Unit III

Non-Linear Data Structure-Tree

Unit III: Non-Linear Data Structure- Tree **(07 Hrs.)**

Basic Terminology of Trees, Binary Tree, Threaded Binary Tree, Binary Search Tree, B & B+ Tree, AVL Tree, Splay Tree and Applications of Trees.

3.1Basic Terminology of Trees

Definition: A tree is a finite set of one or more nodes including:

- A starting node called ***root***.
- The rest of nodes can be split into T1, T2, …, Tn, which are ***subtrees*** of the root.

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.

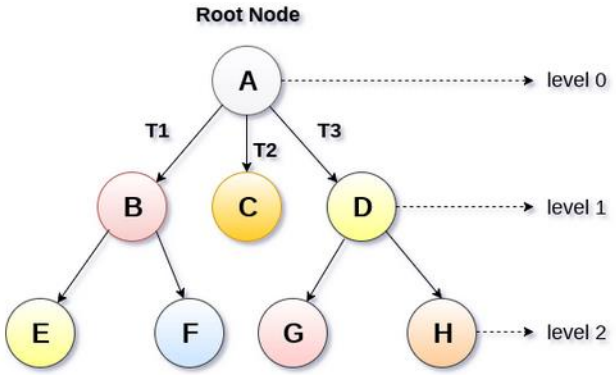


Figure 3.1

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.

3.2 Binary Tree

A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes. A binary tree is an important class of a tree data structure in which a node can have at most two children.

Child node in a binary tree on the left is termed as 'left child node' and node in the right is termed as the 'right child node.'
In the figure mentioned below, the root node 8 has two children 3 and 10; then this two child node again acts as a parent node for 1 and 6 for left parent node 3 and 14 for right parent node 10. Similarly, 6 and 14 has a child node.

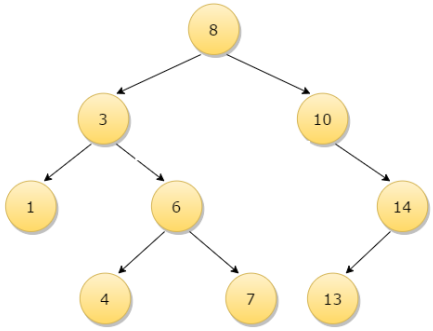


Figure 3.2

A binary tree may also be defined as follows:

- A binary tree is either an empty tree
- Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again

Types of Binary Tree

There are three different types of binary trees that will be discussed in this lesson:

- **Full binary tree**: Every node other than leaf nodes has 2 child nodes.
- **Complete binary tree**: All levels are filled except possibly the last one, and all nodes are filled in as far left as possible.
- **Perfect binary tree**: All nodes have two children and all leaves are at the same level.

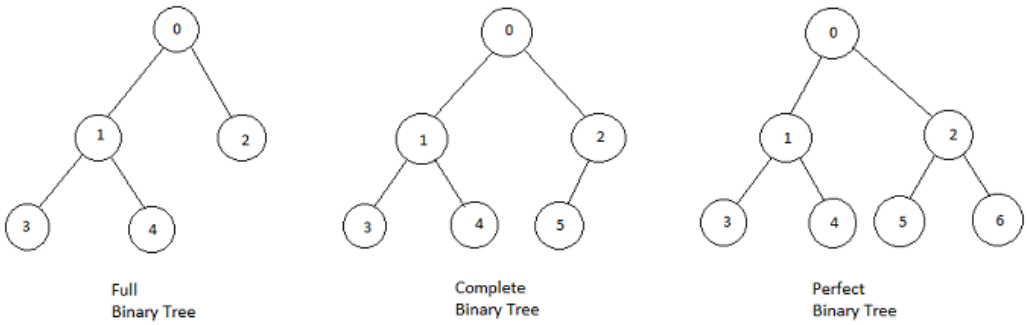


Figure 3.3

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**.

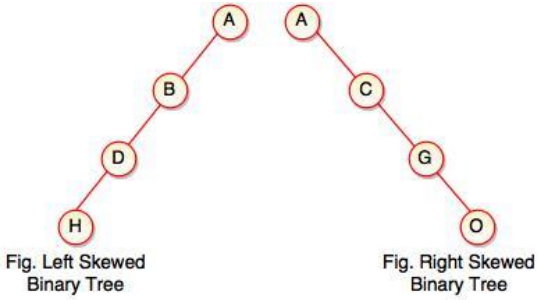


Figure 3.4

- 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.

Tree Traversal

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

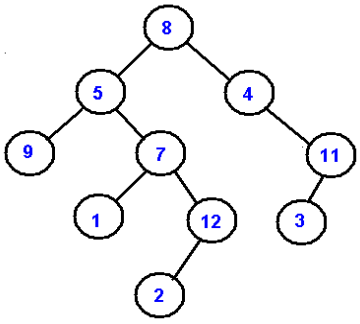
- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

As an example consider the following tree and its four traversals:



PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Figure 3.5

To **traverse a binary tree in preorder**, following operations are carried out:

1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

Note: Preorder traversal is also known as VLR traversal.

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Visit(t);
```

```
                    Preorder(t->lchild);
                    Preorder(t->rchild);
            }
}
```

To traverse a binary tree in inorder traversal, following operations are carried out:

1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

Note: Inorder traversal is also known as LVR traversal.

```
Algorithm inorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

To traverse a binary tree in postorder traversal, following operations are carried out:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

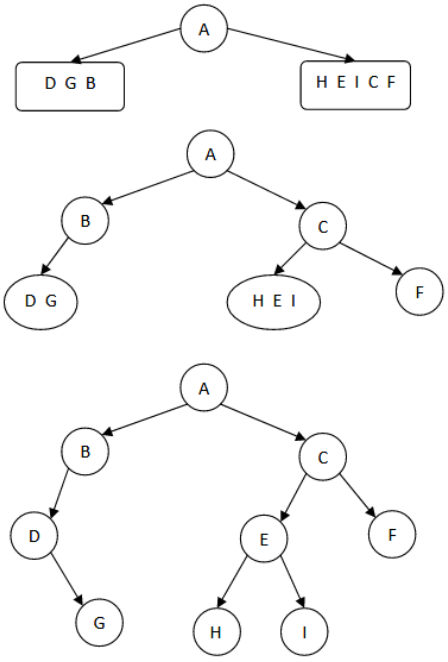
Note: Postorder traversal is also known as LRV traversal.

```
Algorithm postorder(t)
/*t is a binary tree .Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```

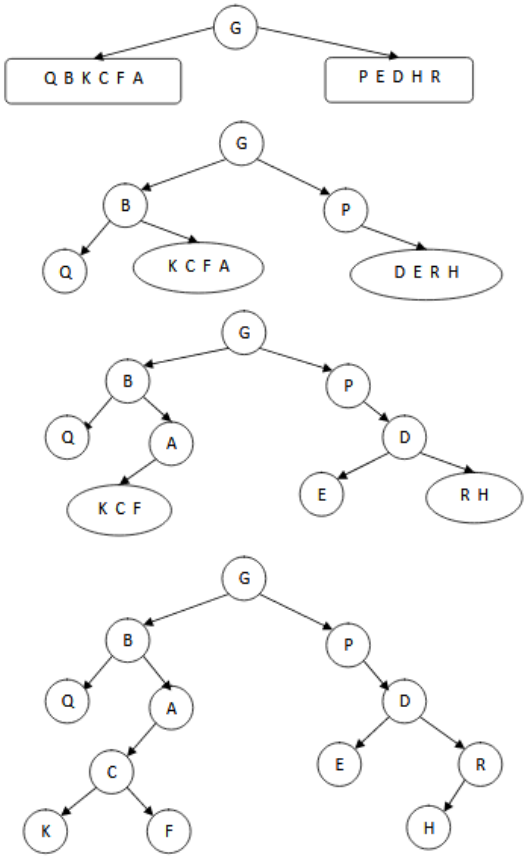
Construct a tree for the given Inorder and Postorder traversals

Inorder : D G B A H E I C F

Postorder : G D B H I E F C A



Construct a tree for the given Inorder and Preorder traversals
Preorder : G B Q A C K F P D E R H
Inorder : Q B K C F A G P E D H R



3.3 Threaded Binary Tree

3.3.1 Threads

A binary tree with n nodes has n+ 1 null links. These null links can be replaced by pointers to nodes called threads. Threads are constructed using the following rules:

1. A null right child pointer in a node is replaced by a pointer to the inorder successor of p(i.e., the node that would be visited after p when traversing the tree inorder).
2. A null left child pointer in a node is replaced by a pointer to the inorder predecessor of p.

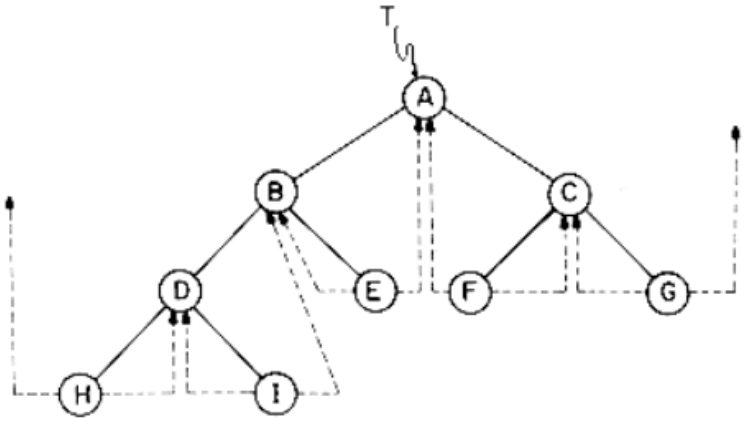


Figure 3.6

In figure 5.10 we see that two threads have been left dangling in LCHILD(A) and RCHILD(D). In order that we leave no loose threads we will assume a head node for all threaded binary trees. Then the complete memory representation for the tree of figure 5.10 is shown in figure 5.11. The tree T is the left subtree of thehead node. Two additional fields of the node structure,LBIT and RBIT are introduced here. If ptr->LBIT=*0*, then ptr->left-child contains a thread, otherwise it contains a pointer to the left child. Similarly, ptr->RBIT=*0*, then ptr->right-child contains a thread, otherwise it contains a pointer to the left child

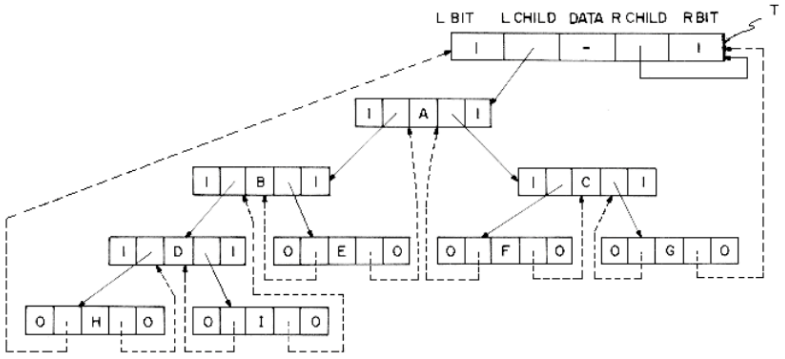


Figure 3.7

Memory Representation of Threaded Tree

We assume that an empty binary tree is represented by its head node as

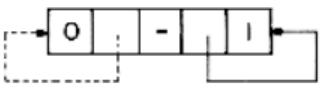


Figure 3.8

Advantages of Thread Binary Tree

Non-recursive pre-order, in-order and post-order traversal can be implemented without a stack.

Disadvantages of Thread Binary Tree

1. Insertion and deletion operation becomes more difficult.
2. Tree traversal algorithm becomes difficult.
3. Memory required to store a node increases. Each node has to store the information whether the links is normal links or threaded links.

3.4 Binary Search Tree

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. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root. This rule will be recursively applied to all the left and right sub-trees of the root. Following is a pictorial representation of BST −

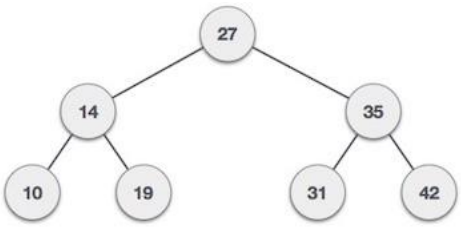


Figure 3.9

We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Search

We describe a recursive algorithm to search for a key k in a treeT first,if Tis empty, the search fails. Second, if k is equal to the key inT's root, the search is successful. Otherwise, we searchT's left or right subtree recursively for k depending on whether it is less or greater than the key in the root.

Algorithm

```
bool Search(TreeNode* b, KeyType k)
        {
            if (b == 0) return 0;
            if (k == b->data) return 1;
            if (k < b->data) return Search(b->LeftChild,k);
            if (k > b->data) return Search(b->RightChild,k);
        }
```

Insert

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
typedef TreeNode* TreeNodePtr;

Node*  Insert(TreeNodePtr& b, KeyType k)
   {
       if (b == 0) {b = new TreeNode; b->data= k; return b;}
       if (k == b->data) return 0; // don't permit duplicates if (k < b->data) Insert(b->LeftChild,
k);
       if (k > b->data) Insert(b->RightChild, k);
   }
```

Delete

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

1.The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.

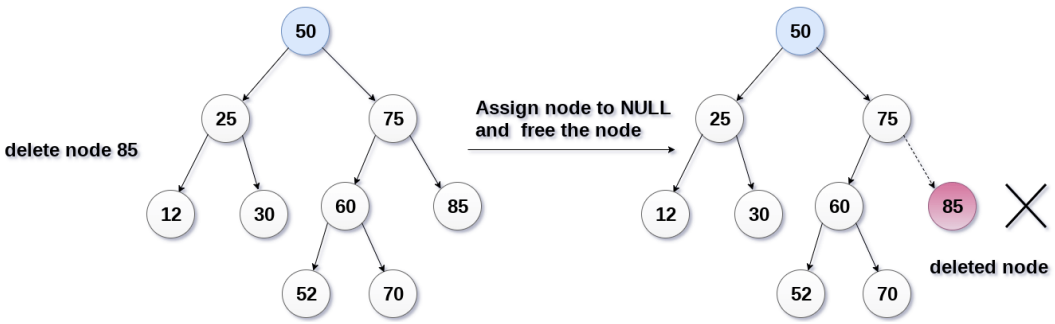


Figure 3.10

2. The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.

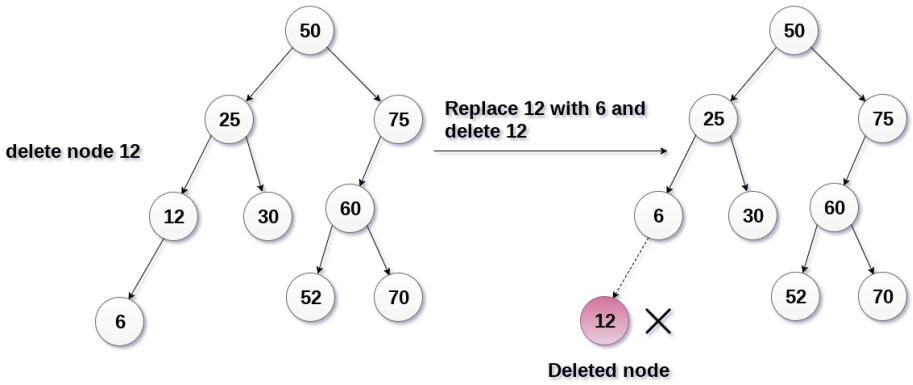


Figure 3.11

3.The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below. 6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.

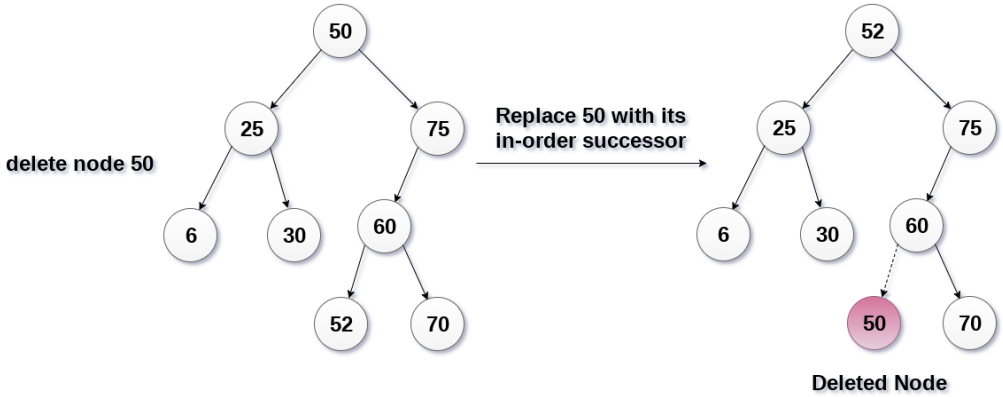


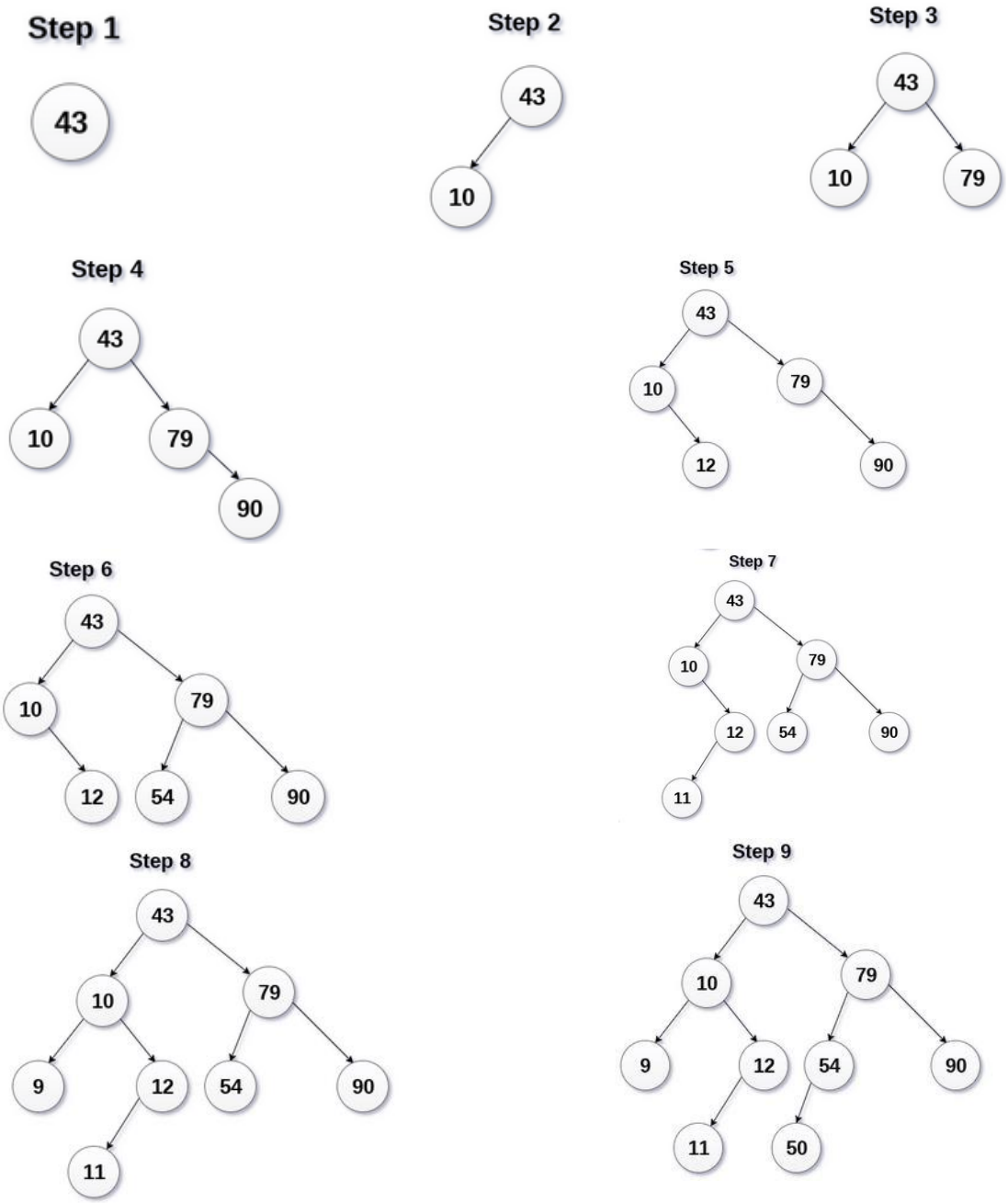
Figure 3.12

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.



3.5 B-Tree

B-Tree is a self-balancing search tree. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size

B-Tree of Order m has the following properties

- **Property #1** - All **leaf nodes** must be **at same level**.

- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

The following table defines the number of subtrees for non-root nodes:

Order	Number of Subtrees	
	Minimum	Maximum
3	2	3
4	2	4
5	3	5
6	3	6
...	...	...
m	Ceiling |m/2|	m

Figure 3.13

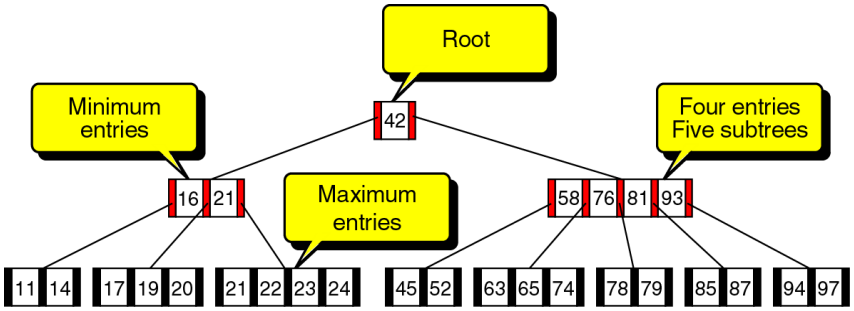


Figure 3.14

A B-tree of order; m=5 has

Min entry : Ceiling|5/2| - 1 = 2 entries.

Max entry: 5 – 1 = 4 entries.

Min subtrees : Ceiling|5/2|=3

Max subtrees: 5

Example

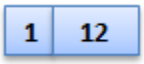
Create a B-Tree with key :- 1,12,8,2,25,6,14,28,17,7,52,16,48,68,3,26,29,53,55,45,67.

Order = 5

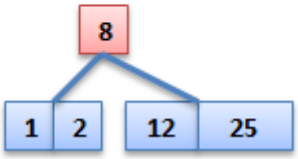
Procedure for adding key in b-tree

Step1. Add first key as root node.

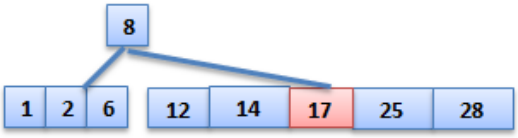
Step2. Add next key at the appropriate place in sorted order.

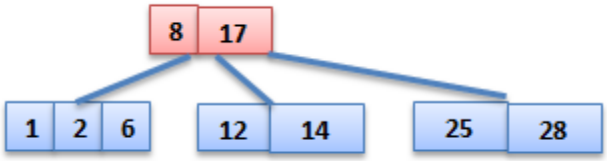


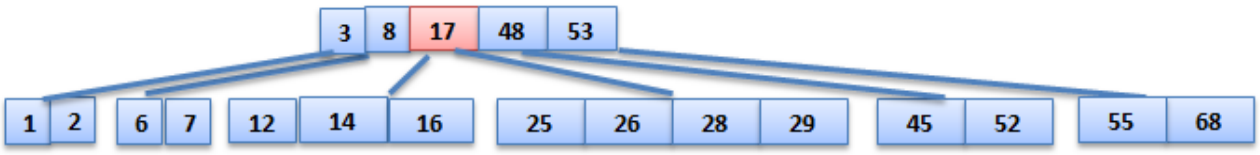
Step3. Same process applied until root node full. if root node full then spliting process applied.

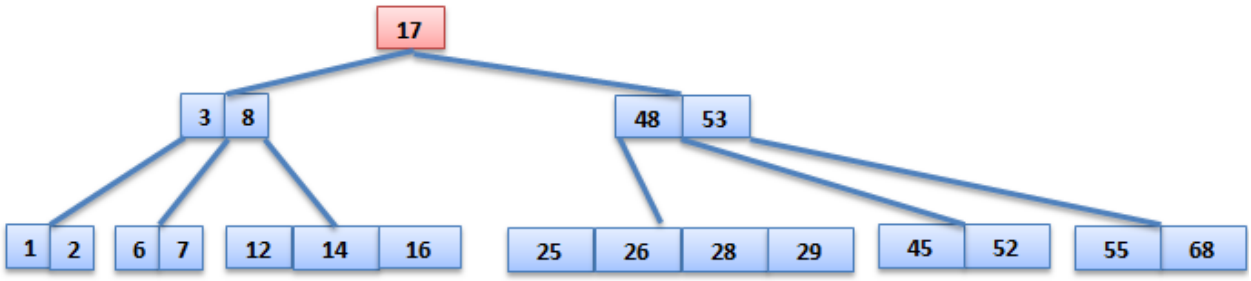


Some important steps







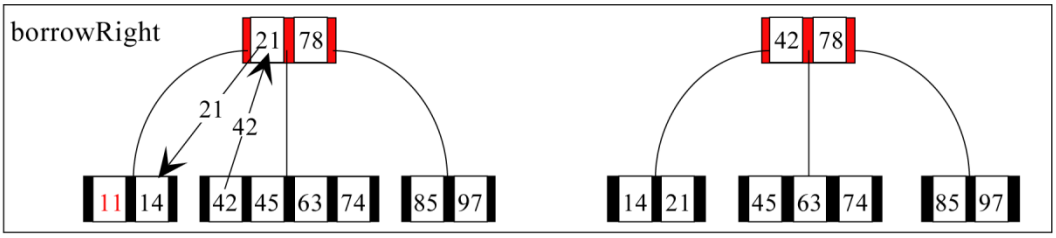


B-Tree Deletion

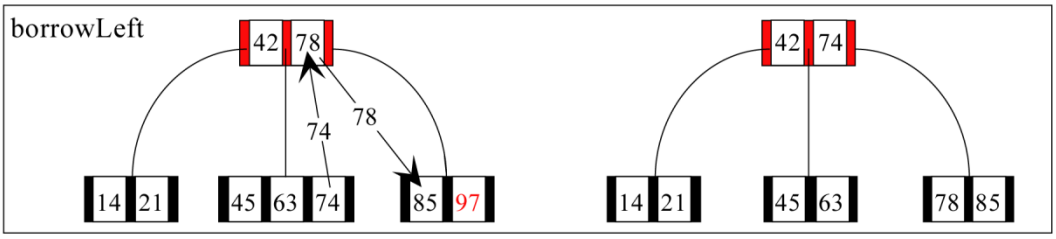
For deletion in b tree we wish to remove from a leaf. There are different possible case for deletion in b tree.

Let k be the key to be deleted, x the node containing the key. Then the cases are:

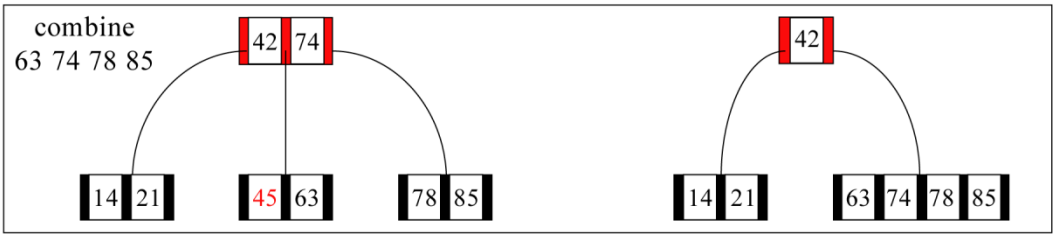
(a) Delete 11



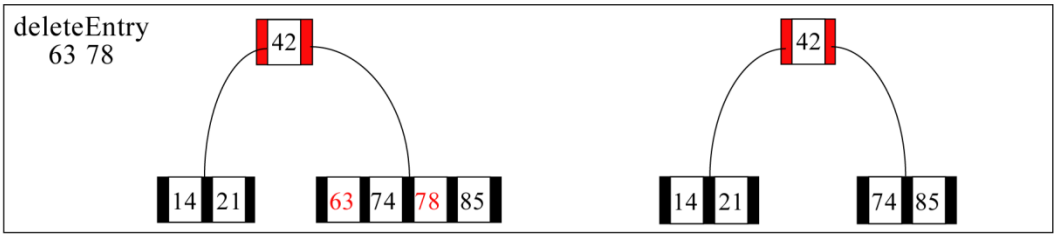
(b) Delete 97



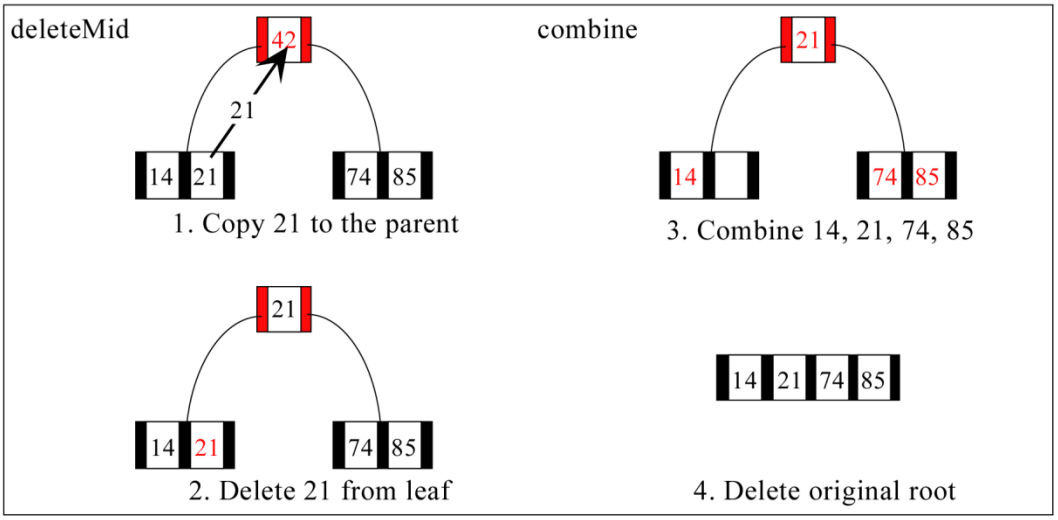
(c) Delete 45



(d) Delete 63 and 78



(e) Delete 42



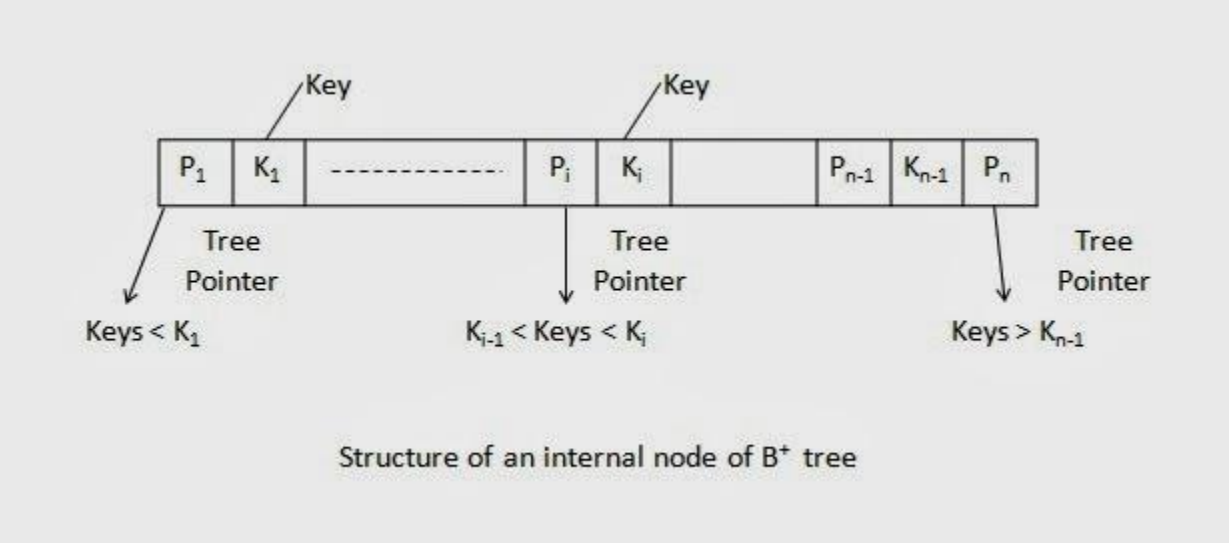
1.6 B+ Tree

B+ tree is a variation of [B-tree data structure](). In a B+ tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree structure of a leaf node differ from the structure of internal nodes.

The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
The leaf nodes of the B+ tree are linked together to provide ordered access on the search field to the
records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Structure of Internal node

The structure of the internal nodes is shown below:

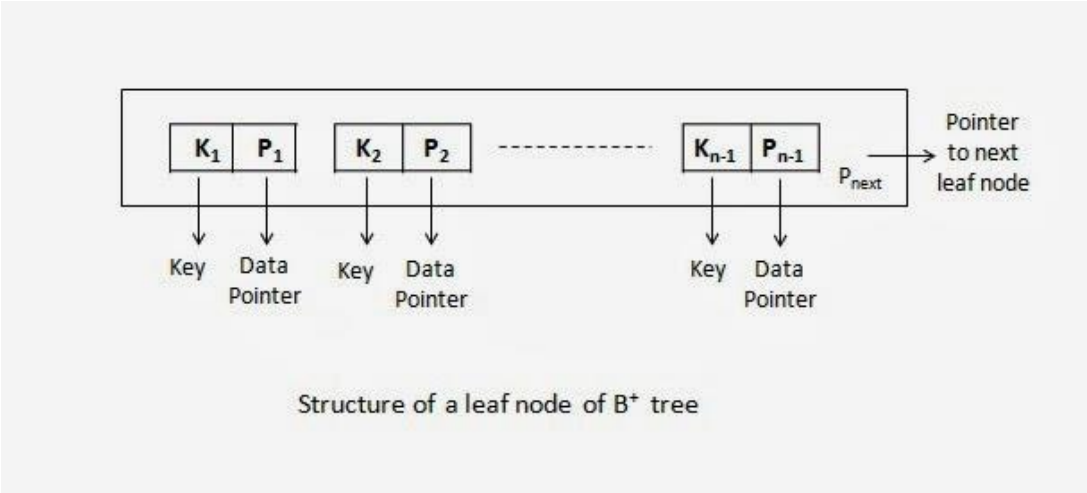


Structure of an internal node of B+ tree

- Each internal node is of the form < P1, K1, P2, K2 . . . Pn-1, Kn-1,Pn > where Ki is the key and Pi is a tree pointer
- Within each internal node, K1 < K2, . . . < Kn-1
- For all search field value x in the subtree pointed at by Pi, we have Ki-1 x <= Ki.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least ⌈(P/2)⌉ tree pointers.

Structure of a leaf node

The structure of a leaf node of a B+ tree is shown below:



Structure of a leaf node of B+ tree

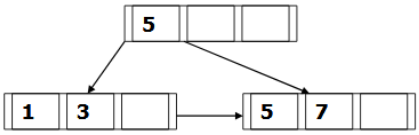
- Each leaf node is of the form <<K1, P1>, <K2, P2> . . . <Kn-1, Pn-1>, Pnext> Within each leaf node, K1 < K2 . . . < Kn-1.
- Pi is a data pointer that points to the record whose search field value is Ki.
- Each leaf node has at least ⌈(P/2)⌉ values.
- All leaf nodes are at the same level.

Construct a B+ tree of order 3 for the following data 3, 5, 7, 9, 2, 4, 6, 8, 10

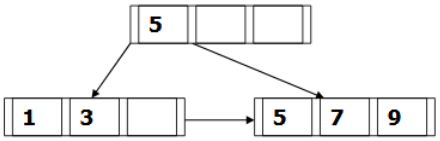
Insert 1

Insert 3, 5

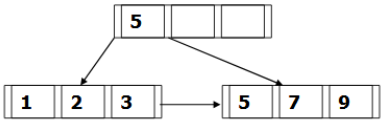
Insert 7



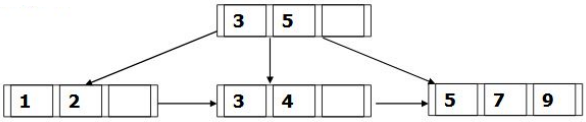
Insert 9



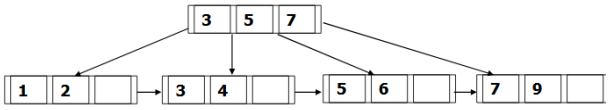
Insert 2



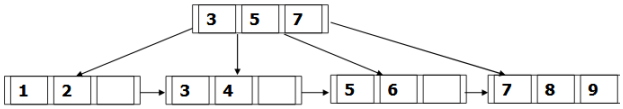
Insert 4



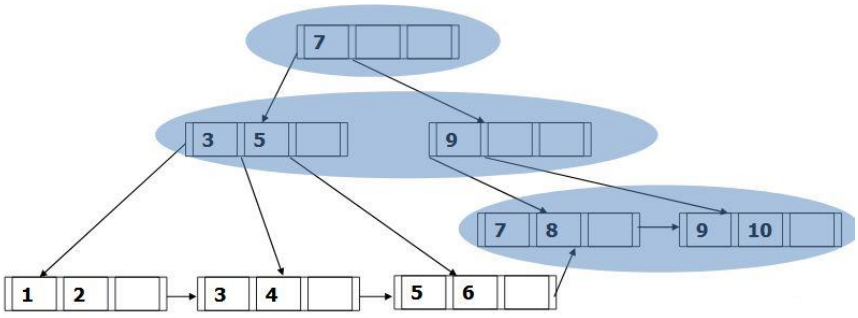
Insert 6



Insert 8



Insert 10



3.7 AVL Tree

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −

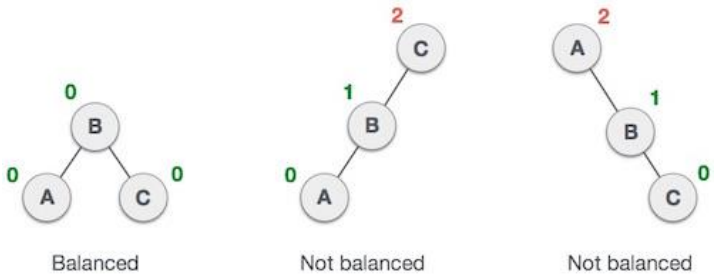


Figure 3.15

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.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- 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

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 −

Figure 3.16

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

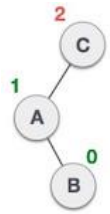
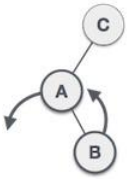
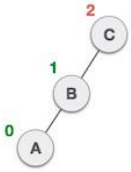
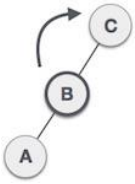
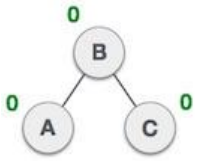
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

Figure 3.17

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

Left-Right 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
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

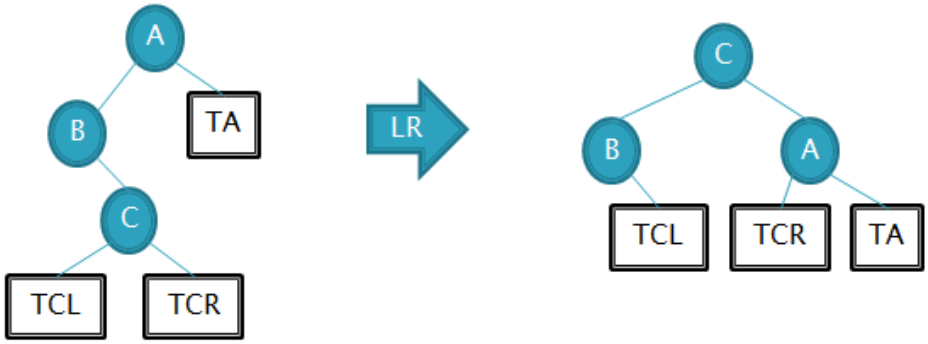
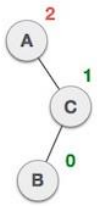
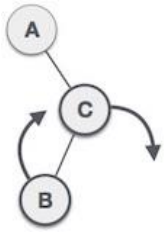
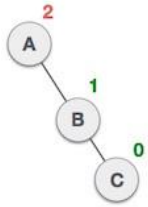
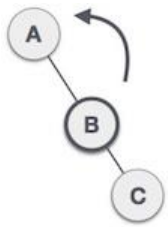
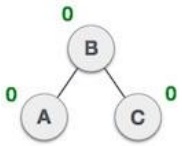


Figure 3.18

Right-Left Rotation

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

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



The tree is now balanced.

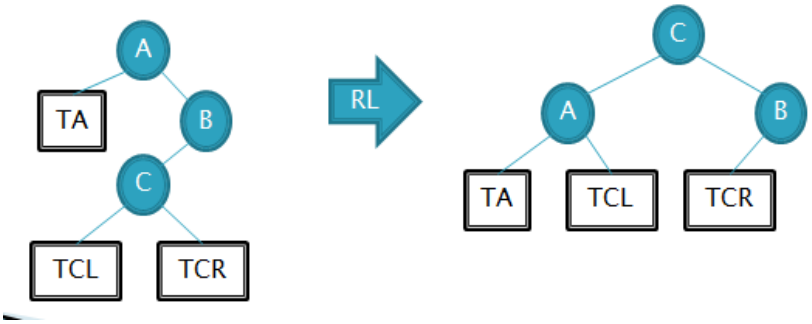
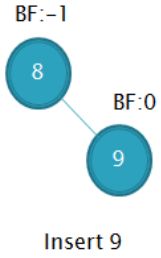


Figure 3.19

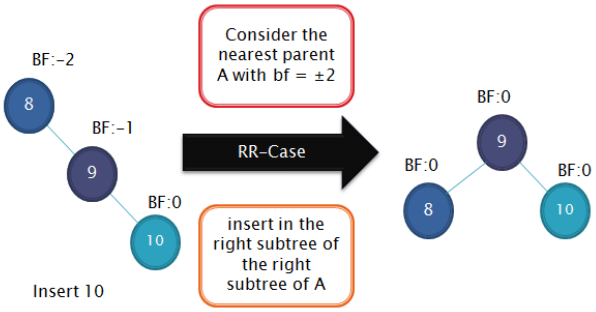
Construct an AVL Tree for given keys 8, 9, 10, 2, 1, 5, 3, 6, 4, 7, 11, 12

Step 1

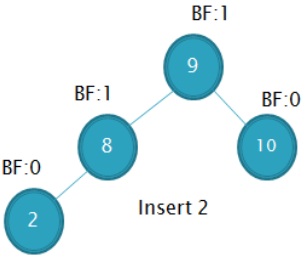
Step 2



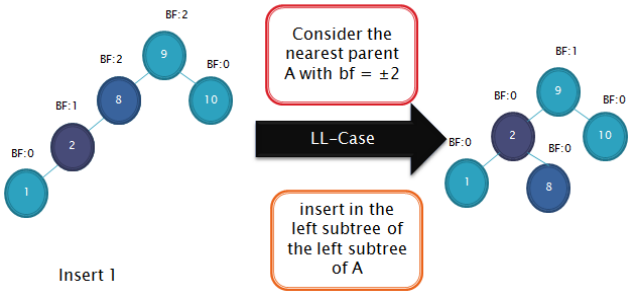
Step 3



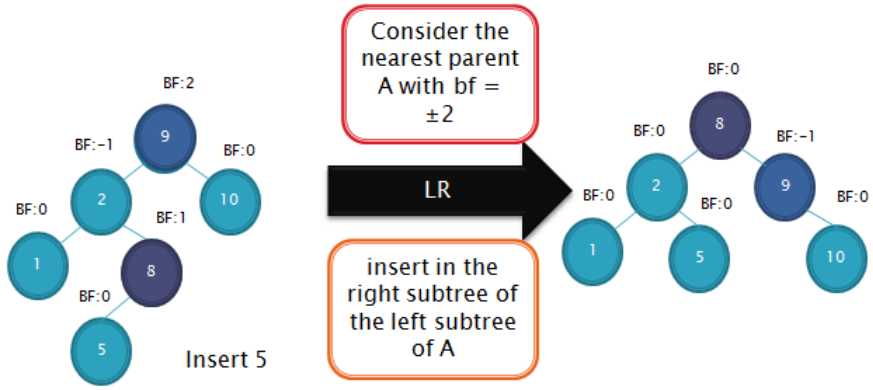
Step 4



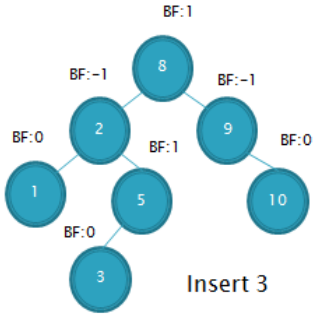
Step 5



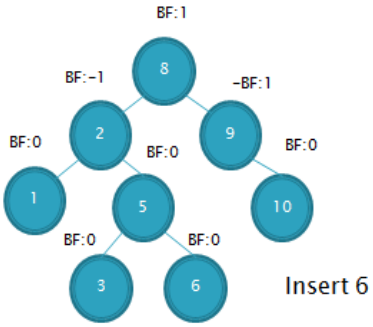
Step 6



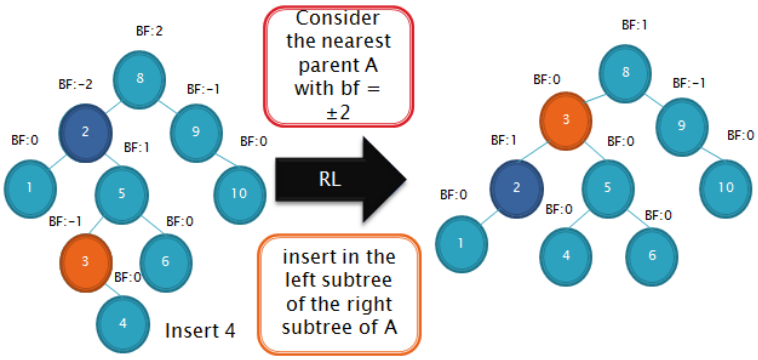
Step 7



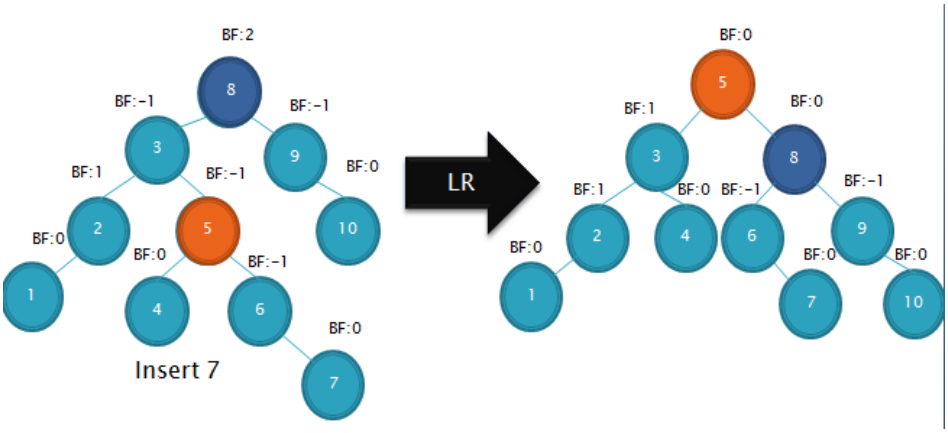
Step 8



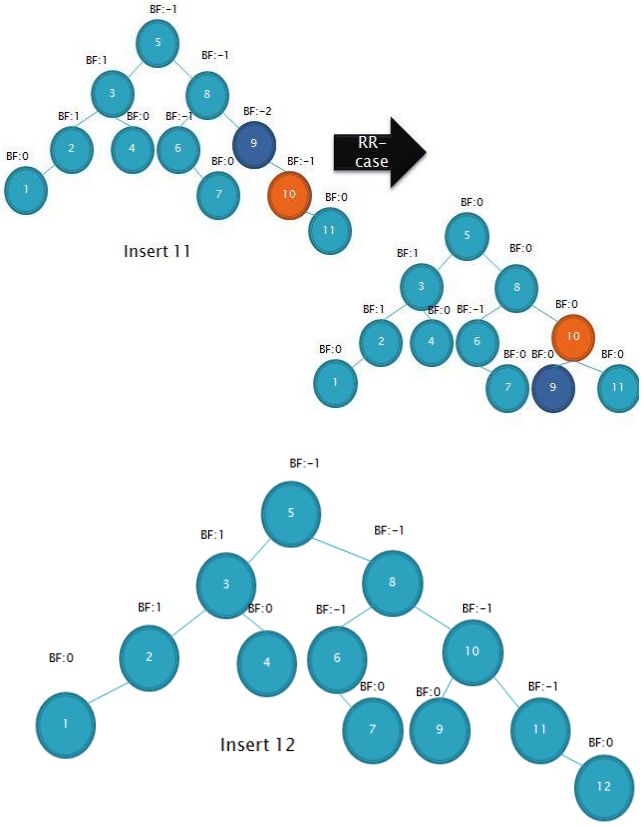
Step 9



Step 10



Step 11



3.8 Splay Tree

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

Example

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...

Figure 3.20

Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Figure 3.21

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

Figure 3.22

Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

Figure 3.23

Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...

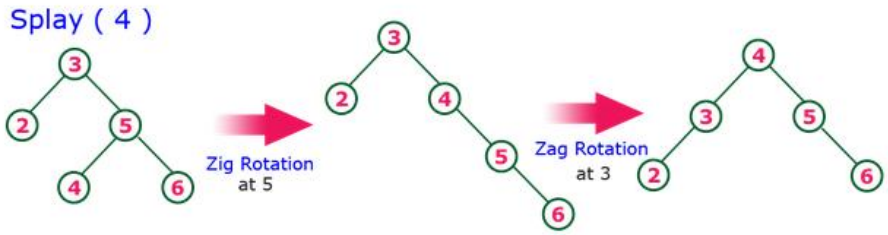


Figure 3.24

Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...

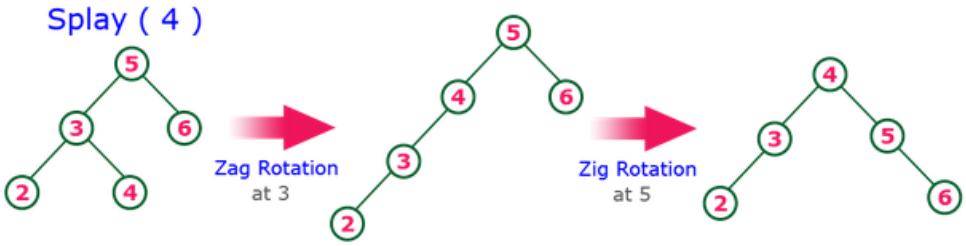


Figure 3.25

Every Splay tree must be a binary search tree but it is need not to be balanced tree.

```
SPLAY(T, n)
   while n.parent != NULL //node is not root

      if n.parent == T.root //node is child of root, one rotation
         if n == n.parent.left //left child
            RIGHT_ROTATE(T, n.parent)
         else //right child
            LEFT_ROTATE(T, n.parent)
```

```
else //two rotations
    p = n.parent
    g = p.parent

    if n.parent.left == n and p.parent.left == p //both are left children
        RIGHT_ROTATE(T, g)
        RIGHT_ROTATE(T, p)
    else if n.parent.right == n and p.parent.right == p //both are right children
        LEFT_ROTATE(T, g)
        LEFT_ROTATE(T, p)
    else if n.parent.right == n and p.parent.left == p
        LEFT_ROTATE(T, p)
        RIGHT_ROTATE(T, g)
    else
        RIGHT_ROTATE(T, p)
        LEFT_ROTATE(T, g)
```

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- Step 1 - Check whether tree is Empty.
- Step 2 - If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- Step 4 - After insertion, **Splay** the **newNode**

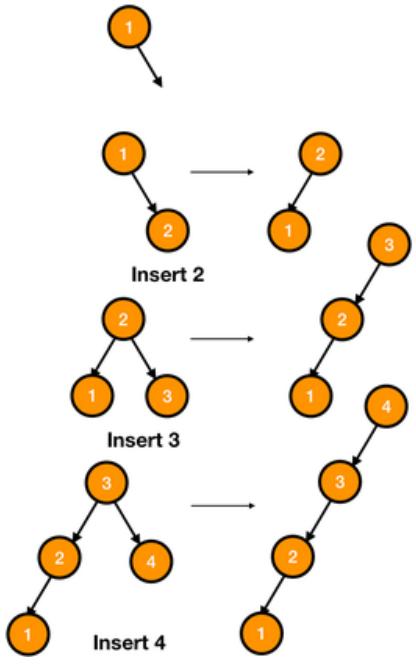


Figure 3.26

```
INSERT(T, n)
   temp = T.root
   y = NULL
   while temp != NULL
      y = temp
      if n.data < temp.data
         temp = temp.left
      else
         temp = temp.right
   n.parent = y
   if y==NULL
      T.root = n
   else if n.data < y.data
      y.left = n
   else
      y.right = n

   SPLAY(T, n)
```

Searching in a Splay Tree
Searching is just the same as a normal binary search tree, we just splay the node which was searched to the root

```
SEARCH(T, n, x)
   if x == n.data
      SPLAY(T, n)
      return n
   else if x < n.data
      return search(T, n.left, x);
   else if x > n.data
      return search(T, n.right, x);
   else
      return NULL
```

Following are the different cases to delete a key **k** from splay tree.

1. If **Root is NULL:** We simply return the root.
2. Else Splay the given key k. If k is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
3. If new root's key is not same as **k**, then return the root as **k** is not present.
4. Else the key **k** is present.
    - Split the tree into two trees **Tree1** = root's left subtree and **Tree2** = root's right subtree and delete the root node.
    - Let the root's of **Tree1** and **Tree2** be **Root1** and **Root2** respectively.
    - If **Root1** is NULL: Return **Root2**.
    - Else, Splay the maximum node (node having the maximum value) of **Tree1**.
    - After the Splay procedure, make **Root2** as the right child of **Root1** and return **Root1**.

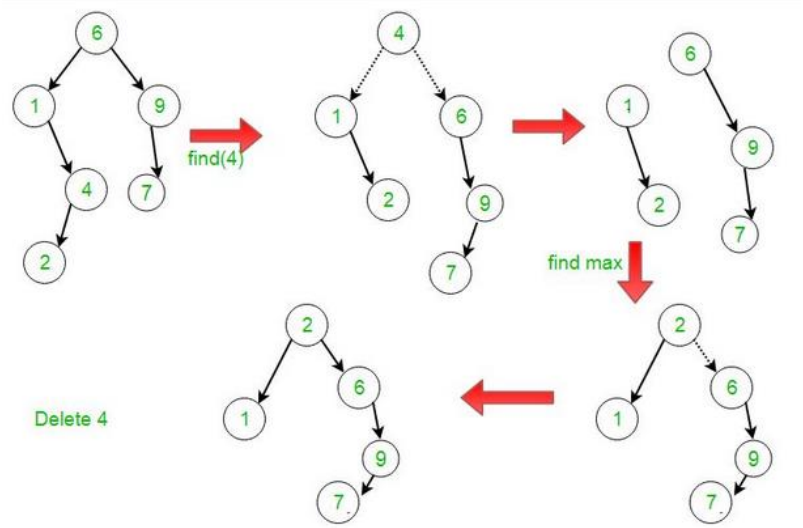


Figure 3.27

```
DELETE(T, n)
   left_subtree = new splay_tree
   right_subtree = new splay_tree
   left_subtree.root = T.root.left
   right_subtree = T.root.right
   if left_subtree.root != NULL
      left_subtree.root.parent = NULL
   if right_subtree.root != NULL
      right_subtree.root.parent = NULL

   if left_subtree.root != NULL
      m = MAXIMUM(left_subtree, left_subtree.root)
      SPLAY(left_subtree, m)
      left_subtree.root.right = right_subtree.root
      T.root = left_subtree.root
   else
      T.root = right_subtree.root
```

3.9 Applications of Trees

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree : They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie : Used to implement dictionaries with prefix lookup.
8. Suffix Tree : For quick pattern searching in a fixed text.
9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks
10. As a workflow for compositing digital images for visual effects.