UNIT 7 ADVANCED TREES

7.0 INTRODUCTION

Linked list representations have great advantages of flexibility over the contiguous representation of data structures. But, they have few disadvantages also. Data structures organised as trees have a wide range of advantages in various applications and it is best suited for the problems related to information retrieval.
These data structures allow the searching, insertion and deletion of node in theordered list to be achieved in the minimum amount of time.

The data structures that we discuss primarily in this unit are Binary Search Trees, AVL trees and B-Trees. We cover only fundamentals of these data structures in this unit. Some of these trees are special cases of other trees and Trees are having a large number of applications in real life.

7.1 OBJECTIVES

After going through this unit, you should be able to

- know the fundamentals of Binary Search trees;
- perform different operations on the Binary Search Trees;
- understand the concept of AVL trees;

- understand the concept of B-trees, and
- perform various operations on B-trees.

7.2 BINARY SEARCH TREES

A Binary Search Tree is a binary tree that is either empty or a node containing a key value, left child and right child.

By analysing the above definition, we note that BST comes in two variants namely empty BST and non-empty BST.

The empty BST has no further structure, while the non-empty BST has three components.

The non-empty BST satisfies the following conditions:

a) The key in the left child of a node (if exists) is less than the key in its parentnode.
b) The key in the right child of a node (if exists) is greater than the key in itsparent node.
c) The left and right subtrees of the root are again binary search trees.

The following are some of the operations that can be performed on Binary searchtrees:

- Creation of an empty tree
- Traversing the BST
- Counting internal nodes (non-leaf nodes)
- Counting external nodes (leaf nodes)
- Counting total number of nodes
- Finding the height of tree
- Insertion of a new node
- Searching for an element
- Finding smallest element
- Finding largest element
- Deletion of a node.

7.2.1 Traversing a Binary Search Tree

Binary Search Tree allows three types of traversals through its nodes. They are as follow:

1. Pre Order Traversal
2. In Order Traversal
3. Post Order Traversal

In Pre Order Traversal, we perform the following three operations:

1. Visit the node
2. Traverse the left subtree in preorder

3. Traverse the right subtree in preorder

In Order Traversal,we perform the following three operations:

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder.

In Post Order Traversal, we perform the following three operations:

1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

Consider the BST of *Figure 7.1*

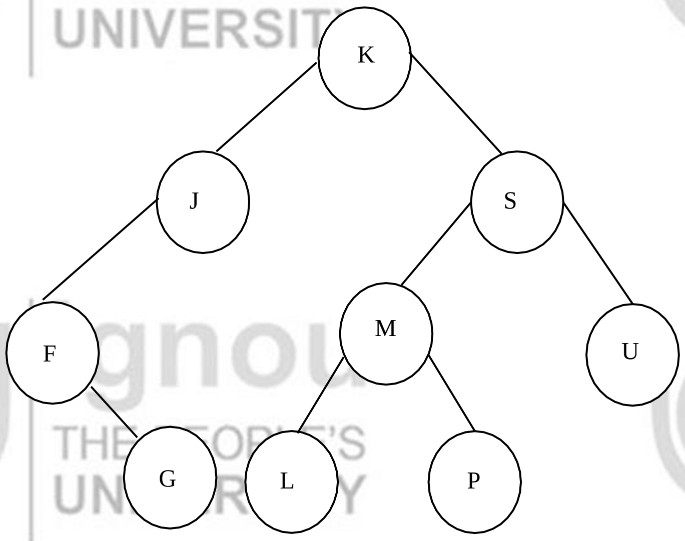


Figure 7.1: A Binary Search Tree(BST)

The following are the results of traversing the BSTof *Figure 7.1:*

Preorder : K J F G S M L P U
Inorder : F G J K L M P S U
Postorder: G F J L P M U S K

7.2.2 Insertion of a node into a Binary Search Tree

A binary search tree is constructed by the repeated insertion of new nodes into abinary tree structure.

Insertion must maintain the order of the tree. The value to the left of a given nodemust be less than that node and value to the right must be greater.

In inserting a new node, the following two tasks are performed :

- Tree is searched to determine where the node is to be inserted.
- On completion of search, the node is inserted into the tree

Example: Consider the BST of *Figure 7.2* After insertion of a new node consisting of value 5, the BST of Figure 7.3 results.

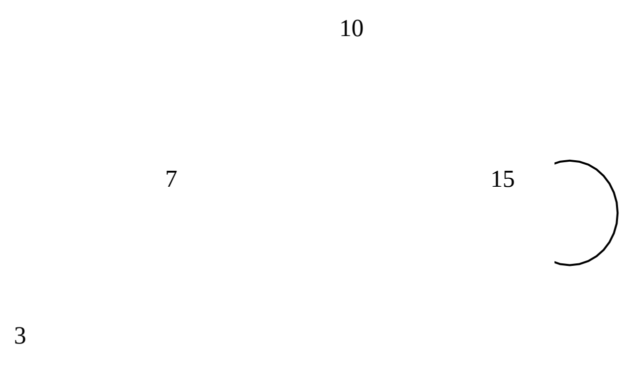


Figure 7.2: A non-empty BST

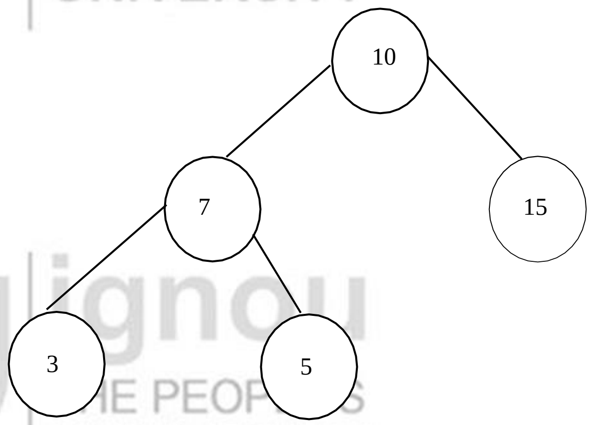


Figure 7.3: Figure 7.2 after insertion of 5

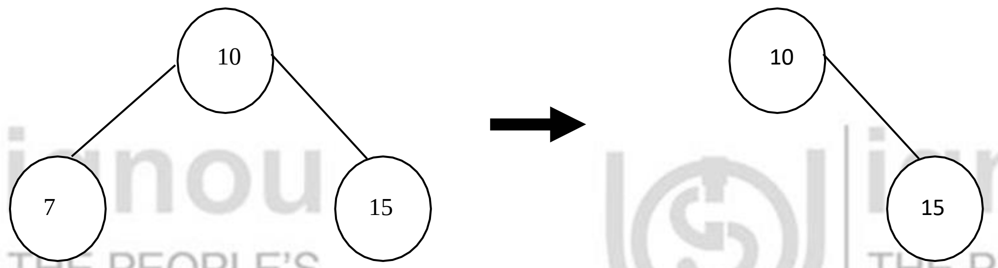
7.2.3 Deletion of a node from a Binary Search Tree

The algorithm to delete a node with key from a binary search tree is not simple where as many cases needs to be considered.

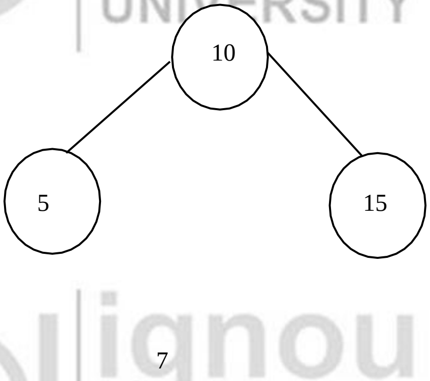
- If the node to be deleted has no sons, then it may be deleted without further adjustment to the tree.
- If the node to be deleted has only one subtree, then its only son can be movedup to take its place.
- The node *p* to be deleted has two subtrees, then its inorder successor *s* musttake its place. The inorder successor cannot have a left subtree. Thus, the right son of *s* can be moved up to take the place of *s*.

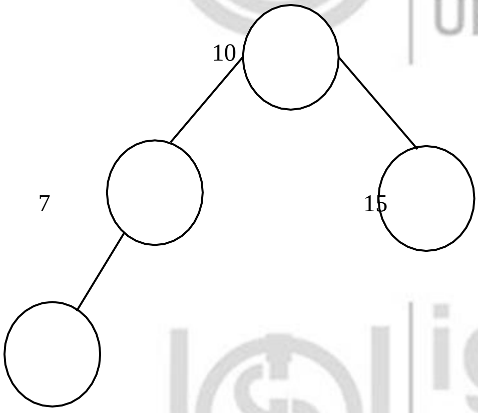
Example: Consider the following cases in which node 5 needs to be deleted.

1. The node to be deleted has no children.

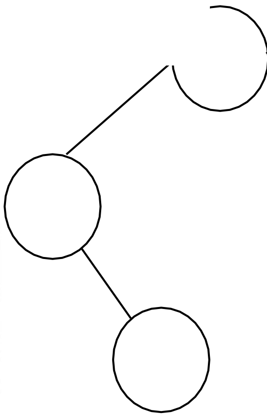


2. The node has one child





3. The node to be deleted has two children. This case is complex. The orderof the binary tree must be kept intact.



☞ Check Your Progress 1

1) What are the different ways of traversing a Binary Search Tree?

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the major features of a Binary Search Tree?

……………………………………………………………………………………

……………………………………………………………………………………

7.3 AVL TREES

An AVL tree is a binary search tree which has the following properties:

- The sub-tree of every node differs in height by at most one.
- Every sub tree is an AVL tree.

Figure 7.4 depicts an AVL tree.

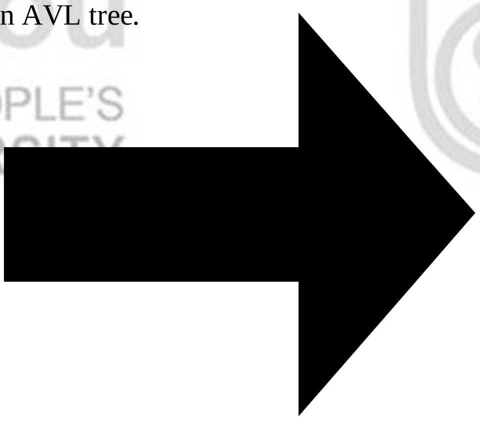


Figure 7.4 : Balance requirement for an AVL tree: the left and right subtree differ by atmost one in height

AVL stands for the names of G.M. Adelson – Velskii and E.M. Landis, two Russian mathematicians, who came up with this method of keeping the tree balanced.

An AVL tree is a binary search tree which has the balance property and in addition to its key, each node stores an extra piece of information: the current balance of its subtree. The three possibilities are:

▶ Left – HIGH (balance factor -1)
The left child has a height that is greater than the right child by 1.

▶ BALANCED (balance factor 0) Both children have the same height

▶ RIGHT – HIGH (balance factor +1)
The right child has a height that is greater by 1.

An AVL tree which remains balanced guarantees O(log n) search time, even in the worst case. Here, n is the number of nodes. The AVL data structure achieves this property by placing restrictions on the difference in heights between the sub-trees of a given node and rebalancing the tree even if it violates these restrictions.

7.3.1 Insertion of a node into an AVL tree

Nodes are initially inserted into an AVL tree in the same manner as an ordinarybinary search tree.

However, the insertion algorithm for an AVL tree travels back along the path it took to find the point of insertion and checks the balance at each node on the path.

If a node is found that is unbalanced (if it has a balance factor of either -2 or +2)then rotation is performed, based on the inserted nodes position relative to the node being examined (the unbalanced node).

7.3.2 Deletion of a node from an AVL tree

The deletion algorithm for AVL trees is a little more complex as there are severalextra steps involved in the deletion of a node. If the node is not a leaf node, then it has at least one child. Then the node must be swapped with either its in-order successor or predecessor. Once the node has been swapped, we can delete it.

If a deletion node was originally a leaf node, then it can simply be removed.

As done in insertion, we traverse back up the path to the root node, checking the balance of all nodes along the path. If unbalanced, then the respective node is found and an appropriate rotation is performed to balance that node.

7.3.3 AVL tree rotations

AVL trees and the nodes it contains must meet strict balance requirements to maintain O(log n) search time. These balance restrictions are maintained usingvarious rotation functions.

The four possible rotations that can be performed on an unbalanced AVL tree are given below. The before and after status of an AVL tree requiring the rotation are shown (refer to *Figures 7.5, 7.6, 7.7 and 7.8*).

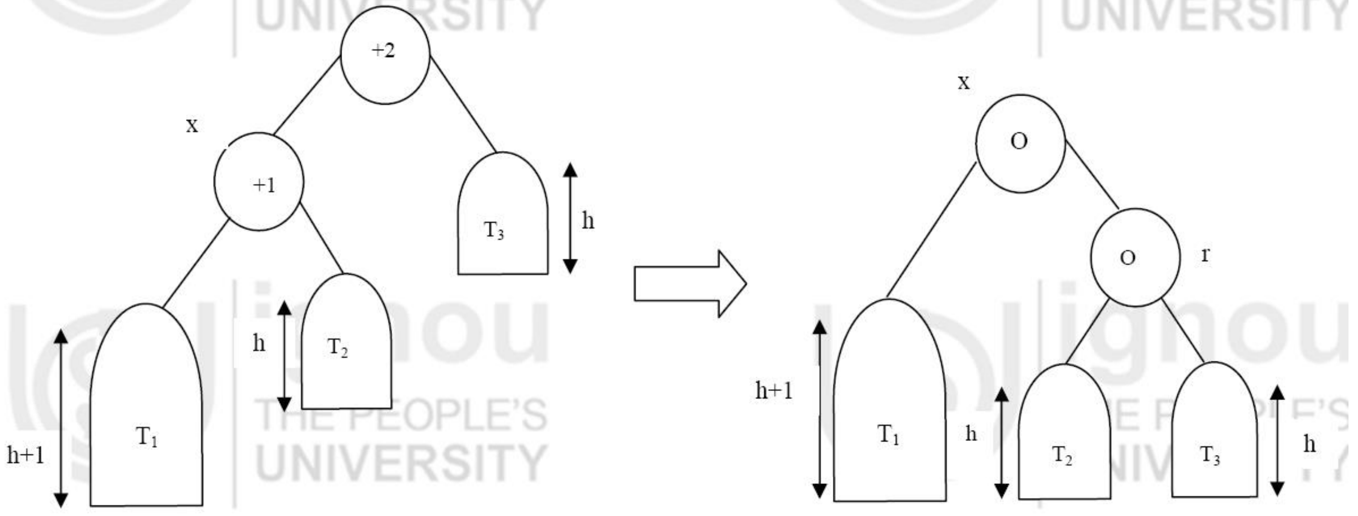


Figure 7.5: LL Rotation

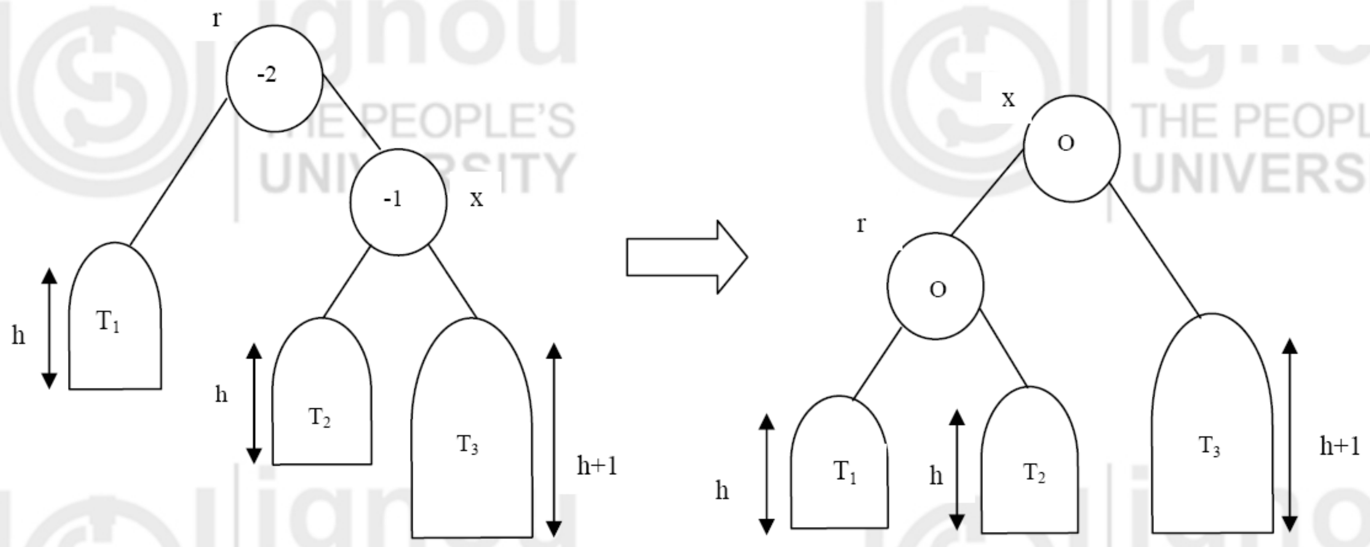


Figure: 7.6 RR Rotati

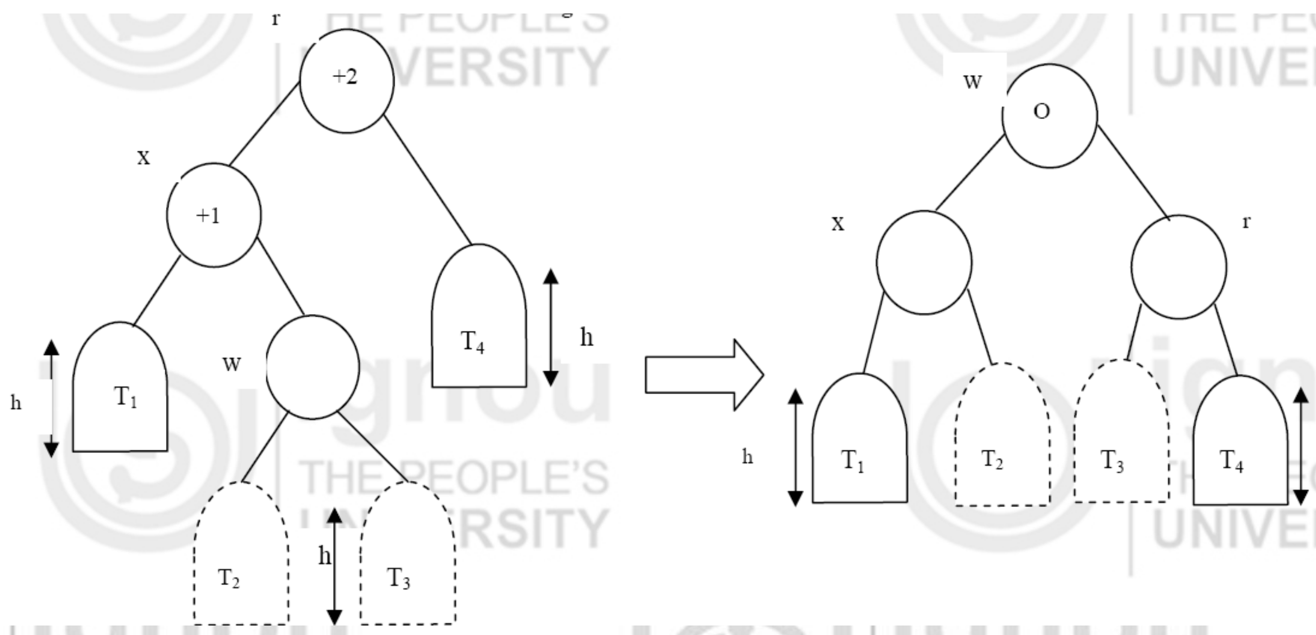


Figure 7.7: LR Rotation

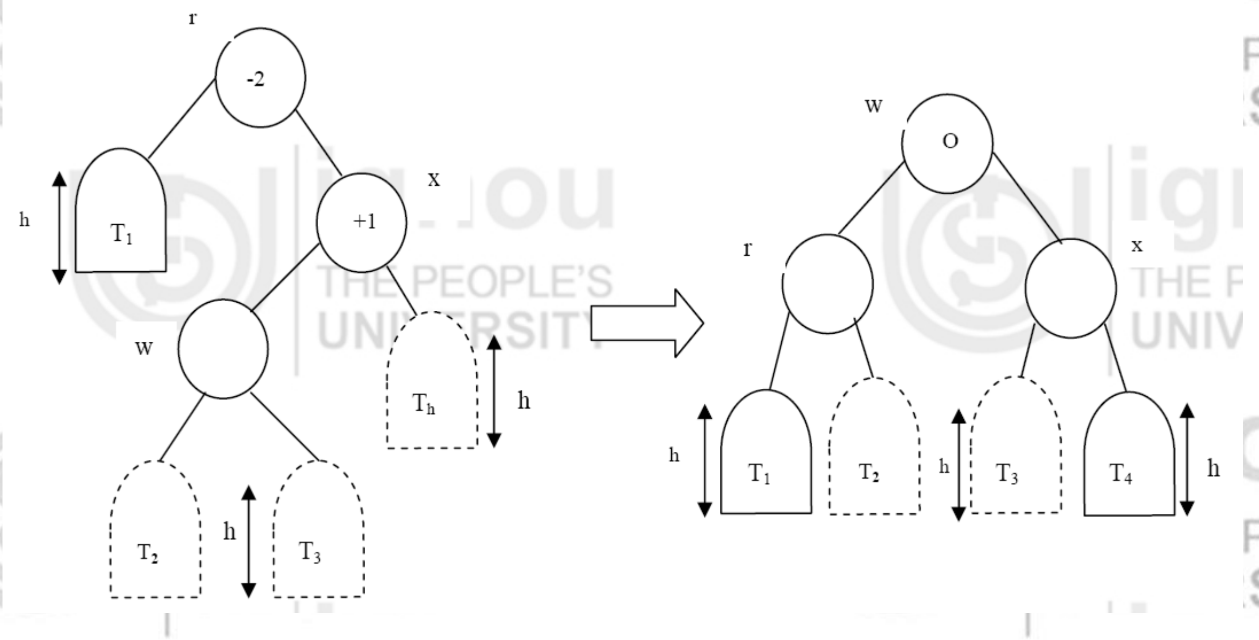


Figure 7.8: RL Rotation

Example: (Single rotation in AVL tree, when a new node is inserted into the AVL tree (LL Rotation)) (refer to *Figure 7.9*).

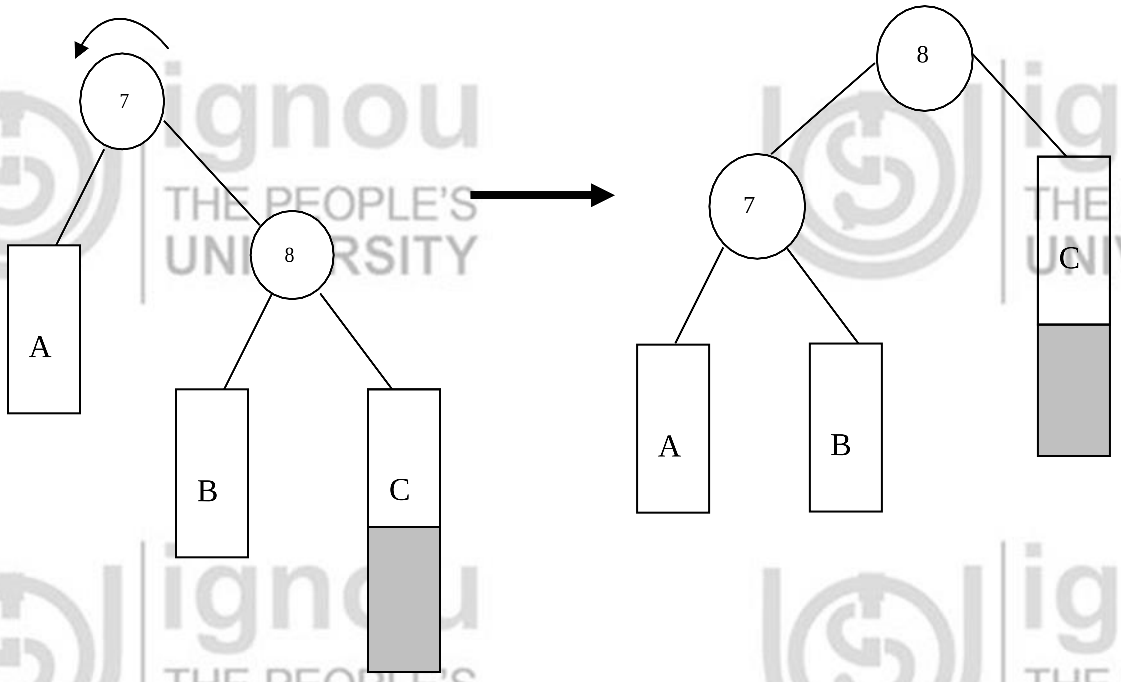


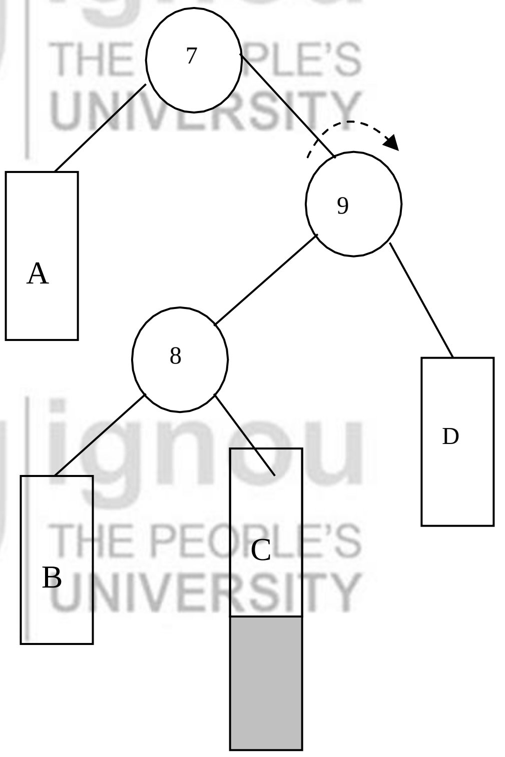
Figure 7.9: LL Rotation

The rectangles marked A, B and C are trees of equal height. The shaded rectangle stands for a new insertion in the tree C. Before the insertion, the tree was balanced, for the right child was taller then the left child by one.

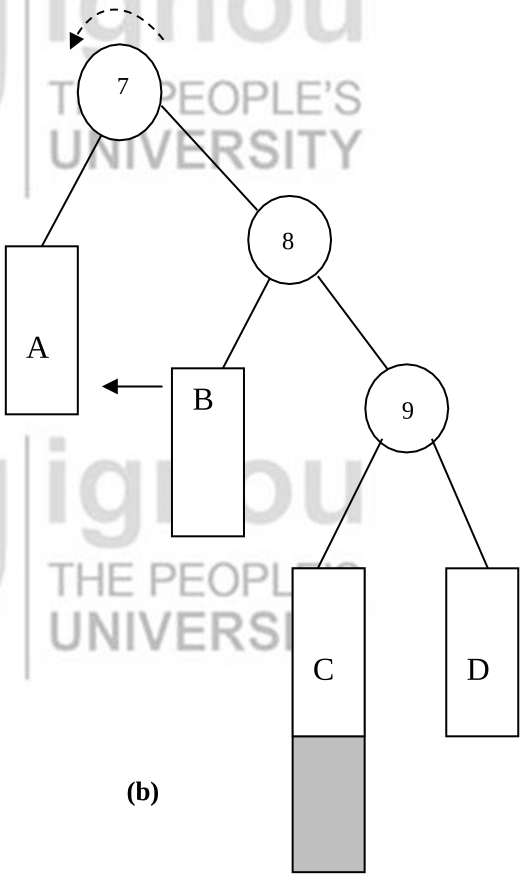
The balance was broken when we inserted a node into the right child of 7, since the difference in height became 7.

To fix the balance we make 8 the new root, make c the right child move the old root (7) down to the left together with its left subtree A and finally move subtree B across and make it the new right child of 7.

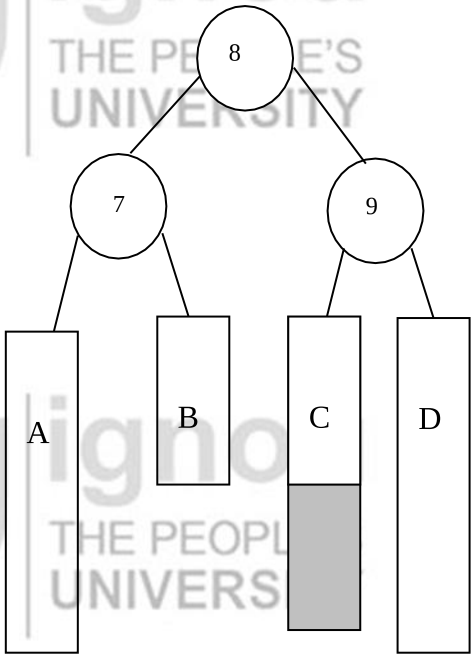
Example: (Double left rotation when a new node is inserted into the AVL tree (RL rotation)) (refer to *Figure 7.10 (a),(b),(c)*).



(a)



(b)



(c)

Figure 7.10: Double left rotation when a new node is inserted into the AVL tree

A node was inserted into the subtree C, making the tree off balance by 2 at the root. We first make a right rotation around the node 9, placing the C subtree into the left child of 9.

Then a left rotation around the root brings node 9 (together with its children) up a level and subtree A is pushed down a level (together with node 7). As a result we get correct AVL tree equal balance.

An AVL tree can be represented by the following structure:

```
struct avl  {
            struct node *left;
            int info;
            int bf;
            struct node *right;
};
```

bf is the balance factor, info is the value in the node.

7.3.4 Applications of AVL Trees

AVL trees are applied in the following situations:

- There are few insertion and deletion operations
- Short search time is needed
- Input data is sorted or nearly sorted

AVL tree structures can be used in situations which require fast searching. But, the large cost of rebalancing may limit the usefulness.

Consider the following:

1. A classic problem in computer science is how to store information dynamically so as to allow for quick look up. This searching problem arises often in dictionaries, telephone directory, symbol tables for compilers and while storing business records etc. The records are stored in a balanced binary tree, based on the keys (alphabetical or numerical) order. The balanced nature of the tree limits its height to O (log n), where *n* is the number of inserted records.

2. AVL trees are very fast on searches and replacements. But, have a moderately high cost for addition and deletion. If application does a lot more searches and replacements than it does addition and deletions, the balanced (AVL) binary tree is a good choice for a data structure.

3. AVL tree also has applications in file systems.

☞ **Check Your Progress 2**

1) Define the structure of an AVL tree.

……………………………………………………………………………

……………………………………………………………………………

7.4 B – TREES

B-trees are special m–ary balanced trees used in databases because their structure allows records to be inserted, deleted and retrieved with guaranteed worst case performance.

A B-Tree is a specialised multiway tree. In a B-Tree each node may contain a large number of keys. The number of subtrees of each node may also be large. A B-Tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that height of the tree is relatively small.

This means that only a small number of nodes must be read from disk to retrieve an item.

A B-Tree of order m is multiway search tree of order m such that

- All leaves are on the bottom level
- All internal nodes (except root node) have atleast m/2 (non empty) children
- The root node can have as few as 2 children if it is an internal node and can have no children if the root node is a leaf node
- Each leaf node must contain atleast (m/2) – 1 keys.

The following is the structure for a B-tree :struct btree

```
{     int count;                  // number of keys stored in the current node
      item_type key[3];           // array to hold 3 keys
      long branch [4];              // array of fake pointers (records numbers)
};
```

Figure 7.11 depicts a B-tree of order 5.

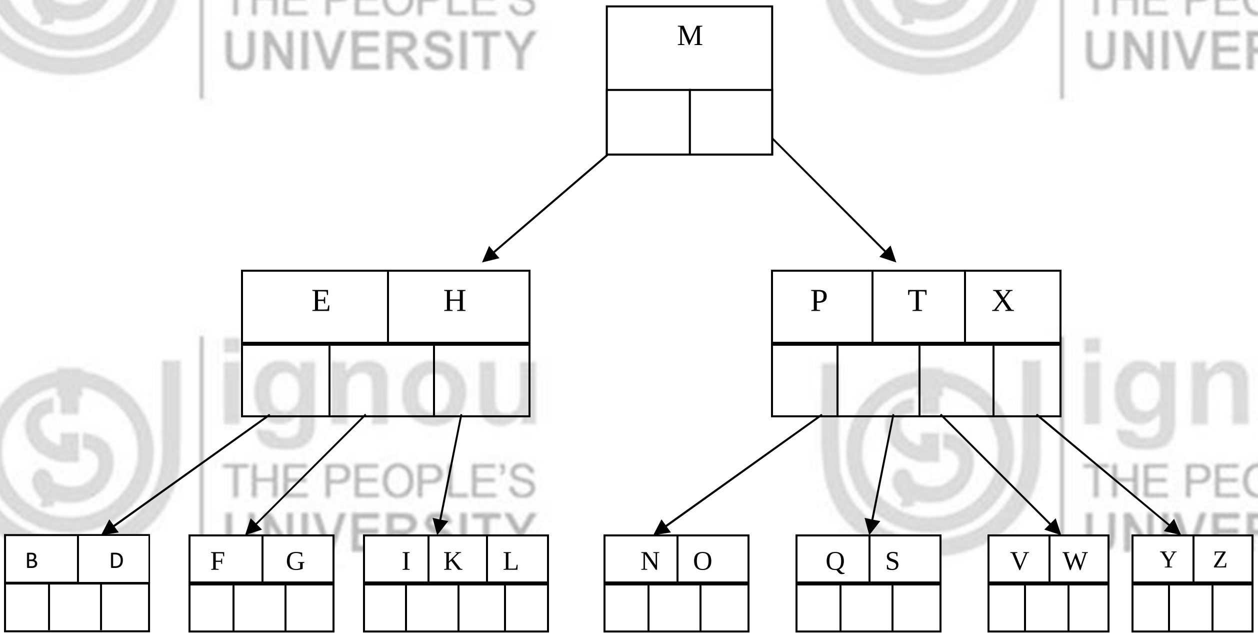


Figure 7.11: A B-tree of order 5

7.4.1 Operations on B-Trees

The following are various operations that can be performed on B-Trees:

- Search
- Create
- Insert

B-Tree strives to minimize disk access and the nodes are usually stored on diskAll the

nodes are assumed to be stored in secondary storage rather than primary storage. All references to a given node are preceded by a read operation. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with write operation.

The following is the algorithm for searching a B-tree:

B-Tree Search (x, k)

```
i < - 1
while i < = n [x] and k > key$_i$[x]
        do i ← i + 1
if i < = n [x] and k = key$_1$ [x]
        then return (x, i)
if leaf [x]
        then return NIL
else Disk – Read (c$_i$[x])
        return B – Tree Search (C$_i$[x], k)
```

The search operation is similar to binary tree. Instead of choosing between a left and right child as in binary tree, a B-tree search must make an n-way choice.

The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to desired value, the child pointer to the immediate left to that value is followed.

The exact running time of search operation depends upon the height of the tree.The following is the algorithm for the creation of a B-tree:

B-Tree Create (T)

```
        x ← Allocate-Node ( )
        Leaf [x] ← Truen
        [x] ← 0
        Disk-write (x)
        root [T] ← x
```

The above mentioned algorithm creates an empty B-tree by allocating a new root that has no keys and is a leaf node.

The following is the algorithm for insertion into a B-tree:

B-Tree Insert (T,K)

```
r ← root (T)
if n[r] = 2t – 1
   then S ← Allocate-Node ( )
        root[T] ← S
        leaf [S] ← FALSE
        n[S] ← 0
        C1 ← r
        B–Tree-Split-Child (s, I, r)
        B–Tree-Insert-Non full (s, k)
        else
        B – Tree-Insert-Non full (r, k)
```

To perform an insertion on B-tree, the appropriate node for the key must be located. Next, the key must be inserted into the node.

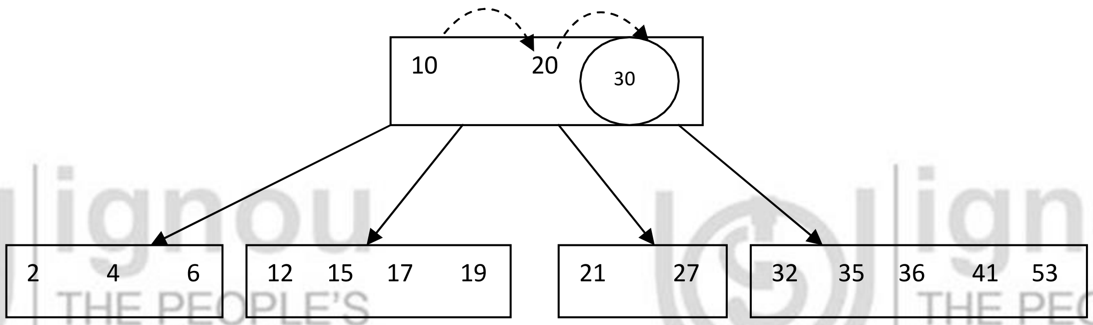
If the node is not full prior to the insertion, then no special action is required.

If node is full, then the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full. Else, another split operation is required.

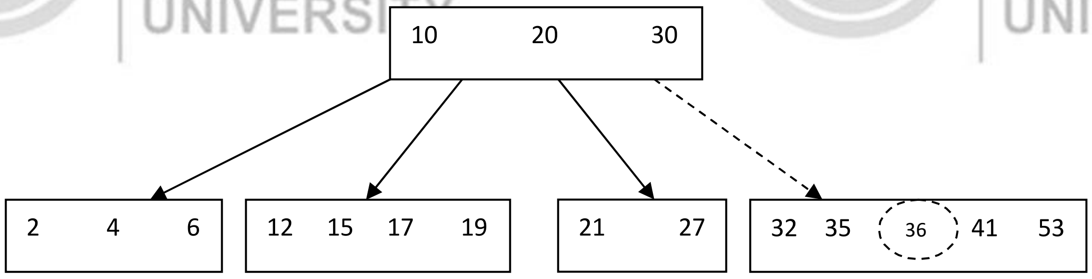
This process may repeat all the way up to the root and may require splitting the root node.

Example: Insertion of a key 33 into a B-Tree (w/split) (refer to *Figure 7.12)*

Step 1: Search first node for key nearest to 33. Key 30 was found.



Step 2: Node pointed by key 30, is searched for inserting 33. Node is shifted upwards.



Step 3: Key 33 is inserted between 32 and 35.

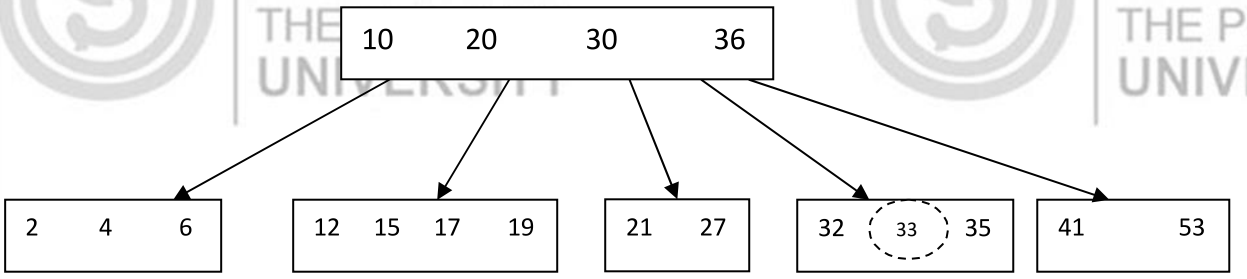
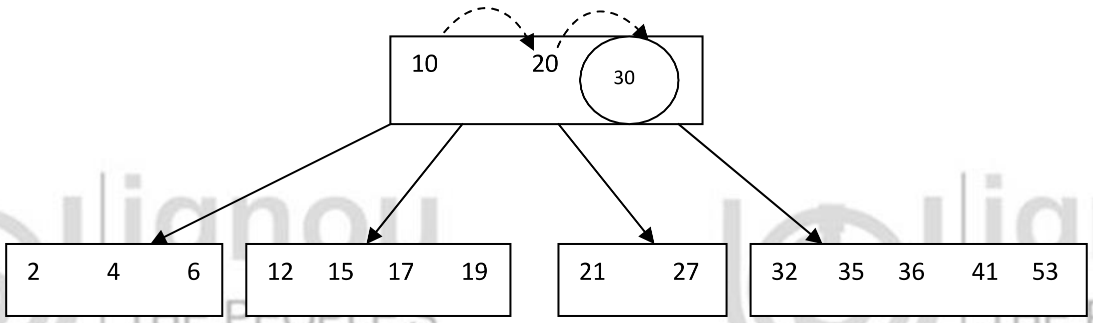


Figure 7.12 : A B-tree

Deletion of a key from B-tree is possible, but care must be taken to ensure that the properties of b-tree are maintained if the deletion reduces the number of keys in a node below the minimum degree of tree, this violation must be connected by combining several nodes and possibly reducing the height if the tree. If the key has children, the children must be rearranged.

Example (Searching of a B – Tree for key 21(refer to Figure 7.13))

Step 1: Search for key 21 in first node. 21 is between 20 and 30.



Step2 : Searching is conducted on the nodes connected by 30.

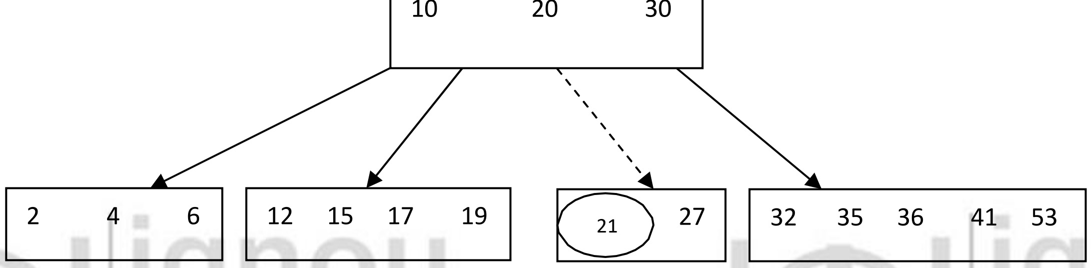


Figure 7.13 : A B-tree

7.4.2 Applications of B-trees

A database is a collection of data organised in a fashion that facilitates updation, retrieval and management of the data. Searching an unindexed database containing n keys will have a worst case running time of O (n). If the same data is indexed with a b-tree, then the same search operation will run in O(log n) time. Indexing large amounts of data can significantly improve search performance.

☞ Check Your Progress 3

1) Create a B – Tree of order 5 for the following:
CNGAHEKQMSWLTZDPRXYS

……………………………………………………………………………………....

……………………………………………………………………………………

2) Define a multiway tree of order m.

…………………………………………………………………………………

……………………………………………………………………………………

7.5 SPLAY TREES

Addition of new records in a Binary tree structure always occurs as leaf nodes, which are further away from the root node making their access slower. If this new record is to be accessed very frequently, then we cannot afford to spend much time in reaching it but would require it to be positioned close to the root node. This would call for readjustment or rebuilding of the tree to attain the desired shape. But, this process of rebuilding the tree every time as the preferences for the records change is tedious and time consuming. There must be some measure so that the tree adjusts itself automatically as the frequency of accessing the records changes. Such a self-adjusting tree is the Splay tree.

Splay trees are self-adjusting binary search trees in which every access for insertion or retrieval of a node, lifts that node all the way up to become the root, pushing the other nodes out of the way to make room for this new root of the modified tree. Hence, the frequently accessed nodes will frequently be lifted up and remain around the root position; while the most infrequently accessed nodes would move farther and farther away from the root.

This process of readjusting may at times create a highly imbalanced splay tree, wherein a single access may be extremely expensive. But over a long sequence of accesses, these expensive cases may be averaged out by the less expensive ones to produce excellent results over a long sequence of operations. The analytical tool used for this purpose is the Amortized algorithm analysis. This will be discussed in detail in the following sections.

7.5.1 Splaying Steps

Readjusting for tree modification calls for rotations in the binary search tree. Single rotations are possible in the left or right direction for moving a node to the root position. The task would be achieved this way, but the performance of the tree amortized over many accesses may not be good.

Instead, the key idea of splaying is to move the accessed node two levels up the tree at each step. Basic terminologies in this context are:

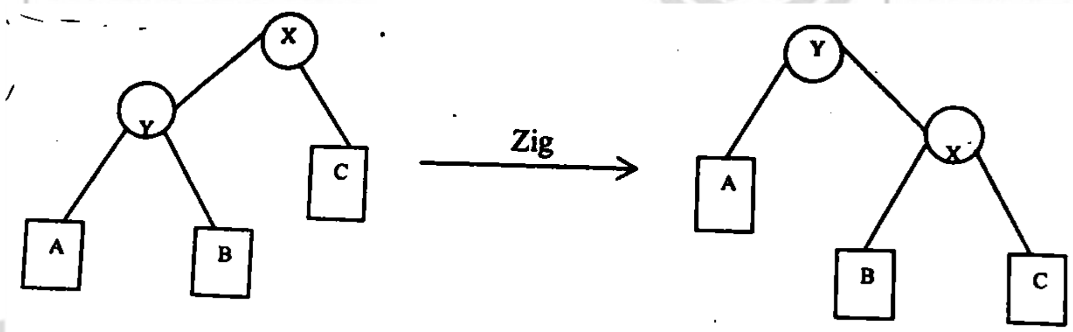
Zig: Movement of one step down the path to the left to fetch a node up. Zag: Movement of one step down the path to the right to fetch a node up.

With these two basic steps, the possible splay rotations are:Zig-Zig: Movement of two steps down to the left.

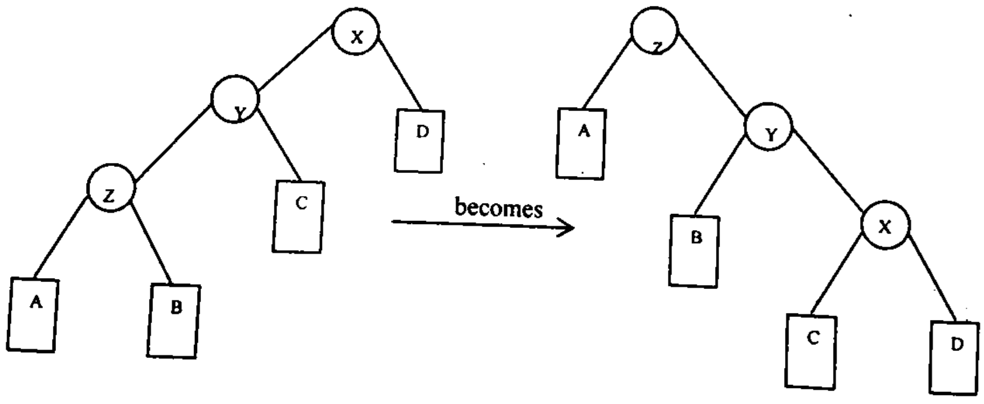
Zag-Zag: Movement of two steps down to the right.Zig-Zag: Movement of one step left and then right. Zag-Zig: Movement of one step right and then left.

Figure 7.14 depicts the splay rotations.

Zig:



Zig-Zig:



Zig-Zag:

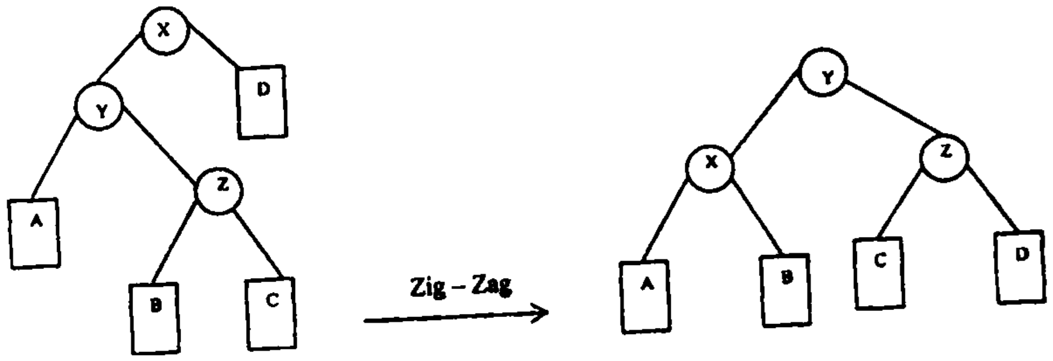


Figure 7.14: Splay rotations

Splaying may be top-down or bottom-up. In bottom-up splaying, splaying begins atthe accessed node, moving up the chain to the root. While in top-down splaying, splaying begins from the top while searching for the node to access. In the next section, we would be discussing the top-down splaying procedure:

As top-down splaying proceeds, the tree is split into three parts:

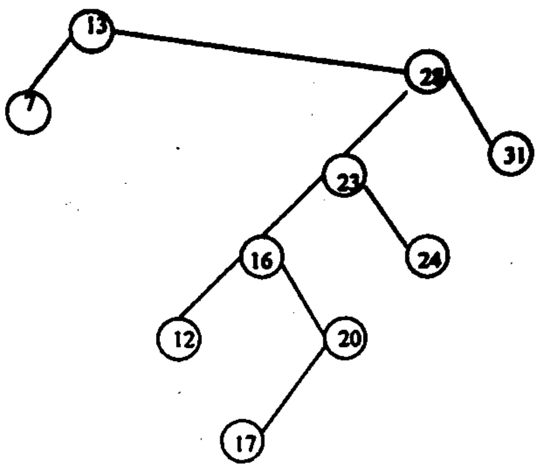
a) **<u>Central SubTree</u>**: This is initially the complete tree and may contain the target node. Search proceeds by comparison of the target value with the root and ends with the root of the central tree being the node containing the target if present ornull node if the target is not present.

b) **<u>Left SubTree</u>**: This is initially empty and is created as the central subtree is splayed. It consists of nodes with values less than the target being searched.

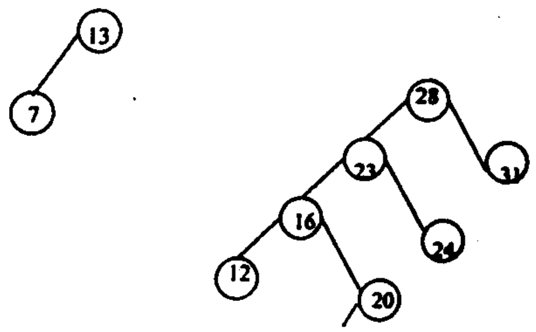
c) **<u>Right SubTree</u>**: This is also initially empty and is created similar to left subtree. It consists of nodes with values more than the target node.

Figure 7.15 depicts the splaying procedure with an example, attempting to splay at 20.

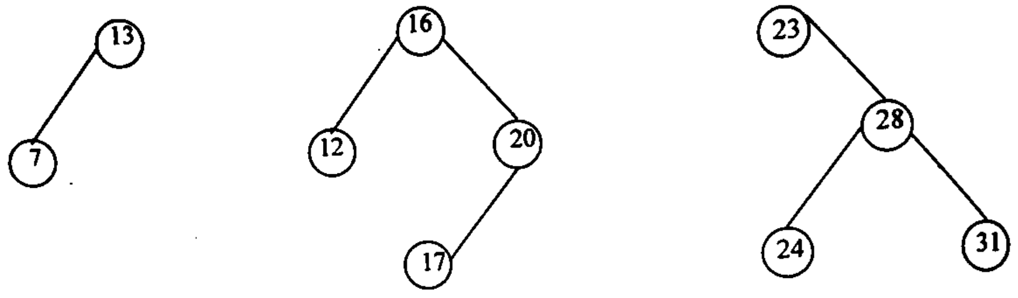
Initially,



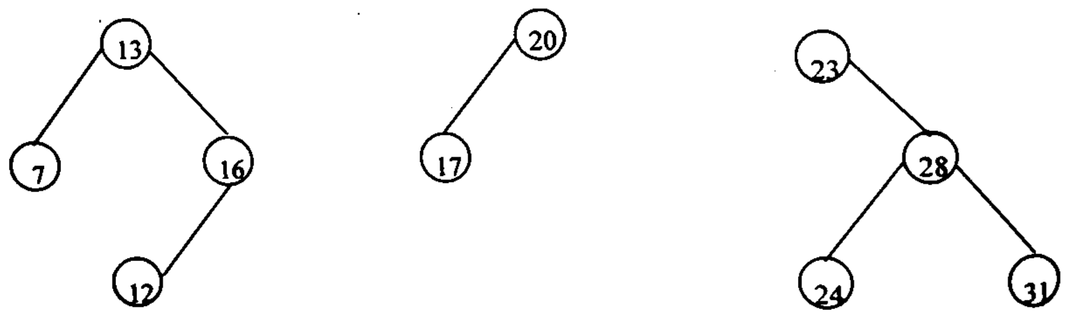
The first step is Zig-Zag:



The next step is Zig-Zig:



The next step is the terminal zig:



Finally, reassembling the three trees, we get:

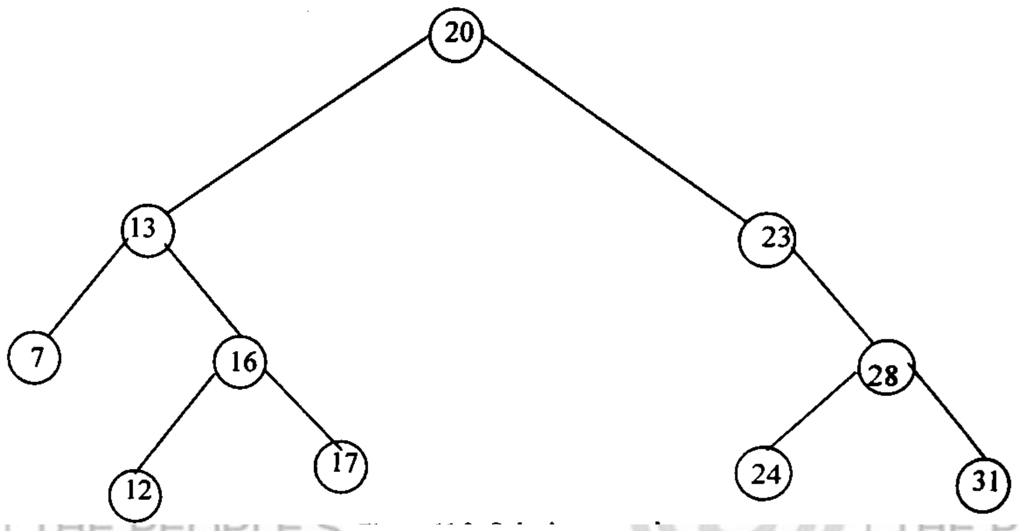


Figure 7.15: Splaying procedure

7.5.2 Splaying Algorithm

Insertion and deletion of a target key requires splaying of the tree. In case of insertion, the tree is splayed to find the target. If, target key is found, then we have a duplicate and the original value is maintained. But, if it is not found, then the target is inserted as the root.

In case of deletion, the target is searched by splaying the tree. Then, it is deleted from the root position and the remaining trees reassembled, if found.

Hence, splaying is used both for insertion and deletion. In the former case, to find the proper position for the target element and avoiding duplicity and in the latter case to bring the desired node to root position.

Splaying procedure

For splaying, three trees are maintained, the central, left and right subtrees. Initially, the central subtree is the complete tree and left and right subtrees are empty. The target key is compared to the root of the central subtree where the following two conditions are possible:

a) Target > Root: If target is greater than the root, then the search will be more to the right and in the process, the root and its left subtree are shifted to the left tree.

b) Target < Root: If the target is less than the root, then the search is shifted to theleft, moving the root and its right subtree to right tree.

We repeat the comparison process till either of the following conditions are satisfied:

a) Target is found: In this, insertion would create a duplicate node. Hence, originalnode is maintained. Deletion would lead to removing the root node.

b) Target is not found and we reach a null node: In this case, target is inserted inthe null node position.

Now, the tree is reassembled. For the target node, which is the new root of our tree,the largest node is the left subtree and is connected to its left child and the smallest node in the right subtree is connected as its right child.

Amortized Algorithm Analysis

In the amortized analysis, the time required to perform a set of operations is the average of all operations performed. Amortized analysis considers a long sequence of operations instead of just one and then gives a worst-case estimate. There are three different methods by which the amortized cost can be calculated and can be differentiated from the actual cost. The three methods, namely, are:

- Aggregate analysis: It finds the average cost of each operation. That is, T(n)/n. The amortized cost is same for all operations.
- Accounting method: The amortized cost is different for all operations and charges a credit as prepaid credit on some operations.
- Potential method: It also has different amortized cost for each operation andcharges a credit as the potential energy to other operations.

There are different operations such as stack operations (push, pop, multipop) and an increment which can be considered as examples to examine the above three methods.

Every operation on a splay tree and all splay tree operations take O(log n) amortized time.

7.6 RED-BLACK TREES

A Red-Black Tree (RBT) is a type of Binary Search tree with one extra bit of storage per node, i.e. its color which can either be red or black. Now the nodes can have anyof the color (red, black) from root to a leaf node. These trees are such that they guarantee O(log n) time in the worst case for searching.

Each node of a red black tree contains the field color, key, left, right and p (parent). If a child or a parent node does not exist, then the pointer field of that node contains NULL value.

7.6.1 Properties of a Red-Black Tree

Any binary search tree should contain following properties to be called as a red-black tree.

1. Each node of a tree should be either red or black.
2. The root node is always black.
3. If a node is red, then its children should be black.
4. For every node, all the paths from a node to its leaves contain the same numberof black nodes.

We define the number of black nodes on any path from but not including a node x down to a leaf, the black height of the node is denoted by bh (x).
Figure 7.16 depicts a Red-Black Tree.

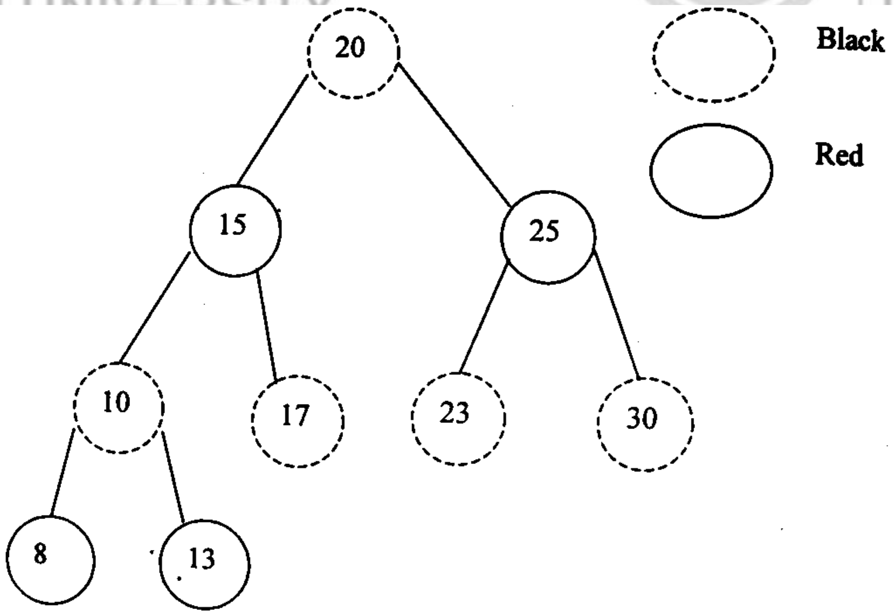


Figure 7.16: A Red-Black tree

Red-black trees contain two main operations, namely INSERT and DELETE. When the tree is modified, the result may violate red-black properties. To restore the tree properties, we must change the color of the nodes as well as the pointer structure. We can change the pointer structure by using a technique called rotation which preserves inorder key ordering. There are two kinds of rotations: left rotation and right rotation (refer to *Figures 7.17 and 7.18*).

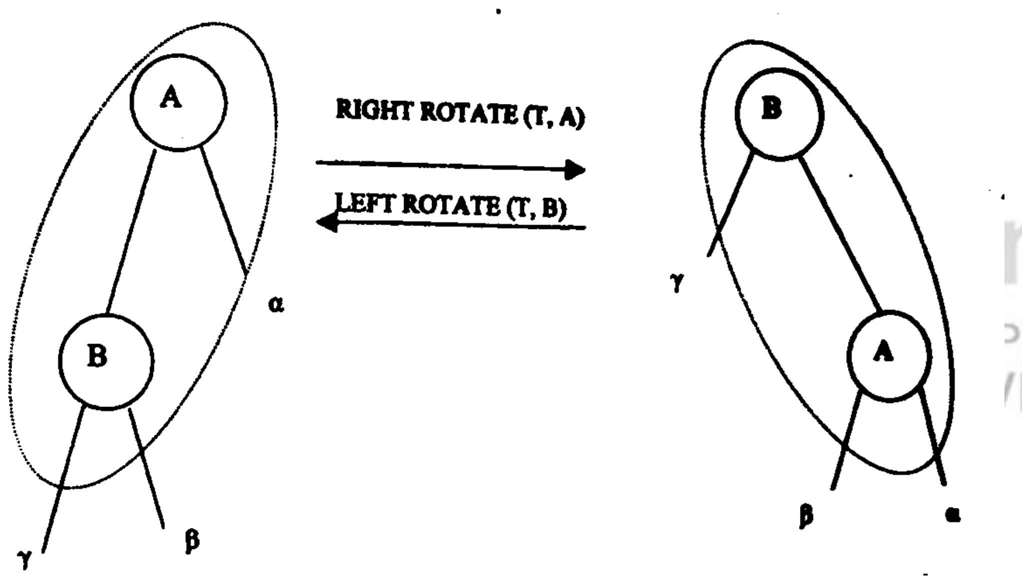


Figure 7.17: Left rotation **Figure 7.18: Right rotation**

When we do a left rotation on a node y, we assume that its right child x is non null. The left rotation makes x as the new root of the subtree with y as x's left child and x's left child as y's right child.

The same procedure is repeated vice versa for the right rotation.

7.6.2 Insertion into a Red-Black Tree

The insertion procedure in a red-black tree is similar to a binary search tree i.e., the insertion proceeds in a similar manner but after insertion of nodes x into the tree T, we color it red. In order to guarantee that the red-black properties are preserved, we then fix up the updated tree by changing the color of the nodes and performing rotations. Let us write the pseudo code for insertion.

The following are the two procedures followed for insertion into a Red-Black Tree:

Procedure 1: This is used to insert an element in a given Red-Black Tree. It involves the method of insertion used in binary search tree.

Procedure 2: Whenever the node is inserted in a tree, it is made red, and after insertion, there may be chances of loosing Red-Black Properties in a tree, and, so, some cases are to be considered inorder to retain those properties.

During the insertion procedure, the inserted node is always red. After inserting a node, it is necessary to notify that which of the red-black properties are violated. Letus now look at the execution of fix up. Let Z be the node which is to be inserted andis colored red. At the start of each iteration of the loop,

1. Node Z is red
2. If P(Z) is the root, then P(Z) is black
3. Now if any of the properties i.e. property 2 is violated if Z is the root and is red OR when property 4 is violated if both Z and P(Z) are red, then we consider 3 cases in the fix up algorithm. Let us now discuss those cases.

Case 1(Z's uncle y is red): This is executed when both parent of Z (P(Z)) and uncle of Z, i.e. y are red in color. So, we can maintain one of the property of Red-Black tree by making both P(Z) and y black and making point of P(Z) to be red, thereby maintaining one more property. Now, this while loop is repeated again until color of y is black.

Case 2 (Z's uncle is black and Z is the right child): So, make parent of Z to be Z itself and apply left rotation to newly obtained Z.

Case 3 (Z's uncle is black and Z is the left child): This case executes by making parent of Z as black and P(P(Z)) as red and then performing right rotation to it i.e., to(P(Z)).

The above 3 cases are also considered conversely when the parent of Z is to the right of its own parent. All the different cases can be seen through an example.Consider a red-black tree drawn below with a node z (17 inserted in it) (refer to *Figure 7.19*).

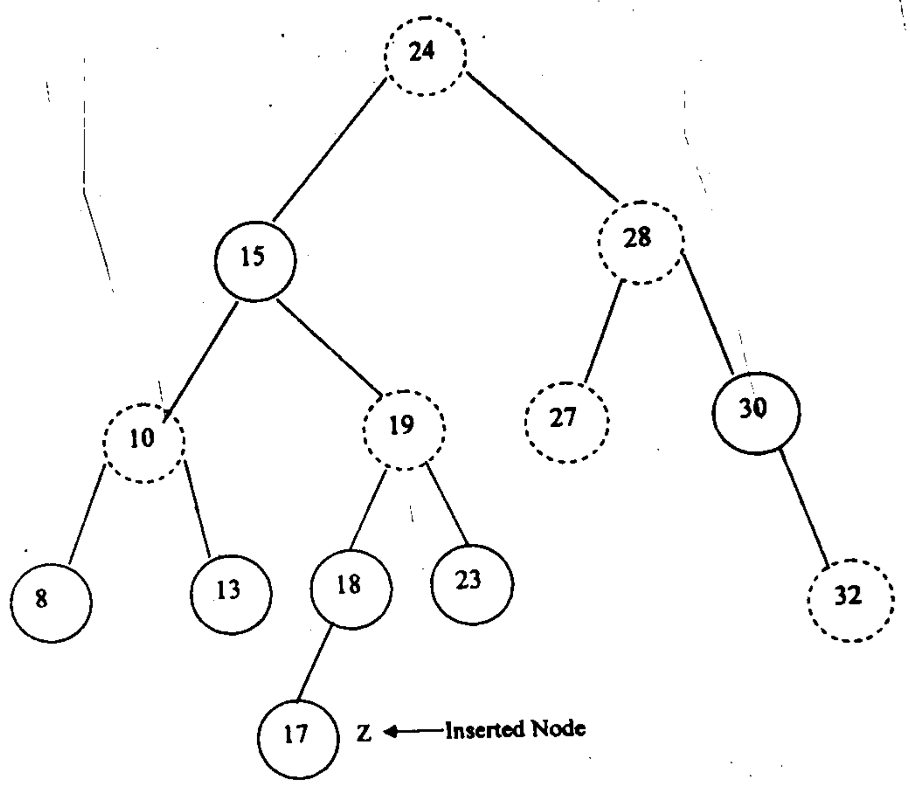


Figure 7.19: A Red-Black Tree after insertion of node 17

Before the execution of any case, we should first check the position of P(Z) i.e. ifit is towards left of its parent, then the above cases will be executed but, if it is towards the right of its parent, then the above 3 cases are considered conversely.

Now, it is seen that Z is towards the left of its parent (refer to *Figure 7.20*). So, the above cases will be executed and another node called y is assigned which is the uncle of Z and now cases to be executed are as follows:

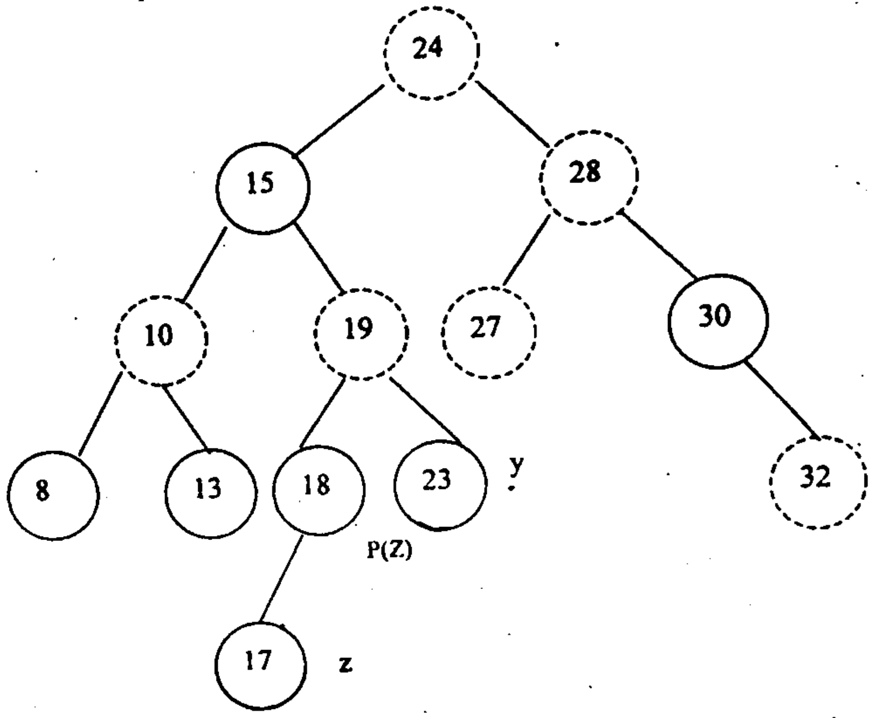


Figure 7.20: Z is to the left of it's parent

Case 1: Property 4 is violated as both z and parent(z) are red (refer to *Figure 7.21*).

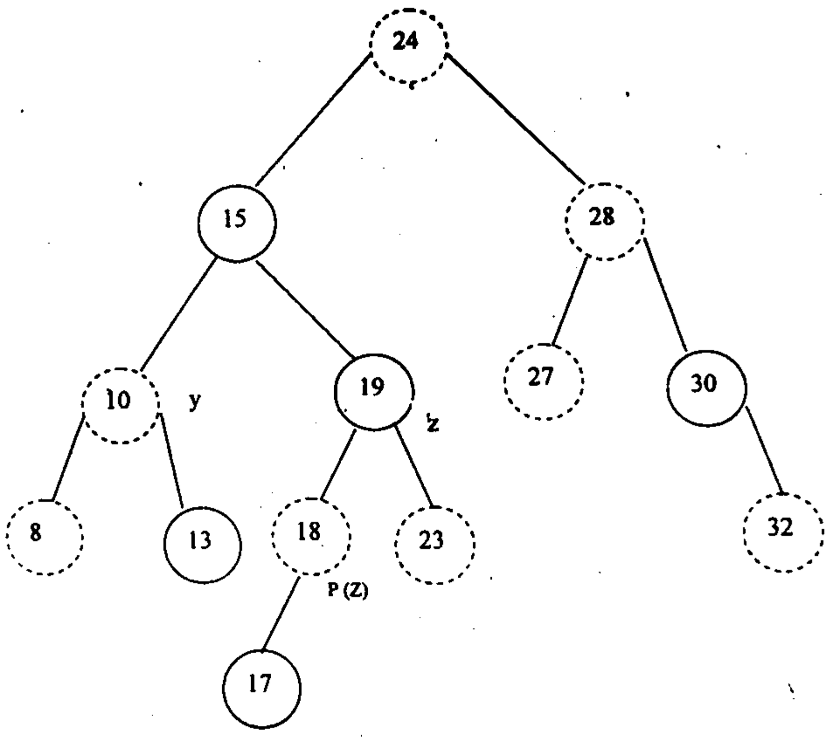


Figure 7.21: Both Z and P(Z) are red

Now, let us check to see which case is executed.

Case 2: The application of this case results in *Figure 7.22*.

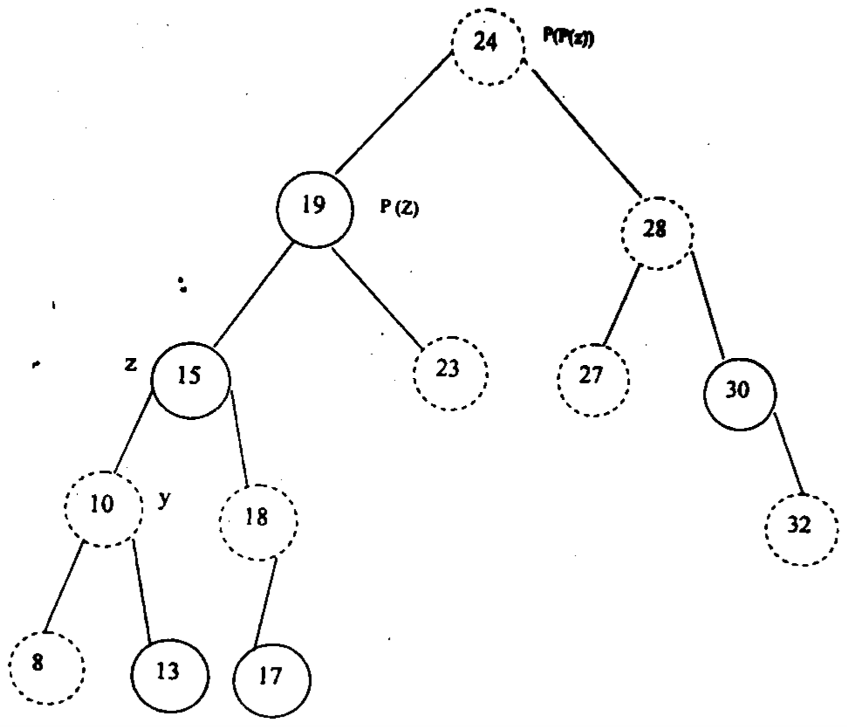


Figure 7.22: Result of application of case-2

Case 3: The application of this case results in *Figure 7.23*.

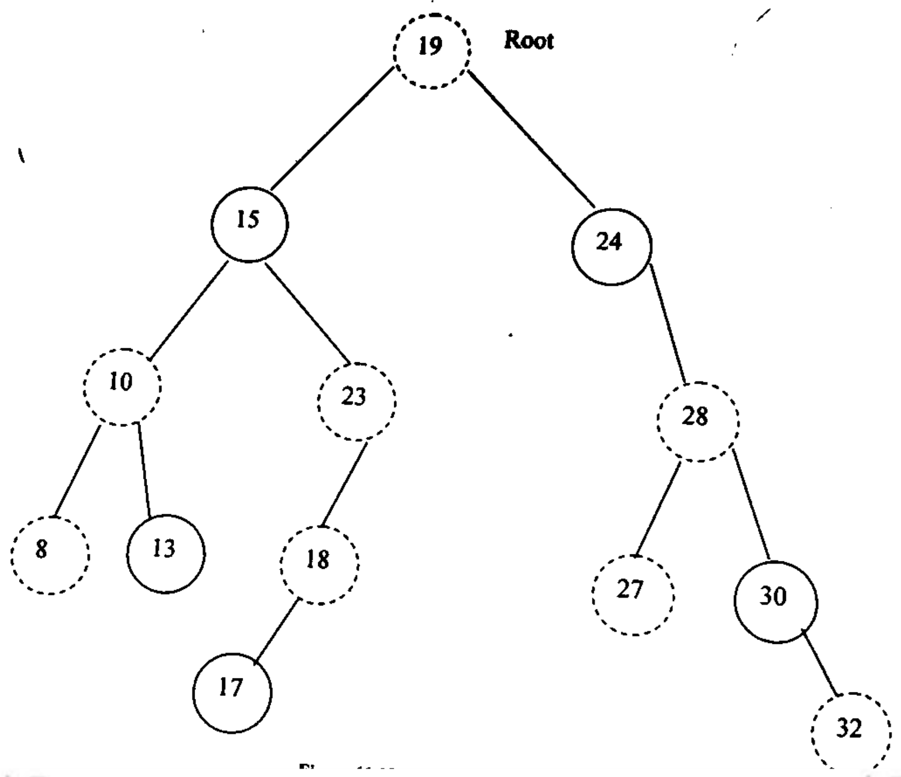


Figure 7.23: Result of application of case-3

Finally, it resulted in a perfect Red-Black Tree (*Figure 7.23*).

7.6.3 Deletion from a Red-Black Tree

Deletion in a RBT uses two main procedures, namely,

Procedure 1: This is used to delete an element in a given Red-Black Tree. It involves the method of deletion used in binary search tree.

Procedure 2: Whenever the node is deleted from a tree, and after deletion, there may be chances of losing Red-Black Properties in a tree and so, some cases are to be considered in order to retain those properties.

This procedure is called only when the successor of the node to be deleted is Black, but if y is red, the red- black properties still hold and for the following reasons:

- No red nodes have been made adjacent
- No black heights in the tree have changed
- y could not have been the root

Now, the node (say x) which takes the position of the deleted node (say z) will be called in procedure 2. Now, this procedure starts with a loop to make the extra black up to the tree until

- X points to a black node
- Rotations to be performed and recoloring to be done
- X is a pointer to the root in which the extra black can be easily removed

This while loop will be executed until x becomes root and its color is red. Here, a new node (say w) is taken which is the sibling of x.

There are four cases which we will be considering separately as follows:

Case 1: If color of w's sibling of x is red

Since W must have black children, we can change the colors of w and p (x) and then left rotate p (x) and the new value of w to be the right node of parent of x. Now, the conditions are satisfied and we switch over to case 2, 3 and 4.

Case 2: If the color of w is black and both its children are also black.

Since w is black, we make w to be red leaving x with only one black and assign parent (x) to be the new value of x. Now, the condition will be again checked, i.e. x = left (p(x)).

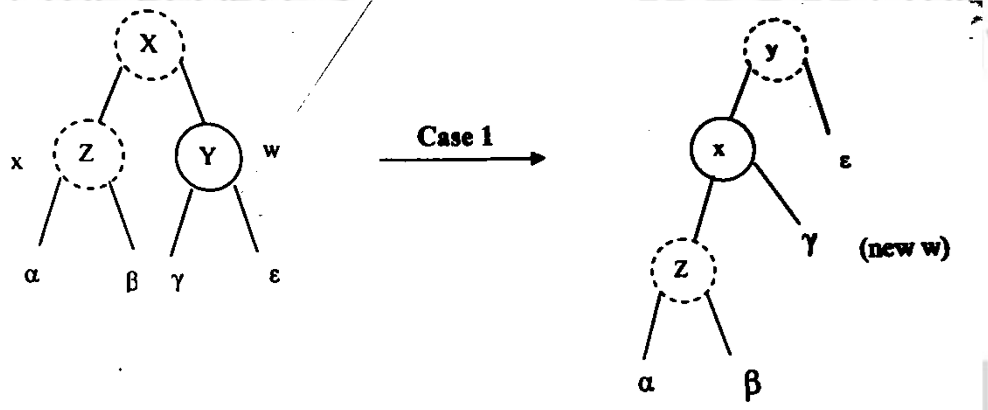


Figure 7.24: Application of case-1

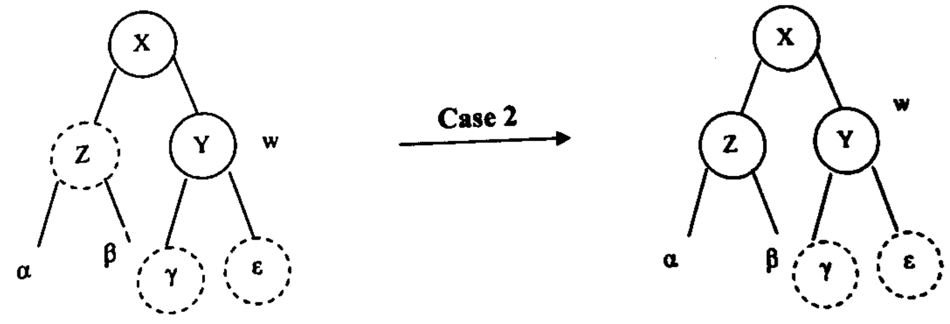


Figure 7.25: Application of case-2

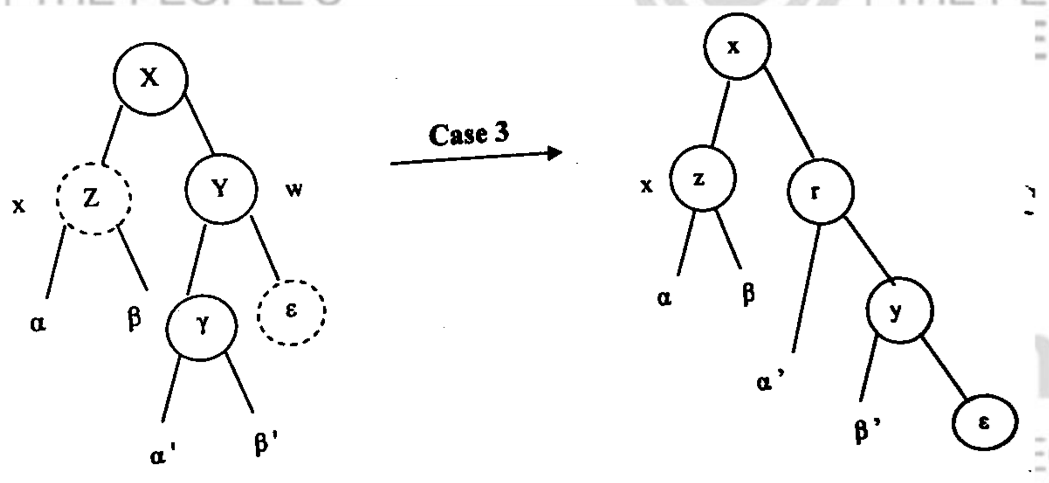


Figure 7.26: Application of case-3

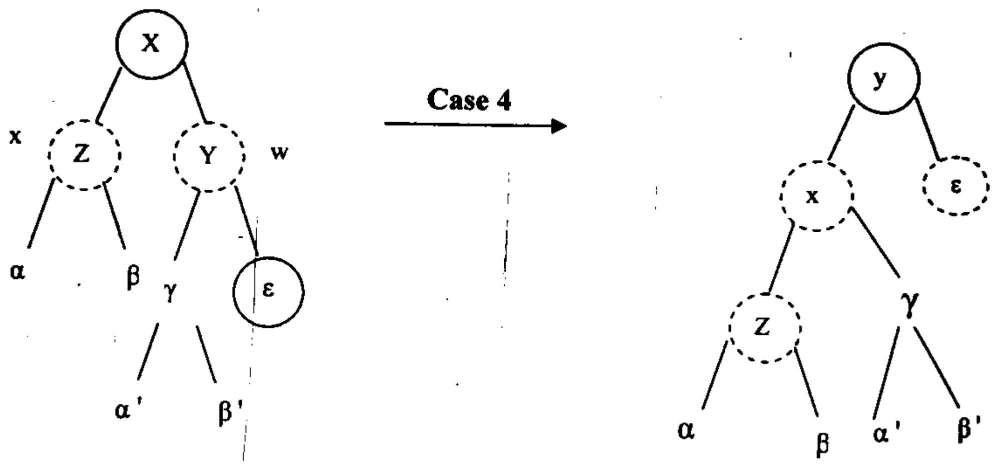


Figure7.27: Application of case-4

Case 3: If the color of w is black, but its left child is red and w's right child is black.

After entering case-3, we change the color of left child of w to black and that of w to be red and then perform right rotation on w without violating any of the black properties. The new sibling w of x is now a black node with a red right child and thus case 4 is obtained.

Case 4: When w is black and w's right child is red.

This can be done by making some color changes and performing a left rotation on p(x). We can remove the extra black on x, making it single black. Setting x to be the root causes the while loop to terminate.

Note: In the above *Figures 7.24, 7.25, 7.26* and *7.27* , α, α', β, β', γ, ε are assumed to be either red or black depending upon the situation.

7.7 AA-TREES

Red-Black trees have introduced a new property in the binary search tree, i.e., an extra property of color (red, black). But, as these trees grow, in their operations like insertion, deletion, it becomes difficult to retain all the properties, especially in case of deletion. Thus, a new type of binary search tree can be described which has no property of having a color, but has a new property introduced based on the color which is the information for the new. This information of the level of a node is stored in a small integer (may be 8 bits). Now, AA-trees are defined in terms of level of each node instead of storing a color bit with each node. A red-black tree used to have various conditions to be satisfied regarding its color and AA-trees have also been designed in such a way that it should satisfy certain conditions regarding its new property, i.e., level.

The level of a node will be as follows:

1. Same of its parent, if the node is red.
2. One if the node is a leaf.
3. Level will be one less than the level of its parent, if the node is black.

Any red-black tree can be converted into an AA-tree by translating its color structure to levels such that left child is always one level lower than its parent and right child is always same or at one level lower than its parent. When the right child is at same level to its parent, then a horizontal link is established between them. Thus, we conclude that it is necessary that horizontal links are always at the right side and that there may not be two consecutive links. Taking into consideration of all the above properties, we show a AA-tree as follows (refer to *Figure 7.28*).

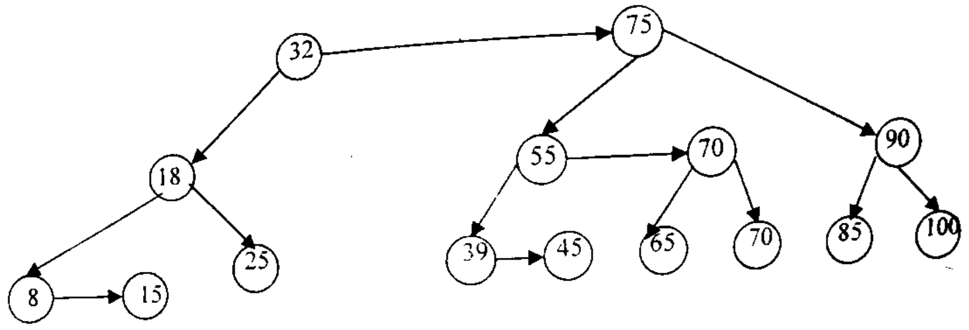


Figure 7.28: AA-tree

After having a look at the AA-tree above, we now look at different operations that can be performed at such trees.

The following are various operations on a AA-tree:

1. Searching: Searching is done by using an algorithm that is similar to the searchalgorithm of a binary search tree.
2. Insertion: The insertion procedure always start from the bottom level. But,while performing this function, either of the two problems can occur:

(a) Two consecutive horizontal links (right side)
(b) Left horizontal link.

While studying the properties of AA-tree, we said that conditions (a) and (b) should not be satisfied. Thus, in order to remove conditions (a) and (b), we use two new functions namely skew( ) and split( ) based on the rotations of the node, so that all the properties of AA-trees are retained.

The condition that (a) two consecutive horizontal links in an AA-tree can be removed by a left rotation by split( ) whereas the condition (b) can be removed by right rotations through function show( ). Either of these functions can remove these condition, but can also arise the other condition. Let us demonstrate it with an example. Suppose, in the AA-tree of *Figure 7.28*, we have to insert node 50.

According to the condition, the node 50 will be inserted at the bottom level in such a way that it satisfies Binary Search tree property also (refer to *Figure 7.29*).

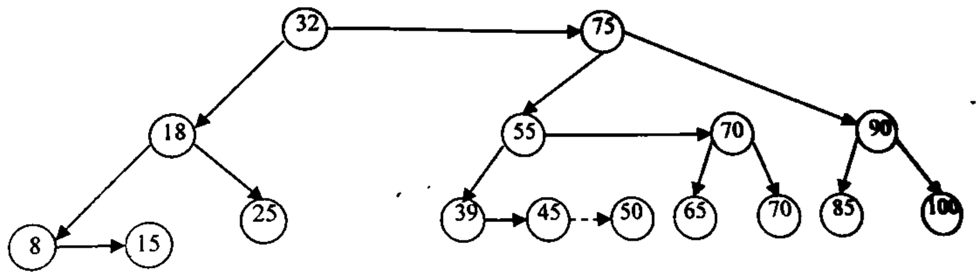


Figure7.29: After inserting node 50

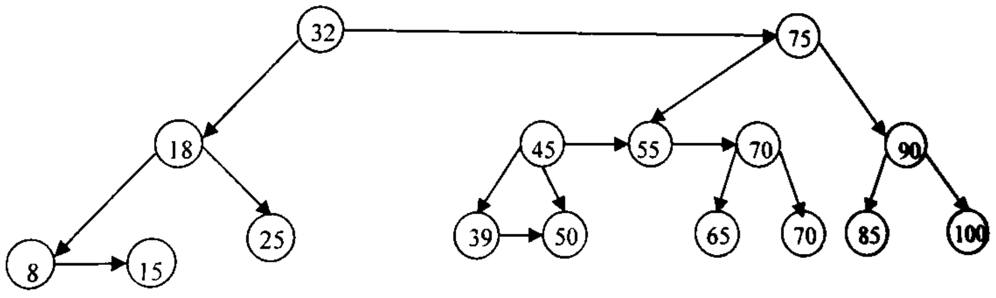


Figure 7.30: Split at node 39(left rotation)

Now, we should be aware as to how this left rotation is performed. Remember, that rotation is introduced in Red-black tree and these rotations (left and right) are the same as we performed in a Red-Black tree. Now, again split ( ) has removed its condition but has created skew conditions (refer to *Figure 7.30*). So, skew ( ) function will now be called again and again until a complete AA-tree with a no falsecondition is obtained.

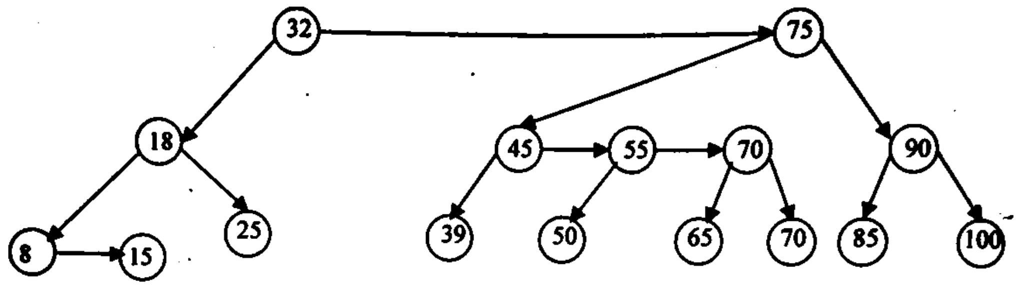


Figure 7.31: Skew at 55 (right rotation)

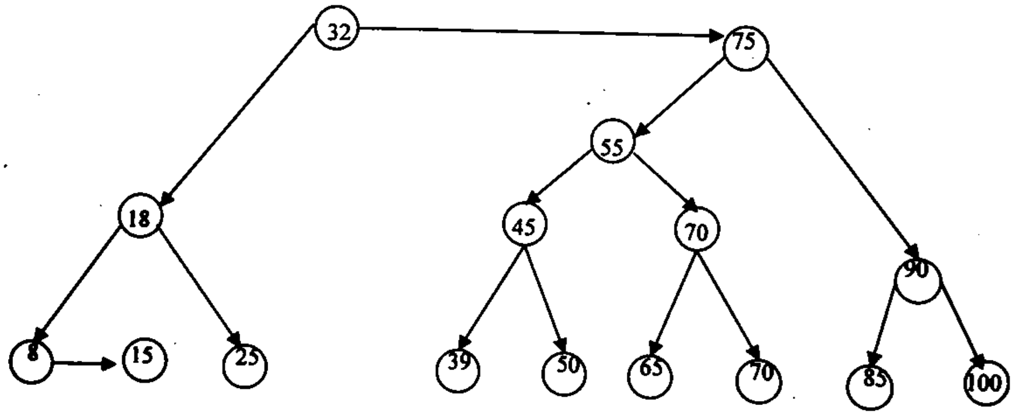


Figure7.32: Split at 45

A skew problem arises because node 90 is two-level lower than its parent 75 and so in order to avoid this, we call skew / split function again.

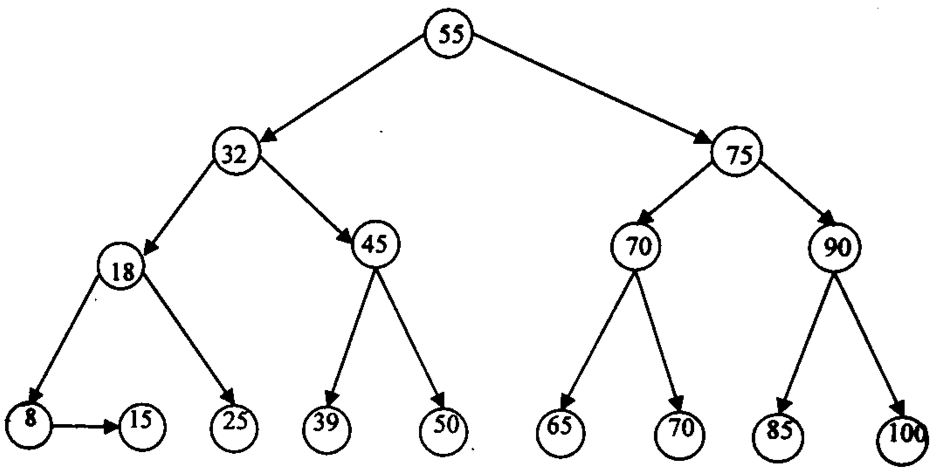


Figure: 7.33 : The Final AA-tree

Thus, introducing horizontal left links, in order to avoid left horizontal links and making them right horizontal links, we make 3 calls to skew and then 2 calls to split to remove consecutive horizontal links (refer to *Figures 7.31, 7.32* and *7.33*).

A Treap is another type of Binary Search tree and has one property different from other types of trees. Each node in the tree stores an item, a left and right pointer and a priority that is randomly assigned when the node is created. While assigning the priority, it is necessary that the heap order priority should be maintained: node's priority should be at least as large as its parent's. A treap is both binary search tree with respect to node elements and a heap with respect to node priorities.

7.8 SUMMARY

In this unit, we discussed Binary Search Trees, AVL trees and B-trees.

The striking feature of Binary Search Trees is that all the elements of the left subtree of the root will be less than those of the right subtree. The same rule is applicable for all the subtrees in a BST. An AVL tree is a Height balanced tree. The heights of left and right subtrees of root of an AVL tree differ by 1. The same rule is applicable for all the subtrees of the AVL tree. A B-tree is a m-ary binary tree. There can be multiple elements in each node of a B-tree. B-trees are used extensively to insert , delete and retrieve records from the databases.

7.9 SOLUTIONS/ANSWERS

Check Your Progress 1

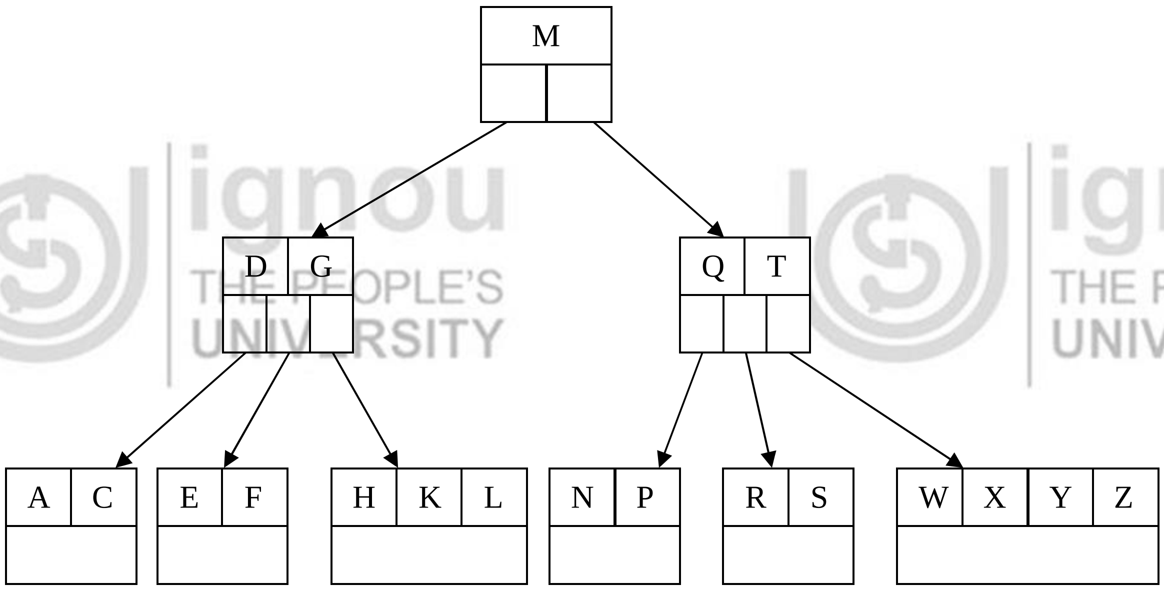
1) preorder, postorder and inorder
2) The major feature of a Binary Search Tree is that all the elements whose valuesis less than the root reside in the nodes of left subtree of the root and all the elements whose values are larger than the root reside in the nodes of right subtree of the root. The same rule is applicable to all the left and right subtrees of a BST.

Check Your Progress 2

1) The following is the structure of an AVL tree:struct avl {
struct node *left;
int info;int bf;
struct node *right;
};

Check Your Progress 3

1)



2) A multiway tree of order n is an ordered tree where each node has at most m children. For each node, if k is the actual no. of children in the node, then k-1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is multiway search tree of order m.

7.10 FURTHER READINGS

1. *Data Structures using C and C ++* by Yedidyah Hangsam, Moshe J.Augenstein and Aaron M. Tanenbaum, PHI Publications.

2. *Fundamentals of Data Structures in C* by R.B. Patel, PHI Publications.

Reference Websites

http:// www.cs.umbc.edu http://www.fredosaurus.com