UNIT 7 ADVANCED TREES

Structure **Page Nos.**

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 the ordered 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 parent node.
b) The key in the right child of a node (if exists) is greater than the key in its parent 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 search trees:

- 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*

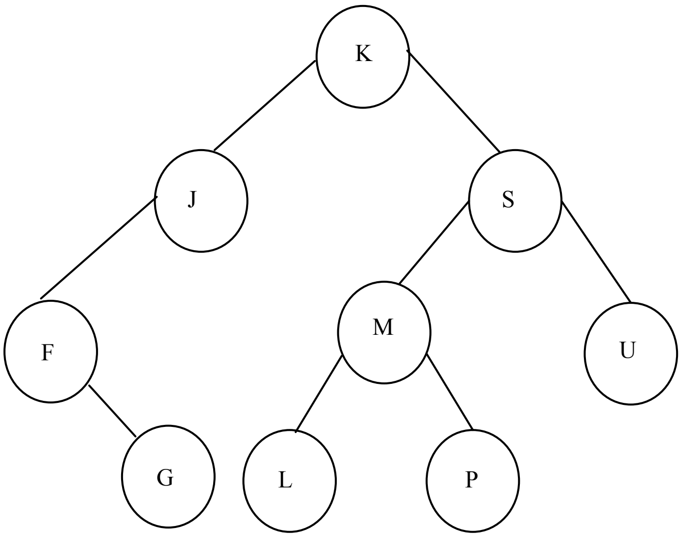


Figure 7.1: A Binary Search Tree(BST)

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 a binary tree structure.

Insertion must maintain the order of the tree. The value to the left of a given node must 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.

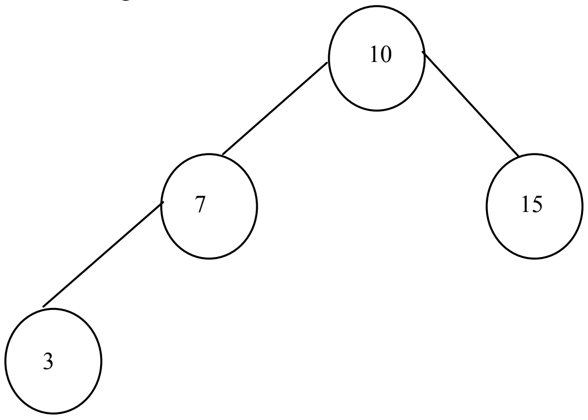


Figure 7.2: A non-empty

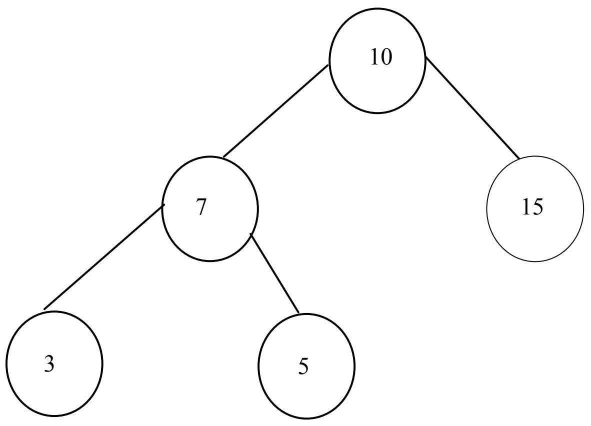


Figure 7.3: Figure 7.2 after insertion of 5

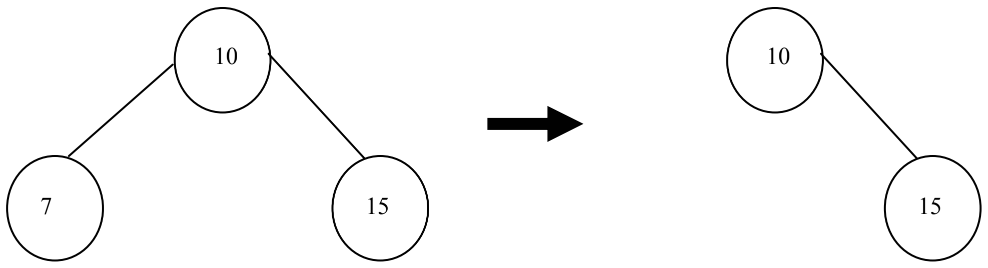
7.2.3 Deletion of a node from a Binary Search Tree

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

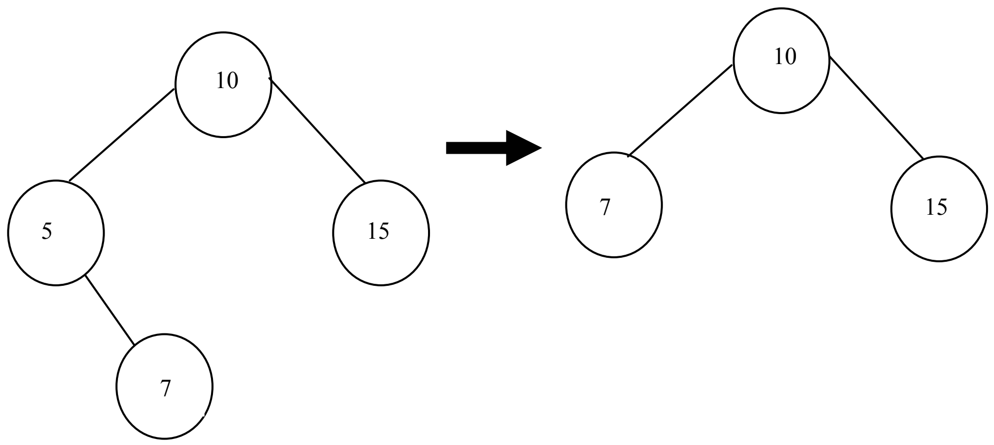
- If the node to be deleted has no sons, then it may be deleted without further adjustment to the tree.
- If the node to be deleted has only one subtree, then its only son can be moved up to take its place.
- The node *p* to be deleted has two subtrees, then its inorder successor *s* must take its place. 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 order of 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.

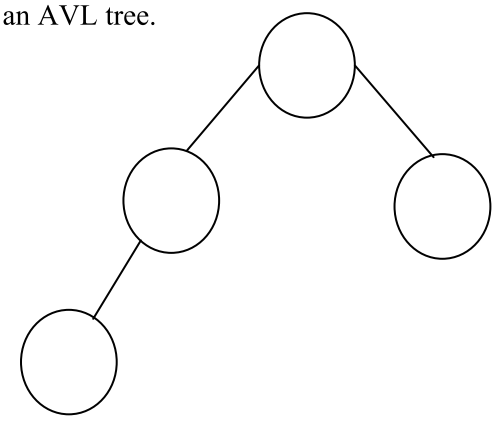


Figure 7.4 : Balance requirement for an AVL tree: the left and right subtree differ by at most one in height

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 ordinary binary 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 several extra 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 using various 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*).

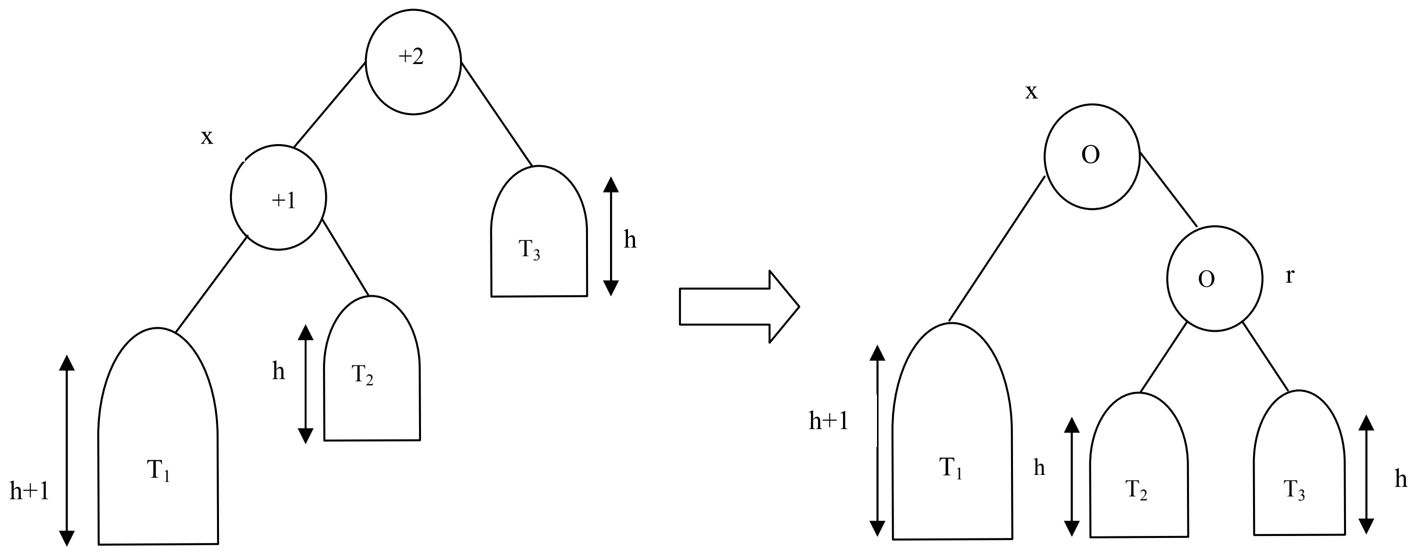


Figure 7.5: LL Rotation

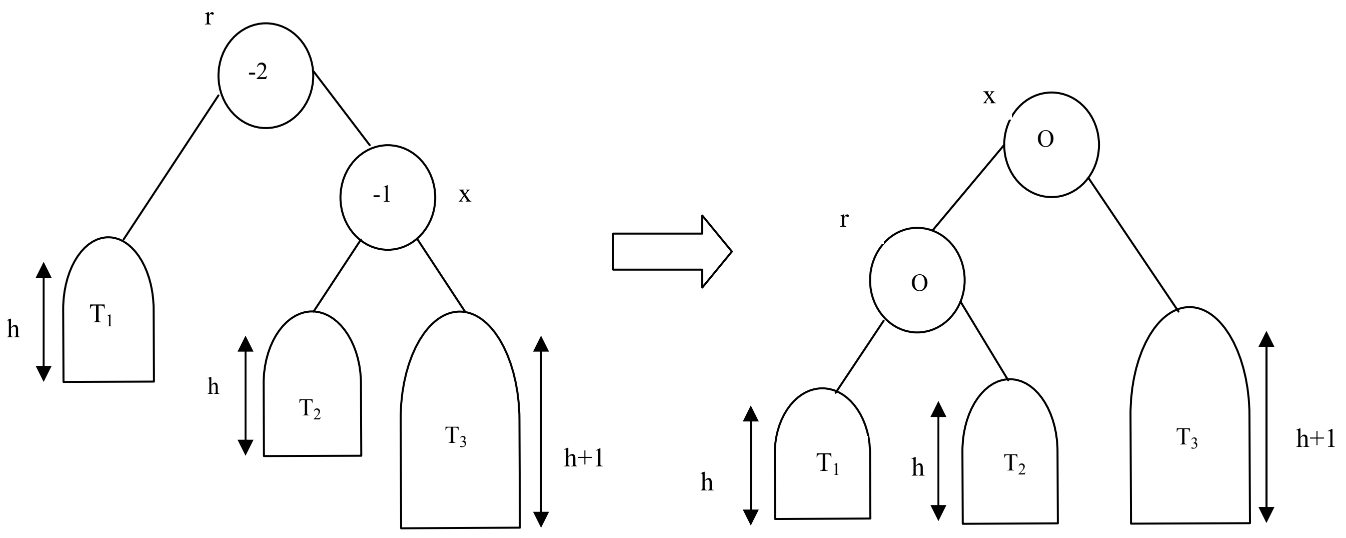


Figure 7.6: RR Rotation

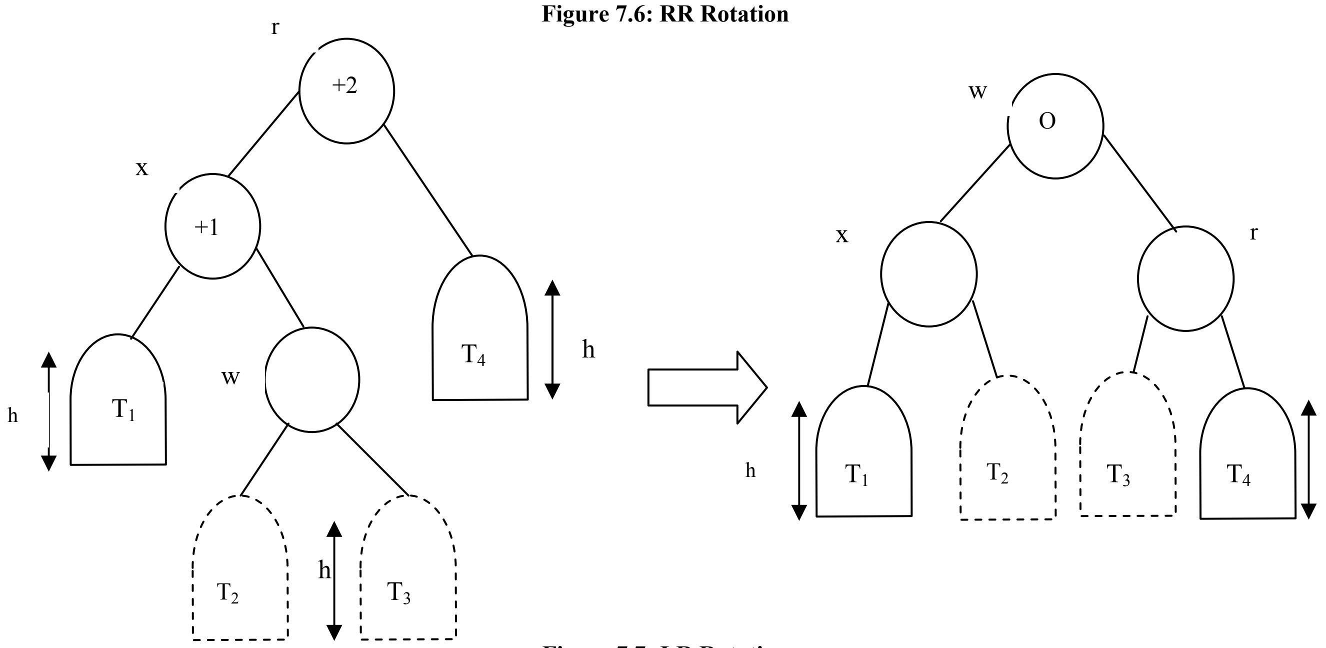


Figure 7.7: LR Rotation

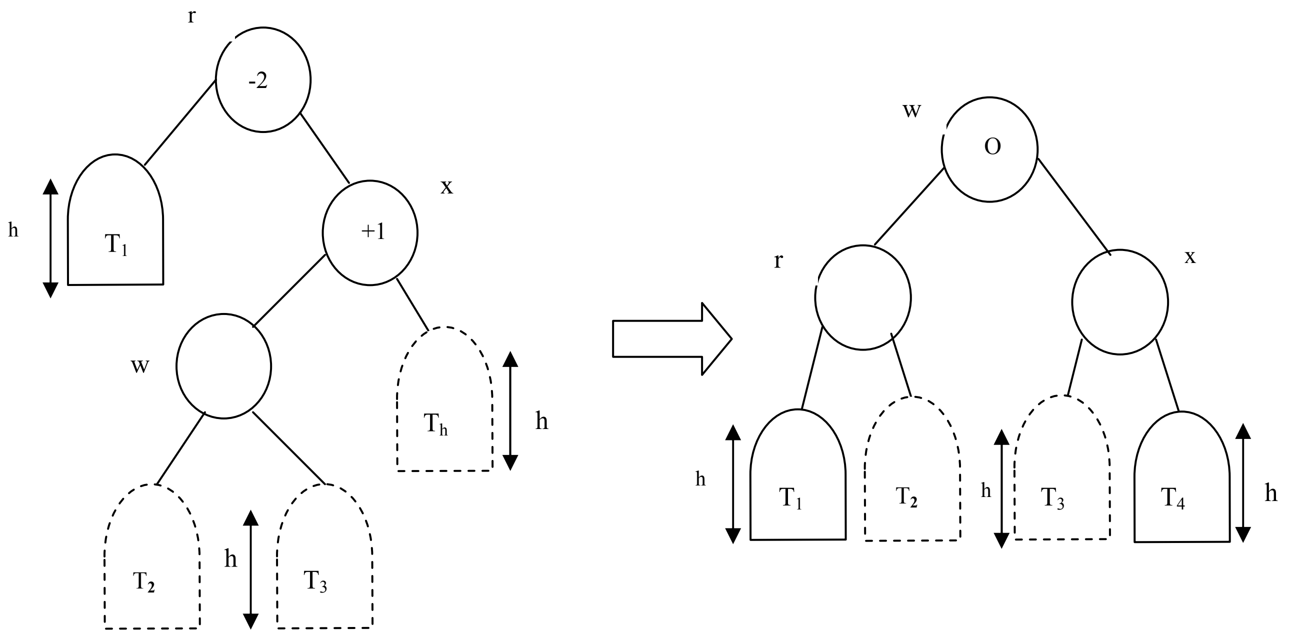


Figure 7.8: RL Rotation

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

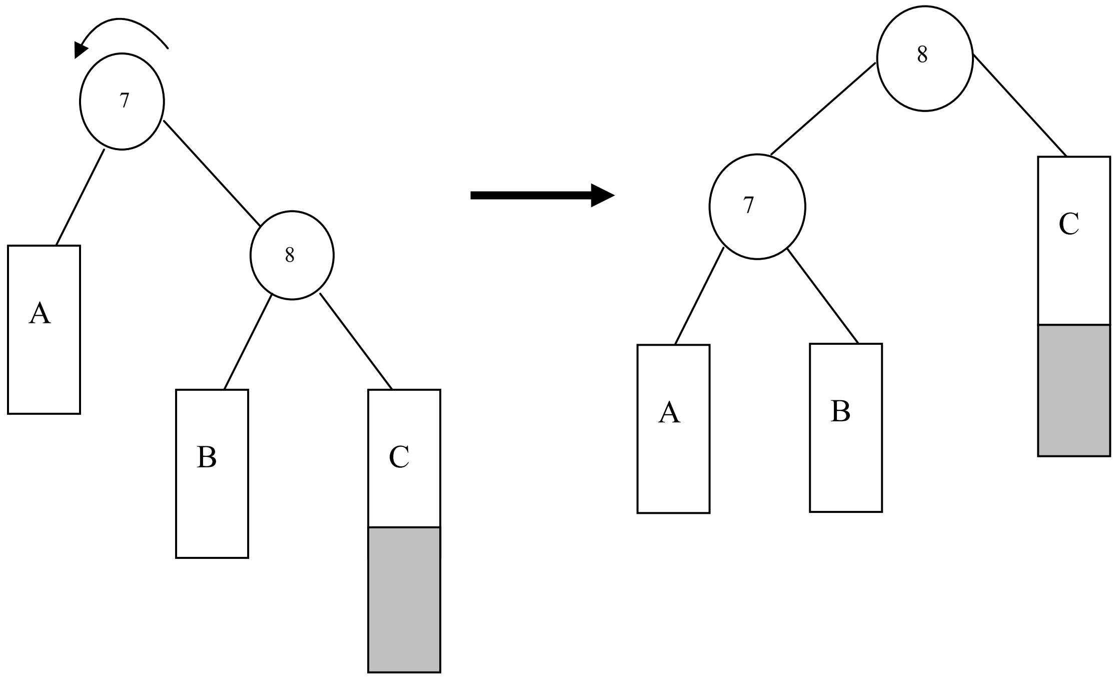


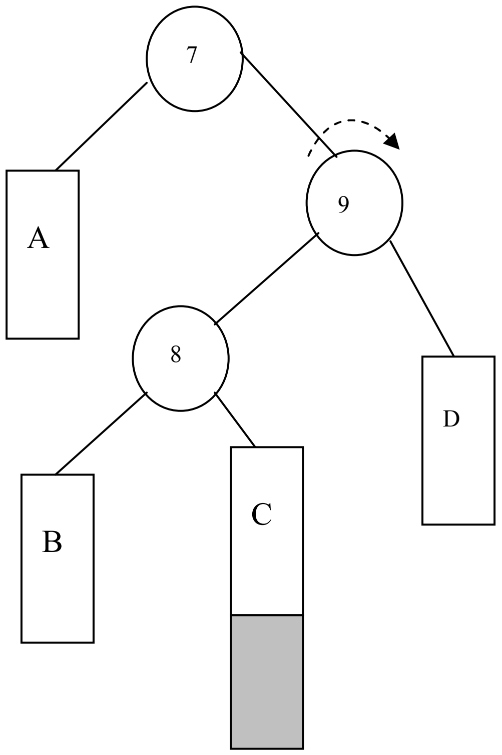
Figure 7.9: LL Rotation

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)

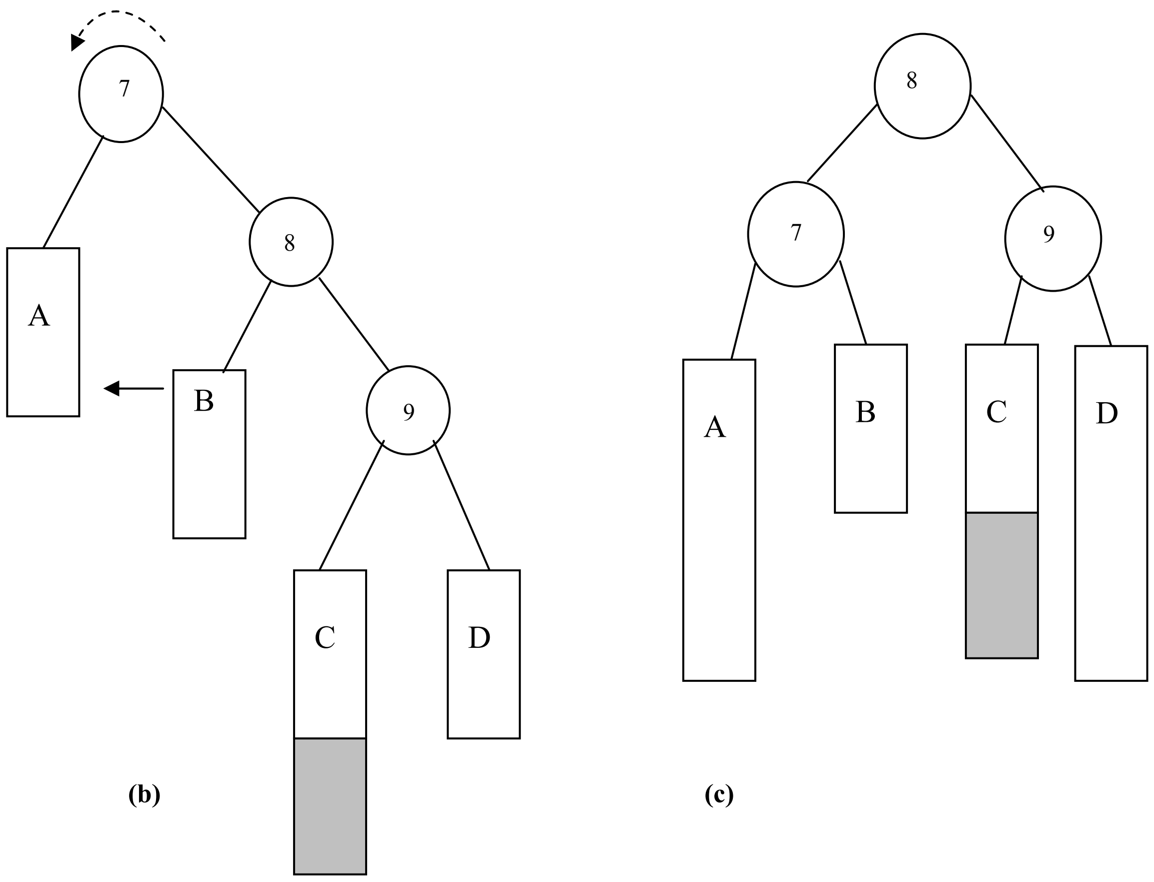


Figure 7.10: Double left rotation when a new node is inserted into the AVL tree

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.

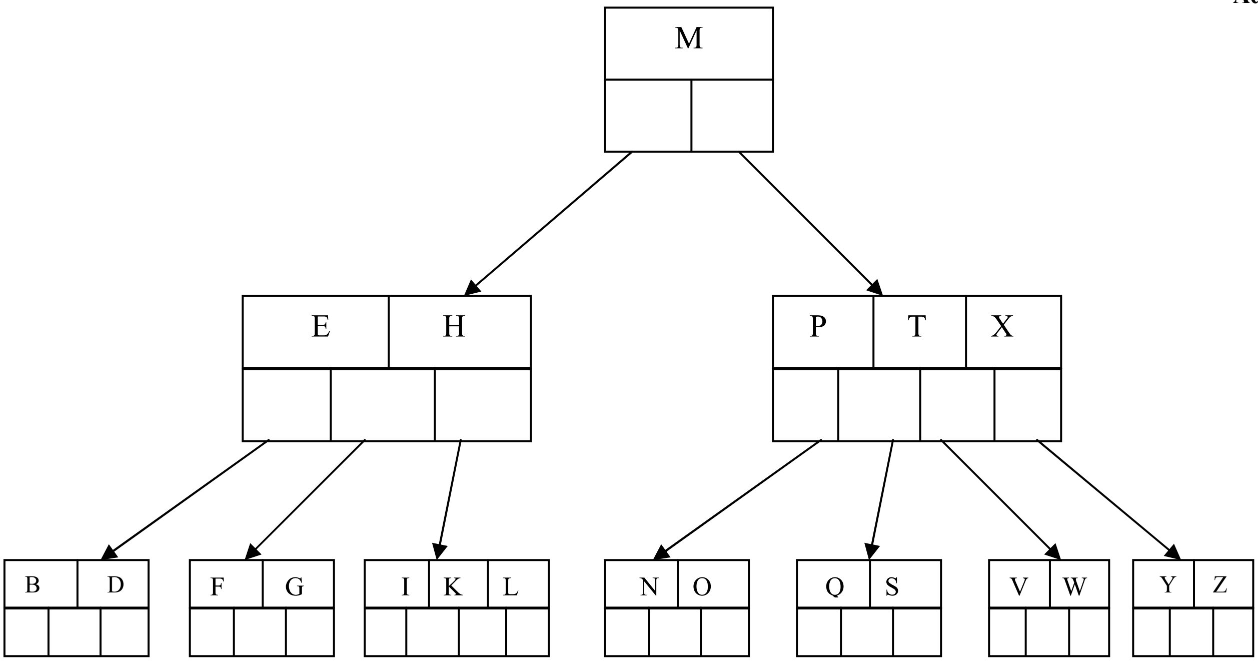


Figure 7.11: A B-tree of order 5

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 disk

All 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] ← True
n [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
                $C_1$ ← 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.

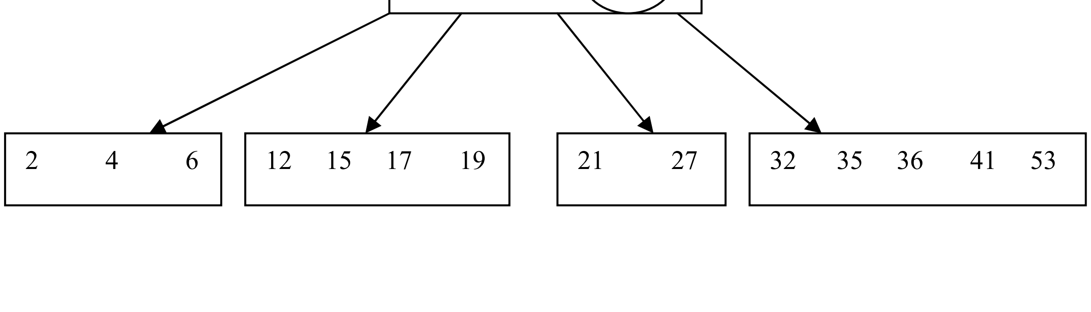
If the node is not full prior to the insertion, then no special action is required.

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.

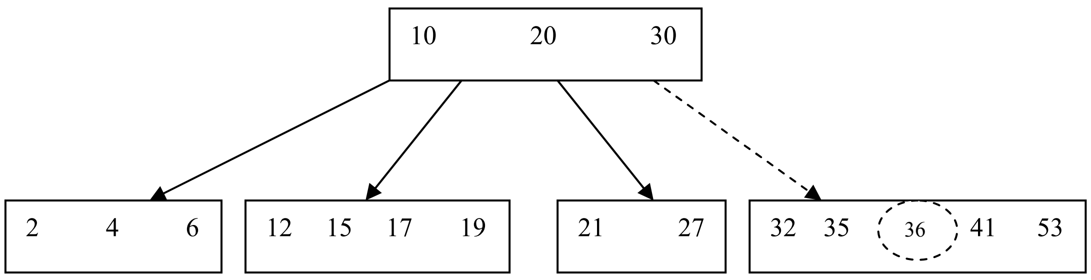
This process may repeat all the way up to the root and may require splitting the root node.

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 split and 36 is shifted upwards.



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

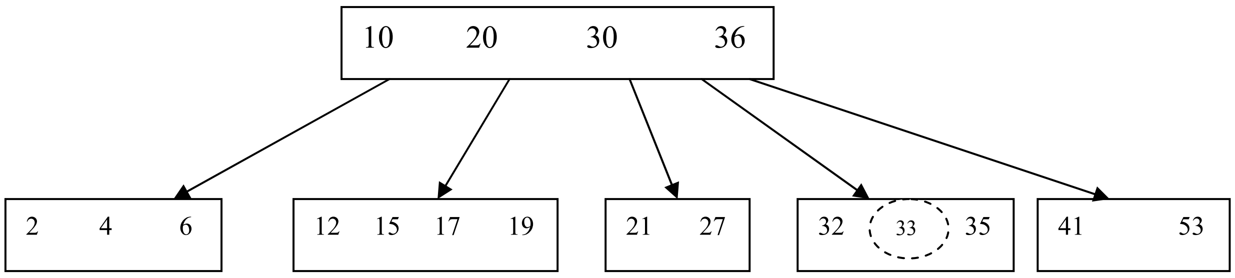
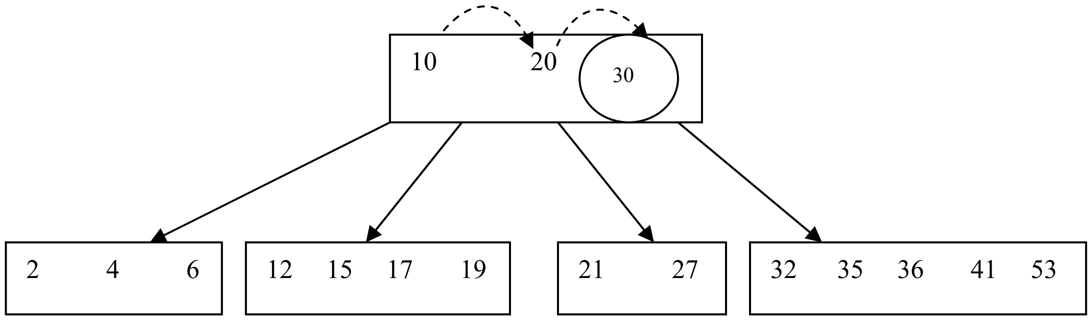


Figure 7.12 : A B-tree

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.

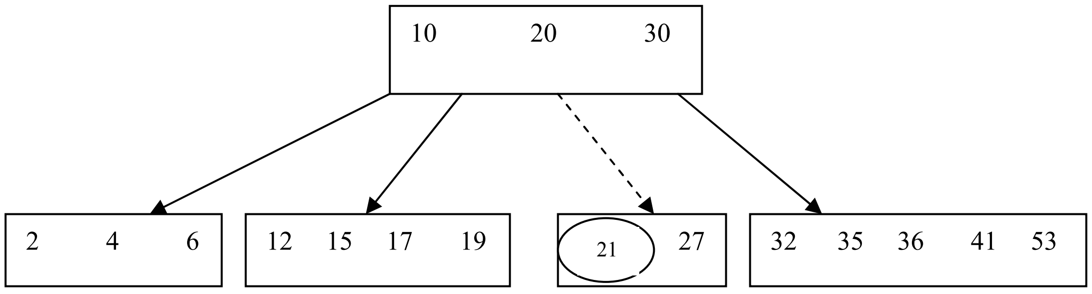


Figure 7.13 : A B-tree

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 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.

7.6 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 values is 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.

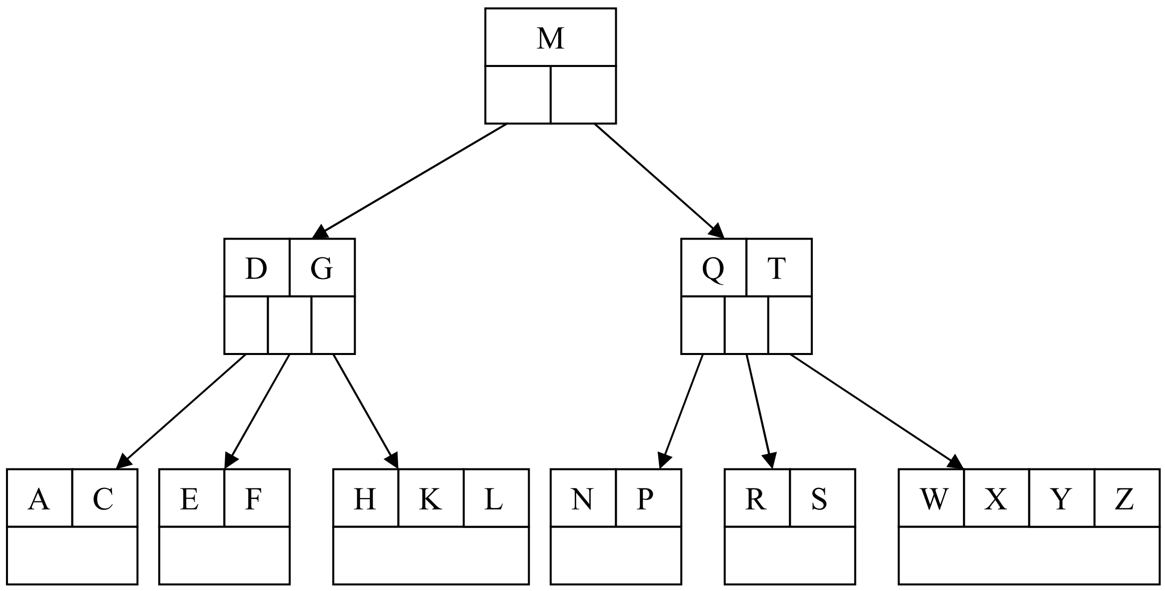
Check Your Progress 2

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.

7.7 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