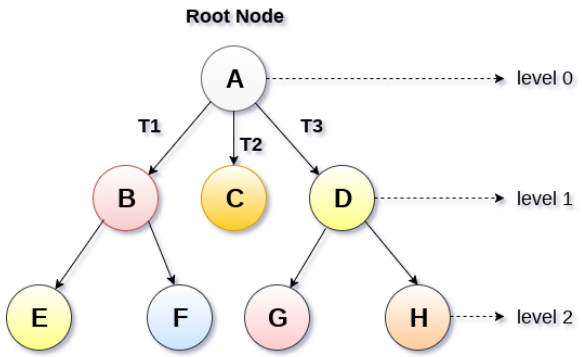
Tree

- A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, A node can have any number of children nodes but it can have only a single parent.
- The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.



Tree

Basic terminology

- **Root Node** :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
- **Sub Tree** :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.
- **Leaf Node** :- The node of tree, which doesn't have any child node, is called leaf node. Leaf node is the bottom most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Path** :- The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is A→ B → E.
- **Ancestor node** :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F have the ancestors, B and A.
- **Degree** :- Degree of a node is equal to number of children, a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.
- **Level Number** :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

Static representation of tree

```
1.  #define MAXNODE 500
2.  struct treenode {
3.      int root;
4.      int father;
5.      int son;
6.      int next;
7.  }
```

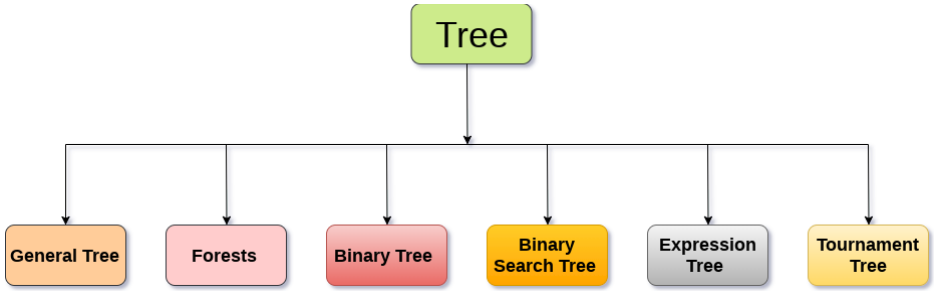
Dynamic representation of tree

```
1.  struct treenode
2.  {
3.      int root;
4.      struct treenode *father;
5.      struct treenode *son
6.      struct treenode *next;
```

```
7.      }
```

Types of Tree

The tree data structure can be classified into six different categories.

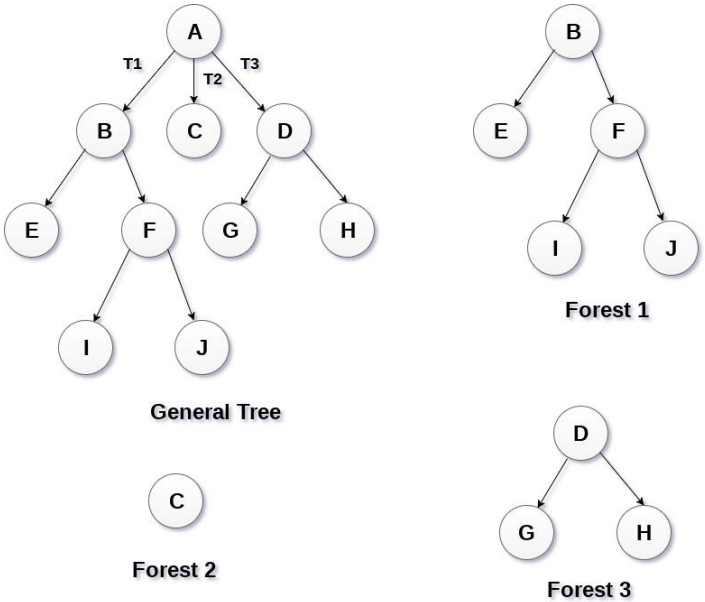


General Tree

General Tree stores the elements in a hierarchical order in which the top level element is always present at level 0 as the root element. All the nodes except the root node are present at number of levels. The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them. A node may contain any number of sub-trees. The tree in which each node contain 3 sub-tree, is called ternary tree.

Forests

Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.



Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0 children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

Binary Search Tree

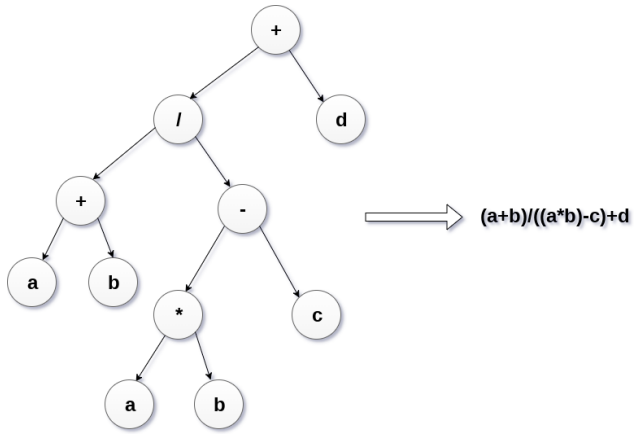
Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions. Expression tree is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands. Expression trees are widely used to solve algebraic expressions like (a+b)*(a-b). Consider the following example.

Q. Construct an expression tree by using the following algebraic expression.

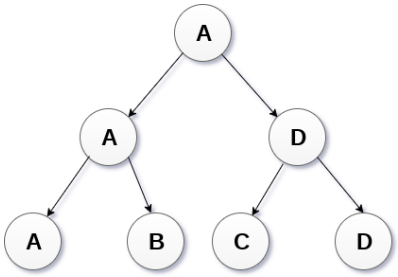
(a + b) / (a*b - c) + d



Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players. Tournament tree can also be called as selection tree or winner tree. External nodes represent the players among which a match is being played while the internal nodes represent the winner of the match played. At the top most level, the winner of the tournament is present as the root node of the tree.

For example, tree .of a chess tournament being played among 4 players is shown as follows. However, the winner in the left sub-tree will play against the winner of right sub-tree.

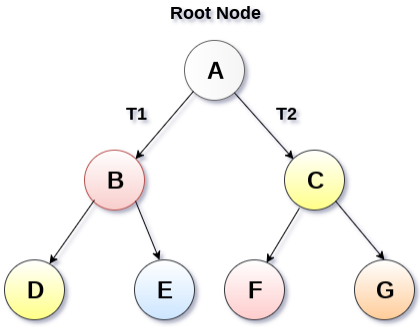


Binary Tree

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

A binary Tree is shown in the following image.

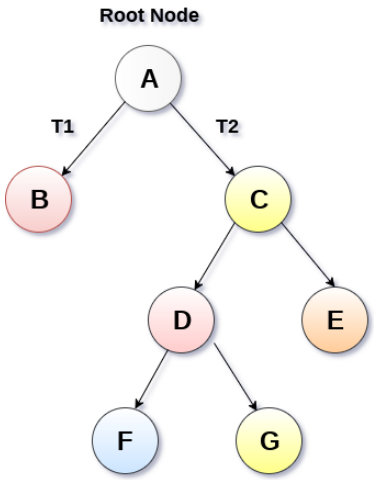


Binary Tree

Types of Binary Tree

1. Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

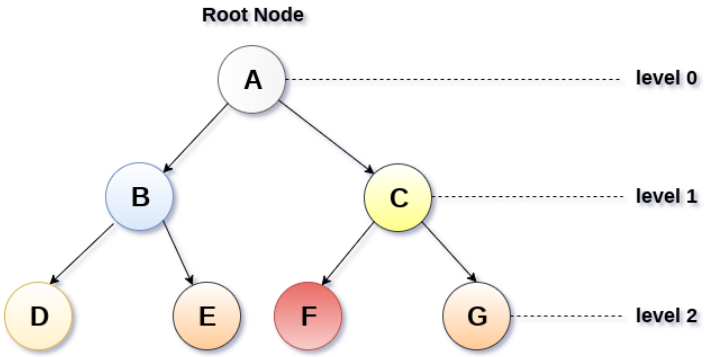


Strictly Binary Tree

A strictly binary tree is shown in the following figure.

2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.



Complete Binary Tree

Binary Tree Traversal

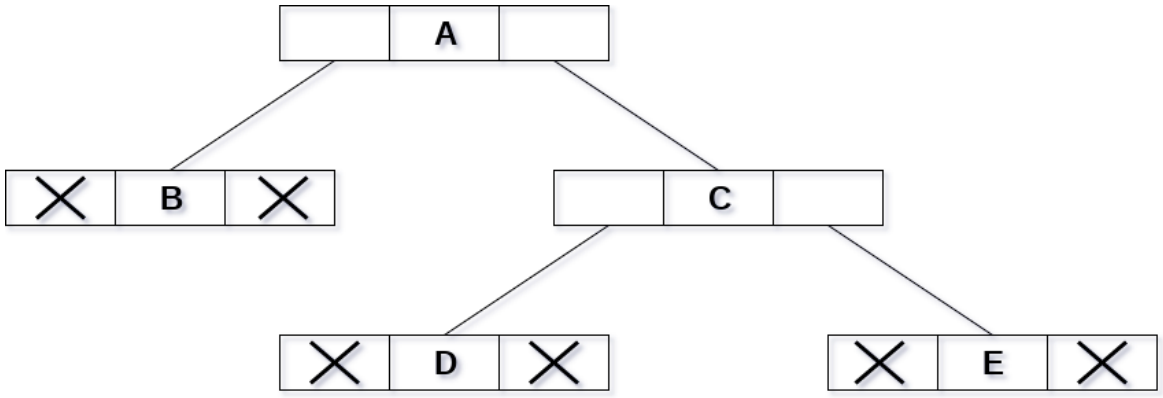
SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order Traversal	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

Binary Tree representation

There are two types of representation of a binary tree:

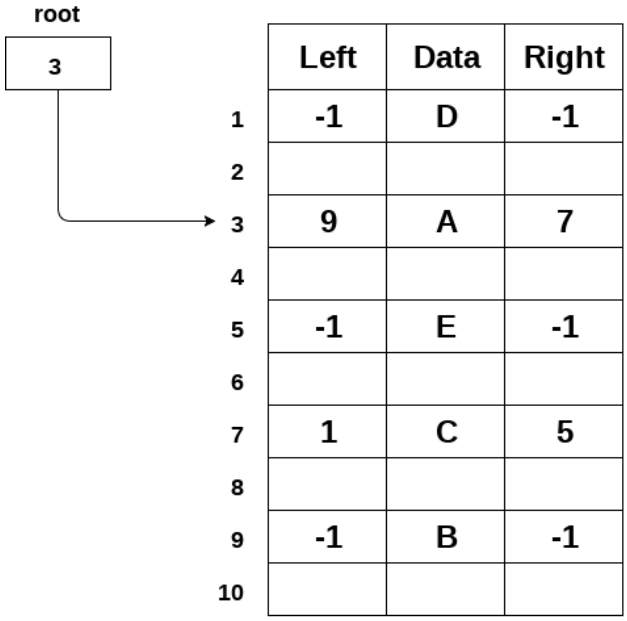
1. Linked Representation

In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.
Consider the binary tree given in the figure below.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.

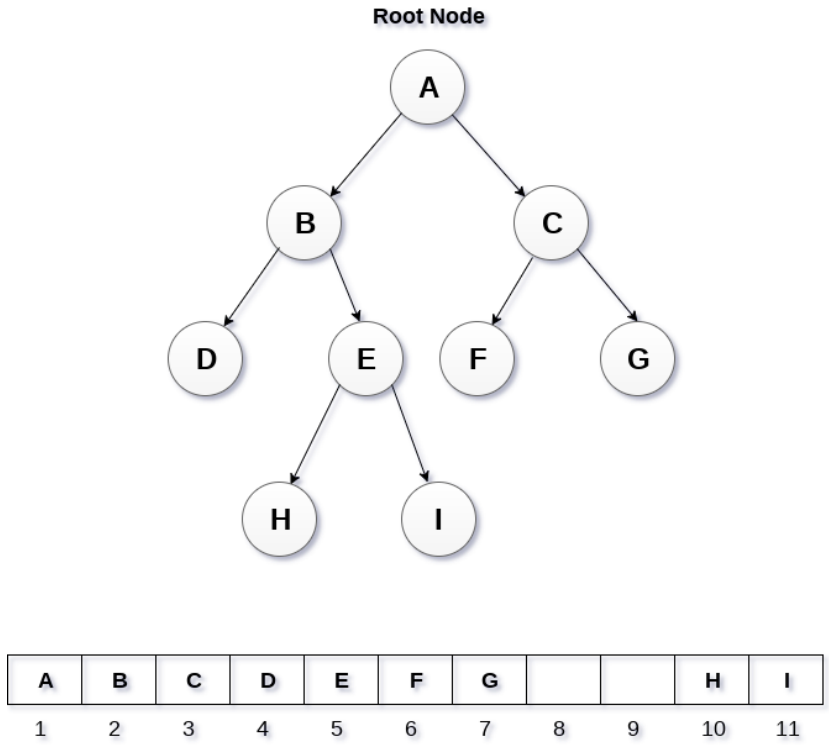


	Left	Data	Right
1	-1	D	-1
2			
3	9	A	7
4			
5	-1	E	-1
6			
7	1	C	5
8			
9	-1	B	-1
10			

Memory Allocation of Binary Tree using linked Representation

2. Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.

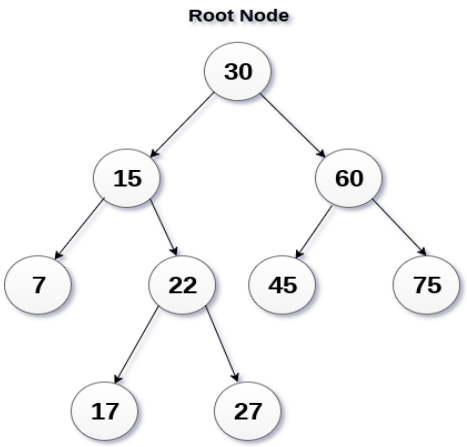


Sequential Representation of Binary Tree

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Advantages of using binary search tree

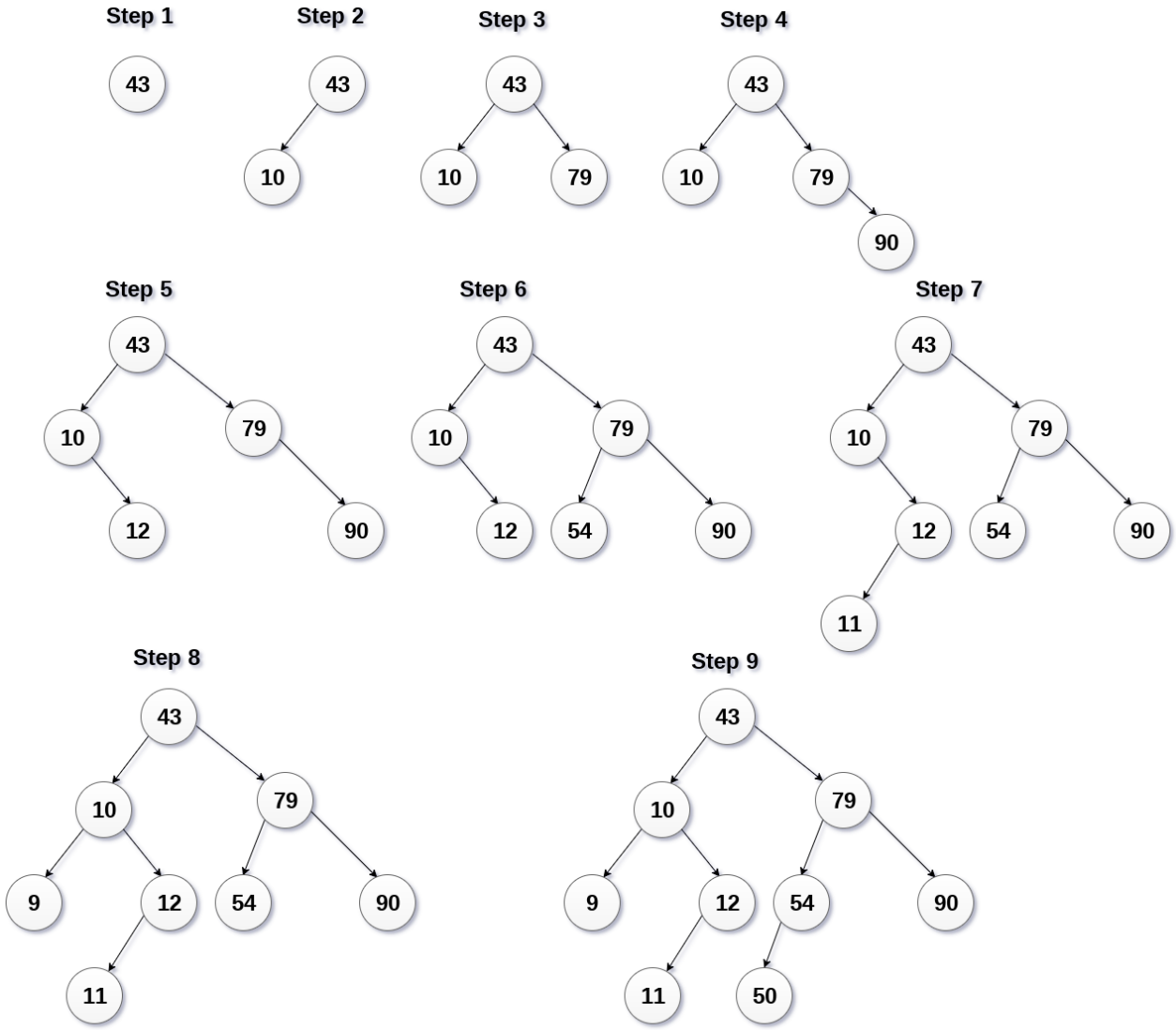
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $o(\log_2 n)$ time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Q. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

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.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.

3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

Program to implement BST operations

```
1.  #include <iostream>
2.  #include <stdlib.h>
3.  using namespace std;
4.  struct Node {
5.     int data;
6.     Node *left;
7.     Node *right;
8.  };
9.  Node* create(int item)
10. {
11.    Node* node = new Node;
12.    node->data = item;
13.    node->left = node->right = NULL;
14.    return node;
15. }
16.
17. void inorder(Node *root)
18. {
19.    if (root == NULL)
20.       return;
21.
22.    inorder(root->left);
23.    cout<< root->data << "   ";
24.    inorder(root->right);
25. }
26. Node* findMinimum(Node* cur)
27. {
28.    while(cur->left != NULL) {
29.       cur = cur->left;
30.    }
31.    return cur;
32. }
33. Node* insertion(Node* root, int item)
34. {
35.    if (root == NULL)
36.       return create(item);
37.    if (item < root->data)
38.       root->left = insertion(root->left, item);
39.    else
40.       root->right = insertion(root->right, item);
41.
42.    return root;
43. }
44.
45. void search(Node* &cur, int item, Node* &parent)
46. {
47.    while (cur != NULL && cur->data != item)
48.    {
49.       parent = cur;
50.
51.       if (item < cur->data)
52.          cur = cur->left;
53.       else
54.          cur = cur->right;
55.    }
```

```
56.     }
57.
58.     void deletion(Node*& root, int item)
59.     {
60.        Node* parent = NULL;
61.        Node* cur = root;
62.
63.        search(cur, item, parent);
64.        if (cur == NULL)
65.           return;
66.
67.        if (cur->left == NULL && cur->right == NULL)
68.        {
69.           if (cur != root)
70.           {
71.              if (parent->left == cur)
72.                 parent->left = NULL;
73.              else
74.                 parent->right = NULL;
75.           }
76.           else
77.              root = NULL;
78.
79.           free(cur);
80.        }
81.        else if (cur->left && cur->right)
82.        {
83.           Node* succ  = findMinimum(cur- >right);
84.
85.           int val = succ->data;
86.
87.           deletion(root, succ->data);
88.
89.           cur->data = val;
90.        }
91.
92.        else
93.        {
94.           Node* child = (cur->left)? Cur- >left: cur->right;
95.
96.           if (cur != root)
97.           {
98.              if (cur == parent->left)
99.                 parent->left = child;
100.             else
101.                parent->right = child;
102.          }
103.
104.          else
105.             root = child;
106.          free(cur);
107.       }
108.    }
109.
110.    int main()
111.    {
112.      Node* root = NULL;
113.      int keys[8];
114.      for(int i=0;i<8;i++)
115.       {
116.       cout << "Enter value to be inserted";
117.       cin>>keys[i];
```

```
118.          root = insertion(root, keys[i]);
119.       }
120.
121.       inorder(root);
122.       cout<<"\n";
123.       deletion(root, 10);
124.       inorder(root);
125.       return 0;
126.    }
```

Output:

```
Enter value to be inserted? 10
Enter value to be inserted? 20
Enter value to be inserted? 30
Enter value to be inserted? 40
Enter value to be inserted?  5
Enter value to be inserted? 25
Enter value to be inserted? 15
Enter value to be inserted?  5

5       5       10      15      20      25      30      40
5       5       15      20      25      30      40
```

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

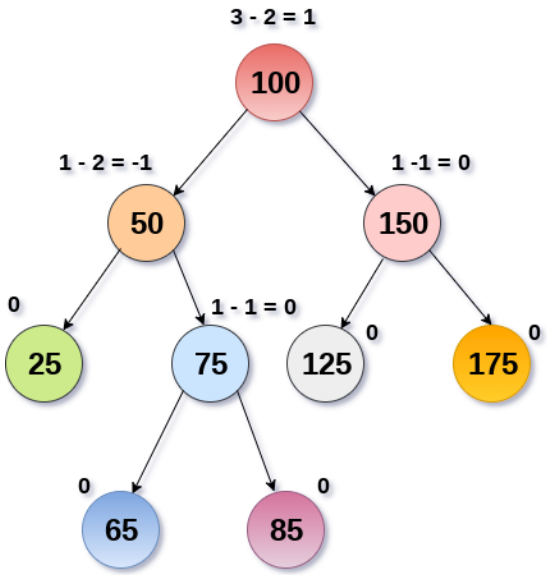
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Why AVL Tree ?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

B Tree

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.

While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.

Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

Insert the node 8 into the B Tree of order 5 shown in the following image.

8 will be inserted to the right of 5, therefore insert 8.

The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.

Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
   - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
   - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

53 is present in the right child of element 49. Delete it.

Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.
The final B tree is shown as follows.

Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.