UNIT IV
Trees

Introduction Terminology
Representation of trees,
Binary trees abstract data type
Properties of binary trees
Binary tree representation
Binary tree traversals: In order, preorder, post order
Binary search trees Definition
Operations:searching BST, insert into BST, delete from a BST, Height of a BST.

Trees: Non-Linear data structure

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represent hierarchical relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

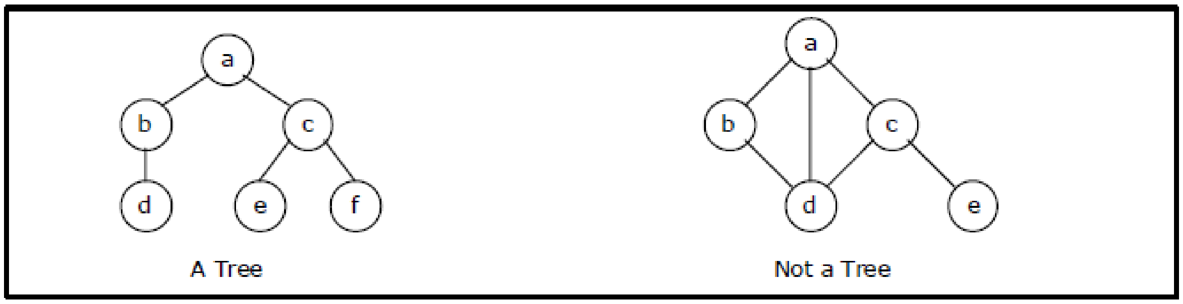


Figure 5.1.1 A Tree and a not a tree

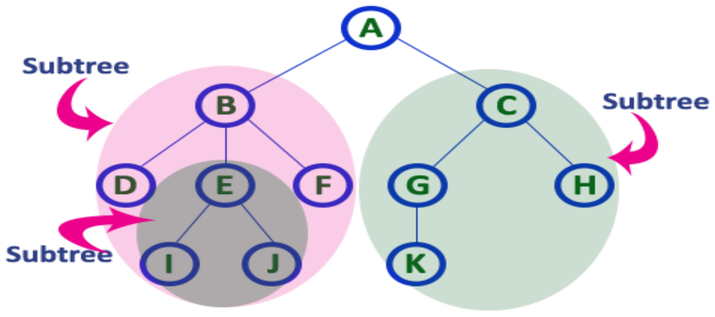
Tree is a popular data structure used in wide range of applications. A tree data structure can be defined as follows...

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

A tree data structure can also be defined as follows...

A tree is a finite set of one or more nodes such that:

There is a specially designated node called the root. The remaining nodes are partitioned into n>=0 disjoint sets T1, ..., Tn, where each of these sets is a tree. We call T1, ..., Tn are the subtrees of the root.



A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

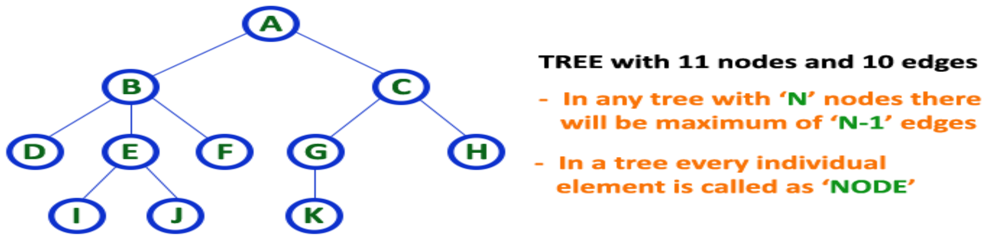
Advantages of trees

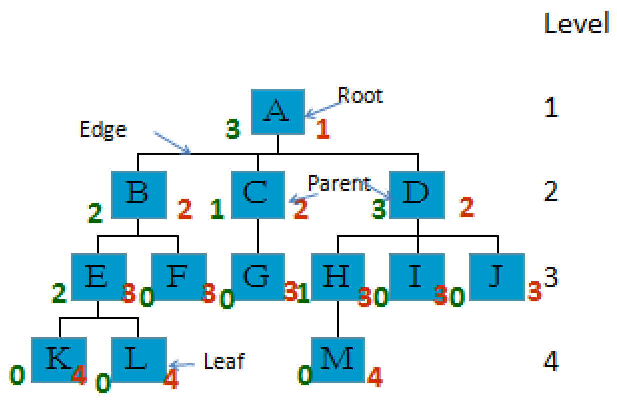
Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move sub trees around with minimum effort

Introduction Terminology

In a Tree, Every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. Example


A
B
C
D
E
F
G
H
I
J
K
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'



1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree. In above tree, **A** is a **Root** node

2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as PARENT NODE. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children". e.g., Parent (A,B,C,D).

4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes. e.g., Children of D are (H, I,J).

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with same parent are called as Sibling nodes. Ex: Siblings (B,C, D)

6. Leaf

In a tree data structure, the node which does not have a child (or) node with degree zero is called as LEAF Node. In simple words, a leaf is a node with no child.

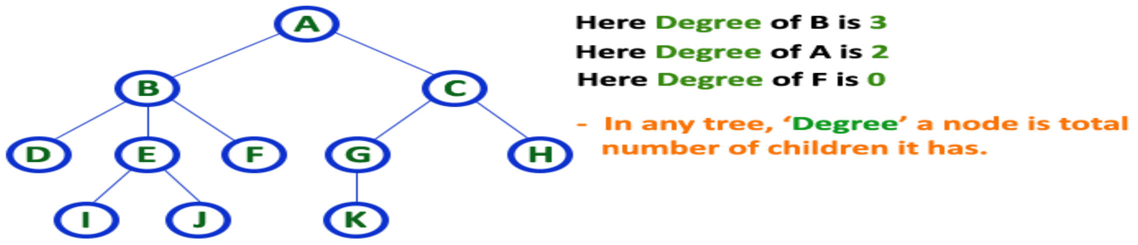
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node. Ex: (K,L,F,G,M,I,J)

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.
In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes. Ex:B,C,D,E,H

8. Degree

In a tree data structure, the total number of children of a node (or)number of subtrees of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step). Some authors start root level with 1.

10. Height

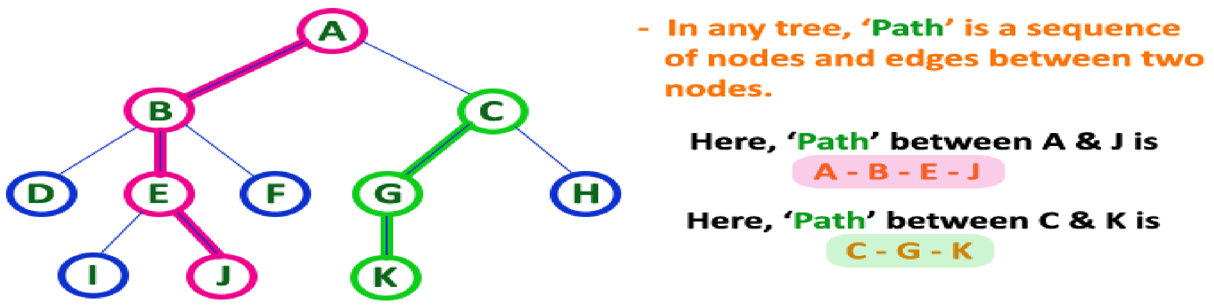
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

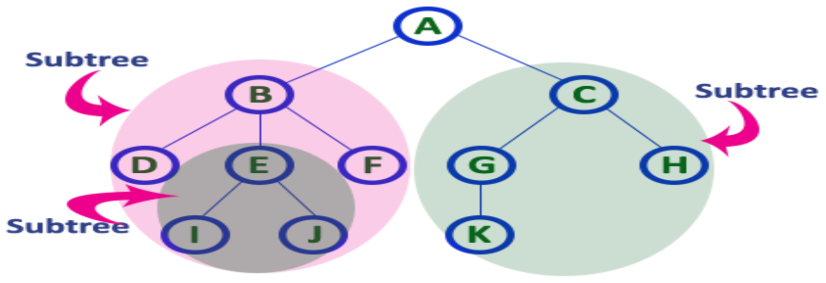
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.


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

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



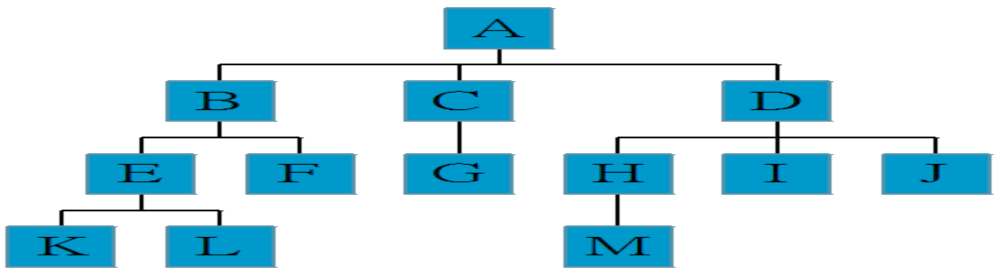
Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1.List Representation

2. Left Child - Right Sibling Representation

Consider the following tree...



1. List Representation

In this representation, we use two types of nodes one for representing the node with data and another for representing only references. We start with a node with data from root node in the tree. Then it is linked to an internal node through a reference node and is linked to any other node directly. This process repeats for all the nodes in the tree.

The above tree example can be represented using List representation as follows...

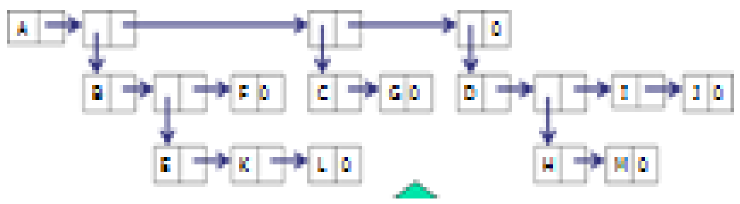
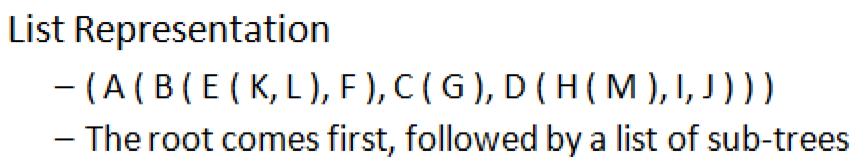


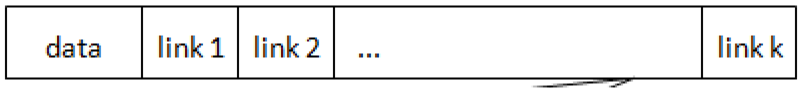
Fig: List representation of above Tree



List Representation

– (A (B (E (K, L), F), C (G), D (H (M), I, J)))

– The root comes first, followed by a list of sub-trees

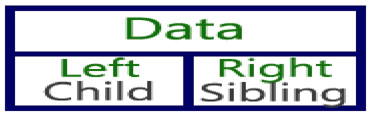


data	link 1	link 2	...	link k

Fig: Possible node structure for a tree of degree k

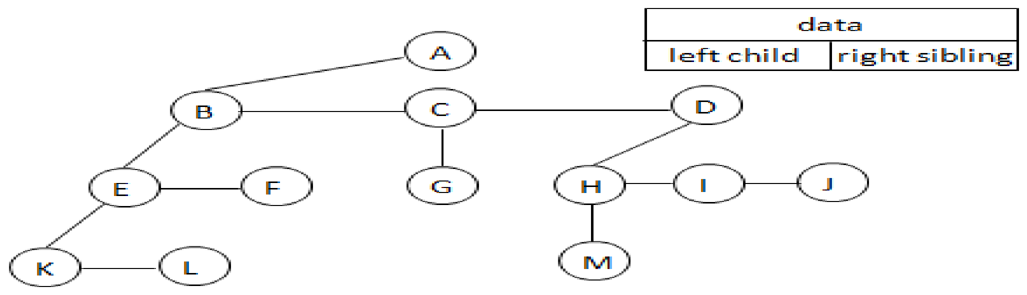
2. Left Child - Right Sibling Representation

In this representation, we use list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise that field stores NULL.

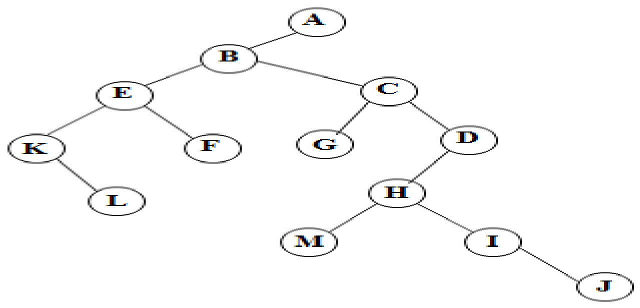
The above tree example can be represented using Left Child - Right Sibling representation as follows...



Representation as a Degree –Two Tree

To obtain degree-two tree representation of a tree, rotate the right- sibling pointers in the left child-right sibling tree clockwise by 45 degrees. In a degree-two representation, the two children of anode are referred as left and right children.

*Figure 5.6: Left child-right child tree representation of a tree (p.191)



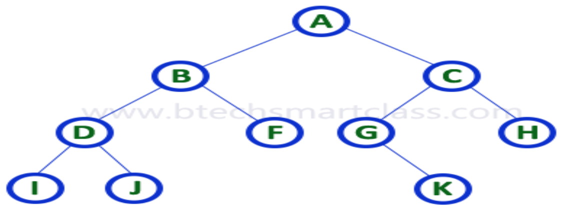
Binary Trees

Introduction

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

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

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children. Example



There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

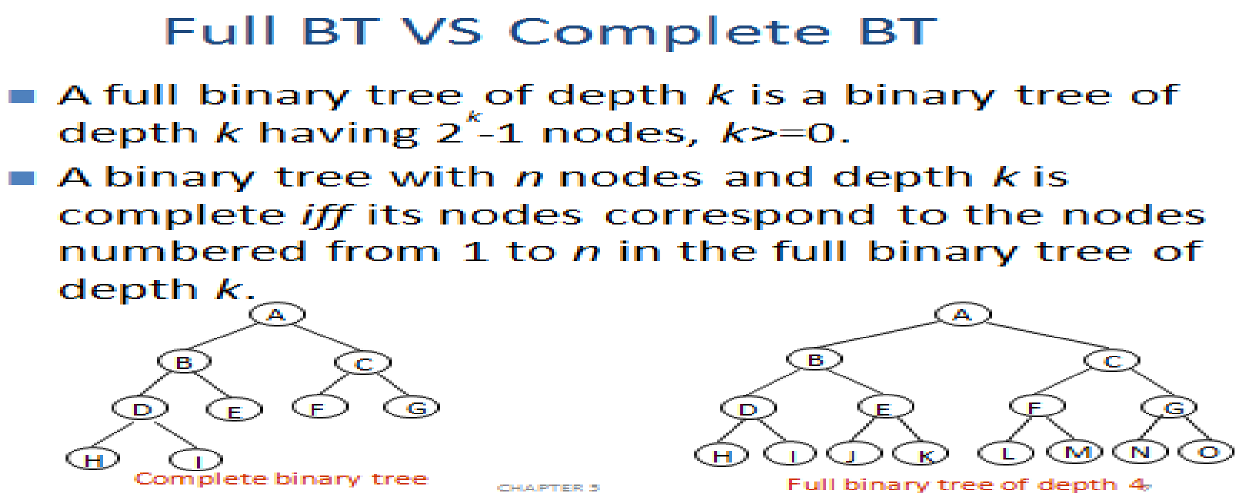
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree

2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

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



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*.

3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

Abstract Data Type

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called left subtree and right subtree.

ADT contains specification for the binary tree ADT.

Structure *Binary_Tree*(abbreviated *BinTree*) is

objects: a finite set of nodes either empty or consisting of a root node, left *Binary_Tree*, and right *Binary_Tree*.

Functions:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

Bintree Create()::= creates an empty binary tree

Boolean IsEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

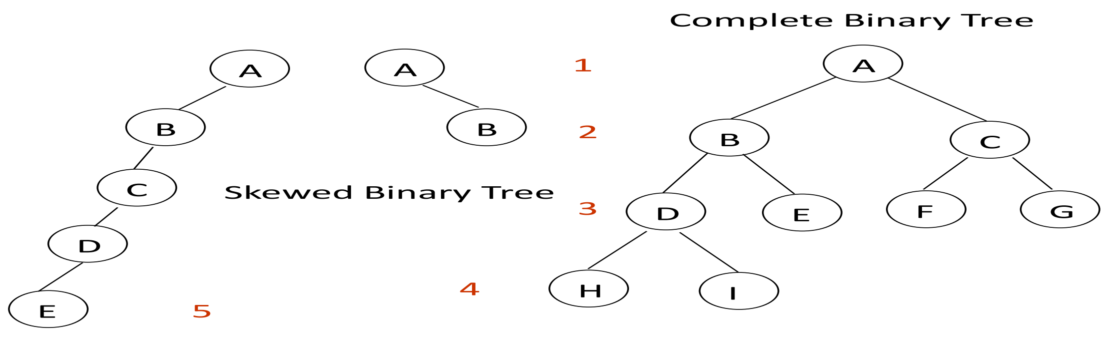
BinTree MakeBT(*bt1*, *item*, *bt2*)::= return a binary tree whose left subtree is *bt1*, whose right subtree is *bt2*, and whose root node contains the data *item*

Bintree Lchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the left subtree of *bt*

element Data(*bt*)::= if (IsEmpty(*bt*)) return error else return the data in the root node of *bt*

Bintree Rchild(*bt*)::= if (IsEmpty(*bt*)) return error else return the right subtree of *bt*

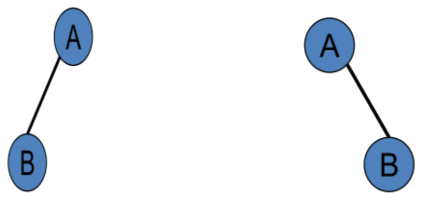
Samples of Trees



CHAPTER 5 10

Differences between A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered.



Above two trees are different when viewed as binary trees. But same when viewed as trees.

Properties of Binary Trees

1.Maximum Number of Nodes in BT

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

Proof By Induction:

Induction Base: The root is the only node on level i=1.Hence ,the maximum number of nodes on level i=1 is $2^{i-1}=2^0=1$.

Induction Hypothesis: Let I be an arbitrary positive integer greater than 1.Assume that maximum number of nodes on level i-1 is 2^{i-2}.

Induction Step: The maximum number of nodes on level i-1 is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2,the maximum number of nodes on level i is two times the maximum number of nodes on level i-1,or 2^{i-1}.

The maximum number of nodes in a binary tree of depth k is $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$

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 the number of nodes of degree 2, then $n_0=n_2+1$.

PROOF: Let *n* and *B* denote the total number of nodes and branches in *T*. Let n_0, n_1, n_2 represent the nodes with zero children, single child, and two children respectively.

$$B+1=n \rightarrow B=n_1+2n_2 \Longrightarrow n_1+2n_2+1=n,$$

$$n_1+2n_2+1=n_0+n_1+n_2 \Longrightarrow n_0=n_2+1$$

3. A *full binary tree* of depth *k* is a binary tree of depth *k* having 2 -1 nodes, *k*>=0.

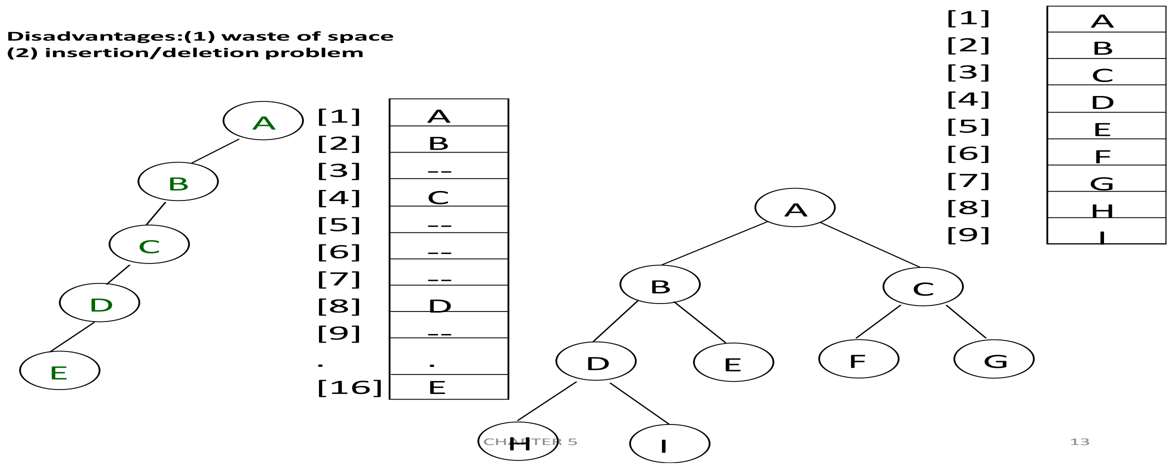
A binary tree with *n* nodes and depth *k* is *complete iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k.*

Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are 1)Array Representation 2)Linked List Representation

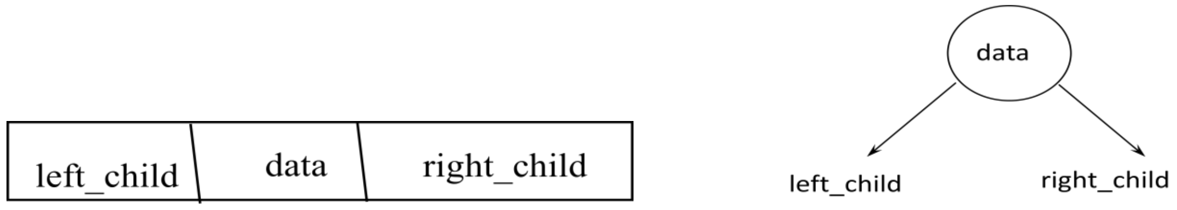
1)Array Representation: In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of

A complete binary tree with *n* nodes (depth = log *n* + 1) is represented sequentially, then for any node with index *i*, 1<=*i*<=*n*, we have: a) *parent*(*i*) is at *i*/2 if *i*!=1. If *i*=1, *i* is at the root and has no parent. b)*left_child*(*i*) ia at 2*i* if 2*i*<=*n*. If 2*i*>n, then *i* has no left child. c) *right_child*(*i*) is at 2*i*+1 if 2*i* +1 <=*n*. If 2*i* +1 >n, then *i* has no right child.



2. Linked Representation

We use linked list to represent a binary tree. In a linked list, every node consists of three fields. First field, for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...

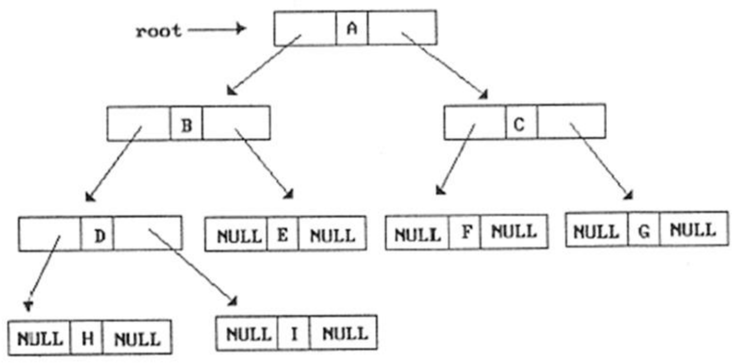


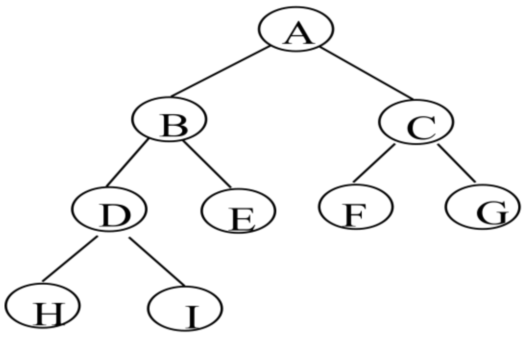
```
typedef struct node *tree_pointer;

typedef struct node {

 int data;

 tree_pointer left_child, right_child;};
```



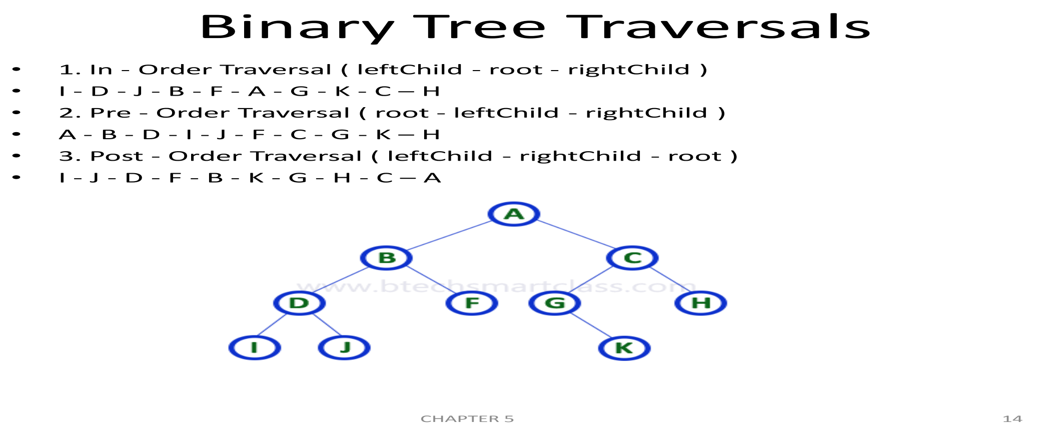


Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1)In - Order Traversal 2)Pre - Order Traversal 3)Post - Order Traversal



Binary Tree Traversals

- 1. In - Order Traversal (leftChild - root - rightChild)
- I - D - J - B - F - A - G - K - C – H
- 2. Pre - Order Traversal (root - leftChild - rightChild)
- A - B - D - I - J - F - C - G - K – H
- 3. Post - Order Traversal (leftChild - rightChild - root)
- I - J - D - F - B - K - G - H - C – A

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I'then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C – H

Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

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

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root

for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.
That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Algorithm

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

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

3. Post - Order Traversal (leftChild - rightChild - root)

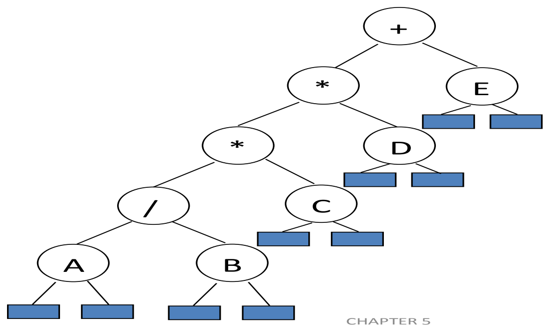
In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.
Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using 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.

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

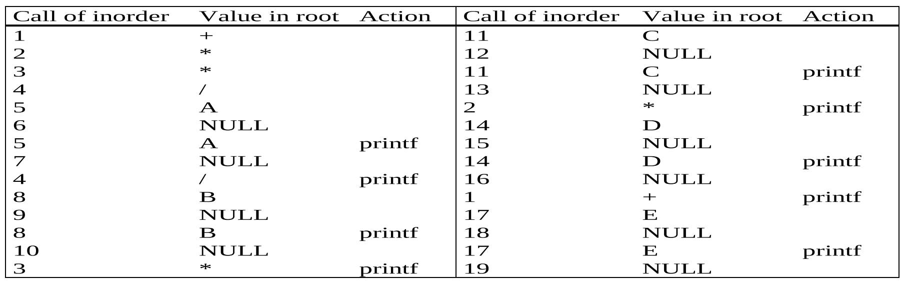
Arithmetic Expression Using BT



inorder traversal
A / B * C * D + E
infix expression
preorder traversal
+ * * / A B C D E
prefix expression
postorder traversal
A B / C * D * E +
postfix expression
level order traversal
+ * E * D / C A B

CHAPTER 5 15

Trace Operations of Inorder Traversal



Call of inorder	Value in root	Action	Call of inorder	Value in root	Action
1	+		11	C	
2	*		12	NULL	
3	*		11	C	printf
4	/		13	NULL	
5	A		2	*	printf
6	NULL		14	D	
5	A	printf	15	NULL	
7	NULL		14	D	printf
4	/	printf	16	NULL	
8	B		1	+	printf
9	NULL		17	E	
8	B	printf	18	NULL	
10	NULL		17	E	printf
3	*	printf	19	NULL	

CHAPTER 5 16

Iterative Inorder Traversal (using stack)

```
void iter_inorder(tree_pointer node)
{
 int top= -1;        /* initialize stack */
 tree_pointer stack[MAX_STACK_SIZE];
 for (;;) {
  for (; node; node=node->left_child)
    add(&top, node);          /* add to stack */
  node= delete(&top);        /* delete from stack */
  if (!node) break;               /* empty stack */
  printf("%D", node->data);
  node = node->right_child;
 }
}
```

In iterative inorder traversal, we must create our own stack to add and remove nodes as in recursion.

Analysis: Let n be number of nodes in tree, every node of tree is placed on and removed from the stack exactly once. So time complexity is O(n). The space requirement is equal to the depth of tree which is O(n).

Level Order Traversal (Using Queue) -- Traversal without Stack

```
void level_order(tree_pointer ptr)          /* level order tree traversal */
{
 int front = rear = 0;
 tree_pointer queue[MAX_QUEUE_SIZE];
 if (!ptr) return; /* empty queue */
 addq(front, &rear, ptr);
 for (;;) {
  ptr = deleteq(&front, rear);
if (ptr) {
    printf("%d", ptr->data);
    if (ptr->left_child)
     addq(front, &rear, ptr->left_child);
    if (ptr->right_child)
     addq(front, &rear, ptr->right_child);
  }
  else break;
 }       }
```

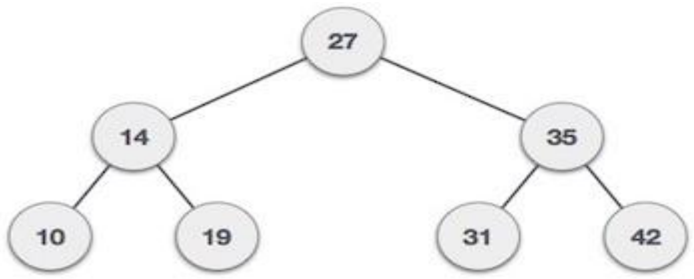
Level order Traversal is implemented with circular queue. In this order, we visit the root first, then root's left child followed by root's right child. We continue in this manner, visiting the nodes at each new level from left most to right most nodes.

We begin by adding root to the queue. The function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. The level order traversal for above arithmetic expression is + * E * D / C A B.

Binary Search Trees

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.



We're going to implement tree using node object and connecting them through references.

Definition: A binary search tree (BST) is a binary tree. It may be empty. If it is not empty,then all nodes follows the below mentioned properties −

- Every element has a unique key.
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The keys in a nonempty right subtree larger than the key in the root of subtree.
- The left and right subtrees are also binary search trees.

left sub-tree and right sub-tree and can be defined as −

left_subtree (keys) $\leq$ node (key) $\leq$ right_subtree (keys)

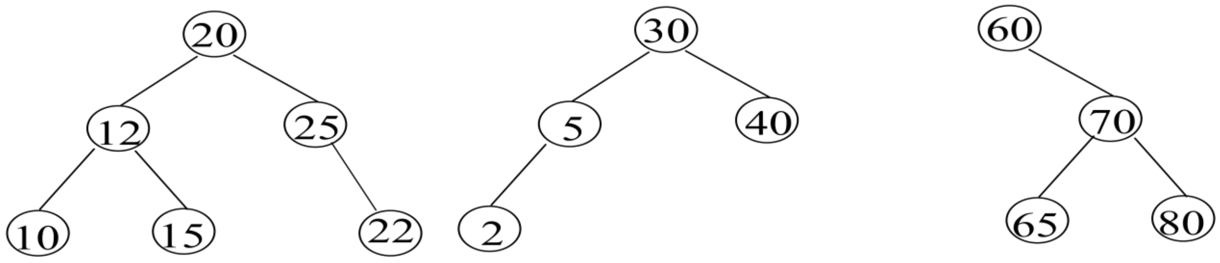


Fig: Example Binary Search Trees

ADT for Dictionary:

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following −

- **Search** − search an element in a binary search tree.
- **Insert** − insert an element into a binary search tree / create a tree.
- **Delete** − Delete an element from a binary search tree.
- **Height** -- Height of a binary search tree.

Searching a Binary Search Tree

Let an element k is to search in binary search tree. Start search from root node of the search tree. If root is NULL, search tree contains no nodes and search unsuccessful. Otherwise, compare k with the key in the root. If k equals the root's key, terminate search, if k is less than key value, search element k in left subtree otherwise search element k in right subtree. The function search recursively searches the subtrees.

Algorithm:Recursive search of a Binary Search Tree

```
tree_pointer search(tree_pointer root,  int key)
{
/* return a pointer to the node that contains key. If there is no such
node, return NULL */
  if (!root) return NULL;
  if (key == root->data) return root;
  if (key < root->data)
      return search(root->left_child,  key);
  return search(root->right_child,key);
}
```

Algorithm: Iteraive search of a Binary Search Tree

```
tree_pointer search2(tree_pointer tree, int key)
{
  while (tree) {
    if (key == tree->data) return tree;
    if (key < tree->data)
       tree = tree->left_child;
    else tree = tree->right_child;
  }
return NULL;
}
```

Analysis of Recursive search and Iterative Search Algorithms:
If h is the height of the binary search tree, both algorithms perform search in O(h) time. Recursive search requires additional stack space which is O(h).

Inserting into a Binary Search Tree

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left sub tree and insert the data. Otherwise search empty location in right sub tree and insert the data.

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

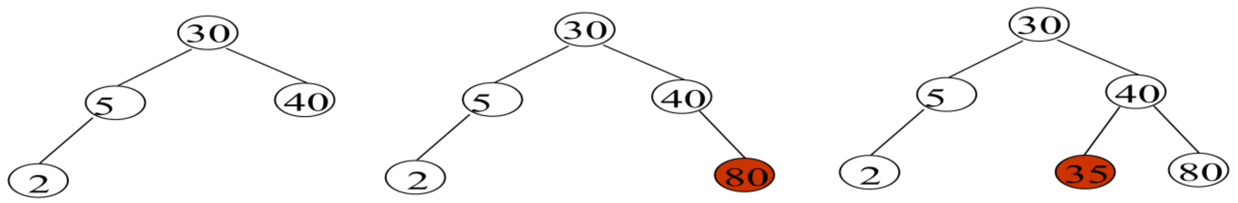
Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above step until we reach a node (e.i., reach to NULL) where search terminates.

Step 7: After reaching a last node, then insert the newNode as left child if newNode is smaller or equal to that node else insert it as right child.



Insert 80 Insert 35

Algorithm

```
Create newnode
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 newnode
end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;
  //if tree is empty, create root node
  if(root == NULL) {
    root = tempNode;
  }else {
    current = root;
    parent  = NULL;
    while(1) {
      parent = current;
      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
```

```
            return;           }
        }


        //go to right of the tree
        else {
          current = current->rightChild;
          //insert to the right
          if(current == NULL) {
            parent->rightChild = tempNode;
            return;
          }
        }     }
    }
  }
```

Deleting a node

Remove operation on binary search tree is more complicated, than insert and search. Basically, in can be divided into two stages:

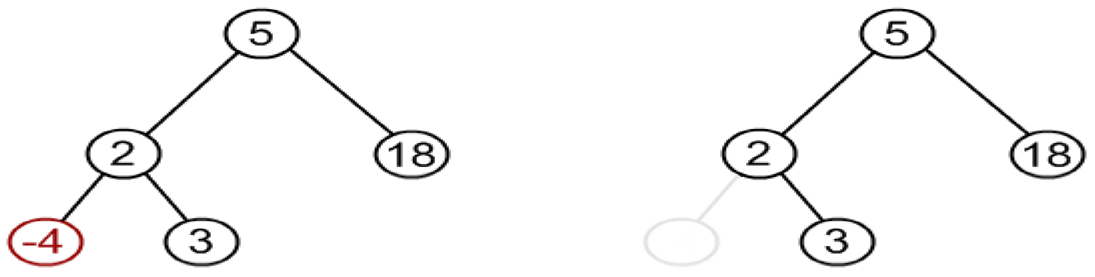
- search for a node to remove
- if the node is found, run remove algorithm.

Remove algorithm in detail

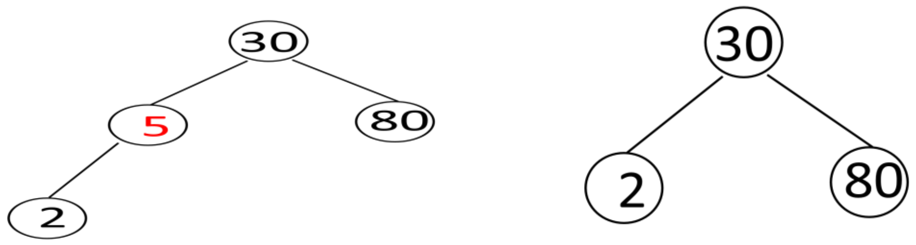
Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below**.**

1.Node to be removed has no children. --This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

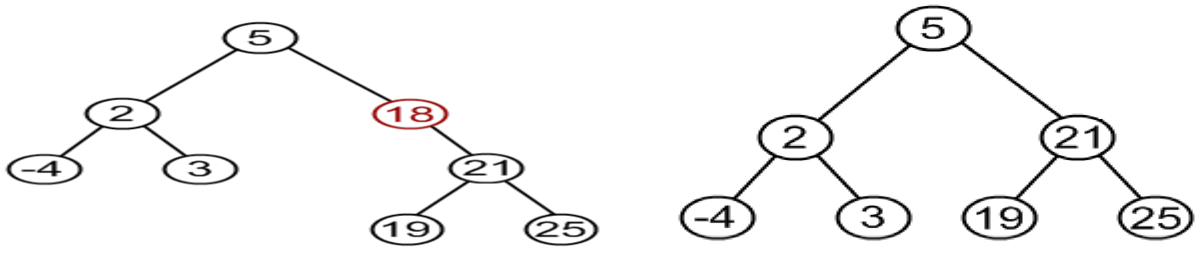
Example. Remove -4 from a BST.



2.Node to be removed has one child. In this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.



3.Node to be removed has two children. --This is the most complex case. The deleted node can be replaced by either largest key in its left subtree or the smallest in its right subtree. Preferably which node has one child.



Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above step.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto steps 2

Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

```
/* deletion in binary search tree */
  void deletion(struct treeNode **node, struct treeNode **parent, int data) {
        struct treeNode *tmpNode, *tmpParent;
        if (*node == NULL)
                return;
        if ((*node)->data == data) {
                /* deleting the leaf node */
                if (!(*node)->left && !(*node)->right) {
                        if (parent) {
                                /* delete leaf node */
                                if ((*parent)->left == *node)
                                        (*parent)->left = NULL;
                                else
                                        (*parent)->right = NULL;
                                free(*node);
                        } else {
                                /* delete root node with no children */
                                free(*node);
                        }
                /* deleting node with one child */
                } else if (!(*node)->right && (*node)->left) {
                        /* deleting node with left child alone */
                        tmpNode = *node;
                        (*parent)->right = (*node)->left;
                        free(tmpNode);
                        *node = (*parent)->right;
                } else if ((*node)->right && !(*node)->left) {
                        /* deleting node with right child alone */
                        tmpNode = *node;
                        (*parent)->left = (*node)->right;
                        free(tmpNode);
                        (*node) = (*parent)->left;
                } else if (!(*node)->right->left) {
                        /*
                         * deleting a node whose right child
                         * is the smallest node in the right
                         * subtree for the node to be deleted.
                         */

                        tmpNode = *node;

                        (*node)->right->left = (*node)->left;

                        (*parent)->left = (*node)->right;
                        free(tmpNode);
                        *node = (*parent)->left;
                } else {
                        /*
                         * Deleting a node with two children.
                         * First, find the smallest node in
                         * the right subtree.  Replace the
                         * smallest node with the node to be
                         * deleted. Then, do proper connections
                         * for the children of replaced node.
                         */
                        tmpNode = (*node)->right;
                        while (tmpNode->left) {
```

```
                        tmpParent = tmpNode;
                        tmpNode = tmpNode->left;
                  }
                  tmpParent->left = tmpNode->right;
                  tmpNode->left = (*node)->left;
                  tmpNode->right =(*node)->right;
                  free(*node);
                  *node = tmpNode;
            }
      } else if (data < (*node)->data) {
            /* traverse towards left subtree */
            deletion(&(*node)->left, node, data);
      } else if (data > (*node)->data) {
            /* traversing towards right subtree */
            deletion(&(*node)->right, node, data);
      }
}
```

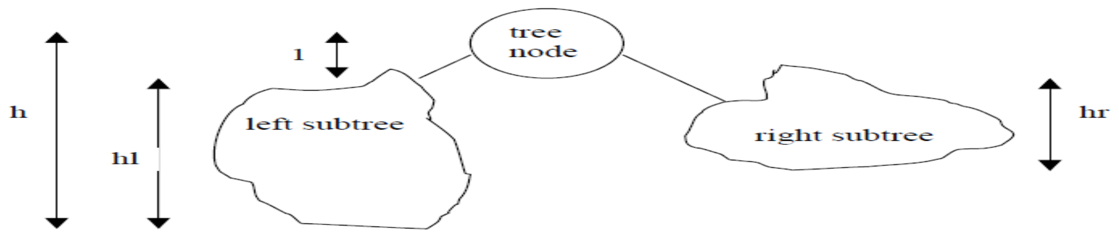
Height of a Binary Search Tree:

Height of a Binary Tree For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of –1.

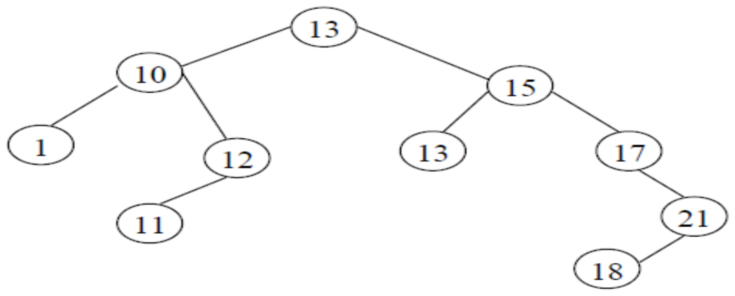
The following height function in pseudocode is defined recursively

```
int height( BinaryTree Node t) {
    if t is a null tree
        return  -1;
    hl = height( left subtree of t);
    hr = height( right subtree of t);
    h = 1 + maximum of hl and hr;

    return h;
}
```



For example, the following tree has a height of 4. Its left subtree has height 2 and its right subtree 3.

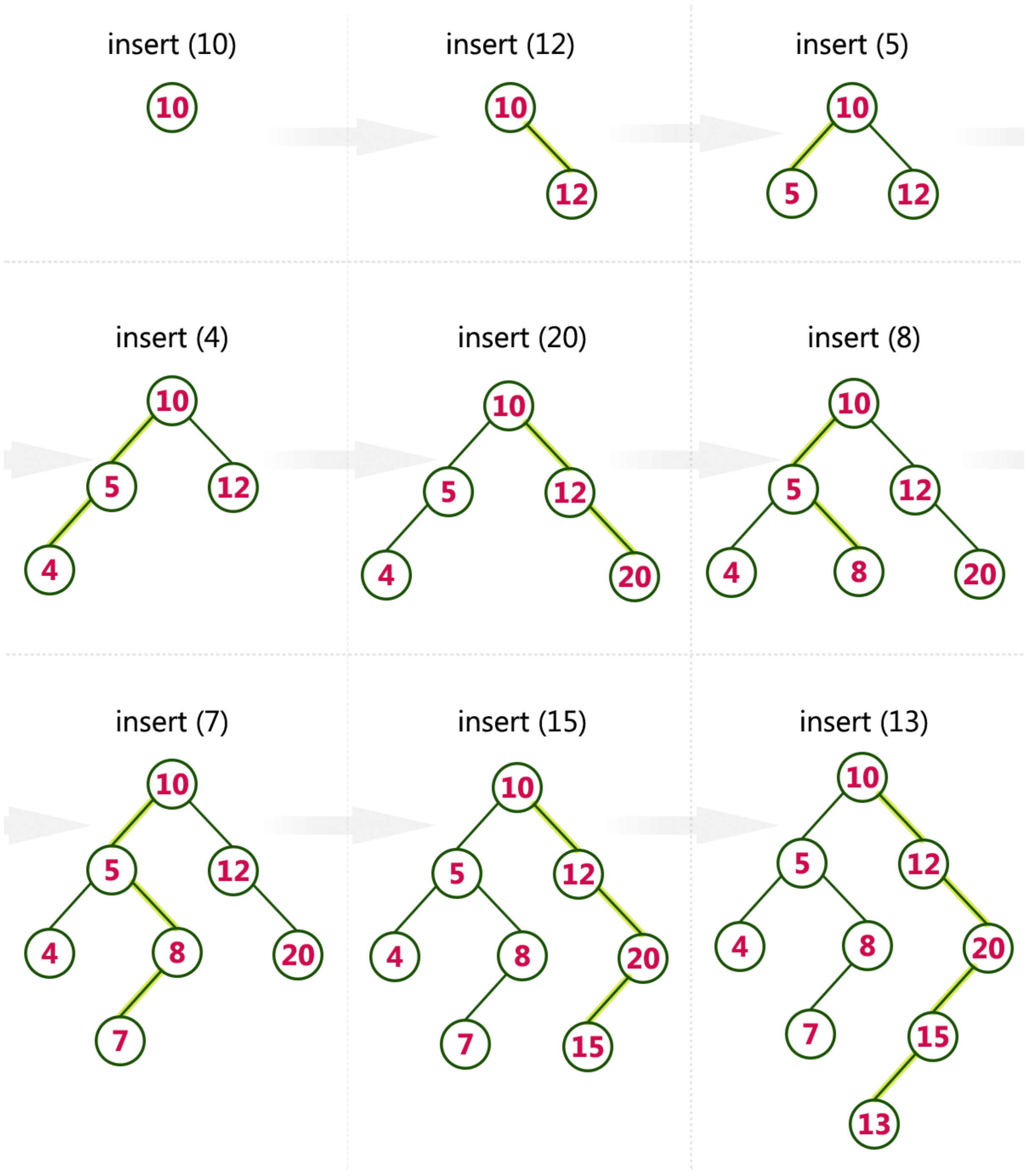


Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...


insert (10)
insert (12)
insert (5)
insert (4)
insert (20)
insert (8)
insert (7)
insert (15)
insert (13)