

# Chapter 6 Tree

## 6.1. TREE CONCEPT AND DEFINITIONS

There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement. Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential. Then there are other specialized data structures like, stacks and queues that allows us to solve complicated problems (e.g. Maze traversal) using these restricted data structures. One other data structure is the hash table that allows users to program applications that require frequent search and updates. They can be done in  $O(1)$  in a hash table.

One of the disadvantages of using an array or linked list to store data is the time necessary to search for an item. Since both the arrays and Linked Lists are **linear structures** the time required to search a “linear” list is proportional to the size of the data set. For example, if the size of the data set is  $n$ , then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of  $n$ . So imagine doing the search on a linked list (or array) with  $n = 10^6$  nodes. Even on a machine that can do million comparisons per second, searching for  $m$  items will take roughly  $m$  seconds. This not acceptable in today’s world where speed at which we complete operations is extremely important. Time is money. Therefore it seems that better (more efficient) data structures are needed to store and search data.

The data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the minimum time complexity is  $O(n)$  as the data have to be accessed in order/sequentially. As the size of data increases, the time to access each data element increases linearly which is not acceptable in today's computational world. The tree data structure allows quicker in  $O(\log n)$  even for huge amount of data and easier access to the data as it is a non-linear data structure. In this chapter you will learn about another data structure called **tree**. A tree is a widely used abstract data type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

## Tree Data Structure

A tree is a non-linear data structure which organizes data in hierarchical structure in a recursive manner. It is a collection of nodes connected by directed (or undirected) edges. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more **subtrees**, each of which is also a tree.

A tree has following general properties:

- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*
- A unique path traverses from the root to each node.

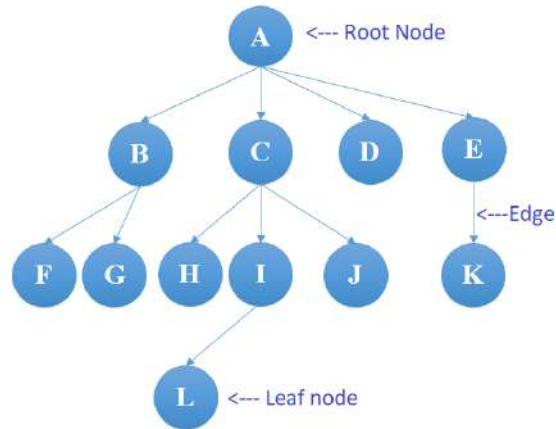


Figure 6.1: A tree data structure

In the above Figure 6.1, the node A is **root** of the tree. Root of a tree is a special and first node from which the tree originates and it therefore doesn't have **parent**. A tree can have only one root. A is a **parent** of nodes B, C, D, and E. These nodes B, C, D, and E are children of A. The B is parent of F and G. D has no child. E has a single child K. The root A has four subtrees. The node C has 3 subtrees. The L has no subtrees. A connecting link between any two nodes is called as **edge**. The arrowheads on the edges indicate the direction of the connection.

Each node can have *arbitrary* number of children (or subtrees). Nodes with no children are called **leaves nodes** or **external nodes**. In the Figure 6.1, D, F, G, J, K and L are leaves. Nodes, which are not leaves, are called **internal nodes**. Internal nodes have at least one child. The **degree of a node** is the total number of children the node has. The degree of node C is 3. The degree of node I is 1. The degree of node H is zero. The **degree of a tree** is the greatest number of degree of node that the tree has. The degree of the tree in Figure 6.1 is 4 since the highest degree of node A is 4.

Nodes with the same parent are called **siblings**. In the Figure 6.1, B, C, D and E are called siblings. The **depth of a node** is the number of edges from the root to the node. The depth of K is 2 and the depth of L is 3. The **height of a node** is the number of edges from the node to the deepest leaf. The height of C is 2. The **height of a tree** is a height of a root. The root node is said to be at **level 0** and children of root are at level 1 and so on. The node L is at level 3.

A sequence of nodes and edges from one node to another node is called as a **path** between those two nodes. The path between nodes A and L is **A-C-I-L**. Every node has zero or one or more subtree. This means a child of a parent node is a **subtree** which has every property of tree (See Figure 6.2).

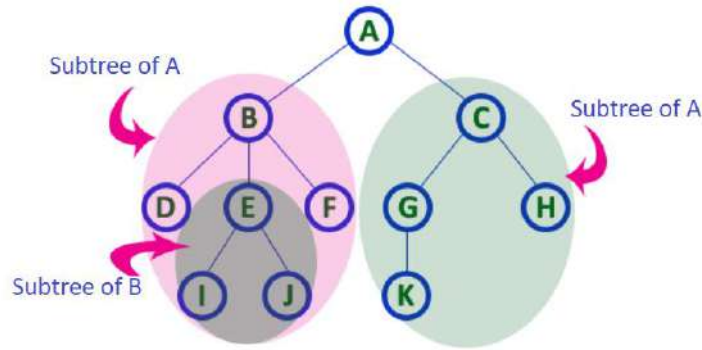


Figure 6.2: A tree data structure showing subtree of node A and B

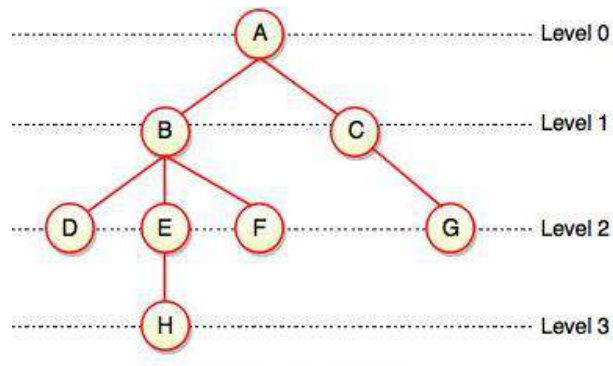


Figure 6.3: Levels of tree

## Real World Examples that uses Tree Data Structure

One example of a tree structure that you probably use every day is a file system. In a file system, directories, or folders, are structured as a tree.

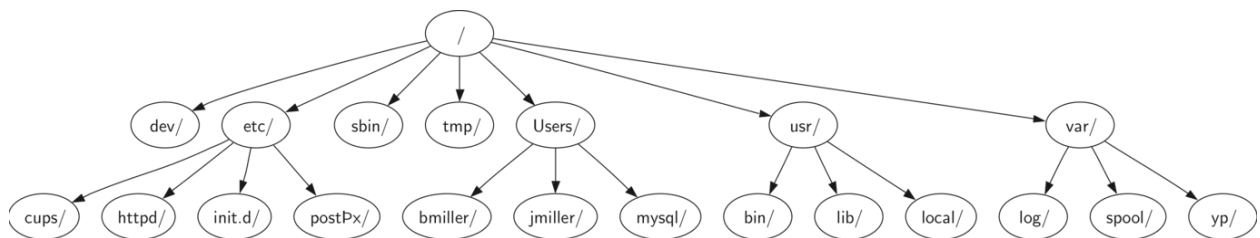


Figure 6.4: A small part of the Unix file system hierarchy

In the file system tree, you can follow a path from the root ( / ) to any directory. That path will uniquely identify that subdirectory (and all the files in it). Another important property of trees, derived from their hierarchical nature, is that you can move entire sections of a tree (called a subtree) to a different position in the tree without affecting the lower levels of the hierarchy. For example, we could take the entire subtree starting with /etc/, detach etc/ from the root and reattach it under usr/.

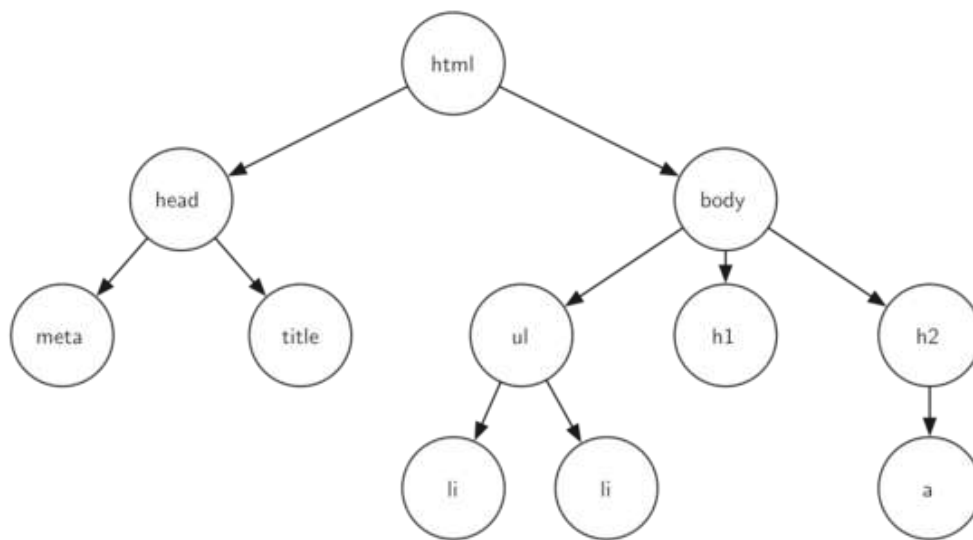
Another example of a tree is a web page. The following is an example of a simple web page written using HTML. Figure 6.5 shows the tree that corresponds to each of the HTML tags used to create the page.

```

<html>
<head>
  <title>simple</title>
</head>
<body>
  <h1>A simple web page</h1>
  <ul>
    <li>List item one</li>
    <li>List item two</li>
  </ul>
  <h2><a href="https://www.google.com">Google</a></h2>
</body>
</html>

```

(a) the HTML tags used to create the page



(b) A tree corresponding to the markup elements of a web page

Figure 6.5: HTML tags and its representation in Tree

The HTML source code and the tree accompanying the source illustrate another hierarchy. Notice that each level of the tree corresponds to a level of nesting inside the HTML tags. The first tag in the source is `<html>` and the last is `</html>`. All the rest of the tags in the page are inside the pair.

## A General Tree

Many organizations are hierarchical in nature, such as the military and most businesses. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, and so on. If we wanted to model this company with

a data structure, it would be natural to think of the president in the root node of a tree, the vice presidents at level 1, and their subordinates at lower levels in the tree as we go down the organizational hierarchy.

Because the number of vice presidents is likely to be more than two, this company's organization cannot easily be represented by a binary tree. We need instead to use a tree whose nodes have an arbitrary number of children. Unfortunately, when we permit trees to have nodes with an arbitrary number of children, they become much harder to implement than binary trees. To distinguish them from binary trees, we use the term **general tree**.

A general tree is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree).

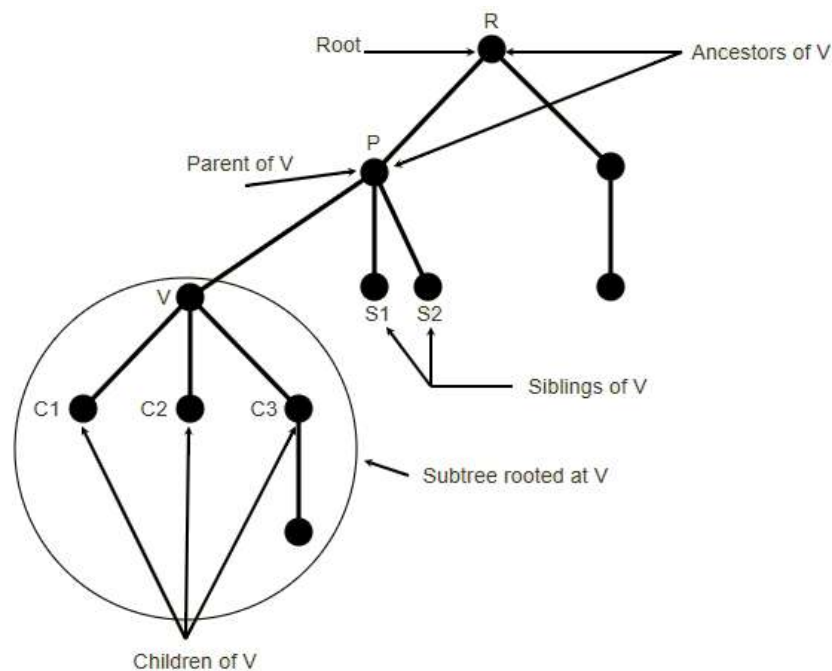


Figure 6.6: Notation for general trees. Node P is the parent of nodes V, S1, and S2. Thus, V, S1, and S2 are children of P. Nodes R and P are ancestors of V. Nodes V, S1, and S2 are called siblings. The oval surrounds the subtree having V as its root.

Each node in a tree has precisely one parent, except for the root, which has no parent. From this observation, it immediately follows that a tree with  $n$  nodes must have  $n-1$  edges because each node, aside from the root, has one edge connecting that node to its parent.

## 6.2. BINARY TREE AND ITS VARIATIONS

### Definitions and properties

A general tree has no restrictions on the number of children each node can have. Binary tree is a special type of tree data structure and has a restriction on number of children of a node. Every node in a binary tree can have maximum of two children. The children are referred to as left child or right child. In a binary tree,

every node can have either 0 children or 1 child or 2 children but not more than 2 children. This is one of the most commonly used trees.

A binary tree is made up of a finite set of elements called nodes. This set either is empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root. Disjoint means that they have no nodes in common. The roots of these subtrees are children of the root.

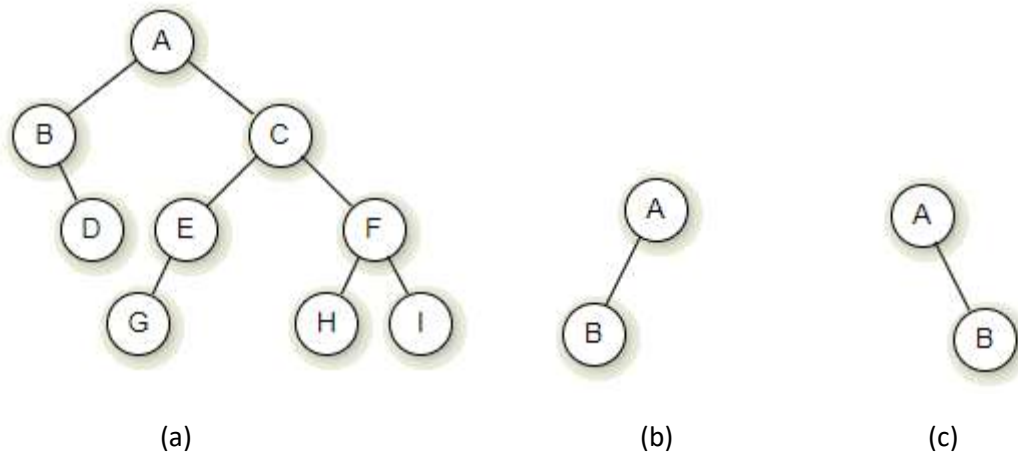


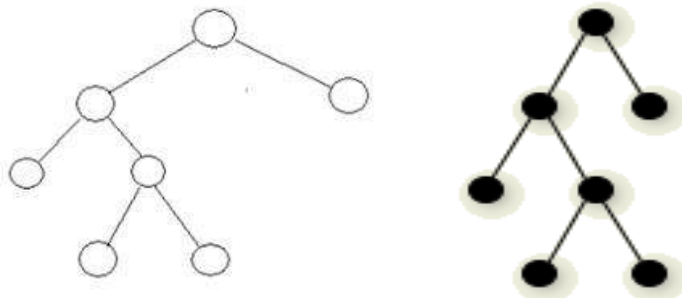
Figure 6.7: A binary Trees

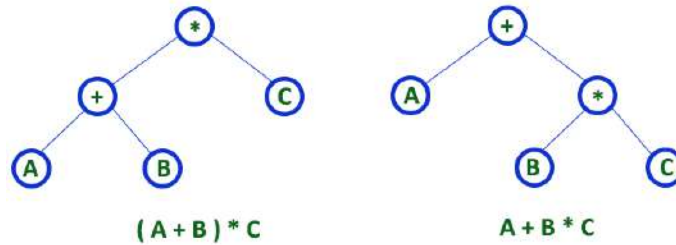
## Types of Binary Tree

### 1) Full Binary Tree

A binary tree in which each node has exactly zero or two children is called a **full binary tree**. It is also called as strictly binary tree. The **Huffman coding tree** is an example of a full binary tree. In a full tree, there are no nodes with exactly one child. In a full binary tree:

$$\text{Number of Leaf nodes} = \text{Number of Internal nodes} + 1$$

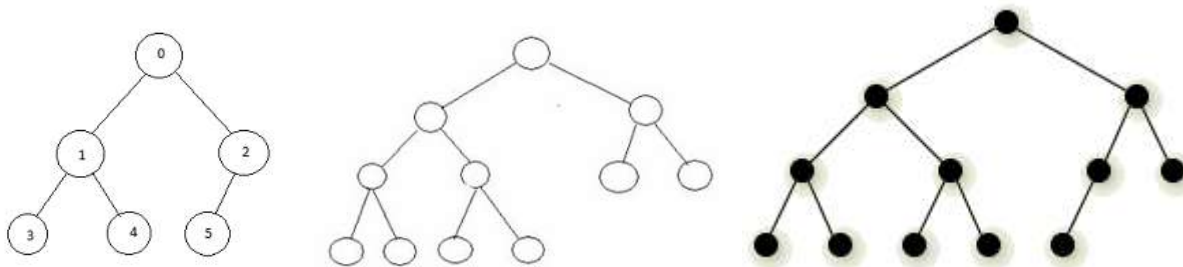




## 2) Complete Binary Tree

A complete binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right. In the complete binary tree of height  $d$ , all levels except possibly level  $d$  are completely full. The bottom level has its nodes filled in from the left side.

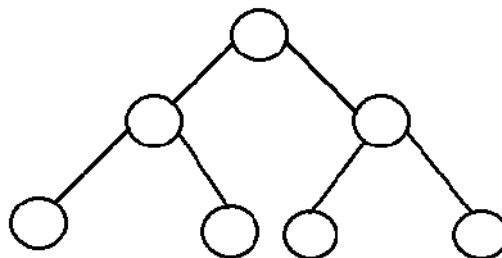
A **complete binary tree** is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. The **heap** data structure is an example of a complete binary tree. A complete binary tree of the height  $h$  has between  $2^h$  and  $2^{(h+1)}-1$  nodes. Here are some examples:



A complete binary tree is also called as almost complete binary tree.

## 3) Perfect binary tree

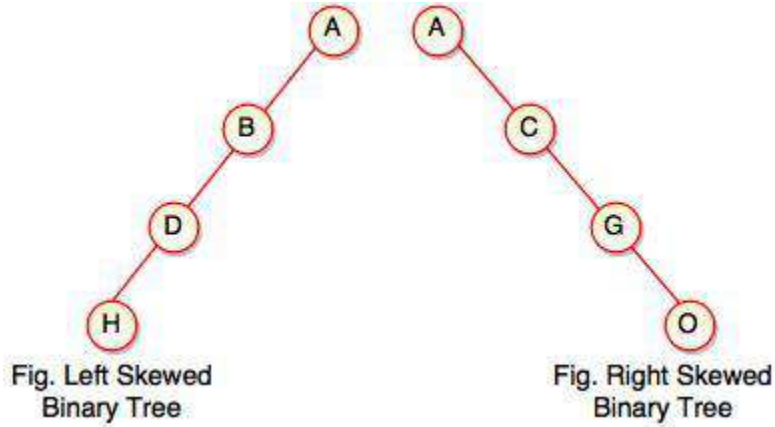
It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



Total number of nodes in a Perfect Binary Tree with height  $h$  is  $2^{h+1} - 1$ .

#### 4) Skewed Binary Tree

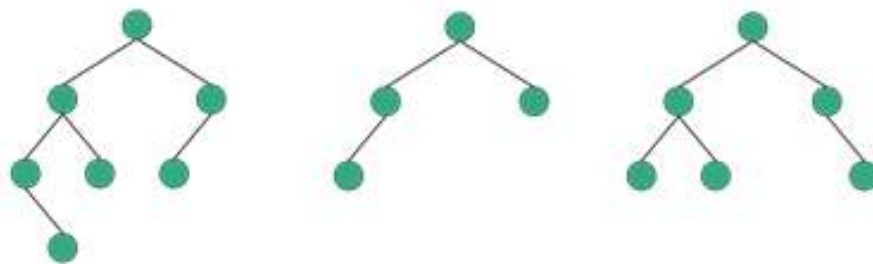
If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**. In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.



In a left skewed tree, most of the nodes have the left child without corresponding right child. In a right skewed tree, most of the nodes have the right child without corresponding left child.

#### 5) Balanced Binary Tree

Balanced Binary Tree is a Binary tree in which height of the left and the right sub-trees of every node may differ by at most 1. AVL Tree and Red-Black Tree are well-known data structure to generate/maintain Balanced Binary Search Tree. Search, insert and delete operations cost  $O(\log n)$  time in such balanced tree.



#### 6) 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.





## Representation of Binary Tree in Memory

1. Array Representation of Binary Tree
2. Linked List Representation of Binary Tree

In array representation of a binary tree, we use 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  $2n + 1$ . If a tree is complete binary tree, then this method will be very efficient.

1. The root R is stored in TREE [0]
2. For a node at index i, its left child is stored in TREE [2 \* i + 1] and its right child is stored into TREE [2 \* i + 2].

```

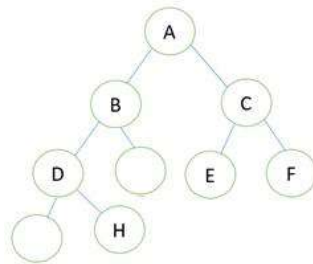
graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    B --- H((H))
    C --- E((E))
    C --- F((F))
  
```

It can be represented in array as:

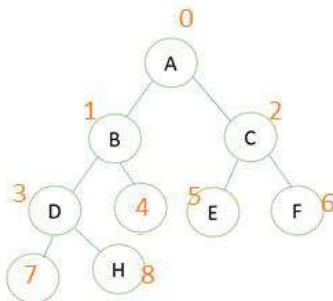
0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	-	E	F	-	H		

If you follow the following steps, it will be easy to represent a binary tree using array:

1. If the tree is already complete binary tree then go to step 3.
2. If the tree is not a complete binary tree, make it complete binary tree by adding dummy nodes as below:



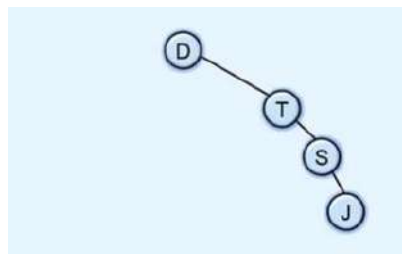
3. Now number the each node of the tree starting from 0. Assign 0 for root node, 1 to left child, 2 to right child and so on as:



4. Now assign the node value to the array with index  $i$  where  $i$  = assigned number to the node in step 3. Ignore the dummy nodes. Then the array becomes:

0	1	2	3	4	5	6	7	8	9	10
A	B	C	D	-	E	F	-	H		

**(Student practice:** represent the following binary tree using array representation method)

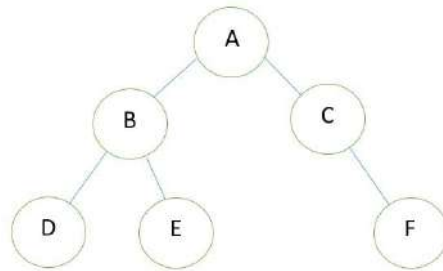


## 8) Linked List Representation of Binary Tree

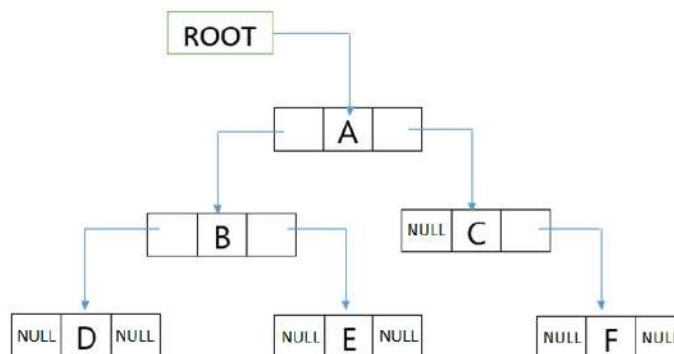
We can represent the binary tree using linked list data structure. For this each node in linked list represent a node of the binary tree. Each list node needs three fields. One field to hold data and the other two fields to hold the addresses of the left child node and right child node.



This node can be recursively used to represent a binary tree. If a node doesn't have a left child, it will be assigned a null value. If the node doesn't have a right child, it will be assigned a null value. Consider a binary tree as



This binary tree is represented using linked list as shown as

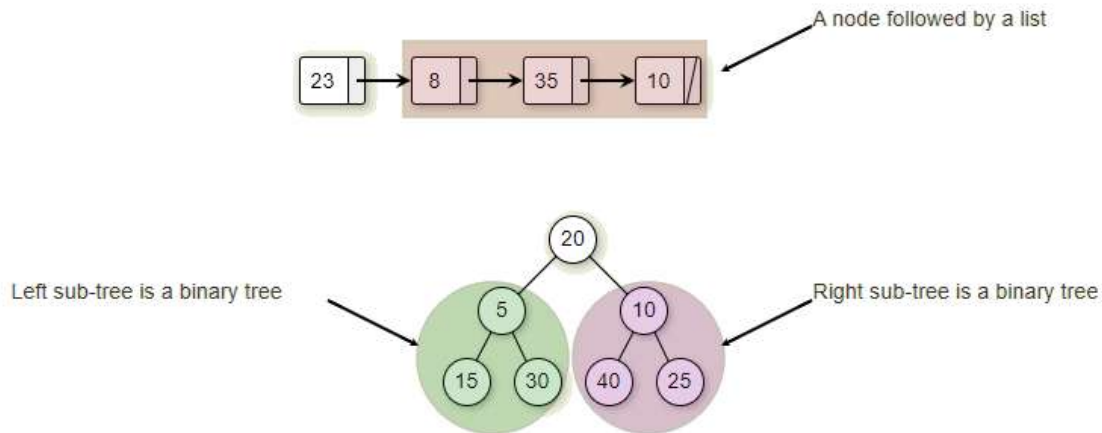


*A node implementation in C++*

```
class node {  
    int data;  
    node* left;  
    node* right;  
};
```

## Binary Tree as a Recursive Data Structure

A recursive data structure is a data structure that is partially composed of smaller or simpler instances of the same data structure. For example, linked lists and binary trees can be viewed as recursive data structures. A list is a recursive data structure because a list can be defined as either (1) an empty list or (2) a node followed by a list. A binary tree is typically defined as (1) an empty tree or (2) a node pointing to two binary trees, one its left child and the other one its right child.



The recursive relationships used to define a structure provide a natural model for any recursive algorithm on the structure.

### 6.3. BASIC OPERATION IN BINARY TREE

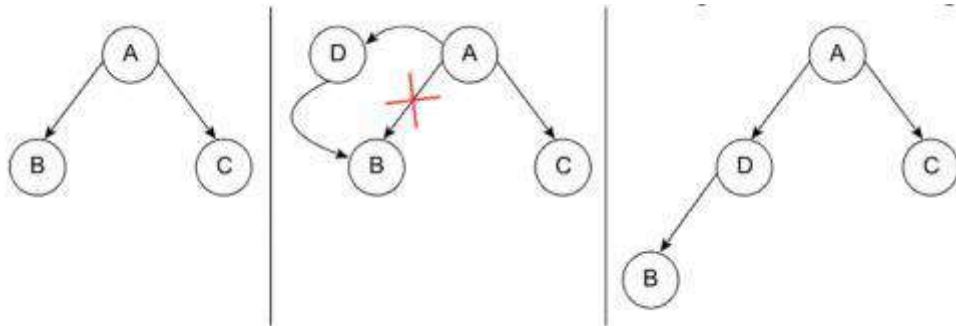
Like other data structures, various operation can be done with the data stored in binary tree. The basic operations that can be done in binary tree are:

#### Insertion

Nodes can be inserted into binary trees in between two other nodes or added after a leaf node. In binary trees, a node that is inserted is specified as to which child it is.

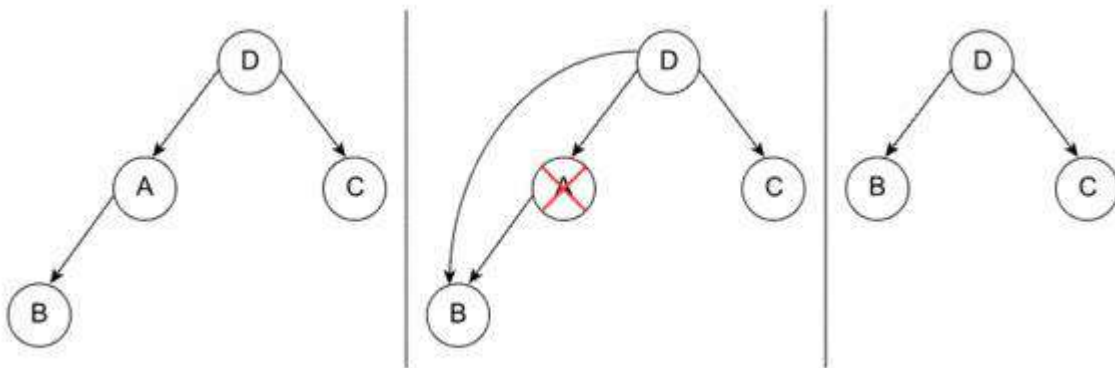
**Case I:** Say that the **external node** being added onto is node A. To add a new node after node A, A assigns the new node as one of its children and the new node assigns node A as its parent.

**Case II:** The process of inserting a node into a binary tree Insertion on **internal nodes** is slightly more complex than on external nodes. Say that the internal node is node A and that node B is the child of A. (If the insertion is to insert a right child, then B is the right child of A, and similarly with a left child insertion.) A assigns its child to the new node and the new node assigns its parent to A. Then the new node assigns its child to B and B assigns its parent as the new node.



## Deletion

Deletion is the process whereby a node is removed from the tree.



### Case I: Node with no child

If a node has no children (external node), deletion is accomplished by setting the child of A's parent to null.

### Case II: Node with one child

If it has one child, set the parent of A's child to A's parent and set the child of A's parent to A's child.

### Case II: Node with two children

In a binary tree, a node with two children cannot be deleted unambiguously. However, in certain binary trees (including binary search trees) these nodes can be deleted, though with a rearrangement of the tree structure.

## Traversal

Pre-order, in-order, and post-order traversal visit each node in a tree by recursively visiting each node in the left and right subtrees of the root.

## 6.4. BINARY TREE TRAVERSALS

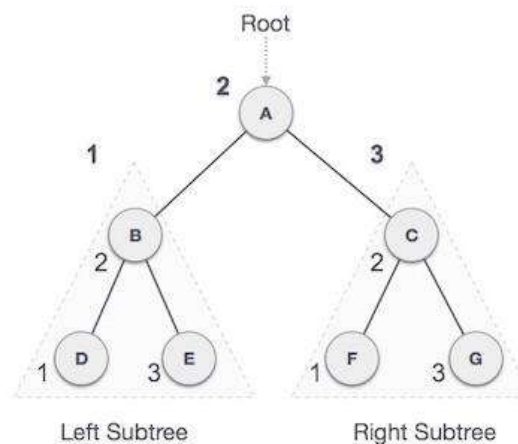
Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a **traversal**. Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.

There are three ways of traversing a tree :

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

### In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. If a binary tree is traversed **in-order**, the output will produce sorted values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The in-order **enumeration** for this tree is **D → B → E → A → F → C → G**.

#### 1) Algorithm

Until all nodes are traversed repeat the following steps:

1. Visit the left subtree in in-order.
2. Visit root node.
3. Visit right subtree in in-order.

## 2) Implementation of in-order traversal

*Tree node:*

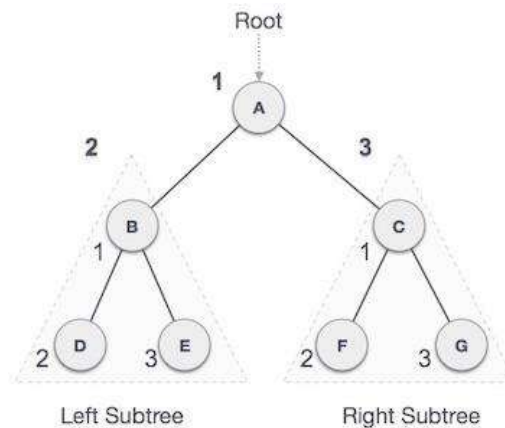
```
struct node {  
    int data;  
    struct node* left;  
    struct node* right;  
};
```

*In-order function:*

```
void inorder(struct node* root){  
    if(root == NULL) return;  
    inorder(root->left);  
    printf("%d ->", root->data);  
    inorder(root->right);  
}
```

## Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The pre-order *enumeration* for this tree is **A → B → D → E → C → F → G**.

## 1) Algorithm

Until all nodes are traversed repeat the following steps:

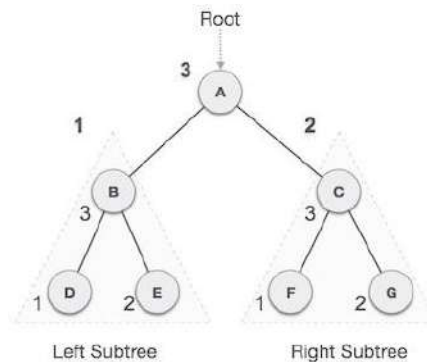
1. Visit root node.
2. Visit the left subtree in pre-order.
3. Visit right subtree in pre-order.

## 2) Implementation of in-order traversal

```
void preorder(struct node* root){  
    if(root == NULL) return;  
    printf("%d ->", root->data);  
    preorder(root->left);  
    preorder(root->right);  
}
```

## Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post- order **enumeration** for this tree is **D → E → B → F → G → C → A**.

## 1) Algorithm

Until all nodes are traversed repeat the following steps:

1. Visit the left subtree in pre-order.
2. Visit right subtree in pre-order.
3. Visit root node.



## 2) Implementation of in-order traversal

```
void postorder(struct node* root) {
    if(root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ->", root->data);
}
```

## Tree traversal implementation in C

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node* left;
    struct node* right;
};

void inorder(struct node* root){
    if(root == NULL) return;
    inorder(root->left);
    printf("%d ->", root->data);
    inorder(root->right);
}

void preorder(struct node* root){
    if(root == NULL) return;
    printf("%d ->", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(struct node* root) {
    if(root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ->", root->data);
}
```

```

struct node* createNode(value){
    struct node* newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

struct node* insertLeft(struct node *root, int value) {
    root->left = createNode(value);
    return root->left;
}

struct node* insertRight(struct node *root, int value){
    root->right = createNode(value);
    return root->right;
}

int main(){
    struct node* root = createNode(1);
    insertLeft(root, 12);
    insertRight(root, 9);

    insertLeft(root->left, 5);
    insertRight(root->left, 6);

    printf("Inorder traversal \n");
    inorder(root);

    printf("\nPreorder traversal \n");
    preorder(root);

    printf("\nPostorder traversal \n");
    postorder(root);
}

```

## 6.5. BINARY SEARCH TREE

A binary search tree (BST) is a binary tree that conforms to the following condition, known as the binary search tree property. All nodes stored in the left subtree of a node whose key value is  $K$  have key values less than or equal to  $K$ . All nodes stored in the right subtree of a node whose key value is  $K$  have key values greater than  $K$ . A “balanced” binary search tree can be searched in  $O(\log n)$  time, where  $n$  is the number of nodes in the tree.

In other words, a **binary search tree** (BST) is a binary tree where each node has a Comparable key value and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

Below figures shows two BSTs for a collection of values. One consequence of the binary search tree property is that if the BST nodes are printed using an **in-order traversal**, then the resulting enumeration will be in sorted order from lowest to highest.

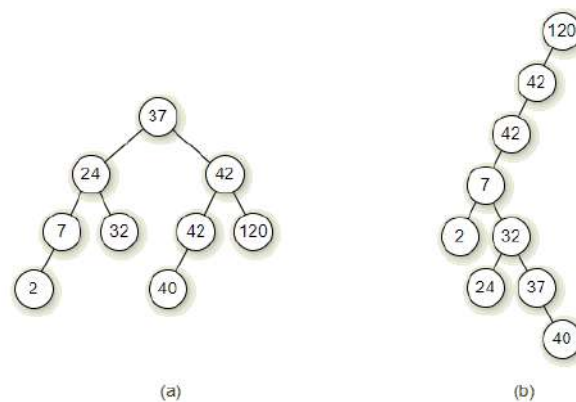


Figure: Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

A binary tree is a binary search tree (BST) if and only if an in-order traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.

**Definition 1:** A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are  $<$  (less) than the key in its parent node
3. The keys in the right subtree  $>$  (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

**Definition 2:** A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:

1. The left sub-tree of a node has a key less than or equal to its parent node's key.
2. The right sub-tree of a node has a key greater than to its parent node's key.

Some definitions of BST allows the duplicate key values (Definition 2) while others not (Definition 2). It depends upon the nature of application for which we are building BST. If the application does not allow nodes with equal keys, then duplication won't be the case. If duplicate keys are allowed, our convention will be to insert the duplicate key in the left subtree.

## 6.6. BINARY SEARCH TREE OPERATIONS

There are a number of operations on BST's that are important to understand. We will discuss some of the basic operations such as how to insert a node into a BST, how to delete a node from a BST and how to search for a node in a BST. The basic BST operations includes:

1. Insertion
2. Deletion
3. Searching
4. Traversing (in-order, pre-order and post-order)

### Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node.

We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, then we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left or right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert (N, T) = N if T is empty  
              = insert (N, T.left) if N < T  
              = insert (N, T.right) if N > T

#### *Implementation of insert operation using recursive method*

```
struct node* insert(struct node* root, int data){
    if (root == NULL) return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}
```

### *Simple algorithm (non-recursive) for insertion*

1. Create a new node with a value and set its left and right to NULL.
2. Check whether the tree is empty or not.
3. If the tree is empty, set the root to a new node.
4. If the tree is not empty, check whether a value of new node is smaller or larger than the node (here it is a root node).
5. If a new node is smaller than or equal to the node, move to its left child.
6. If a new node is larger than the node, move to its right child.
7. Repeat the process until we reach to a leaf node.

## Searching a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on  $N < T$  or  $N > T$ . A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not.

```
Search(N, T) = false  if T is empty
              = true   if T = N
              = search(N, T.left) if N < T
              = search(N, T.right) if N > T
```

### *Implementation of search operation using recursive method*

```
struct node* search(struct node* root, int data)
{
    // Base Cases: root is null or data is present at root
    if (root == NULL || root->data == data)
        return root;

    // data is greater than root's data
    if (root->data < data)
        return search(root->right, data);

    // data is smaller than root's data
    return search(root->left, data);
}
```

### *Implementation of search operation*

```
void search(Node* &cur, int item, Node* &parent)
{
    while (cur != NULL && cur->data != item)
    {
        parent = cur;
```

```

        if (item < cur->data)
            cur = cur->left;
        else
            cur = cur->right;
    }
}

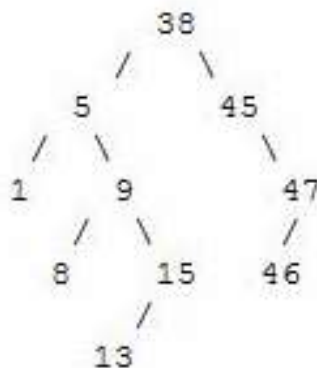
```

### *Simple algorithm (non-recursive) for search operation*

1. Read the element to be searched.
2. Compare this element with the value of root node in a tree.
3. If element and value are matching, display "Node is found" and terminate the function.
4. If element and value are not matching, check whether an element is smaller or larger than a node value.
5. If an element is smaller, continue the search operation in left subtree.
6. If an element is larger, continue the search operation in right subtree.
7. Repeat 2 to 6 steps until we reach to a leaf node.
8. If an element with search value is found, display "Element is found" and terminate the function.
9. If we reach to a leaf node and the search value is not match to a leaf node, display "Element is not found" and terminate the function.

## Deleting a node

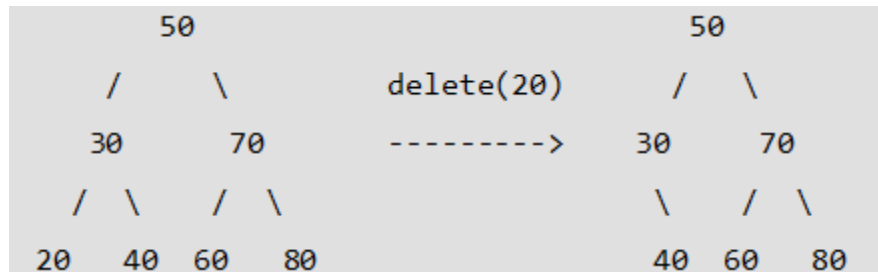
A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree.



The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children)? The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

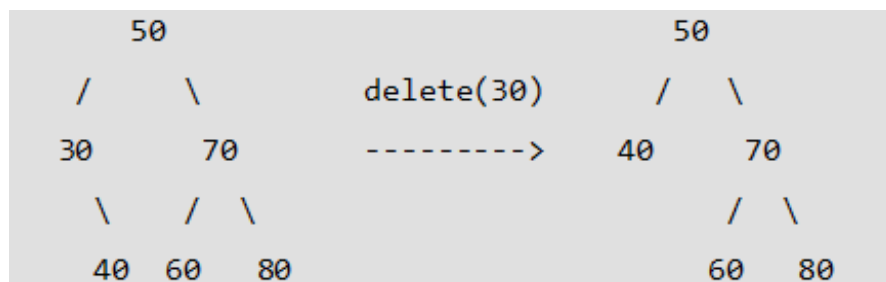
### Case 1: Node to be deleted is leaf

- Simply remove from the tree. This is a very easy case. For example if you want to delete the node 20, then simple assign a NULL value to the left-child pointer of its parent (30) and delete the node 20.

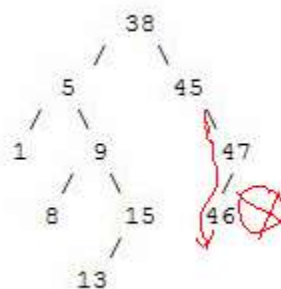


### Case 2: Node to be deleted has only one child

- Method 1:** Copy the child to the node and delete the child. For example, if you want to delete node 30. First copy its child key value (i.e. 40) to this node. Assign the null value to the pointer which points to its child and delete its child.



- Method 2:** This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.

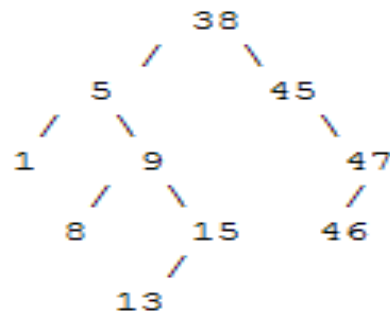


### Case 3: Node to be deleted has two children

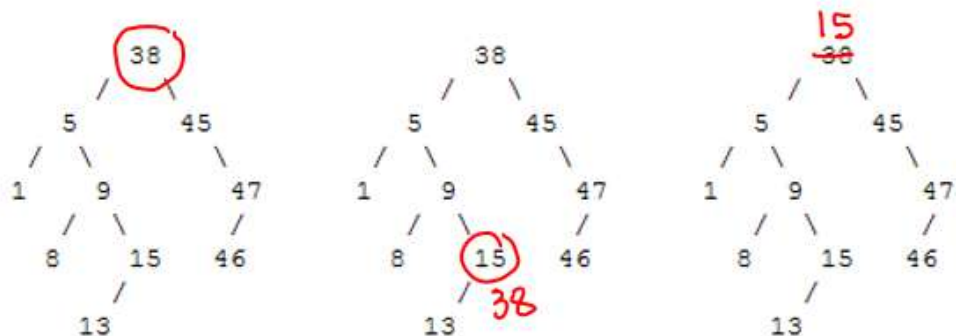
This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to

do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.



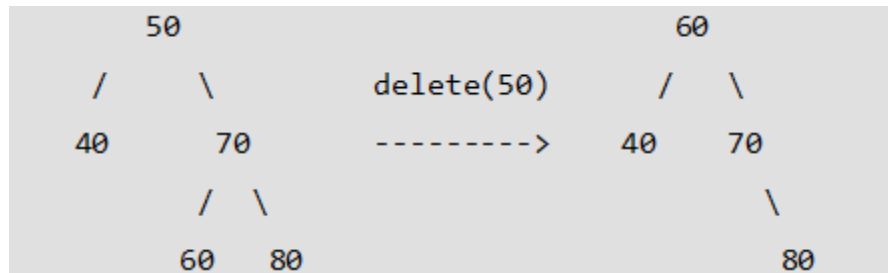
Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.



**Example 2:** Let us consider another tree and we want to delete node 50 which has two children.

- Find in-order successor of the node. Copy contents of the in-order successor to the node and delete the in-order successor. Note that in-order predecessor can also be used.





The important thing to note is, in-order successor is needed only when right child is not empty. In this particular case, in-order successor can be obtained by finding the minimum value in right child of the node.

## 6.7. IMPLEMENTATION OF BINARY SEARCH TREE IN C++

```

#include<iostream>
using namespace std;

//declaration for new bst node
struct bstnode
{
int data;
struct bstnode *left, *right;
};

// create a new BST node
struct bstnode *newNode(int key)
{
struct bstnode *temp = new struct bstnode();
temp->data = key;
temp->left = temp->right = NULL;
return temp;
}

// perform inorder traversal of BST
void inorder(struct bstnode *root)
{
if (root != NULL)
{
inorder(root->left);
cout<<root->data<<" ";
inorder(root->right);
}
}

/* insert a new node in BST with given key */
struct bstnode* insert(struct bstnode* node, int key)
{
//tree is empty;return a new node
if (node == NULL) return newNode(key);

//if tree is not empty find the proper place to insert new node
if (key < node->data)
node->left = insert(node->left, key);

```

```

else
node->right = insert(node->right, key);

    //return the node pointer
return node;
}
//returns the node with minimum value
struct bstnode * minValueNode(struct bstnode* node)
{
struct bstnode* current = node;

    //search the leftmost tree
while (current && current->left != NULL)
current = current->left;

return current;
}
//function to delete the node with given key and rearrange the root
struct bstnode* deleteNode(struct bstnode* root, int key)
{
    // empty tree
if (root == NULL) return root;

    // search the tree and if key < root, go for leftmost tree
if (key < root->data)
root->left = deleteNode(root->left, key);

    // if key > root, go for rightmost tree
else if (key > root->data)
root->right = deleteNode(root->right, key);

    // key is same as root
else
    {
        // node with only one child or no child
if (root->left == NULL)
        {
struct bstnode *temp = root->right;
free(root);
return temp;
        }
else if (root->right == NULL)
        {
struct bstnode *temp = root->left;
free(root);
return temp;
        }

        // node with both children; get successor and then delete the node
struct bstnode* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
root->data = temp->data;

        // Delete the inorder successor
root->right = deleteNode(root->right, temp->data);
    }
}

```

```

        return root;
    }
    // main program
    int main()
    {
        /* Let us create following BST
            40
           / \
          30  60
             \
             65
              \
              70*/

        struct bstnode *root = NULL;
        root = insert(root, 40);
        root = insert(root, 30);
        root = insert(root, 60);
        root = insert(root, 65);
        root = insert(root, 70);

        cout<<"Binary Search Tree created (Inorder traversal):"<<endl;
        inorder(root);

        cout<<"\nDelete node 40\n";
        root = deleteNode(root, 40);
        cout<<"Inorder traversal for the modified Binary Search Tree:"<<endl;
        inorder(root);

        return 0;
    }

```

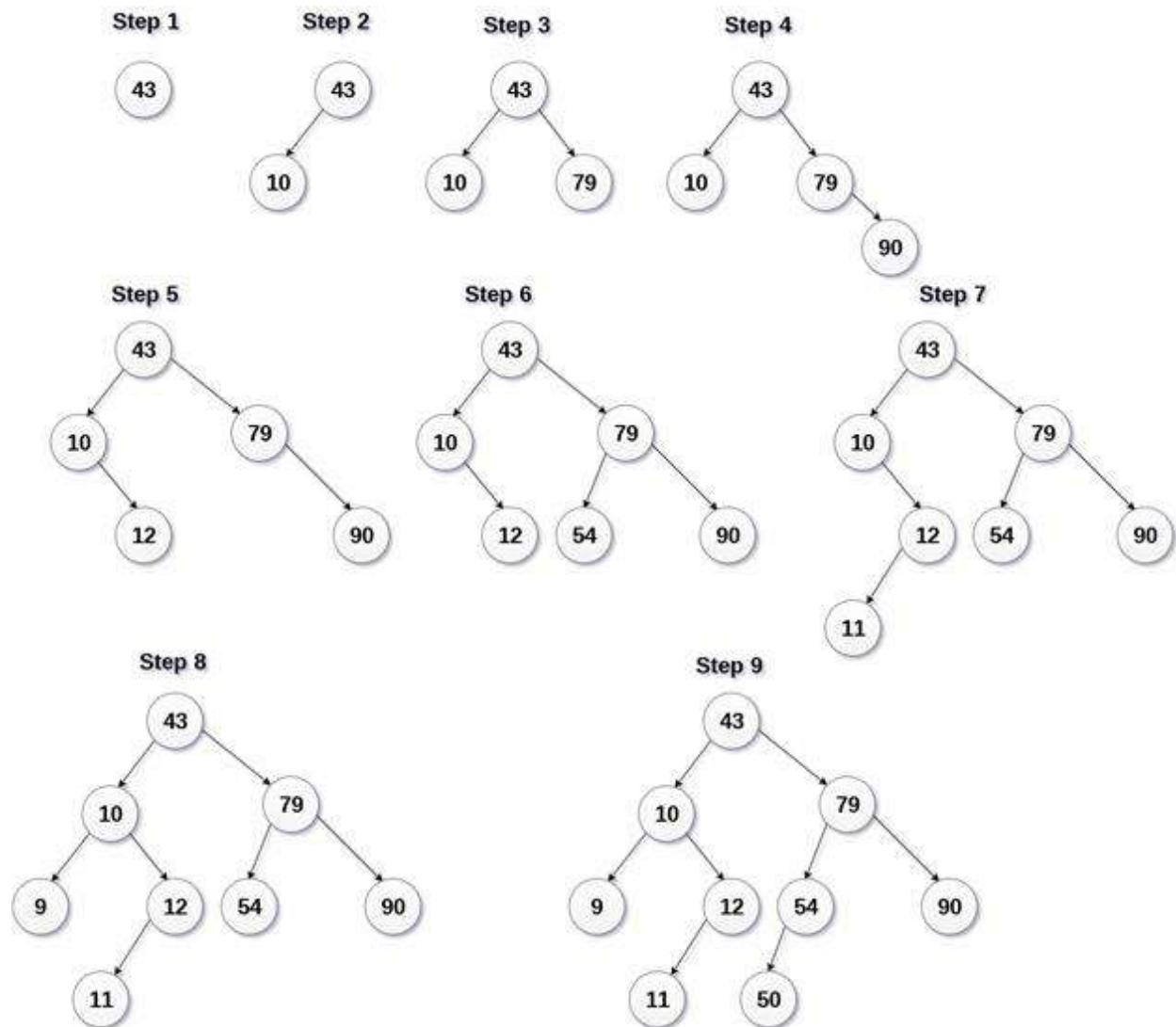
## 6.8. CONSTRUCTING A BINARY SEARCH TREE

**Example 1:** Create the binary search tree using the following data elements. 43, 10, 79, 90, 12, 54, 11, 9, 50

Solution:

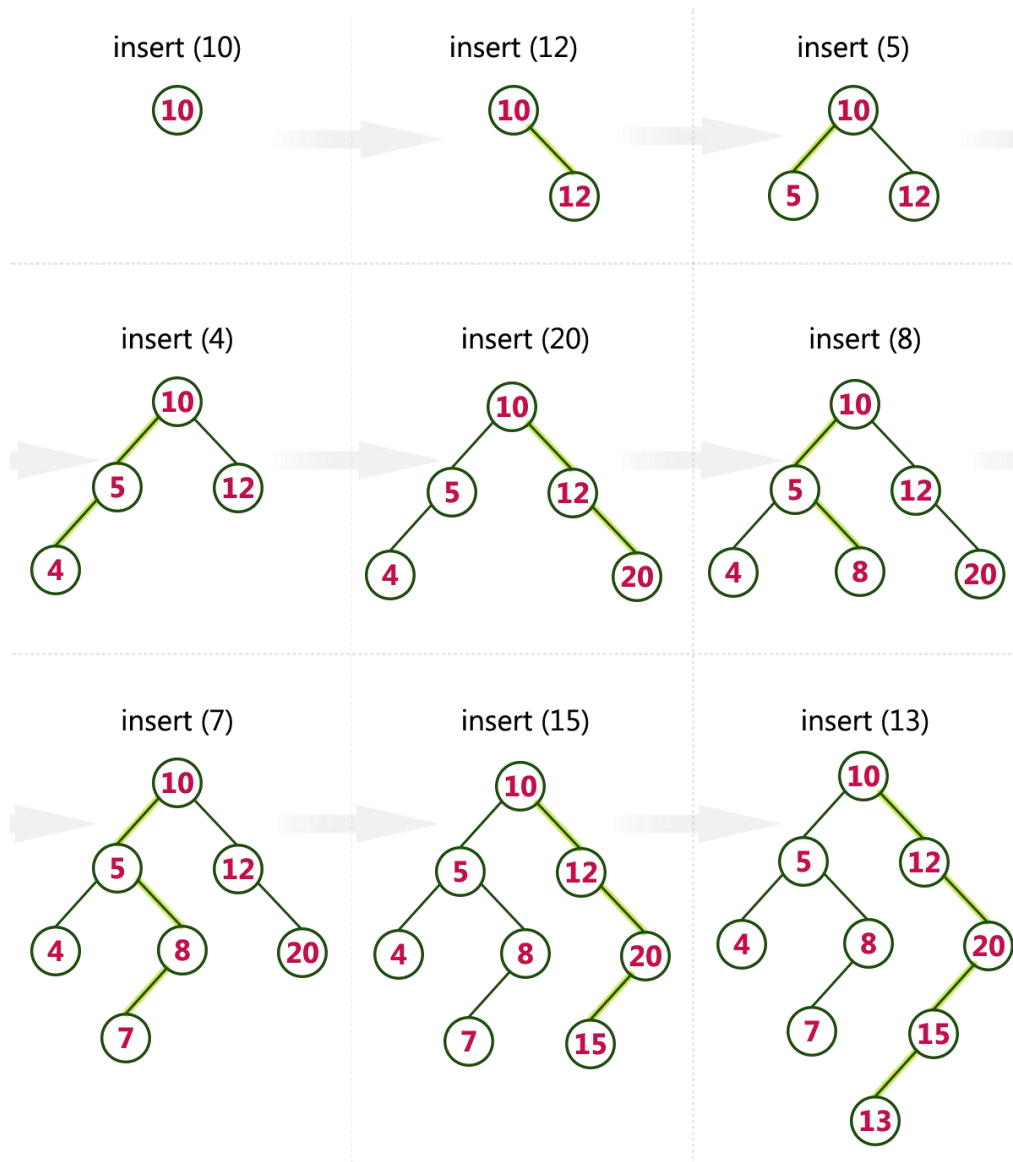
1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



**Example 2:** Construct a Binary Search Tree by inserting the following sequence of numbers- 10, 12, 5, 4, 20, 8, 7, 15 and 13.

Solution: Above elements are inserted into a Binary Search Tree as follows:



## Construct Binary Tree from In-order and Post-order Traversal

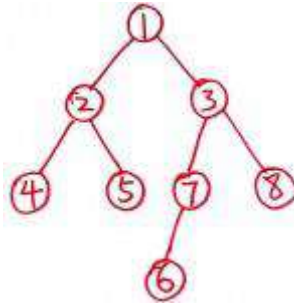
Construct a binary tree using the given in-order and post order sequence:

- in-order: 4 2 5 (1) 6 7 3 8
- post-order: 4 5 2 6 7 8 3 (1)

From the post-order array, we know that last element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in post-order array. Recursively, we can build up the tree.

For this example, the constructed tree is:



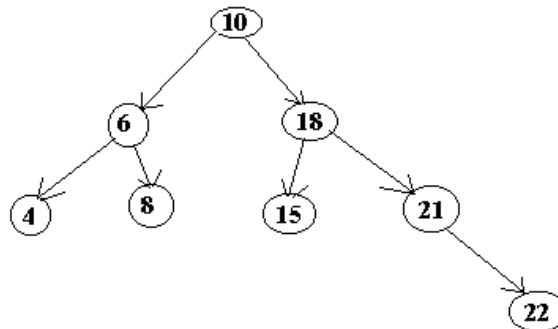
**Student Practice:** Construct a binary tree using the given in-order and pre-order sequence:

- In-order sequence: D B E A F C
- Pre-order sequence: A B D E C F

(Hint: preorder sequence contains the root at first. That is A is the root in this pre-order sequence)

## 6.9. BALANCED TREES

A binary tree is a binary search tree (BST) if and only if an in-order traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.



If the BST is built in a “balanced” fashion, then BST provides **log** time access to each element. Consider an arbitrary BST of the height  $k$ . The total possible number of nodes is given by

$$2^{k+1} - 1$$

In order to find a particular node we need to perform one comparison on each level, or maximum of  $(k+1)$  total. Now, assume that we know the number of nodes and we want to figure out the number of comparisons. We have to solve the following equation with respect to  $k$ :

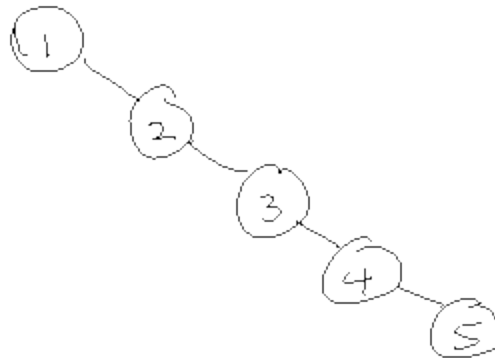
Assume that we have a “balanced” tree with  $n$  nodes. If the maximum number of comparisons to find an entry is  $(k+1)$ , where  $k$  is the height, we have

$$2^{k+1} - 1 = n$$

We obtain

$$k = \log_2(n+1) - 1 = O(\log_2 n)$$

This means, that a “balanced” BST with  $n$  nodes has a maximum order of  $\log(n)$  levels, and thus it takes at most  $\log(n)$  comparisons to find a particular node. This is the most important fact you need to know about BSTs. But building a BST as a balanced tree is not a trivial task. If the data is randomly distributed, then we can expect that a tree can be “almost” balanced, or there is a good probability that it would be. However, if the data already has a pattern, then just naïve insertion into a BST will result in unbalanced trees. For example, if we just insert the data 1, 2, 3, 4, 5 into a BST in the order they come, we will end up with a tree that looks like this:



Binary search trees work well for many applications. But they can be limiting because of their bad worst-case performance  $\text{height} = O(n)$  where  $n$  is number of nodes. Imagine a binary search tree created from a list that is already sorted.

Clearly, the tree will grow to the right or to the left. A binary search tree with this worst-case structure is no more efficient than a regular linked list. A great care needs to be taken in order to keep the tree as balanced as possible. There are many techniques for balancing a tree including AVL trees, and Splay Trees. We will later discuss AVL trees in next section.

One strength of binary search trees (over other data structures) is its ability to maintain items in a way that search can be performed in  $O(\log n)$  Time, where  $n$  is the number of nodes. However this requires that we maintain the tree in a “balanced” form where entries are distributed evenly among left and right sub trees of each node. Since most data sets in real life situations may come almost sorted or in some kind of order (reverse sorted, already sorted). Therefore a naïve insertion into the tree may cause the tree to grow unbalanced. There are two ways to deal with this situation. One option is to randomize the data before inserting to the tree. The cost of doing this operation is  $O(n)$ . The drawback in this approach is that we need to know all the data in advance in order to randomize the set. Later when new data is inserted, we may not be able to apply the randomized algorithm again (unless we randomize the whole set). Therefore we need to think of ways to maintain a tree balanced (at low cost), when frequent insertions are performed.

One solution to this problem is to adopt another search tree structure instead of using a BST at all. An example of such an alternative tree structure is the **2-3 Tree** or the **B-Tree**. But another alternative would be to modify the BST access functions in some way to guarantee that the tree performs well. This is an appealing concept, and the concept works well for heaps, whose access functions maintain the heap in the shape of a complete binary tree. Unfortunately, the heap keeps its balanced shape at the cost of weaker restrictions on the relative values of a node and its children, making it a bad search structure. And

requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update, as we see in this example.

A balanced binary search tree is a tree that keeps its height small (guaranteed to be logarithmic) for a sequence of insertions and deletions. A Balanced Binary Tree is a Binary tree in which height of the left and the right sub-trees of every node may differ by at most 1. AVL Tree and Red-Black Tree are well-known data structure to generate/maintain Balanced Binary Search Tree. Search, insert and delete operations cost  $O(\log n)$  time in such balanced tree.

## 6.10. AVL BALANCED TREES

### Necessity of balanced tree

The Binary Search Tree has a serious deficiency for practical use as a search structure. That is the fact that it can easily become unbalanced, so that some nodes are deep in the tree. In fact, it is possible for a BST with  $n$  nodes to have a depth of  $n$ , making it no faster to search in the worst case than a linked list. If we could keep the tree balanced in some way, then search cost would only be  $\Theta(\log n)$ , a huge improvement.

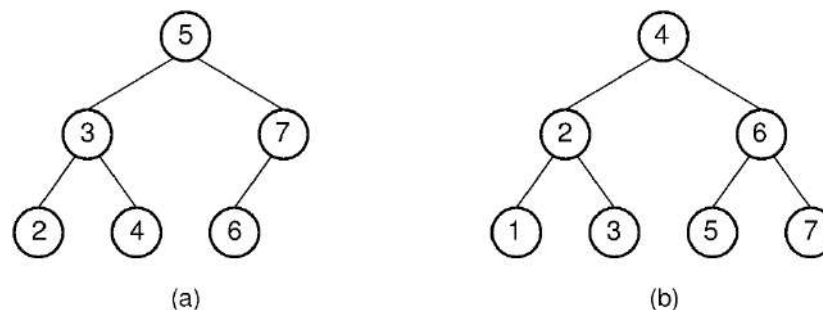


Figure: An attempt to re-balance a BST after insertion can be expensive. (a) A BST with six nodes in the shape of a complete binary tree. (b) A node with value 1 is inserted into the BST of (a). To maintain both the complete binary tree shape and the BST property, a major reorganization of the tree is required.

If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure. The AVL tree works in this way, using insertion and deletion routines altered from those of the BST to ensure that, for every node, the depths of the left and right subtrees differ by at most one.

A different approach to improving the performance of the BST is to not require that the tree always be balanced, but rather to expend some effort toward making the BST more balanced every time it is accessed. One example of such a compromise is called the **splay tree**. The **Red-Black Tree** is also a binary tree, but it uses a different balancing mechanism.

### AVL Tree

AVL tree is a height-balanced binary search tree. That means, an AVL tree is a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.



The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1. It is called balance factor. As long as the tree maintains this property, if the tree contains  $n$  nodes, then it has a depth of at most  $O(\log n)$ . As a result, search for any node will cost  $O(\log n)$ , and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost  $O(\log n)$ , even in the worst case.

$$\text{BalanceFactor} = \text{height}(\text{left-sutree}) - \text{height}(\text{right-sutree})$$

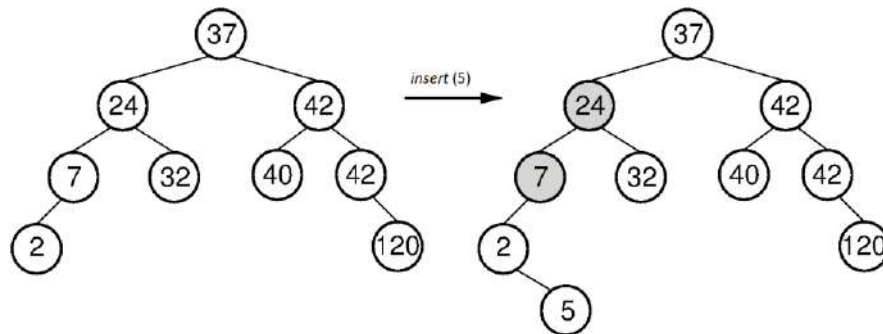
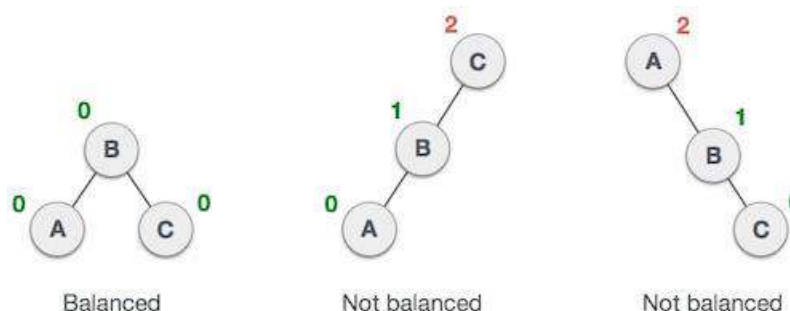


Figure: Example of an insert operation that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

Consider what happens when we insert a node with key value 5, as shown in above Figure. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees.

## 6.11. BALANCING ALGORITHM

In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced. Rotation is the process of moving nodes either to left or to right to make the tree balanced. Here we see that the first tree is balanced and the next two trees are not balanced:



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\text{BalanceFactor} = \text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})$$

If the difference in the height of left and right sub-trees is more than 1, then the tree is rebalanced using some rotation techniques. There are four cases due to which AVL tree may become unbalanced. Therefore there are four different rotations to rebalance the AVL tree.

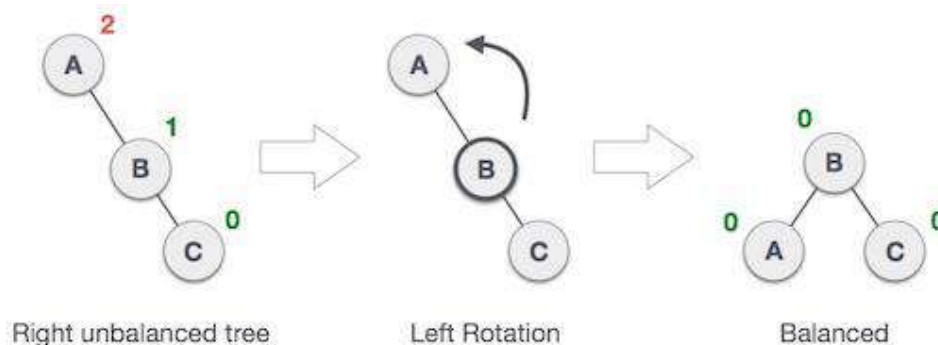
#### AVL Rotations:

1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation (LL Rotation)

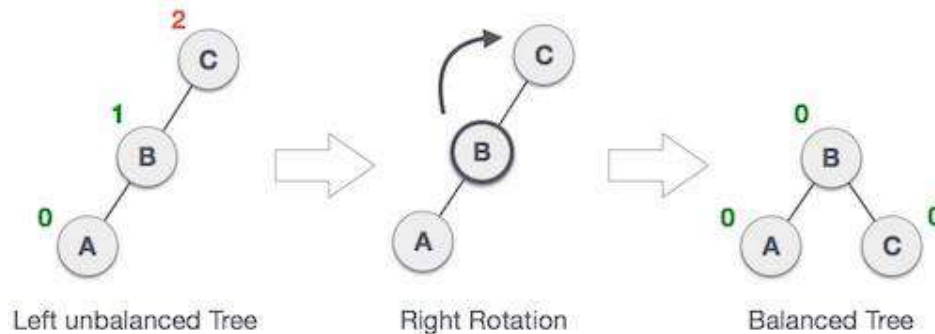
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

### Right Rotation (RR Rotation)

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

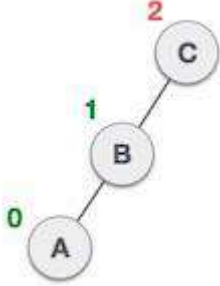
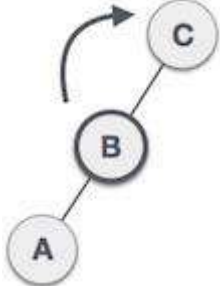
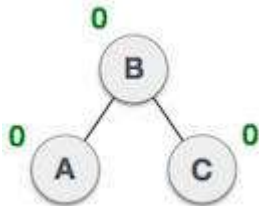


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation (LR Rotation)

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

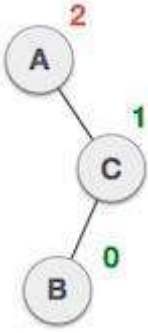
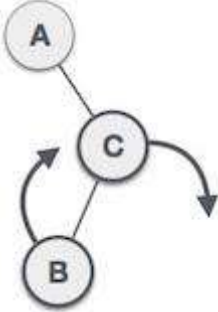
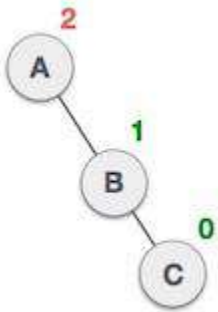
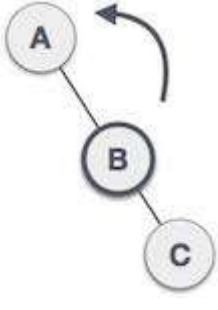
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes <b>C</b> an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of <b>C</b>. This makes <b>A</b>, the left subtree of <b>B</b>.</p>

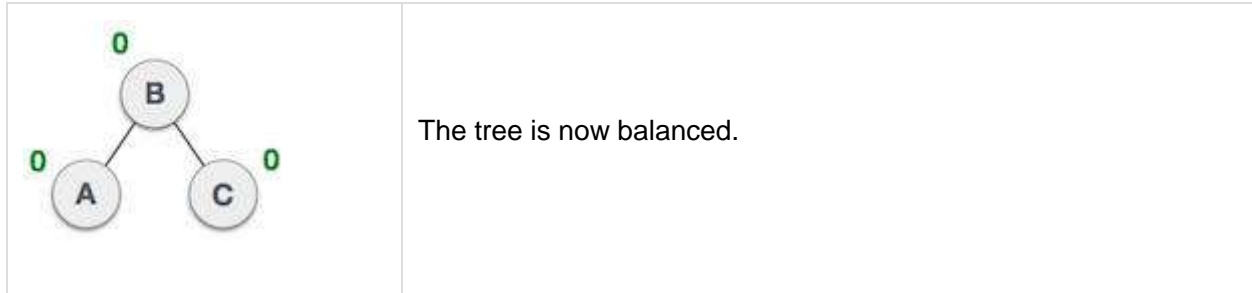
	<p>Node <b>C</b> is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making <b>B</b> the new root node of this subtree. <b>C</b> now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

## Right-Left Rotation (RL Rotation)

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
-------	--------

	<p>A node has been inserted into the left subtree of the right subtree. This makes <b>A</b>, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along <b>C</b> node, making <b>C</b> the right subtree of its own left subtree <b>B</b>. Now, <b>B</b> becomes the right subtree of <b>A</b>.</p>
	<p>Node <b>A</b> is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making <b>B</b> the new root node of the subtree. <b>A</b> becomes the left subtree of its right subtree <b>B</b>.</p>

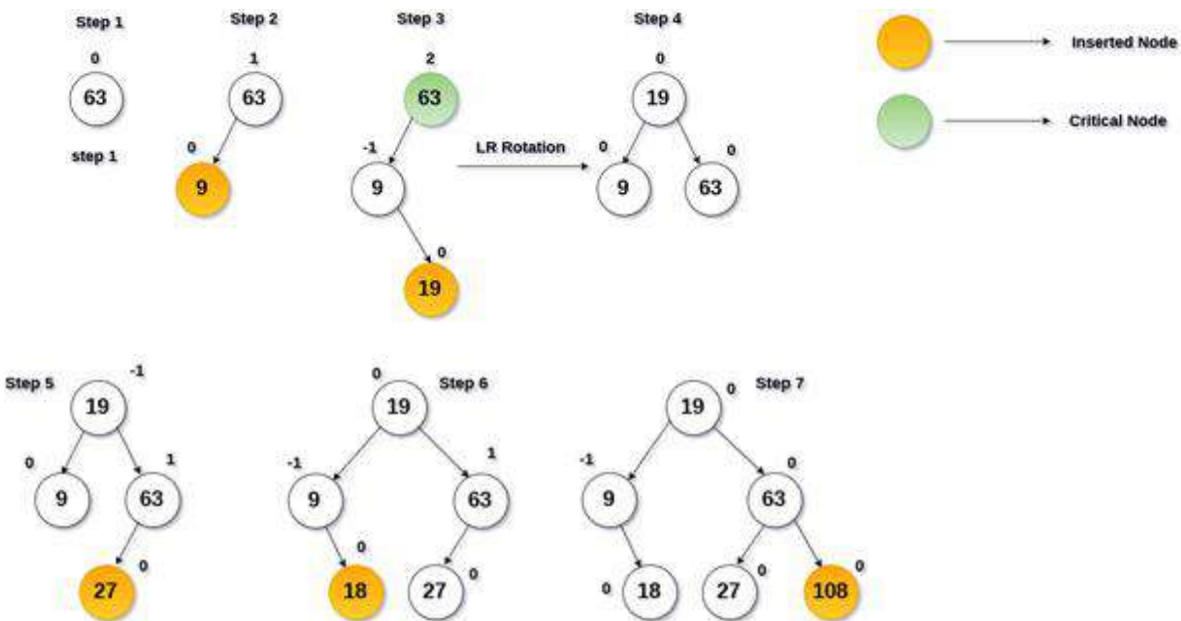


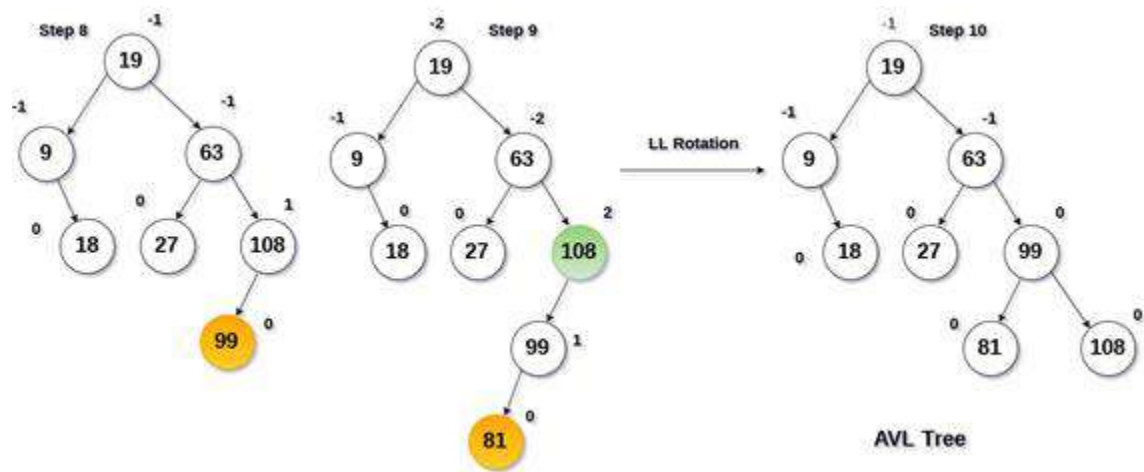
## 6.12. CONSTRUCTION OF AVL TREE

Example 1: Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81

**Solution:** The process of constructing an AVL tree from the given set of elements is shown in the following figure.

At each step when we insert a new node, we must calculate the balance factor to check whether the balance factor differ by at most 1. If it differ by more than 1, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node. All the elements are inserted in order to maintain the order of binary search tree.





Example 2: Construct an AVL Tree from 1, 2, 3, 4, 5, 6, 7, 8.

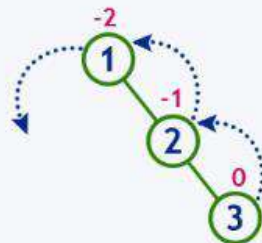
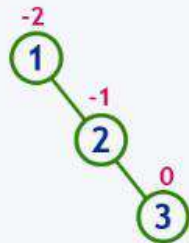
insert 1



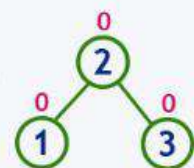
insert 2



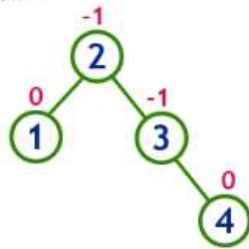
insert 3



After LL Rotation

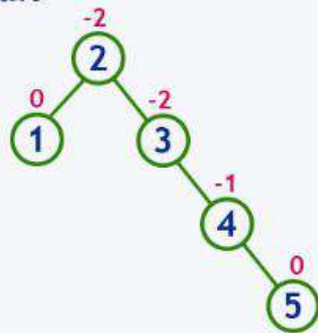


insert 4

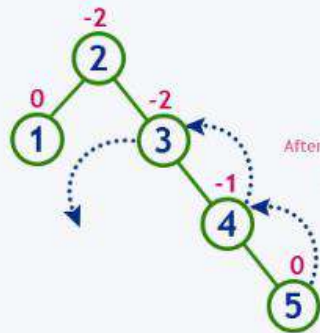


Tree is balanced

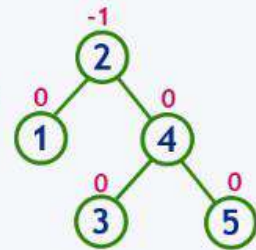
insert 5



Tree is imbalanced

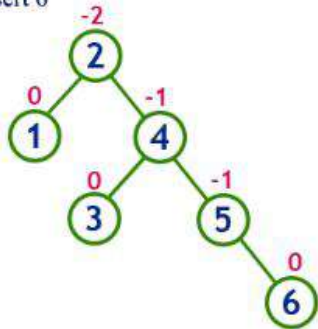


After LL Rotation at 3

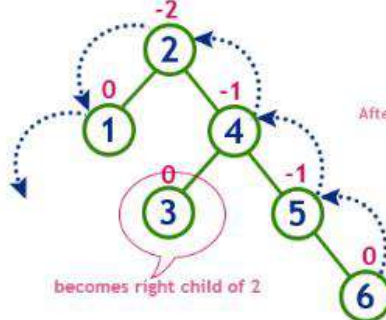


Tree is balanced

insert 6

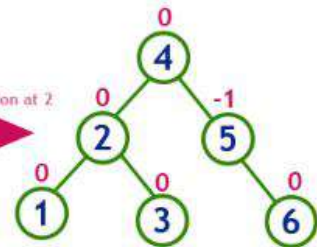


Tree is imbalanced



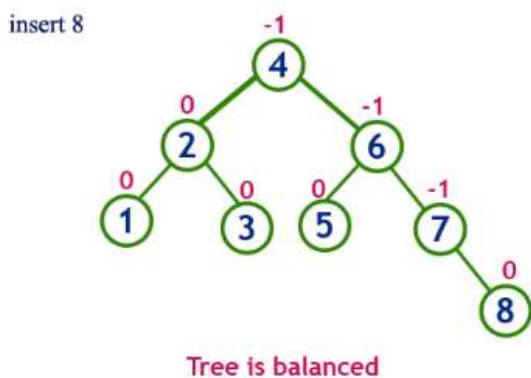
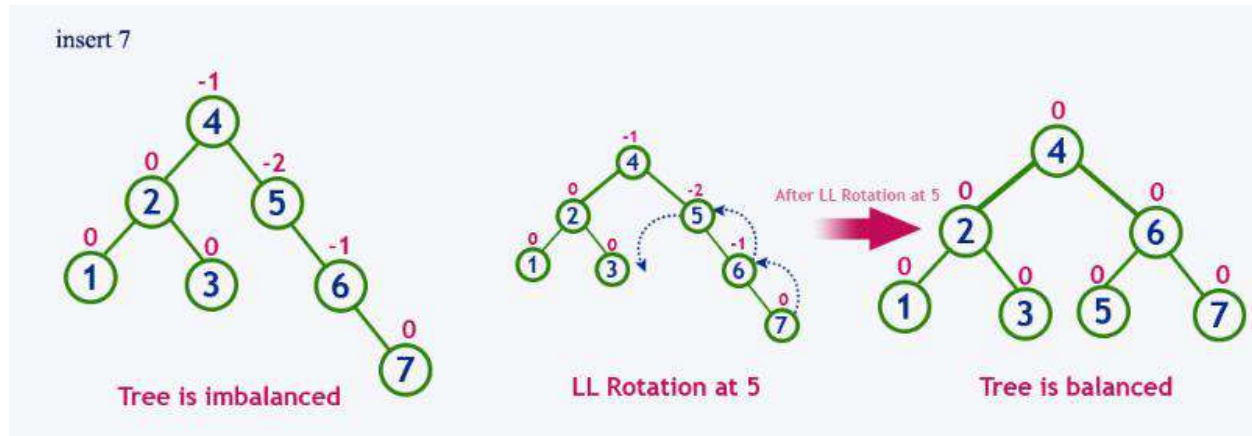
becomes right child of 2

After LL Rotation at 2



Tree is balanced





(**Student Practice:** Construct AVL tree from the sequence: 50, 25, 10, 5, 7, 3, 30, 20, 8, 15)

## 6.13. THE HUFFMAN ALGORITHM

We often represent a set of items in a computer program by assigning a unique code to each item. For example, the standard **ASCII coding** scheme assigns a unique eight-bit value to each character. It takes a certain minimum number of bits to provide enough unique codes so that we have a different one for each character. For example, it takes  $\lceil \log 128 \rceil$  or seven bits to provide the 128 unique codes needed to represent the 128 symbols of the ASCII character set.

The requirement for  $\lceil \log n \rceil$  bits to represent  $n$  unique code values assumes that all codes will be the same length, as are ASCII codes. These are called **fixed-length codes**. If all characters were used equally often, then a fixed-length coding scheme is the most space efficient method. However, you are probably aware that not all characters are used equally often in many applications. For example, the various letters in an English language document have greatly different frequencies of use.

Below Table shows the relative frequencies of the letters of the alphabet. From this table we can see that the letter 'E' appears about 60 times more often than the letter 'Z'. In normal ASCII, the words "DEED" and "MUCK" require the same amount of space (four bytes). It would seem that words such as "DEED", which are composed of relatively common letters, should be storable in less space than words such as "MUCK", which are composed of relatively uncommon letters.

Table: Relative frequencies for the 26 letters of the alphabet as they appear in a selected set of English documents. "Frequency" represents the expected frequency of occurrence per 1000 letters, ignoring case.

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

If some characters are used more frequently than others, is it possible to take advantage of this fact and somehow assign them shorter codes? The price could be that other characters require longer codes, but this might be worthwhile if such characters appear rarely enough. This concept is at the heart of file compression techniques in common use today. The next section presents one such approach to assigning **variable-length codes**, called **Huffman coding**. While it is not commonly used in its simplest form for file compression (there are better methods), Huffman coding gives the flavor of such coding schemes. One motivation for studying Huffman coding is because it provides our first opportunity to see a type of tree structure referred to as a **search tree**.

To keep things simple, these examples for building Huffman trees uses a **sorted list** to keep the partial Huffman trees ordered by frequency. But a real implementation would use a **heap** to implement a **priority queue** keyed by the frequencies.

## Fixed-Length Character Encodings

A character encoding maps each character to a number.

- Computers usually use fixed-length character encodings.
- ASCII code- 8 bits per character
- Unicode- 16 bits per character

Fixed-length encodings are simple, because all encodings have the same length and a given character always has the same encoding.

### *A Problem with Fixed-Length Encodings*

- They tend to waste space
- Example: an English newspaper article with only:
  - upper and lower-case letters (52 characters)
  - spaces and newlines (2 characters)
  - common punctuation (approx. 10 characters)

- total of 64 unique characters → only need \_\_\_\_ bits
- We could gain even more space if we:
  - gave the most common letters shorter encodings (3 or 4 bits)
  - gave less frequent letters longer encodings (> 6 bits)

## Variable-Length Character Encodings

- Variable-length encodings:
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters
  - Example: if we had only four characters

e	01
o	100
s	111
t	00

"test" would be encoded as  
00 01 111 00 → 000111100

- **Challenge:** when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - **Example:** for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
  - **One solution is Huffman encoding**

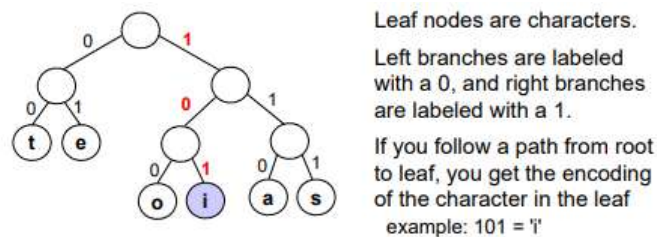
## Huffman Encoding

Huffman coding assigns codes to characters such that the length of the code depends on the relative frequency or **weight** of the corresponding character. Thus, it is a variable-length code. If the estimated frequencies for letters match the actual frequency found in an encoded message, then the length of that message will typically be less than if a fixed-length code had been used. The Huffman code for each letter is derived from a full binary tree called the **Huffman coding tree**, or simply the **Huffman tree**. Each leaf of the Huffman tree corresponds to a letter, and we define the weight of the leaf node to be the weight (frequency) of its associated letter. The goal is to build a tree with the **minimum external path weight**. Define the **weighted path length** of a leaf to be its weight times its depth. The binary tree with minimum external path weight is the one with the minimum sum of weighted path lengths for the given set of leaves. A letter with high weight should have low depth, so that it will count the least against the total path length. As a result, another letter might be pushed deeper in the tree if it has less weight.

The process of building the Huffman tree for  $n$  letters is quite simple. First, create a collection of  $n$  initial Huffman trees, each of which is a single leaf node containing one of the letters. Put the  $n$  partial trees onto a priority queue organized by weight (frequency). Next, remove the first two trees (the ones with lowest weight) from the priority queue. Join these two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees. Put this new tree back into the priority queue. This process is repeated until all of the partial Huffman trees have been combined into one.

## 1) Huffman Encoding:

- Developed by **David Huffman** in 1951, this technique is the basis for all data compression and encoding schemes
- Uses a binary tree
  - To determine the encoding of each character
  - to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII
  - Example of a Huffman tree (for a text with only six chars):



- It is a famous algorithm used for lossless data encoding
- It follows a Greedy approach, since it deals with generating minimum length prefix-free binary codes
- It uses variable-length encoding scheme for assigning binary codes to characters depending on how frequently they occur in the given text. The character that occurs most frequently is assigned the smallest code and the one that occurs least frequently gets the largest code

## 2) The major steps involved in Huffman coding are-

### Step I - Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol

- A Huffman tree, similar to a binary tree data structure, needs to be created having **n** leaf nodes and **n-1** internal nodes
- Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue.
- Initially, all nodes are leaf nodes containing the character itself along with the weight/ frequency of that character
- Internal nodes, on the other hand, contain weight and links to two child nodes

### Step II - Assigning the binary codes to each symbol by traversing Huffman tree

- Generally, bit '0' represents the left child and bit '1' represents the right child

### 3) Huffman Algorithm

#### A. Creating the Huffman Tree

1. Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)
2. Repeat Steps 3 to 5 while heap has more than one node
3. Extract two nodes, say x and y, with minimum frequency from the heap
4. Create a new internal node z with x as its left child and y as its right child.  
Also  $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$
5. Add z to min heap
6. Last node in the heap is the root of Huffman tree

#### B. Traversing the Huffman Tree (Encoding processes)

1. Create an auxiliary array
2. Traverse the tree starting from root node
3. Assign 0 to array while traversing the left child and assign 1 to array while traversing the right child
4. Print the array elements whenever a leaf node is found

### 4) Example

Let's try and create Huffman Tree for the following characters along with their frequencies using the above algorithm-

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

#### A. Build the Huffman Tree

Create a leaf node for each character and build a min heap using all the nodes (The frequency value is used to compare two nodes in min heap)

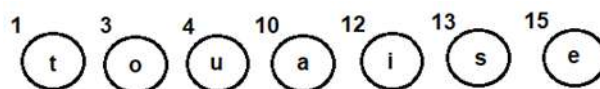


Fig 1: Leaf nodes for each character

(Extract two nodes, say  $x$  and  $y$ , with minimum frequency from the heap. Create a new internal node  $z$  with  $x$  as its left child and  $y$  as its right child. Also  $\text{frequency}(z) = \text{frequency}(x) + \text{frequency}(y)$  and Add  $z$  to min heap)

Extract and Combine node  $t$  and  $o$ . Add the new internal node to priority queue:

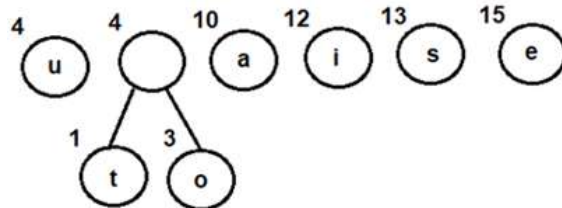


Fig 2: Combining nodes  $o$  and  $t$

Extract and Combine node  $u$  with an internal node having 4 as the frequency. Add the new internal node to priority queue:

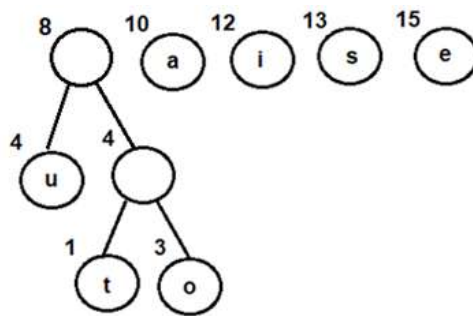


Fig 3: Combining node  $u$  with an internal node having 4 as frequency

Extract and Combine node  $a$  with an internal node having 8 as the frequency. Add the new internal node to priority queue:

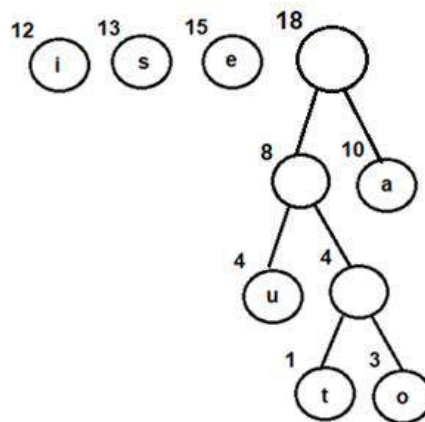


Fig 4: Combining node  $a$  with an internal node having 8 as frequency

Extract and Combine nodes i and s. Add the new internal node to priority queue:

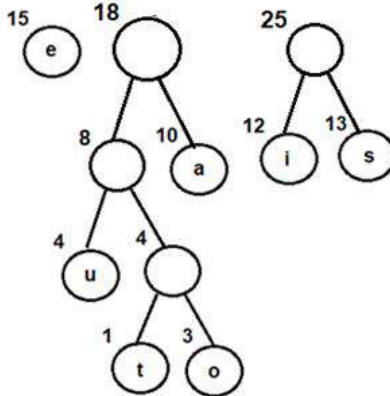


Fig 5: Combining nodes i and s

Extract and Combine node e with an internal node having 18 as the frequency. Add the new internal node to priority queue:

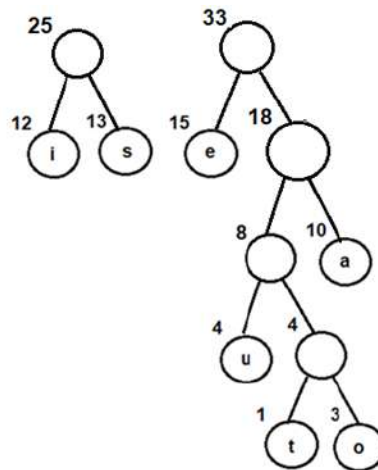


Fig 6: Combining node e with an internal node having 18 as frequency

Finally, Extract and Combine internal nodes having 25 and 33 as the frequency. Add the new internal node to priority queue:

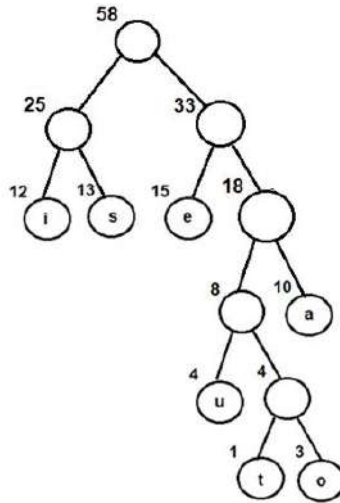


Fig 7: Final Huffman tree obtained by combining internal nodes having 25 and 33 as frequency

### B. Traverse the Huffman Tree (Encoding process)

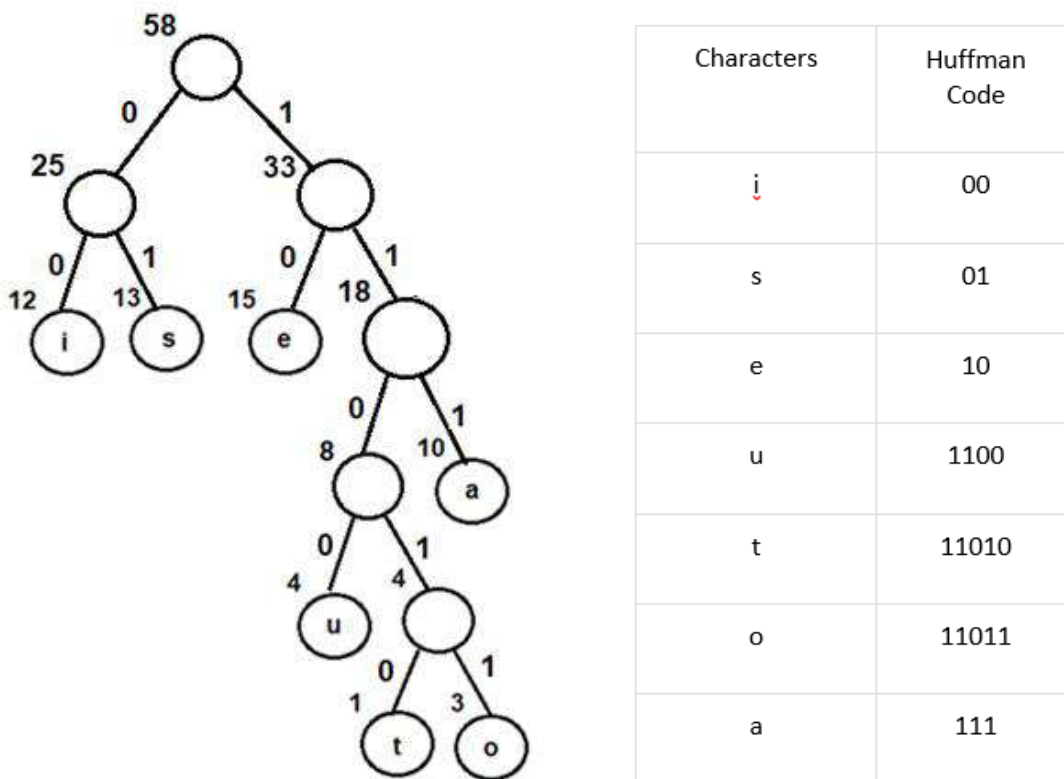


Fig 8: Assigning binary codes to Huffman tree (a) Huffman tree after assigning 0 to the left child and 1 to the right child (b) generated variable length Huffman coding for each character

### C. Encoding and decoding using Huffman tree

Suppose the string “staeiou” needs to be transmitted from computer A (sender) to computer B (receiver) across a network. Using concepts of Huffman encoding at the sender side, the string gets encoded to



**"0111010111100011011110011010"**

Once received at the receiver's side, it will be decoded back by traversing the Huffman tree. For decoding each character, we start traversing the tree from root node. Start with the first bit in the string. A '1' or '0' in the bit stream will determine whether to go left or right in the tree. Print the character, if we reach a leaf node. Suppose the first two 01 leads to character *s*. The first 0 from the root lead to left, the 1 leads to right which is leaf node and it contains character *s*. After this coding of next character start and follow the same process from the root to decode all characters in the text stream. The next is **11010** which leads to character *t*.

**Student Practice:** Build a Huffman tree and Huffman coding for the following characters with given frequencies:

Characters	'o'	'i'	'a'	's'	't'	'e'
Frequency	11	23	25	26	27	40

## 6.14. GAME TREE

A **game tree** is a type of recursive search function that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move. They are very useful for Artificial Intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play. The most commonly-cited example is chess, but they are applicable to many situations.

A **game tree** is a directed graph whose nodes are positions in a game and whose edges are moves. The **complete game tree** for a game is the game tree starting at the initial position and containing all possible moves from each position; the complete tree is the same tree as that obtained from the extensive-form game representation.

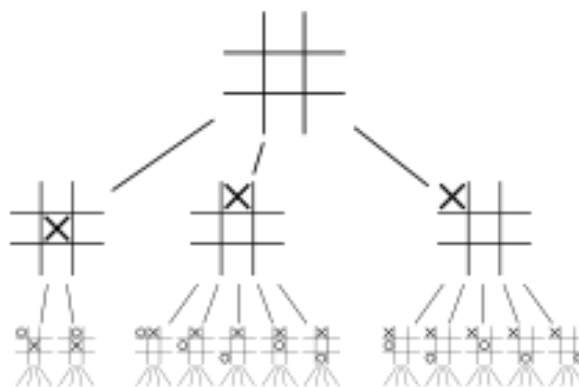


Figure: The first two plies of the game tree for tic-tac-toe.

The diagram shows the first two levels, or *plies*, in the game tree for tic-tac-toe. The rotations and reflections of positions are equivalent, so the first player has three choices of move: in the center, at the edge, or in the corner. The second player has two choices for the reply if the first player played in the center, otherwise five choices. And so on.

The number of leaf nodes in the complete game tree is the number of possible different ways the game can be played. For example, the game tree for tic-tac-toe has 255,168 leaf nodes.

## Limitations

There are a few major limitations to game trees.

**Time:** As mentioned above, game trees are rarely used in real-time scenarios (when the computer isn't given very much time to think). The reason for this will soon become apparent, but to put it brief: the method requires a lot of processing by the computer, and that takes time.

**For the above reason (and others) they work best in turn-based games. They require complete knowledge of how to move:** Games with uncertainty generally do not mix well with game trees. They can be implemented in such games, but it will be very difficult, and the results may prove less than satisfactory. **They are ineffective at accurately ascertaining the best choices in scenarios with many possible choices**

## General Concept

Game trees are generally used in board games to determine the best possible move. For the purpose of this article, Tic-Tac-Toe will be used as an example.

The idea is to start at the current board position, and check all the possible moves the computer can make. Then, from each of those possible moves, to look at what moves the opponent may make. Then to look back at the computers. Ideally, the computer will flip back and forth, making moves for itself and its opponent, until the game's completion. It will do this for every possible outcome (see image below), effectively playing thousands (often more) of games. From the winners and losers of these games, it tries to determine the outcome that gives it the best chance of success.

This can be likened to how people think during a board game - for example "if I make this move, they could make this one, and I could counter with this" etc. Game trees do just that. They examine every possible move and every opponent move, and then try to see if they are winning after as many moves as they can think through.

As one could imagine, there are an extremely large number of possible games. In Tic-Tac-Toe, there are 9 possible first moves (ignoring rotation optimization). There are 8 on the second turn, then 7, 6, etc. In total, there are 255,168 possible games. The goal of a game tree is to look at them all (on the first move) and try to choose a move that will, at worst, make losing impossible, and, at best, win the game.

## 6.15. B TREE.

B tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

B tree is a self-balanced tree as well as a specialized m-way tree that is used for disk access. When the amount of data to be stored is very high, we cannot store the entire data in the main memory. Hence we

store data in the disk. Data access from the disk takes more time when compared to the main memory access.

When the number of keys of the data stored in disks is very high, the data is usually accessed in the form of blocks. The time to access these blocks is directly proportional to the height of the tree.

The B-Tree is a flat tree i.e. the height of the B tree is kept to a minimum. Instead, as many keys are put in each node of the B-tree. By keeping the height of the B-tree to the minimum, the access is faster when compared to other balanced trees like AVL trees.

## The B Tree Rules

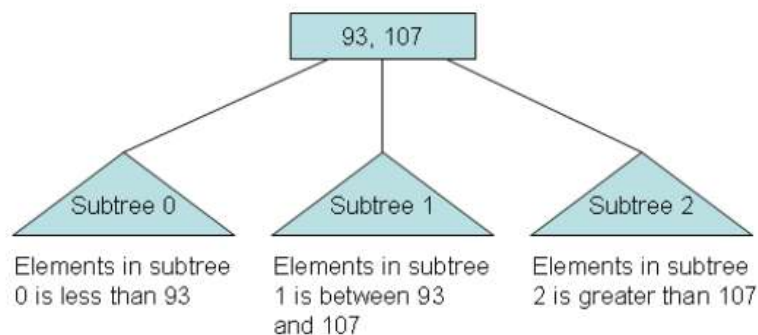
Important properties of a B-tree:

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

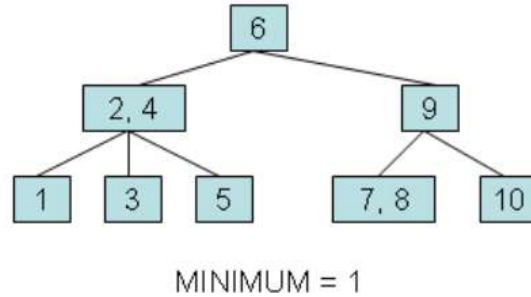
The set formulation of the B-tree rules: Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

- **Rule 1:** The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.
- **Rule 2:** The maximum number of elements in a node is twice the value of MINIMUM.
- **Rule 3:** The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the final used position of the array).
- **Rule 4:** The number of subtrees below a nonleaf node is always one more than the number of elements in the node.
  - Subtree 0, subtree 1, ...
- **Rule 5:** For any nonleaf node:
  1. An element at index  $i$  is greater than all the elements in subtree number  $i$  of the node, and
  2. An element at index  $i$  is less than all the elements in subtree number  $i + 1$  of the node.
- **Rule 6:** Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of a unbalanced tree.

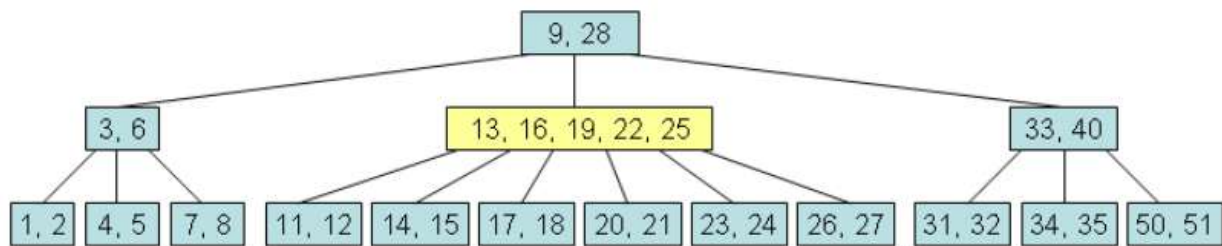
### Example 1



### Example 2



### Example 3



## Basic Operations of B-Tree

Given below are some of the Basic operations of B-Tree.

### 1) Searching

Searching in B tree is similar to that in BST. In the above tree if we want to look for item 3, then we will proceed as follows:

- Compare 3 with root elements. Here,  $3 < 6$  and  $3 < 15$ . So we proceed to the left subtree.
- Compare 3 with 4 in the left subtree. As  $3 < 4$ , we proceed to the left subtree of node 4.
- The next node has two keys, 2 and 3.  $3 > 2$  while  $3 = 3$ . So we have found the key at this node. Return to the found location.

Searching in B tree depends on the height of the tree. It usually takes  $O(\log n)$  amount of time to search for a given item.

### 2) Insertion

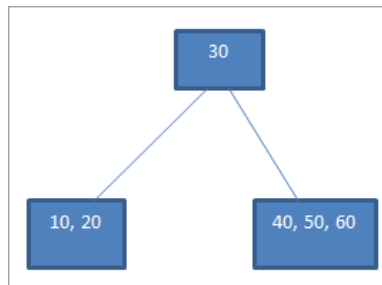
The insertion of a new item in B tree is done at the leaf nodes level.

**Following is the sequence of steps (algorithm) to insert a new item in the B tree.**

- Traverse the B tree to find a location at the leaf nodes to insert the item.
- If the leaf node contains keys  $< m-1$ , then insert a node in increasing order.
- Else if leaf node keys  $= m-1$ , then:

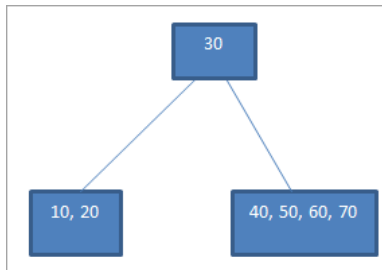
- Insert a new item in an increasing number of items.
- Take the median of the nodes and split the node into two nodes.
- Push median to its parent node.
- If parent node key =  $m-1$ , then repeat the above steps for parent node as well. This is done until we find the appropriate leaf node.
- Finally, insert the element.

We have demonstrated insertion in B tree as follows. Let us insert item 70 in the tree shown below. The given tree is with minimum degree ' $m$ ' = 3. Hence, each node can accommodate,  $2*m - 1 = 5$  keys inside it.



Now we insert item 70. As we can have 5 keys in a node, we compare element 70 with the root element 30. Since  $70 > 30$ , we will insert item 70 in the right subtree.

In the right subtree, we have a node with keys 40, 50, 60. As each node can have 5 keys in it, we will insert the item 70 in this node itself. After insertion, the B-Tree looks as follows.



### 3) Deletion

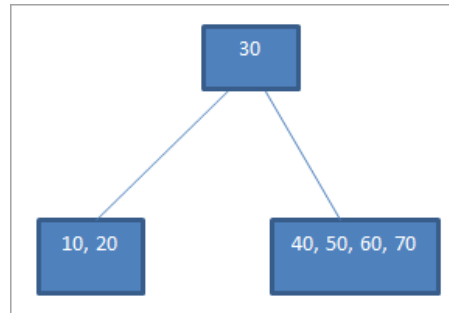
Just like insertion, the deletion of the key is also carried out at leaf nodes level. But unlike insertion, deletion is more complicated. The key to be deleted can be either a leaf node or an internal node.

**To delete a key, we follow the below sequence of steps (algorithm).**

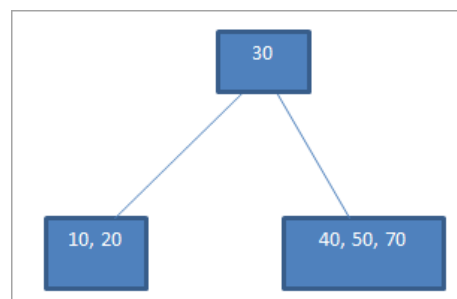
1. Locate the leaf node.
2. In case the number of keys in a node  $> m/2$ , then delete the given key from the node.
3. In case the leaf node is not having  $m/2$  keys, then we need to complete the keys by taking keys from the right or left subtrees to maintain the B tree. We follow these steps:
  - If the left subtree contains  $m/2$  elements then we push its largest element to the parent node and then move the intervening element to the place where the key was deleted.

- If the right subtree contains  $m/2$  elements then we push its smallest element to the parent node and then move the intervening element to the place where the key was deleted.
4. If no node has  $m/2$  nodes then the above steps cannot be performed, thus we create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
  5. If a parent is without  $m/2$  nodes, then we repeat the above steps for the parent node in question. These steps are repeated until we get a balanced B tree.

Given below is an illustration to delete a node from B tree. Consider the following B tree once again.



Let's assume that we want to delete the key 60 from the B tree. If we check the B tree, we can find that the key to be deleted is present in the leaf node. We delete the key 60 from this leaf node. Now the tree will look as shown below:



We can notice that after deleting the key 60, the B tree leaf node satisfies the required properties and we need not do any more modifications to the B tree.

However, if we had to delete key 20, then only key 10 would have remained in the left leaf node. In this case, we would have to shift root 30 to the leaf node and move key 40 to the root.

Thus while deleting a key from B tree, care should be taken and the properties of the B tree should not be violated.

## Applications of B Trees

1. B trees are used to index the data especially in large databases as access to data stored in large databases on disks is very time-consuming.
2. Searching of data in larger unsorted data sets takes a lot of time but this can be improved significantly with indexing using B tree.

## 6.16. B+ TREE

B+ tree is an extension of the B tree. The difference in B+ tree and B tree is that in B tree the keys and records can be stored as internal as well as leaf nodes whereas in B+ trees, the records are stored as leaf nodes and the keys are stored only in internal nodes.

The records are linked to each other in a linked list fashion. This arrangement makes the searches of B+ trees faster and efficient. Internal nodes of the B+ tree are called index nodes.

The B+ trees have two orders i.e. one for internal nodes and other for leaf or external nodes.

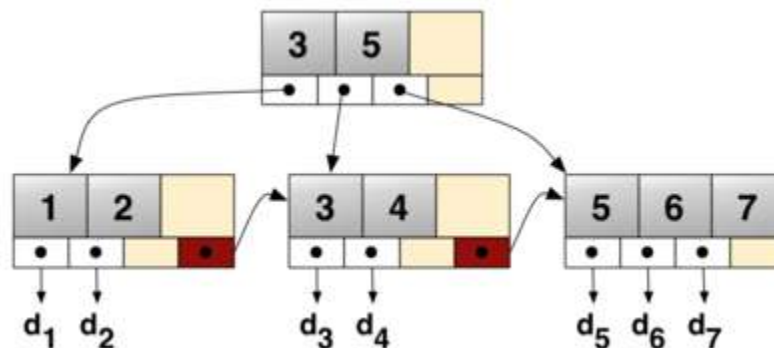


Figure: A simple B+ tree example linking the keys 1–7 to data values d1-d7. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is  $b = 4$ .

A **B+ tree** is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves.<sup>[1]</sup> The root may be either a leaf or a node with two or more children.<sup>[2]</sup>

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fan out (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

As B+ tree is an extension of B-tree, the basic operations that we discussed under the topic B-tree still holds.

While inserting as well as deleting, we should maintain the basic properties of B+ Trees intact. However, deletion operation in the B+ tree is comparatively easier as the data is stored only in the leaf nodes and it will be deleted from the leaf nodes always.

### Advantages of B+ Trees

1. We can fetch records in an equal number of disk accesses.
2. Compared to the B tree, the height of the B+ tree is less and remains balanced.
3. We use keys for indexing.

4. Data in the B+ tree can be accessed sequentially or directly as the leaf nodes are arranged in a linked list.
5. Search is faster as data is stored in leaf nodes only and as a linked list.

## Difference between B Tree and B+ Tree

B Tree	B+ Tree
Data is stored in leaf nodes as well as internal nodes.	Data is stored only in leaf nodes.
Searching is a bit slower as data is stored in internal as well as leaf nodes.	Searching is faster as the data is stored only in the leaf nodes.
No redundant search keys are present.	Redundant search keys may be present.
Deletion operation is complex.	Deletion operation is easy as data can be directly deleted from the leaf nodes.
Leaf nodes cannot be linked together.	Leaf nodes are linked together to form a linked list.