# Binary Trees

- A **tree** is a data structure that is made of nodes and pointers, much like a linked list. The difference between them lies in how they are organized:
  - In a linked list each node is connected to one "successor" node (via next pointer), that is, it is linear.
  - In a tree, the nodes can have several next pointers and thus are not linear.

- The top node in the tree is called the **root** and all other nodes branch off from this one.

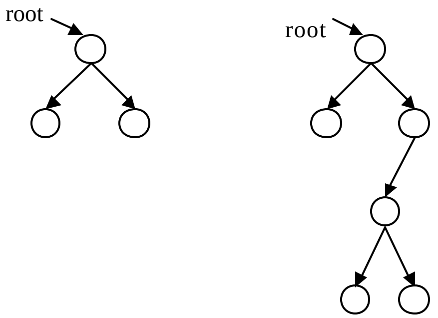
- Every node in the tree can have some number of children. Each child node can in turn be the parent node to its children and so on.

- A common example of a tree structure is the binary tree.

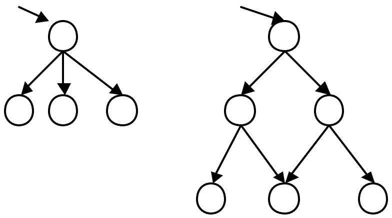
**Definition:** A **binary tree** is a tree that is limited such that each node has only two children.



**Examples**:

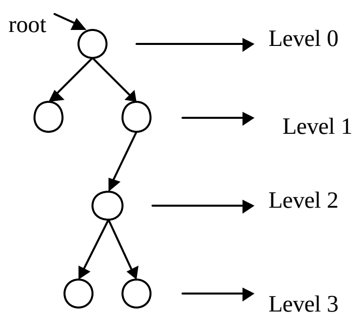



- The following are NOT binary trees:



## Definitions:

- If n1 is the root of a binary tree and n2 is the root of its left or right tree, then n1 is the **parent** of n2 and n2 is the **left** or **right child** of n1.
- A node that has no children is called a **leaf**.
- The nodes are **siblings** if they are left and right children of the same parent.
- The level of a node in a binary tree:
  - The root of the tree has level 0
  - The level of any other node in the tree is one more than the level of its parent.




**Exercise**: Construct all possible 5 binary trees with 3 nodes.



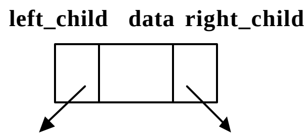
## Implementation

- A binary tree has a natural implementation in linked storage. A separate pointer is used to point the tree (e.g. root)

```
root = NULL;        // empty tree
```

- Each node of a binary tree has both left and right subtrees which can be reached with pointers:

```
struct tree_node{
    int data;
    struct tree_node *left_child;
    struct tree_node *right_child;
};
```



# Traversal of Binary Trees

Linked lists are traversed from first to last sequentially. However, there is no such natural linear order for the nodes of a tree. Different orderings are possible for traversing a binary tree. Three of these traversal orderings are:

- **Preorder**
- **Inorder**
- **Postorder**

These names are chosen according to the step at which the root node is visited.

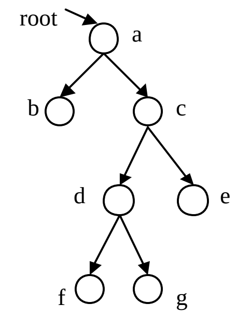
– With **preorder** traversal the node is visited *before* its left and right subtrees,

– With **inorder** traversal the root is visited *between* the subtrees,

– With **postorder** traversal the root is visited *after* both subtrees.



**Example**:




Preorder: a b c d f g e

Inorder: b a f d g c e

Postorder: b f g d e c a

- Because of the recursively defined structure of a binary tree, these traversal algorithms are inherently recursive.

- Recursive definition of a binary tree: A binary tree is either
  - Empty, or
  - A node (called root) together with two binary trees (called left subtree and the right subtree of the root)

- Tree traversal algorithms exploit this fact.

**Preorder**

```
void preorder(struct tree_node * p)
{  if (p !=NULL) {
       printf("%d\n", p->data);
       preorder(p->left_child);
       preorder(p->right_child);
   }
}
```

**Inorder**

```
void inorder(struct tree_node *p)
{  if (p !=NULL) {
       inorder(p->left_child);
       printf("%d\n", p->data);
       inorder(p->right_child);
   }
}
```

**Postorder**

```
void postorder(struct tree_node *p)
{  if (p !=NULL) {
       postorder(p->left_child);
       postorder(p->right_child);
       printf("%d\n", p->data);

   }
}
```



## Finding the maximum value in a binary tree

```
int FindMax(struct tree_node *p) {

     int root_val, left, right, max;
     max = -1; // Assuming all values in the
               // are positive integers

     if (p!=NULL) {
       root_val = p -> data;
       left = FindMax(p ->left_child);
       right = FindMax(p->right_child);

       // Find the largest of the three values.
       if (left > right)
            max = left;
       else
           max = right;
       if (root_val > max)
           max = root_val;
     }
     return max;
}
```

## Adding up all values in a Binary Tree

```
int add(struct tree_node *p) {

   if (p == NULL)
      return 0;
   else
      return (p->data + add(p->left_child)+
                add(p->right_child));
}
```



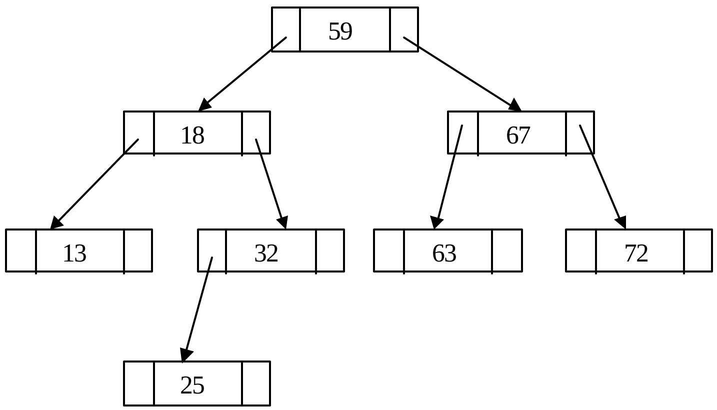
## *Binary Search* Tree

Binary search tree is a binary tree that is either empty or in which each node contains a data value that satisfies the following:

a) all data values in the left subtree are smaller than the data value in the root.
b) the data value in the root is smaller than all values in its right subtree.
c) the left and right subtrees are also binary search tees.

This structure allows us to quickly search for a particular value.

**Example:**



## Searching the Binary Search Tree

```
struct tree_node{
    int data;
    struct tree_node *left_child;
    struct tree_node *right_child;
};


int treeSeach( struct tree_node *p, int target)
{
     if (p==NULL)
        return 0;
     else
       if (p->data == target)
         return 1;
       else if (p->data > target)
         return(treeSearch(p->left_child));
       else
         return(treeSearch(p->right_child));
}
```



## Adding Nodes to a BST

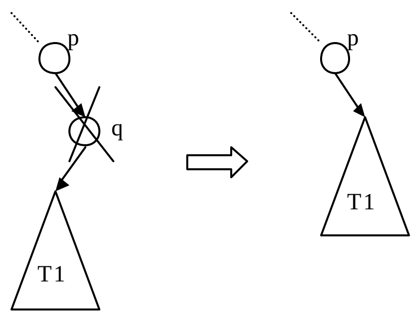
In a BST, a new node will always be inserted at a NULL pointer. We never have to rearrange existing nodes to make room for a new one.

**Example:**
Add values 43, 65, 66 to the example tree given above.

## Deleting Nodes from a BST

a) Deleting a leaf node: Replace the link to the deleted node by NULL.

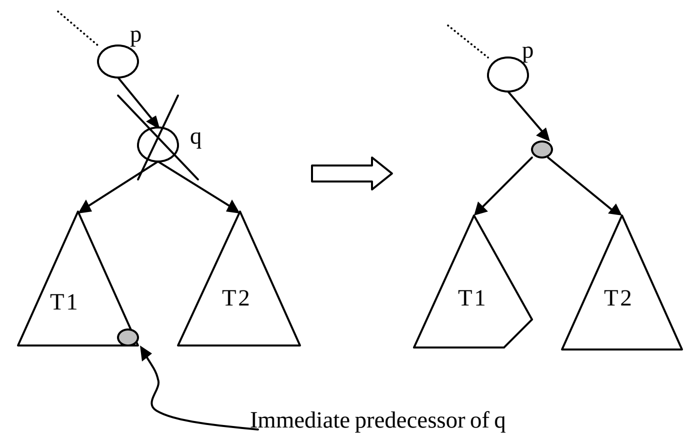
b) Deleting a node with one empty subtree.




c) Deleting a node with both left and right subtrees. Any deleted value that has two children must be replaced by an existing value that is one of the following:

- The largest value in the deleted node's left subtree
- The smallest value in the deleted node's right subtree.





## Exercises:

1) Write a function that will count the leaves of a binary tree.

```
int num_of_leaves(struct tree_node *p)
{
  if (p == NULL)

     return  ___________;

  else  /* check if it is a leaf */

     if (____________ && _______________)

        return 1;

     else

        return (num_of_leaves(___________ +

                num_of_leaves(______________);

}
```



2) Write a function that will find the height of a binary tree. (Height of an empty tree is zero).

```
int height(struct tree_node *p)
{
    int lefth, righth;

    if (p==NULL)

       return ________;

    else {

       lefth = height(___________);

       righth = height(___________);

       if (lefth > righth)

          return (___________);

       else

          return (___________);

    }

}
```

3) Write a function that will interchange all left and right subtrees in a binary tree.

```
void interchange(struct tree_node *p)
{
    struct tree_node *temp;

    if (p != NULL) {

        interchange (___________);

        interchange(____________);

        temp = ________;

        p->left = ________;

        p->right = ________;

    }

}
```