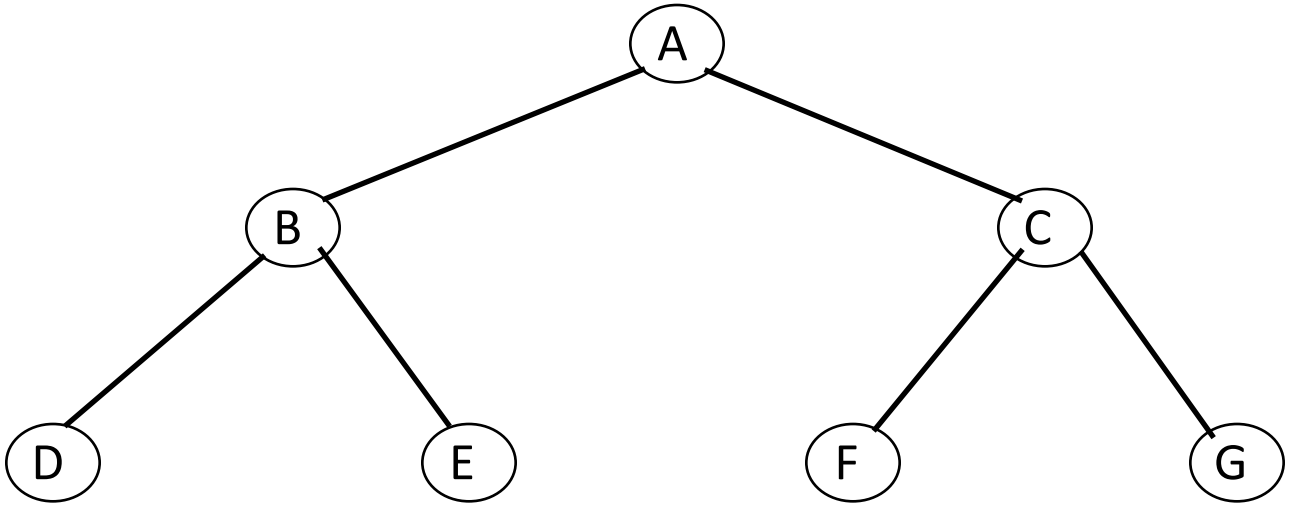


Queue:
E
D
C

Current:
C

A B

Breadth-First Search

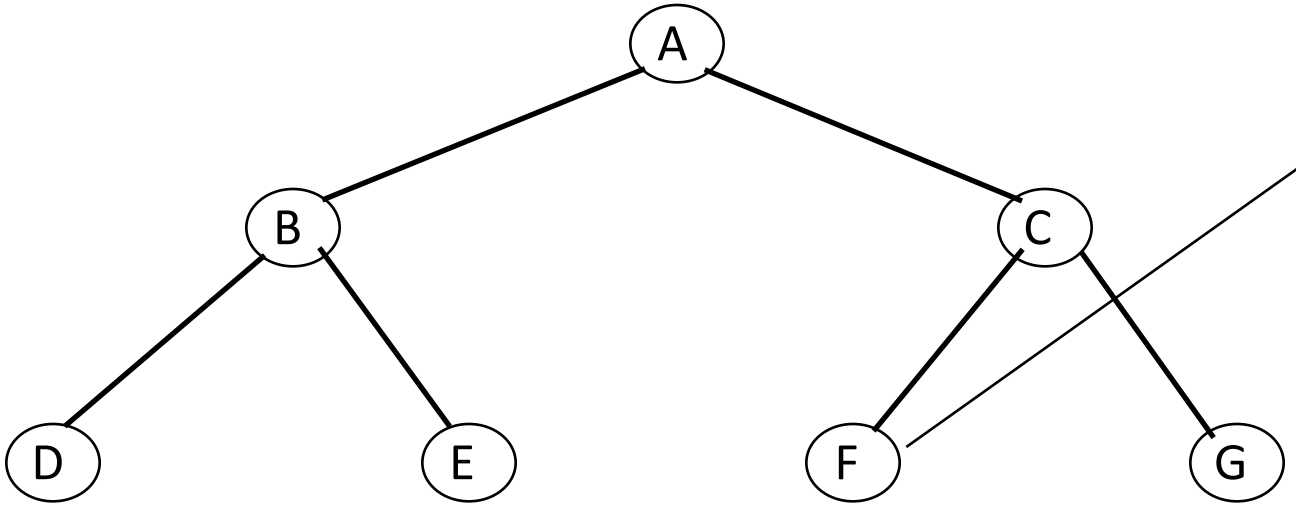


Queue:
E
D

Current:
C

A B C

Breadth-First Search

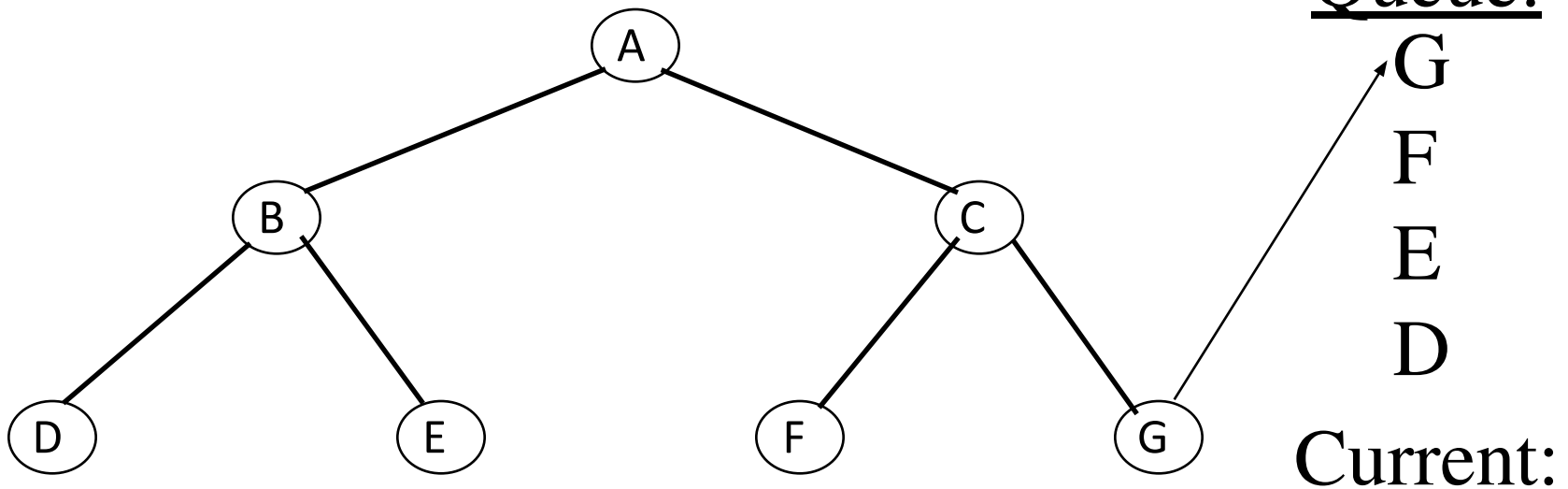


Queue:
F
E
D

Current:
C

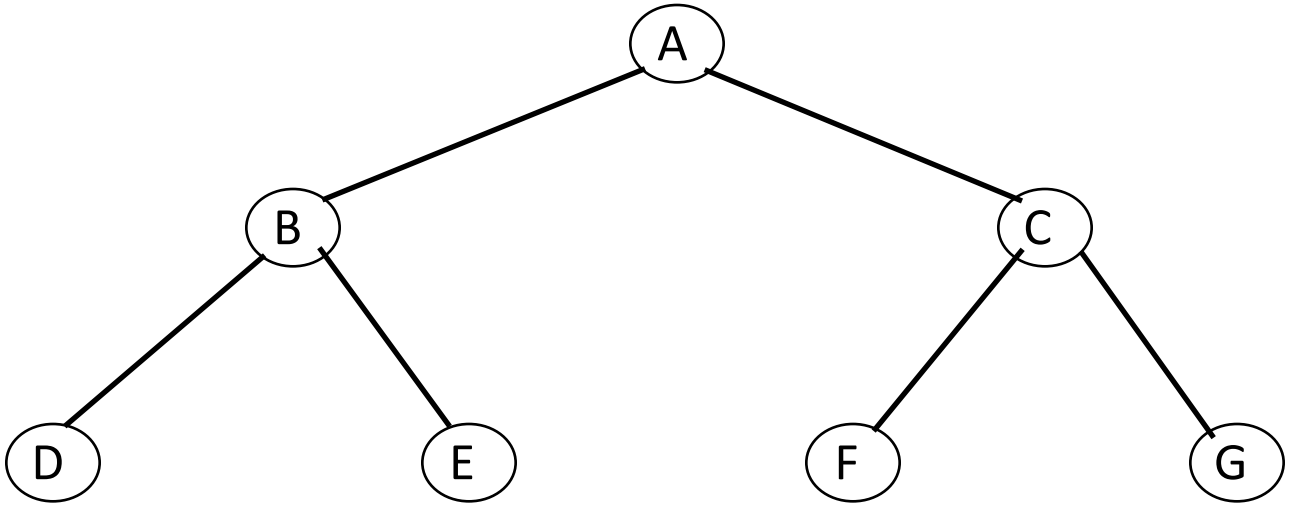
A B C

Breadth-First Search



A B C

Breadth-First Search

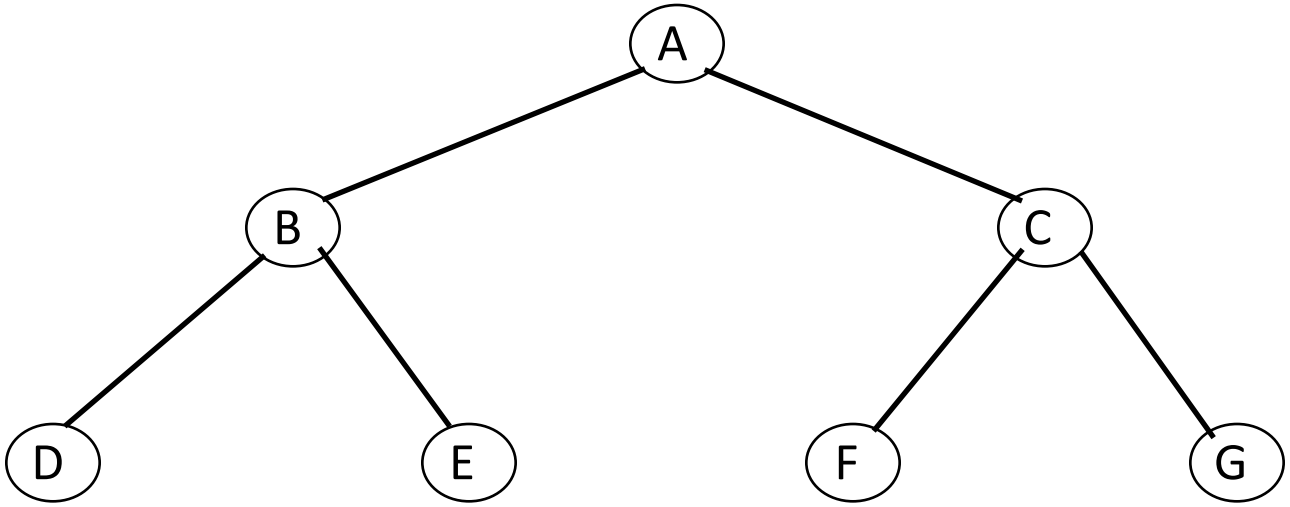


Queue:
G
F
E
D

Current:
D

A B C

Breadth-First Search

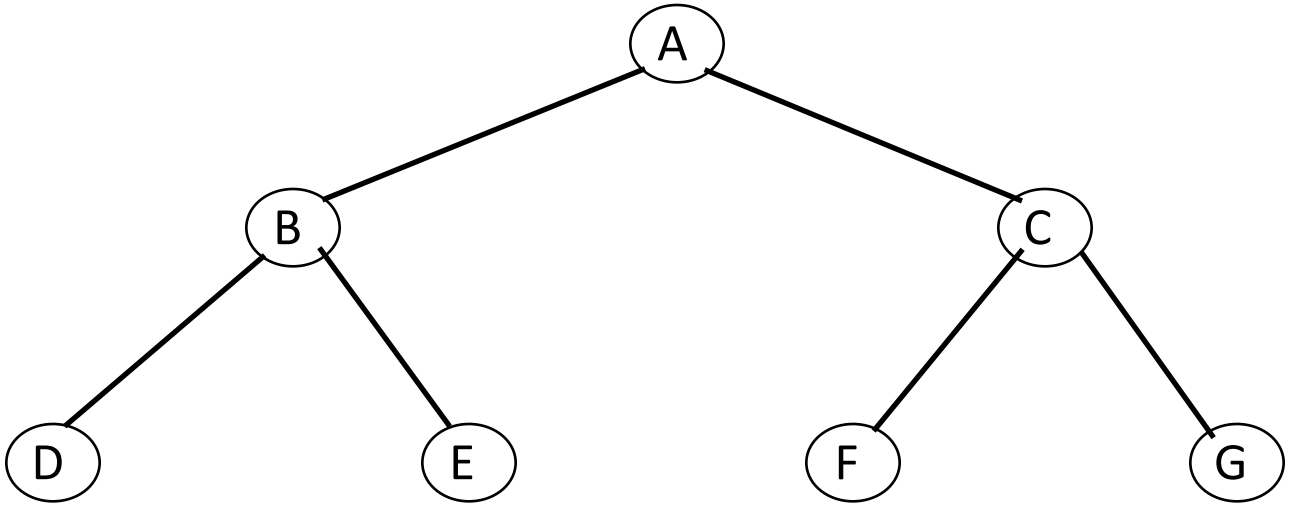


Queue:
G
F
E

Current:
D

A B C D

Breadth-First Search

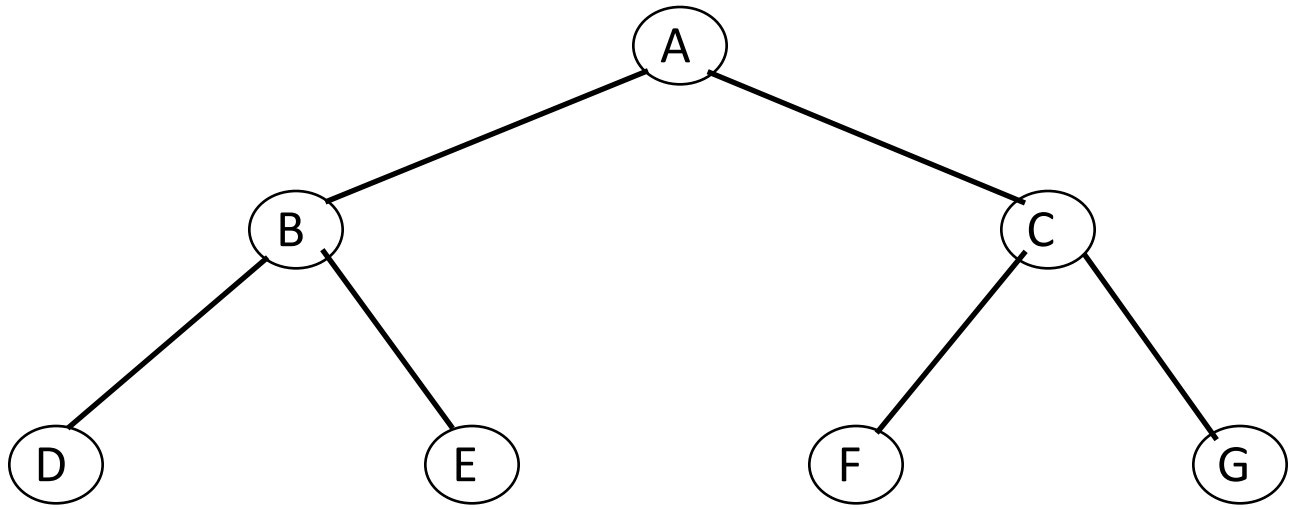


Queue:
G
F
E

Current:
E

A B C D

Breadth-First Search

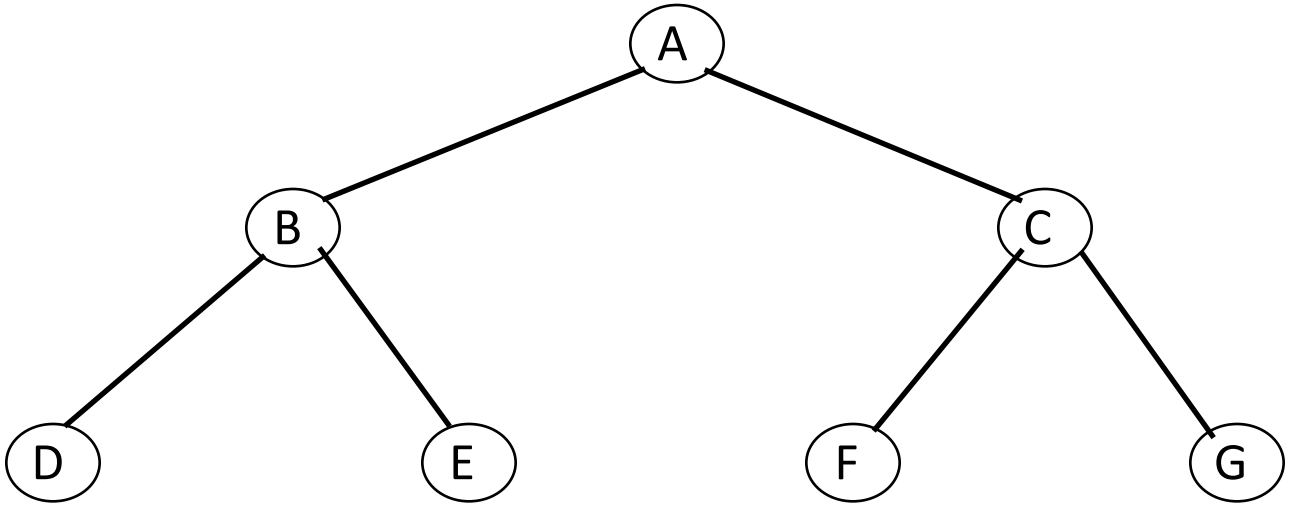


Queue:
G
F

Current:
E

A B C D E

Breadth-First Search

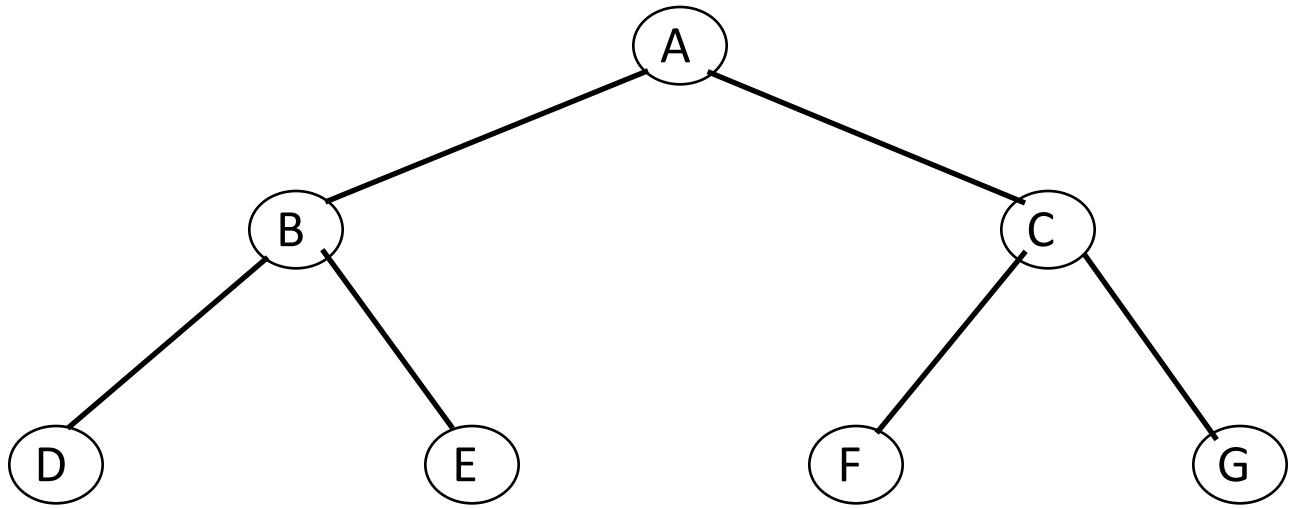


Queue:
G
F

Current:
F

A B C D E

Breadth-First Search

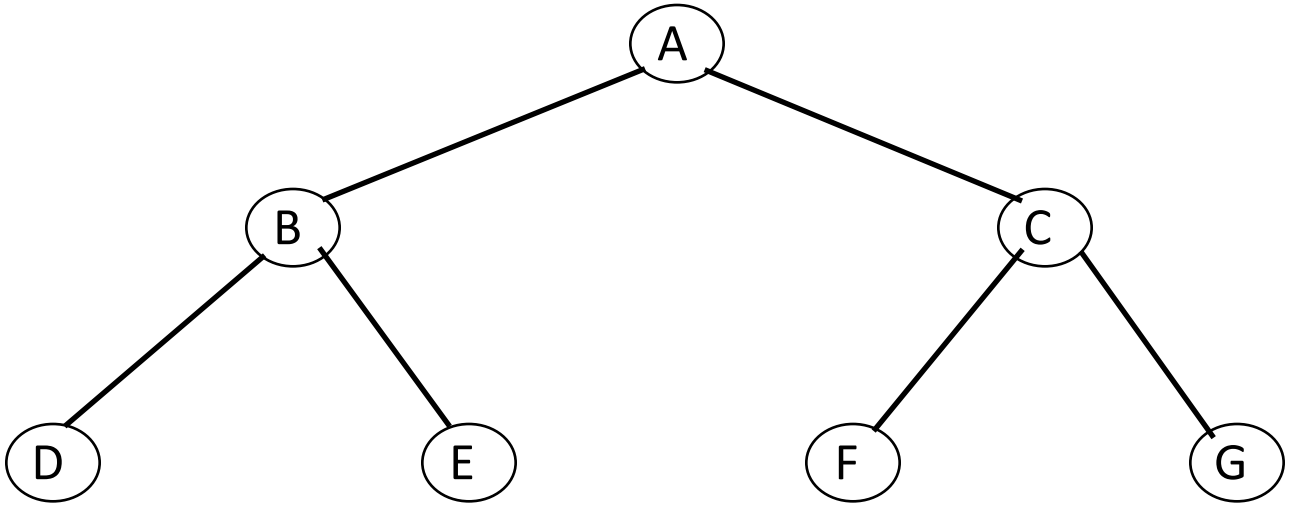


Queue:
G

Current:
F

A B C D E F

Breadth-First Search

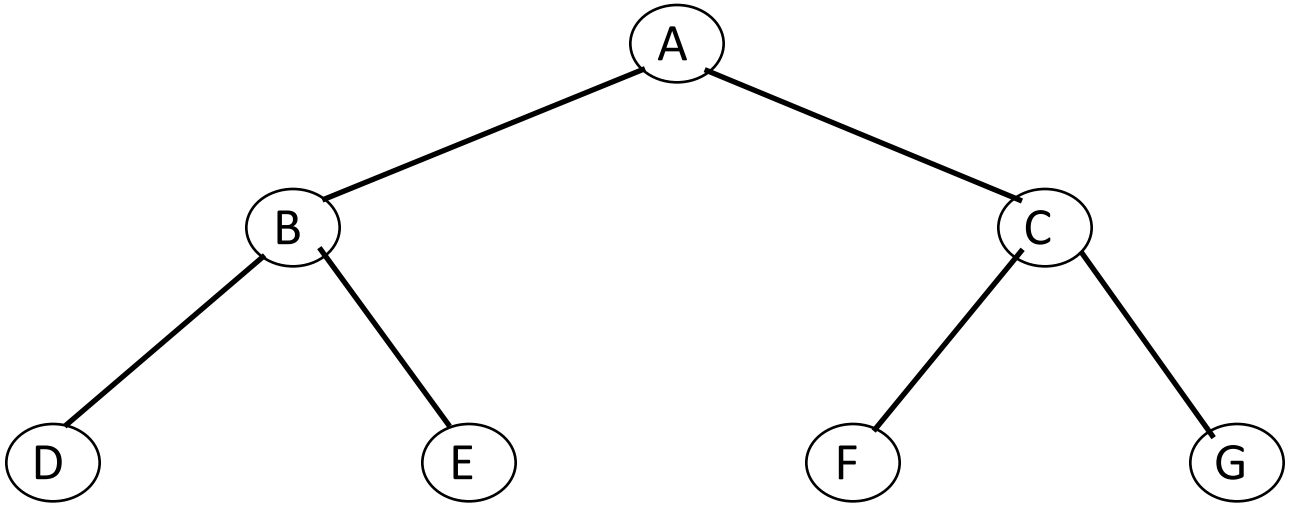


Queue:
G

Current:
G

A B C D E F

Breadth-First Search

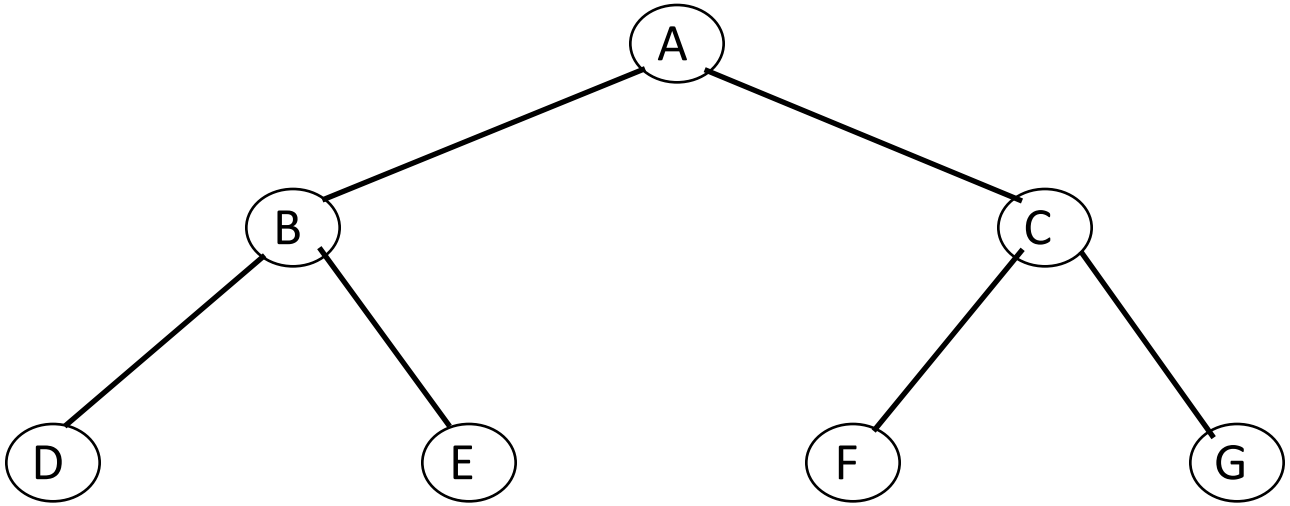


Queue:

Current:
G

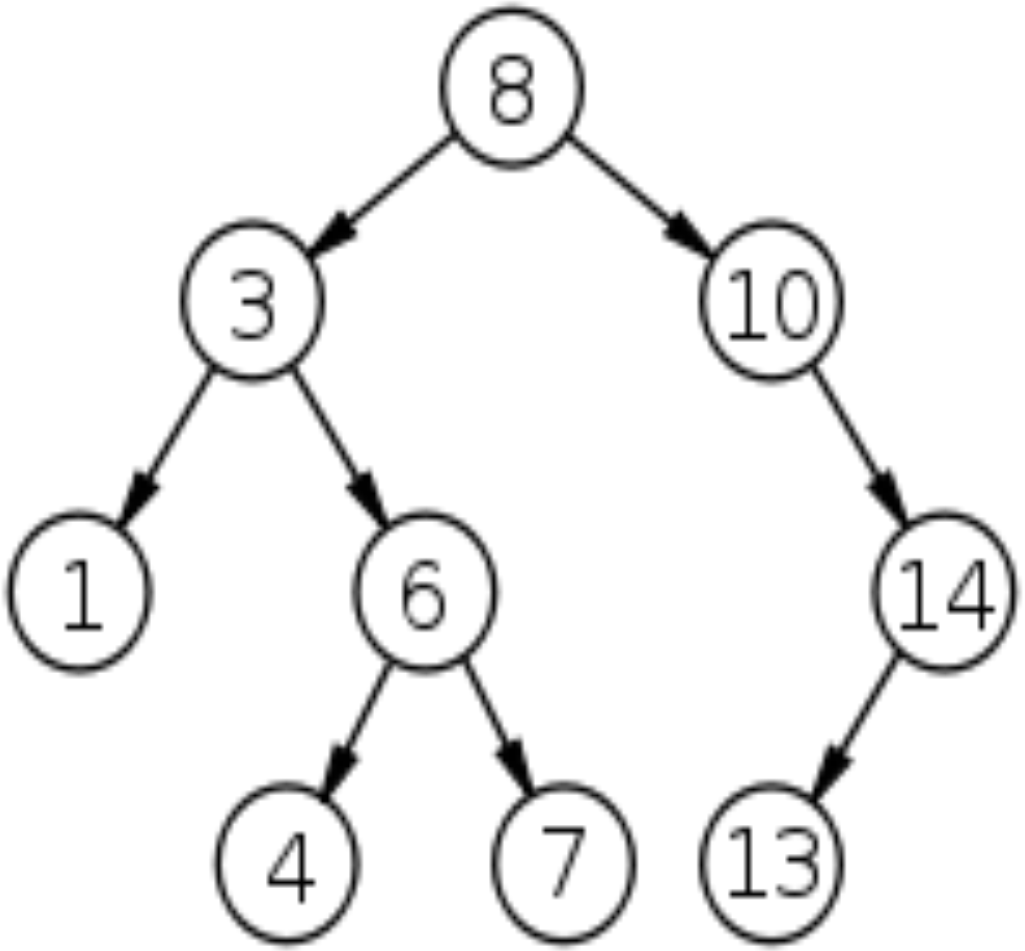
A B C D E F G

Breadth-First Search



A B C D E F G

Based On	Breadth First Search (BFS)	Depth First Search (DFS)
Description Of The Algorithm	**Breadth first search** *(aka. BFS)* is a searching method used to search *(or. explore)* for a node*(or the entire structure)* by traversing from root node and explore the search in **level by level.**	**Depth first search** (aka. DFS) is a searching method used to search (or explore) for a node (or the entire structure) by traversing from root node and explore the search as **deep** as possible until a dead end is found.
Data Structure Used	**Breadth first search** uses a **Queue**.	**Depth first search** uses a **Stack.**
Similar To	Breadth first search is similar to **level-order** traversal.	Depth first search is similar to **pre-order** traversal.
Speed	BFS is **slower** than DFS.	DFS is more **faster** than BFS**.**
Advantage	**Breadth first search** always finds a*shortest path* from the start vertex to any other for*unweighted graphs.*	**Depth first search** may not finds a shortest path.
Algorithm Type	A **greedy algorithm**. *(update neighbors of closest nodes first).*	A **backtracking algorithm** *(explore one path until dead end).*
Time & Space Complexity	**BFS** takes $O(b^d)$ time and $O(b^d)$space	**DFS** takes $O(b^h)$ time and $O(h)$ space
Applications **Traversal Example**	•Finding the **shortest path**. •Testing for ***bipartiteness***. •In **spanning tree**. •Web crawler.	•**Topological sorting**. •Finding the **bridges** of a graph. •Maze generation. •Finding strongly **connected components.**



BFS

8, 3, 10, 1, 6, 14, 4, 7, 3

DFS

3, 1, 6, 4, 7, 10, 14, 13

Example :BFS and DFS

Operations on Binary tree

- Inserting an element.
- Removing an element.
- Searching for an element.
- Deletion for an element.
- Traversing an element.

Applications of Binary Tree

- In compilers, Expression Trees are used which is an application of binary trees.
- Huffman coding trees are used in data compression algorithms.
- Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
- Represent hierarchical data.
- Used in editing software like Microsoft Excel and spreadsheets.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Syntax trees are used for most famous compilers for programming like GCC, and AOCL to perform arithmetic operations.
- For implementing priority queues.
- Used to find elements in less time (binary search tree)

Applications of Binary Tree

- Used to enable fast memory allocation in computers.
- Used to perform encoding and decoding operations.
- Binary trees can be used to organize and retrieve information from large datasets, such as in inverted index and k-d trees.
- Binary trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
- Binary trees can be used to implement searching algorithms, such as in binary search trees which can be used to quickly find an element in a sorted list.
- Binary trees can be used to implement sorting algorithms, such as in heap sort which uses a binary heap to sort elements efficiently.

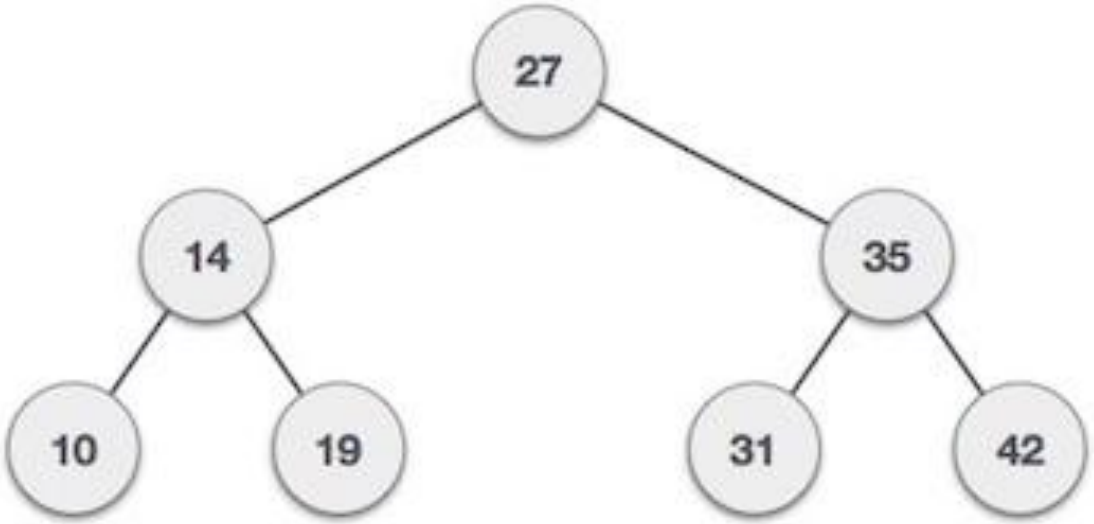
Binary Search Tree (BST)

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than to its parent node's key.

- BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −
  - **left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)**

pictorial representation of BST –

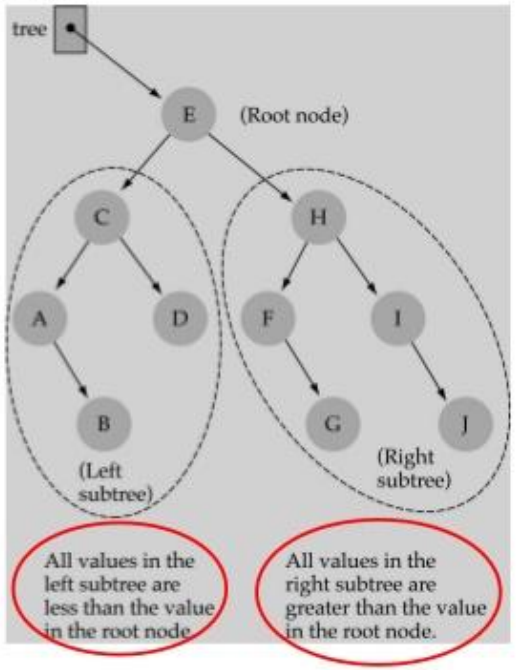
root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.



BST

Binary Search Trees

- **Binary Search Tree Property**:
  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



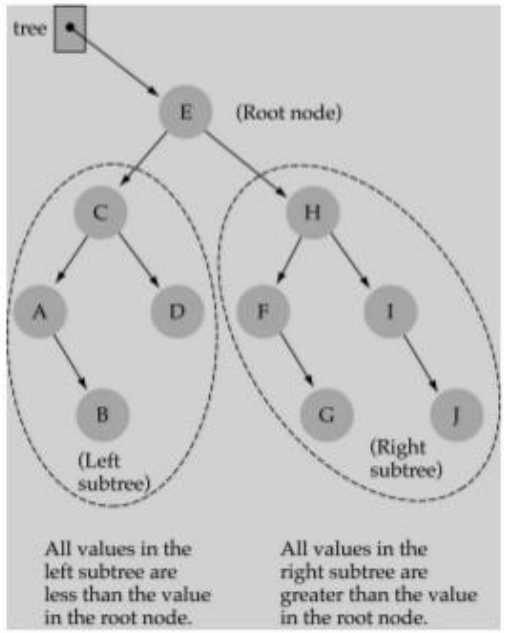
Binary Search Trees

Where is the smallest element?

Ans: leftmost element

Where is the largest element?

Ans: rightmost element



Advantages of using binary search tree

- Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
- It also speed up the insertion and deletion operations as compare to that in array and linked list

BST operations

- Following are the basic operations of a tree −
- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Delete** -deletes a node from the tree
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

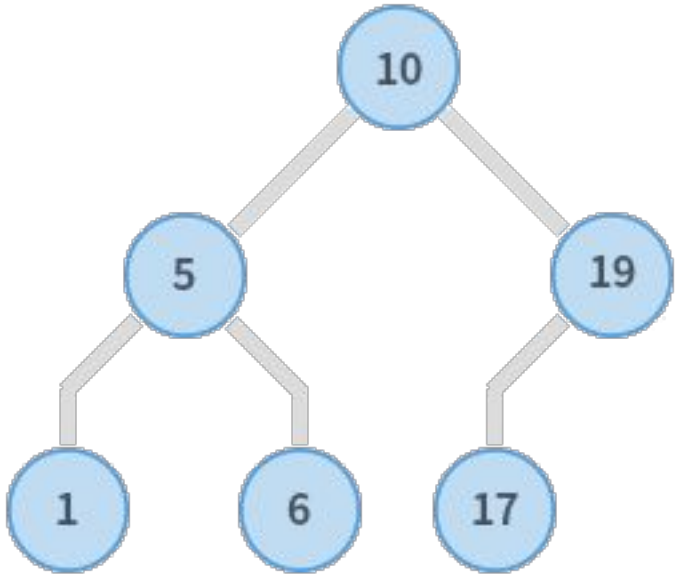


Fig. 1

In Fig. 1, consider the root node with data = 10.

- Data in the left subtree is: [5,1,6]
- All data elements are < 10

- Data in the right subtree is: [19,17]
- All data elements are > 10

In-order traversal

- In in-order traversal, do the following:
- First process left subtree (before processing root node)
- Then, process current root node
- Process right subtree

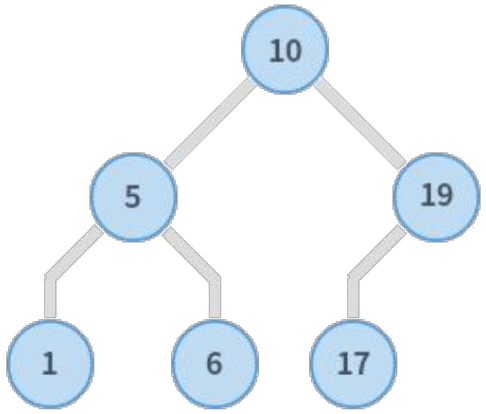


Fig. 1

The function call **stack** is as follows:

inorder(1) completed

popped from the call stack.

```
void inorder(struct node*root)
  {
     if(root)
     {
        inorder(root->left);    //Go to left subtree
        printf("%d ",root->data);    //Printf root->data
        inorder(root->right);     //Go to right subtree
     }
  }
```

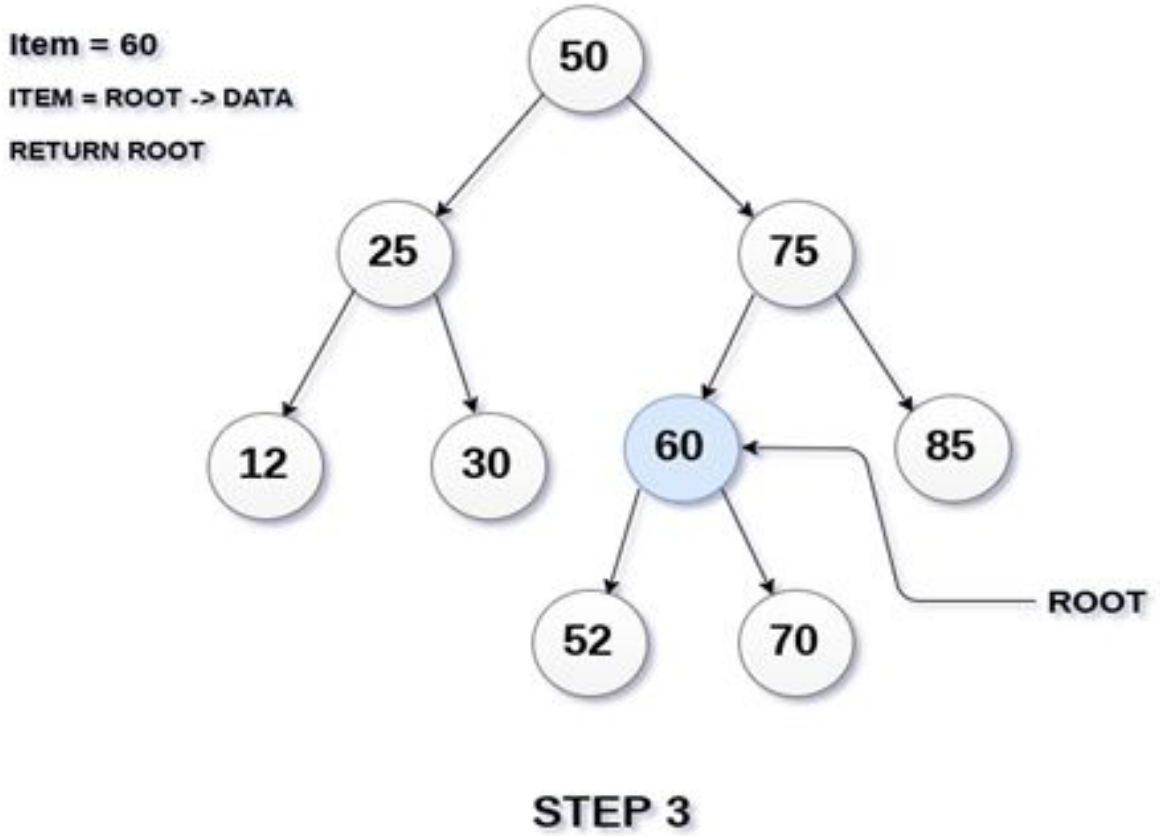
- The order in which BST in Fig. 1 is visited is: **1, 5, 6, 10, 17, 19.**
- The in-order traversal of a BST gives a sorted ordering of the data elements that are present in the BST.

Searching :

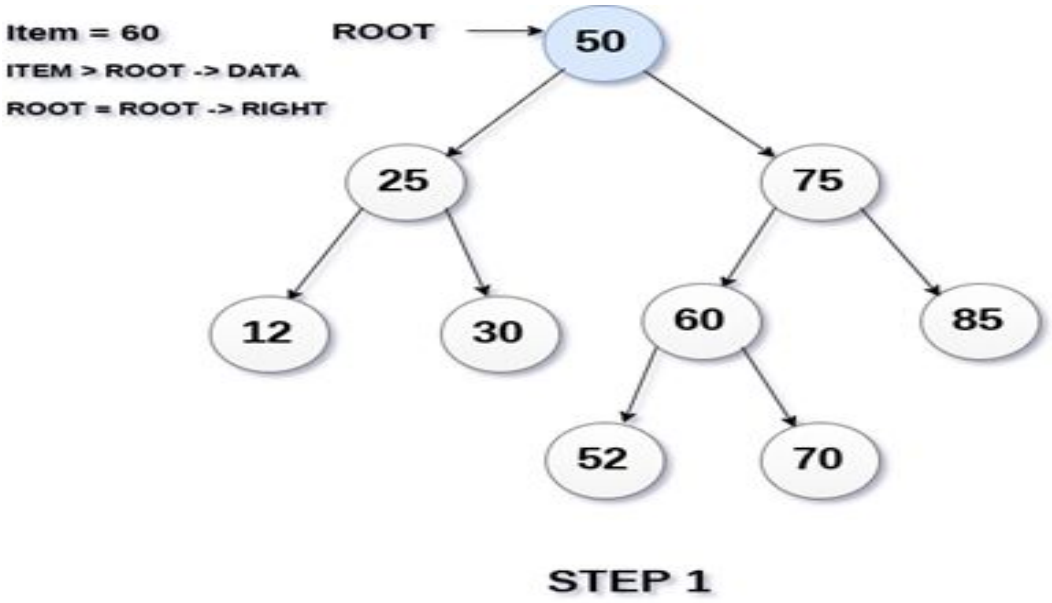
Searching means finding or locating some specific element or node within a data structure.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.

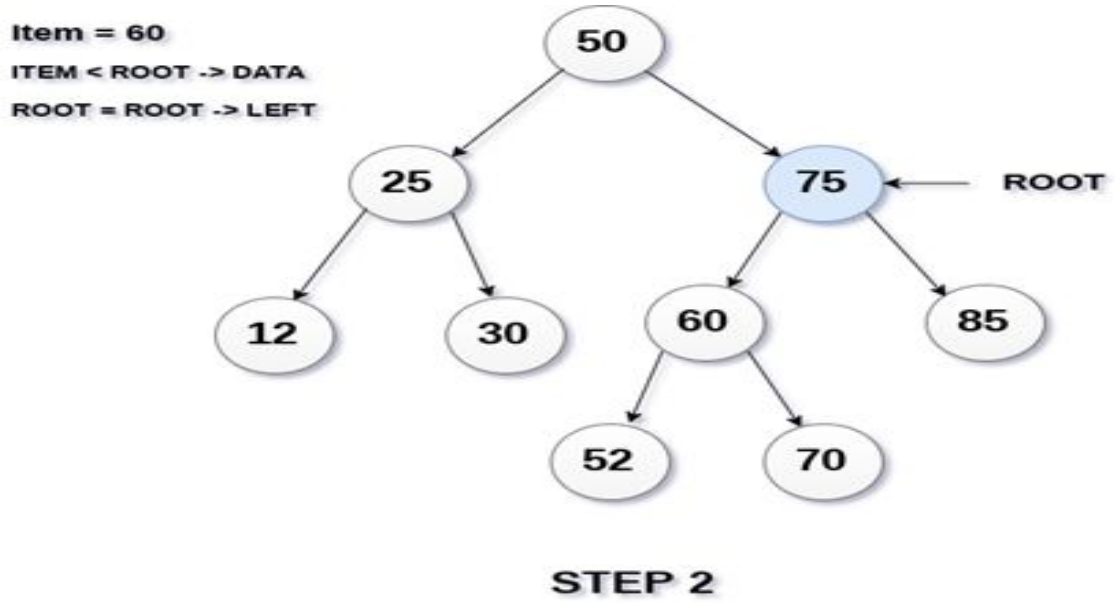
Searching – step by step



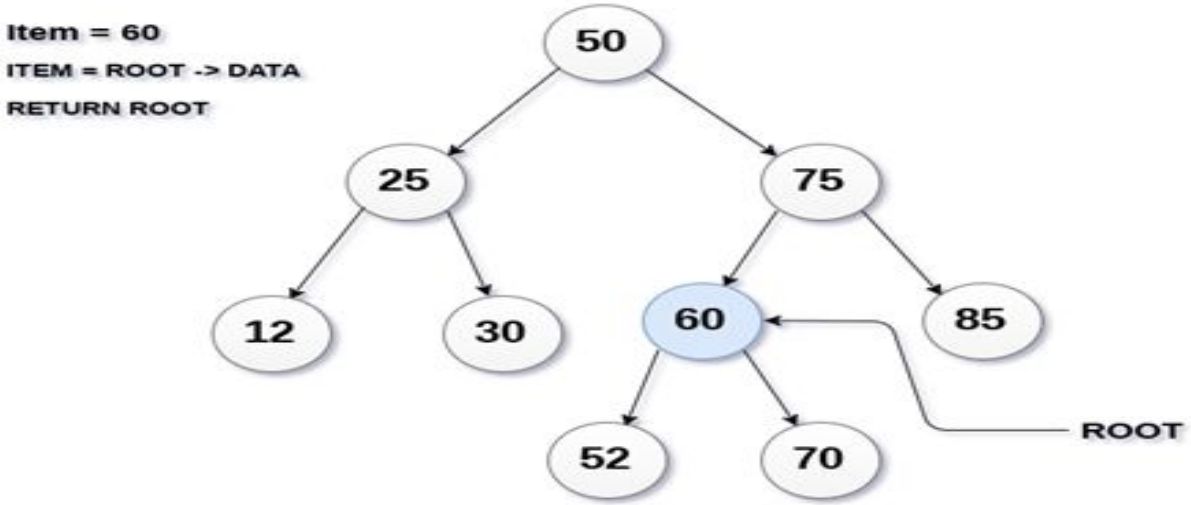
Searching: complete view



STEP 1



STEP 2



STEP 3

Search Algorithm:

- **Search (ROOT, ITEM)**
- **Step 1:** IF ROOT -> DATA = ITEM OR ROOT = NULL
  Return ROOT
  ELSE
  IF ITEM < ROOT -> DATA
  Return search(ROOT -> LEFT, ITEM)
  ELSE
  Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]
- **Step 2:** END

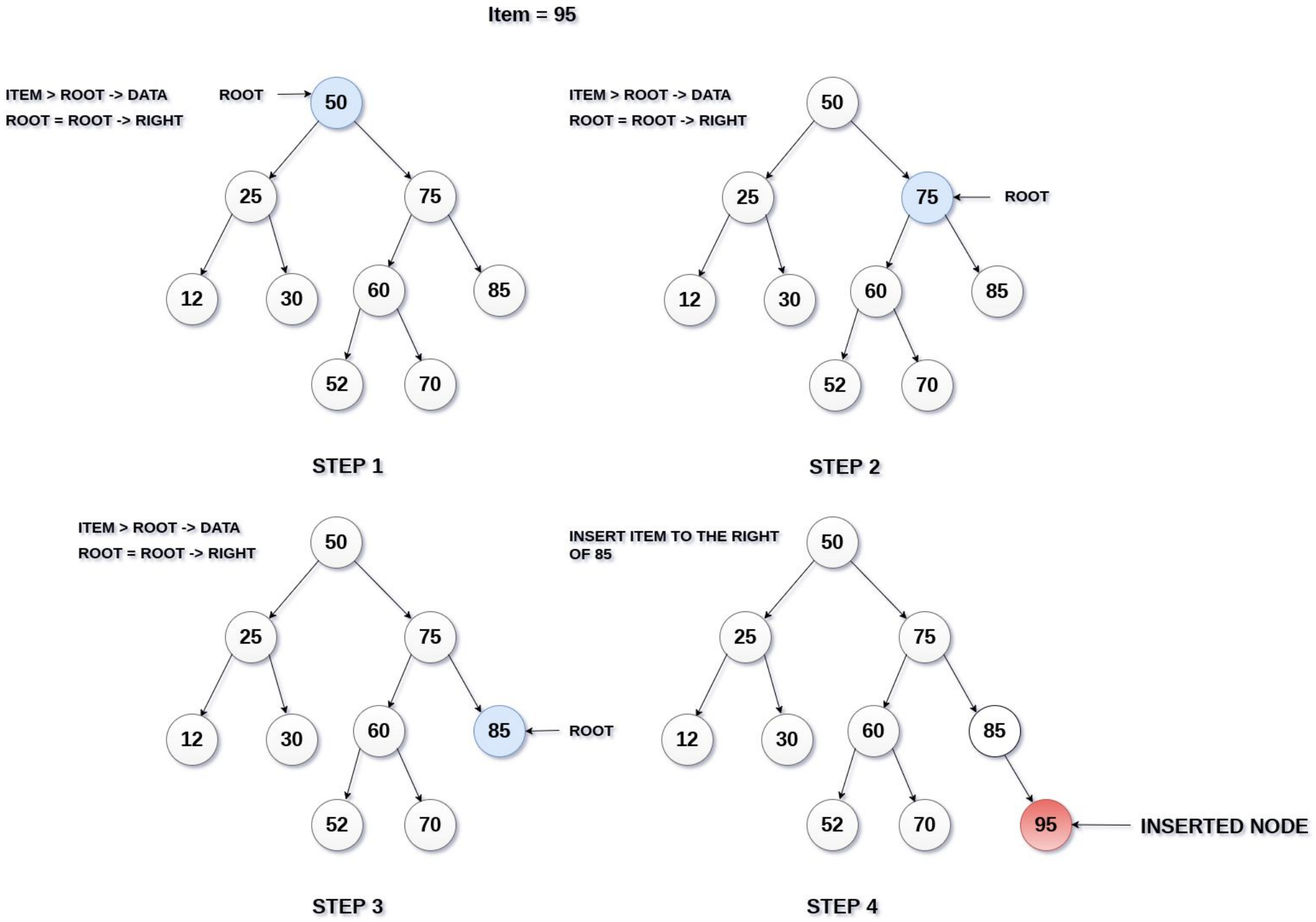
```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node
   endwhile

   return data not found

end if
```

Insertion

- Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert


Item = 95
ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT
ROOT
50
25
75
12
30
60
85
52
70
STEP 1
ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT
50
25
75
ROOT
12
30
60
85
52
70
STEP 2
ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT
50
25
75
12
30
60
85
ROOT
52
70
STEP 3
INSERT ITEM TO THE RIGHT
OF 85
50
25
75
12
30
60
85
52
70
95
INSERTED NODE
STEP 4

Insert Algorithm

- Insert (TREE, ITEM)
- **Step 1:** IF TREE = NULL
  Allocate memory for TREE
  SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  Insert(TREE -> LEFT, ITEM)
  ELSE
  Insert(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
- **Step 2:** END

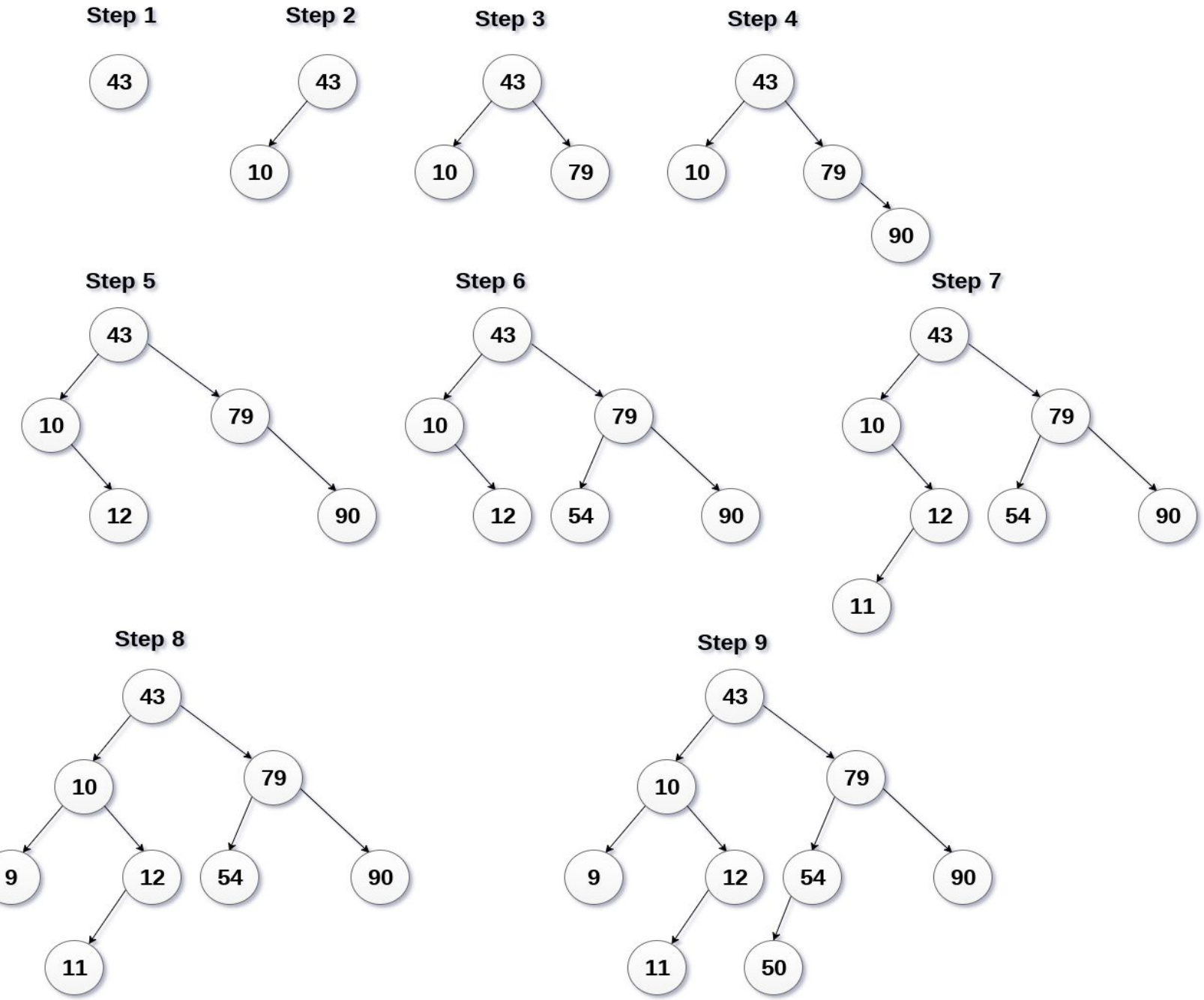
```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile

   insert data

end If
```

Q. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

- Insert 43 into the tree as the root of the tree.
- Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
- Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements



Deletion

- Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

- The node to be deleted is a leaf node
- The node to be deleted has only one child
- The node to be deleted has two children

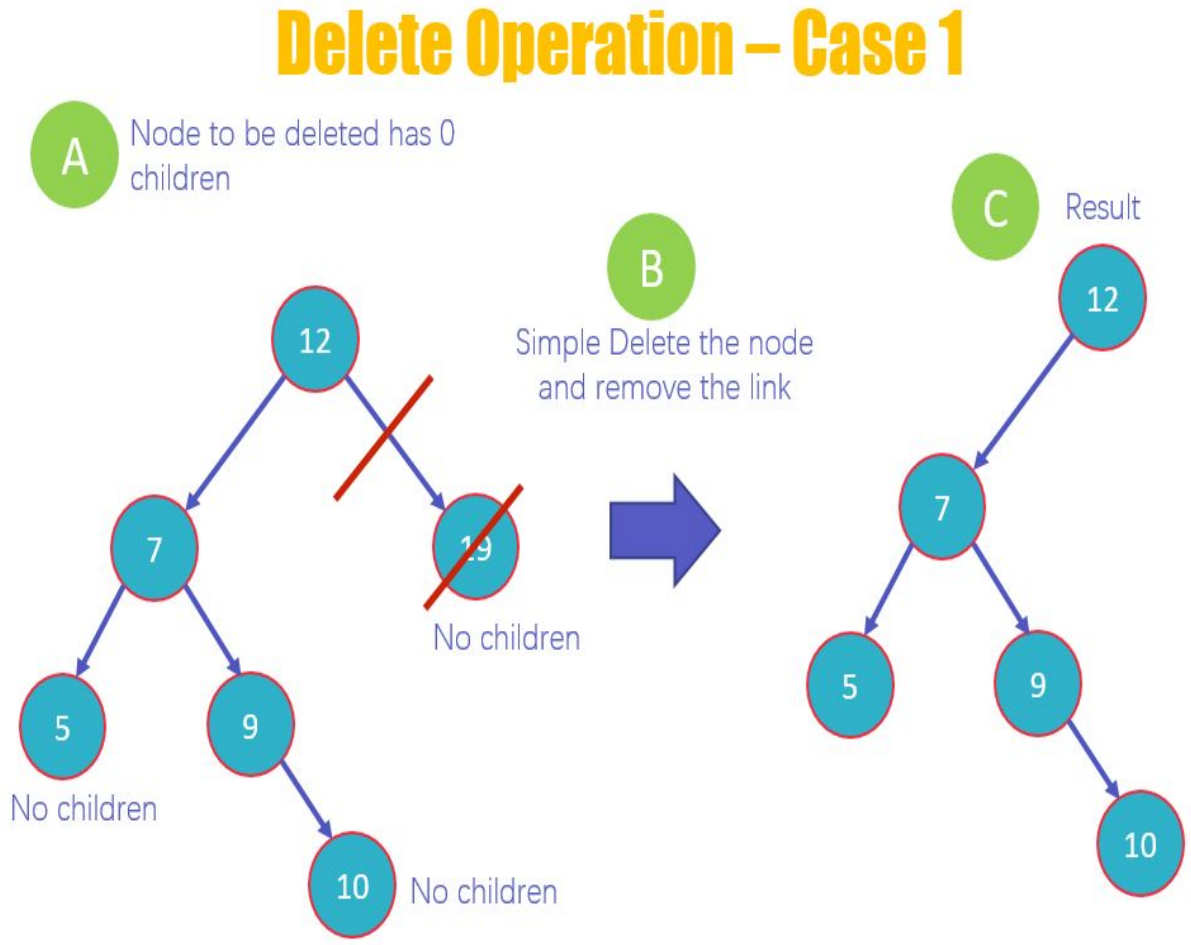
Deletion

- **Case 1-** Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.
- **Case 2 –** Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.
- **Case 3 -**Node with two children: this is the most difficult situation, and it works on the following two rules

3a – **In Order Predecessor:** you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node

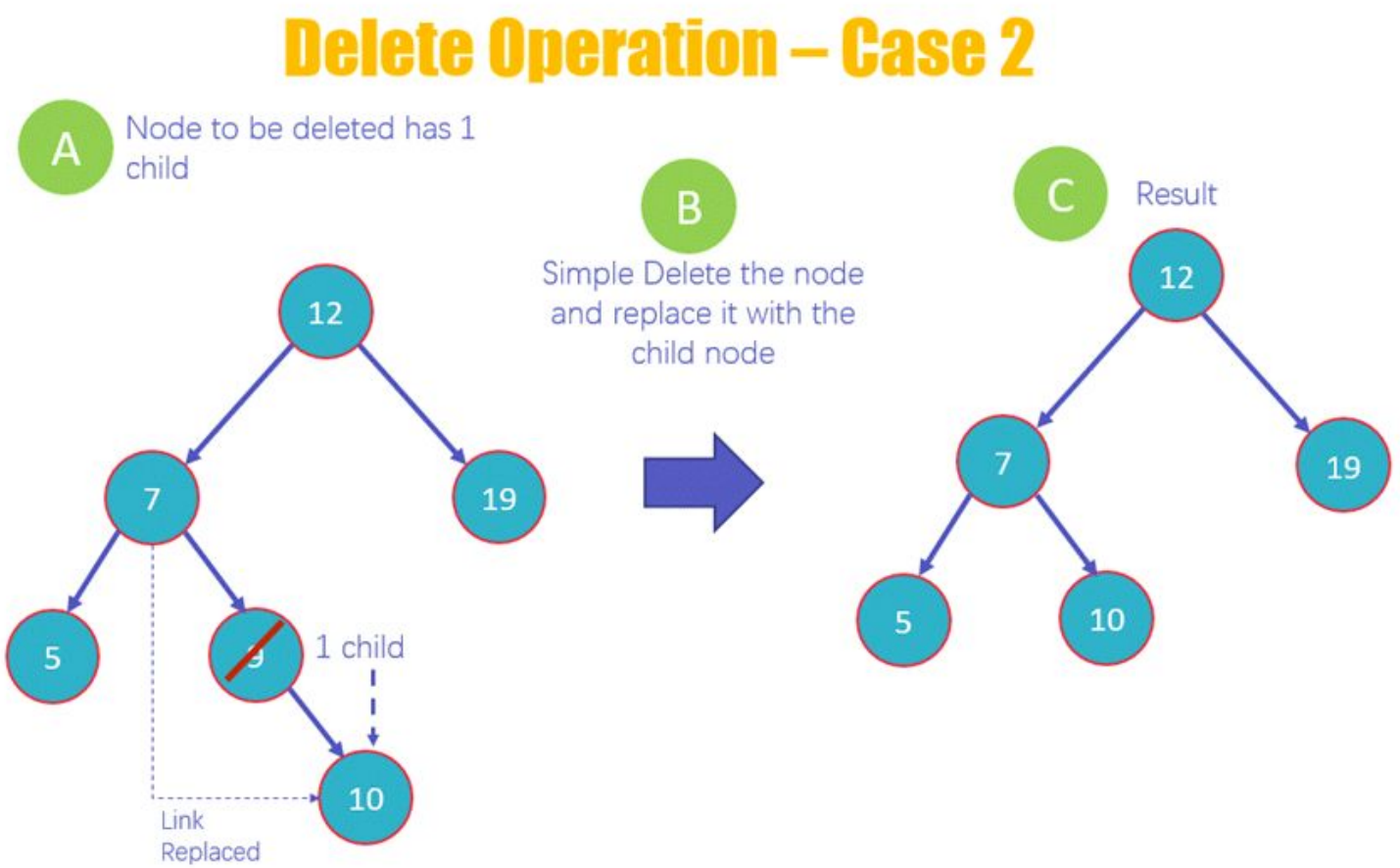
3b – In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node

Node with zero children



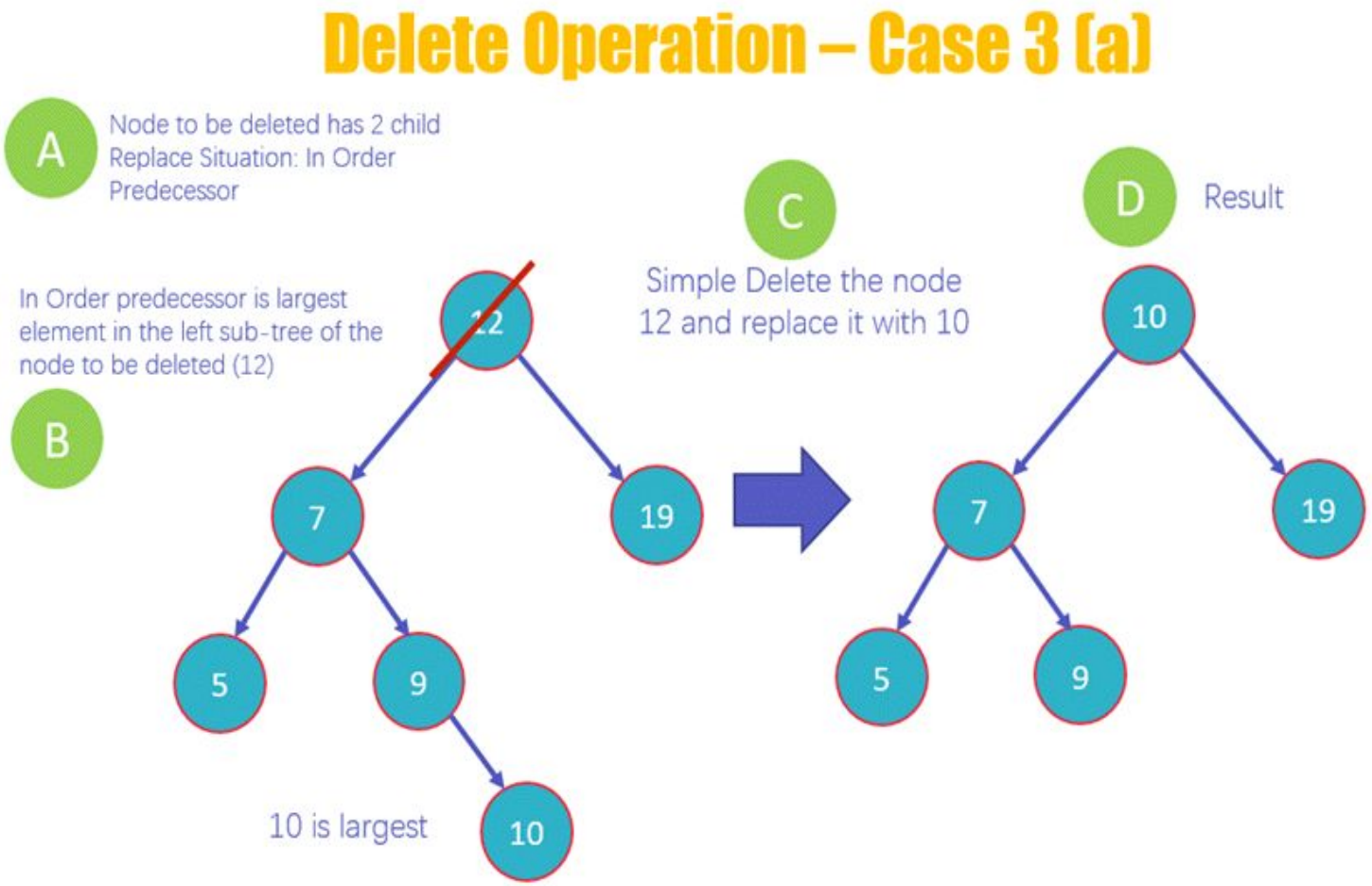
1. In the diagram **19, 10 and 5** have no children. But we will delete 19.
2. Delete the value 19 and remove the link from the node.
3. View the new structure of the BST without 19

Node with one child:



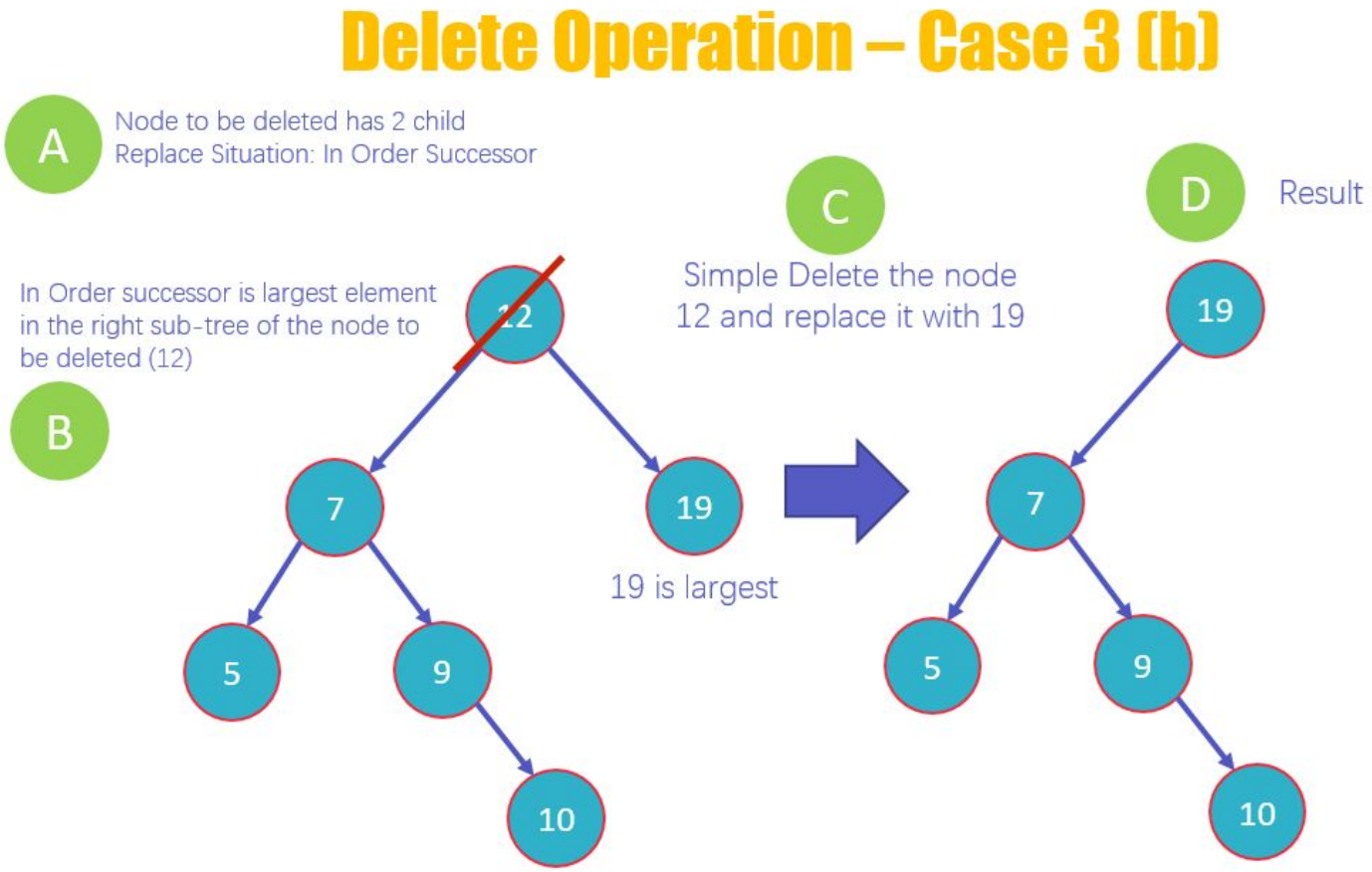
1. In the diagram that 9 has one child.
2. Delete the node 9 and replace it with its child 10 and add a link from 7 to 10
3. View the new structure of the BST without 9

Node with two children



1.deleting the node 12 that has two children
2.The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.
3.Delete the node 12 and replace it with 10 as it is the largest value on the left subtree
4.View the new structure of the BST after deleting 12

Node with two children



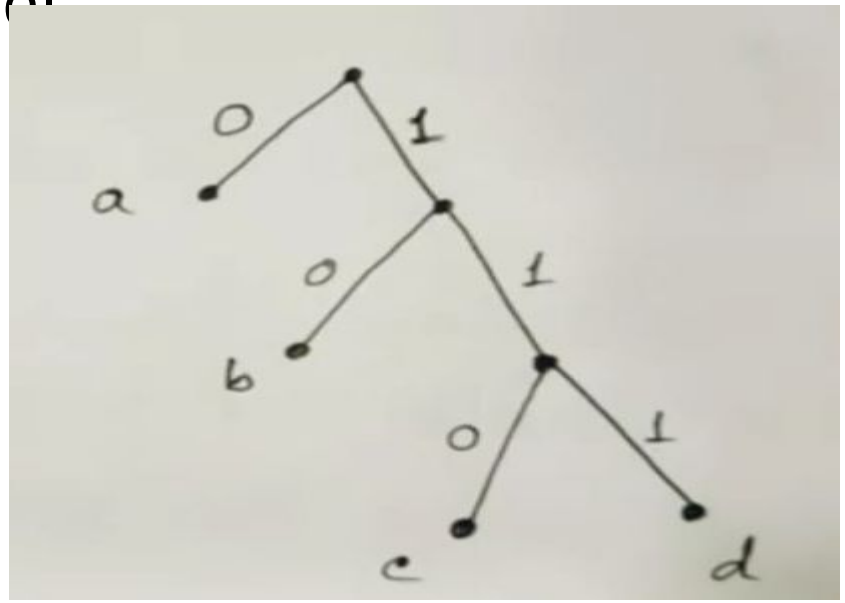
1 Delete a node 12 that has two children
2 The deletion of the node will occur based upon the In Order Successor rule, which means that the largest element on the right subtree of 12 will replace it
3 Delete the node 12 and replace it with 19 as it is the largest value on the right subtree
4 View the new structure of the BST after deleting 12

Pseudo Code for Deleting a Node:

```
delete (value, root):
    Node x = root
    Node y = NULL
# searching the node
    while x:
        y = x
    if x.value < value
        x = x.right
    else if x.value > value
        x = x.left
    else if value == x
        break
# if the node is not null, then replace it with successor
    if y.left or y.right:
        newNode = GetInOrderSuccessor(y)
        root.value = newNode.value
# After copying the value of successor to the root #we're deleting the successor
        free(newNode)
    else
        free(y)
```

Huffman Tree- Prefix codes

- A "prefix code" is a type of encoding mechanism ("code").
- **prefix code**, variable length the entire set of possible encoded values ("codewords") must not contain *any* values that start with any *other* value in the set.
- a=0
- b=10
- c-=110
- D=111



Huffman coding

- Huffman coding is a lossless data compression algorithm
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
- The variable-length codes assigned to input characters are Prefix Codes
- To construct optimal prefix code is called **Huffman coding**.

Huffman coding

- ***Steps to build Huffman Tree***

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps 2 and 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

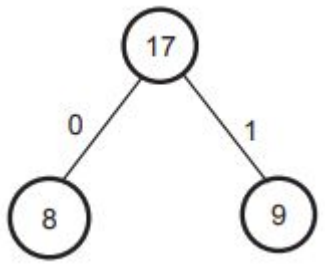
Huffman coding

8, 9, 12, 14, 16, 19.

Step 1 : Increasing order of weights :

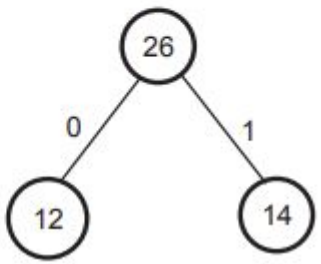
8, 9, 12, 14, 16, 19
↑ ↑

(↑ : First two smallest weights)

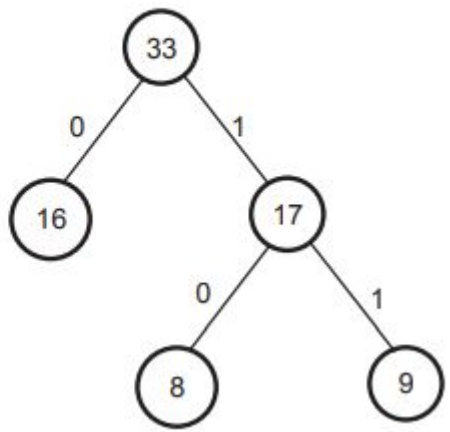


Step 2 : Rewrite sequence of weights in increasing order

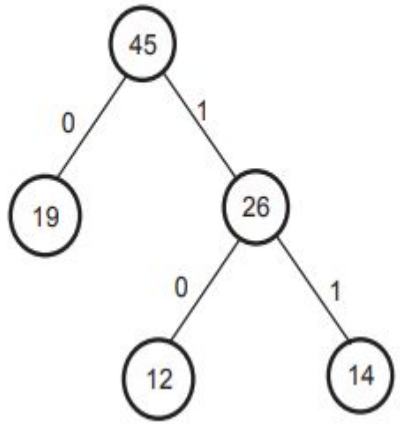
12, 14, 16, 17, 19
↑ ↑



Step 3 : Sequence : 16, 17, 19, 26,
↑ ↑

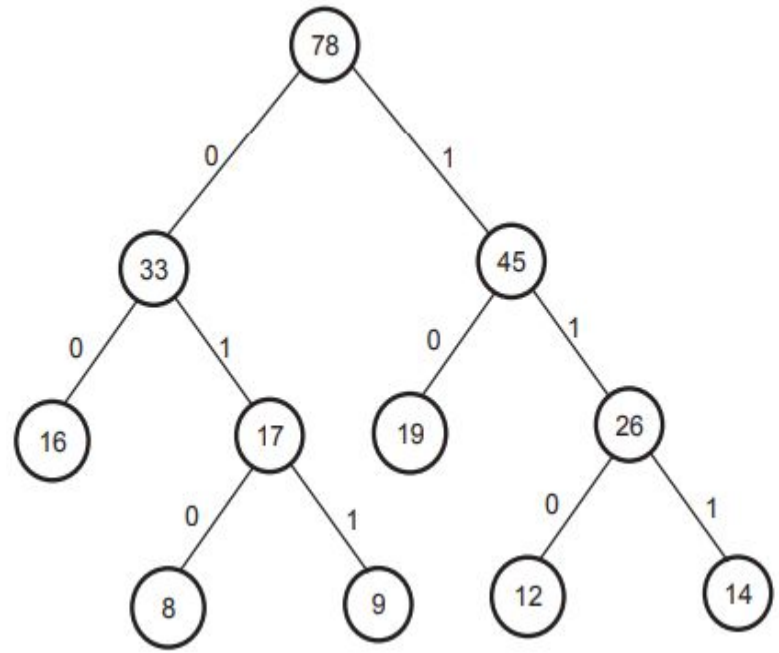


Step 4 : Sequence : 19, 26, 33,



Step 5 : Sequence : 33, 45

Symbols	Binary Prifix Codes
16	00
8	010
9	011
19	10
12	110
14	111



Example

Construct Huffman's Tree and the prefix free code for all characters :

Symbol	A	C	E	H	I
Frequency	3	5	8	2	7

Answer

Symbol	A	C	E	H	I
Prefix Code	011	00	11	010	00

Threaded binary tree- concepts

- Speeding up traversals

- In a binary search tree, there are many nodes that have an empty left child or empty right child or both.

- You can utilize these fields in such a way so that the empty left child of a node points to its in-order predecessor and empty right child of the node points to its in-order successor.

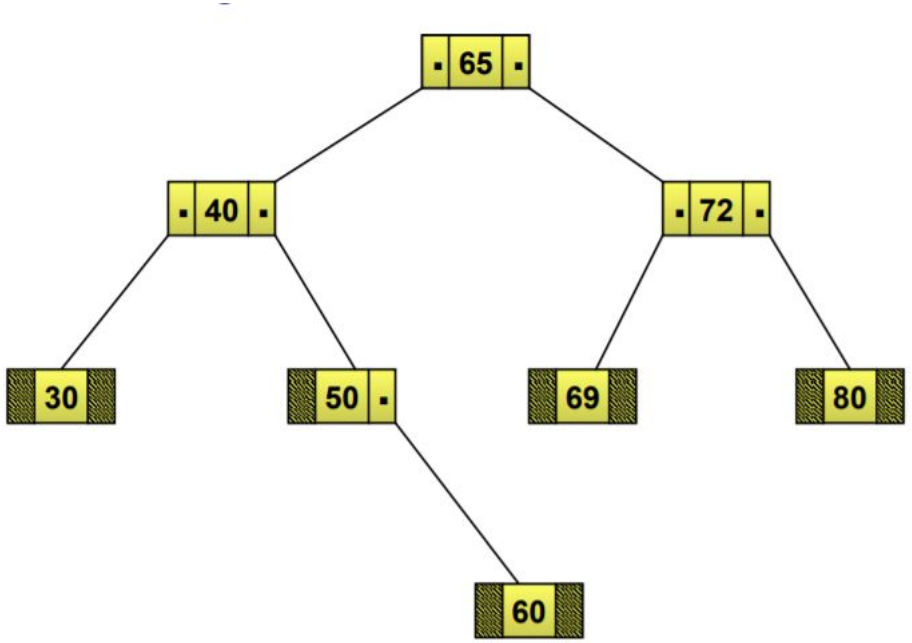
Threading

- One way threading:- A thread will appear in a right field of a node and will point to the next node in the in-order traversal.

- Two way threading:- A thread will also appear in the left field of a node and will point to the preceding node in the in-order traversal.

Threading

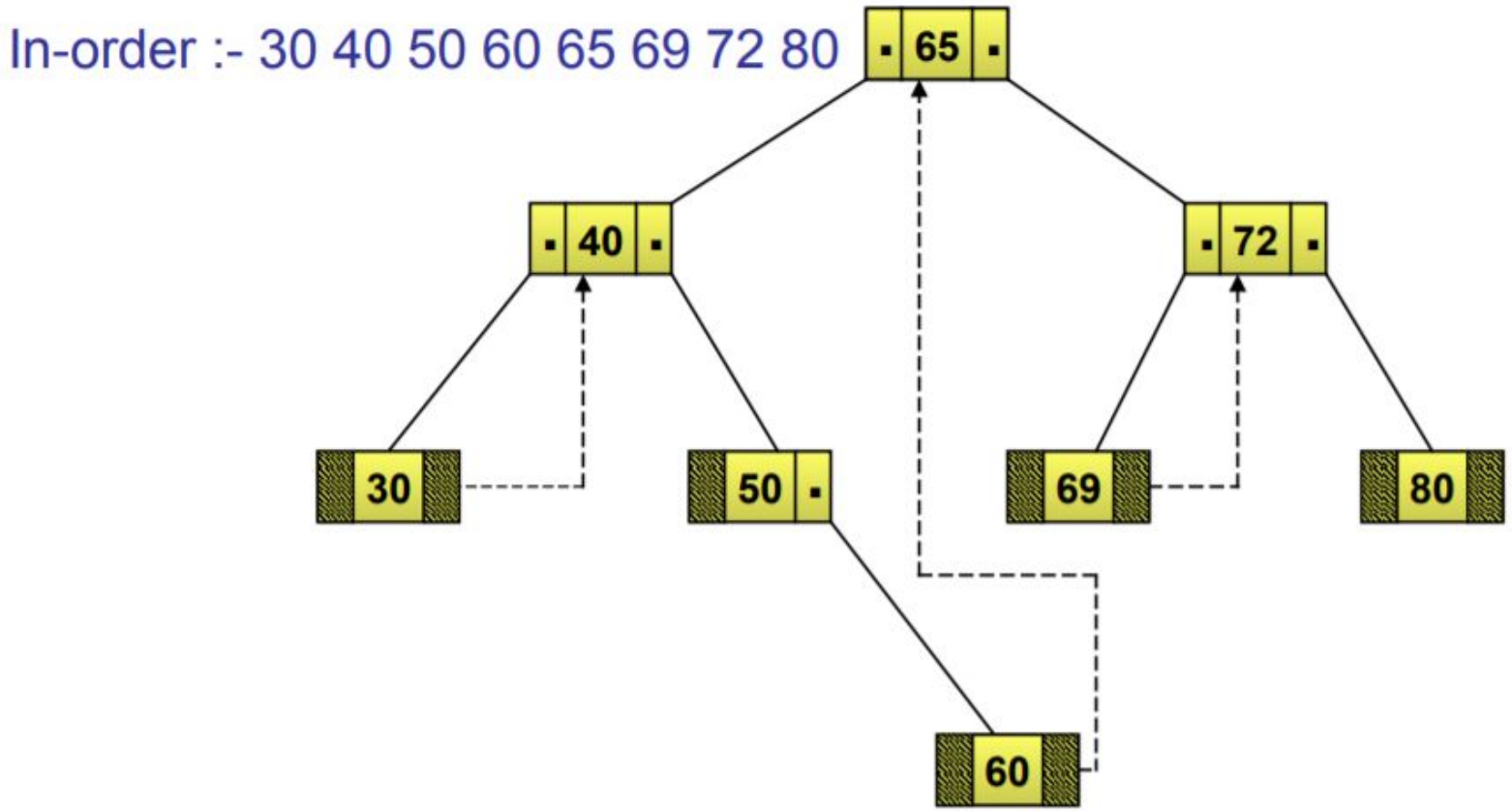
- Most of the nodes in this tree hold a NULL value in their left or right child fields. In this case, it would be good if these NULL fields are utilized for some other useful purpose.



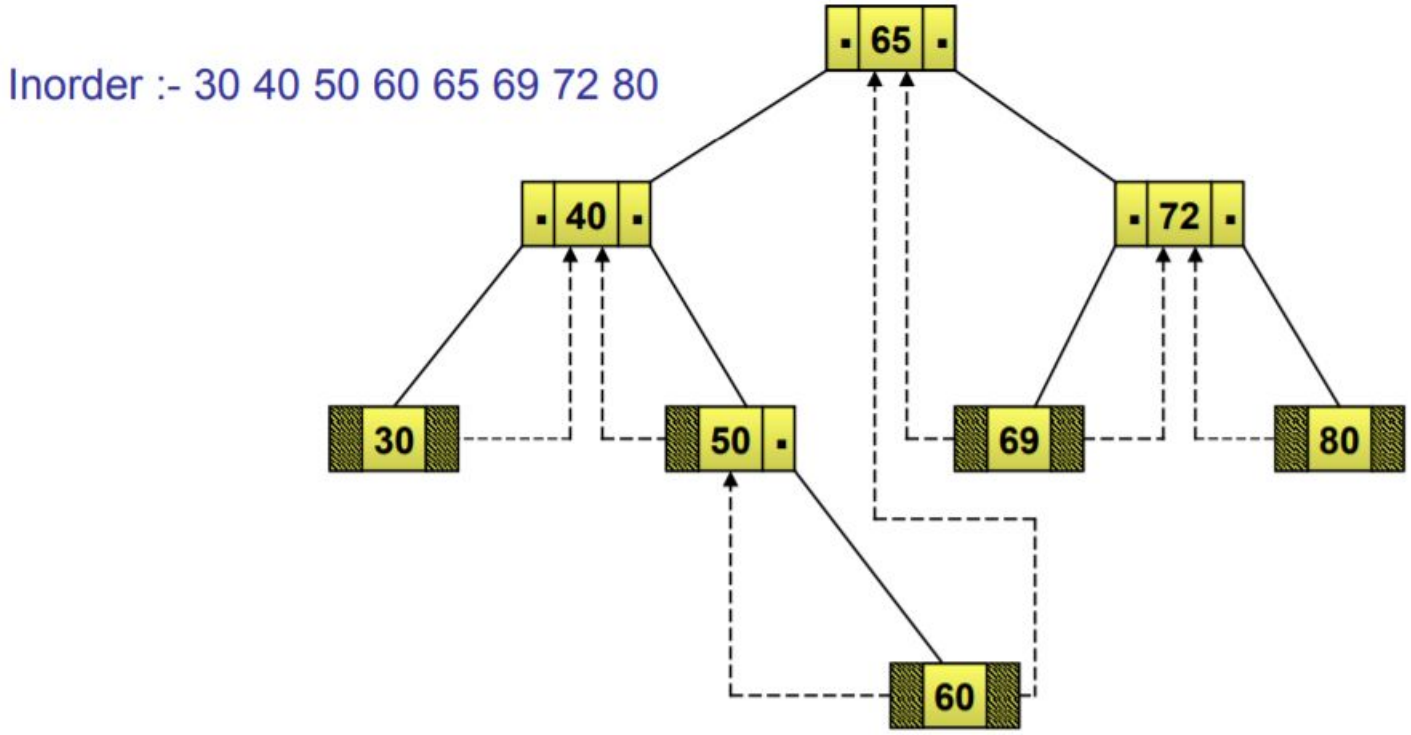
One Way Threaded Binary Trees

- The empty left child field of a node can be used to point to its in-order predecessor.
- Similarly, the empty right child field of a node can be used to point to its in-order successor.
- Such a type of binary tree is known as a one way threaded binary tree.
- A field that holds the address of its in-order successor is known as thread.
- In-order :- 30 40 50 60 65 69 72 80 .

One Way Threaded Binary Trees



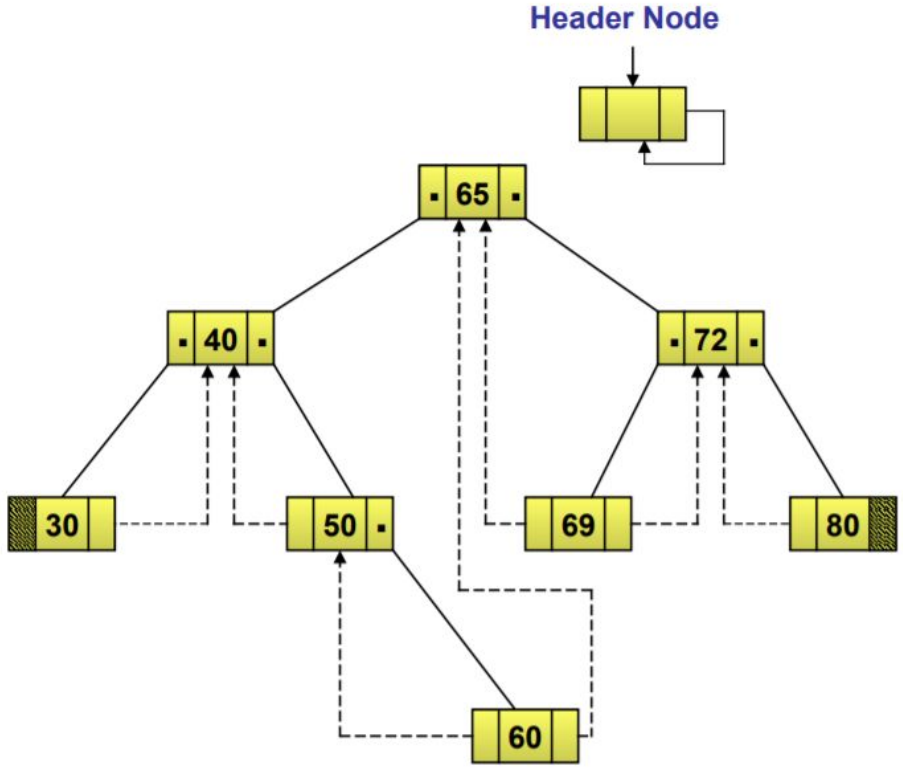
Two way Threaded Binary Trees



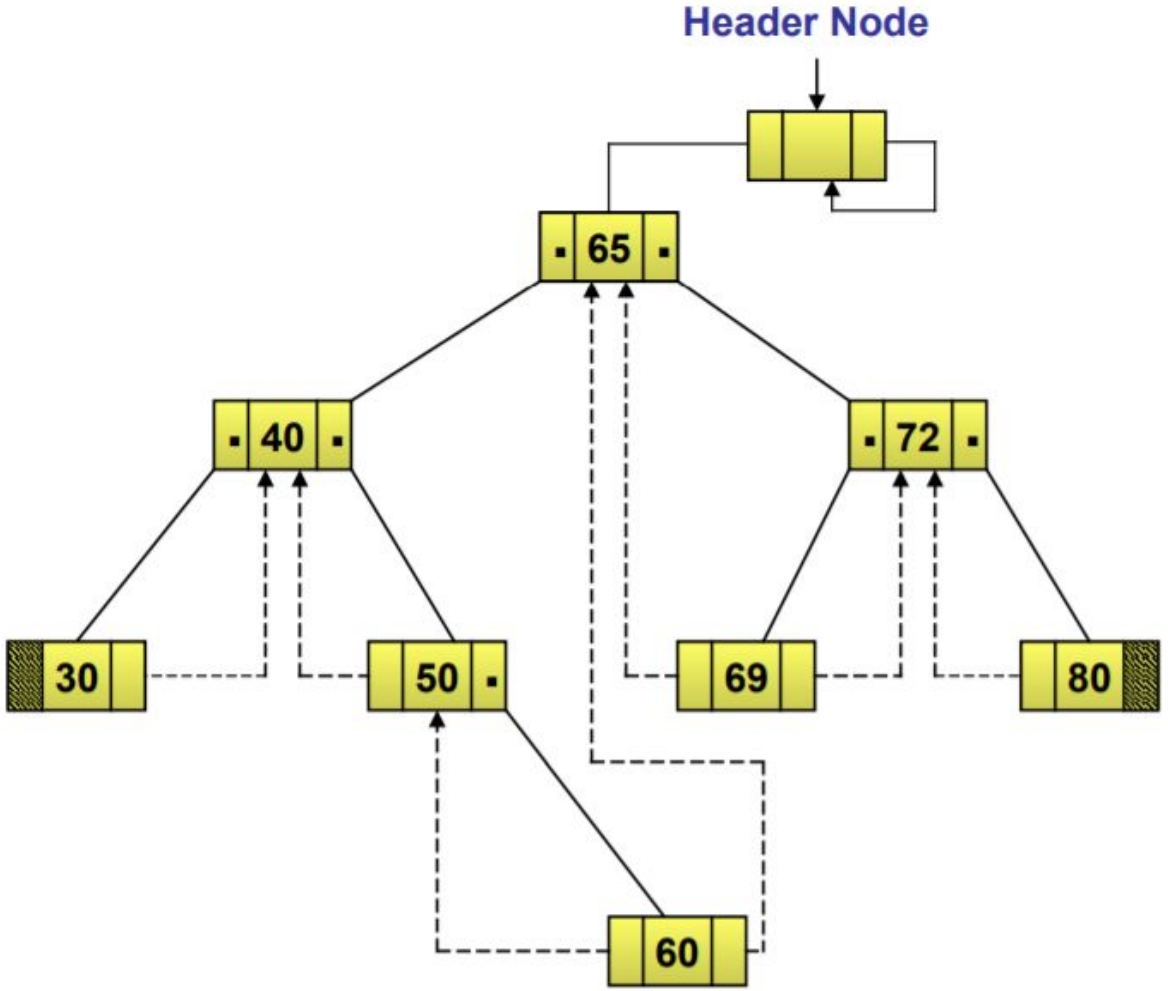
Node 30 does not have an in-order predecessor because it is the first node to be traversed in in-order sequence.
Similarly, node 80 does not have an in-order successor.

Two way Threaded Binary Trees with header Node

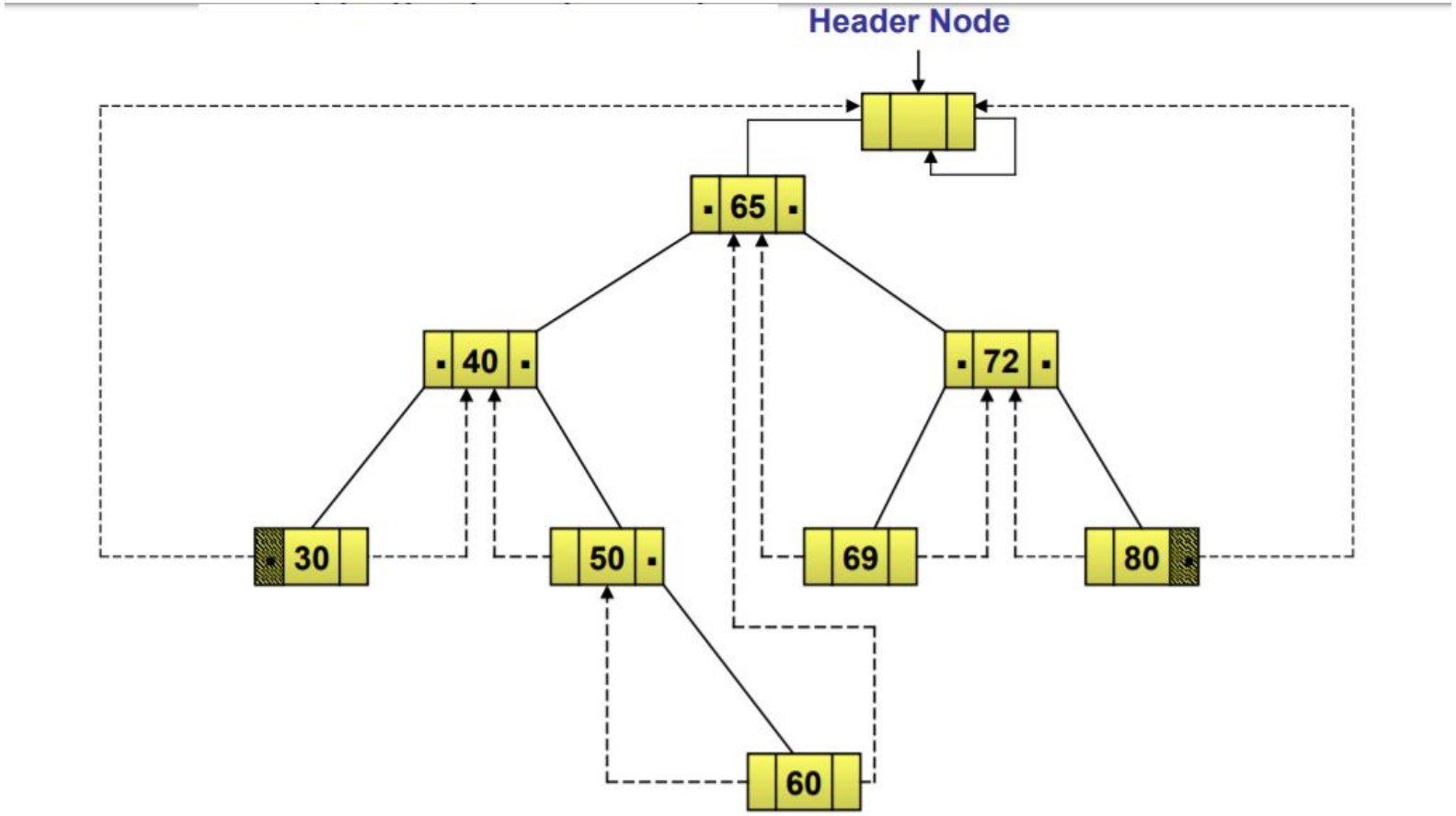
- The right child of the header node always points to itself.
- Therefore, you take a dummy node called the header node.



The threaded binary tree is represented as the left child of the header node.



- The left thread of node 30 and the right thread of node 80 point to the header node.



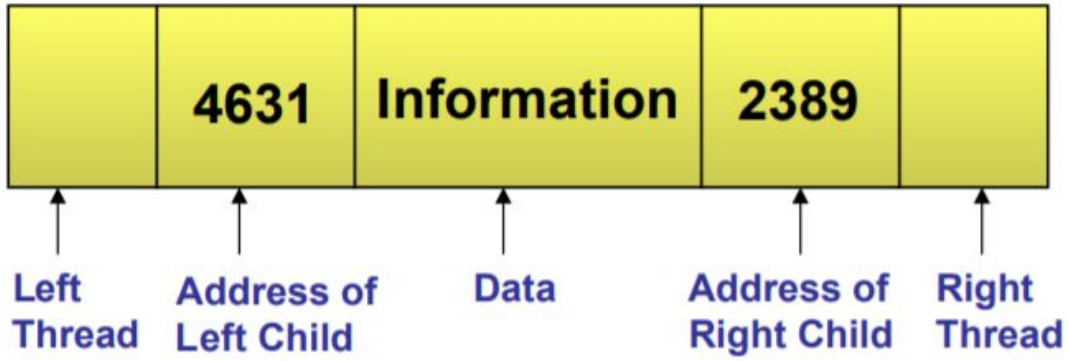
Representing a Threaded Binary Tree

The structure of a node in a threaded binary tree is a bit different from that of a normal binary tree.

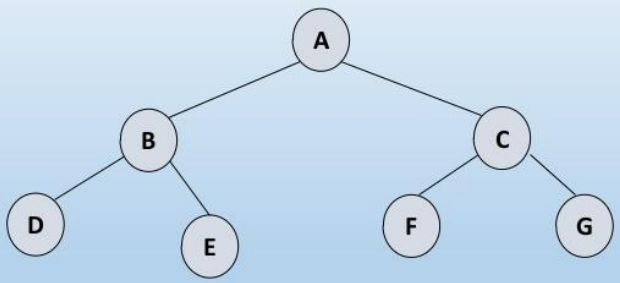
Unlike a normal binary tree, each node of a threaded binary tree contains two extra pieces of information, namely left thread and right thread.

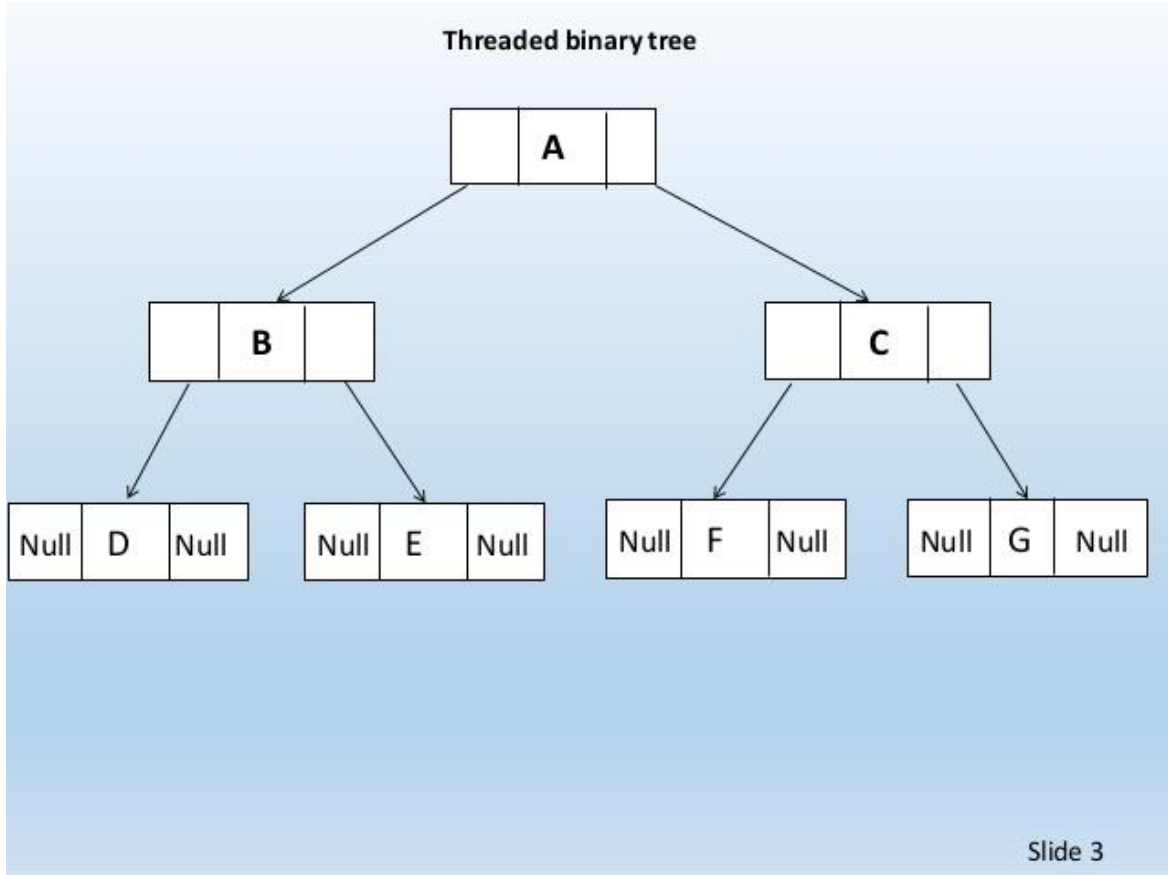
The left and right thread fields of a node can have two values:

- **1**: Indicates a normal link to the child node
- **0**: Indicates a thread pointing to the inorder predecessor or inorder successor



A Simple Binary Tree




Threaded binary tree
A
B
C
Null
D
Null
Null
E
Null
Null
F
Null
Null
G
Null
Slide 3

Threaded Binary Tree

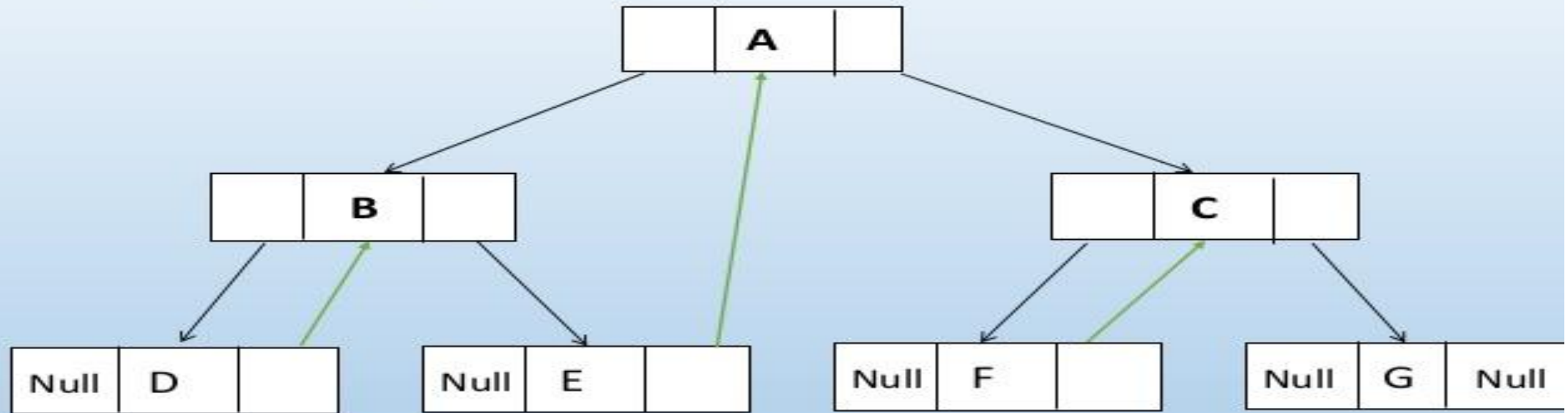
- In above binary tree, there are 8 null pointers & actual 6 pointers.
- In all there are 14 pointers.
- We can generalize it that for any binary tree with n nodes there will be (n+1) null pointers and 2n total pointers.
- The objective here to make effective use of these null pointers.
- A. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.
- According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree: One-Way

- We will use the right thread only in this case.
- To implement threads we need to use in-order successor of the tree

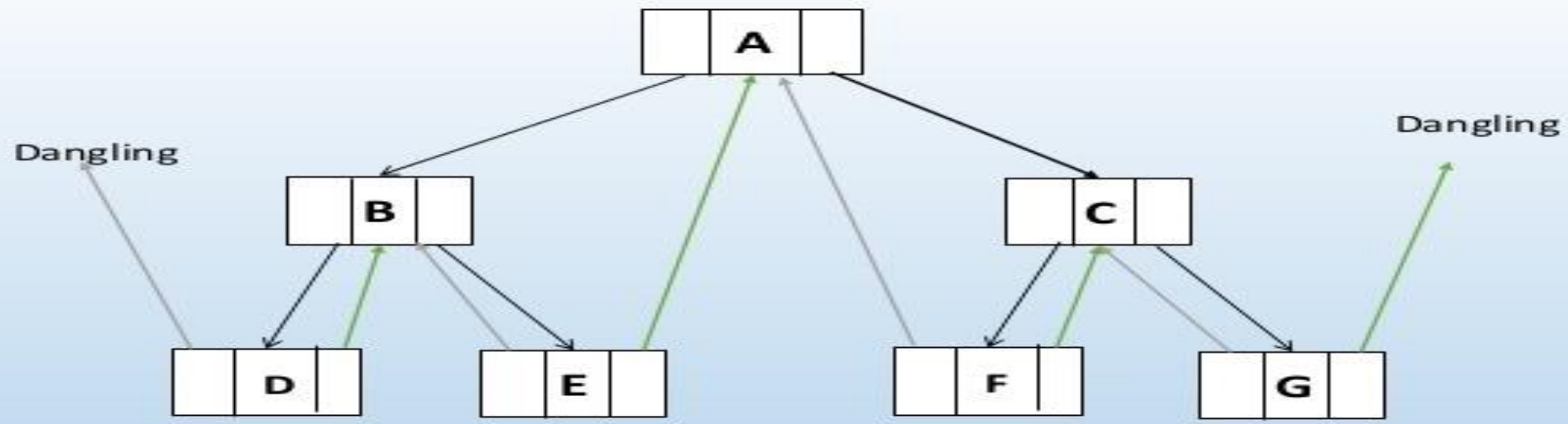
Threaded Binary Tree: One-Way



Inorder Traversal of The tree: D, B, E, A, F, C, G

Two way Threaded Tree/Double Threads

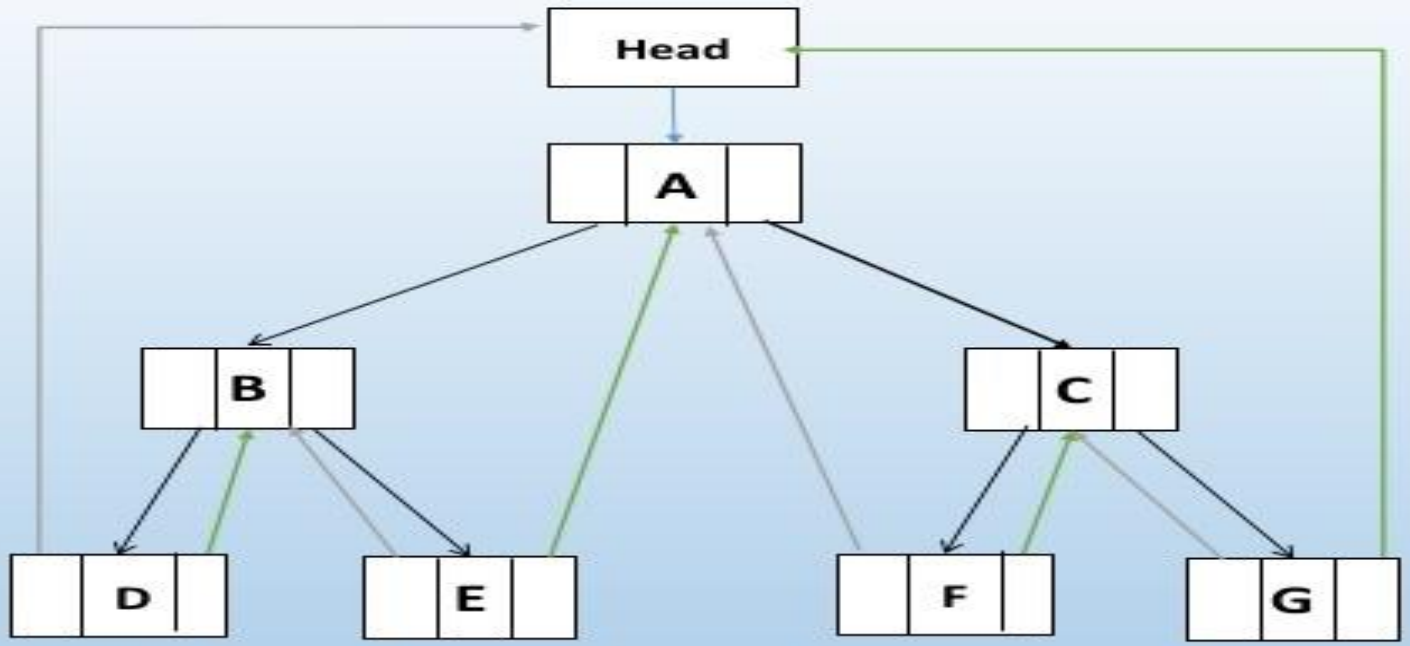
- Again two-way threading has left pointer of the first node and right pointer of the last node will contain the null value. The header nodes is called two-way threading with header node threaded binary tree.



Inorder Traversal of The tree: D, B, E, A, F, C, G

- Dangling can be solved as follows

- Introduce a header node.
- The left and right pointer of the header node are treated as normal links and are initialized to point to header node itself.



Inorder Traversal of The tree: D, B, E, A, F, C, G

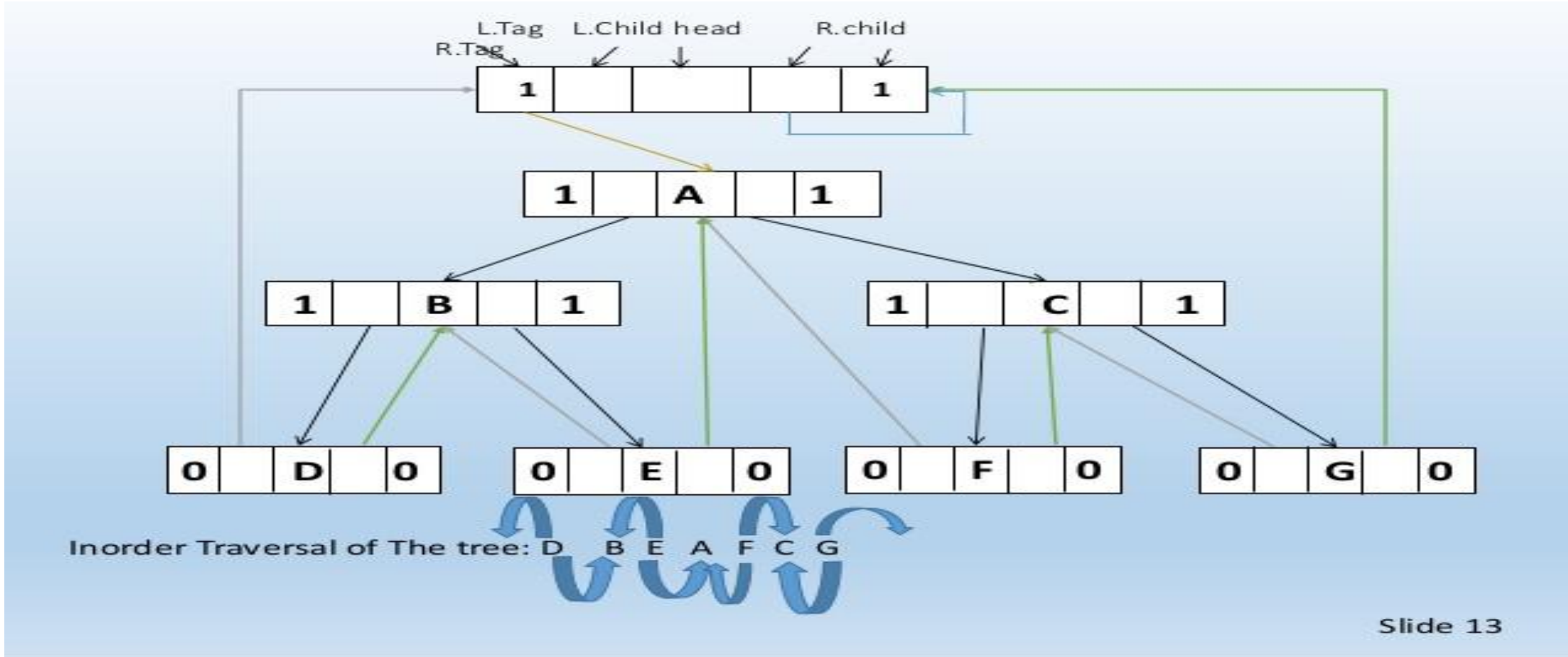
Structure of Thread BT

Node structure

For the purpose of our evaluation algorithm, we assume each node has five fields:

We define this node structure in C as:

```
Struct Node{
        int data;
    struct Node *Left,*Right;
    bool Ltag;
    boolRtag;
        };
```


L.Tag
L.Child head
R.child
R.Tag
1
1
1
A
1
1
B
1
1
C
1
0
D
0
0
E
0
0
F
0
0
G
0
Inorder Traversal of The tree: D B E A F C G

Comparison of Threaded BT

Threaded Binary Trees

- In threaded binary trees, The null pointers are used as thread.
- We can use the null pointers which is a efficient way to use computers memory.
- Traversal is easy. Completed without using stack or reccursive function.
- Structure is complex.
- Insertion and deletion takes more time.

Normal Binary Trees

- In a normal binary trees, the null pointers remains null.
- We can't use null pointers so it is a wastage of memory.
- Traverse is not easy and not memory efficient.
- Less complex than Threaded binary tree.
- Less Time consuming than Threaded Binary tree.

Threaded binary tree

Advantage

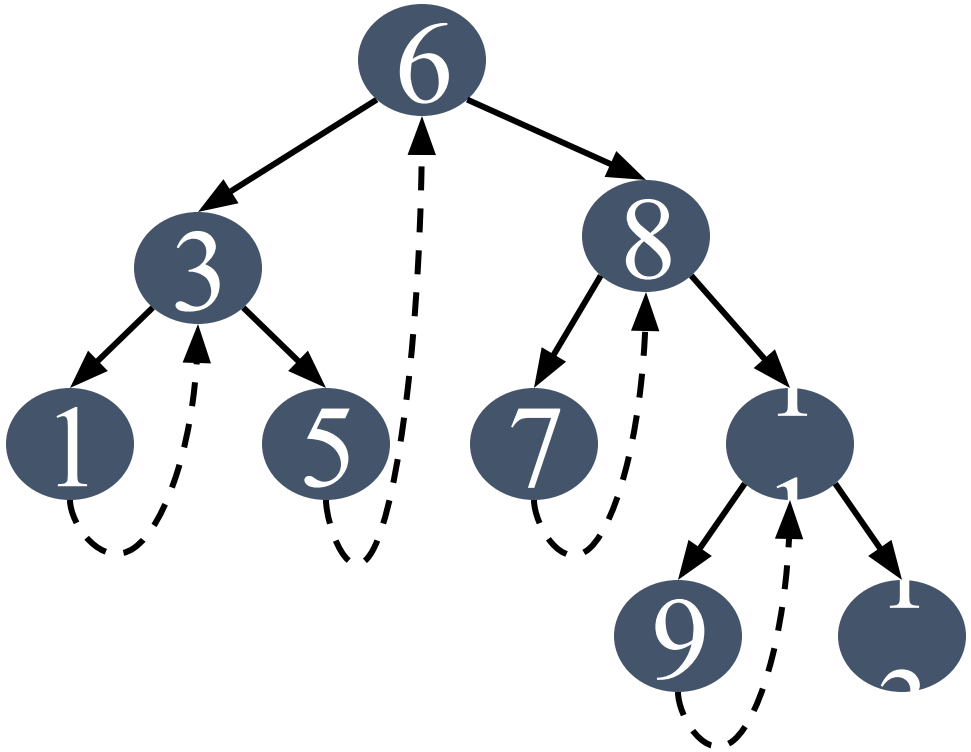
- 1. By doing threading we avoid the recursive method of traversing a Tree , which doesn't use of stack and consumes a lot of memory and time .
- 2 . The node can keep record of its root .
- 3. Backward Traverse is possible.
- 4. Applicable in most types of the binary tree.

Disadvantage

- 1. This makes the Tree more complex .
- 2. More prone to errors when both the child are not present & both values of nodes pointer to their ancestors.
- 3. Lot of time consumes when deletion or insertion is performed.

```
void inOrder(struct Node *root)
{
   struct Node *cur = leftmost(root);
   while (cur != NULL)
    {
        printf("%d ", cur->data);

        // If this node is a thread node, then go to
       // inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

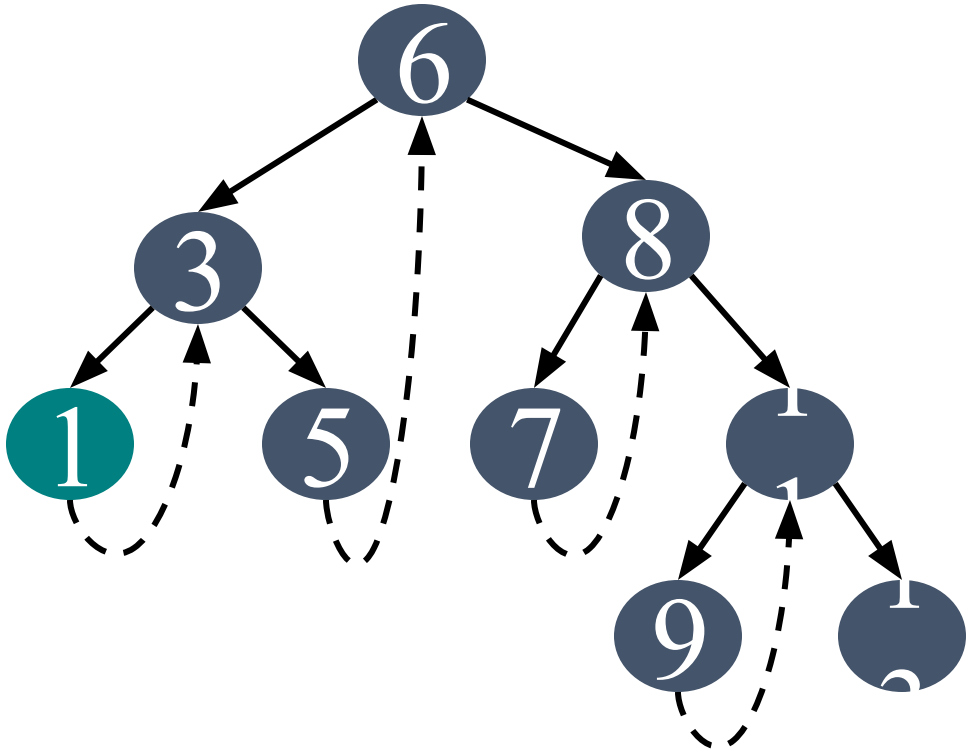
Threaded Tree Example



Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right
- If we follow a link to the right, we go to the leftmost node, print it, and continue

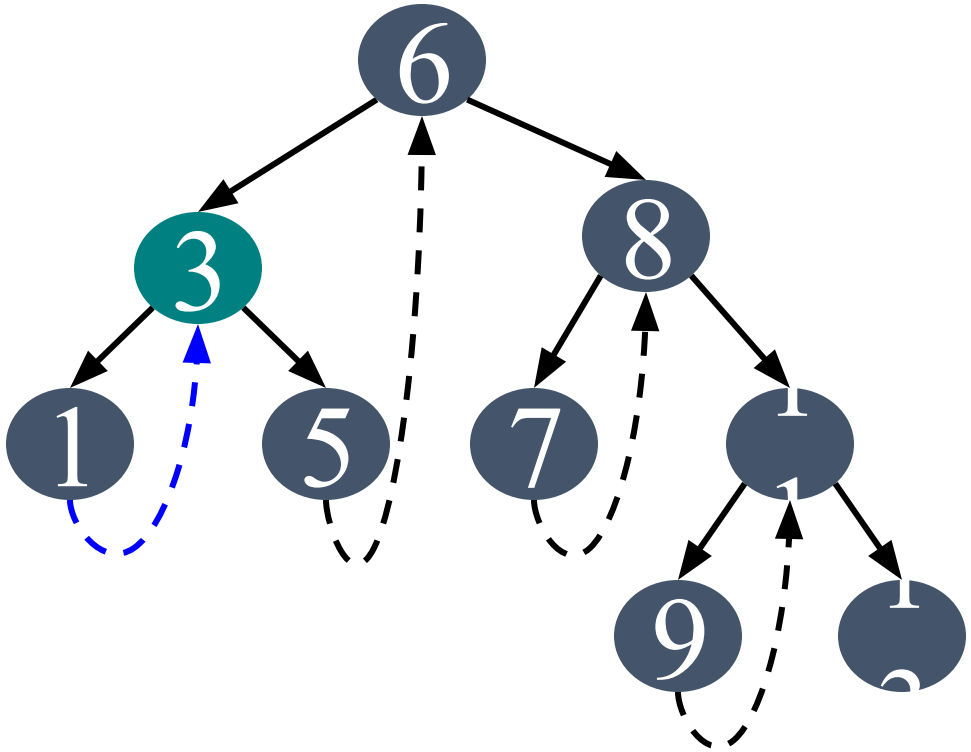
Threaded Tree Traversal



Output
1

Start at leftmost node, print it

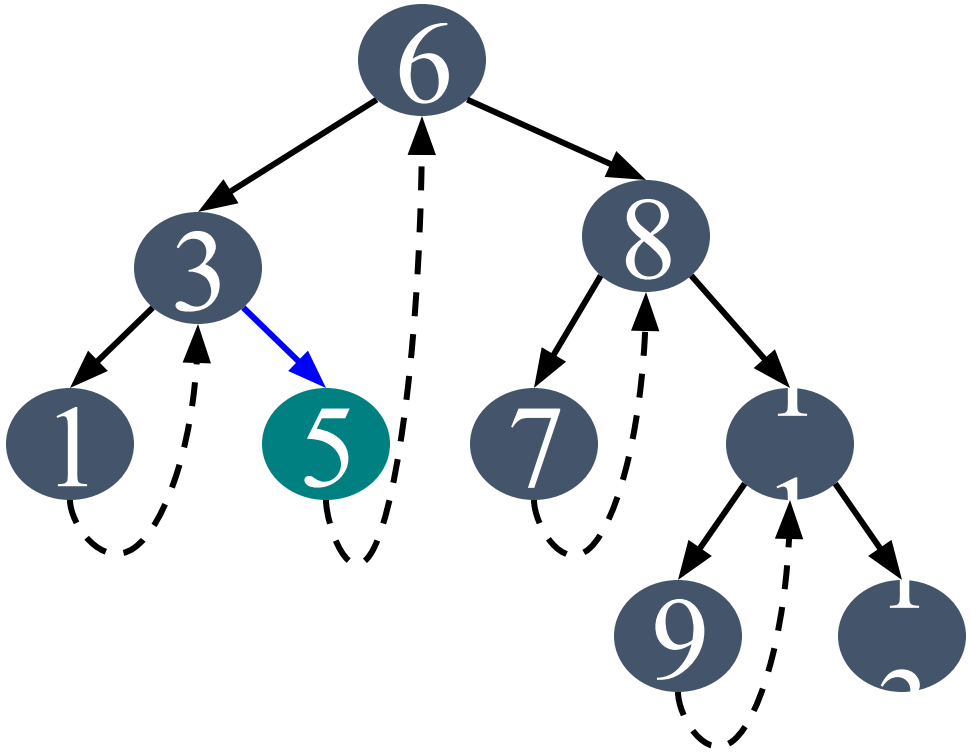
Threaded Tree Traversal



Output
1
3

Follow thread to right, print node

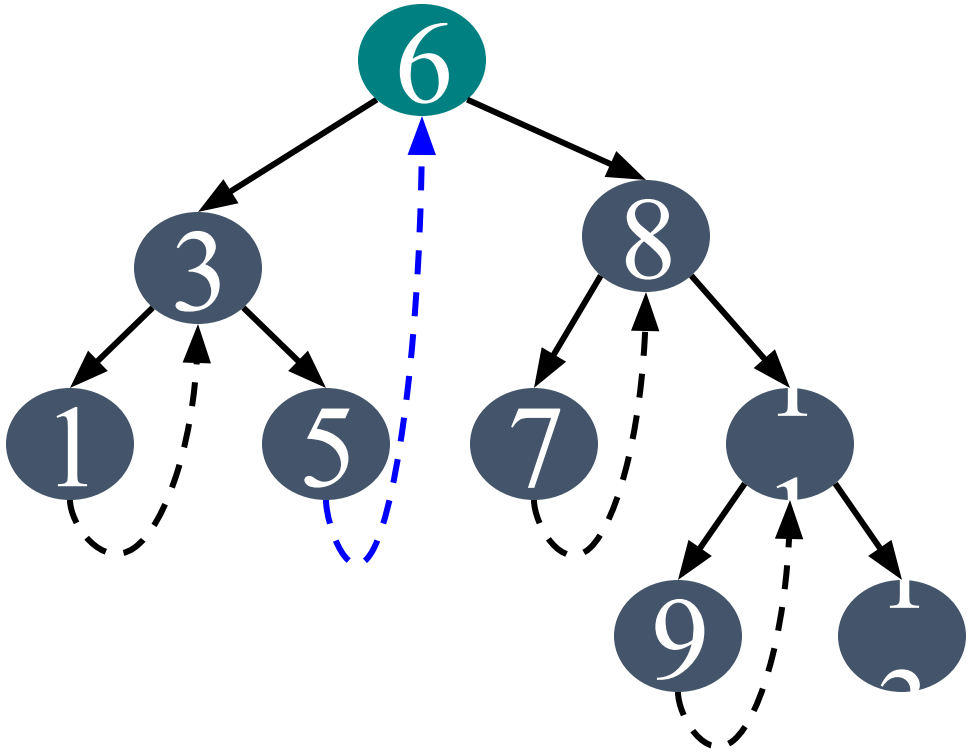
Threaded Tree Traversal



Output
1
3
5

Follow link to right, go to leftmost node and print

Threaded Tree Traversal



Output

1

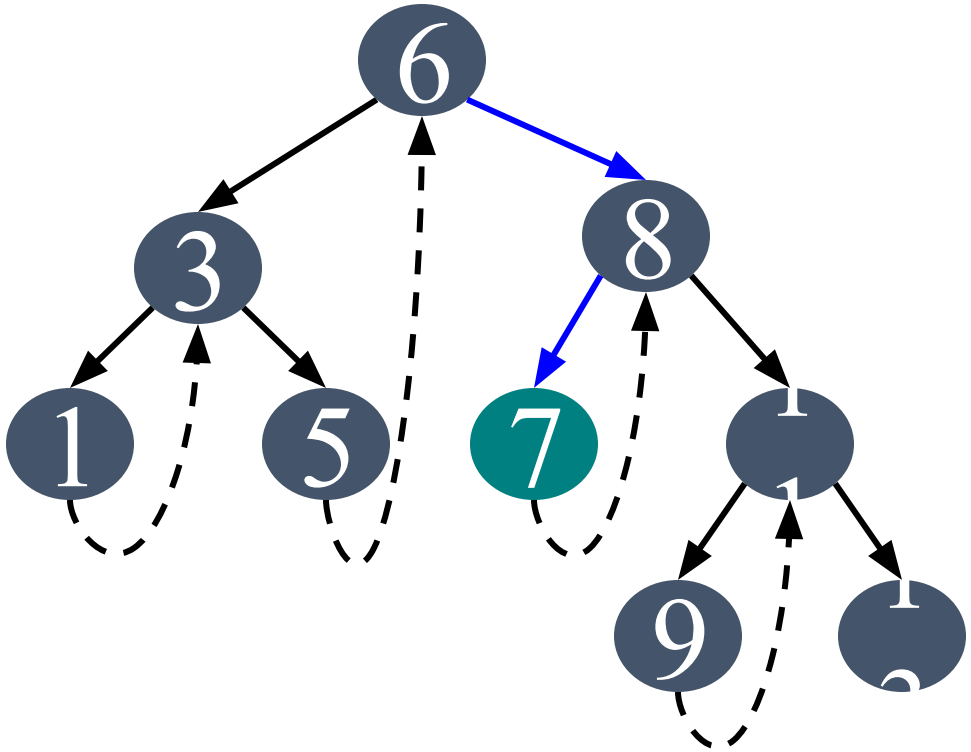
3

5

6

Follow thread to right, print node

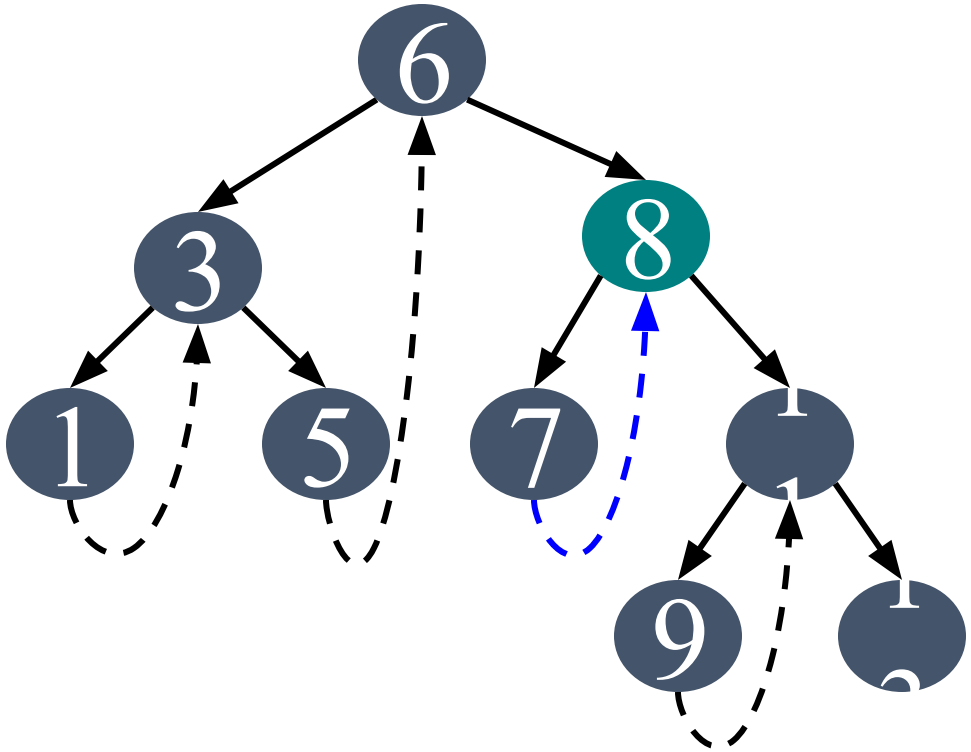
Threaded Tree Traversal



Output
1
3
5
6
7

Follow link to right, go to leftmost node and print

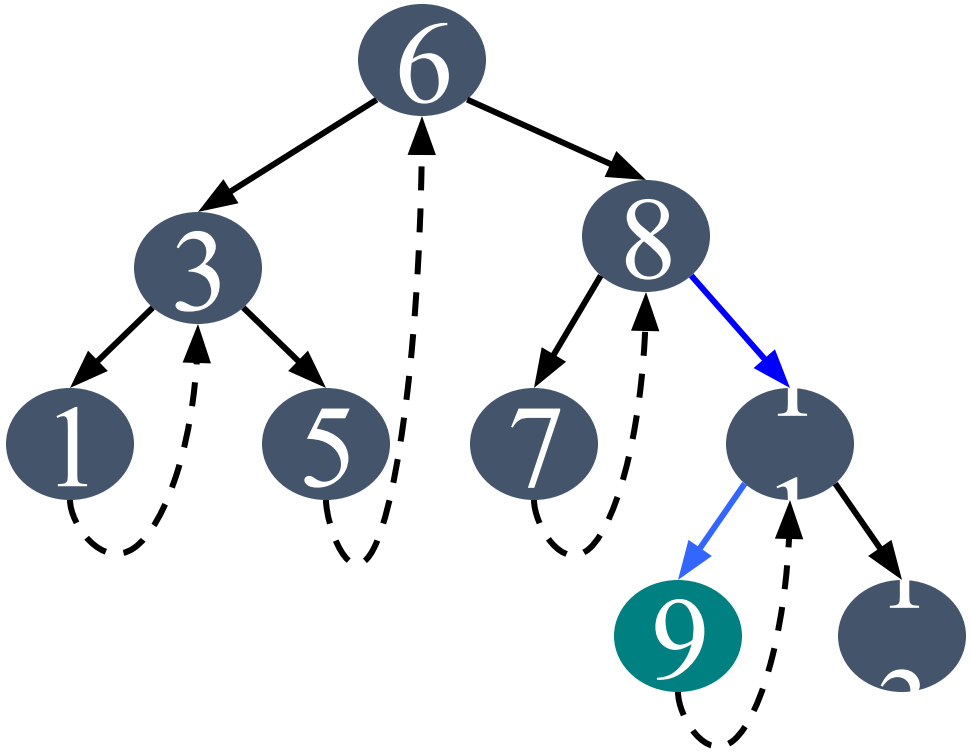
Threaded Tree Traversal



Output
1
3
5
6
7
8

Follow thread to right, print node

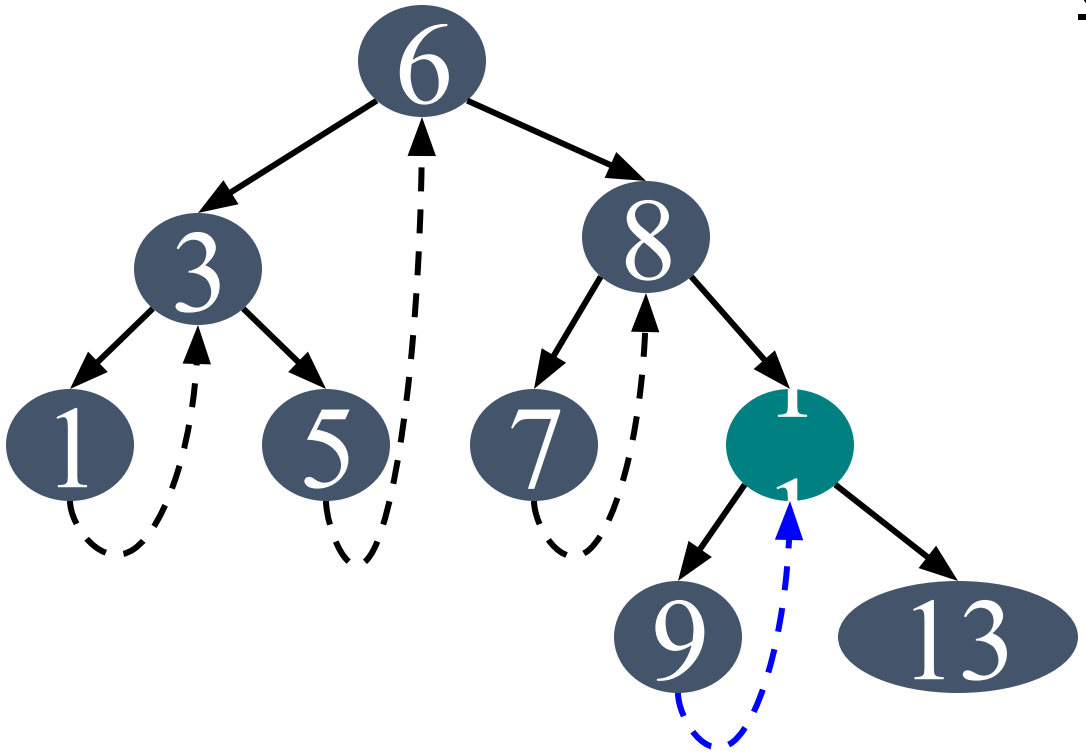
Threaded Tree Traversal



Output
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

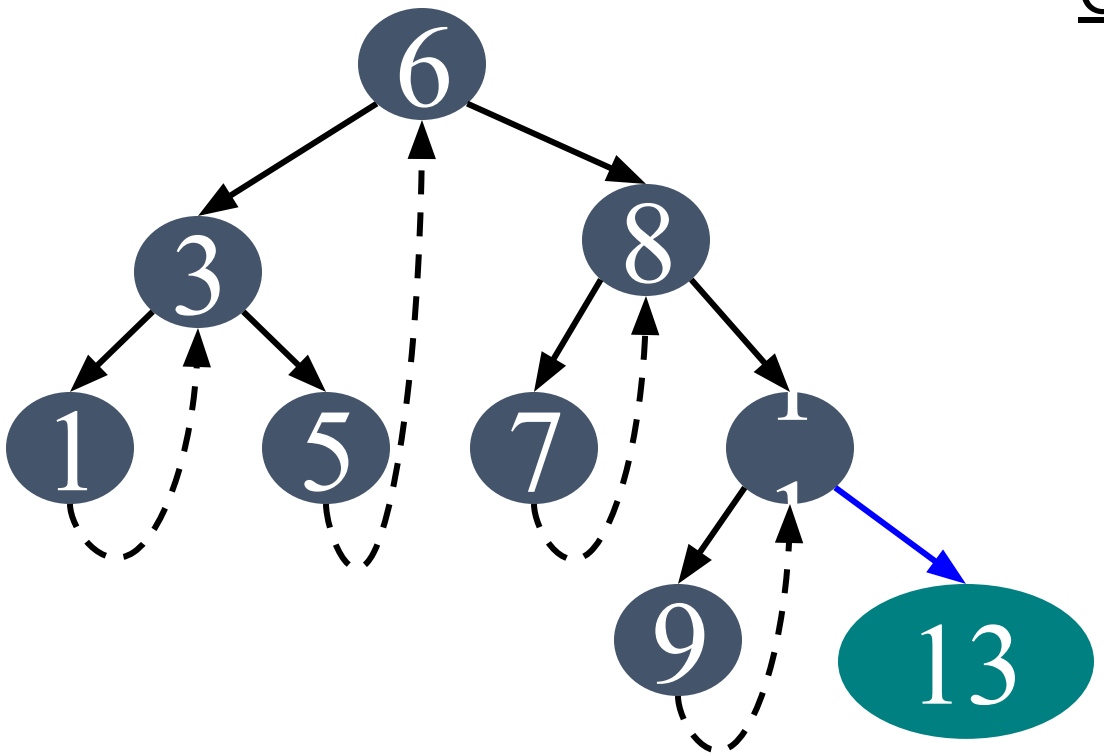
Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11

Follow thread to right, print node

Threaded Tree Traversal



Output
1
3
5
6
7
8
9
11
13

Follow link to right, go to leftmost node and print