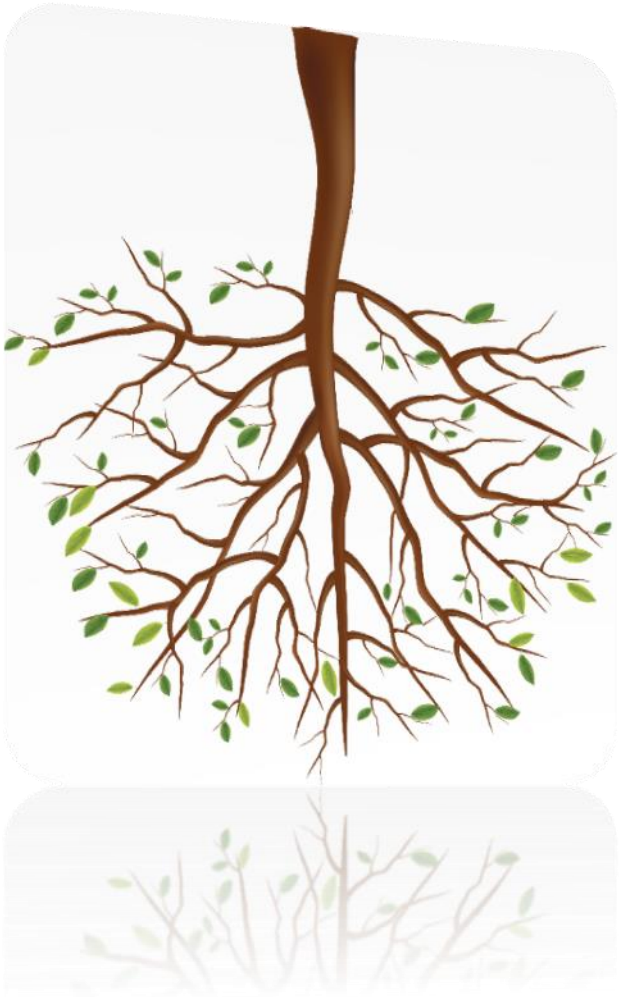
Chapter Six

Tree Structures



Classification of Data Structures

Linear Data Structure

Arrays

Linked Lists

Stacks

Queues

Non-Linear Data Structure

Trees

Graphs

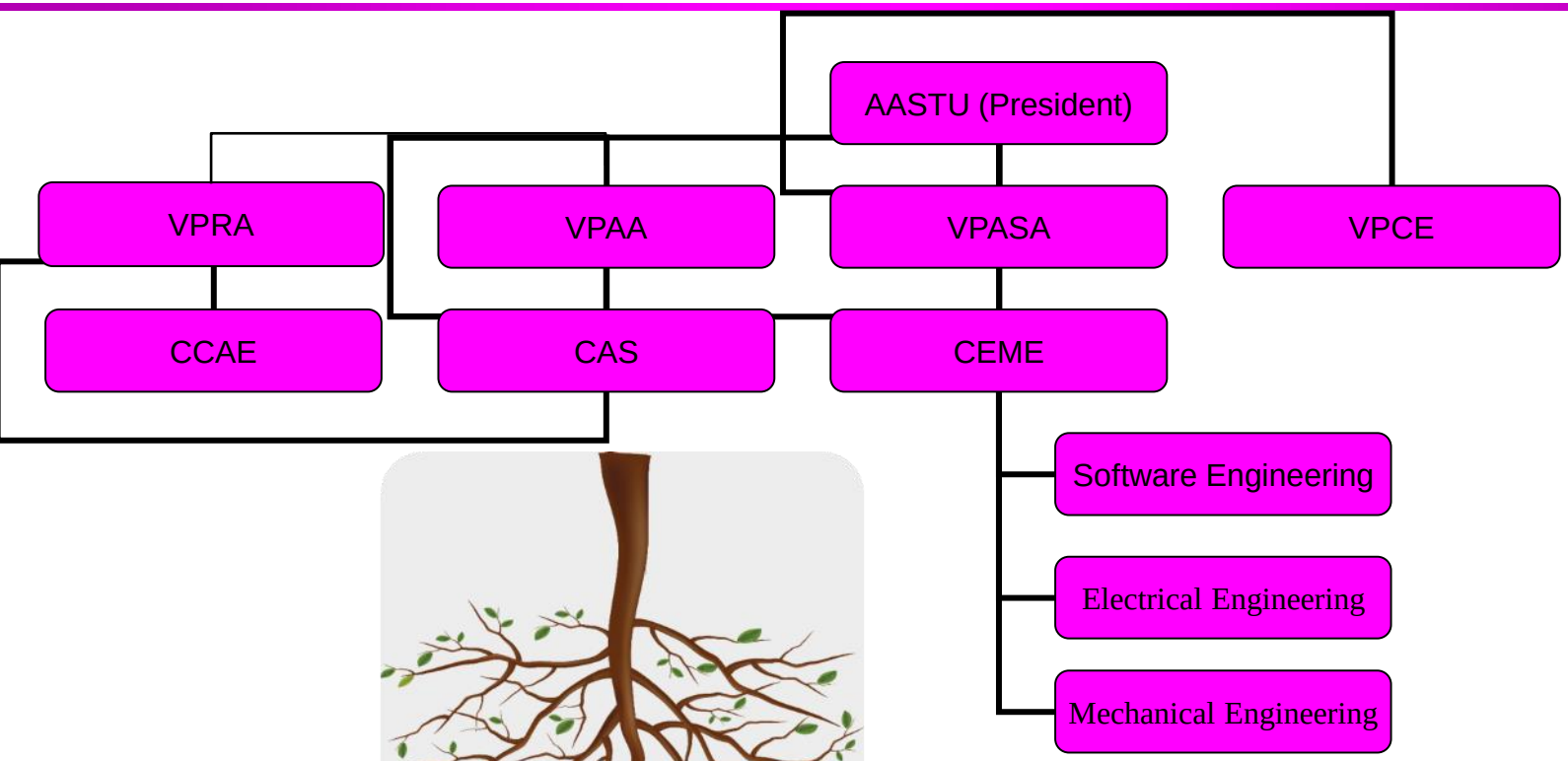
Introduction

- A ***tree*** is a set of ***nodes*** and ***edges*** that connect pairs of nodes.
- It is an abstract model of a ***hierarchical structure***.
- Unlike a real tree, tree data structures are typically depicted ***upside down***, with the ***root*** at the top and the ***leaves*** at the bottom.
- Each node can have a number of ***child*** nodes, and the ***parent*** and child nodes are connected by ***arcs***.
- The ***root*** is the top node that has ***no parent*** nodes.
- The ***leaves*** of the tree are those that have ***no child*** nodes.

Trees Data Structures

- **Tree**
  - A way of representing hierarchical data.
  - Consist of nodes with a parent-child relationship.
  - Each node can have 0 or more children.
  - A node can have at most one parent.

Example


AASTU (President)
VPRA
VPAA
VPASA
VPCE
CCAE
CAS
CEME
Software Engineering
Electrical Engineering
Mechanical Engineering

Applications of Tree in Computer Science

- Storing naturally hierarchical data - File system.
- Organize data for quick search, insertion, deletion - Binary search tree.
- Dictionary
- Network Routing Algorithm.
- etc.

Tree Terminology

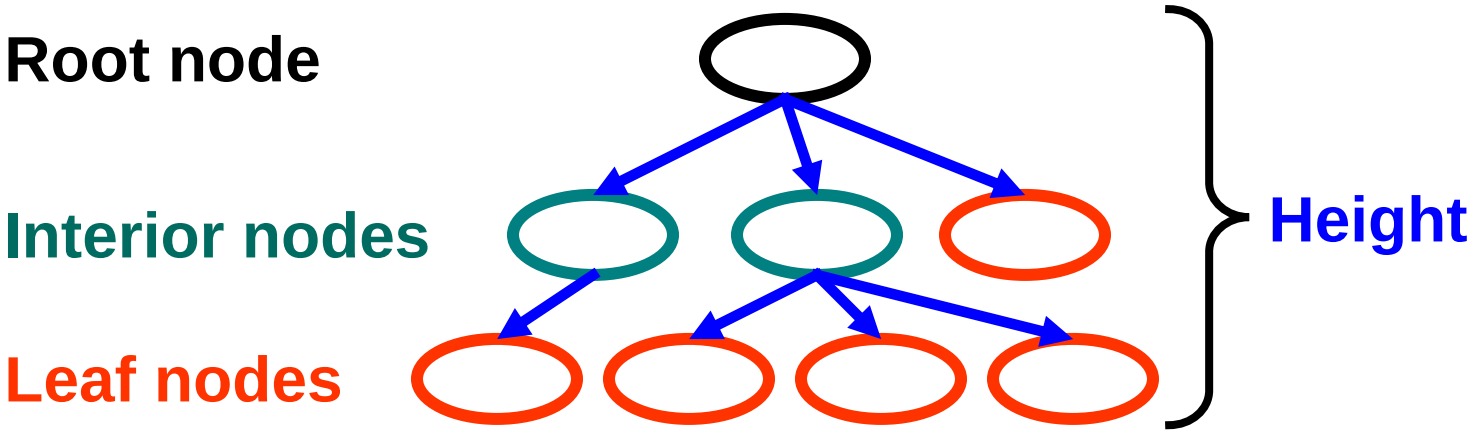
- A ***path*** through the tree to a node is a ***sequence of arcs*** that connect the root to the node.
- The ***length*** of a path is the ***number of arcs*** in the path.
- The ***level*** of a node is equal to the length of the path from the ***root*** to the ***node plus 1***.
- The ***root*** node is said to be at level 1 of the tree, its children at level 2, and so on.
- The ***height*** of a tree is ***equal to the maximum level*** of a node in the tree.

Tree Terminology

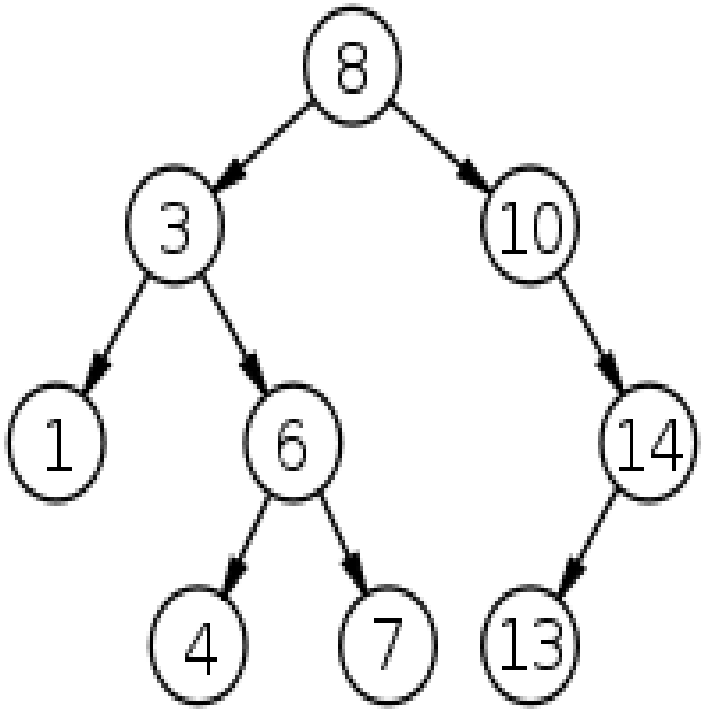
- Root node - a node which has no parent.
- Internal node – which has at least one child
- Leaf node - which has no child
- Ancestor- nodes which are parent, grandparent, grand-grand-parent, etc.
- Depth of a node: means number of ancestors
- Height of a tree: maximum distance from root to leaf
- Descendent nodes: child, child-of-child etc.

Tree Terminology

- Root ⇒ no parent
- Leaf ⇒ no child
- Interior ⇒ non-leaf
- Height ⇒ distance from root to leaf



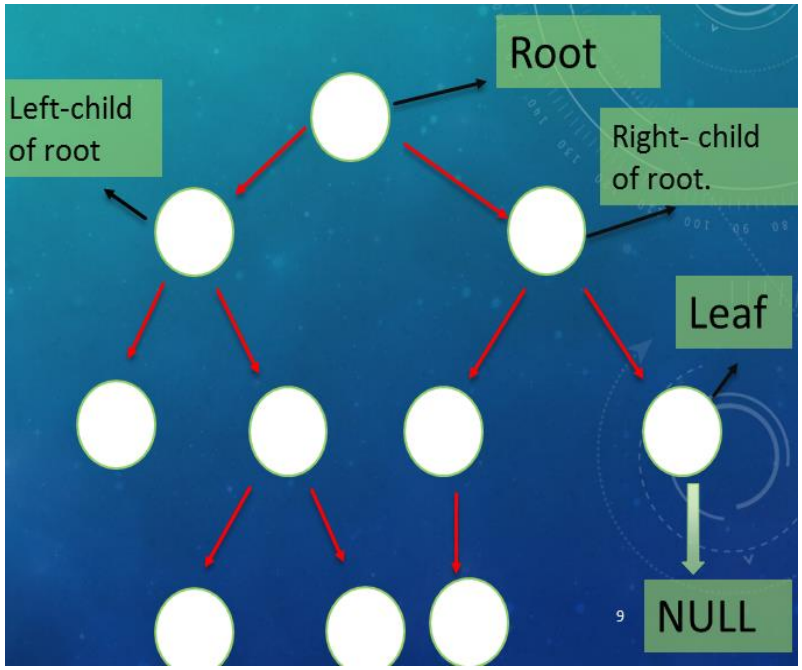
Example of a Tree



- No. of nodes = 9
- Height = 4
- Levels = 1,2,3,4
- Root Node = 8
- Leaves = 1,4,7,13
- Interior Nodes = 3,10,6,14
- Ancestors of 6 = 3,8
- Descendants of 10 = 14,13
- Sibling of 1 = 6

Binary Tree

- A tree in which each node has at most two children called left child and right child.
  - A node have only left and right child or
  - Only left child or
  - Only right child.
  - A leaf node has no left or right child.
  - A leaf node has only NULL.



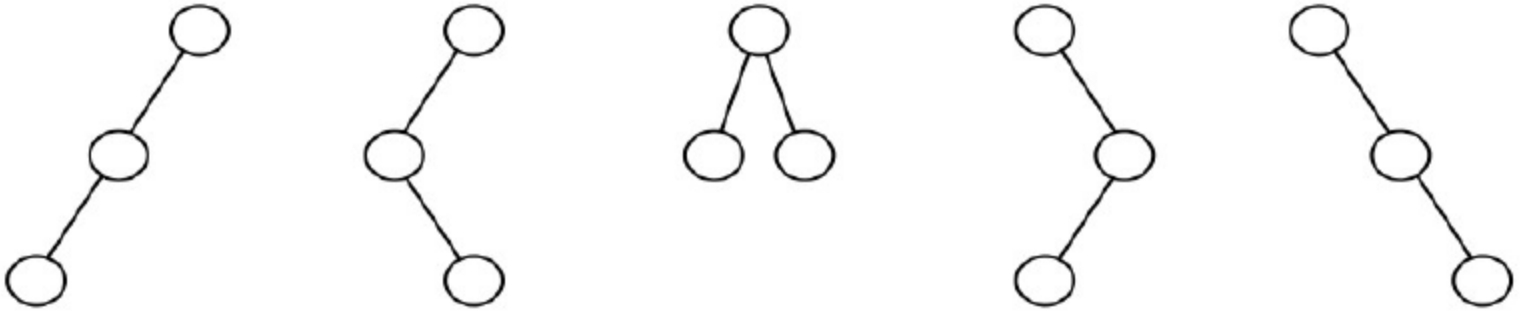
Binary Tree

Examples …

With One Child Binary Tree

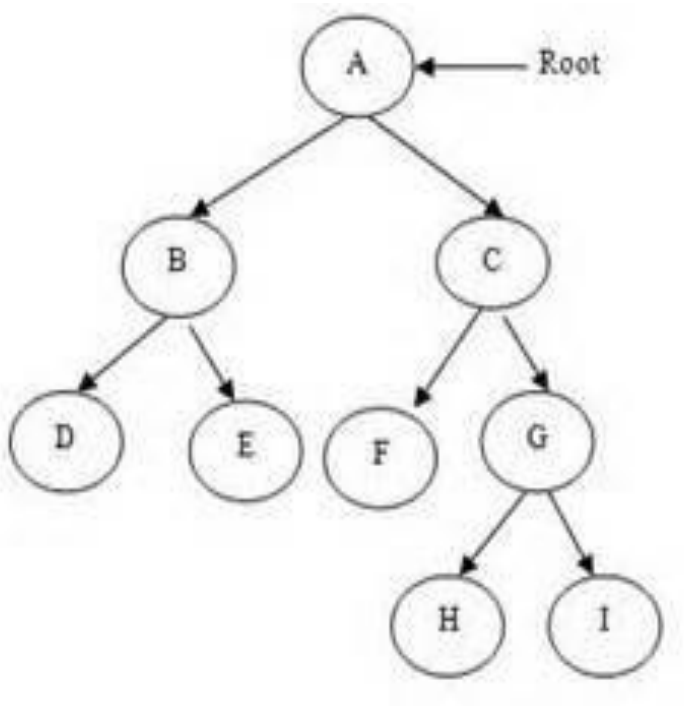
Invalid binary tree

Binary tree



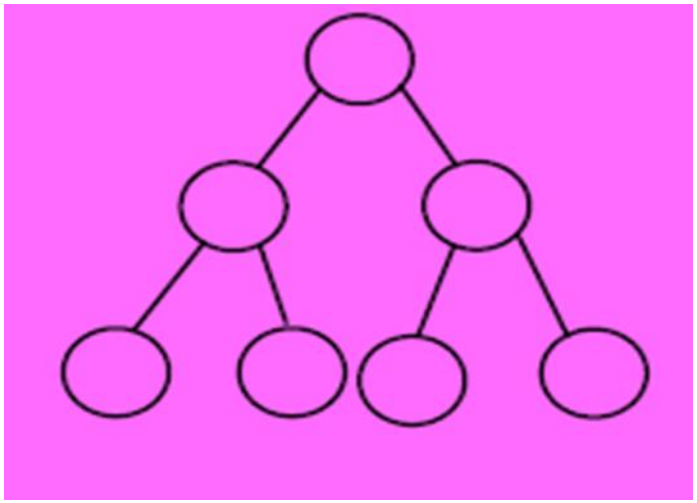
Types of Binary Tree

- Strict binary tree
  - Each node has exactly either two or zero child.
  - Every non-leaf node has non-empty left and right sub trees.
  - A strict BT with n leaves always contains '2n-1' nodes.

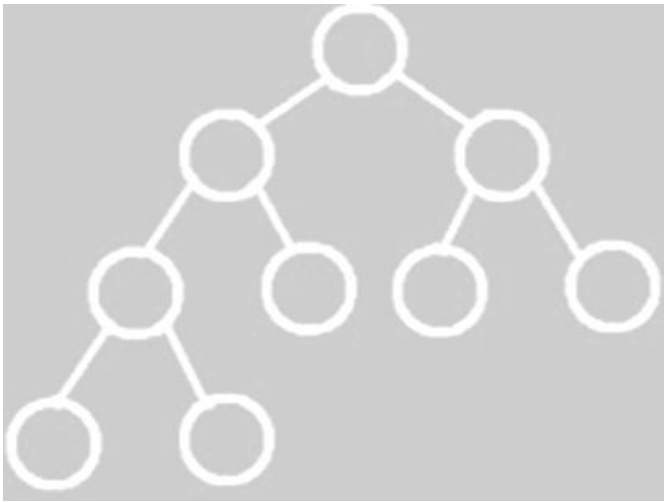


Types of Binary Tree

- Balanced binary tree
  - a binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.



- Complete binary tree
  - a binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree.
  - The deepest level should also be filled from left to right.

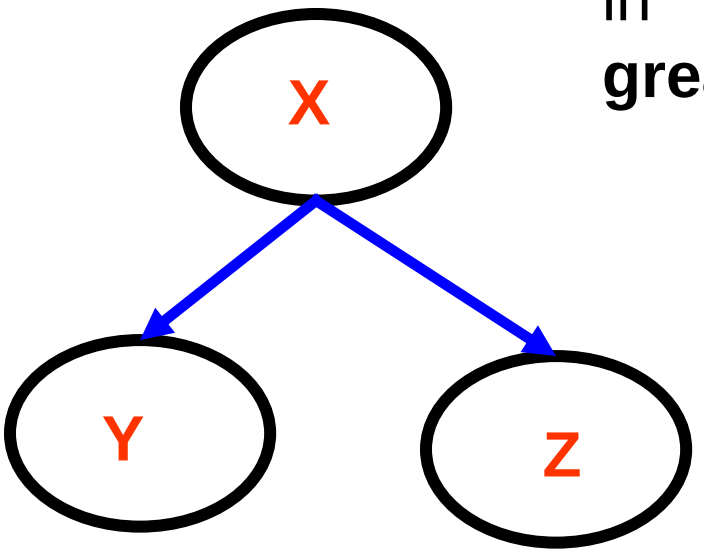


Binary Search Trees (Ordered binary tree)

- A binary search tree is either ***empty*** or in which every node ***contains a key*** and ***satisfied the conditions***:
  1. The key in the ***left child*** of a node (if it exist) is ***less than*** the key in its ***parent node***.
  2. The key in the ***right child*** of a node (if it exist) is ***greater than*** the key in its ***parent node***.
  3. The ***left and right sub-trees*** of the root are again ***binary search trees.***
  4. ***No two entries*** in a binary search tree may have a ***equal keys.***

Binary Search Trees

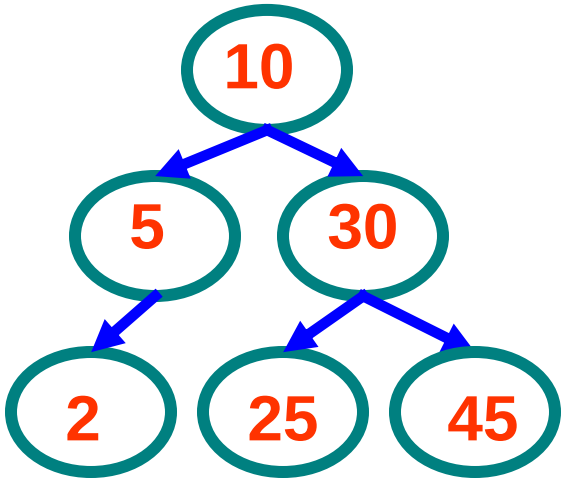
- **In short:**
  - Value at node
    - Smaller values in left sub-tree
    - Larger values in right sub-tree
  - Example
    - X > Y
    - X < Z

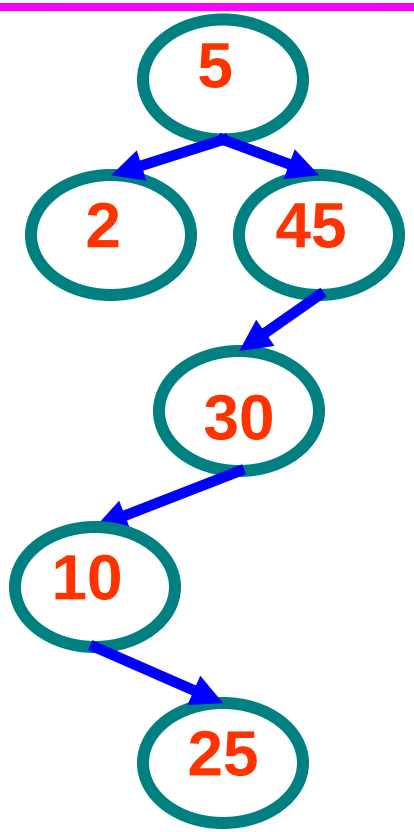


- Value of all the nodes in left ***sub-tree is lesser***.
- Value of all the nodes in **right sub-tree is greater**.

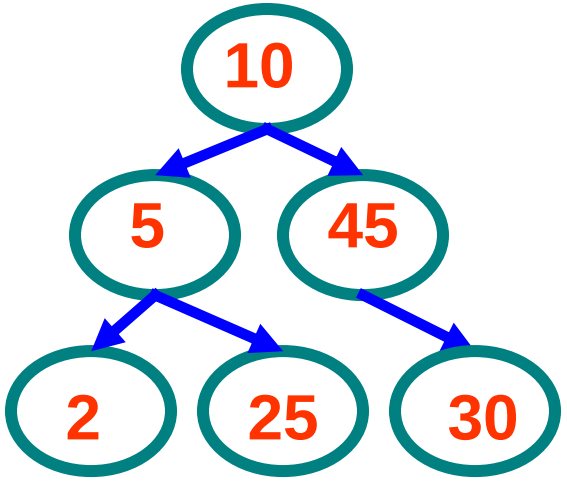
Binary Search Trees

- Examples





Binary search trees

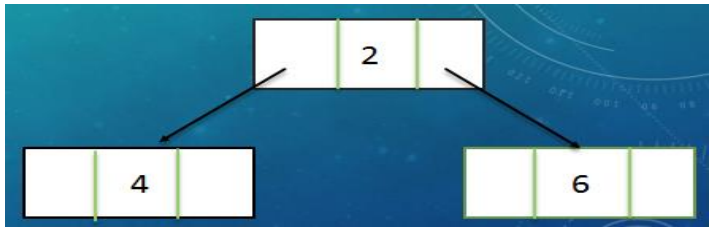


Not a binary search tree

10>25?, False
30>45?, False

Binary Tree Implementation (using linked list)

- Tree is a data structure similar to linked list.
- Instead of pointing to one node ***each node can point to a number of nodes.***



ata Structure of a Binary Tree

```
struct node{
  //declaration of data fields
  node *leftChild,
  node *rightChild;
 };
node *rootPtr = NULL;
```

Example:

```
struct node{
 int key;
 node *left,
 node *right;
  };
node *rootPtr=NULL;
```

Data Structure of Generic Tree (N-ary) Tree

- Each node can have no child, one child, two child or more than two child - called ***generic tree***, tree of a general kind.
- How to design data structure for such a tree?

Possible solution may be:

…But wait, this structure has problem?

```
struct node{

      int data;
      node *firstchild;
      node *secondchild;
      node *threechild;
      node *fourchild;
      node *fivechild;
     };
 node *rootPtr
```

Data Structure of Generic Tree (N-ary) Tree

- We don't know the exact number of children for each node.
  - ***Wastage of memory*** by allocating extra child since we are not using all the pointers in all the cases.
- Possible solution is:
  - At each node there should be a link between all the children of same parent (sibling).
  - Also, remove all the link from parent to their child except first child.

```
struct node{
        int data;
        node *firstchild;
        node *nextsibling;
    };
   node *rootPtr = NULL;
```

Operations on Binary Search Tree

- Basic Operations:
  - Inserting an element into tree
  - Deleting an element from tree
  - Searching for an element
  - Traversing the tree
- Auxiliary operations:
  - Finding – size of tree, height of tree, etc

BST – Insertion

When a node is inserted the ***definition of binary search tree*** should be ***preserved***!

- **Suppose:**
  - rootPtr -points to root node
  - newPtr -points new node
- **Case 1:** There is no data in the tree
  - (i.e. rootPtr is NULL)
  - The node pointed by newPtr should be made the root node.
- **Case 2**: There is data
  - Search the appropriate position.
  - Insert the node in that position.

Algorithm

1. Perform search for value X.
2. Search will end at node Y (if X is not in the tree).
3. If X < Y, insert new leaf X as new left sub-tree for Y.
4. If X > Y, insert new leaf X as new right sub-tree for Y.

- Insertions may unbalance the tree.

Algorithm

1. Input the *data* to be pushed and root node of the tree.
2. newPtr = Create a New Node, parentPtr = rootPtr
3. If (parentPtr == NULL)

   parentPtr = newPtr
4. Else if (*data* < parentPtr ->info) //parentPtr ->info :data part

   parentPtr = parentPtr->left

   GoTo Step 4
5. Else if (*data* > parentPtr->info)

   parentPtr = parentPtr->right

   GoTo Step 4
6. If (*data*< parentPtr->info)

   parentPtr->Left = newPtr
7. Else if (data > parentPtr->info)

   parentPtr ->right = newPtr

Algorithm…

8. Else

    display (“duplicate node”)

    exit

9. newPtr->info = data

10. newPtr->left = NULL

11. newPtr->right = NULL

12. exit

Implementation for BST - Insertion

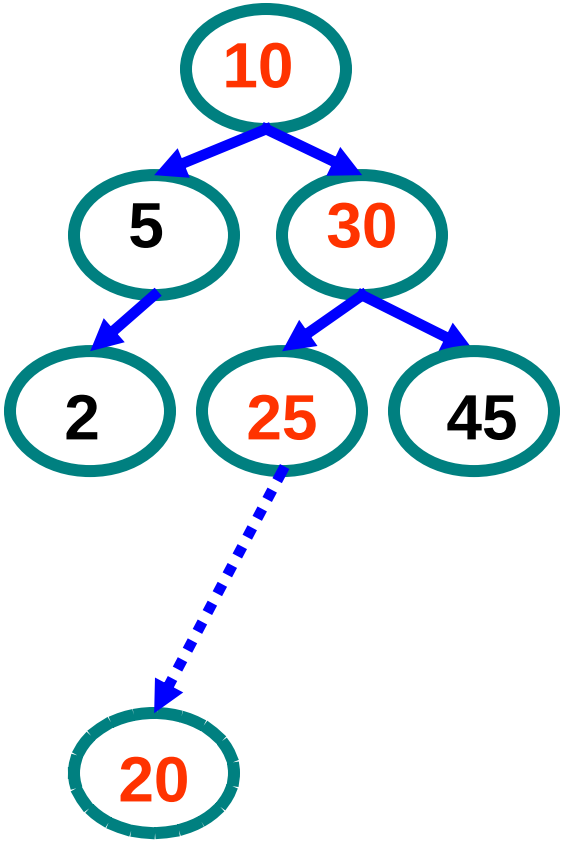
```
//insert in to a binary search tree, RNP=rootnodeptr, NNP=newnodeptr
Node *RNP, *NNP;
void insertBST(Node *RNP, Node *NNP){
    int inserted=0;//a flag to check inserted or not
    Node *temp;
    temp=RNP;
    if(temp==NULL)
{
            temp=NNP;
            RNP=NNP;}
    else{
            while(inserted==0){
                    if(temp->num>NNP->num){
                            if(temp->left==NULL){
                                    temp->left=NNP
                                    inserted=1;}
                            else
                                    temp=temp->left;}
```

Cont…

```
            else {  //if temp->num <NNP->num
                    if(temp->right==NULL){
                            temp->right=NNP;
                            inserted=1;}
                    else
                            temp=temp->right
            }//else
    }//end of while
}//end of function
```

Example

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

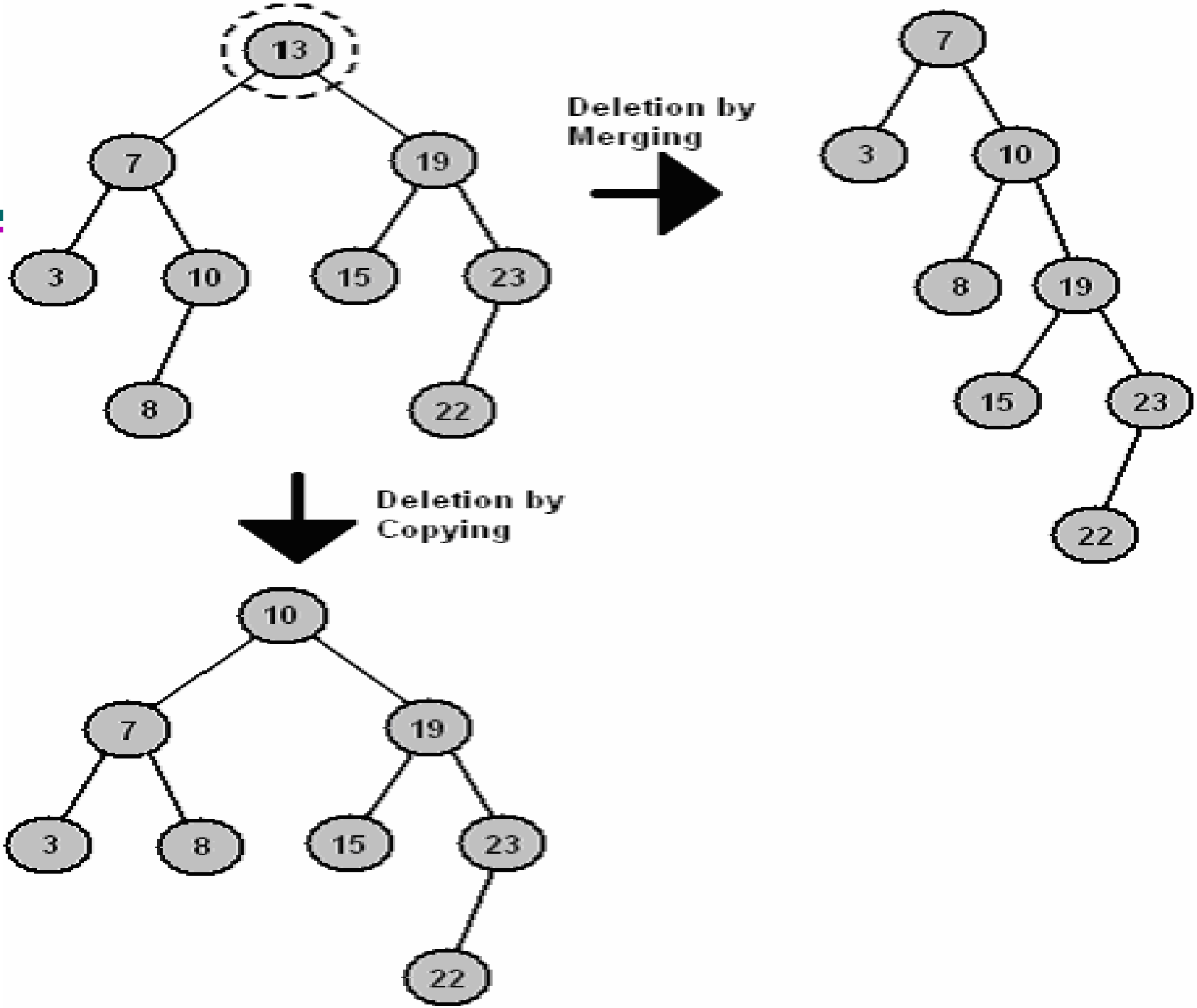
Insert 20 on left

BST - Deletion

- Deletion of a node from a binary search tree can be more problematic.
- The difficulty of the operation depends on the position of the node in the tree. There are three cases:
  1. The node is a leaf. This is the easiest case to deal with. The appropriate pointer of its parent is set to null and the node deleted.
  2. The node has one child. This case is also not complicated. The appropriate pointer of its parent is modified to point to the child of the node.
  3. The node has two children. There is no simple way to delete nodes like this. The following sections discuss two possible ways of dealing with this situation.

Deletion by Merging

- One way of deleting nodes that have two children is called *deletion by merging*.
- This algorithm works by merging the two subtrees of the node and attaching the merged tree to the node's parent.
- In binary search trees, every value in the left subtree is less than every value in the right subtree, so if we can find the largest value in the left subtree, then we can attach the right subtree as the right child of this node, and still preserve the ordered nature of the tree. This process is illustrated in the following figure.
- To find the largest value in the left subtree we need only keep tracking the right child pointer until a null is encountered.
- Symmetrically, this algorithm can also work by finding the smallest value in the right subtree, and attaching the left subtree to it.


13
7
19
3
10
15
23
8
22
Deletion by
Merging
7
3
10
8
19
15
23
22
Deletion by
Copying
10
7
19
3
8
15
23
22

Deletion by Copying

- Deletion by copying works in a similar fashion to deletion by merging.
  - First, we find the largest value in the left subtree.
  - Next, we copy this node, so that it replaces the node being deleted.
- This process is illustrated in the above figure.
  - The 10 node is a valid new root to the tree because it is greater than all nodes in the left subtree, and less than all nodes in the right subtree.
- A slight complication with this algorithm can occur if the largest value in the left subtree has a left child.
- In this case, the left child is attached to the nodes parent instead.

Algorithm

1. Perform search for value X
2. If X is a leaf, delete X
3. Else // must delete internal node

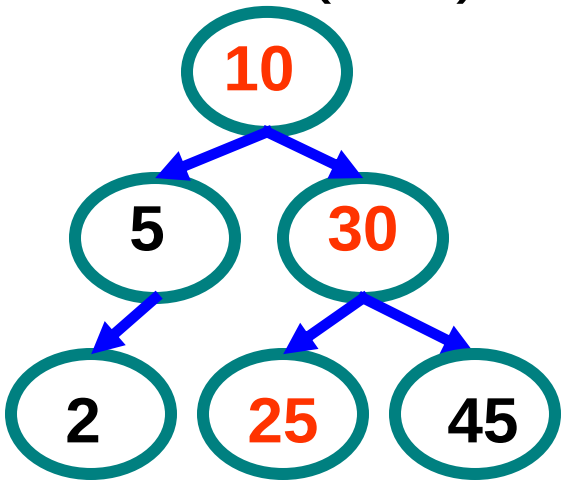
   a) Replace with largest value Y on left subtree

   OR smallest value Z on right subtree

   b) Delete replacement value (Y or Z) from subtree

Example Deletion (Leaf)

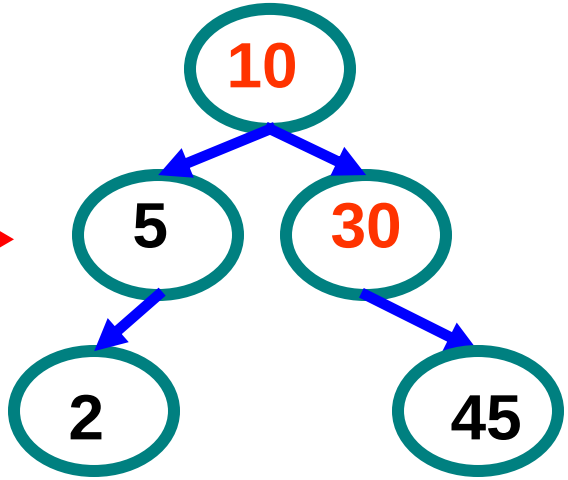
- Delete (25)



10 < 25, right

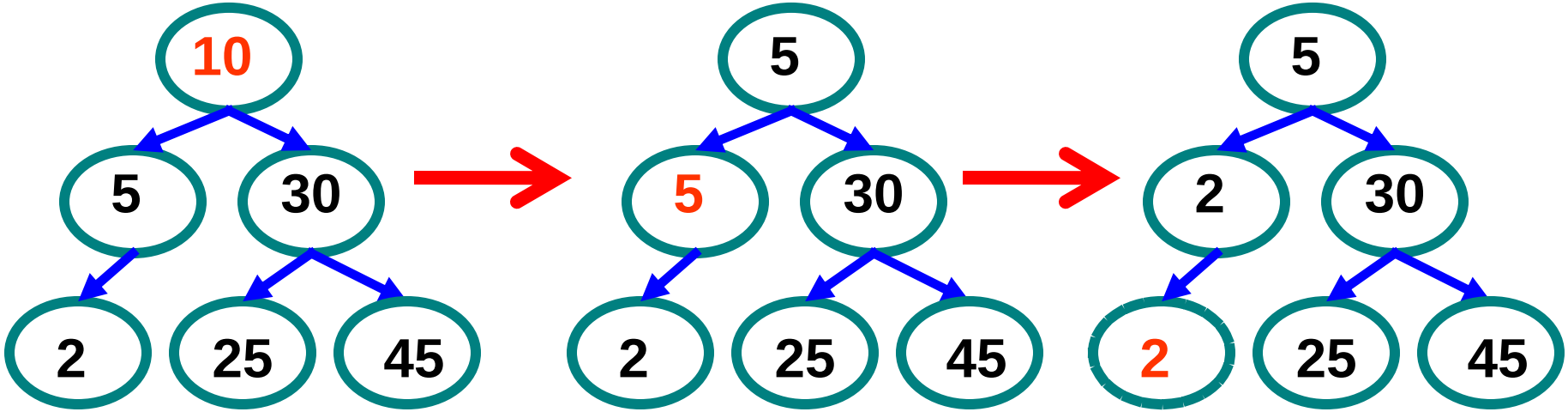
30 > 25, left

25 = 25, delete



Example Deletion (Internal node)

- Delete (10)



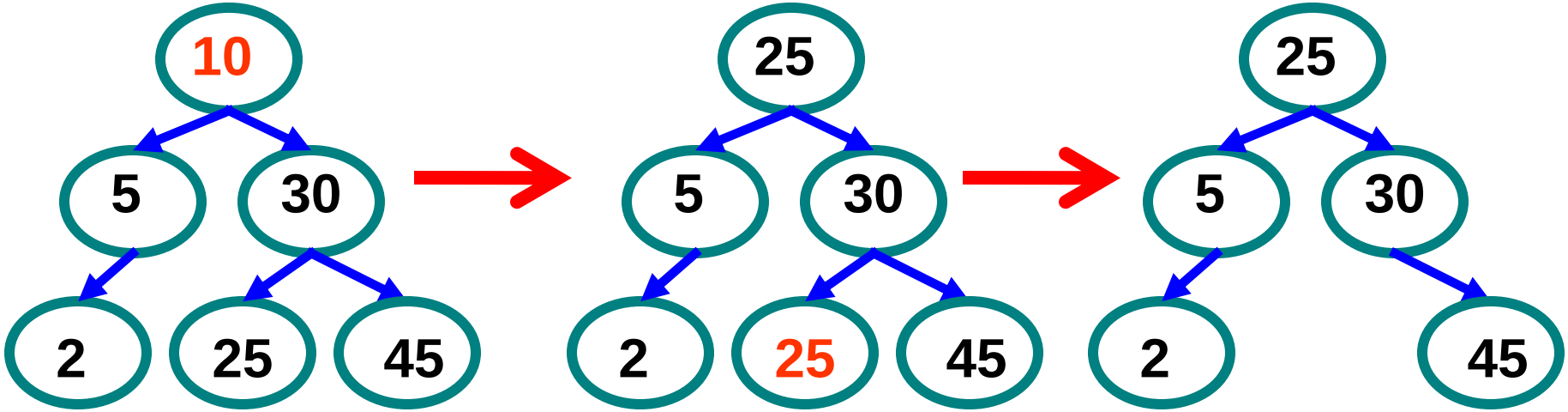
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Implementation (deletion by copy)

```
PDNP=Previous Node of the "to be deleted Node"
void deleteBST(Node *RNP, Node *PDNP,int x){
     Node *DNP;//DNP=DeleteNodePointer
     if(RNP==NULL)
         cout<<"Data Not Found";
    else if(RNP->num > x)
            deleteBST(RNP->left, RNP, x);
    else if(RNP->num < x)
         deleteBST(RNP->right, RNP, x);
    else //if RNP->num==x {
            DNP=RNP;
            if((DNP->left==NULL)&&(DNP->right==NULL)){//leaf node
                if (PDNP->left==DNP){
                    PDNP->left=NULL;
                    delete DNP;
                }
                else{
                    PDNP->right==NULL;
                    delete DNP;
                }}
```

Implementation….

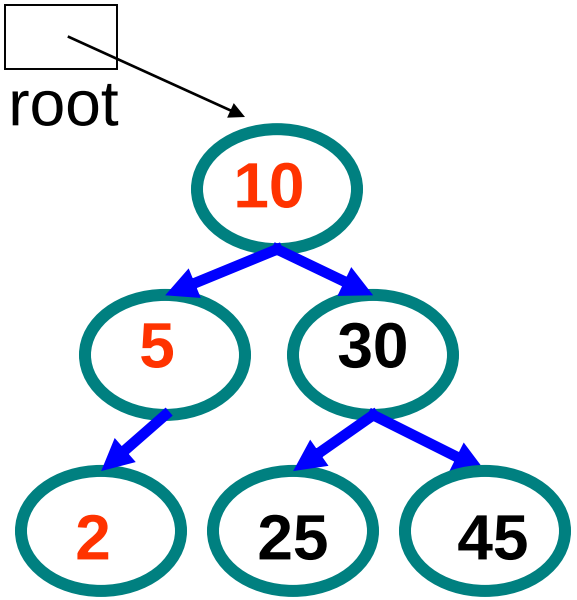
```
else if(DNP->left!=NULL){
            PDNP=DNP;
            DNP=DNP->left;
            while(DNP->right!=NULL){
            PDNP=DNP;
            DNP=DNP->right;
            }
            RNP->num=DNP->num;
            deleteBST(DNP,PDNP,DNP->num);
        }//else
        else//has only a right child
        {
            PDNP=DNP;
        DNP=DNP->right;
        while(DNP->left!=NULL){
        PDNP=DNP;
        DNP=DNP->left;
        }
        RNP->num=DNP->num;
        deleteBST(DNP,PDNP,DNP->num);
 }//else   } }
```

BST- Search

```
node *find( node *n, int key) {
      while (n != NULL) {
      if (n->data == key)          // Found it
            return n;
          if (n->data > key)              // In left subtree
            n = n->left;
          else                            // In right subtree
            n = n->right;
      }
      return null;
}
//call function
node * n = find( root, 5);
```

Example Search

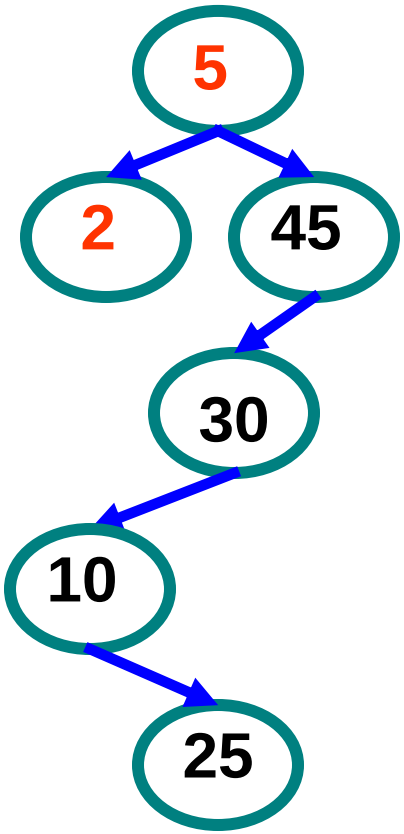
- find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

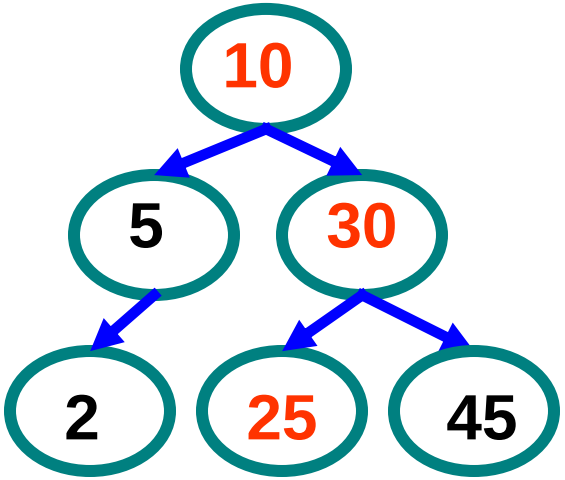


5 > 2, left

2 = 2, found

Example Search

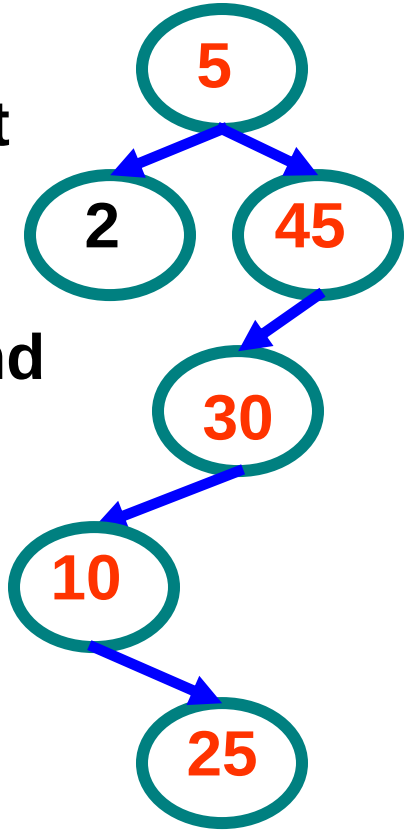
- find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

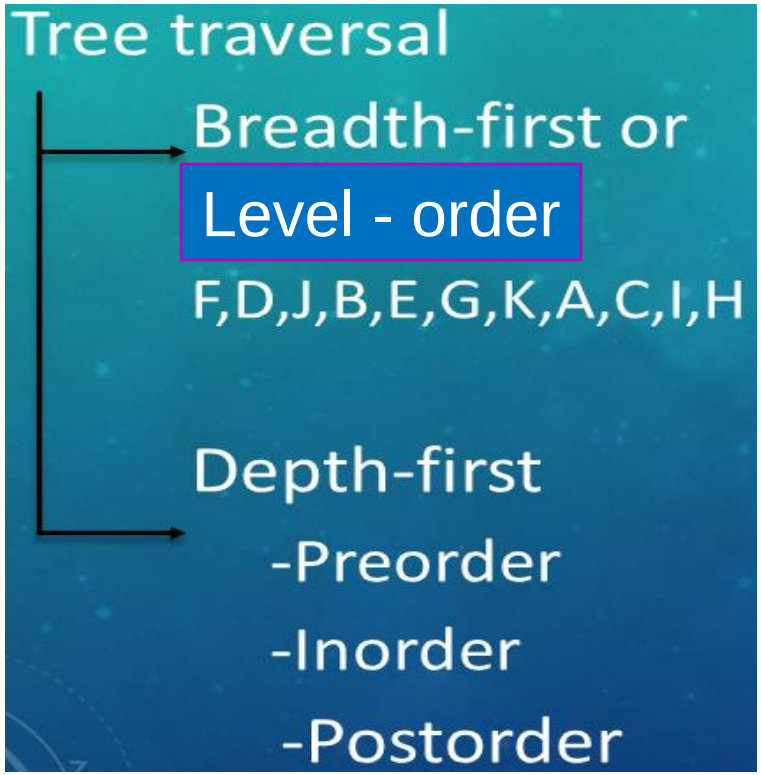
30 > 25, left

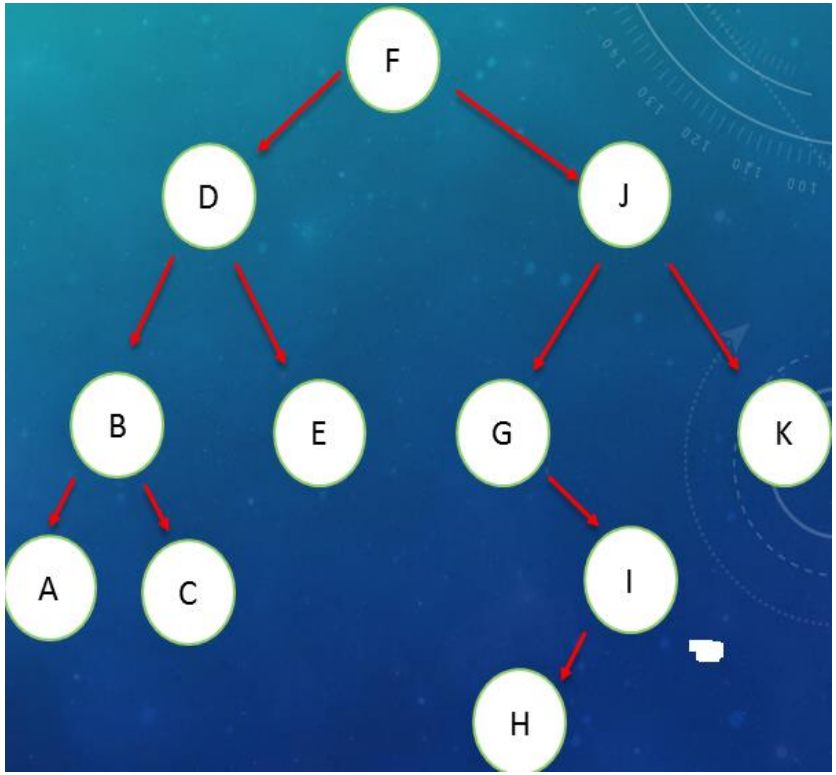
10 < 25, right

25 = 25, found

Traversing Binary Trees

- Tree traversal is one of the most common operations performed on tree data structures.
- It is a way in which ***each node in the tree is visited exactly once*** in a systematic manner.





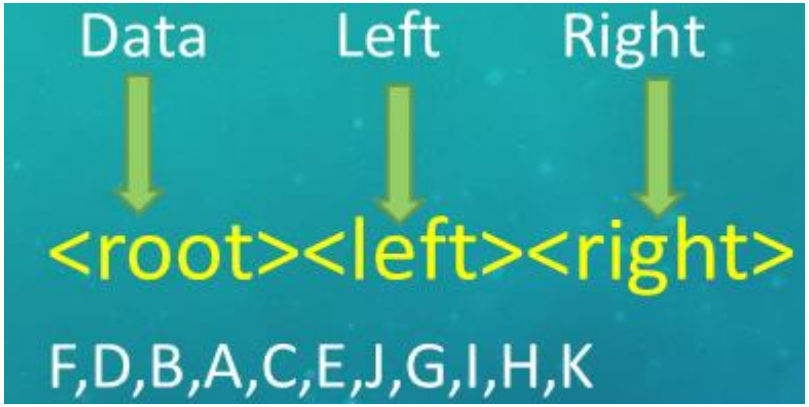
Depth First Traversal

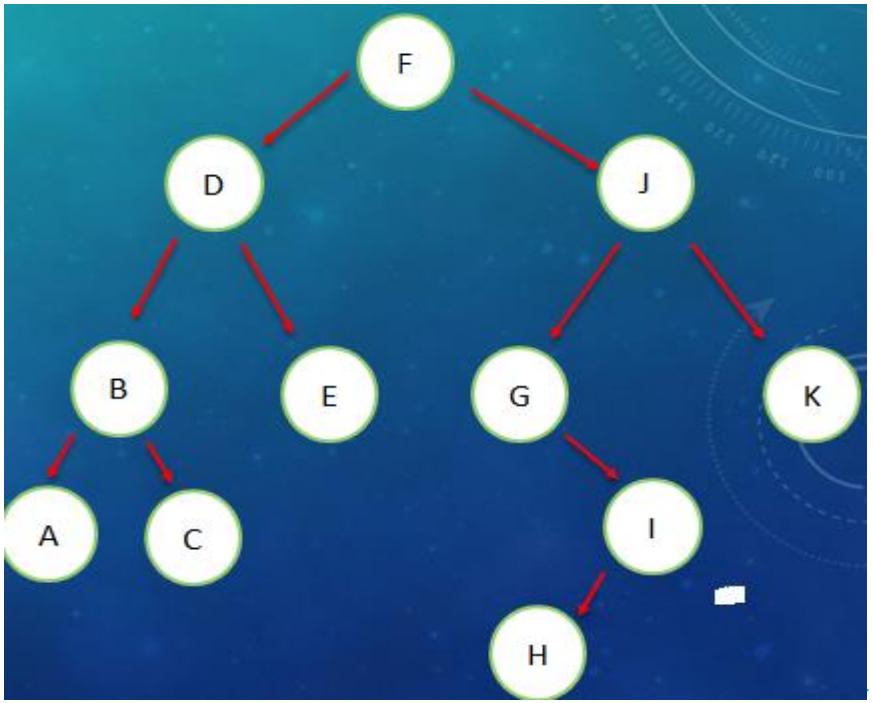
- Pre order traversal - traversing binary tree in the order of ***parent, left and right*** **(PLR).**
- In order traversal - traversing binary tree in the order of ***left, parent and right*** **(LPR).**
- Post order traversal - traversing binary tree in the order of ***left, right and parent*** **(LRP).**

Pre Order Traversal (Recursive)

- Steps:
  1. Visit the **root** node
  2. Traverse the **left sub tree** in preorder
  3. Traverse the **right sub tree** in preorder

PLR



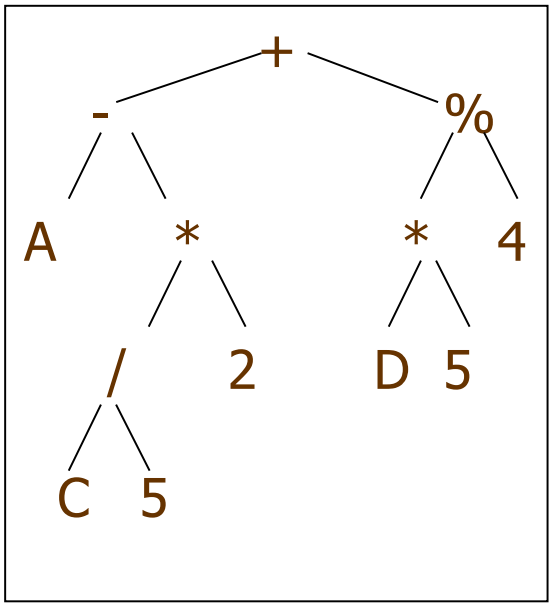


Pre Order Traversal (Recursive)

```
void preorder (node *CurrNodePtr){

   if(CurrNodePtr ! = NULL) {
        cout<< CurrNodePtr->data; // or any operation
        preorder(CurrNodePtr->left);
        preorder(CurrNodePtr->right);
      }
}
```

PLR



Output: + - A * / C 5 2 % * D 5 4

In Order Traversal (Recursive)

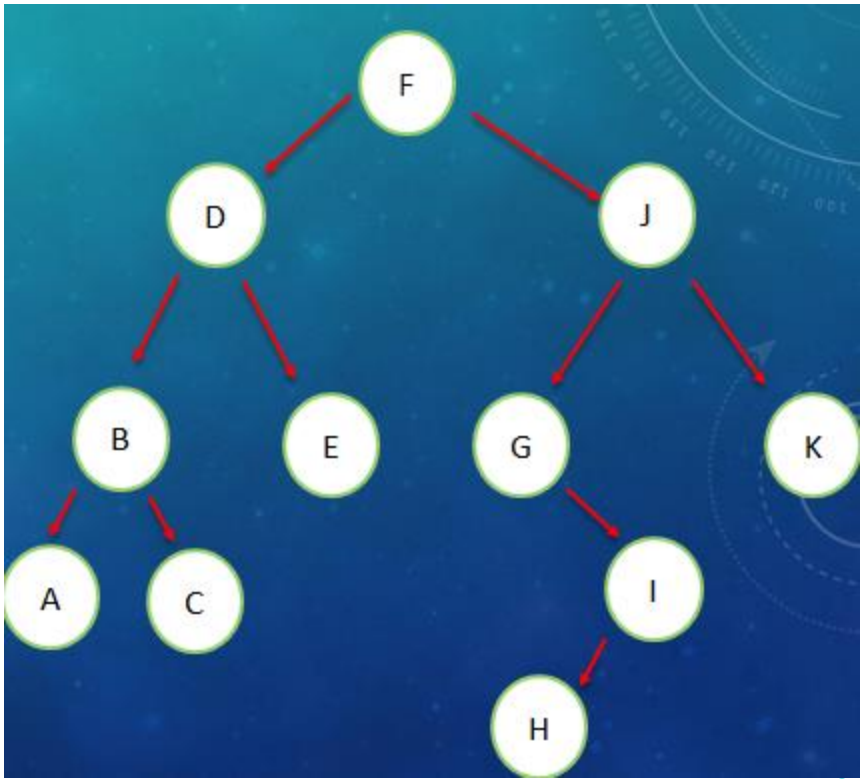
Steps:

1.Traverse the **left sub tree** in-order **LPR**

2.Visit the **root** node

3.Traverse the **right sub tree** in-order

[The **left sub tree** is traversed **recursively**, before visiting the root. After visiting the root the **right sub tree** is traversed **recursively**, in order fashion.]

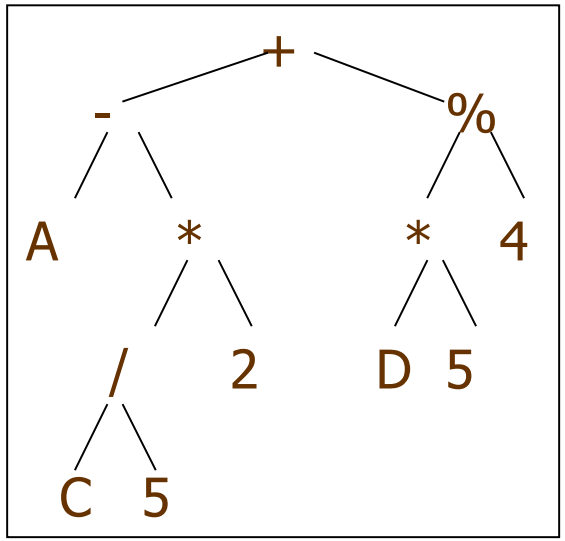


In Order Traversal (Recursive)

```
void inorder (node *CurrNodePtr){

    if(CurrNodePtr ! = NULL){
        inorder(CurrNodePtr->left);
        cout<< CurrNodePtr->data;
        inorder(CurrNodePtr->right);
    }
}
```

LPR



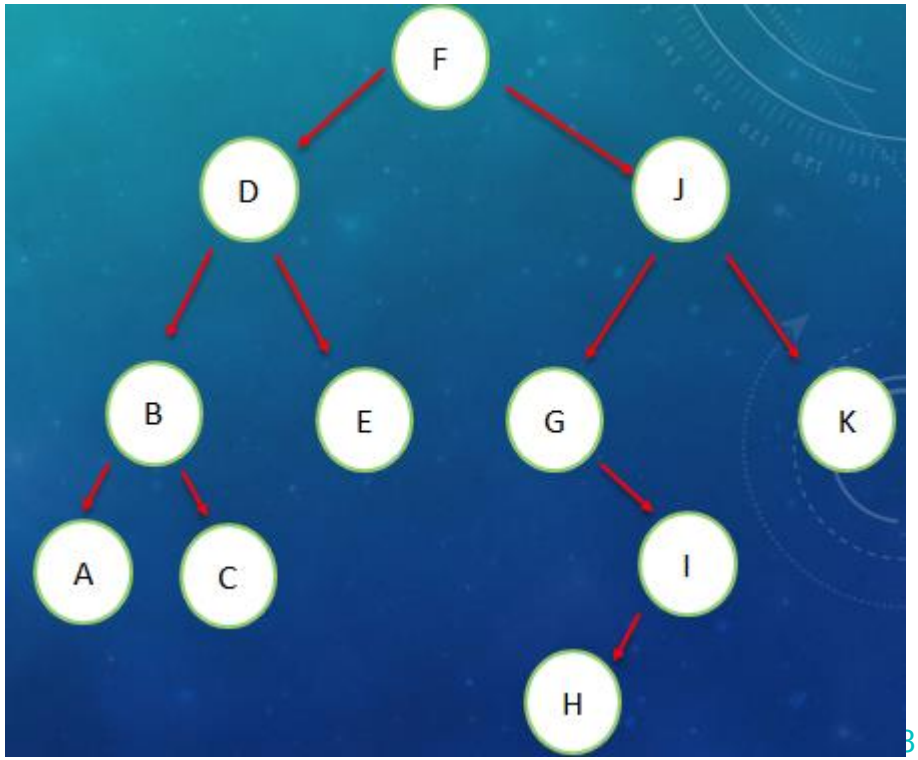
Output: A – C / 5 * 2 + D * 5 % 4

Post Order Traversal (Recursive)

- Steps:

LRP

1. Traverse the **left sub tree** in post order
2. Traverse the **right sub tree** in post order
3. Visit the **root** node

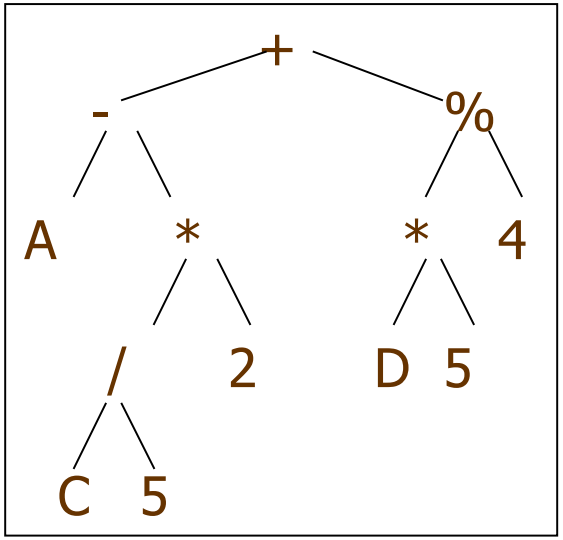


Post Order Traversal (Recursive)

```
void postorder (node *CurrNodePtr){

    if(CurrNodePtr ! = NULL){
        postorder(CurrNodePtr->left);
        postorder(CurrNodePtr->right);
        cout<< CurrNodePtr->data;
     }
}
```

LRP



Output: A C 5 / 2 * - D 5 * 4 % +