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*

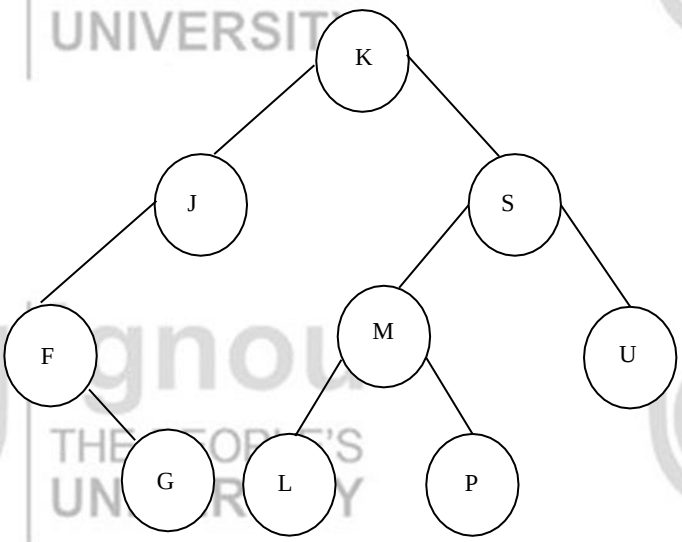


Figure 7.1: A Binary Search Tree(BST)

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.

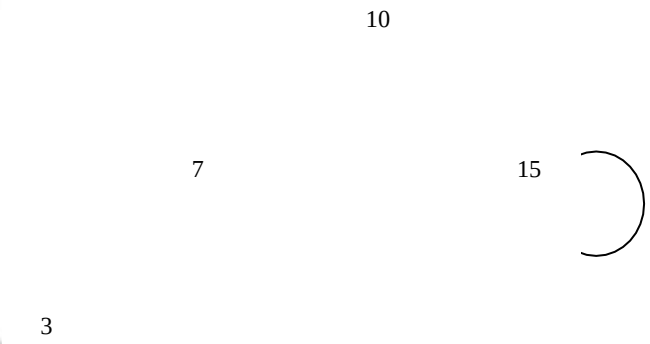


Figure 7.2: A non-empty BST

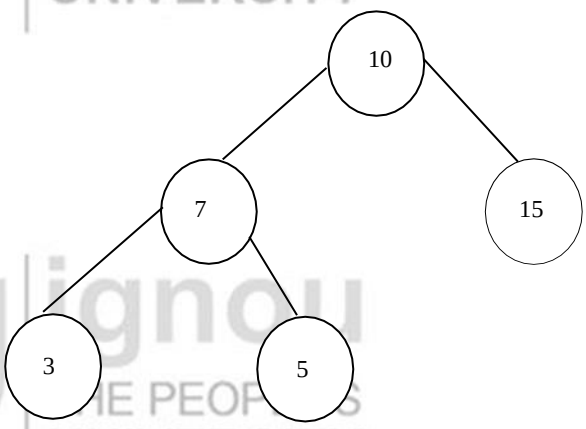


Figure 7.3: Figure 7.2 after insertion of 5

7.2.3 Deletion of a node from a Binary Search Tree

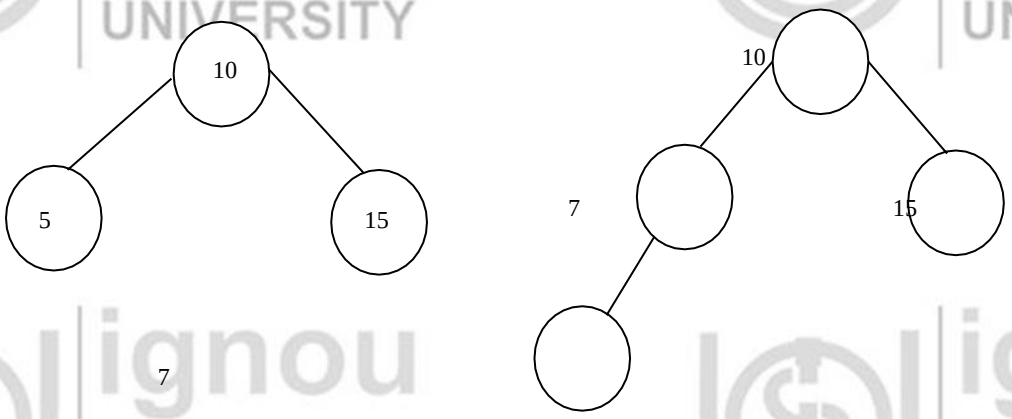
The algorithm to delete a node with key from a binary search tree is not simple where as many cases needs to be considered.

- 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?

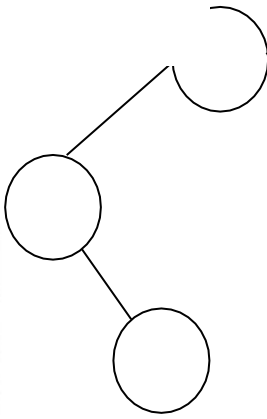
……

……

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.

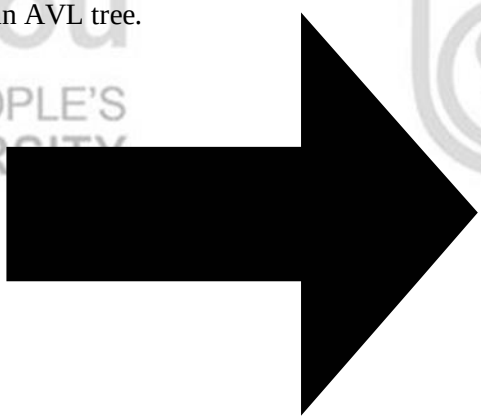


Figure 7.4 : Balance requirement for an AVL tree: the left and right subtree differ by atmost one in height

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*).

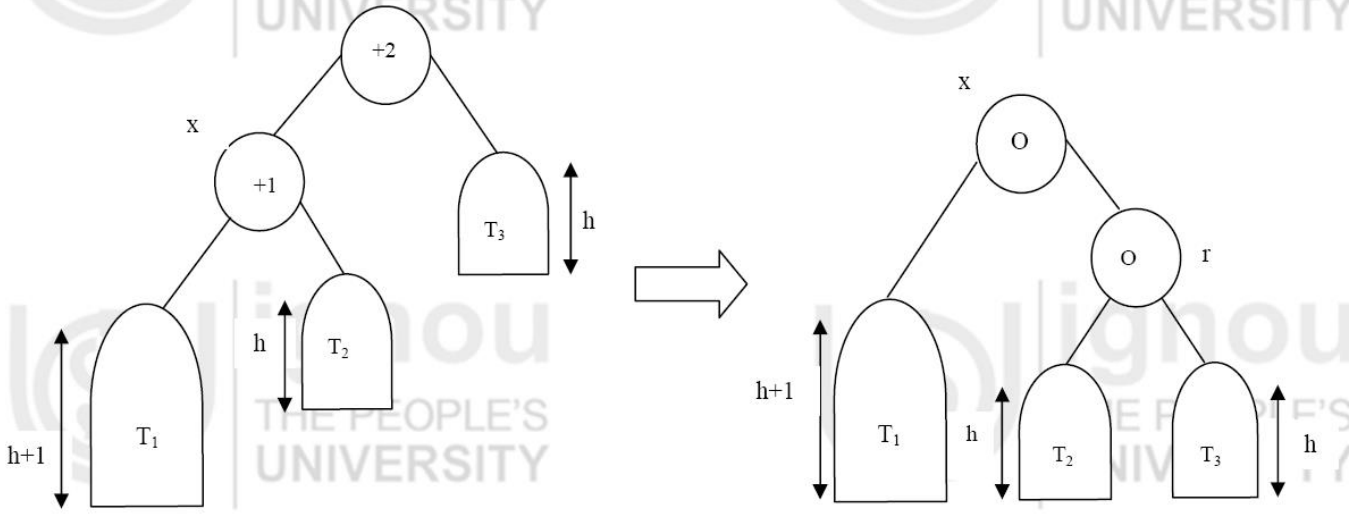


Figure 7.5: LL Rotation

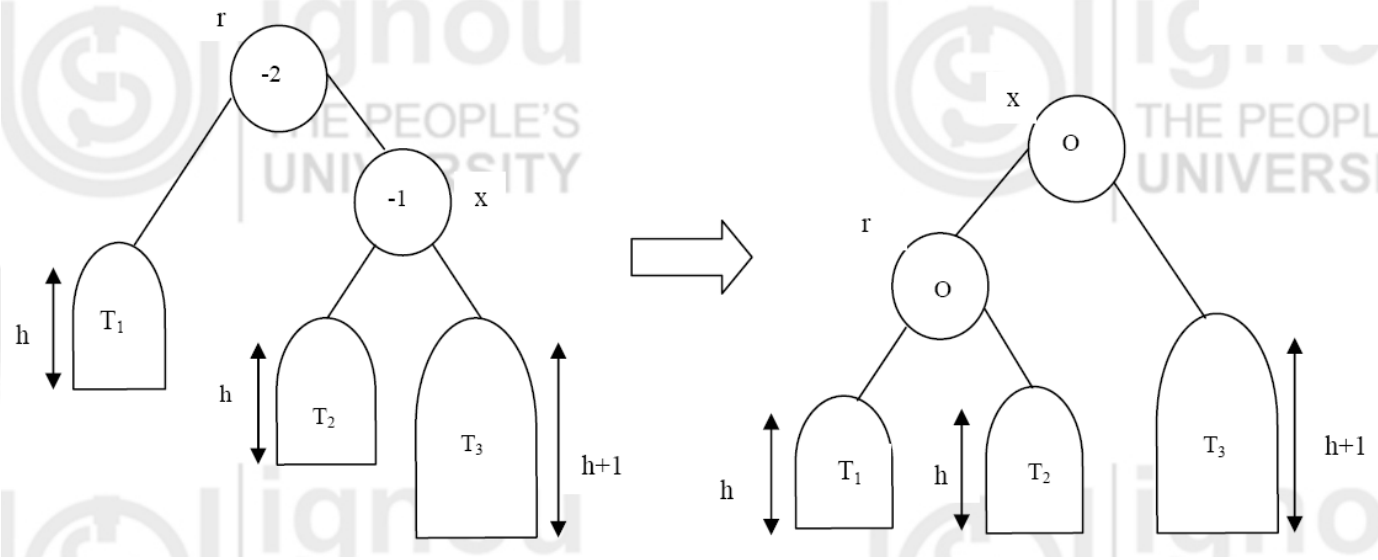


Figure: 7.6 RR Rotati

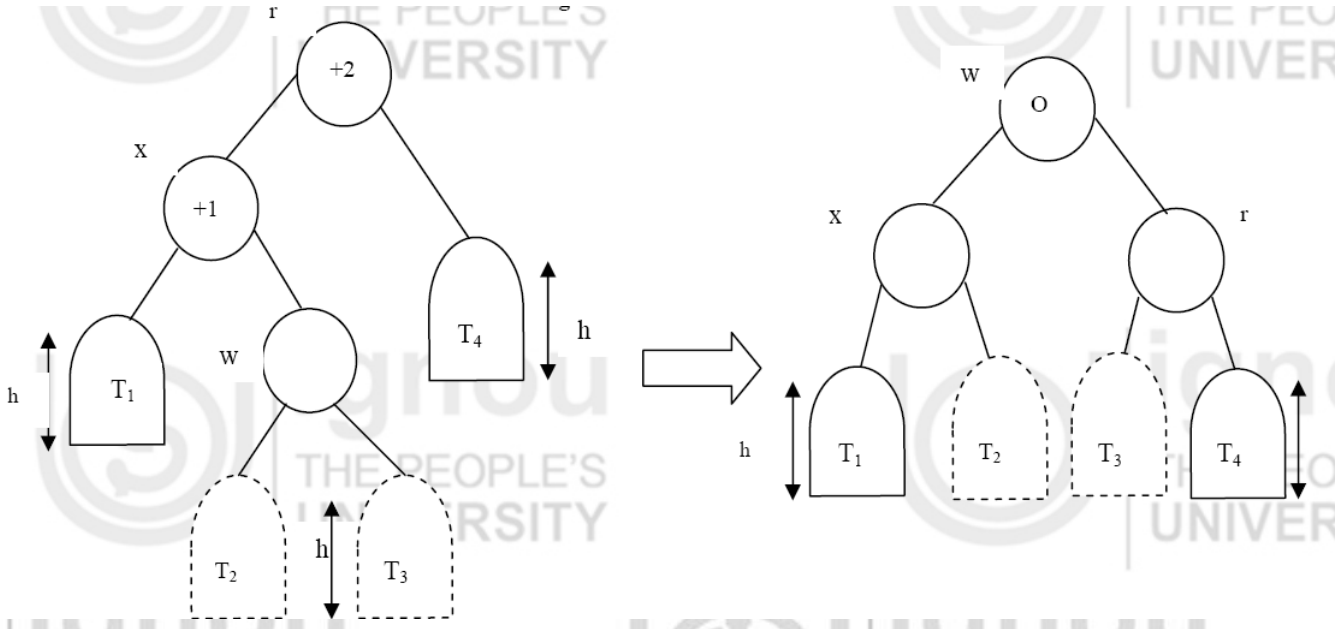


Figure 7.7: LR Rotation

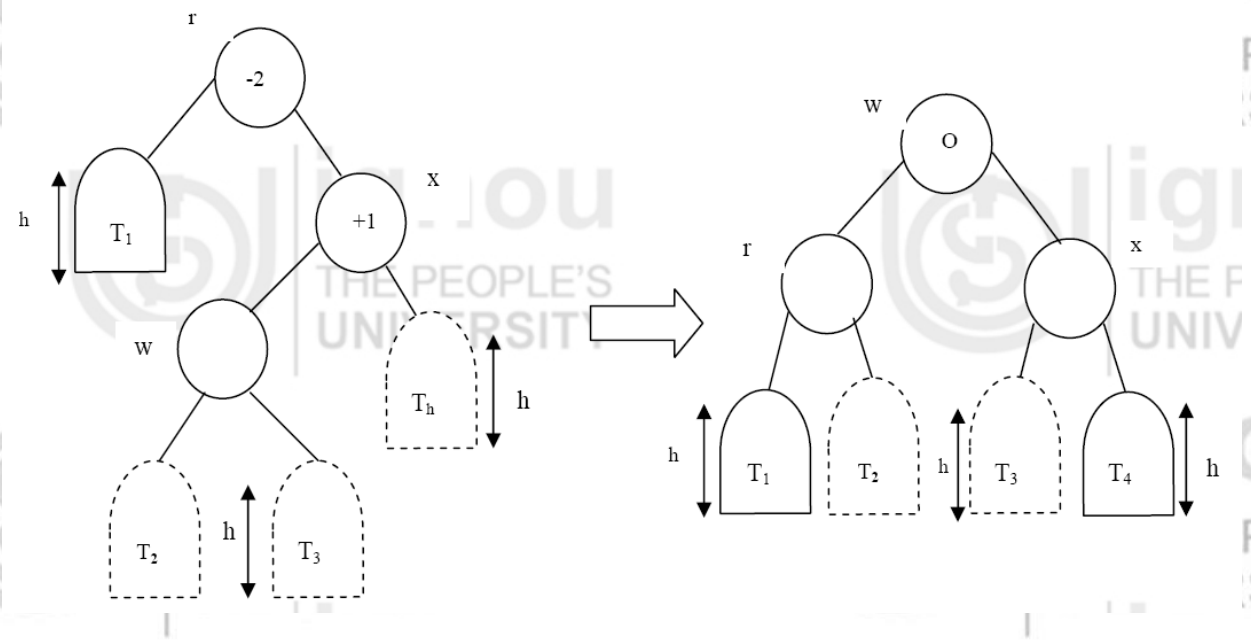


Figure 7.8: RL Rotation

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

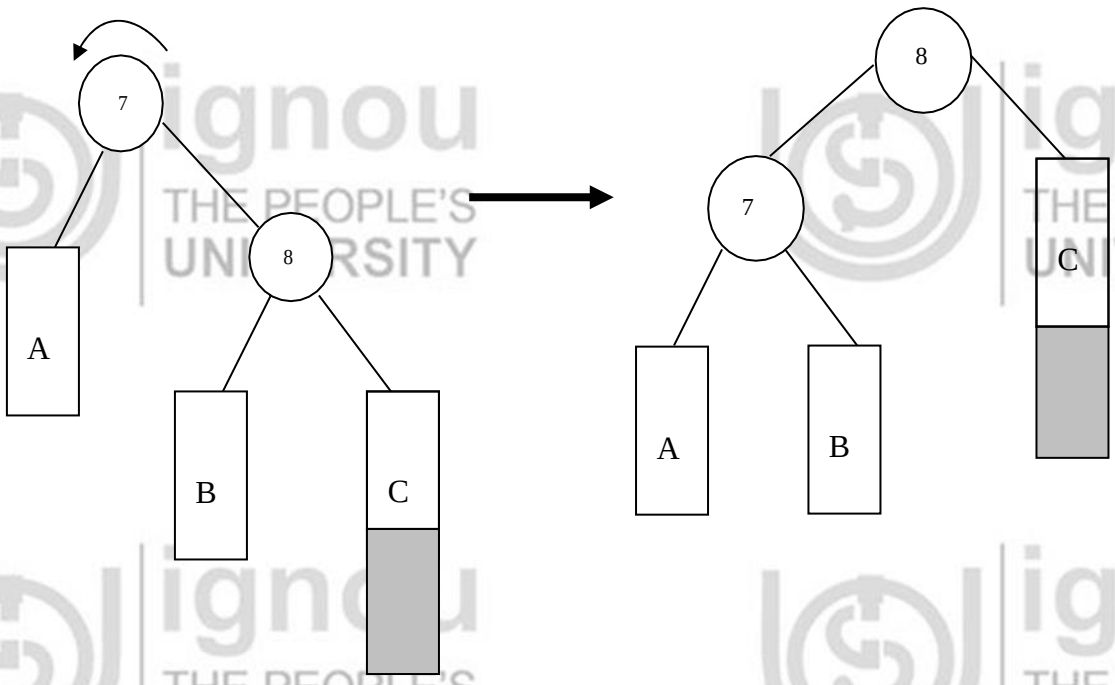


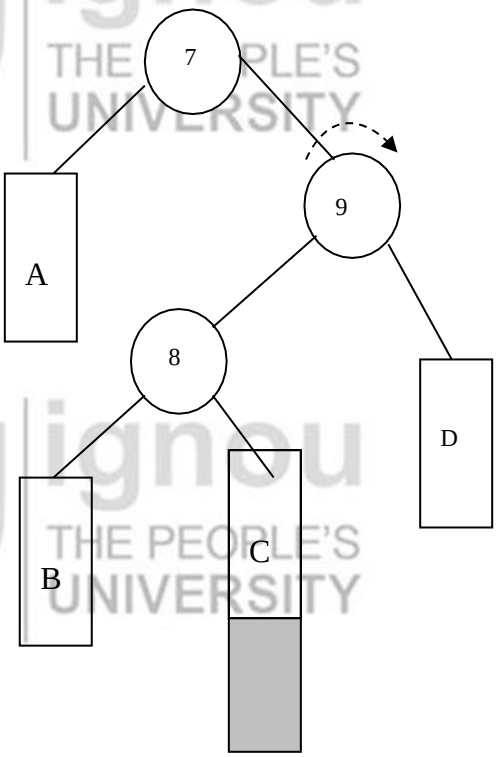
Figure 7.9: LL Rotation

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.

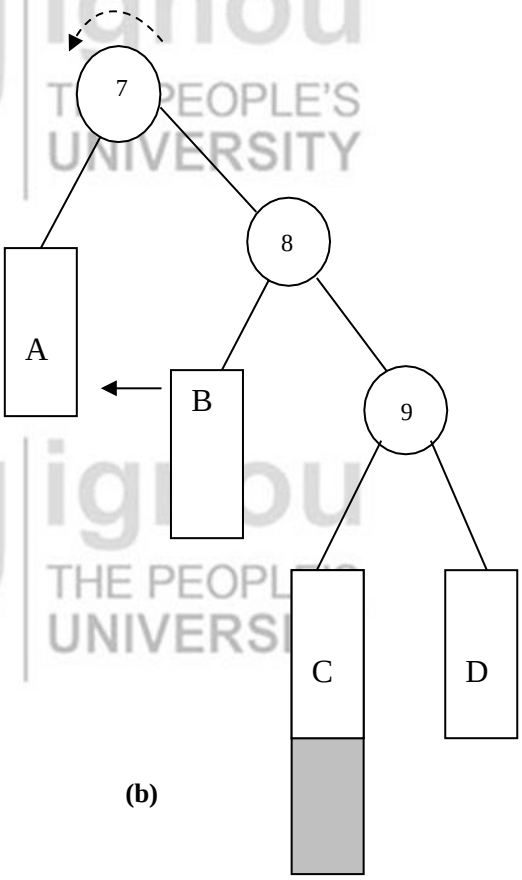
The balance was broken when we inserted a node into the right child of 7, since the difference in height became 7.

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.

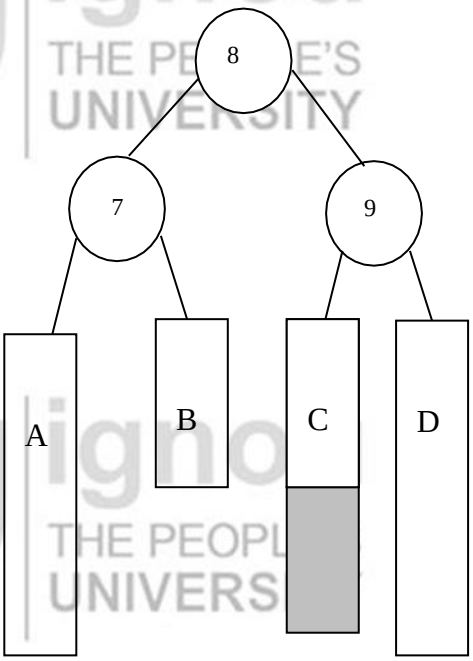
Example: (Double left rotation when a new node is inserted into the AVL tree (RL rotation)) (refer to *Figure 7.10 (a),(b),(c)*).



(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.

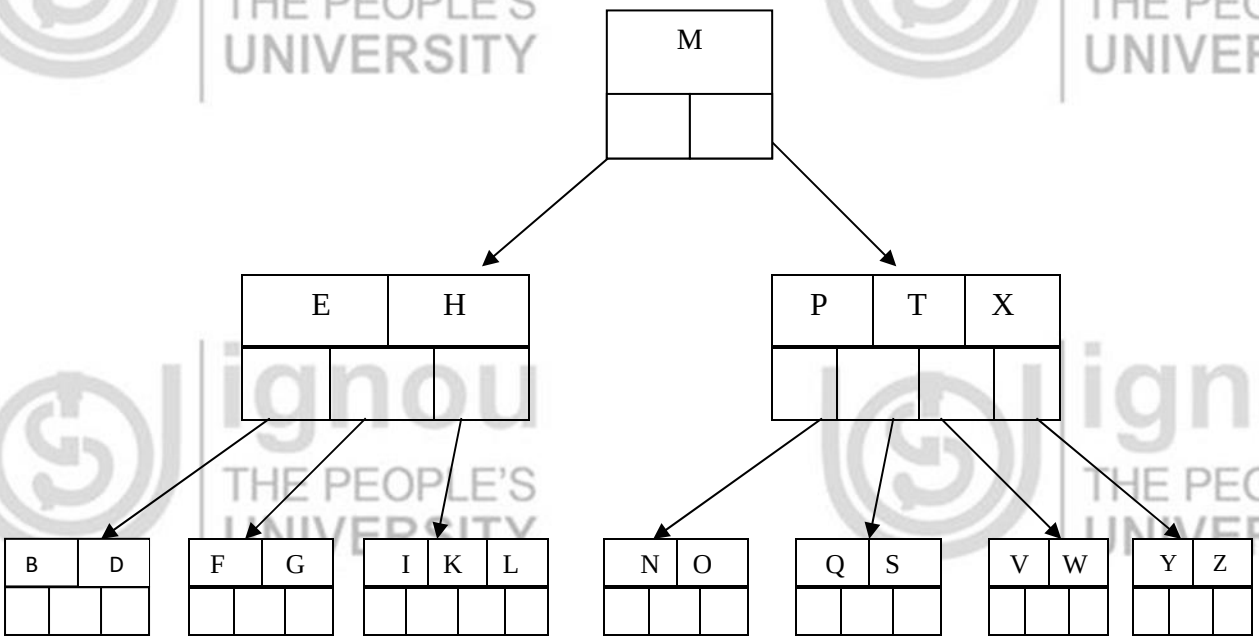


Figure 7.11: A B-tree of order 5

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.

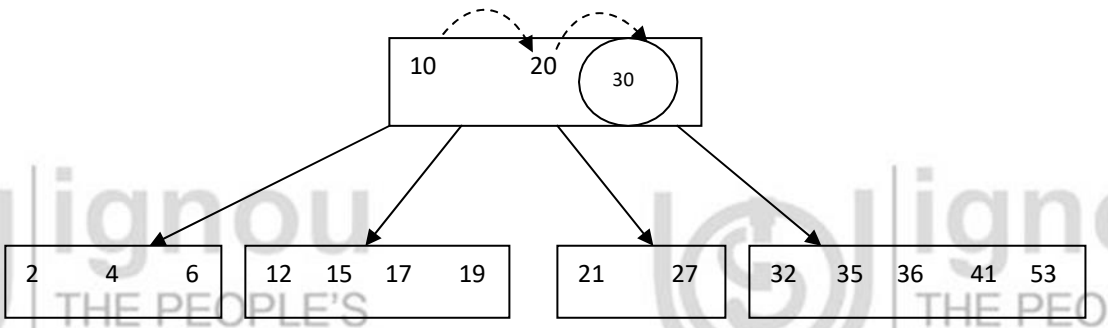
If the node is not full prior to the insertion, then no special action is required.

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.

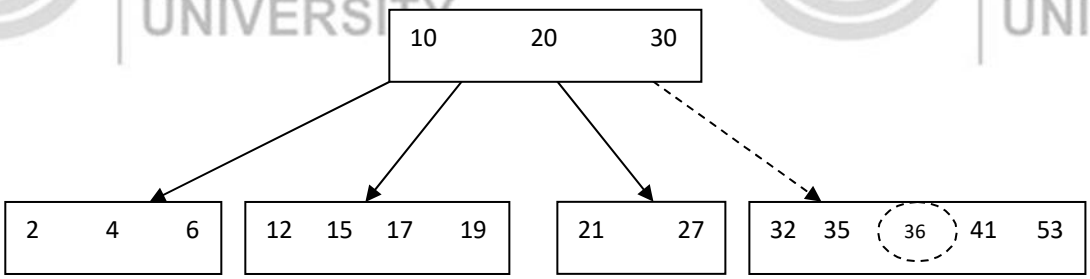
This process may repeat all the way up to the root and may require splitting the root node.

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.

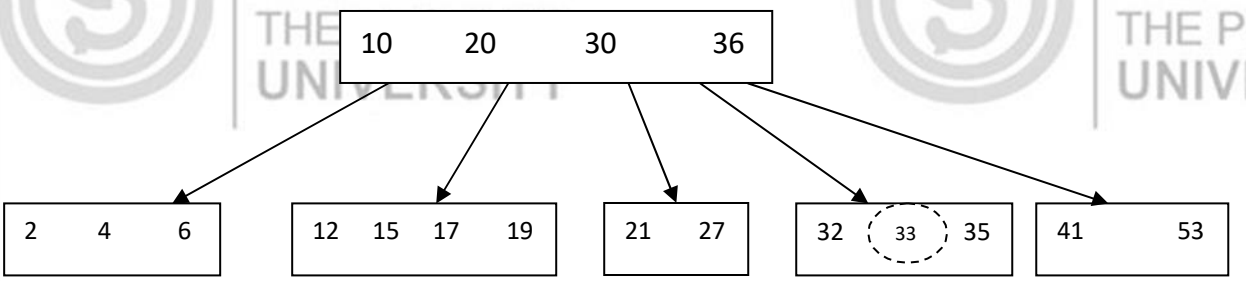
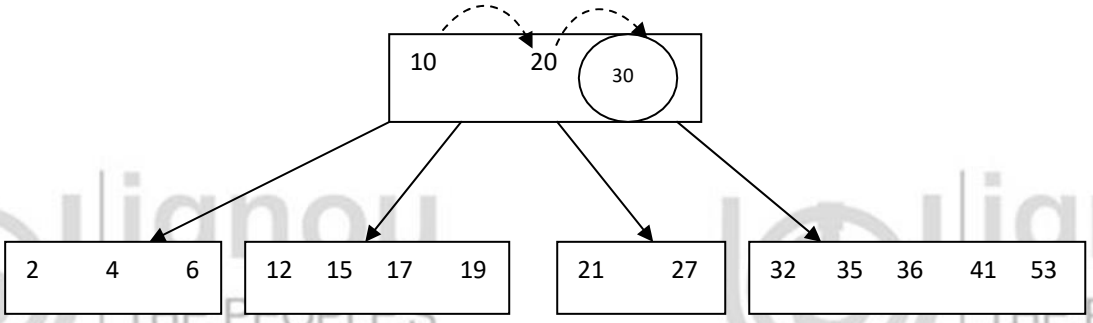


Figure 7.12 : A B-tree

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.

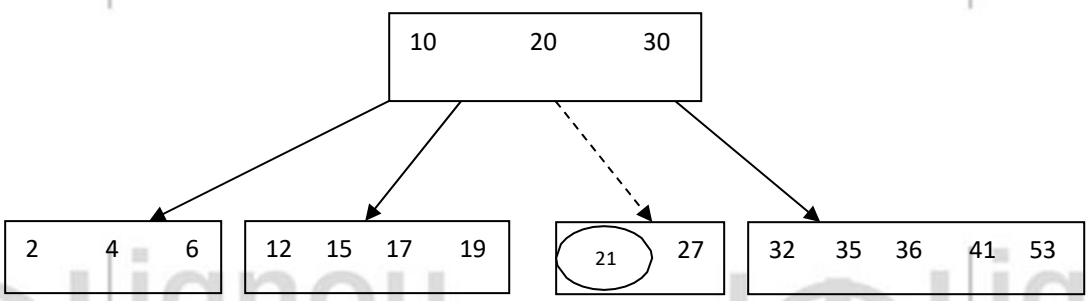


Figure 7.13 : A B-tree

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:

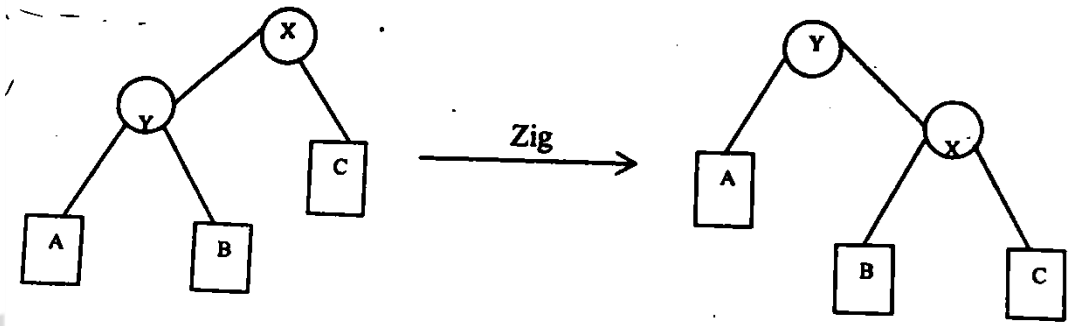
Zig: Movement of one step down the path to the left to fetch a node up. Zag: Movement of one step down the path to the right to fetch a node up.

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

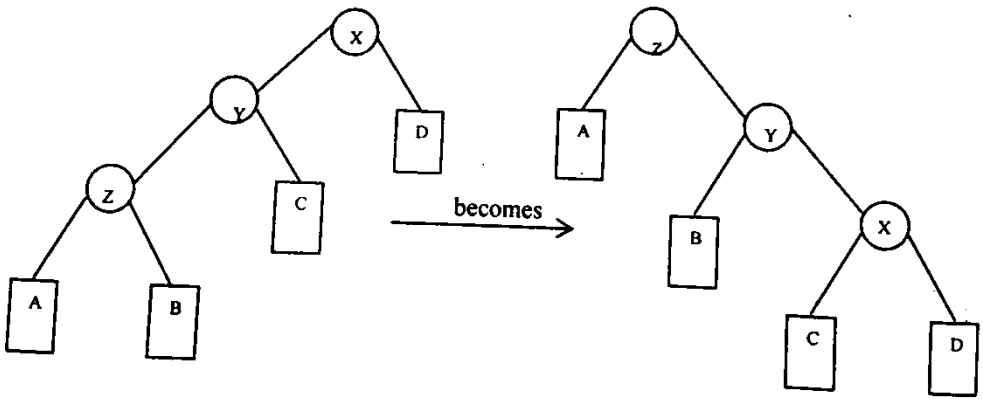
Zag-Zag: Movement of two steps down to the right.Zig-Zag: Movement of one step left and then right. Zag-Zig: Movement of one step right and then left.

Figure 7.14 depicts the splay rotations.

Zig:



Zig-Zig:



Zig-Zag:

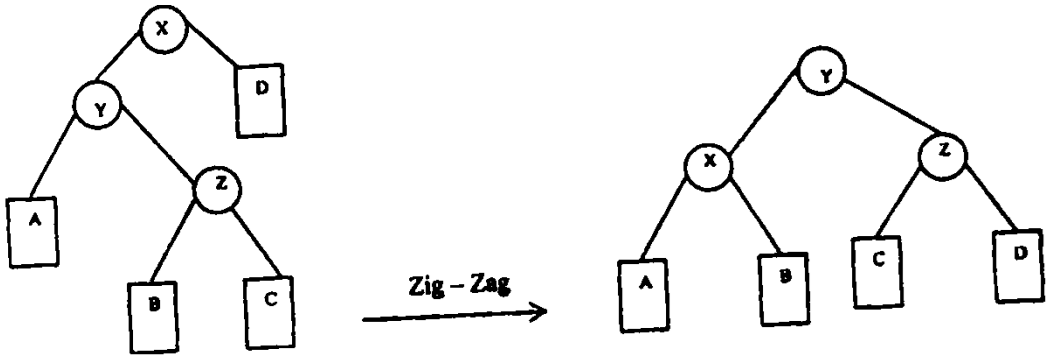


Figure 7.14: Splay rotations

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:

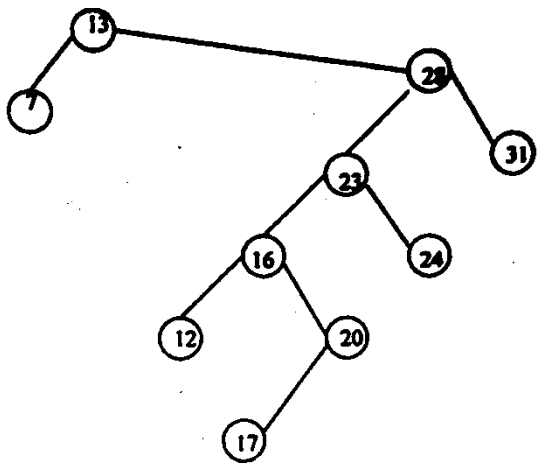
a) **Central SubTree**: This is initially the complete tree and may contain the target node. Search proceeds by comparison of the target value with the root and ends with the root of the central tree being the node containing the target if present ornull node if the target is not present.

b) **Left SubTree**: 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.

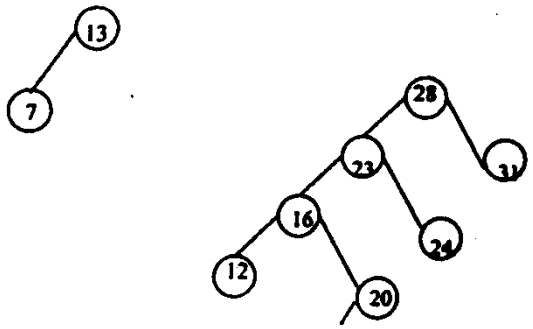
c) **Right SubTree**: This is also initially empty and is created similar to left subtree. It consists of nodes with values more than the target node.

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

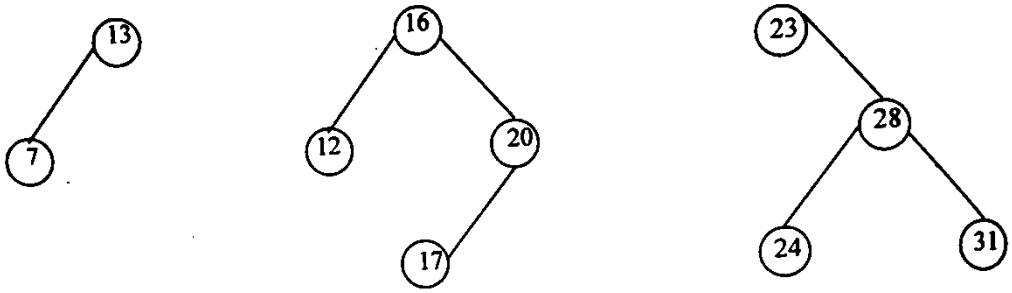
Initially,



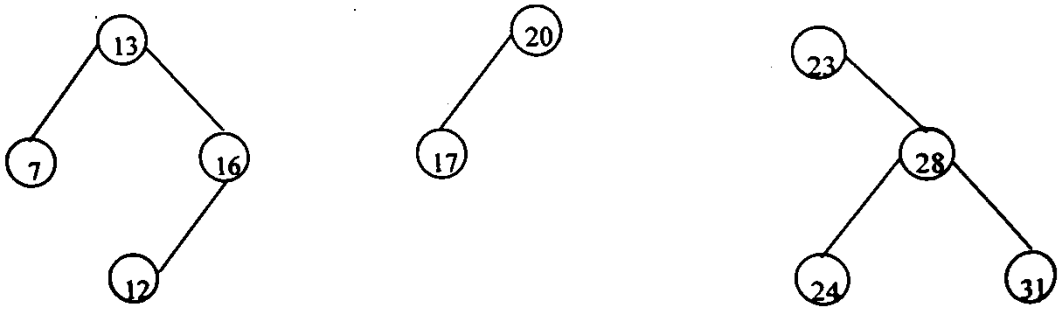
The first step is Zig-Zag:



The next step is Zig-Zig:



The next step is the terminal zig:



Finally, reassembling the three trees, we get:

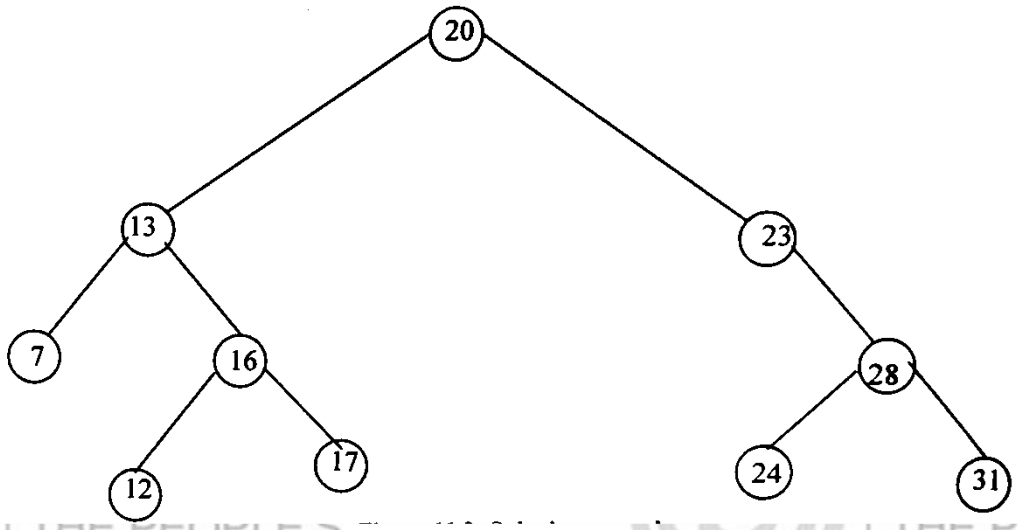


Figure 7.15: Splaying procedure

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.

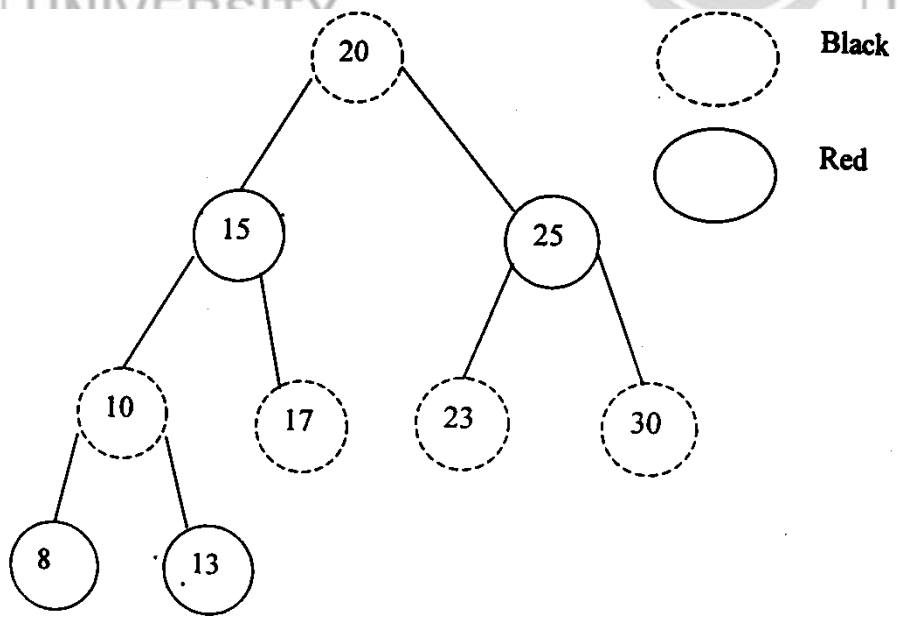


Figure 7.16: A Red-Black tree

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*).

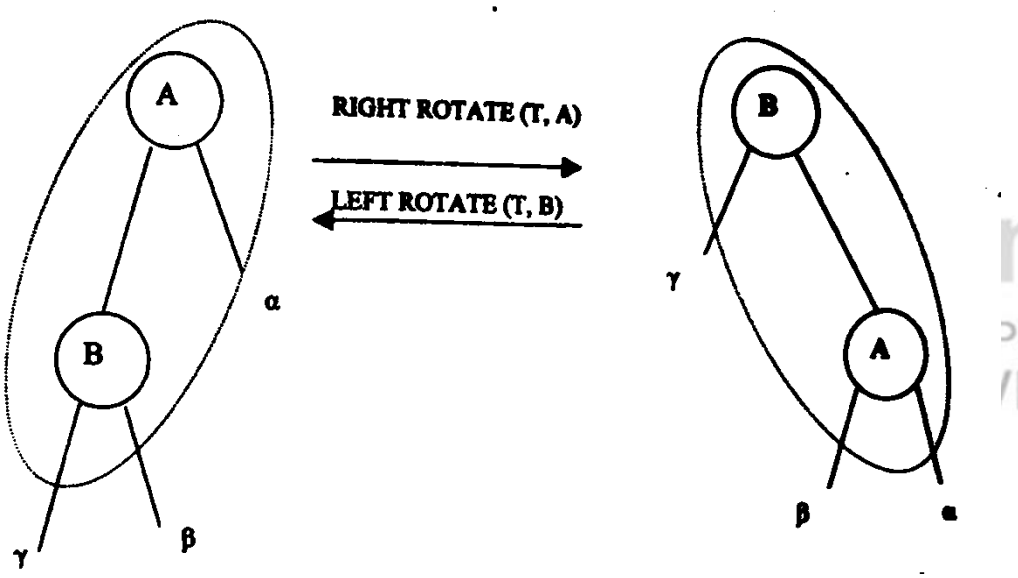


Figure 7.17: Left rotation

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*).

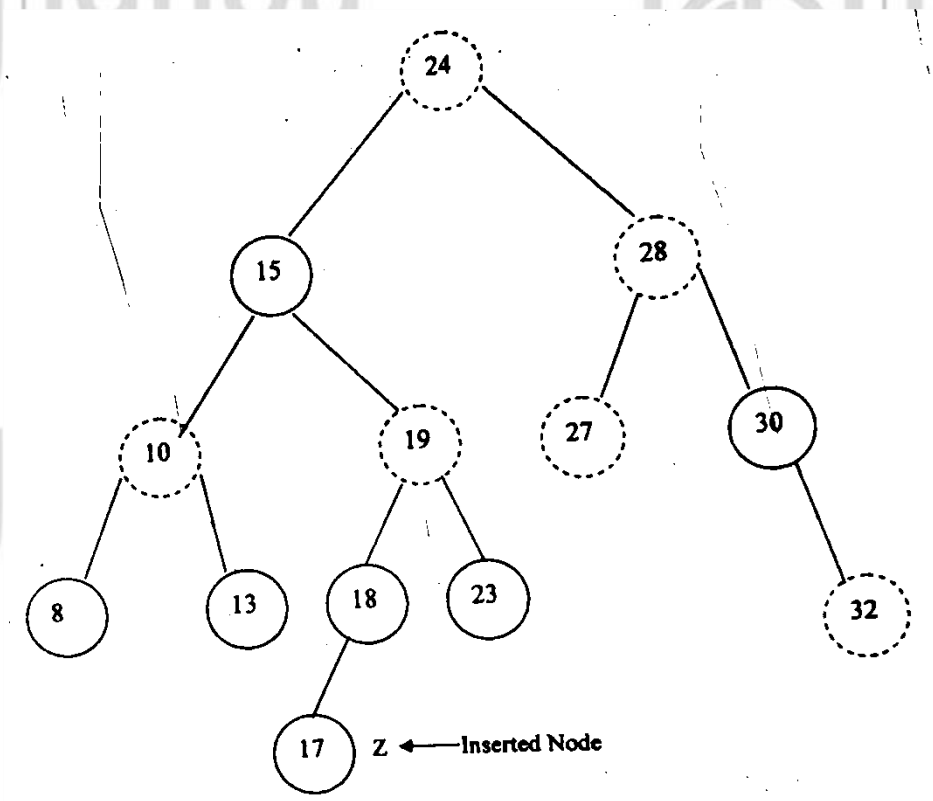


Figure 7.19: A Red-Black Tree after insertion of node 17

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:

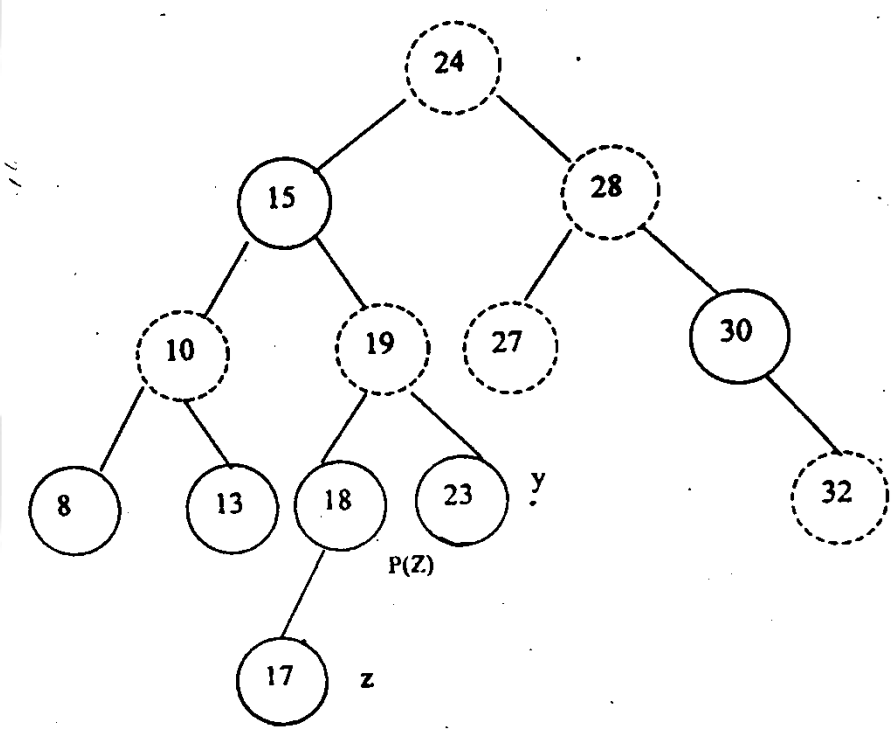


Figure 7.20: Z is to the left of it's parent

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

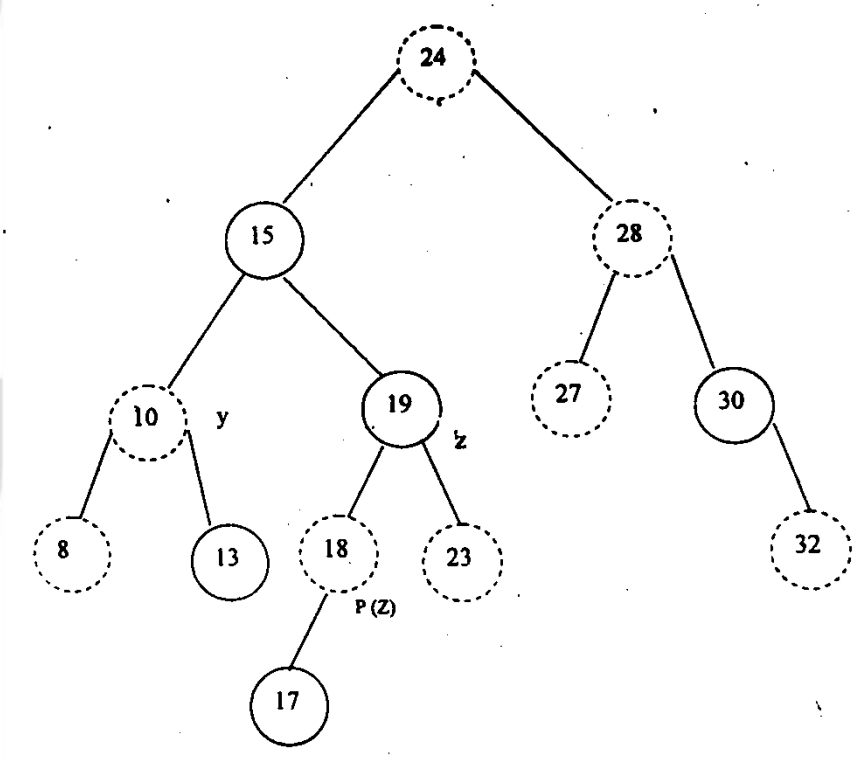


Figure 7.21: Both Z and P(Z) are red

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

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

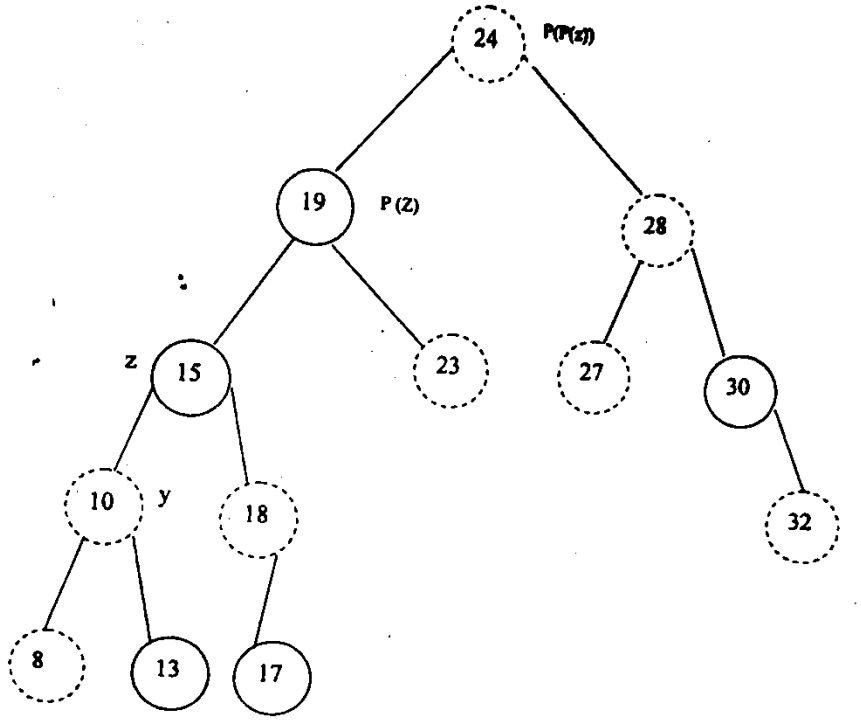


Figure 7.22: Result of application of case-2

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

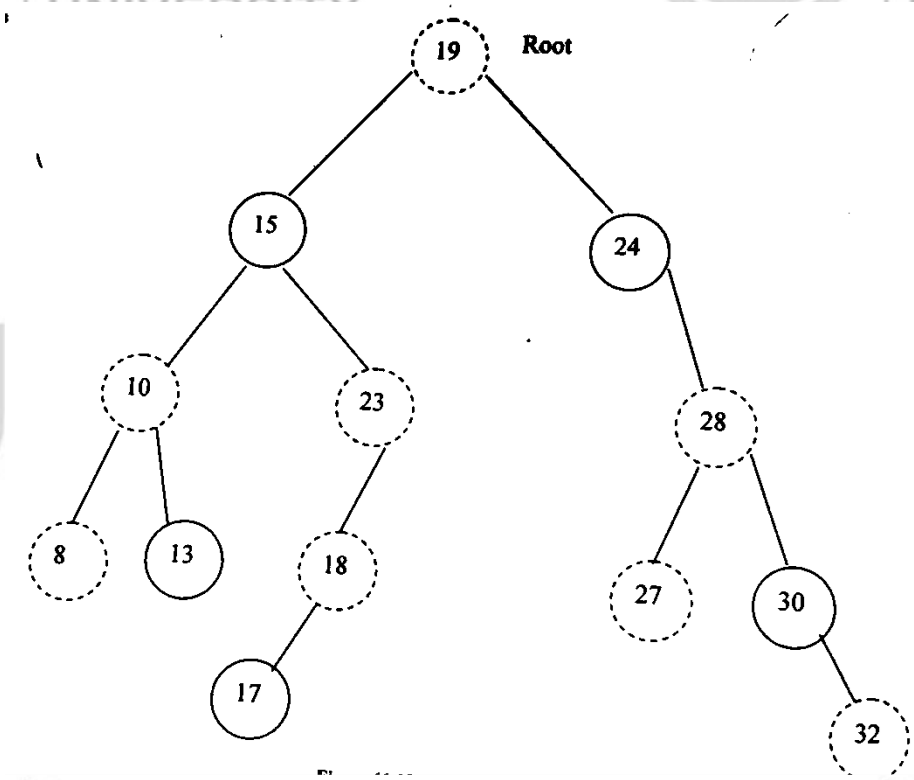


Figure 7.23: Result of application of case-3

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)).

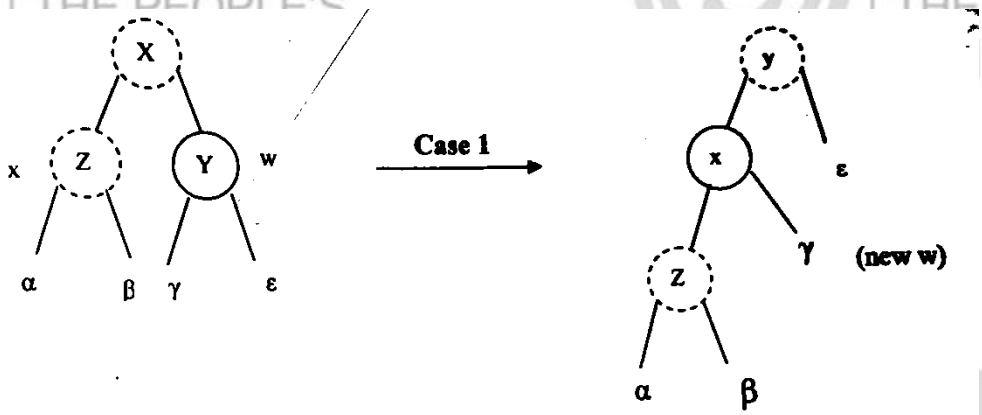


Figure 7.24: Application of case-1

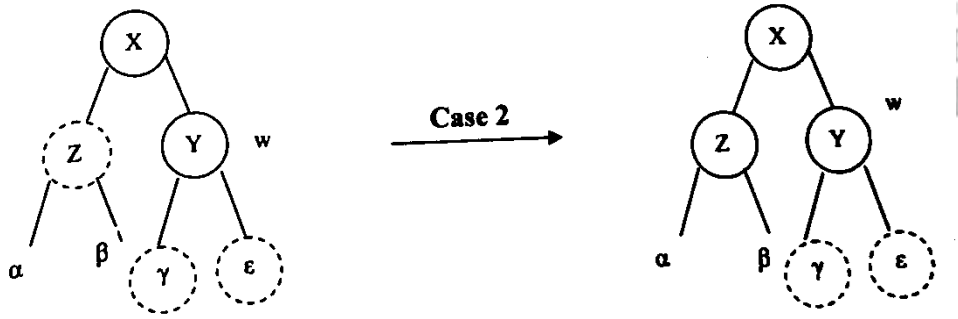


Figure 7.25: Application of case-2

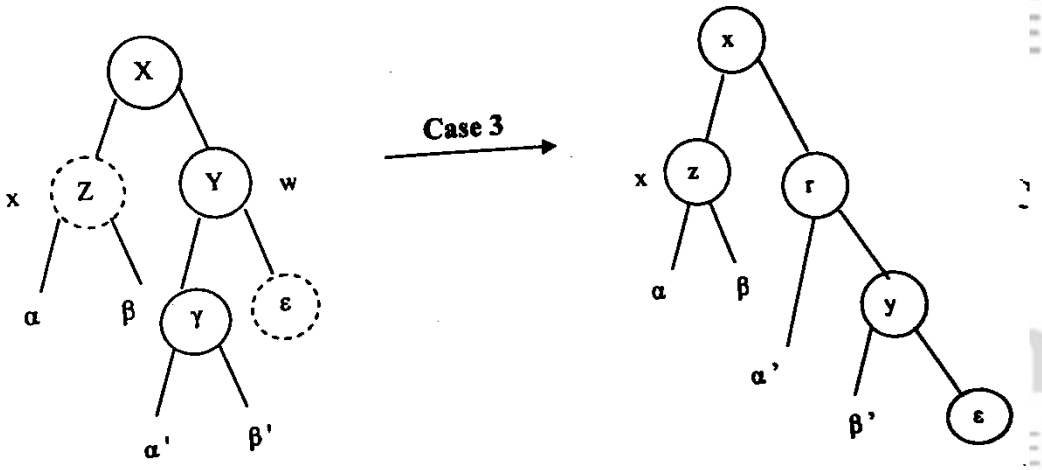


Figure 7.26: Application of case-3

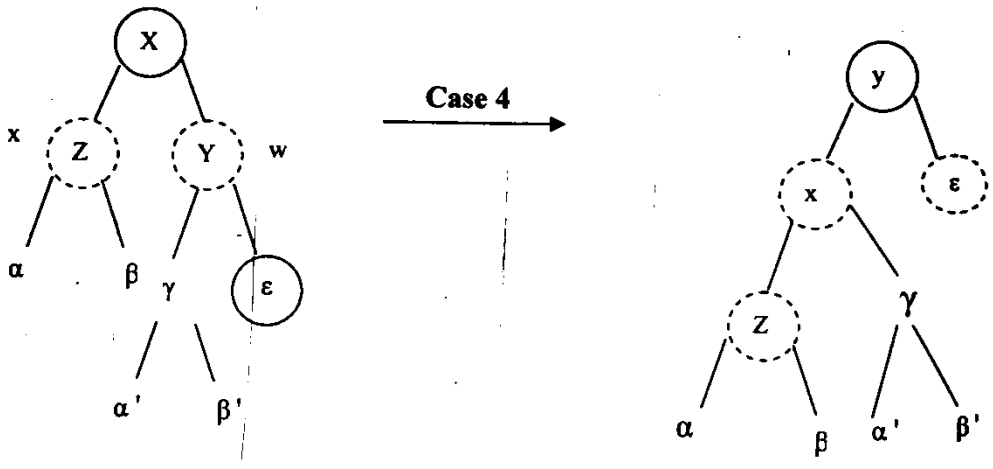


Figure7.27: Application of case-4

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*).

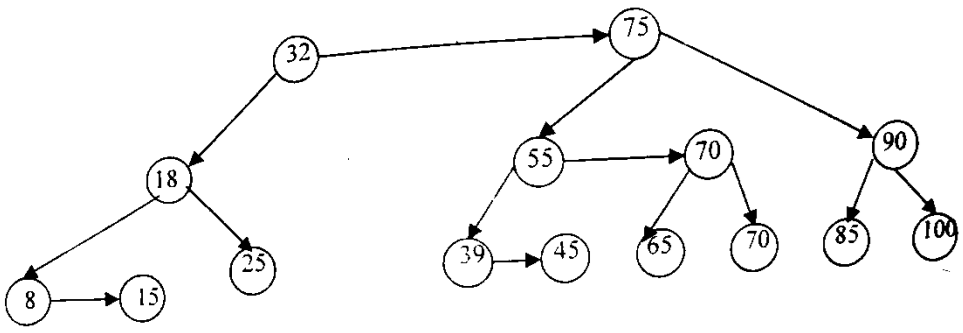


Figure 7.28: AA-tree

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*).

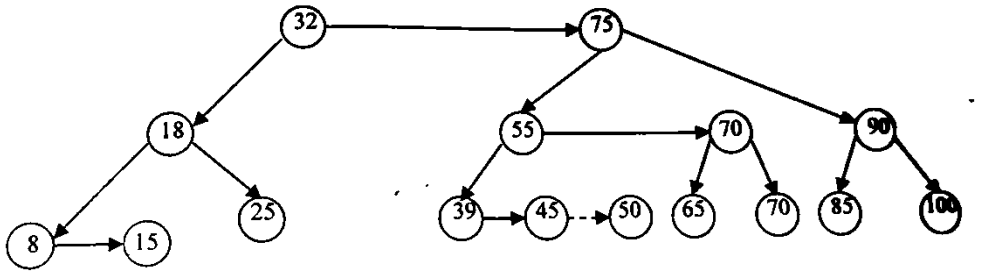


Figure7.29: After inserting node 50

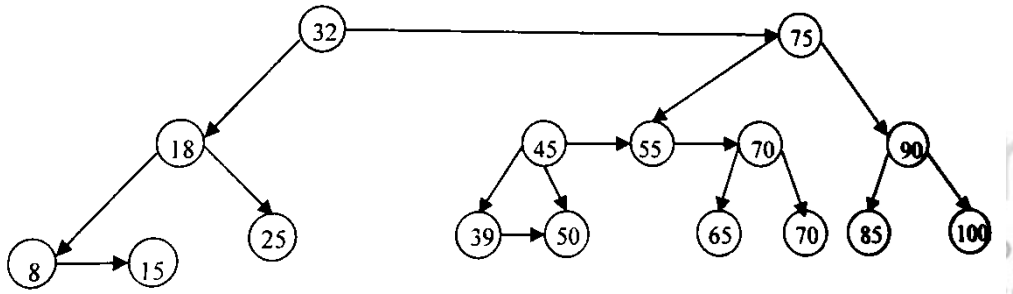


Figure 7.30: Split at node 39(left rotation)

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.

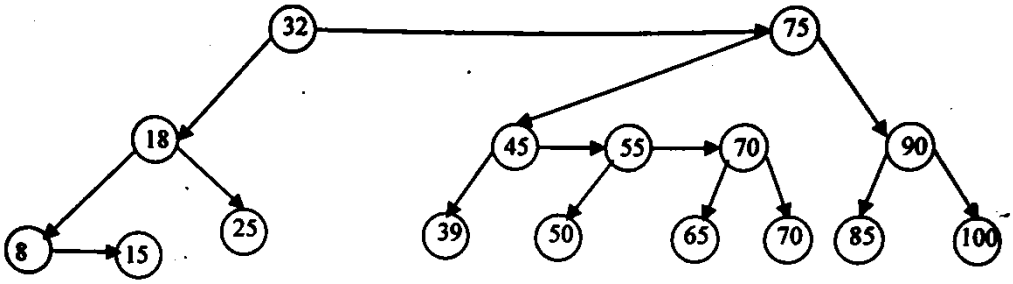


Figure 7.31: Skew at 55 (right rotation)

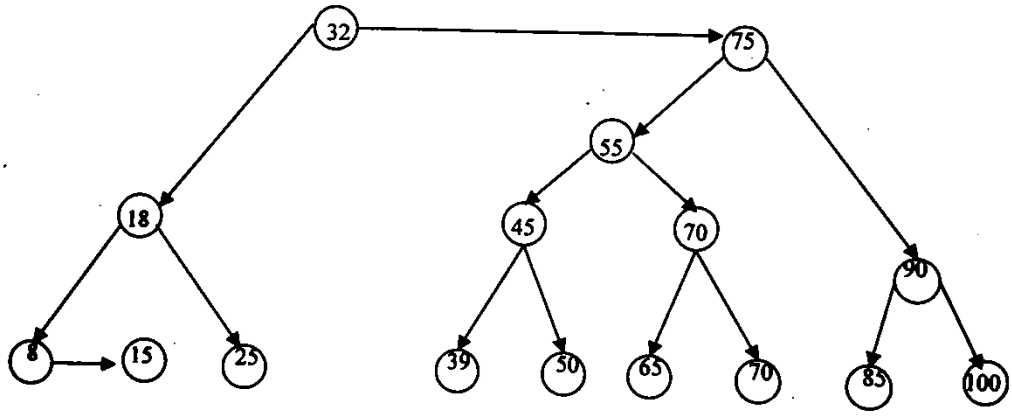


Figure7.32: Split at 45

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.

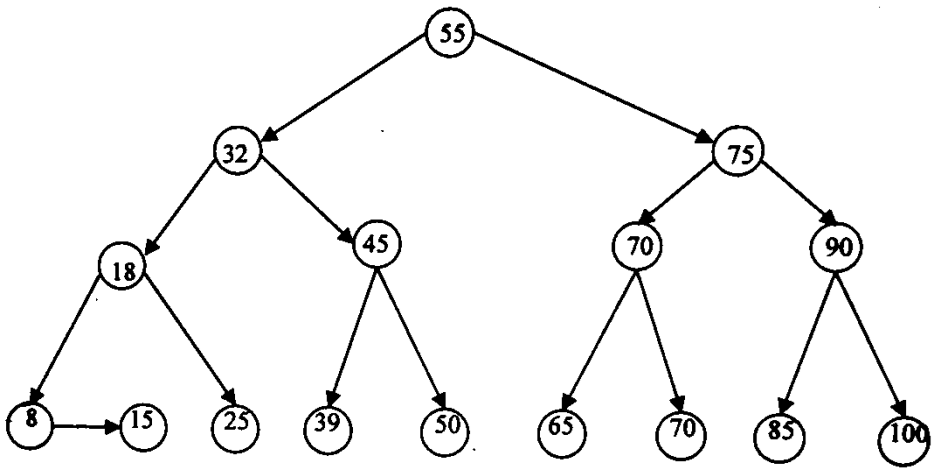


Figure: 7.33 : The Final AA-tree

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.

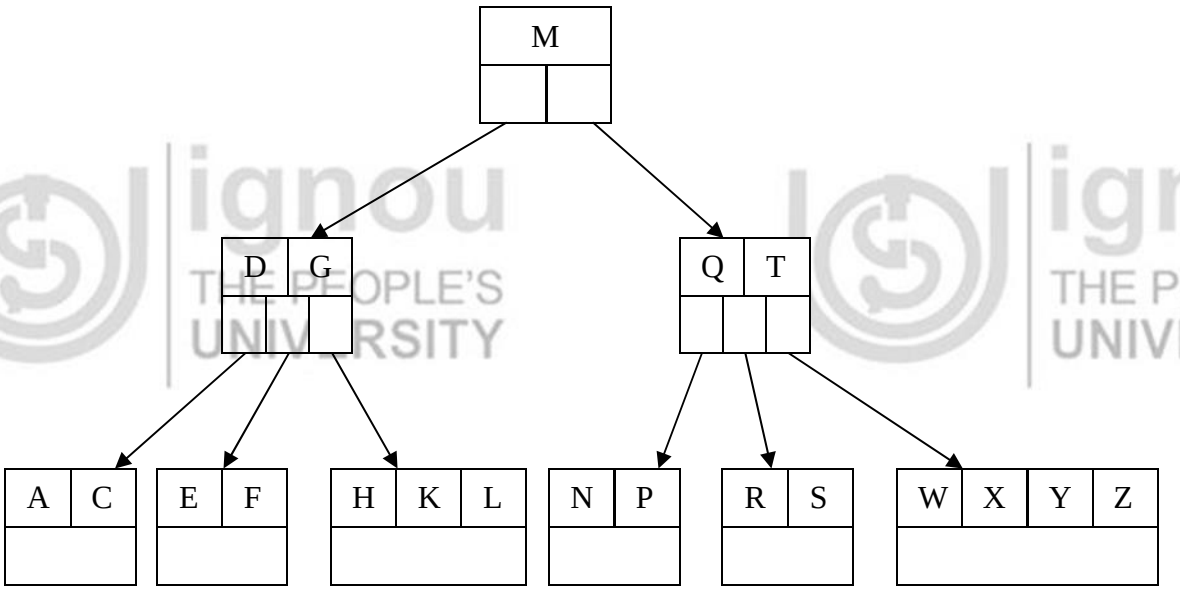
Check Your Progress 2

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

UNIT 8 GRAPHS

Structure **Page Nos.**

8.0 INTRODUCTION

In this unit, we will discuss a data structure called Graph. In fact, graph is a general tree with no parent-child relationship. Graphs have many applications in computer science and other fields of science. In general, graphs represent a relatively less restrictive relationship between the data items. We shall discuss about both undirected graphs and directed graphs. The unit also includes information on different algorithms which are based on graphs.

8.1 OBJECTIVES

After going through this unit, you should be able to

- know about graphs and related terminologies;
- know about directed and undirected graphs along with their representations;
- know different shortest path algorithms;
- construct minimum cost spanning trees;
- apply depth first search and breadth first search algorithms, and
- finding strongly connected components of a graph.

8.2 DEFINITIONS

A graph **G** may be defined as a finite set **V** of vertices and a set **E** of edges (pair of connected vertices). The notation used is as follows:

Graph G = (V, E)
Consider the graph of *Figure 8.1.*

The set of vertices for the graph is V = {1, 2, 3, 4, 5}.
The set of edges for the graph is E = {(1,2), (1,5), (1,3), (5,4), (4,3), (2,3) }.

The elements of E are always a pair of elements.

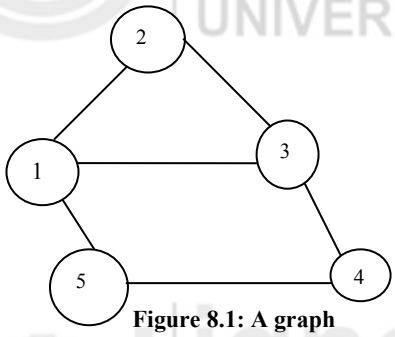


Figure 8.1: A graph

It may be noted that unlike nodes of a tree, graph has a very limited relationship between the nodes (vertices). There is no direct relationship between the vertices 1 and 4 although they are connected through 3.

Directed graph and Undirected graph: If every edge (a,b) in a graph is marked by a direction from a to b, then we call it a Directed graph (digraph). On the other hand, if directions are not marked on the edges, then the graph is called an Undirected graph.

In a Directed graph, the edges (1,5) and (5,1) represent two different edges whereas in an Undirected graph, (1,5) and (5,1) represent the same edge. Graphs are used in various types of modeling. For example, graphs can be used to represent connecting roads between cities.

Graph terminologies :

Adjacent vertices: Two vertices a and b are said to be adjacent if there is an edge connecting a and b. For example, in *Figure 8.1,* vertices 5 and 4 are adjacent.

Path: A path is defined as a sequence of distinct vertices, in which each vertex is adjacent to the next. For example, the path from 1 to 4 can be defined as a sequence of adjacent vertices (1,5), (5,4).

A *path*, **p**, of length, **k**, through a graph is a sequence of connected vertices:

$$\mathbf{p = <v_0, v_1, ..., v_k>}$$

Cycle : A graph contains *cycles* if there is a path of non-zero length through the graph, $p = <v_0, v_1, ..., v_k>$ such that $v_0 = v_k$.

Edge weight : It is the cost associated with edge.

Loop: It is an edge of the form (*v*,*v*).

Path length : It is the number of edges on the path.

Simple path : It is the set of all distinct vertices on a path (except possibly first and last).

Spanning Trees: A *spanning tree* of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.

There are different representations of a graph. They are:

- Adjacency list representation
- Adjacency matrix representation

Adjacency list representation

An Adjacency list representation of a Graph G = {V, E} consists of an array of adjacency lists denoted by *adj of V* list. For each vertex uєV, adj[u] consists of all vertices adjacent to u in the graph G.

Consider the graph of *Figure 8.2.*

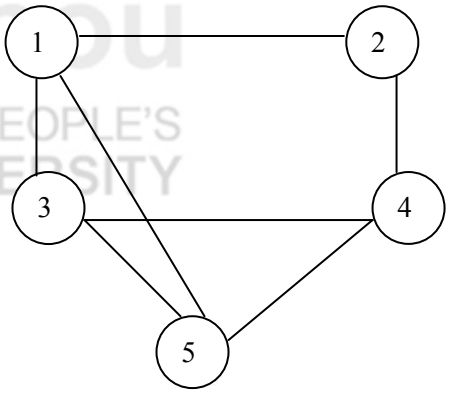


Figure 8.2: A Graph

The following is the adjacency list representation of graph of *Figure 8.2:*

adj [1] = {2, 3, 5}
adj [2] = {1, 4}
adj [3] = {1, 4, 5}
adj [4] = {2, 3, 5}
adj [5] = {1, 3, 4}

An adjacency matrix representation of a Graph G=(V, E) is a matrix A(a_{ij}) such that

a_{ij} = { 1 if edge (i, j) belongs to E
0 otherwise

The adjacency matrix for the graph of *Figure 8.2* is given below:

	1	**2**	**3**	**4**	**5**
1	0	1	1	0	1
2	1	0	0	1	1
3	1	0	0	1	1
4	0	1	1	0	1
5	1	0	1	1	0

Observe that the matrix is symmetric along the main diagonal. If we define the adjacency matrix as A and the transpose as A^T , then for an undirected graph G as above, $A = A^T$.

Graph connectivity :

A connected graph is a graph in which path exists between every pair of vertices.

A strongly connected graph is a directed graph in which every pair of distinct vertices are connected with each other.

A weakly connected graph is a directed graph whose underlying graph is connected, but not strongly connected.

A complete graph is a graph in which there exists edge between every pair of vertices.

☞ Check Your Progress 1

1) A graph with no cycle is called _____ graph.
2) Adjacency matrix of an undirected graph is _____ on main diagonal.
3) Represent the following graphs(*Figure 8.3* and *Figure 8.4*) by adjacency matrix:

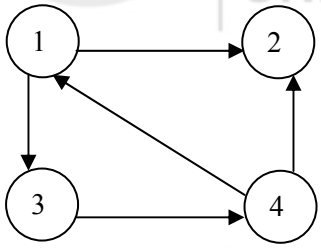


Figure 8.3: A Directed Graph

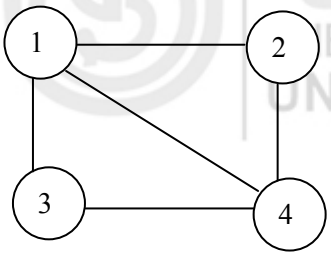


Figure 8.4: A Graph

8.3 SHORTEST PATH ALGORITHMS

A driver takes shortest possible route to reach destination. The problem that we will discuss here is similar to this kind of finding shortest route in a graph. The graphs are weighted directed graphs. The weight could be time, cost, losses other than distance designated by numerical values.

Single source shortest path problem : To find a shortest path from a single source to every vertex of the Graph.

Consider a Graph G = (V, E). We wish to find out the shortest path from a single source vertex sϵV, to every vertex vϵV. The single source shortest path algorithm (Dijkstra's Algorithm) is based on assumption that no edges have negative weights.

The procedure followed to find shortest path are based on a concept called relaxation. This method repeatedly decreases the upper bound of actual shortest path of each vertex from the source till it equals the shortest-path weight. Please note that shortest path between two vertices contains other shortest path within it.

8.3.1 Dijkstra's Algorithm

Djikstra's algorithm (named after its discover, Dutch computer scientist E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination with non-negative weight edge.

It turns out that one can find the shortest paths from a given source to *all* vertices (points) in a graph in the same time. Hence, this problem is sometimes called the *single-source shortest paths* problem. Dijkstra's algorithm is a greedy algorithm, which finds shortest path between all pairs of vertices in the graph. Before describing the algorithms formally, let us study the method through an example.

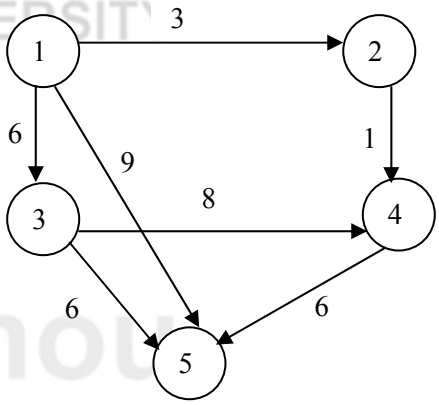


Figure 8.5: A Directed Graph with no negative edge(s)

Dijkstra's algorithm keeps two sets of vertices:

S is the set of vertices whose shortest paths from the source have already been determined

Q = V-S is the set of remaining vertices .

The other data structures needed are:

d array of best estimates of shortest path to each vertex from the source

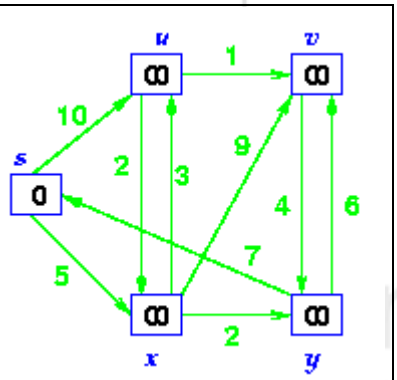
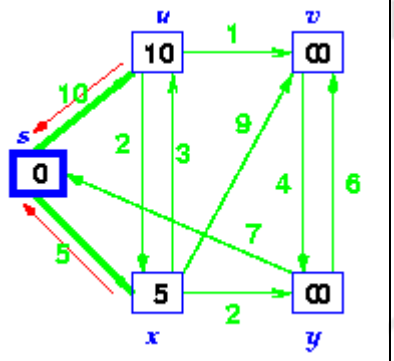
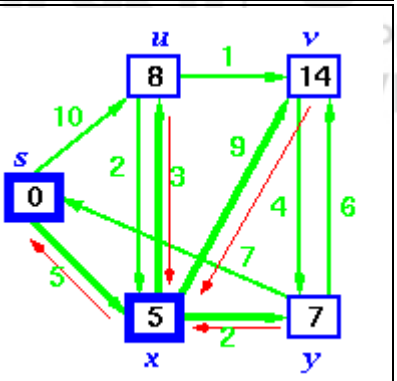
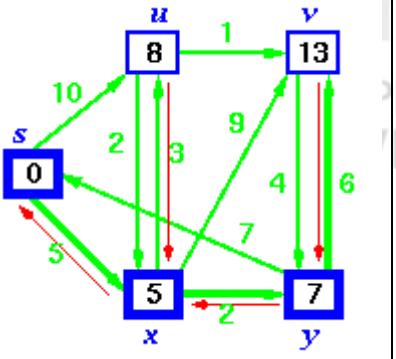
pi an array of predecessors for each vertex. *predecessor* is an array of vertices to which shortest path has already been determined.

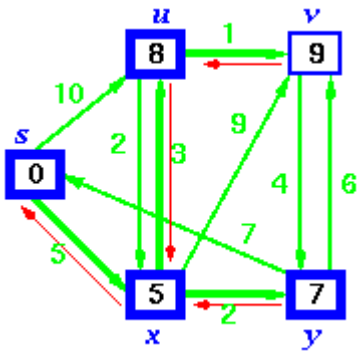
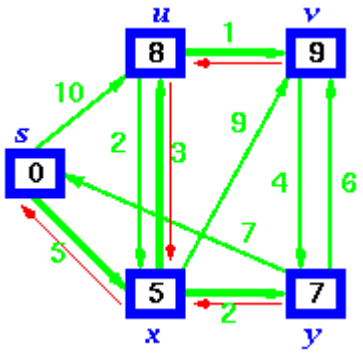
The basic operation of Dijkstra's algorithm is edge relaxation. If there is an edge from u to v, then the shortest known path from s to u can be extended to a path from s to v by adding edge (u,v) at the end. This path will have length d[u]+w(u,v). If this is less than d[v], we can replace the current value of d[v] with the new value.

The predecessor list is an array of indices, one for each vertex of a graph. Each vertex entry contains the index of its predecessor in a path through the graph.

Operation of Algorithm

The following sequence of diagrams illustrate the operation of Dijkstra's Algorithm. The bold vertices indicate the vertex to which shortest path has been determined.

	Initialize the graph, all the vertices have infinite costs except the source vertex which has zero cost
	From all the adjacent vertices, choose the closest vertex to the source **s**. As we initialized d[s] to 0, it's **s**. (shown in bold circle) Add it to **S** Relax all vertices adjacent to **s**, i.e u and x Update vertices u and x by 10 and 5 as the distance from s.
	Choose the nearest vertex, **x**. Relax all vertices adjacent to **x** Update predecessors for **u, v** and **y**. Predecessor of x = s Predecessor of v = x ,s Predecessor of y = x ,s add x to S
	Now **y** is the closest vertex. Add it to **S**. Relax **v** and adjust its predecessor.

	**u** is now closest, add it to S and adjust its adjacent vertex, **v**.
	Finally, add **v to S**. The predecessor list now defines the shortest path from each node to **s**.

Dijkstra's algorithm

```
 * Initialise d and pi*
for each vertex v in V( g )
      g.d[v] := infinity
      g.pi[v] := nil
      g.d[s] := 0;
* Set S to empty *
S := { 0 }
Q := V(g)
* While (V-S) is not null*
while not Empty(Q)
```

1. Sort the vertices in **V-S** according to the current best estimate of their distance from the source
   u := Extract-Min (Q);

2. Add vertex **u**, the closest vertex in **V-S**, to **S**,
   AddNode(S, u);

3. Relax all the vertices still in **V-S** connected to **u**
   relax(Node u, Node v, double w[][])
   if d[v] > d[u] + w[u]v] then
   d[v] := d[u] + w[u][v]
   pi[v] := u

In summary, this algorithm starts by assigning a weight of infinity to all vertices, and then selecting a source and assigning a weight of zero to it. Vertices are added to the set for which shortest paths are known. When a vertex is selected, the weights of its adjacent vertices are relaxed. Once all vertices are relaxed, their predecessor's vertices

are updated (pi). The cycle of selection, weight relaxation and predecessor update is repeated until the shortest path to all vertices has been found.

Complexity of Algorithm

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q. In this case, the running time is $\Theta(n^2)$.

8.3.2 Graphs with Negative Edge costs

We have seen that the above Dijkstra's single source shortest-path algorithm works for graphs with non-negative edges (like road networks). The following two scenarios can emerge out of negative cost edges in a graph:

- Negative edge with non- negative weight cycle reachable from the source.
- Negative edge with non-negative weight cycle reachable from source.

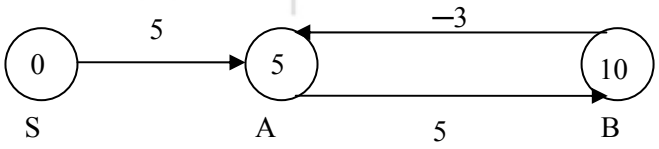


Figure 8.6 : A Graph with negative edge and non-negative weight cycle

The net weight of the cycle is 2(non-negative)(refer to *Figure 8.6*).

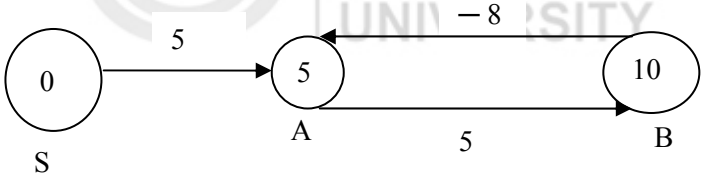


Figure 8.7: A graph with negative edge and negative weight cycle

The net weight of the cycle is −3(negative) (refer to *Figure 8.7*). The shortest path from A to B is not well defined as the shortest path to this vertex are infinite, i.e., by traveling each cycle we can decrease the cost of the shortest path by 3, like (S, A, B) is path (S, A, B, A, B) is a path with less cost and so on.

Dijkstra's Algorithm works only for directed graphs with non-negative weights (cost).

8.3.3 Acyclic Graphs

A path in a directed graph is said to form a cycle is there exists a path (A,B,C,…..P) such that A = P. A graph is called acyclic if there is no cycle in the graph.

8.3.4 All Pairs Shortest Paths Algorithm

In the last section, we discussed about shortest path algorithm which starts with a single source and finds shortest path to all vertices in the graph. In this section, we shall discuss the problem of finding shortest path between all pairs of vertices in a graph. This problem is helpful in finding distance between all pairs of cities in a road atlas. All pairs shortest paths problem is mother of all shortest paths problems.

In this algorithm, we will represent the graph by adjacency matrix.

The weight of an edge C_{ij} in an adjacency matrix representation of a directed graph is represented as follows

$$C_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of the directed edge from i to j i.e (i,j) if } i \neq j \text{ and (i j) belongs to E} \\ \infty & \text{if } i \neq j \text{ and (i, j) does not belong to E} \end{cases}$$

Given a directed graph G = (V, E), where each edge (v, w) has a non-negative cost C(v , w), for all pairs of vertices (v, w) to find the lowest cost path from v to w.

The All pairs shortest paths problem can be considered as a generalisation of single-source-shortest-path problem, by using Dijkstra's algorithm by varying the source node among all the nodes in the graph. If negative edge(s) is allowed, then we can't use Dijkstra's algorithm.

In this section we shall use a recursive solution to all pair shortest paths problem known as Floyd-Warshall algorithm, which runs in $O(n^3)$ time.

This algorithm is based on the following principle. For graph G let V = {1, 2, 3,…,n}.Let us consider a sub set of the vertices {1, 2, 3, …..,k. For any pair of vertices that belong to V, consider all paths from i to j whose intermediate vertices are from {1, 2, 3, ….k}. This algorithm will exploit the relationship between path p and shortest path from i to j whose intermediate vertices are from {1, 2, 3, ….k-1} with the following two possibilities:

1. If k is not an intermediate vertex in the path p, then all the intermediate vertices of the path p are in {1, 2, 3, ….,k-1}. Thus, shortest path from i to j with intermediate vertices in {1, 2, 3, ….,k-1} is also the shortest path from i to j with vertices in {1, 2, 3, …, k}.

2. If k is an intermediate vertex of the path p, we break down the path p into path p1 from vertex i to k and path p2 from vertex k to j. So, path p1 is the shortest path from i to k with intermediate vertices in {1, 2, 3, …,k-1}.

During iteration process we find the shortest path from i to j using only vertices (1, 2, 3, …, k-1} and in the next step, we find the cost of using the k^{th} vertex as an intermediate step. If this results in lower cost, then we store it.

After n iterations (all possible iterations), we find the lowest cost path from i to j using all vertices (if necessary).

Note the following:

Initialize the matrix

C[i][j] = ∞ if (i, j) does not belong to E for graph G = (V, E)

Initially, D[i][j] = C[i][j]

We also define a path matrix P where P[i][j] holds intermediate vertex k on the least cost path from i to j that leads to the shortest path from i to j .

Algorithm (All Pairs Shortest Paths)

N = number of rows of the graph
D[i[j] = C[i][j]
For k from 1 to n
  Do for i = 1 to n
    Do for j = 1 to n
      D[i[j]= minimum($d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$))
    Enddo
  Enddo
Enddo

where $d_{ij}^{(k-1)}$ = minimum path from i to j using k-1 intermediate vertices
where $d_{ik}^{(k-1)}$ = minimum path from j to k using k-1 intermediate vertices
where $d_{kj}^{(k-1)}$ = minimum path from k to j using k-1 intermediate vertices

Program 8.1 gives the program segment for the All pairs shortest paths algorithm.

```
AllPairsShortestPaths(int N, Matrix C, Matrix P, Matrix D)
{
        int i, j, k

        if i = j then C[i][j] = 0
        for ( i = 0; i < N; i++)
        {
                for (j = 0; j < N; j++)
                {
                  D[i][j] = C[i][j];
                P[i][j] = -1;
                }
          D[i][j] = 0;
        }

        for (k=0; k<N; k++)
        {
                for (i=0; i<N; i++)
                {
                        for (j=0; J<N; J++)
                        {
                          if (D[i][k] + D[k][j]  < D[i][j])
                          {
                            D[i][j] = D[i][k] + D[k][j];
                            P[i][j] = k;
                          }
                        }
        }
         }
}

/********** End *************/
```

Program 8.1 : Program segment for All pairs shortest paths algorithm

From the above algorithm, it is evident that it has $O(N^3)$ time complexity.

Shortest path algorithms had numerous applications in the areas of Operations Research, Computer Science, Electrical Engineering and other related areas.

☞ Check Your Progress 2

1) _____ is the basis of Dijkstra's algorithm

2) What is the complexity of All pairs shortest paths algorithm?

……..

8.4 MINIMUM COST SPANNING TREES

A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree (with no cycle). A graph may have many spanning trees.

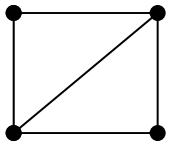


Figure 8.8: A Graph

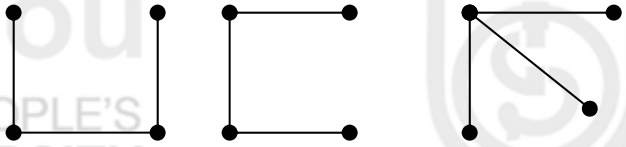


Figure 8.9 : Spanning trees of the Graph of Figure 8.9

Consider the graph of *Figure 8.8*. It's spanning trees are shown in *Figure 8.9*.
Now, if the graph is a weighted graph (length associated with each edge). The weight of the tree is just the sum of weights of its edges. Obviously, different spanning trees have different weights or lengths. Our objective is to find the minimum length (weight) spanning tree.

Suppose, we have a group of islands that we wish to link with bridges so that it is possible to travel from one island to any other in the group. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the minimum cost spanning tree.

8.4.1 Kruskal's Algorithm

Krushkal's algorithm uses the concept of *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it links two trees together. If it forms a cycle, it would simply mean that it links two nodes that were already connected. So, we reject it.

The steps in Kruskal's Algorithm are as follows:

1. The forest is constructed from the graph G - with each node as a separate tree in the forest.
2. The edges are placed in a priority queue.
3. Do until we have added **n**-1 edges to the graph,
    1. Extract the cheapest edge from the queue.
    2. If it forms a cycle, then a link already exists between the concerned nodes. Hence reject it.
    3. Else add it to the forest. Adding it to the forest will join two trees together.

The forest of trees is a partition of the original set of nodes. Initially all the trees have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

Let us see the sequence of operations to find the Minimum Cost Spanning Tree(MST) in a graph using Kruskal's algorithm. Consider the graph of *Figure 8.10., Figure 8.11* shows the construction of MST of graph of *Figure 8.10.*

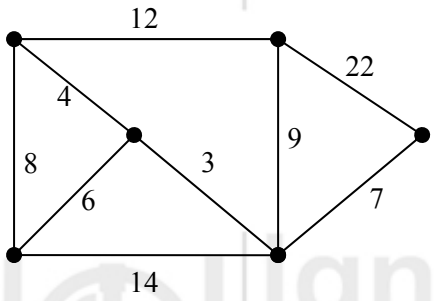
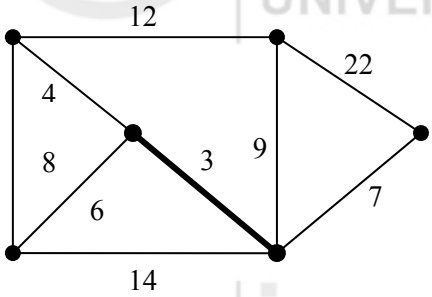


Figure 8.10 : A Graph



Step 1

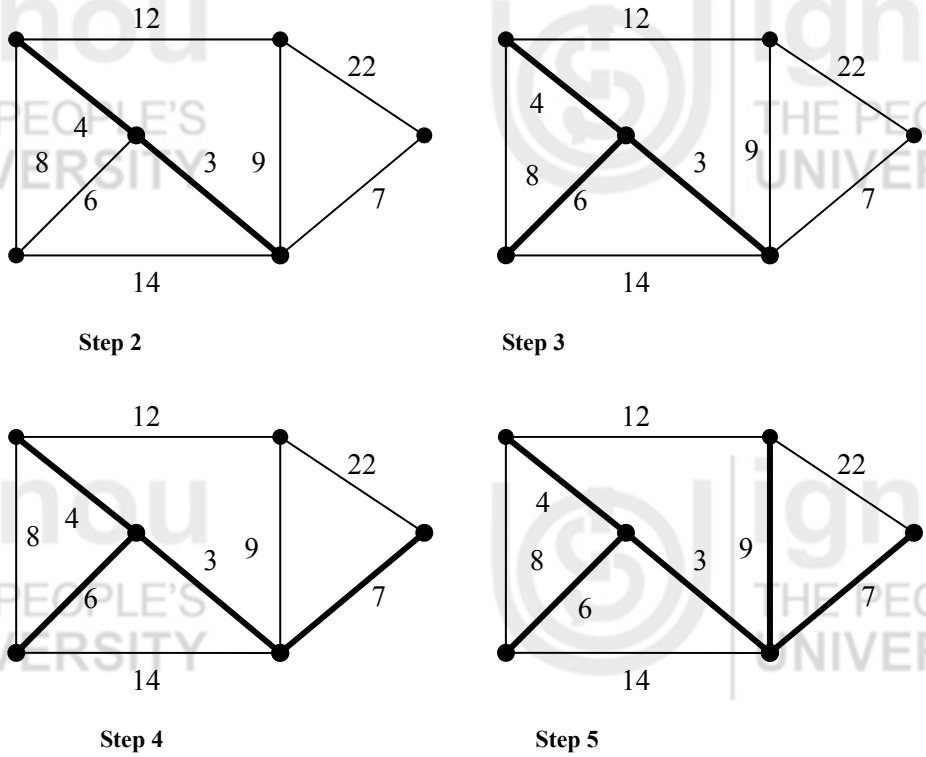


Figure 8.11 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Kruskal's algorithm

The following are various steps in the construction of MST for the graph of *Figure 8.10* using Kruskal's algorithm.

Step 1 : The lowest cost edge is selected from the graph which is not in MST (initially MST is empty). The lowest cost edge is 3 which is added to the MST (shown in bold edges)

Step 2: The next lowest cost edge which is not in MST is added (edge with cost 4).

Step 3 : The next lowest cost edge which is not in MST is added (edge with cost 6).

Step 4 : The next lowest cost edge which is not in MST is added (edge with cost 7).

Step 5 : The next lowest cost edge which is not in MST is 8 but will form a cycle. So, it is discarded . The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

8.4.2 Prim's Algorithm

Prim's algorithm uses the concept of sets. Instead of processing the graph by sorted order of edges, this algorithm processes the edges in the graph randomly by building up disjoint sets.

It uses two disjoint sets A and $\bar{A}$. Prim's algorithm works by iterating through the nodes and then finding the shortest edge from the set A to that of set A (i.e. out side A), followed by the addition of the node to the new graph. When all the nodes are processed, we have a minimum cost spanning tree.

Rather than building a sub-graph by adding one edge at a time, Prim's algorithm builds a tree one vertex at a time.

The steps in Prim's algorithm are as follows:

Let G be the graph with n vertices for which minimum cost spanning tree is to be generated.

Let T be the minimum spanning tree.
Let T be a single vertex x.
while (T has fewer than n vertices)
{
find the smallest edge connecting T to G-T
add it to T
}

Consider the graph of *Figure 8.10. Figure 8.12* shows the various steps involved in the construction of Minimum Cost Spanning Tree of graph of *Figure 8.10.*

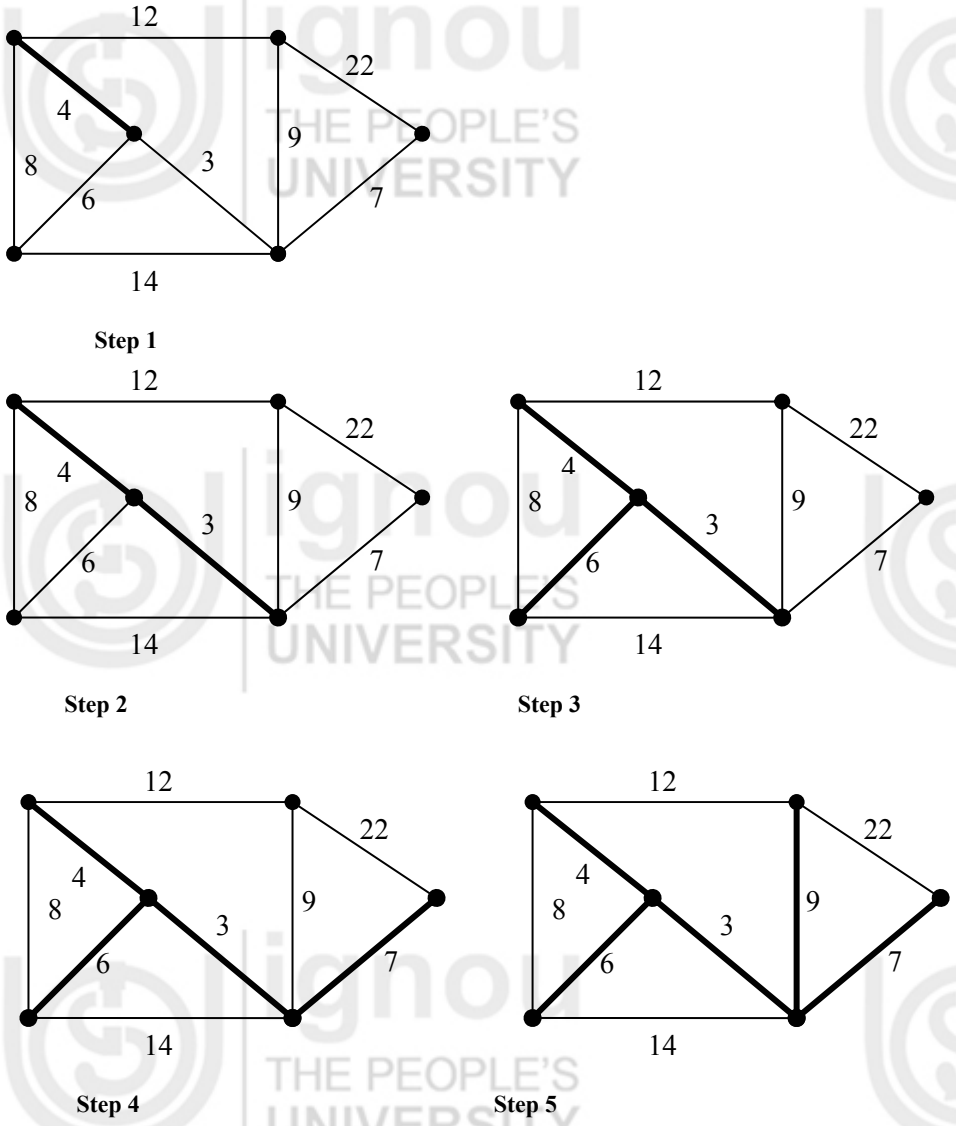


Figure 8.12 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Prim's algorithm

The following are various steps in the construction of MST for the graph of *Figure 8.10* using Prim's algorithm.

Step 1 : We start with a single vertex (node). Now the set A contains this single node and set A contains rest of the nodes. Add the edge with the lowest cost from A to A. The edge with cost 4 is added.

Step 2: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 3) is selected and added to MST.

Step 3: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 6) is selected and added to MST.

Step 4: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 73) is selected and added to MST.

Step 5: The next lowest cost edge to the set not in MST is 8 but forms a cycle. So, it is discarded. The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

Comparison of Kruskal's algorithm and Prim's algorithm

	Kruskal's algorithm	**Prim's algorithm**
Principle	Based on generic minimum cost spanning tree algorithms	A special case of generic minimum cost spanning tree algorithm. Operates like Dijkstra's algorithm for finding shortest path in a graph.
Operation	Operates on a single set of edges in the graph	Operates on two disjoint sets of edges in the graph
Running time	O(E log E) where E is the number of edges in the graph	O(E log V), which is asymptotically same as Kruskal's algorithm

For the above comparison, it may be observed that for dense graphs having more number of edges for a given number of vertices, Prim's algorithm is more efficient.

8.4.3 Applications

The minimum cost spanning tree has wide applications in different fields. It represents many complicated real world problems like:

1. Minimum distance for travelling all cities at most one (travelling salesman problem).
2. In electronic circuit design, to connect n pins by using n-1 wires, using least wire.
3. Spanning tree also finds their application in obtaining independent set of circuit equations for an electrical network.

8.5 BREADTH FIRST SEARCH (BFS)

When BFS is applied, the vertices of the graph are divided into two categories. The vertices, which are visited as part of the search and those vertices, which are not visited as part of the search. The strategy adopted in breadth first search is to start search at a vertex(source). Once you started at source, the number of vertices that are visited as part of the search is 1 and all the remaining vertices need to be visited. Then, search the vertices which are adjacent to the visited vertex from left to order. In this way, all the vertices of the graph are searched.

Consider the digraph of *Figure 8.13*. Suppose that the search started from S. Now, the vertices (from left to right) adjacent to S which are not visited as part of the search are B, C, A. Hence, B,C and A are visited after S as part of the BFS. Then, F is the unvisited vertex adjacent to B. Hence, the visit to B, C and A is followed by F. The unvisited vertex adjacent of C is D. So, the visit to F is followed by D. There are no

unvisited vertices adjacent to A. Finally, the unvisited vertex E adjacent to D is visited.

Hence, the sequence of vertices visited as part of BFS is S, B, C, A, F, D and E.

8.6 DEPTH FIRST SEARCH (DFS)

The strategy adopted in depth first search is to search deeper whenever possible. This algorithm repeatedly searches deeper by visiting unvisited vertices and whenever an unvisited vertex is not found, it backtracks to previous vertex to find out whether there are still unvisited vertices.

As seen, the search defined above is inherently recursive. We can find a very simple recursive procedure to visit the vertices in a depth first search. The DFS is more or less similar to pre-order tree traversal. The process can be described as below:

Start from any vertex (source) in the graph and mark it visited. Find vertex that is adjacent to the source and not previously visited using adjacency matrix and mark it visited. Repeat this process for all vertices that is not visited, if a vertex is found visited in this process, then return to the previous step and start the same procedure from there.

If returning back to source is not possible, then DFS from the originally selected source is complete and start DFS using any unvisited vertex.

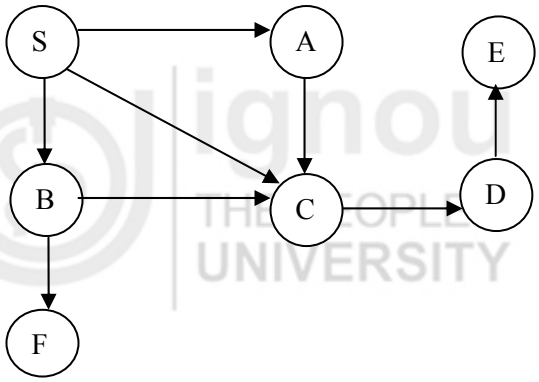


Figure 8.13 : A Digraph

Consider the digraph of *Figure 8.13*. Start with S and mark it visited. Then visit the next vertex A, then C and then D and at last E. Now there are no adjacent vertices of E to be visited next. So, now, backtrack to previous vertex D as it also has no unvisited vertex. Now backtrack to C, then A, at last to S. Now S has an unvisited vertex B. Start DFS with B as a root node and then visit F. Now all the nodes of the graph are visited.

Figure 8.14 shows a DFS tree with a sequence of visits. The first number indicates the time at which the vertex is visited first and the second number indicates the time at which the vertex is visited during back tracking.

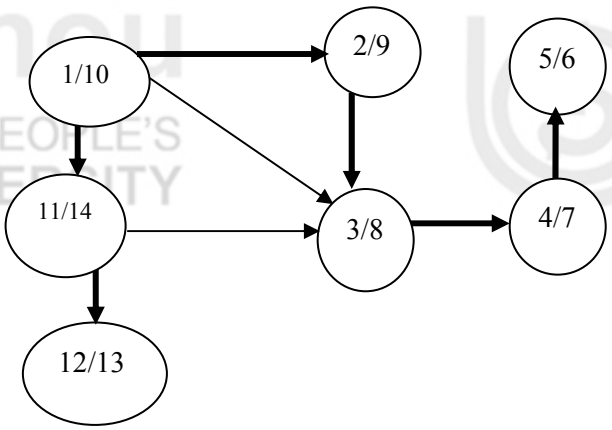


Figure 8.14 : DFS tree of digraph of Figure 8.13

The DFS forest is shown with shaded arrow in *Figure 8.14*.

Algorithm for DFS

Step 1: Select a vertex in the graph and make it the source vertex and mark it visited.

Step 2: Find a vertex that is adjacent to the souce vertex and start a new search if it is not already visited.

Step 3: Repeat step 2 using a new source vertex. When all adjacent vertices are visited, return to previous source vertex and continue search from there.

If n is the number of vertices in the graph and the graph is represented by an adjacency matrix, then the total time taken to perform DFS is $O(n^2)$. If G is represented by an adjacency list and the number of edges of G are e, then the time taken to perform DFS is O(e).

8.7 FINDING STRONGLY CONNECTED COMPONENTS

A beautiful application of DFS is finding a strongly connected component of a graph.

Definition: For graph G = (V, E) , where V is the set of vertices and E is the set of edges, we define a strongly connected components as follows:

U is a sub set of V such that u, v belongs to U such that, there is a path from u to v and v to u. That is, all pairs of vertices are reachable from each other.

In this section we will use another concept called transpose of a graph. Given a directed graph G a transpose of G is defined as G^T. G^T is defined as a graph with the same number of vertices and edges with only the direction of the edges being reversed. G^T is obtained by transposing the adjacency matrix of the directed graph G.

The algorithm for finding these strongly connected components uses the transpose of G, G^T.

G = (V, E), G^T = (V, E^T), where E^T = { (u, v): (v, u) belongs to E }

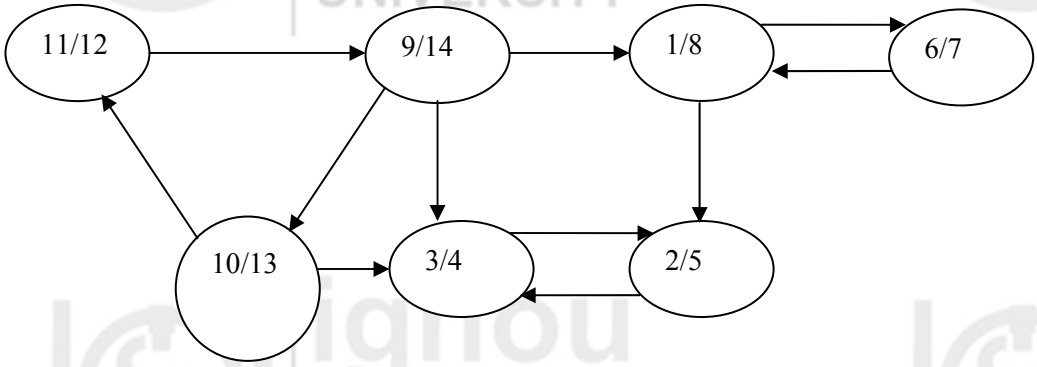


Figure 8.15: A Digraph

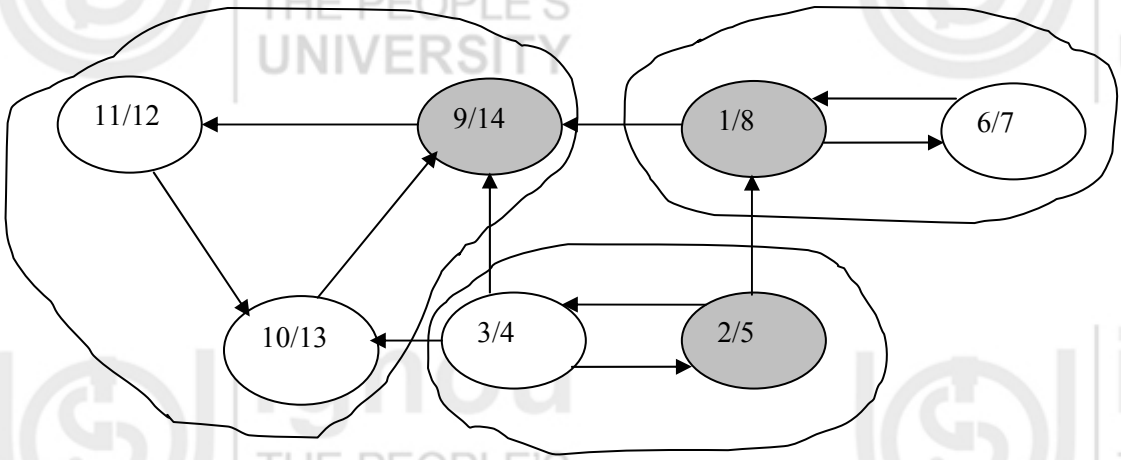


Figure 8.16: Transpose and strongly connected components of digraph of Figure 8.15

Figure 8.15 shows a directed graph with sequence in DFS (first number of the vertex shows the discovery time and second number shows the finishing time of the vertex during DFS. *Figure 8.16* shows the transpose of the graph in *Figure 8.15* whose edges are reversed. The strongly connected components are shown in zig-zag circle in *Figure 8.16*.

To find strongly connected component we start with a vertex with the highest finishing time and start DFS in the graph G^T and then in decreasing order of finishing time. DFS with vertex with finishing time 14 as root finds a strongly connected component. Similarly, vertices with finishing times 8 and then 5, when selected as source vertices also lead to strongly connected components.

Algorithm for finding strongly connected components of a Graph:

Strongly Connected Components (G)

where d[u] = discovery time of the vertex u during DFS , f[u] = finishing time of a vertex u during DFS, G^T = Transpose of the adjacency matrix

Step 1: Use DFS(G) to compute f[u] $\forall u \in V$
Step 2: Compute G^T
Step 3: Execute DFS in G^T

Step 4: Output the vertices of each tree in the depth-first forest of Step 3 as a separate strongly connected component.

☞ Check Your Progress 3

1) Which graph traversal uses a queue to hold vertices that are to be processed next ?

……

……

2) Which graph traversal is recursive by nature?

……

……

3) For a dense graph, Prim's algorithm is faster than Kruskal's algorithm

True/False

4) Which graph traversal technique is used to find strongly connected component of a graph?

……

……

8.8 SUMMARY

Graphs are data structures that consist of a set of vertices and a set of edges that connect the vertices. A graph where the edges are directed is called directed graph. Otherwise, it is called an undirected graph. Graphs are represented by adjacency lists and adjacency matrices. Graphs can be used to represent a road network where the edges are weighted as the distance between the cities. Finding the minimum distance between single source and all other vertices is called single source shortest path problem. Dijkstra's algorithm is used to find shortest path from a single source to every other vertex in a directed graph. Finding shortest path between every pair of vertices is called all pairs shortest paths problem.

A spanning tree of a graph is a tree consisting of only those edges of the graph that connects all vertices of the graph with minimum cost. Kruskal's and Prim's algorithms find minimum cost spanning tree in a graph. Visiting all nodes in a graph systematically in some manner is called traversal. Two most common methods are depth-first and breadth-first searches.

8.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) an acyclic
2) symmetric
3) The adjacency matrix of the directed graph and undirected graph are as follows:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

(Refer to Figure 8.3)

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

(Refer to Figure 8.3)

Check Your Progress 2

1) Node relaxation
2) $O(N^3)$

Check Your Progress 3

1) BFS
2) DFS
3) True
4) DFS

8.10 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.
3. *Data Structures and Algorithms* by Alfred V.Aho; Addison Wesley.

Reference Websites

http://www.onesmartclick.com/engineering/data-structure.html
http://msdn.microsoft.com/vcsharp/programming/datastructures/
http://en.wikipedia.org/wiki/Graph_theory

UNIT 9 SEARCHING AND SORTING TECHNIQUES

Structure **Page Nos.**

9.0 INTRODUCTION

Searching is the process of looking for something: Finding one piece of data that has been stored within a whole group of data. It is often the most time-consuming part of many computer programs. There are a variety of methods, or algorithms, used to search for a data item, depending on how much data there is to look through, what kind of data it is, what type of structure the data is stored in, and even where the data is stored - inside computer memory or on some external medium.

Till now, we have studied a variety of data structures, their types, their use and so on. In this unit, we will concentrate on some techniques to ***search*** a particular data or piece of information from a large amount of data. There are basically two types of searching techniques, ***Linear or Sequential Search and Binary Search.***

Searching is very common task in day-to-day life, where we are involved some or other time, in searching either for some needful at home or office or market, or searching a word in dictionary. In this unit, we see that if the things are organised in some manner, then search becomes efficient and fast.

All the above facts apply to our computer programs also. Suppose we have a telephone directory stored in the memory in an array which contains Name and Numbers. Now, what happens if we have to find a number? The answer is search that number in the array according to name (given). If the names were organised in some order, searching would have been fast.

So, basically a search algorithm is an algorithm which accepts an argument 'a' and tries to find the corresponding data where the match of 'a' occurs in a file or in a table.

9.1 OBJECTIVES

After going through this unit, you should be able to:

- know the basic concepts of searching;
- know the process of performing the Linear Search;

- know the process of performing the Binary Search and
- know the applications of searching.

9.2 LINEAR SEARCH

Linear search is not the most efficient way to search for an item in a collection of items. However, it is very simple to implement. Moreover, if the array elements are arranged in random order, it is the only reasonable way to search. In addition, efficiency becomes important only in large arrays; if the array is small, there aren't many elements to search and the amount of time it takes is not even noticed by the user. Thus, for many situations, linear search is a perfectly valid approach.

Before studying Linear Search, let us define some terms related to search.

A ***file*** is a collection of records and a record is in turn a collection of fields. A field, which is used to differentiate among various records, is known as a ***'key'***.

For example, the telephone directory that we discussed in previous section can be considered as a file, where each record contains two fields: name of the person and phone number of the person.

Now, it depends on the application whose field will be the 'key'. It can be the name of person (usual case) and it can also be phone number. We will locate any particular record by matching the input argument 'a' with the key value.

The simplest of all the searching techniques is *Linear or Sequential Search.* As the name suggests, all the records in a file are searched sequentially, one by one, for the matching of key value, until a match occurs.

The Linear Search is applicable to a table which it should be organised in an array. Let us assume that a file contains 'n' records and a record has 'a' fields but only one key. The values of key are organised in an array say 'm'. As the file has 'n' records, the size of array will be 'n' and value at position R(i) will be the key of record at position i. Also, let us assume that *'el'* is the value for which search has to be made or it is the search argument.

Now, let us write a simple algorithm for Linear Search.

Algorithm

Here, m represents the unordered array of elements
n represents number of elements in the array and
el represents the value to be searched in the list

```
Sep 1: [Initialize]
        k=0
        flag=1

Step 2: Repeat step 3 for k=0,1,2…..n-1

Step 3: if (m[k]=el )
         then
                  flag=0
                 print "Search is successful" and element is found at location (k+1)
                 stop
         endif
```

Step 4: if (flag=1) then
print "Search is unsuccessful"

endif

Step 5: stop

Program 9.1 gives the program for Linear Search.

```
/*Program for Linear Search*/
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Global Variables*/
int search;
int flag;
/*Function Declarations*/
int input (int *, int, int);
void linear_search (int *, int, int);
void display (int *, int);
/*Functions */
void linear_search(int m[ ], int n, int el)
{
        int k;
        flag = 1;
        for(k=0; k<n; k++)
        {
            if(m[k]==el
            {
                printf("\n Search is Successful\n");
                printf("\n Element : %i Found at location : %i", element, k+1);
                flag = 0;
            }
        }
        if(flag==1)
            printf("\n Search is unsuccessful");
}
void display(int m[ ], int n)
{
        int i;
        for(i=0; i< 20; i++)
        {
            printf("%d", m[i];
        }
}
int input(int m[ ], int n, int el)
{
        int i;
        n = 20;
        el = 30;
        printf("Number of elements in the list : %d", n);
        for(i=0;i<20;i++)
            {
              m[i]=rand( )%100;
            }
        printf("\n Element to be searched :%d", el);
        search = el;
        return n;
        }
        /* Main Function*/
```

```
void main( )
{
        int n, el, m[200];
        number = input(m, n,el);
        el = search;
        printf("\n Entered list as follows: \n");
        display(m, n);
        linear_search(m, n, el);
        printf("\n In the following list\n");
        display(m, n);
}
```

Program 9.1: Linear Search

Program 9.1 examines each of the key values in the array 'm', one by one and stops when a match occurs or the total array is searched.

Example:

A *telephone directory* with n = 10 records and Name field as key. Let us assume that the names are stored in array 'm' i.e. m(0) to m(9) and the search has to be made for name "Radha Sharma", i.e. element = "Radha Sharma".

Telephone Directory

Name	***Phone No.***
Nitin Kumar	25161234
Preeti Jain	22752345
Sandeep Singh	23405678
Sapna Chowdhary	22361111
Hitesh Somal	24782202
R.S.Singh	26254444
Radha Sharma	26150880
S.N.Singh	25513653
Arvind Chittora	26252794
Anil Rawat	26257149

The above algorithm will search for element = "Radha Sharma" and will stop at 6th index of array and the required phone number is "26150880", which is stored at position 7 i.e. 6+1.

Efficiency of Linear Search

How many number of comparisons are there in this search in searching for a given element?

The number of comparisons depends upon where the record with the argument key appears in the array. If record is at the first place, number of comparisons is '1', if record is at last position 'n' comparisons are made.

If it is equally likely for that the record can appear at any position in the array, then, a successful search will take (n+1)/2 comparisons and an unsuccessful search will take 'n' comparisons.

In any case, the order of the above algorithm is ***O(n)***.

☞ Check Your Progress 1

1) Linear search uses an exhaustive method of checking each element in the array against a key value. When a match is found, the search halts. Will sorting the array before using the linear search have any effect on its order of efficiency?

……

2) In a best case situation, the element was found with the fewest number of comparisons. Where, in the list, would the key element be located?

……

9.3 BINARY SEARCH

An unsorted array is searched by *linear search* that scans the array elements one by one until the desired element is found.

The reason for sorting an array is that we search the array "quickly". Now, if the array is sorted, we can employ *binary search*, which brilliantly halves the size of the search space each time it examines one array element.

An array-based binary search selects the middle element in the array and compares its value to that of the key value. Because, the array is sorted, if the key value is less than the middle value then the key must be in the first half of the array. Likewise, if the value of the key item is greater than that of the middle value in the array, then it is known that the key lies in the second half of the array. In either case, we can, in effect, "throw out" one half of the search space or array with only one comparison.

Now, knowing that the key must be in one half of the array or the other, the binary search examines the mid value of the half in which the key must reside. The algorithm thus narrows the search area by half at each step until it has either found the key data or the search fails.

As the name suggests, binary means two, so it divides an array into *two* halves for searching. This search is applicable only to an ***ordered table*** (in either ascending or in descending order).

Let us write an algorithm for Binary Search and then we will discuss it. The array consists of elements stored in ascending order.

Algorithm

Step 1: Declare an array 'k' of size 'n' i.e. k(n) is an array which stores all the keys of a file containing 'n' records

Step 2: I← 0

Step 3: low←0, high←n-1

Step 4: while (low <= high)do
mid = (low + high)/2
if (key=k[mid]) then
write "record is at position", mid+1 //as the array starts from the 0^{th} position
else
if(key < k[mid]) then
high = mid - 1

```
                    else
                            low = mid + 1
                    endif
                endif
            endwhile
```

Step 5: Write “Sorry, key value not found”

Step 6: Stop

Program 9.2 gives the program for Binary Search.

```
/*Header Files*/
#include<stdio.h>
#include<conio.h>
/*Functions*/
void binary_search(int array[ ], int value, int size)
{
        int found=0;
        int high=size-1, low=0, mid;
        mid = (high+low)/2;
        printf(“\n\n Looking for %d\n”, value);
        while((!found)&&(high>=low))
        {
                printf(“Low %d Mid%d High%d\n”, low, mid, high);
                if(value==array[mid] )
                {printf(“Key value found at position %d”,mid+1);
                 found=1;
                }
                else
                {if (value<array[mid])
                        high = mid-1;
                else
                        low = mid+1;
                mid = (high+low)/2;
                }
        }
        if (found==1
        printf(“Search successful”);
        else
        printf(“Key value not found”);
}
/*Main Function*/
void main(void)
{
        int array[100], i;
        /*Inputting Values to Array*/
        for(i=0;i<100;i++)
          { printf(“Enter the name:”);
           scanf(“%d”, array[i]);
          }
        printf(“Result of search %d\n”, binary_searchy(array,33,100));
        printf(“Result of search %d\n”, binary_searchy(array, 75,100));
        printf(“Result of search %d\n”, binary_searchy(array,1,100));
}
```

Program 9.2 : Binary Search

Example:

Let us consider a file of 5 records, i.e., n = 5
And k is a sorted array of the keys of those 5 records.

k	
11	0
22	1
33	2
44	3
55	4

Let key = 55, low = 0, high = 4

Iteration 1: mid = (0+4)/2 = 2
k(mid) = k (2) = 33
Now key > k (mid)
So low = mid + 1 = 3
Iteration 2: low = 3, high = 4 (low <= high)
Mid = 3+4 / 2 = 3.5 ~ 3 (integer value)
Here key > k (mid)
So low = 3+1 = 4
Iteration 3: low = 4, high = 4 (low<= high)
Mid = (4+4)/2 = 4
Here key = k(mid)

So, the record is at mid+1 position, i.e., 5

Efficiency of Binary Search

Each comparison in the binary search reduces the number of possible candidates where the key value can be found by a factor of 2 as the array is divided in two halves in each iteration. Thus, the maximum number of key comparisons are approximately log n. So, the order of binary search is ***O (log n).***

Comparative Study of Linear and Binary Search

Binary search is lots faster than linear search. Here are some comparisons:

NUMBER OF ARRAY ELEMENTS EXAMINED

array size	linear search (avg. case)	binary search (worst case)
8	4	4
128	64	8
256	128	9
1000	500	11
100,000	50,000	18

A ***binary search*** on an array is $O(log_2 n)$ because at each test, you can "throw out" one half of the search space or array whereas a ***linear search*** on an array is $O(n)$.

It is noteworthy that, for very small arrays a ***linear search*** can prove faster than a ***binary search***. However, as the size of the array to be searched increases, the binary

search is the clear winner in terms of number of comparisons and therefore overall speed.

Still, the binary search has some drawbacks. First, it requires that the data to be searched be in sorted order. If there is even one element out of order in the data being searched, it can throw off the entire process. When presented with a set of unsorted data, the efficient programmer must decide whether to sort the data and apply a binary search or simply apply the less-efficient linear search. Is the cost of sorting the data is worth the increase in search speed gained with the binary search? If you are searching only once, then it is probably to better do a linear search in most cases.

☞ Check Your Progress 2

1) State True or False
   a. The order of linear search in worst case is O (n/2) True/False
   b. Linear search is more efficient than Binary search. True/False
   c. For Binary search, the array has to be sorted in ascending order only. True/False
2) Write the Binary search algorithm where the array is sorted in descending order.
   ……
   ……

9.4 APPLICATIONS

The searching techniques are applicable to a number of places in today's world, may it be Internet, search engines, on line enquiry, text pattern matching, finding a record from database, etc.

The most important application of searching is to track a particular record from a large file, efficiently and faster.

Let us discuss some of the *applications of Searching* in the world of computers.

1. Spell Checker

This application is generally used in ***Word Processors***. It is based on a program for checking spelling, which it checks and searches sequentially. That is, it uses the concept of *Linear Search*. The program looks up a word in a list of words from a dictionary. Any word that is found in the list is assumed to be spelled correctly. Any word that isn't found is assumed to be spelled wrong.

2. Search Engines

Search engines use software robots to survey the Web and build their databases. Web documents are retrieved and indexed using keywords. When you enter a query at a search engine website, your input is checked against the search engine's keyword indices. The best matches are then returned to you as hits. For checking, it uses any of the Search algorithms.

Search Engines use software programs known as robots, spiders or crawlers. A robot is a piece of software that automatically follows hyperlinks from one document to the next around the Web. When a robot discovers a new site, it sends information back to its main site to be indexed. Because Web documents are one of the least static forms of publishing (i.e., they change a lot), robots also update previously catalogued sites. How quickly and comprehensively they carry out these tasks vary from one search engine to the next.

3. String Pattern matching

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit, search and transport documents over the Internet, and to display documents on printers and computer screens. Web 'surfing' and Web searching are becoming significant and important computer applications, and many of the key computations in all of this document processing involves character strings and string pattern matching. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing. This is accomplished using ***trie*** data structure, which is a tree-based structure that allows for faster searching in a collection of strings.

9.5 INTERNAL SORTING

In internal sorting, all the data to be sorted is available in the high speed main memoryof the computer. We will study the following methods of internal sorting:

1. Insertion sort
2. Bubble sort
3. Quick sort
4. Two-way Merge sort
5. Heap sort

9.5.1 Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example (refer to *Figure 9.1*) before presenting the formal algorithm.

Example : Sort the following list using the insertion sort method:

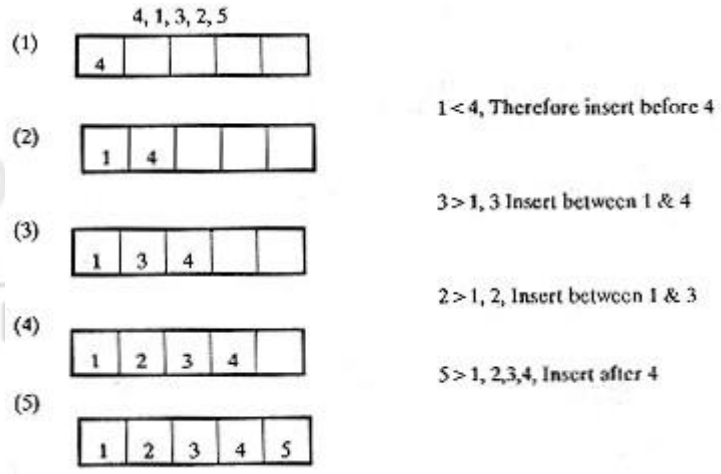


Figure 9.1 : Insertion sort

Thus to find the correct position search the list till an item just greater than the targetis found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

9.5.2 Bubble Sort

In this sorting algorithm, multiple swappings take place in one pass. Smaller elements move or 'bubble' up to the top of the list, hence the name given to the algorithm.

In this method, adjacent members of the list to be sorted are compared.If the item on top is greater than the item immediately below it, then they are swapped. This processis carried on till the list is sorted.

The detailed algorithm follows:

Algorithm: BUBBLE SORT

1. Begin
2. Read the n elements
3. for i=1 to n
for j=n downto i+1
if a[j] <= a[j-1]
swap(a[j],a[j-1])
4. End // of Bubble Sort

Total number of comparisons in Bubble sort :

= (N-1) +(N-2) . . . + 2 + 1

= (N-1)*N / 2 =$O(N^2)$

This inefficiency is due to the fact that an item moves only to the next position in each pass.

9.5.3 Quick Sort

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases. Thebasis of quick sort is the *divide and conquer* strategy i.e. Divide the problem [list to besorted] into sub-problems [sub-lists], until solved sub problems [sorted sub-lists] are found. This is implemented as follows:

Choose one item A[I] from the list A[ ].

Rearrange the list so that this item is in the proper position, i.e., all preceding itemshave a lesser value and all succeeding items have a greater value than this item.

1. Place A[0], A[1] .. A[I-1] in sublist 1
2. A[I]
3. Place A[I + 1], A[I + 2] ... A[N] in sublist 2

Repeat steps 1 & 2 for sublist1 & sublist2 till A[ ] is a sorted list.As can be seen, this

algorithm has a recursive structure.

The *divide'* procedure is of utmost importance in this algorithm. This is usually implemented as follows:

1. Choose A[I] as the dividing element.
2. From the left end of the list (A[O] onwards) scan till an item A[R] is found whose value is greater than A[I].

3. From the right end of list [A[N] backwards] scan till an item A[L] is foundwhose value is less than A[1].

4. Swap A[R] & A[L].

5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.

6. At this point, sublist1 & sublist are ready.

7. Now do the same for each of sublist1 & sublist2.

Program 9.3 gives the program segment for Quick sort. It uses recursion.

```
Quicksort(A,m,n)int
A[ ],m,n
{
        int i, j, k;if
        m<n
        {
                i=m;
                j=n+1;
                k=A[m];
                do
                    do
                    ++i;
                    while (A[i] < k);
                  do
                    --j;
                    while (A[j] > k);
                    if (i < j)
                    {
                        temp = A[i];
                        A[i] = A[j];
                        A[j] = temp;
                    }
                    while (i<j);
 temp = A[m];
 A[m] = A[j];
 A[j] = temp;
 Quicksort(A,m,j-1);
 Quicksort(A,j+1,n);
}
```

Program 9.3 : Quick Sort

The Quick sort algorithm uses the $O(N \log_2 N)$ comparisons on average. The performance can be improved by keeping in mind the following points.

1. Switch to a faster sorting scheme like insertion sort when the sublist size becomes comparatively small.

2. Use a better dividing element in the implementations.

It is also possible to write the non-recursive Quick sort algorithm.

9.5.4 2-Way Merge Sort

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea in this is to divide the list into a number of sublists, sort each of these sublists and merge them to get a single sorted list. The illustrative implementation of 2 way mergesort sees the input initially as n lists of size 1. These are merged to get n/2 lists of size 2. These n/2 lists are merged pair wise and so on till a single list is obtained. This canbe better understood by the following example. This is also called *Concatenate sort. Figure 9.2* depicts 2-way merge sort.

Mergesort is the best method for sorting linked lists in random order. The totalcomputing time is of the $0(n \log_2 n)$.

The disadvantage of using mergesort is that it requires two arrays of the same size andspace for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

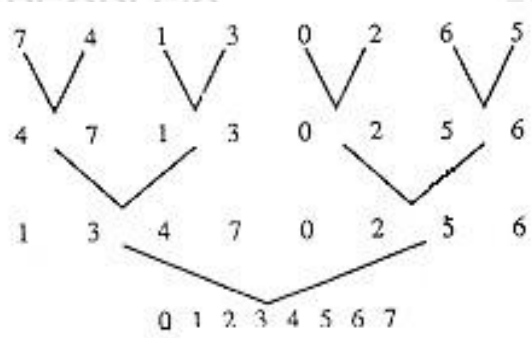


Figure 9.2: 2-way merge sort

Mergesort is the best method for sorting linked lists in random order. The total computing time is of the $0(n \log_2 n)$.

The disadvantage of using mergesort is that it requires two arrays of the same size and space for the merge phase. That is, to sort a list of size n, it needs space for 2n elements.

9.5.5 Heap Sort

We will begin by defining a new structure called *Heap*. *Figure 9.3* illustrates aBinary tree.

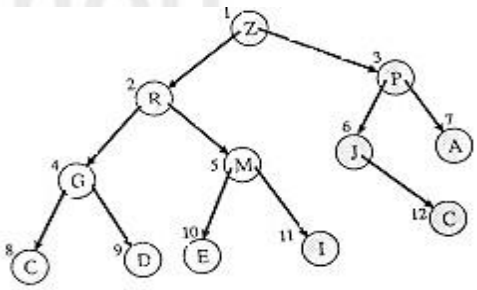


Figure 9.3: A Binary Tree

A complete binary tree is said to satisfy the 'heap condition' if the key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value.

Trees can be represented as arrays, by first numbering the nodes (starting from the root) from left to right. The key values of the nodes are then assigned to array positions whose index is given by the number of the node. For the example tree, thecorresponding array is depicted in *Figure 9.4.*

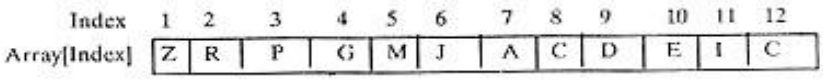


Figure 9.4: Array for the binary tree of figure 10.3

The relationships of a node can also be determined from this array representation. If a node is at position j, its children will be at positions 2j and 2j + 1. Its parent will be at position ⌊J/2⌋.

Consider the node M. It is at position 5. Its parent node is, therefore, at position ⌊5/2⌋ = 2 i.e. the parent is R. Its children are at positions 2 × 5 & (2 × 5) + 1, i.e.10 + 11 respectively i.e. E & I are its children.

A *Heap* is a complete binary tree, in which each node satisfies the heap condition, represented as an array.

We will now study the operations possible on a heap and see how these can becombined to generate a sorting algorithm.

The operations on a heap work in 2 steps.

5. The required node is inserted/deleted/or replaced.
6. The above operation may cause violation of the heap condition so the heap is traversed and modified to rectify any such violations.

Examples: Consider the insertion of a node R in the heap 1.

1. Initially R is added as the right child of J and given the number 13.
2. But, R > J. So, the heap condition is violated.
3. Move R upto position 6 and move J down to position 13.
4. R > P. Therefore, the heap condition is still violated.
5. Swap R and P.
4. The heap condition is now satisfied by all nodes to get the heap of *Figure 9.5.*

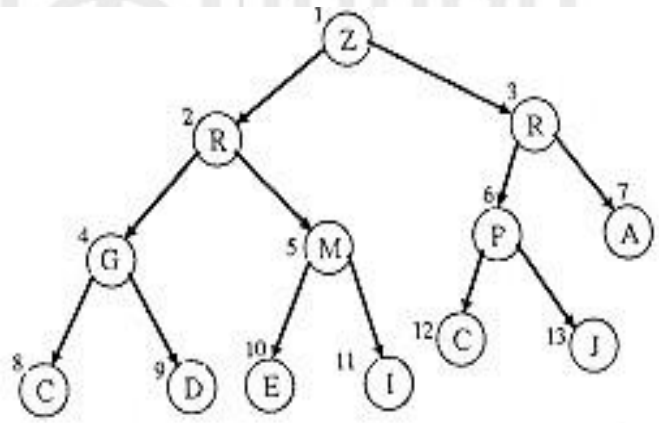


Figure 9.5: A Heap

This algorithm is guaranteed to sort n elements in ($n \log_2 n$) time.

We will first see two methods of heap construction and then removal in order from theheap to sort the list.

1. Top down heap construction

   - Insert items into an initially empty heap, satisfying the heap condition at all steps.

2. Bottom up heap construction

   - Build a heap with the items in the order presented.
   - From the right most node modify to satisfy the heap condition.

We will exemplify this with an example.

Example: Build a heap of the following using top down approach for heap construction.

PROFESSIONAL

Figure 9.6 shows different steps of the top down construction of the heap.

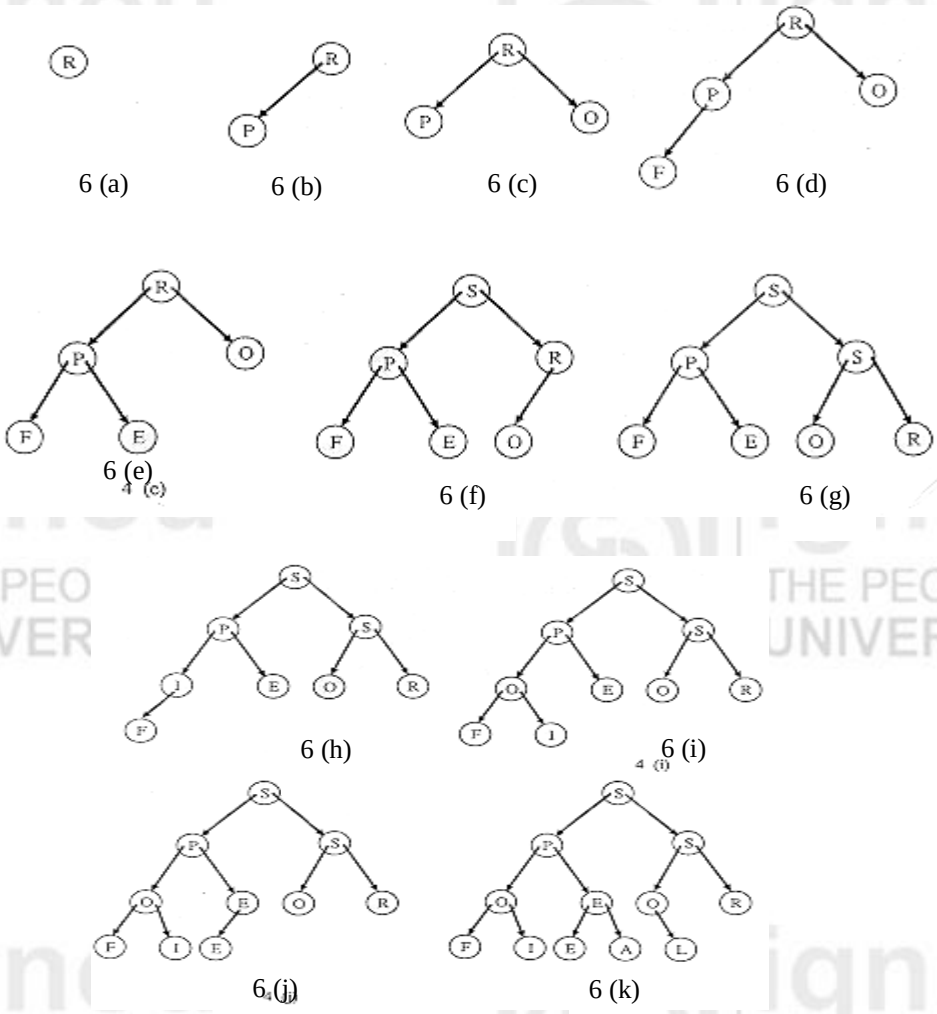


Figure 9.6: Heap Sort (Top down Construction)

Example: The input file is (2,3,81,64,4,25,36,16,9, 49). When the file is interpreted asa binary tree, it results in *Figure 9.7*. *Figure 9.8* depicts the heap.

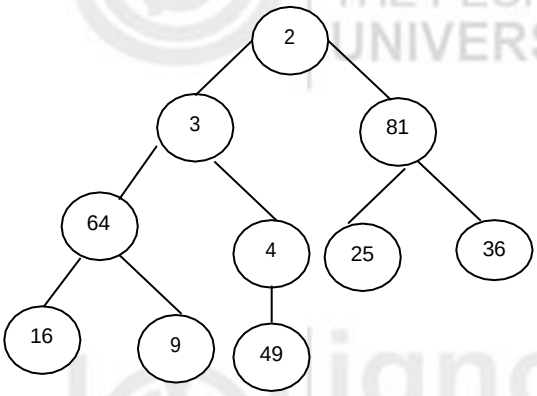


Figure 9.7: A Binary tree 9.7

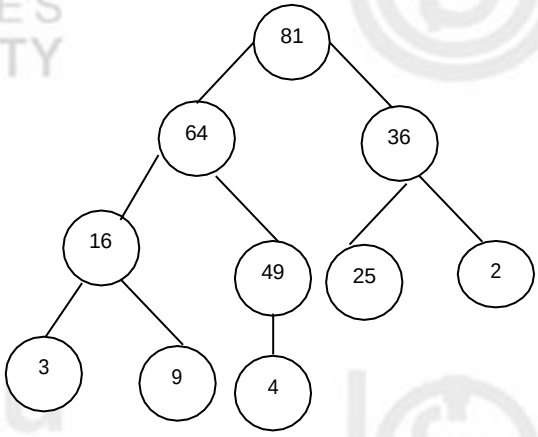
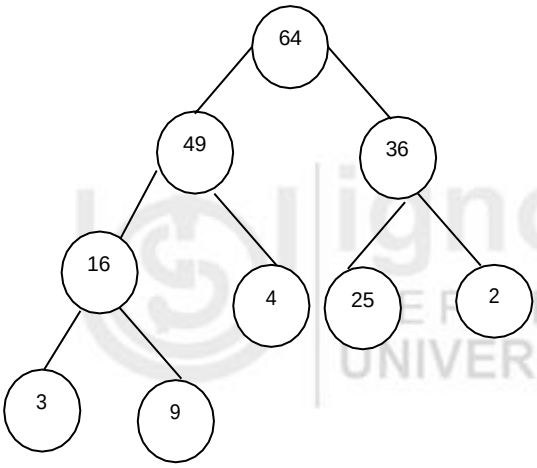
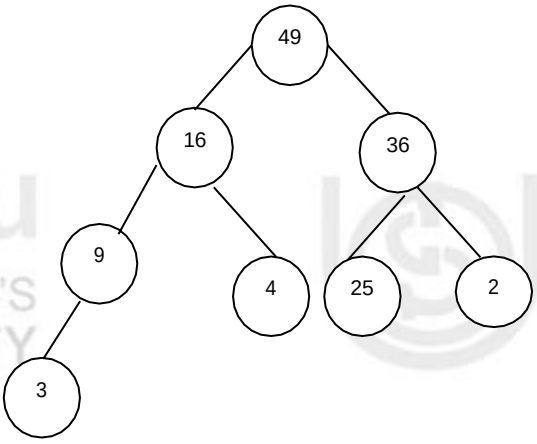


Figure 9.8: Heap of figure

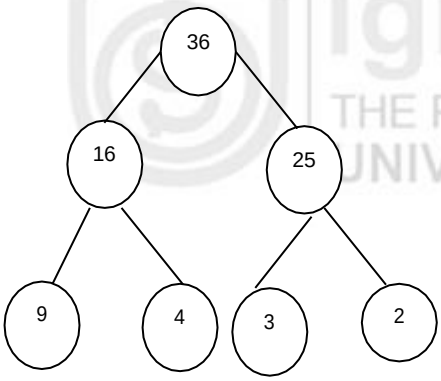
Figure 9.9 illustrates various steps of the heap of *Figure 9.8* as the sorting takesplace.



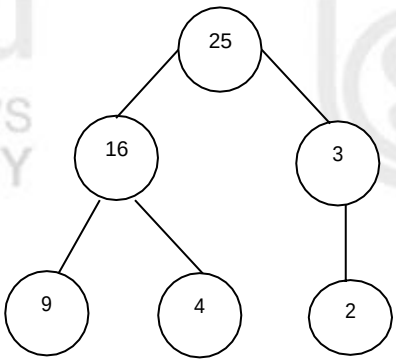
Sorted: 81
Heap size: 9



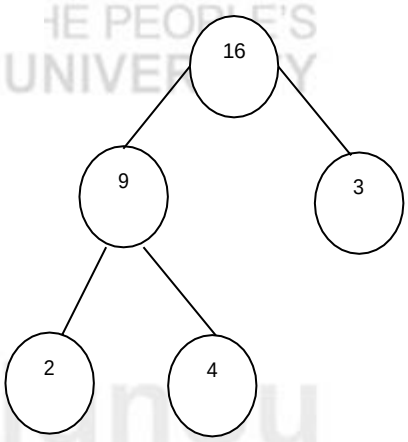
Sorted: 81,64
Heap size: 8



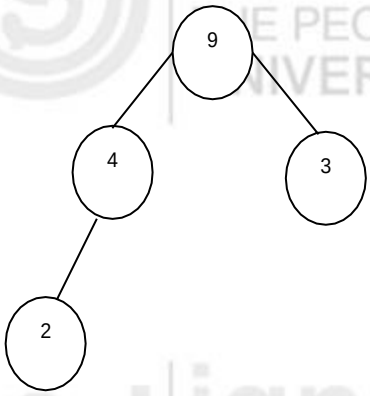
Sorted: 81,64,49
Heap size:7



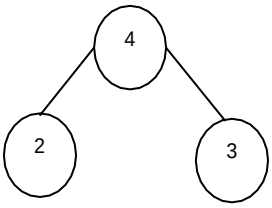
Sorted:81,64,49,36
Heap size:6



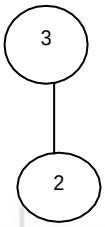
Sorted: 81, 64, 49, 36, 25
Size: 5



Sorted:81, 64, 49, 36, 25, 16
Size:4



Sorted: 81, 64, 49, 36, 25, 16, 9
Size: 3



Sorted:81, 64, 49, 36, 25, 16, 9, 4
Size: 2

Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3
Size : 1

Sorted: 81, 64, 49, 36, 25, 16, 9, 4, 3, 2
Result

Figure 9.9 : Various steps of figure 10.8 for a sorted file

9.6 SORTING ON SEVERAL KEYS

So far, we have been considering sorting based on single keys. But, in real life applications, we may want to sort the data on several keys. The simplest example is that of sorting a deck of cards. The first key for sorting is the suit-clubs, spades, diamonds and hearts. Then, within each suit, sorting the cards in ascending order fromAce, twos to king. This is thus a case of sorting on 2 keys.

Now, this can be done in 2 ways.

- Sort the 52 cards into 4 piles according to the suit.
- Sort each of the 4 piles according to face value of the cards.

- Sort the 52 cards into 13 piles according to face value.
- Stack these piles in order and then sort into 4 piles based on suit.

The first method is called the MSD (Most Significant Digit) sort and the second method is called the LSD (Least Significant Digit) sort. Digit stands for a key. Thoughthey are called sorting methods, MSD and LSD sorts only decide the *order* of sorting. The actual sorting could be done by any of the sorting methods discussed in this unit.

☞ Check Your Progress 3

1) The complexity of Bubble sort is ___
2) Quick sort algorithm uses the programming technique of ____
3) Write a program in 'C' language for 2-way merge sort.
4) The complexity of Heap sort is _____

9.7 SUMMARY

Searching is the process of looking for something. Searching a list consisting of 100000 elements is not the same as searching a list consisting of 10 elements. We discussed two searching techniques in this unit namely Linear Search and Binary Search. Linear Search will directly search for the key value in the given list. Binary search will directly search for the key value in the given sorted list. So, the major difference is the way the given list is presented. Binary search is efficient in most of the cases. Though, it had the overhead that the list should be sorted before search can start, it is very well compensated through the time (which is very less when comparedto linear search) it takes to search. There are a large number of applications of Searching out of whom a few were discussed in this unit.

9.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) No
2) It will be located at the beginning of the list

Check Your Progress 2

1)

(a) F
(b) F
(c) F

Check Your Progress 3

1) $O(N^2)$ where N is the number of elements in the list to be sorted.
2) Divide and Conquer.
3) O(NlogN) where N is the number of elements to be sorted.

9.9 FURTHER READINGS

Reference Books

1. *Fundamentals of Data Structures in C++* by E. Horowitz, Sahai and D. Mehta, Galgotia Publications.

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

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

Reference Websites

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