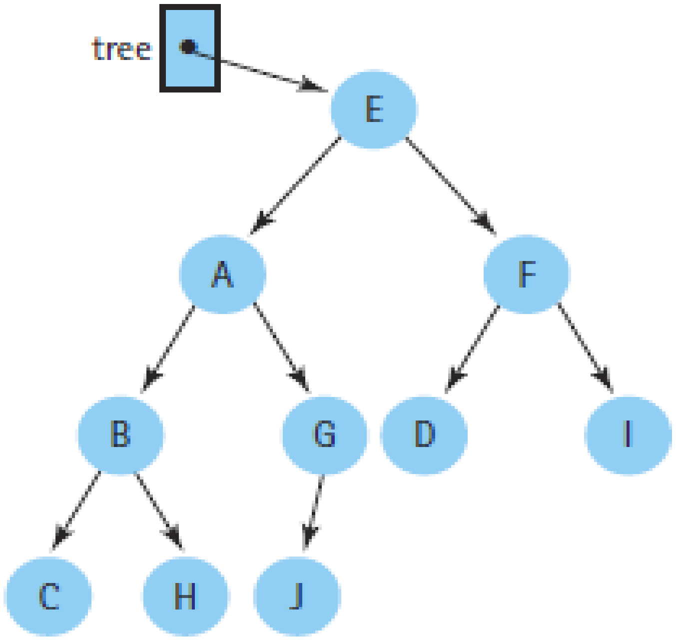
Trees

- Binary trees: introduction (complete and extended binary trees),
- Memory representation (sequential, linked)
- Binary tree traversal: pre-order, in-order and post-order (traversal algorithms using stacks)

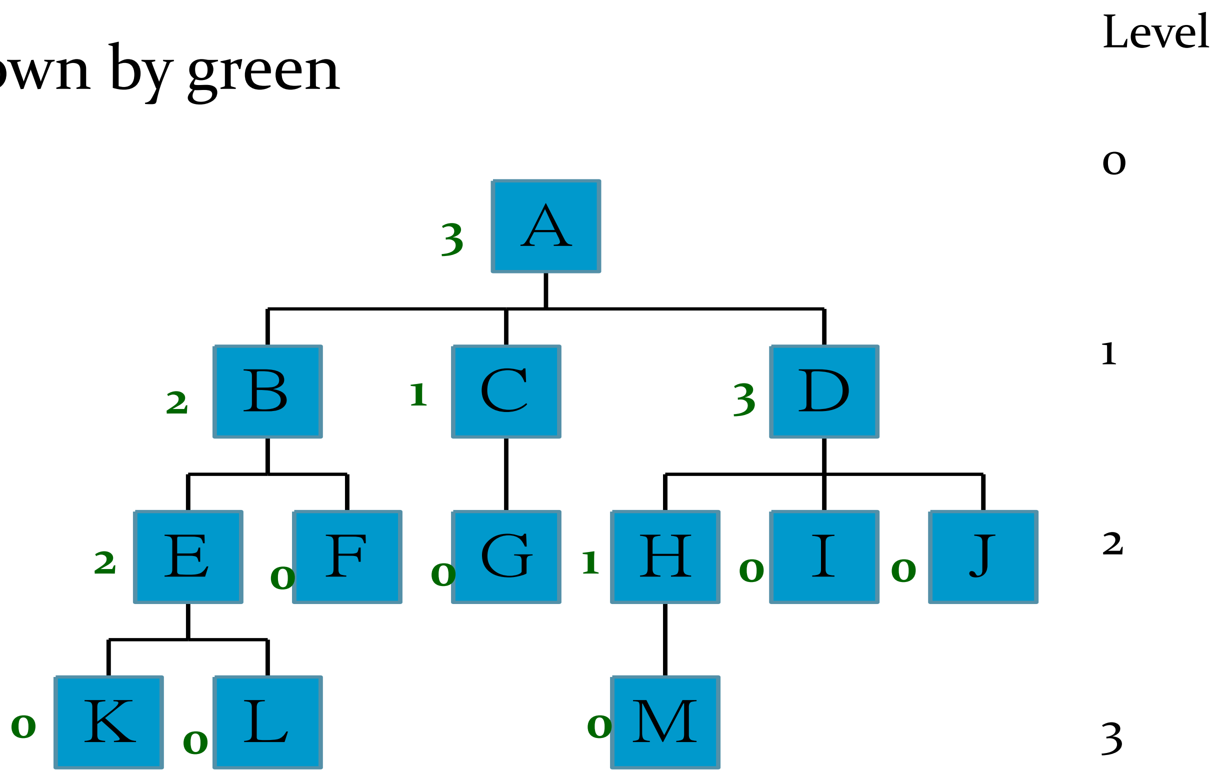
Tree



- Collection of ***nodes*** or
  Finite set of nodes
- This collection can be empty
- Nodes and Edges
- ***Root***
- ***Parent, Child, Siblings, Grand parent, Grand child, Ancestors, Decendents***
- Every node except the root has one parent
- ***Subtree***
- ***Degree*** of a node is # of its sub trees

Level and Depth

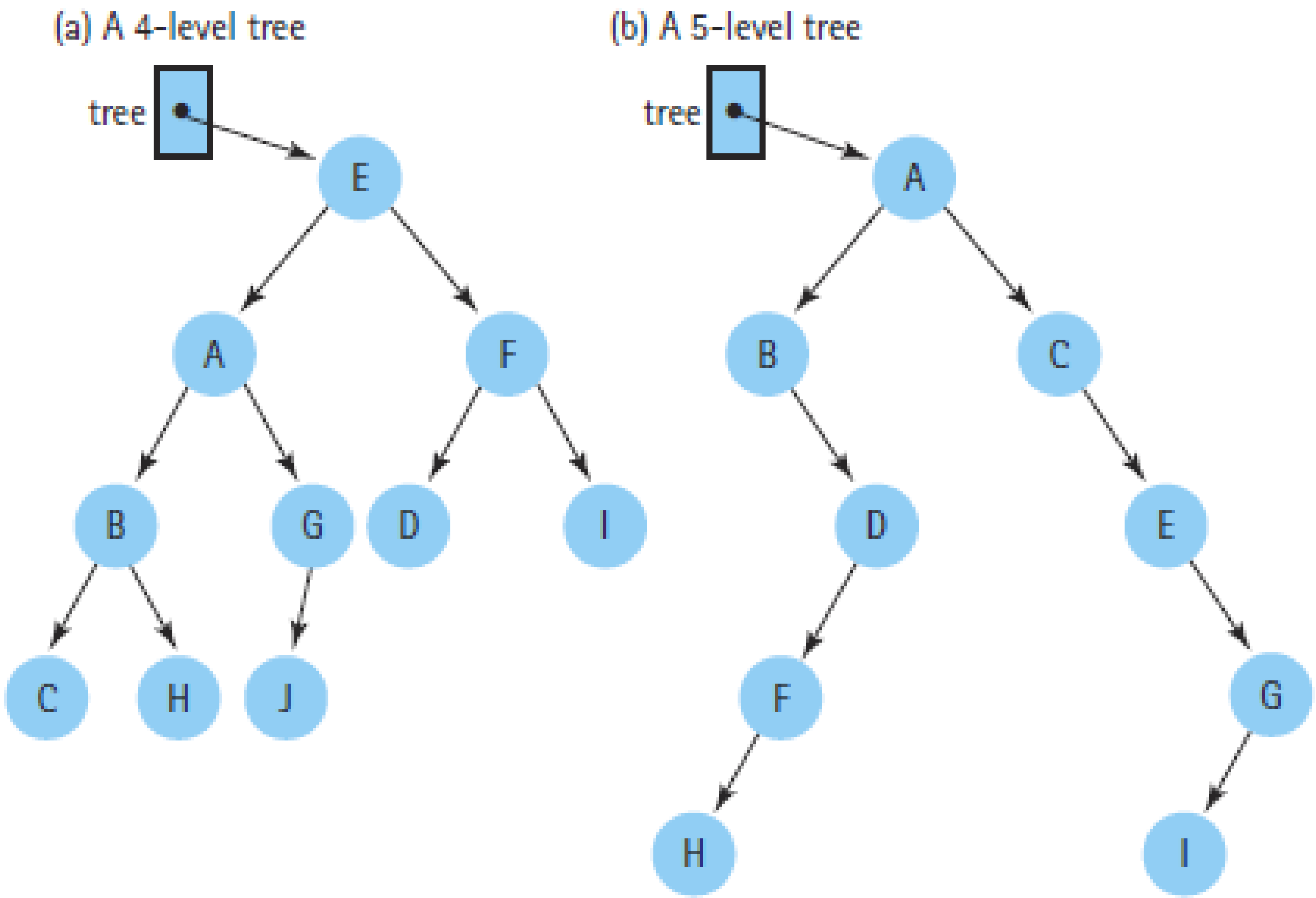
node (13)
degree of a node(shown by green number)
leaf (terminal)
nonterminal
parent
children
sibling
ancestor
descendant
level of a node
height(depth) of a tree (4)



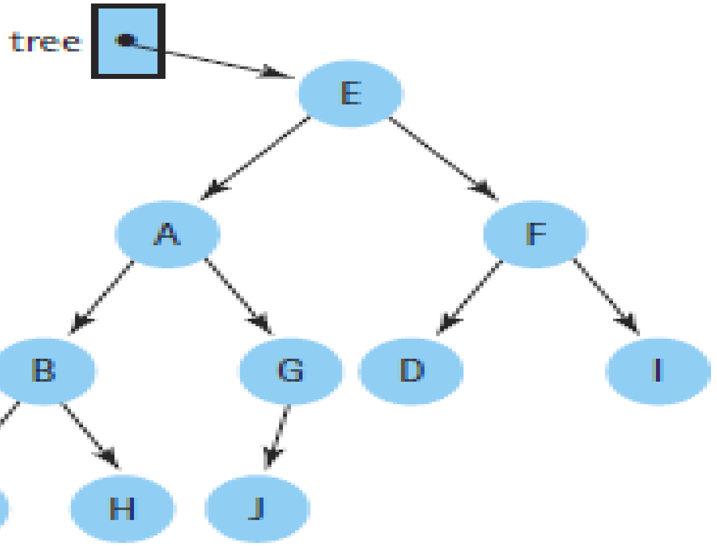
Terminology

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors of a node are all the nodes along the path from the root to the node.

Tree levels

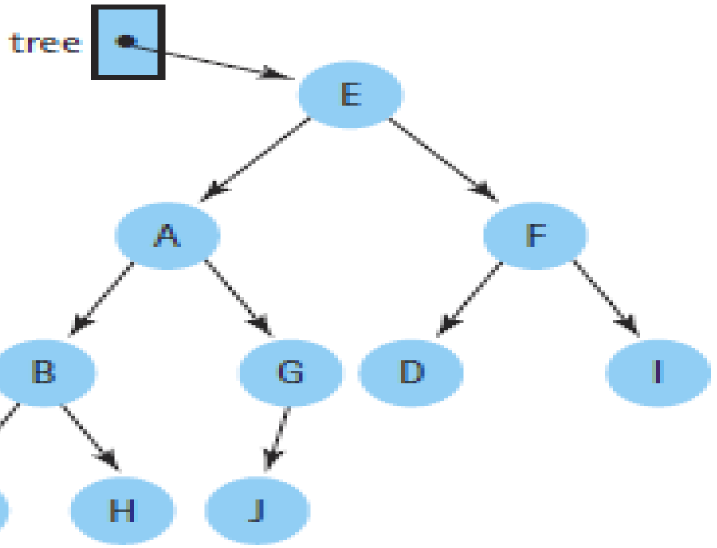

(a) A 4-level tree
tree
E
A
F
B
G
D
I
C
H
J
(b) A 5-level tree
tree
A
B
C
D
E
F
G
H
I

Tree



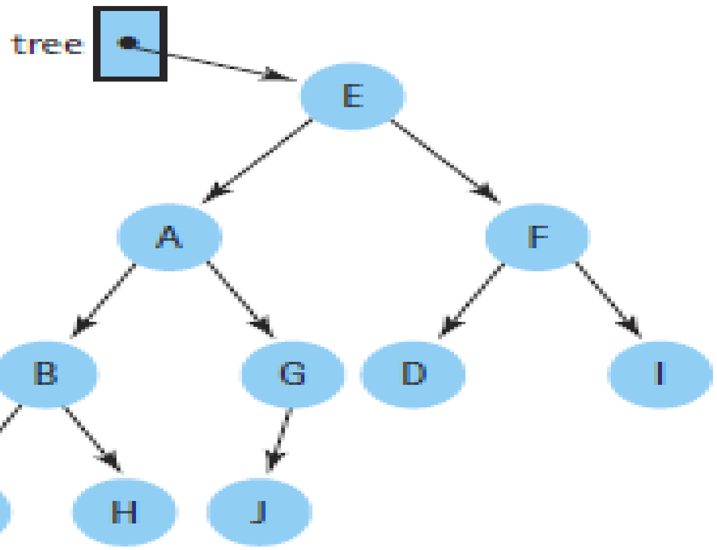
- A ***path*** from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, …….., n_k
- The ***length*** of this path is the number of edges on the path
- There is a path of length zero from every node to itself
- There is exactly one path from the root to each node in a tree

Tree



- ***Height*** of a node is the length of a longest path from this root node to a leaf
- All leaves are at height zero
- Height of a tree is the height of its root (maximum level)

Tree



- ***Depth*** of a node is the length of path from deepest node to the root.
- Root is at depth zero
- Depth of a tree is the depth of its deepest leaf that is equal to the height of this tree

The **depth** of a node is the number of edges from the root to the node.
A root node will have a depth of 0.
Depth is the distance to root node to the particular node.
Height is the distance to the deepest leaf node to the root node.
The **height** of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0.
Height and depth of a tree is equal but height and depth of a node is not equal because the height is calculated by traversing from the given node to the deepest possible leaf. Depth is calculated from traversal from root to the given node.

Difference between tree and binary tree

1.A general **tree** is a data structure in that each node can have infinite number of children.

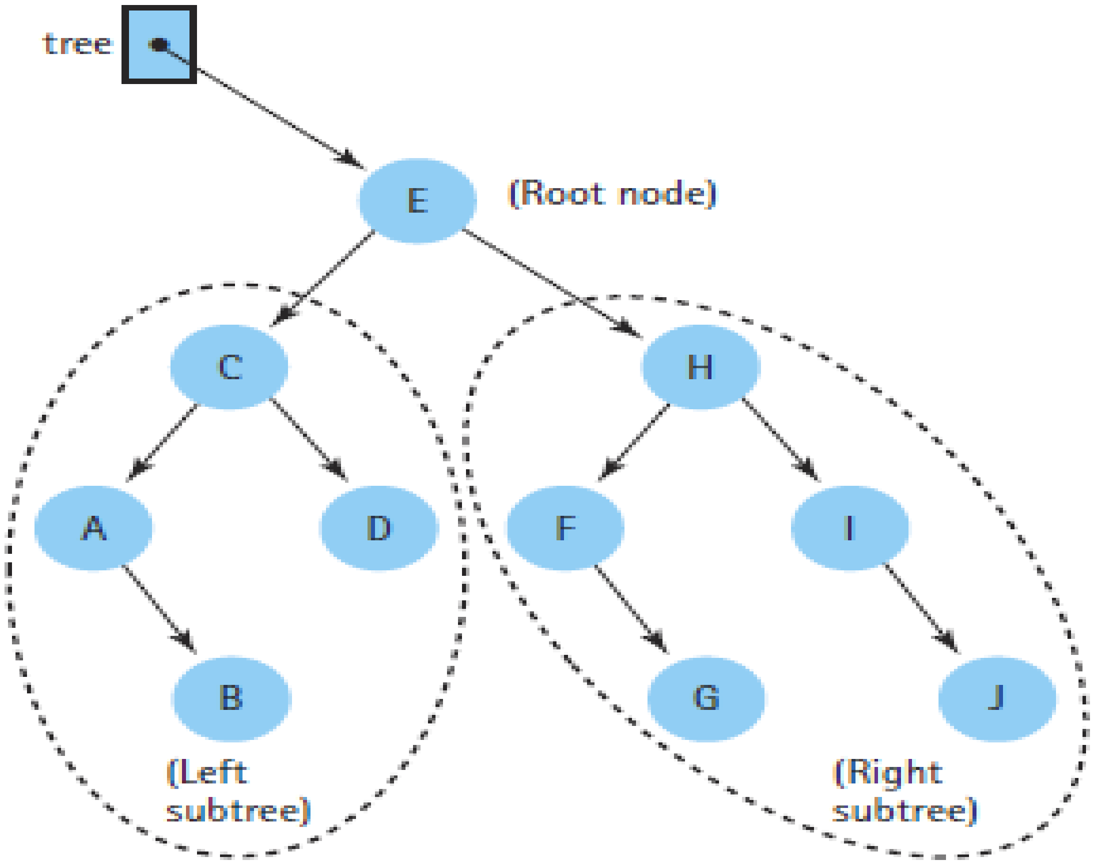
2. A **Binary tree** is a data structure in that each node has at most two nodes left and right.

3.There is no limit on the degree of node **in a** general **tree**.

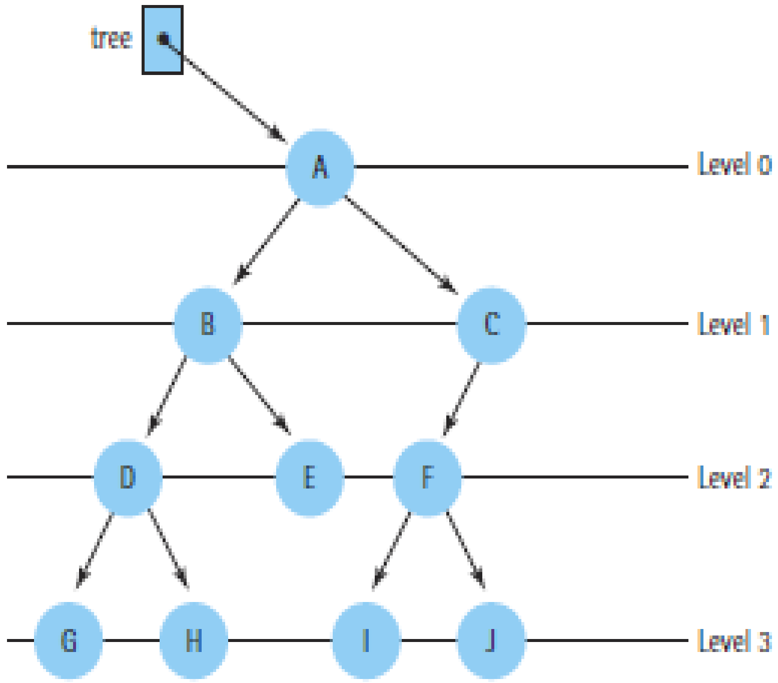
4.Nodes **in a binary tree** cannot have more than degree 2

Binary Trees

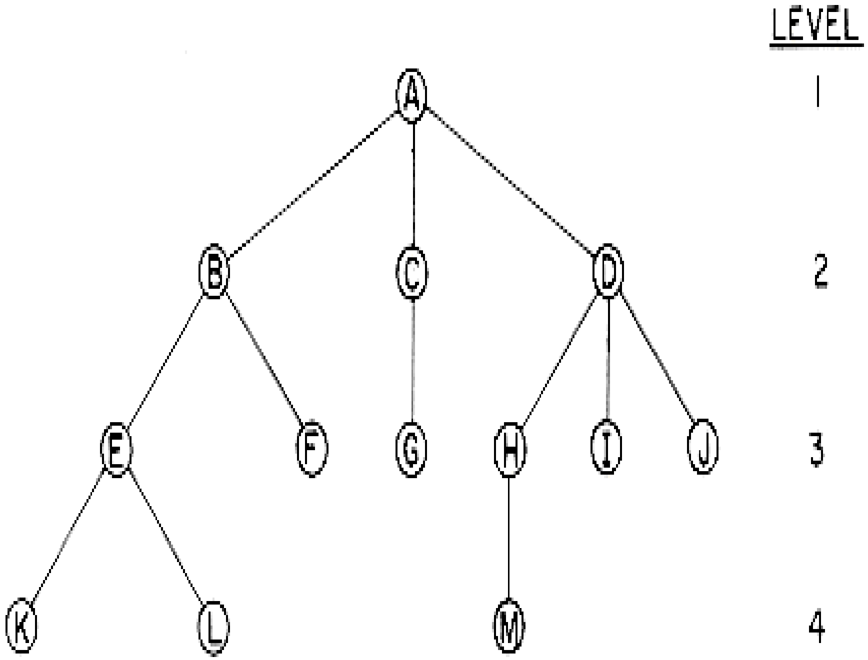
- There is no node with degree greater than two



Binary Tree

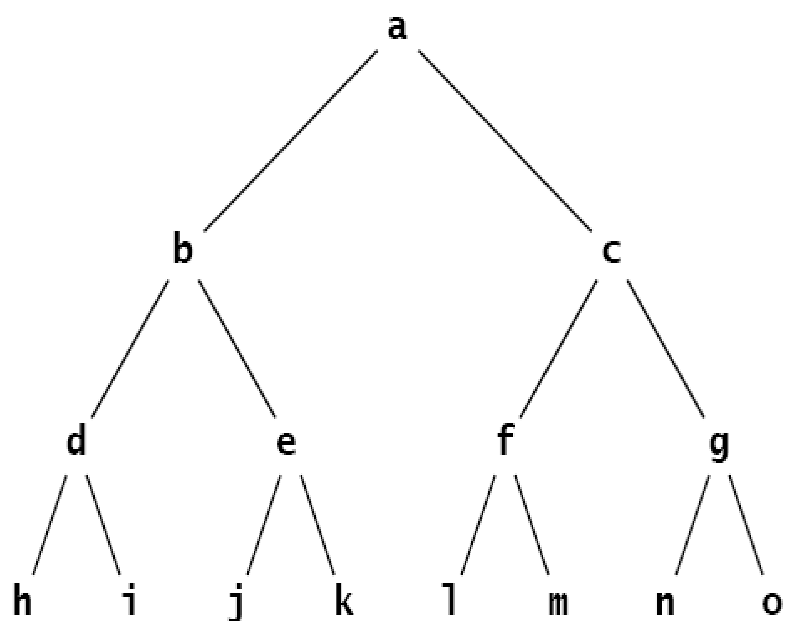


Binary tree

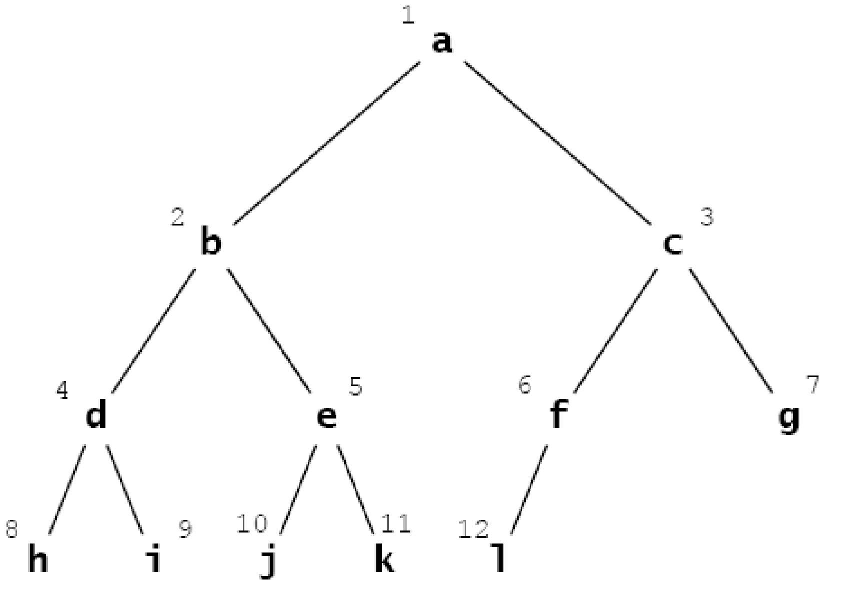


Not a binary tree

Binary Tree

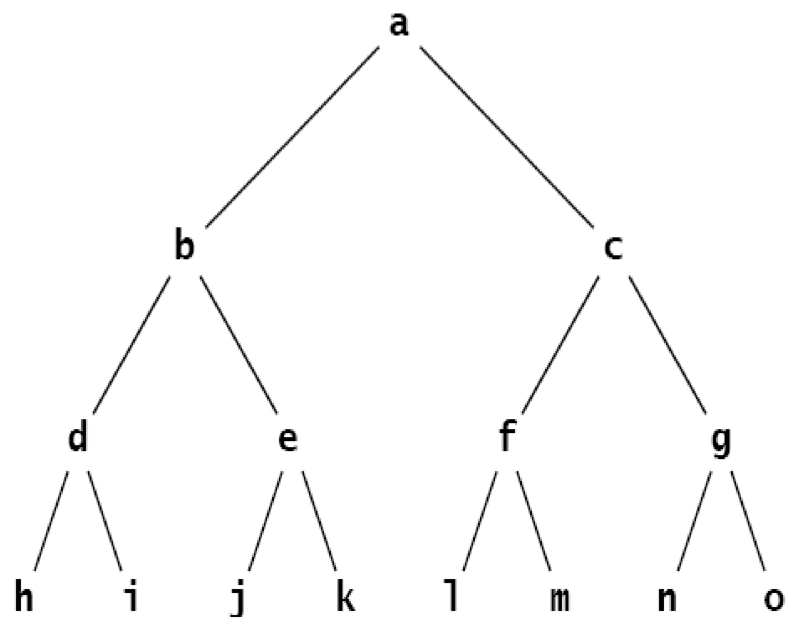


Full binary tree

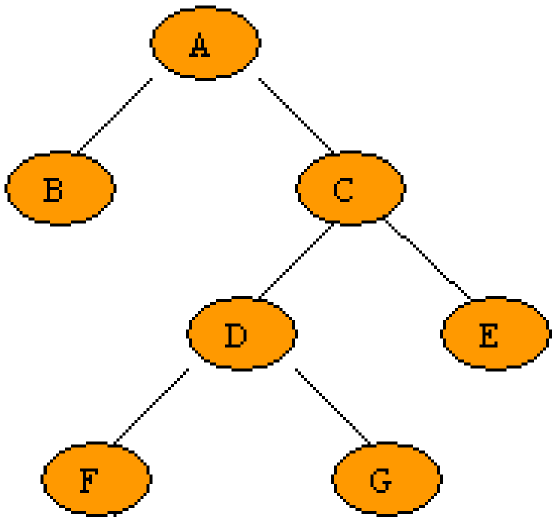


Complete binary tree

Full Binary Tree

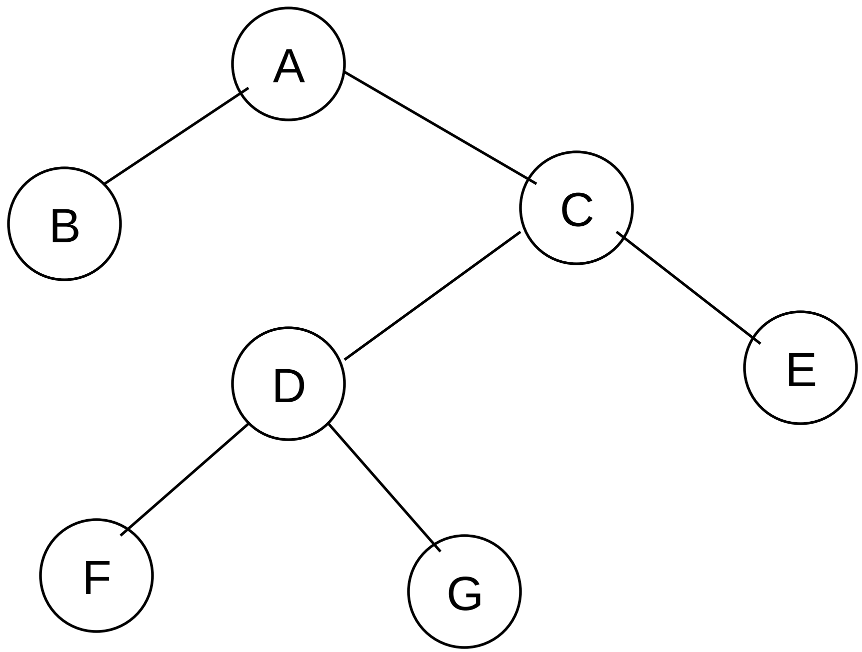


A full binary tree (sometimes proper binary tree or 2-tree or strictly binary tree or extended binary tree) is a tree in which every node other than the leaves has two children.



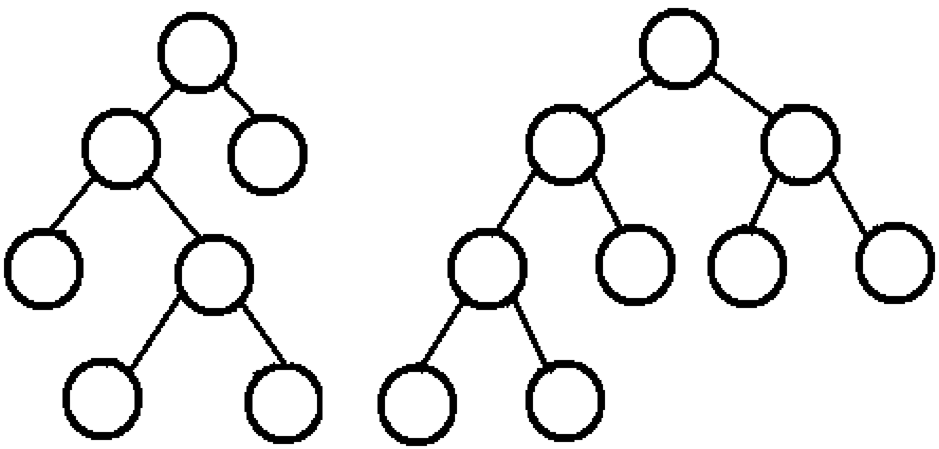
Full/Strictly binary trees

- If every non-leaf node in a binary tree must have two subtree left and right sub-trees, the tree is called a strictly binary tree.
- A strictly binary tree with n leaves always contains 2n -1 nodes.
- Here, B,E,F,G are leaf nodes. So n=4, No. of nodes=2*4-1=7

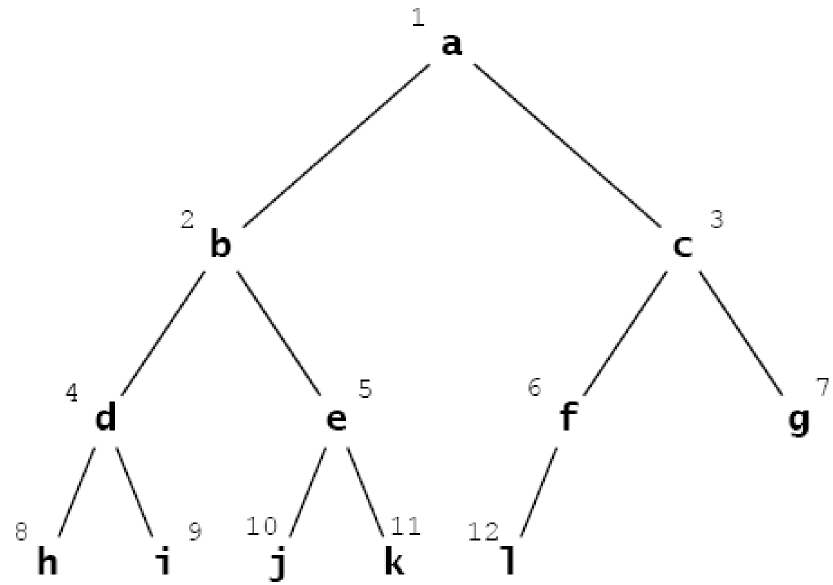


Complete Binary Tree

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

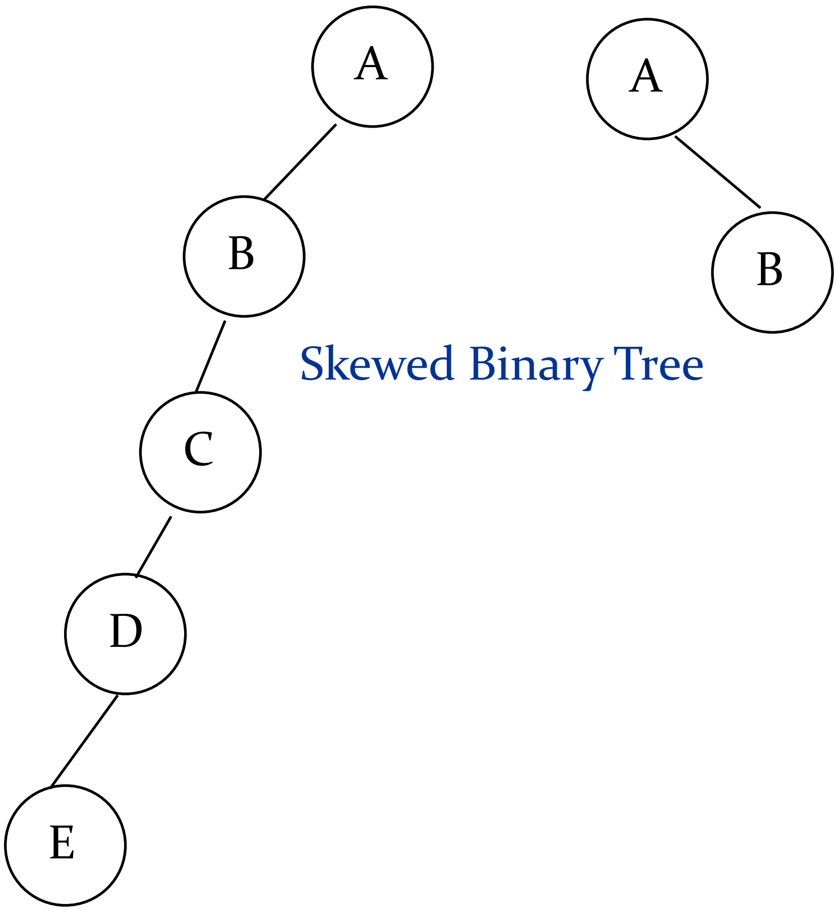


full tree   complete tree



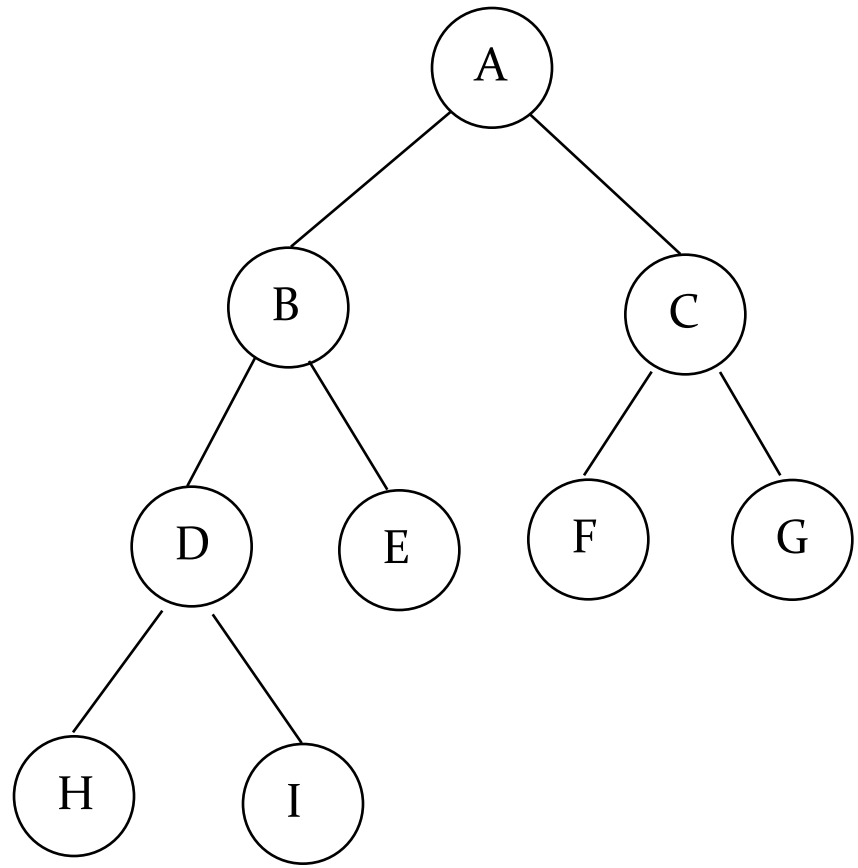
Full v.s. **Complete Binary Trees**.
A **full binary tree**(sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children.
A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Samples of Trees



Skewed Binary Tree

Complete Binary Tree



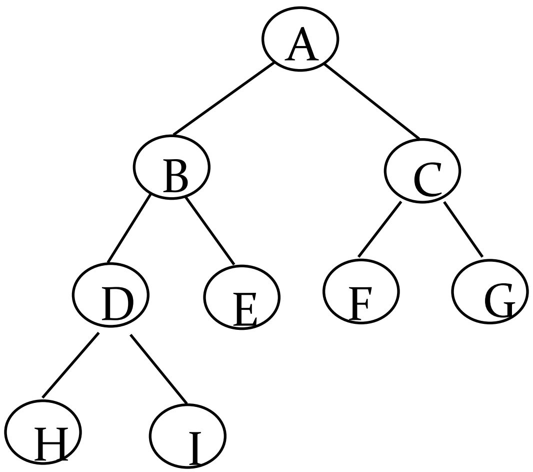
Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is $2^{i+1}-1$, $i>=0$.

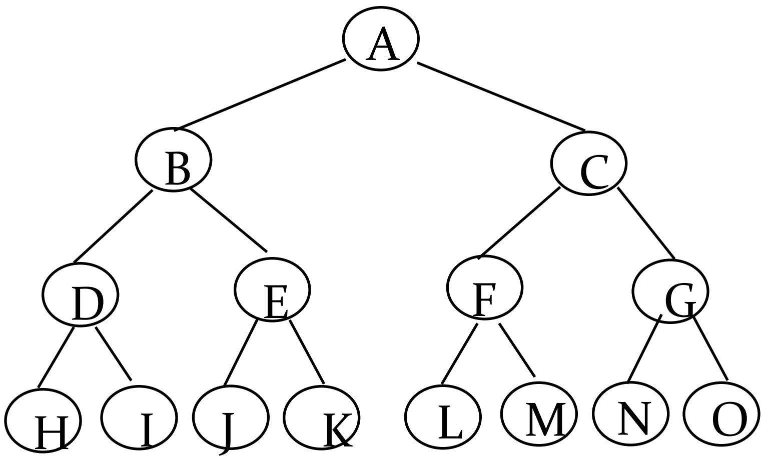
- The maximum number of nodes in a binary tree
  of height h is $2^{h}-1$, $k>=1$.

Full BT vs. Complete BT

- A full binary tree of depth k is a binary tree of depth 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.



Complete binary tree



Full binary tree of depth 4

Complete Binary Tree

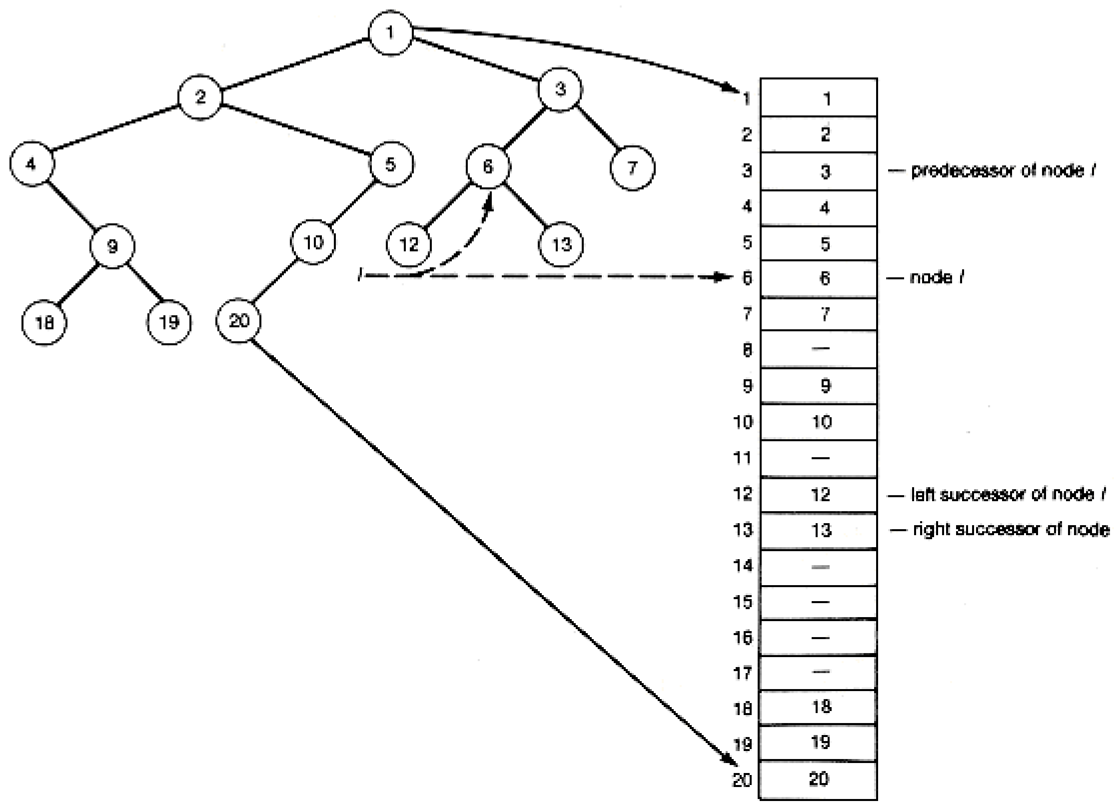
- If a complete binary tree with n nodes
  (depth= $\lfloor \log n + 1 \rfloor$)

is represented sequentially,

then for any node with index i, $1<=i<=n$, we have:

  - *parent*(i) is at $\lfloor i/2 \rfloor$ if $i!=1$. If $i=1$, i is at the root and has no parent.
  - *leftChild*(i) is at $2i$ if $2i<=n$. If $2i>n$, then i has no left child.
  - *rightChild*(i) is at $2i+1$ if $2i+1<=n$. If $2i+1>n$, then i has no right child.

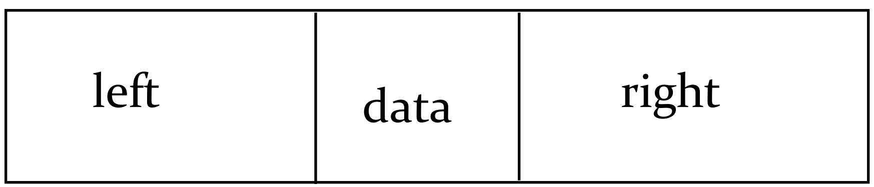
Sequential Representation of binary tree

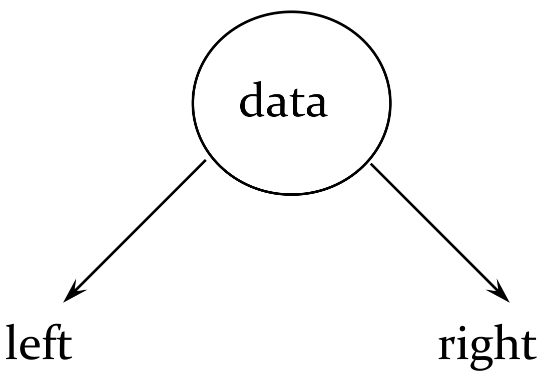


Linked Representation

```
struct btnode {
 int data;
 btnode *left, *right;

};
```





Applications of Binary tree

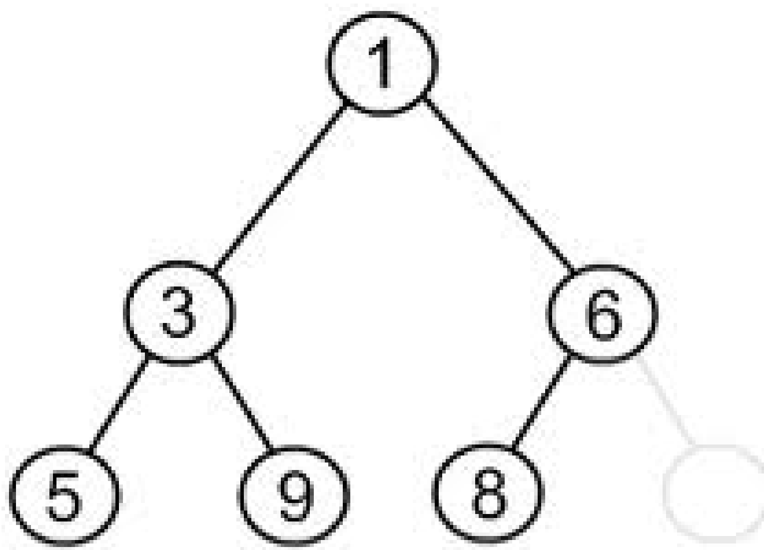
- Implementing routing table in router
- Expression evaluation
- To solve database problems such as indexing.
- Data Compression code.

Tree Traversals

- Pre-Order
  - N L R
- In-Order
  - L N R
- Post-Order
  - L R N

Tree Traversals

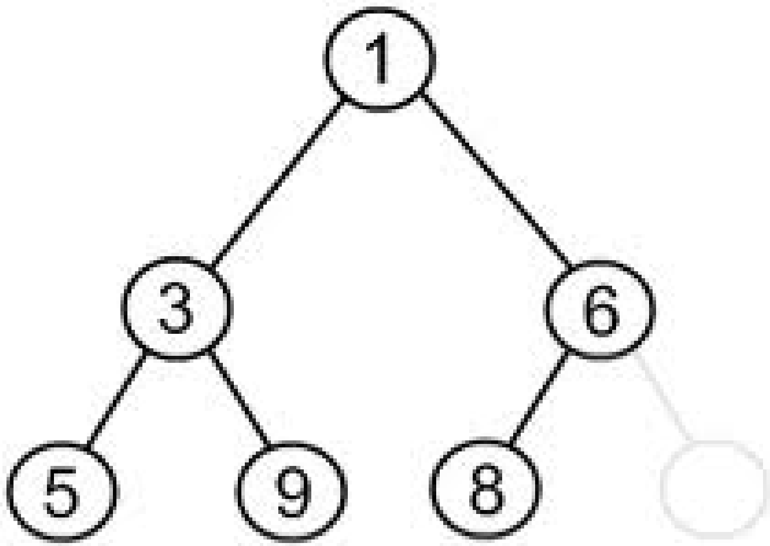
Pre-Order(NLR)



1, 3, 5, 9, 6, 8

Tree Traversals

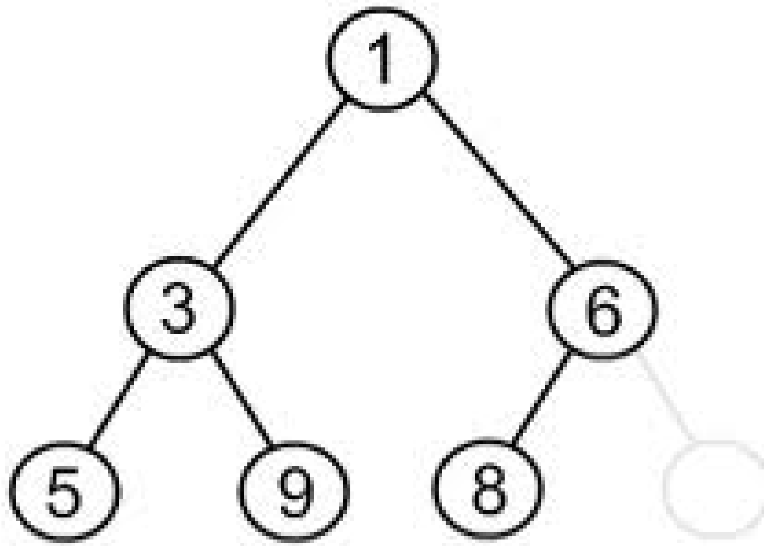
In-Order(LNR)



5, 3, 9, 1, 8, 6

Tree Traversals

Post-Order(LRN)



5, 9, 3, 8, 6, 1

Preorder traversal

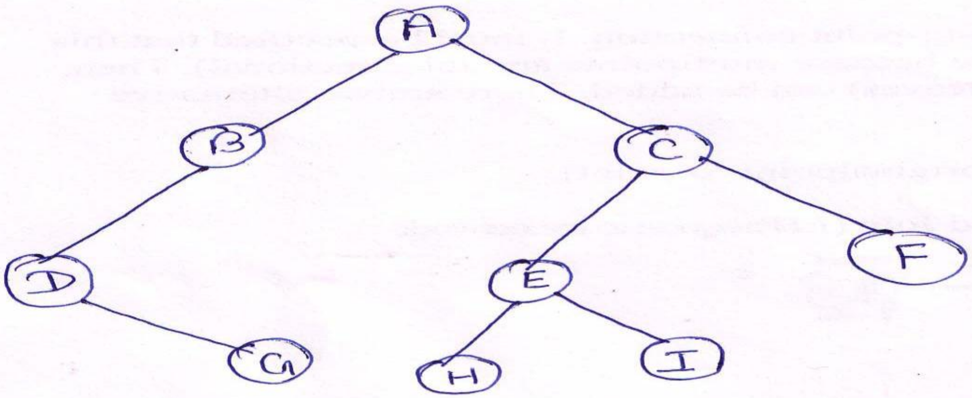
PREORDER(INFO,LEFT,RIGHT,ROOT**)**

1. **[initially push NULL onto STACK , and initialize PTR]**
   Set TOP = 1, STACK[1] = NULL and PTR = ROOT
2. Repeat steps 3 to 5 while PTR≠NULL
3. Apply PROCESS to PTR →info
4. **[right child?]**
   If PTR →right ≠ NULL , then **[push on stack]**
   Set TOP = TOP+1, and STACK[TOP] = right[ptr]
   [end of if structure]
5. **[left child?]**
   If PTR →left ≠ NULL , then
   Set PTR = PTR →left
   Else: **[pop from stack]**
   Set PTR = STACK[TOP] and TOP = TOP-1
   [end of if structure]

[end of step 2 loop]

6.exit

example



Stack

A

C, B

C, D

C, G

C

F

F, F

F, I, H

F, I

F

Print

A

A B

A B D

A B D G

A B D G C

A B D G C

A B D G C E

A B D G C E H

A B D G C E H I

A B D G C E H I F

In-order traversal

INORDER(INFO,LEFT,RIGHT,ROOT)

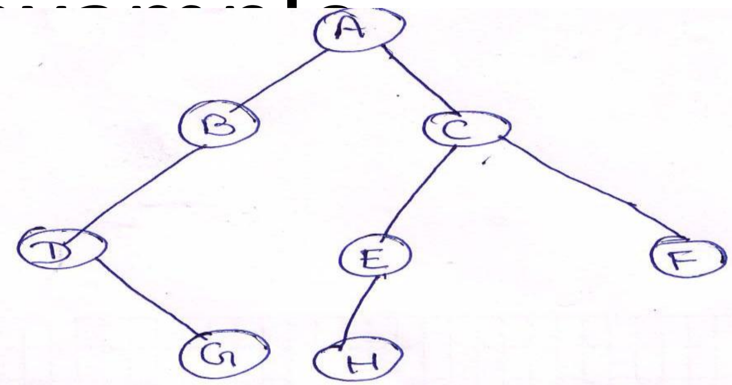
1. **[initially push NULL onto STACK , and initialize PTR]**
   Set TOP = 1,   STACK[1] = NULL   and   PTR = ROOT
2. Repeat step 2 while PTR[Left] ≠NULL
   a) set TOP = TOP+1   and   STACK[TOP] = PTR   **[pushes left most path onto stack]**

   b) set PTR = PTR →left
   [end of loop]

Set PTR = STACK[TOP]   and   TOP = TOP-1   **[pop root of sub-tree from stack]**

3. Repeat steps 5 to 7 while PTR ≠ NULL
4. Apply PROCESS to PTR →info
5. **[right child?]** If PTR →right ≠ NULL then:
   a) Set PTR = PTR →right
   b) go to step 2
   [end of if structure]
6. Set PTR = STACK[TOP]   and   TOP = TOP-1   **[backtracking]**
   [end of step 4 loop]
7. Exit

example



Stack	Print
A	
A, B	
A, B, D	
A, B	D
A, B, G	
A, B	D G
A	D G B
C	D G B A
C, E	D G B A
C, E, H	D G B A
C, E	D G B A H
C	D G B A H E
F	D G B A H E C
—	D G B A H E C F

Postorder traversal

1.P= Root.

2. Push(Stack, Root)

3. Repeat step 4 to 13 while (stack is non empty)

4. P=Stack[Top]

5. If (Left[P]!=Null && Visited[Left[P]]==FALSE)

6. Push(Stack, Left[P])

7. Else

8.If(Right[P]!=Null && Visited[Right[P]]==FALSE)

9. Push(Stack, Right[P])

10. Else

11. Pop(Stack)

12. Print Info[P]

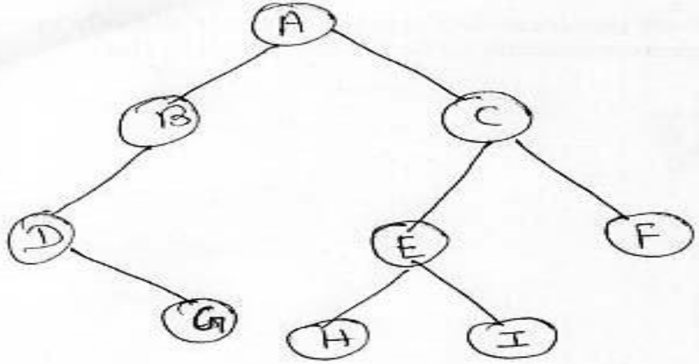
13.visited[P]=TRUE

[End of if]

[End of while]

14.Stop

```
#define BOOL int
struct node {
 int info;
 struct node *left;
 struct node
   *right;
BOOL visited;
 };
```



Stack		Print
A	P = A	
A, B	P = B	
A, B, D	P = D	
A, B, D, G	P = G visited true	G
A, B, D	P = D visited true	GD
A, B	P = B visited true	GDB
A	P = A	
A, C	P = C	
A, C, E	P = E	
A, C, E, H	P = H visited true	GDBH
A, C, E	P = E	
A, C, E, I	P = I visited true	GDBHI
A, C, E	P = E visited true	GDBHIE
A, C	P = C	
A, C, F	P = F visited true	GDBHIEF
A, C	P = C visited true	GDBHIEFC
A	P = A visited true	GDBHIEFCA

Tree traversal Applications

- **Pre-order**
  – Tree copying
  – Counting the number of nodes
  – Counting the number of leaves
  – Prefix notation from a expression tree
- **Post-order**
  – Deleting a binary tree
  – All stack oriented programming languages – mostly functional languages which fully works on nested functions.
  – Calculator programs
  – postfix notation in an expression tree – used in calculators
- **In-order**
  – we can extract the sorted values in a BST

Problem: Create a tree from the given traversals

preorder: F A E K C D H G B

inorder: E A C K F H D B G

Solution: The tree is drawn from the root as follows:

(a) The root of tree is obtained by choosing the first node of preorder. Thus F is the root of the proposed tree

(b) The left child of the tree is obtained as follows:

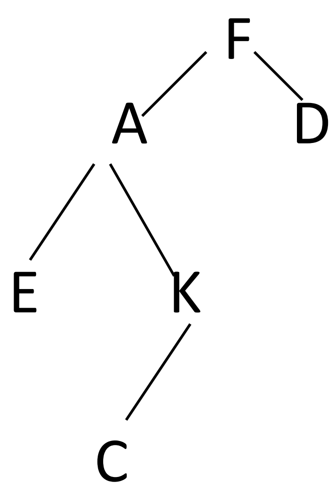
   (a) Use the inorder traversal to find the nodes to the left and right of the root node selected from preorder. All nodes to the left of root node(in this case F) in inorder form the left subtree of the root(in this case E A C K)

   (b) All nodes to the right of root node (in this case F) in inorder form the right subtree of the root (H D B G)

   (c) Follow the above procedure again to find the subsequent roots and their subtrees on left and right.

- F is the root   Nodes on left subtree(left of F):E A C K (from inorder)
  Nodes on right subtree(right of F):H D B G(from inorder)
- The root of left subtree:
- From preorder: **A** E K C , Thus the root of left subtree is A
- **D** H G B , Thus the root of right subtree is D
- Creating left subtree first:
  From inorder: elements of left subtree of A are: **E** (root of left)
  elements of right subtree of A are: **C** K (root of right)

Thus tree till now is:

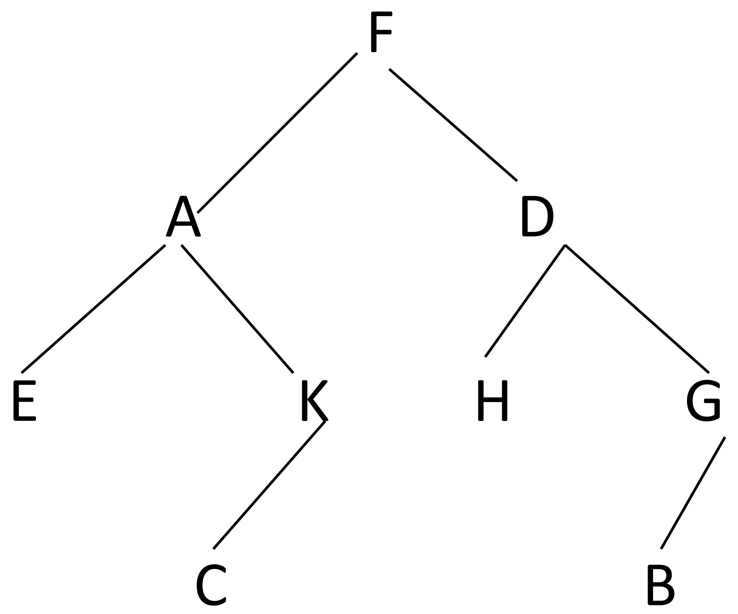


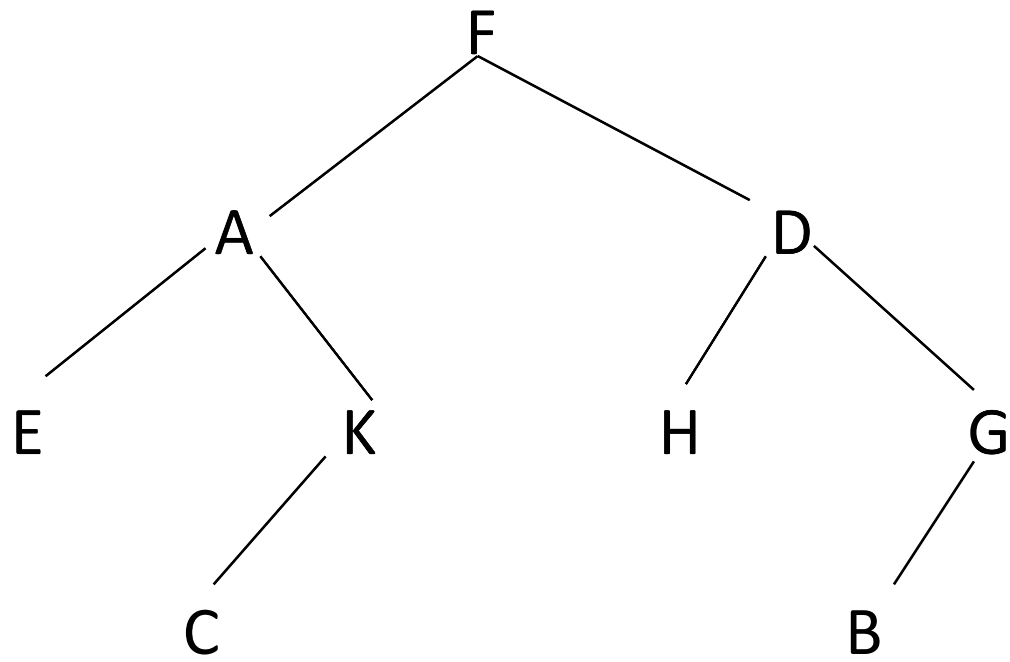
As K is to the left of C in preorder

- Creating the right subtree of F
- The root node is D
- From inorder, the nodes on the left of D are: **H** (left root of D)

  the nodes on the right of D are: B **G (right root of D)**

Thus the tree is:

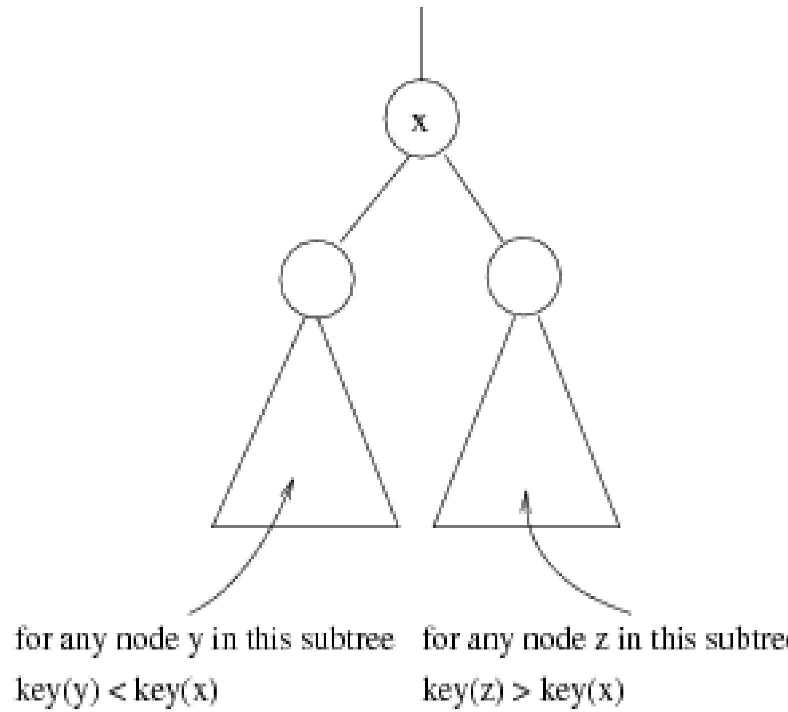



F
A
D
E
K
H
G
C
B

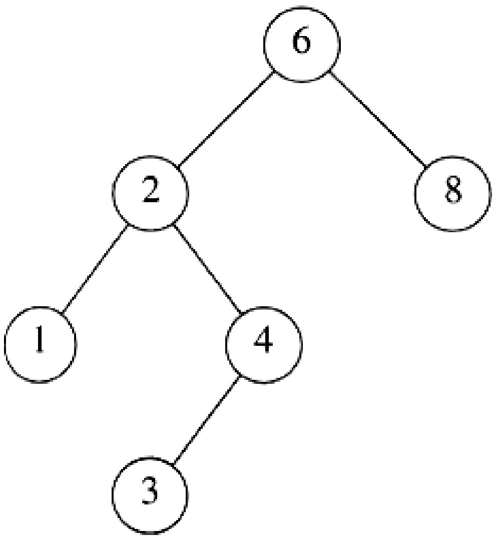
- Ex:
- Draw the tree :
  - Preorder: ABDGCEHIF
  - Inorder: DGBAHEICF
  - PostOrder: GDBHIEFCA

Binary Search Trees (BST)

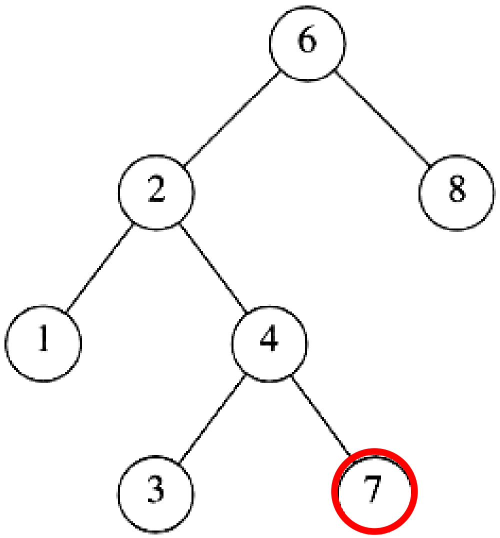
- A data structure for efficient searching, insertion and deletion.
- Binary search tree property
  - For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X



Binary Search Trees



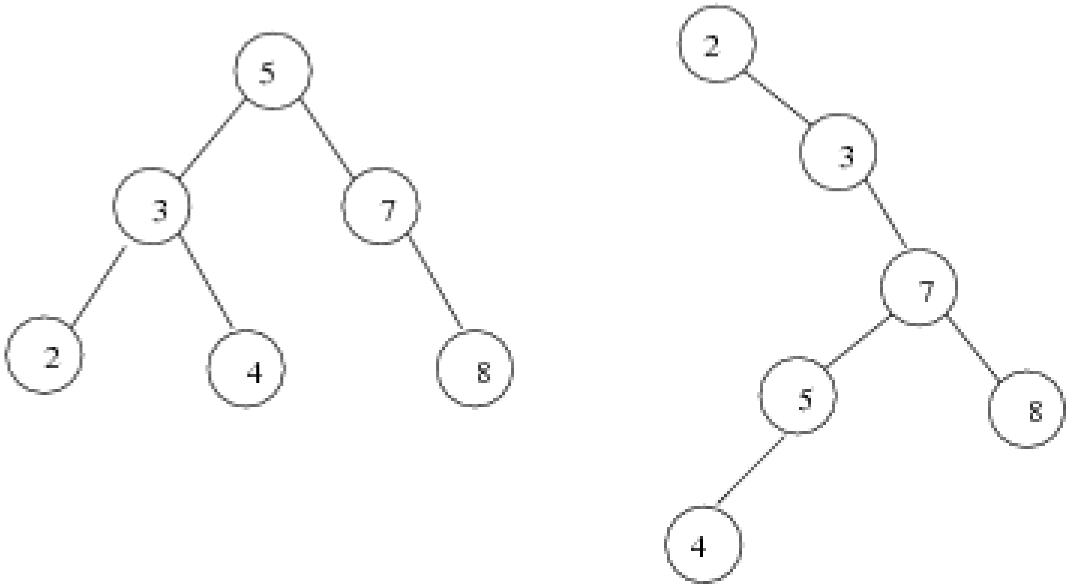
A binary search tree



Not a binary search tree

Binary Search Trees

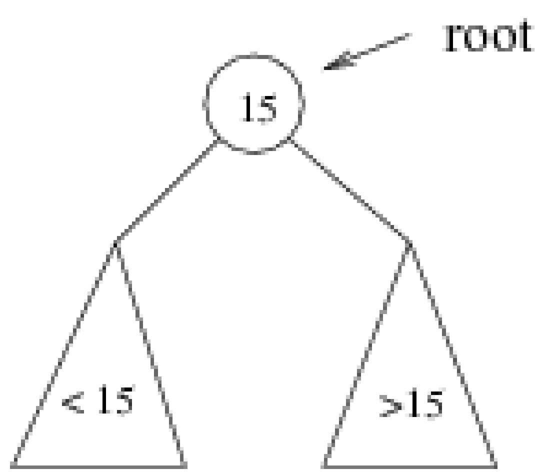
The same set of keys may have different BSTs



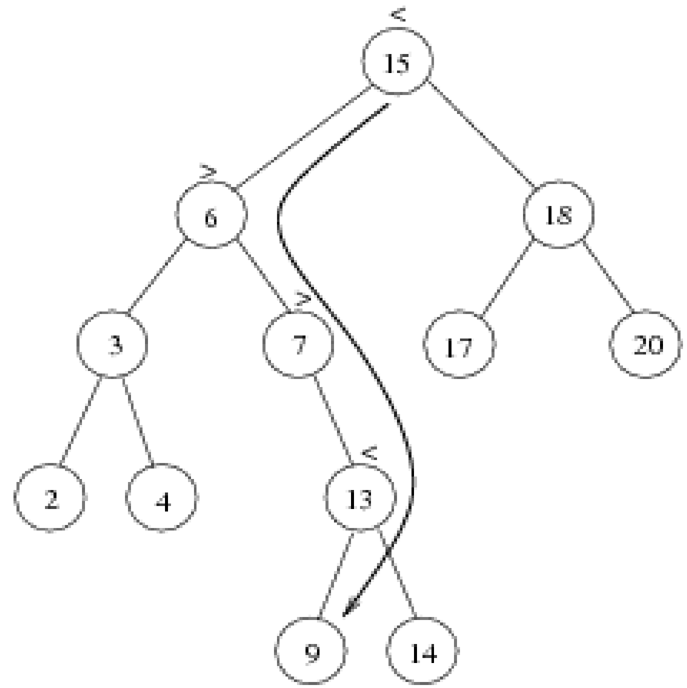
- Average depth of a tree is $O(\log N)$
- Maximum depth of a tree is $O(N)$

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left sub-tree.
- If we are searching for a key > 15, then we should search in the right sub-tree.



Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

BST(searching)

FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

1) If ROOT == NULL then: [tree empty]
   Set LOC= NULL and PAR = NULL and return
2) If ITEM == ROOT →info , then: [item at root]
   Set LOC = ROOT and PAR = NULL and return
3) If ITEM < ROOT →info , then: [intialize ptr and save]
   Set PTR = ROOT →left and SAVE = ROOT
   Else:
   Set PTR = ROOT →right and SAVE = ROOT
   [End of if statement]
4) **Repeat step 5 and 6 while PTR≠NULL**
5) If ITEM == PTR →info then: [item found]
   Set LOC = PTR and PAR = SAVE and return
6) If ITEM < PTR →info ,then:
   Set SAVE = PTR and PTR = PTR →left
   Else:
   Set SAVE = PTR and PTR = PTR →right
   [End of while loop at step 4]
7) Exit

BST(Insertion)

INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

Complexity of inserting Node in Binary Sear tree in worst case is O(n).
But the worst case complexity in balanced tree (AVL tree) is O(logn).

1) Call **FIND(INFO,LEFT,RIGHT,ROOT, ITEM,LOC,PAR)**
2) a) if AVAIL==NULL then :
   write OVERFLOW and exit
   b) Set NEW = AVAIL, AVAIL = AVAIL→left and NEW →info = ITEM
   c) NEW →left = NULL and NEW →right = NULL
3) If PAR==NULL then: **[If tree is empty]**
   Set ROOT = NEW
   else if: ITEM < PAR →info then: **[If element < parent]**
   Set PAR →left = NEW
   else : **[If element > parent]**
   Set PAR →right = NEW
4) exit

BST (Deletion)

CaseA (Info, Left, Right, Root, Loc, Par)

This procedure deletes node N at location Loc, where N doesn't have 2 children. Par=Null → N is root node; Child=Null → N has no child

1) **[Initilizes Child]**

If Loc→left == Null and Loc→right == Null, then

Set Child = Null

Else if   Loc→left ≠ Null, then

Set Child = Loc→left

Else

Set Child = Loc→right

[End of if structure]

2) If Par ≠ Null, then

If Loc = Par→left, then   Set Par→left = Child

Else Set Par→right = Child

[End of if structure]

Else   Set Root = Child

[End of if structure]

3) Return

BST (Deletion)

CaseB(Info, Left, Right, Root, Loc, Par)

This procedure deletes node N at location Loc, where N has 2 children. Par=Null → N is root node; Suc → inorder successor, ParSuc → Parent of inorder successor

1) [Find Suc and ParSuc]

a) Set Ptr = Loc→right and Save = Loc

b) Repeat while Ptr→left ≠ Null

Set Save = Ptr and Ptr = Ptr→left

c) Set Suc = Ptr and ParSuc = Save

2) [Delete inorder successor using Procedure of CaseA]

Call **CaseA (Info, Left, Right, Root, Suc, ParSuc)**

3) [Replace Node N by its inorder successor]

a) If Par ≠ Null, then

If Loc == Par→left, then Set Par→left = Suc

Else Set Par→right = Suc

[End of if structure]

Else Set Root = Suc

[End of if structure]

b) Set Suc→left = Loc→left and Suc→right = Loc→right

4) Return

Algorithm: **DEL(INFO, LEFT,RIGHT,ROOT,AVAIL,ITEM)**

This procedure deletes ITEM from the tree.

1) **[Find the location of ITEM and it's parent]**

   Call FIND(INFO,LEFT,RIGHT,ROOT,ITEM,LOC,PAR)

2) **[ITEM in tree?]**

   If LOC == NULL, then:

   write: ITEM not in tree, and exit

   [End of if structure]

3) **[Delete node containing ITEM]**

   If LOC→right ≠ NULL and LOC →left ≠ NULL, then:

   Call CaseB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

   Else:

   Call CaseA(INFO, LEFT,RIGHT,ROOT,LOC,PAR)

   [End of if structure]

4) **[Return deleted node to the AVAIL list]**

   Set LOC→right = AVAIL and AVAIL = LOC

5) Exit.

Deletion in Binary Search Tree

```
struct node* deleteNode(struct node* root, int key)
{
   // base case
   if (root == NULL) return root;
   // If the key to be deleted is smaller than the root's key,
   // then it lies in left subtree
   if (key < root->key)
      root->left = deleteNode(root->left, key);

   // If the key to be deleted is greater than the root's key,
   // then it lies in right subtree
   else if (key > root->key)
      root->right = deleteNode(root->right, key);
   // if key is same as root's key, then This is the node
   // to be deleted
   else
   {
```

```
// node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

Application of BST

- Storing a set of names, and being able to lookup based on a prefix of the name. (Used in internet routers.)