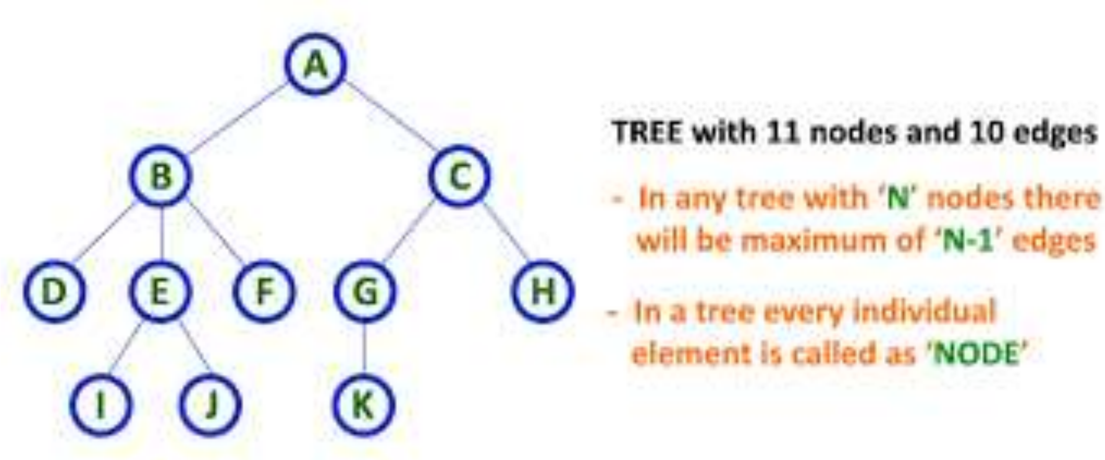
MODULE IV

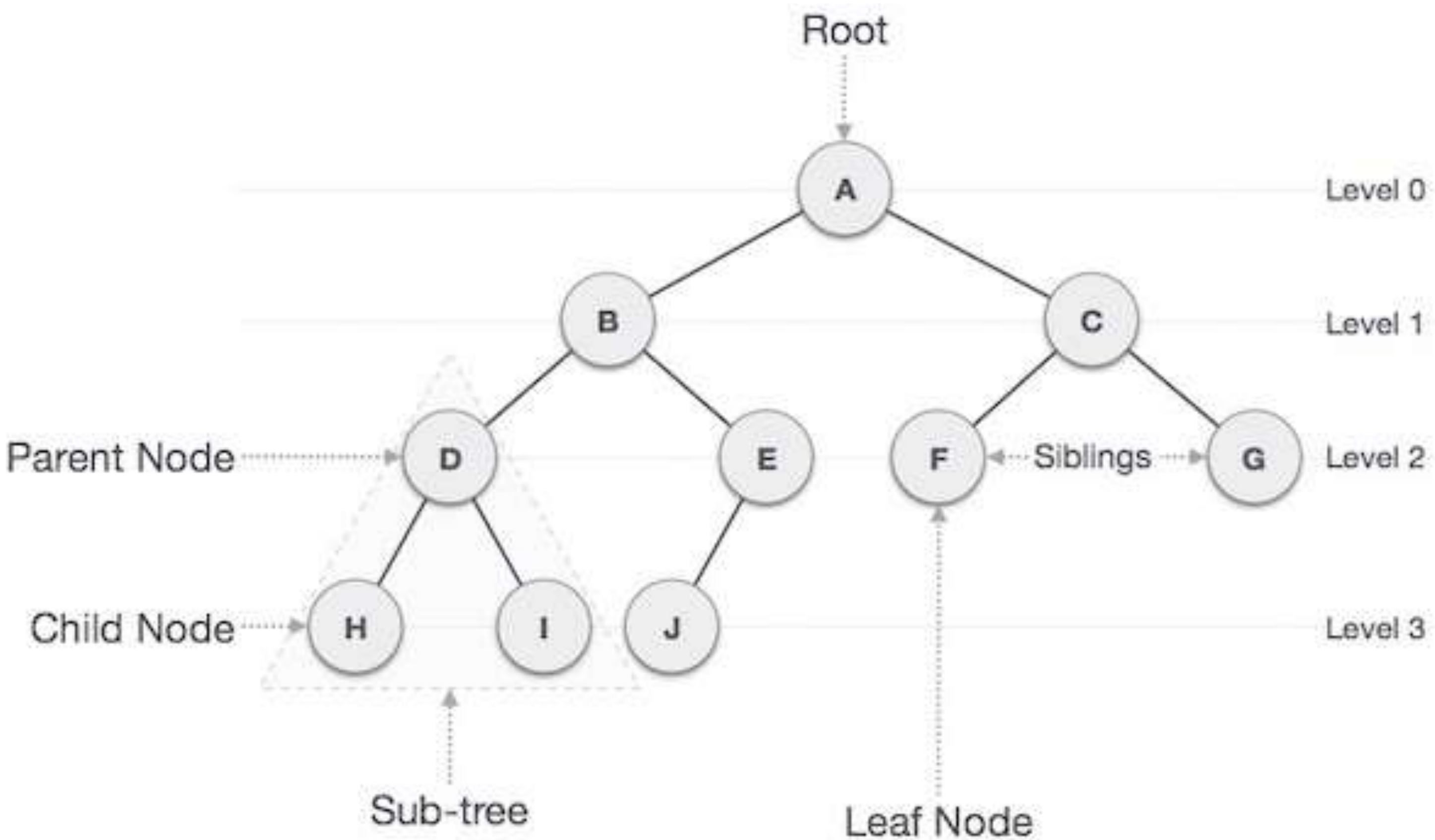
TREES

Trees

- Non-Linear Data Structure
- Requires a two dimensional representation
- Tree is used when a hierarchical relationship among data is to be preserved
- Ancestor/Predecessor – Successor relationship



Trees - Basic Terminologies



Trees - Basic Terminologies

- **Node**
  - This is the main Component of any tree
  - Node stores the actual data and links to other nodes
- **Parent**
  - Parent of a node is the immediate predecessor of a node
- **Child**
  - Child of a node is the immediate Successor of a node
- **Link (also known as edge or branch)**
  - This is a pointer to a node in a tree
  - There may have more than one links from a node
- **Root**
  - Specially designated node – which has no parent

Trees - Basic Terminologies

- **Leaf (also known as terminal nodes)**
  - Node which is at the end of a tree which do not have any child
- **Level** (level of a node)
  - It is the rank in the hierarchy
  - Root has level 0
  - If Parent is at level "L", its child will be at Level "L+1"
- **Height (also known as Depth)**
  - Maximum number of nodes that is possible in a path starting from the root node to a leaf node
  - Height $\boxed{H = L_{max} + 1}$, where L_{max} is the Maximum level

Trees - Basic Terminologies

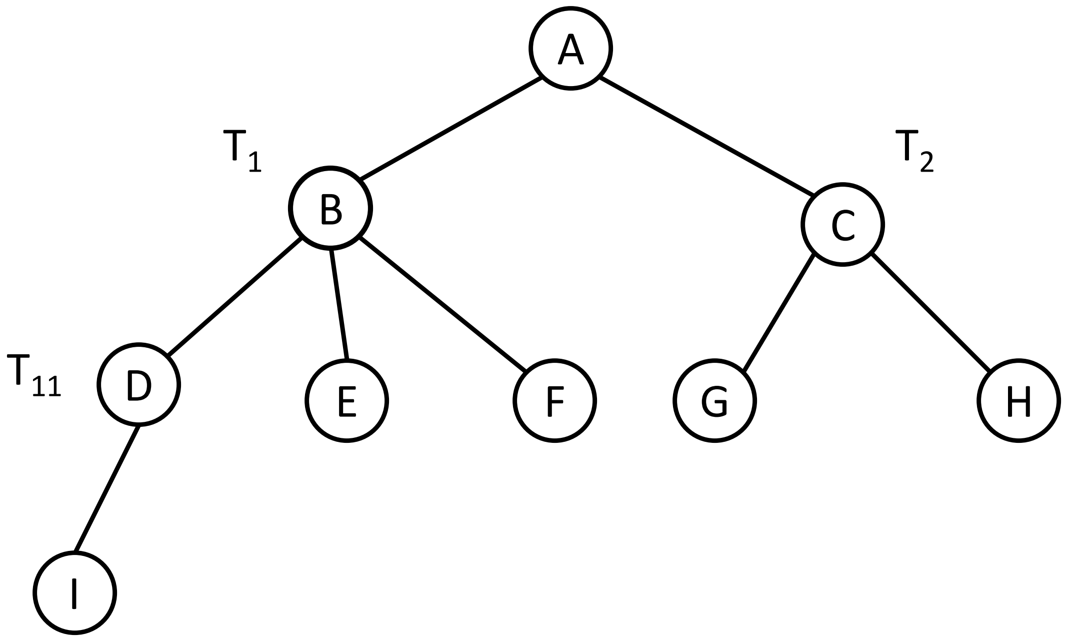
- **Degree** (degree of a node)
  - Maximum number of children that is possible for a node
- **Sibling**
  - Nodes which have same parent
- **Internal and External nodes**
  - Leaf nodes are known as external nodes, other nodes are known as internal nodes

- **There will be <u>only one path</u> from one node to another in a tree**

Tree - definition

- A tree is a finite set of one or more nodes such that
  i. There is a specially designated node called the root
  ii. The remaining nodes are portioned into n disjoined sets $T_1,T_2,\ldots..,T_n$ (n>0) where each T_i (i = 1,2,….,n) is a tree. $T_1,T_2,\ldots..,T_n$ are called the subtrees of the root

String notation of a Tree



$T \rightarrow (A(T_1,T_2))$

$T_1 \rightarrow (B(T_{11},E,F)$

$T_2 \rightarrow (C(G,H))$

$T_{11} \rightarrow (D(I))$

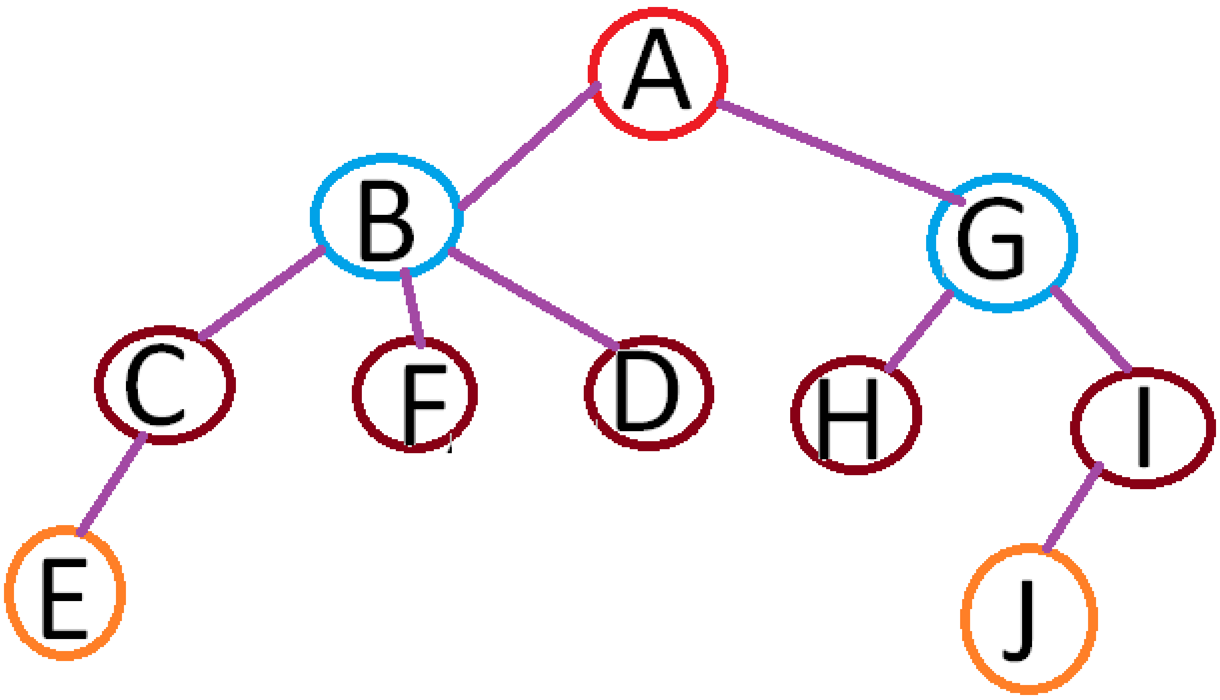
<u>String Notation</u>

$T \rightarrow \mathbf{(}A(B(D(I),E,F),C(G,H))\mathbf{)}$

String notation of a Tree

T-> (A(B(C(E),F,D),G(H,I(J))))

T-> (A(B(C(E),F,D),G(H,I(J))))



Binary Trees

- Is a special form of Tree
- Binary tree T can be defined as a finite set of nodes, such that:
  i. T is empty (called the empty binary tree) or
  ii. T contains a specially designated node called the root of T and the remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called the left subtree and right subtree respectively.
- Each node can have maximum 2 children
  - Left Child and Right Child

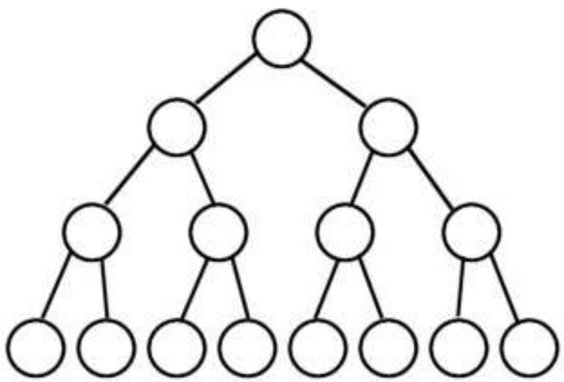
Tree and Binary Tree

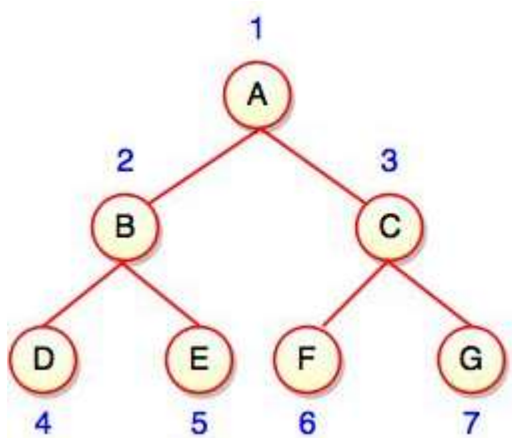
- Tree cannot be empty, where a Binary Tree Can be Empty
- Node in a tree can have Any number of children. In a binary tree a node can have at most two children, so degree of a node will not exceed 2
- Every Binary tree is a tree. But every tree may not be a binary tree.

Binary Tree

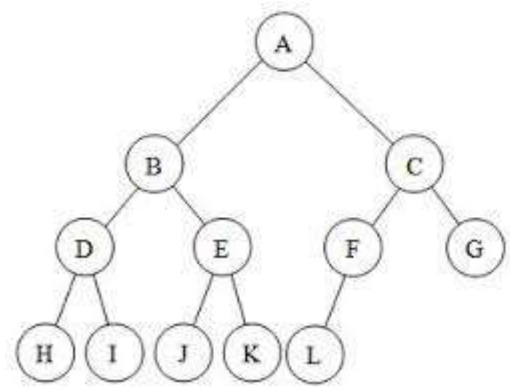
- **Full Binary Tree**
  - A binary tree is a full binary tree if it contains the maximum possible number of nodes at all levels
    - Except leaf nodes - all have two children
- **Complete Binary Tree**
  - A binary tree is a Complete binary tree if all its levels, except possibly the last level have the maximum number of possible nodes.
  - Also **all the nodes in the last level appear as far left as possible**
- **A full binary tree is a complete binary tree. But a complete binary tree may not be a full binary tree always**

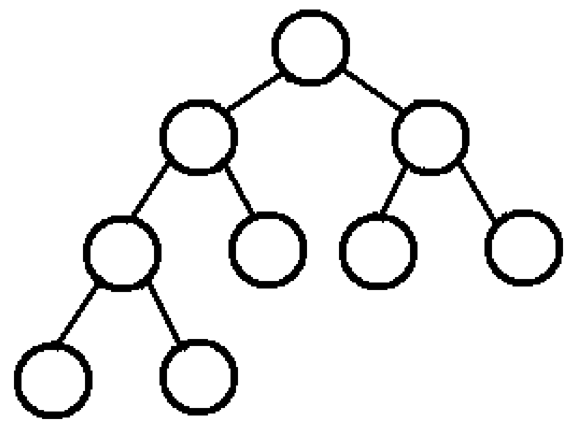
Full Binary Tree





Complete Binary Tree





Binary Tree

- The maximum number of nodes on level *"l"* is 2^l where $l >= 0$
- The maximum number of nodes possible in a binary tree of height *"h"* is $2^h - 1$
- The minimum number of nodes possible in a binary tree of height *"h"* is h
- For any non-empty binary tree, if there are n nodes there will be $n-1$ edges
- For any non-empty binary tree, if n_0 is the number of *leaf nodes* (*degree = 0*) and n_2 is the number of *internal nodes* (*degree =2*), then $n_0 = n_2 + 1$

Binary Tree Representation

- Hierarchical relationship between parent and child should be maintained
- Two approaches
  - **Arrays Representation**
    - Linear or sequential representation
    - Do not require the overhead of maintaining pointers or links
  - **Linked List Representation**
    - Using pointers

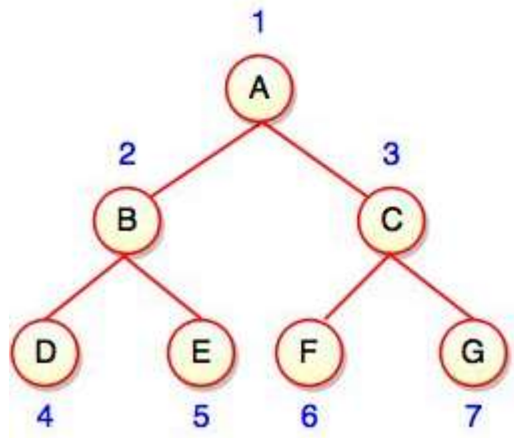
Binary Tree – Array Representation

- Static representation – a block of memory for an array is allocated before storing the actual tree.
- Once allocated, the size of the tree is restricted as permitted by the memory
- Nodes are stored level by level (from Level 0)
- Root node is stored in first memory (index 1)

Binary Tree – Array Representation

- Rules to decide the location of each node in a binary tree
- The root node is at location 1 (Index 1)
- For any node with index i, 1 < i <= n (for some n nodes)
  a) Parent(i) = i/2   // if i = 5 , parent will be in the index 5/2 = 2.5 ≈ 2
     - For the node when i = 1, there is no parent // ie. Root node
  b) Left Child(i) = 2 * i
     - If 2*i > n, then i has no left child
  c) Right Child(i) = (2 * i)+1
     - If 2*i + 1 > n, then i has no Right child

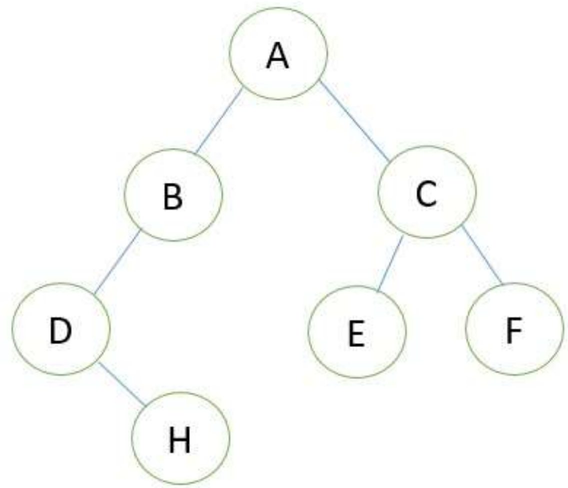
Binary Tree – Array Representation



A	B	C	D	E	F	G

1 2 3 4 5 6 7

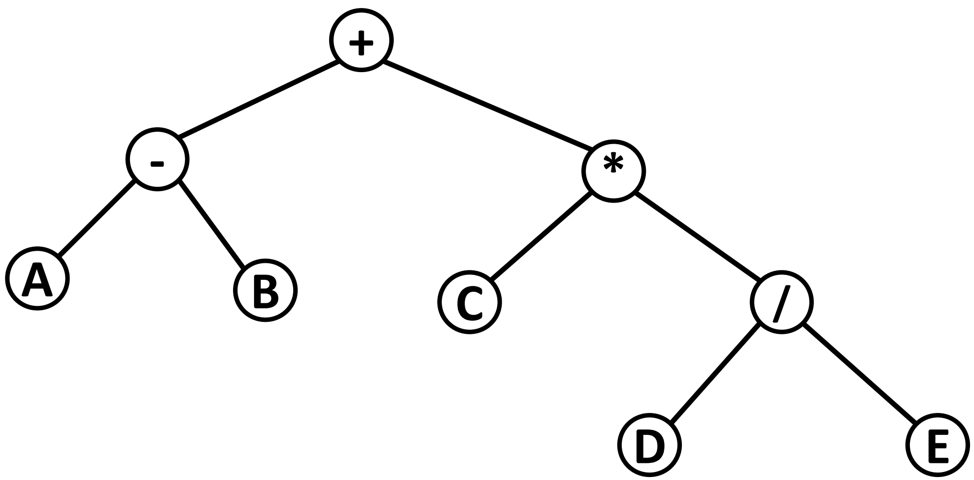
Binary Tree – Array Representation



A	B	C	D		E	F		H						

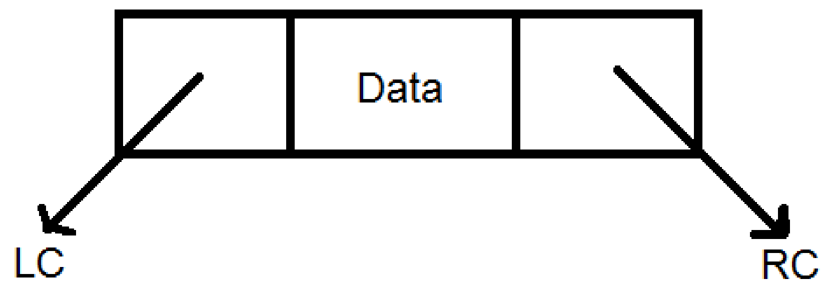
Binary Tree – Array Representation

- (A – B) + C * (D/E)   // Expression tree



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	-	*	A	B	C	/							D	E

Binary Tree - Linked List Representation



- Data is the information content of the node
- LC and RC are the two link fields used to store the addresses of Left child and Right child of a node
- If one knows the address of the root node, then from it any other node can be accessed

Binary Tree Representation

- Check limitations and advantages of Array representation
- Compare that with linked List

Binary Tree Traversals

- Traversal operation is used to visit each node in the tree exactly once
- A full traversal on a binary tree gives a linear ordering of the data in the tree
- If the binary tree contains an arithmetic expression then its traversal may give the expression in infix notation, prefix notation and postfix notation.

Binary Tree Traversals

- **Inorder (L R_0 R)**
  - Traverse the left sub tree of the root node in inorder
  - Visit the Root node
  - Traverse the Right sub tree of the root node in inorder
- **Preorder (R_0 L R)**
  - Visit the Root node
  - Traverse the left sub tree of the root node in preorder
  - Traverse the Right sub tree of the root node in preorder
- **Postorder (L R R_0)**
  - Traverse the left sub tree of the root node in postorder
  - Traverse the Right sub tree of the root node in postorder
  - Visit the Root node

Inorder Traversal (L R_0 R)

Algorithm Inorder(Ptr) // initially Ptr will be Root

//Start from Root node

1) If(Ptr ≠ NULL) then // If it is not an empty node
   a) Inorder (Ptr → LC) // Traverse left sub tree in inorder
   b) Visit (Ptr) // Visit the node
   c) Inorder (Ptr → RC) // Traverse right sub tree in inorder
2) Endif
3) Stop

Preorder Traversal (R_0 L R)

Algorithm Preorder(Ptr) // initially Ptr will be Root

// Start from Root

1) If(Ptr ≠ NULL) then // If it is not an empty node
   a) Visit (Ptr) // Visit the node
   b) Preorder (Ptr → LC) // Traverse left sub tree in preorder
   c) Preorder (Ptr → RC) // Traverse left sub tree in preorder
2) Endif
3) Stop

Postorder Traversal (L R R_0)

Algorithm Postorder (Ptr) // initially Ptr will be Root

// Start from Root node

1) If(ptr ≠ NULL) then // If it is not an empty node
   a) Postorder (Ptr → LC) // Traverse left sub tree in postorder
   b) Postorder (Ptr → RC) // Traverse left sub tree in postorder
   c) Visit (Ptr) // Visit the node
2) Endif
3) Stop

Formation of Binary Tree From Traversals

- From a single traversal it is not possible to create a unique binary tree
- Two traversals are essential
  - One should be inorder traversal
  - Other can be Preorder or Postorder
- **Basic Principle**
  - If Preorder is given – First node is the root node
  - If Postorder is given – Last node is Root node
  - Once root is identified, its left and right sub trees can be identified from the inorder traversal

    (The same method is repeated in the sub-trees)

Non Recursive B.T Traversals (Iterative)

- **Preorder**

1) Push(Root)
2) **While**(Top ≠ 0) do // while stack is not empty
   1) Ptr = Pop()
   2) **If**(Ptr ≠ NULL)
      i. Visit (Ptr)
      ii. Push(RChild[Ptr]) , if there is a RChild for Ptr
      iii. Push(LChild[Ptr]) , if there is a LChild for Ptr
   3) **EndIf**
3) **End While**
4) Stop

Non Recursive B.T Traversals (Iterative)

➢ Inorder

1) Create an empty stack **S**.
2) Initialize **current** node as root
3) Push the current node to Stack **S** and set **current = current → Left_Child** until current is NULL
   (ie. If ***current* is NULL** stop Step 3, **go to step 4**)
4) If current is NULL and stack is not empty then
   a) X= Pop() // Pop the top item from stack.
   b) Print the popped item X
   c) Set **current = X → Right_Child** // Right Child of Popped Item
   d) Go to step 3.
5) If **current is NULL** and **stack is empty** then Finished

Inorder – iterative

1) Set **curr = Root**
2) **While** (curr != NULL || Stack **s** is not empty)
   1) **While** (curr != NULL)

      s.push(curr)

      curr = curr->left

      /* Reach the left most Node of the *curr* Node */
   2) **End While**
   3) X = s.pop()
   4) **Display X**
   5) curr = X->right;

      /* we have visited the node and its left subtree. Now, it‘s right subtree's turn */
3) **End While**

Non Recursive B.T Traversals (Iterative)

- **Postorder**

// Here Two Stacks are used St1 and St2

1) Push Root into St1
2) While(St1 is not empty)
    1) X = St1.Pop() // Pop the node from St1
    2) St2.Push(X) // Push it into St2.
    3) St1.Push(X →LC) , if Left child is not NULL
    4) St1.Push(X →RC) , if Right Child is not NULL

       //Push the left and right child nodes of popped node into St1.
3) EndWhile
4) Pop out all the nodes from St2 and print it.

Binary Search Tree

- It’s a Binary Tree
- For any node “n”, value of “n” is Larger than every node in Left Subtree and Smaller than every node in Right Subtree
- All the elements in a BST will be unique. Ie. there will not be any duplicate elements.

Searching an Item in BST

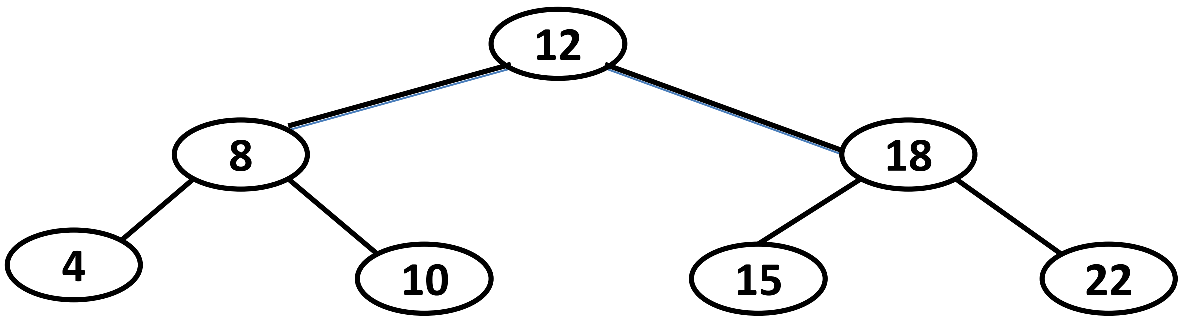
- Suppose ITEM is the item to be searched in a binary search tree.
- We will start from root node R.
  - If ITEM is the data in the node we will stop – success
  - If ITEM is less than the value in the node, we will proceed to the left child
  - If ITEM is larger, we will proceed to the right child
- This process will be continued until we reach a dead end (ITEM is not Present)

Searching an Item in BST - Algorithm

Steps:

```
ptr = ROOT, flag = FALSE                                  // Start from the root
While (ptr ≠ NULL) and (flag = FALSE) do
    Case: ITEM < ptr→DATA                                 // Go to the left sub-tree
        ptr = ptr→LCHILD
    Case: ptr→DATA = ITEM                                 // Search is successful
        flag = TRUE
    Case: ITEM > ptr→DATA                                 // Go to the right sub-tree
        ptr = ptr→RCHILD
    EndCase
EndWhile
If (flag = TRUE) then                                     // Search is successful
    Print "ITEM has found at the node", ptr
Else
    Print "ITEM does not exist: Search is unsuccessful"
EndIf
```

Binary Search Tree - Searching



If Search item is 10

- Root - 12
- **10** <12 --- 12→LC is 8
- **10** > 8 --- 8→RC is 10
- **10** = 10 ---- Success

If Search item is 17

- Root - 12
- **17** >12 --- 12→RC is 18
- **17** < 18 --- 18→LC is 15
- **17** > 15 ---- 15→RC is **NULL**
- Search Failed

Binary Search Tree - Insertion

- While inserting a new node initially the binary tree is searched (with the item is to be inserted) from its Root node.
- If the item is to be inserted already exists, do nothing.
- Otherwise the item will be inserted at the dead end where the search halts.

BST Insertion - Algorithm

// Let X be the data of the node to be inserted, initially Root will be NULL (empty tree)

1) Ptr = Root, Flag = False
2) **While** (Ptr ≠ NULL) and (Flag = False) **do** // Start from Root
   1) **If** (X < Ptr → Data) **then** // Go to Left Subtree
      1) Ptr1 = Ptr
      2) Ptr = Ptr → LChild
   2) **Else If** (X > Ptr → Data) // Go to Right Subtree
      1) Ptr1 = Ptr
      2) Ptr = Ptr → RChild
   3) **Else** // Node exists
      1) Flag = True
      2) Print " Item X already exists"
      3) Exit // Quit the execution
   4) **EndIf**
3) **End While**

BST Insertion - Algorithm

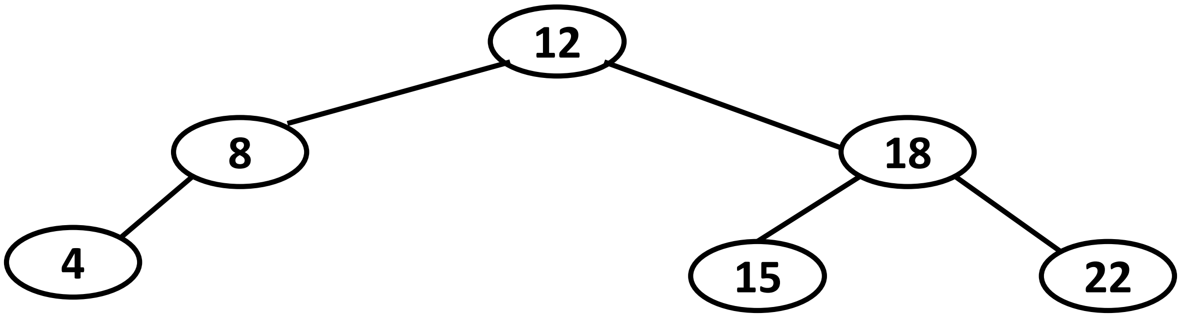
4) **If** (Ptr = NULL) **then**
    1) Create a new node – New
    2) New → Data = X
    3) New → LChild = NULL
    4) New → RChild = NULL
    5) **If** (Root = NULL)

        Root = New
    6) **Else If** (X > Ptr1 → Data) **then**

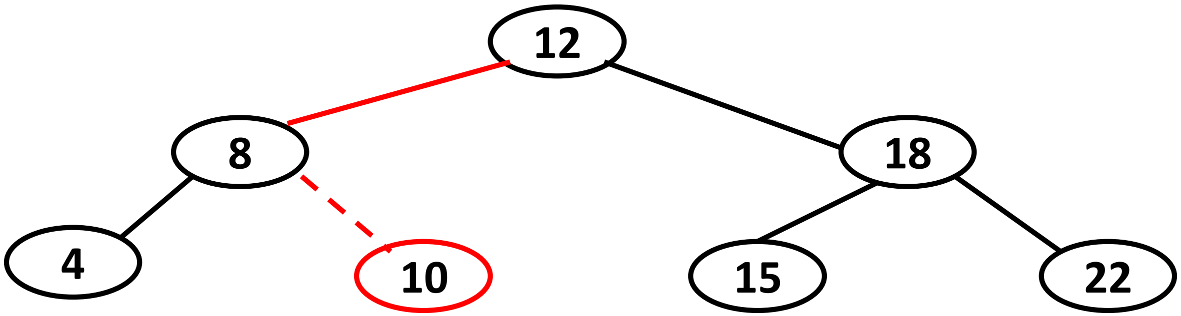
        Ptr1 → RChild = New
    7) **Else**

        Ptr1 → LChild = New
    8) **End if**
5) **Endif**
6) Stop

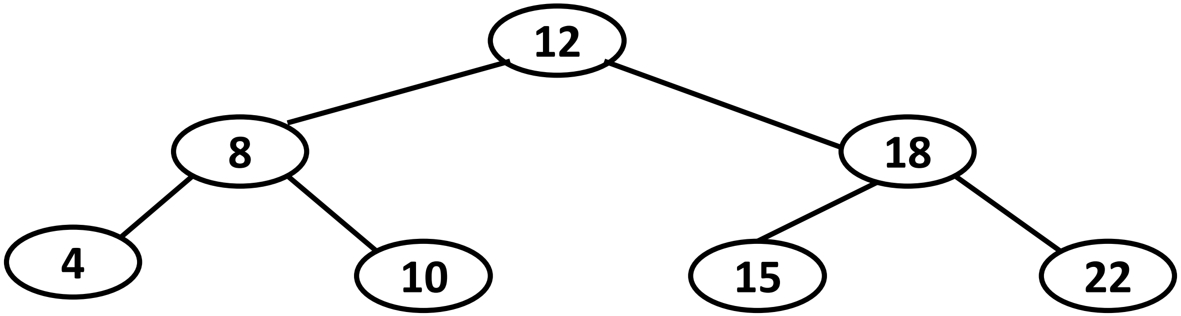
BST Insertion - Example



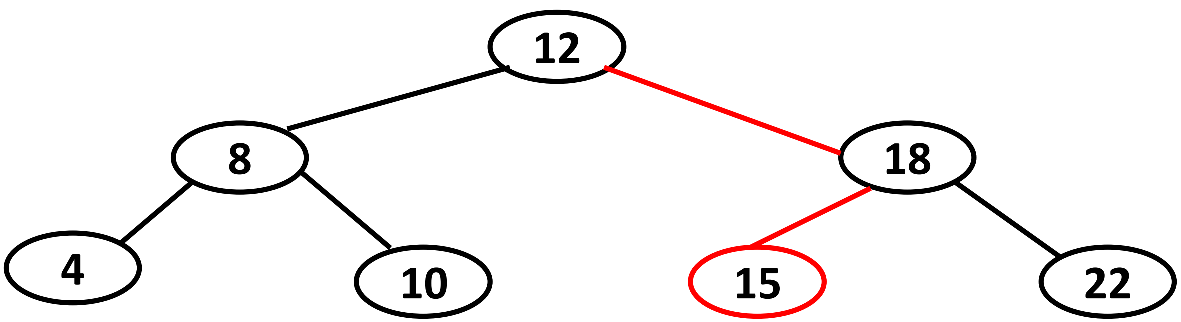
- If Item 10 is to be added It will check as in the figure and item will be added as a Right Child of node 8



BST Insertion - Example



- If Item 15 is to be added It will check as in the figure and item will find that 15 already exists. So it will stop without inserting a new node



Binary Search Tree - Deletion

- The deletion of a node N depends on the number of its children
- **Three cases** are there
    1) N is leaf node
    2) N has exactly one Child
    3) N has two children

Binary Search Tree - Deletion

- **N is a leaf node**
  - Here N is simply deleted from the Tree by setting the pointer of N in the Parent(N) by NULL value
- **N has exactly one child**
  - Here N is deleted from the Tree by replacing the pointer of N in Parent(N) by the pointer of the only child of N

Binary Search Tree - Deletion

- **N has two children**
  - N is deleted from Tree by first deleting ***Succ(N***) from Tree (by using case 1 & case 2) and then replacing the data content in node N by the data content in node ***Succ(N)***.

    (It should be verified that Succ(N) has no Left child)
  - Reset the left child of Parent of ***Succ(N***) by the right child of ***Succ(N)***

BST Deletion - Algorithm

Algorithm Delete_BST(X) //Let **X** be the data in the node to be deleted

1) Ptr = Root, Flag = False
2) **While** (Ptr ≠ NULL) and (Flag = False) **do**
3) **If**(X < Ptr → Data) **then**
    1) Parent = Ptr
    2) Ptr = Ptr → Lchild
4) **Else if** (X > Ptr → Data) **then**
    1) Parent = Ptr
    2) Ptr = Ptr → Rchild
5) **Else if**(X = Ptr → Data) **then**
    1) Flag = True
6) **EndIf**
7) **End While**

// Steps to Find
// the location of
//the node

// Deciding the case of Deletion

8) If (Flag = False) **then** // node does not exist

1) Print “Item Not Found”

2) Exit

9) EndIf

10) If(Ptr → Lchild = NULL) and (Ptr → Rchild = NULL) **then**

Case = 1 // node has no child

11) Else if(Ptr → Lchild ≠ NULL) and (Ptr → Rchild ≠ NULL) **then**

Case = 3 // node contains left and right child

12) Else

Case = 2 // node contains only one childe

13) EndIf

// Deletion in Case 1

14) If (Case = 1) **then**

- **1) If**(Parent → Lchild = Ptr) **then** // if node is a left child

  Parent → Lchild = NULL

- **2) Else** // if node is a right child

  Parent → Rchild = NULL

- **3) EndIf**

- 4) Return Ptr (deleted node) to the memory bank

15) EndIf

// Deletion in Case 2

16) If (Case = 2) then

   1) If(Parent → Lchild =Ptr) then // if node is a left child

      1) If(Ptr → Lchild = NULL) **then** // if node has no left child

         Parent → Lchild = Ptr → Rchild

      2) Else

         Parent → Lchild = Ptr → Lchild

      3) EndIf

   2) Else If(Parent → Rchild = Ptr) then // if node is a right child

      1) If(Ptr → Lchild = NULL) **then** // if node has no left child

         Parent → Rchild = Ptr → Rchild

      2) Else

         Parent → Rchild = Ptr → Lchild

      3) EndIf

   3) EndIf

   4) Return Ptr (deleted node) to the memory bank

17) EndIf

//Deletion in Case 3

18) **If** (Case = 3) **then**

1) Ptr1= **Succ(Ptr)** // Find the Inorder Successor of Ptr
2) Item1 = Ptr1 → Data
3) **Delete_BST(Item1)** // Delete the Inorder Successor
4) Ptr → Data = Item1

// Replace the data with the data of the inorder successor

19) **EndIf**

20) Stop

Finding Inorder Successor

Algorithm Succ(Ptr)

1) Ptr1 = Ptr → Rchild // move to the right subtree
   1) **If** (Ptr1 ≠ NULL) **then** // right subtree not empty
      1) **While**(Ptr1 → Lchild ≠ NULL) **do** // move to the
      2) Ptr1 = Ptr1 → Lchild // left most end
      3) **EndWhile**
   2) **EndIf**
   3) Return(Ptr1)
2) Stop

- Check the application of Binary trees in the text book