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:

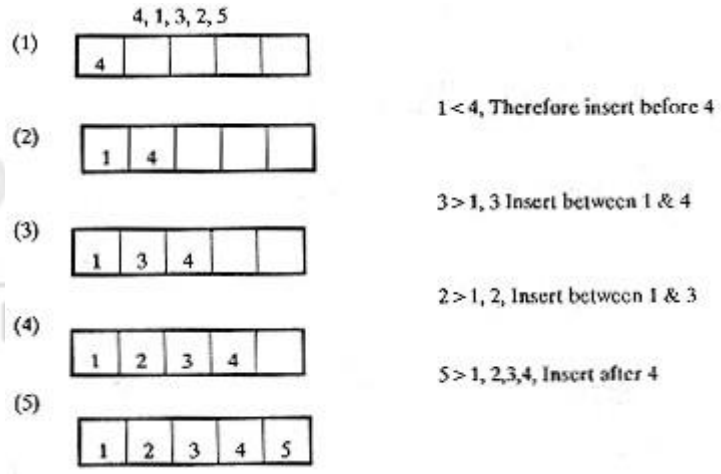


Figure 9.1 : Insertion sort

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.

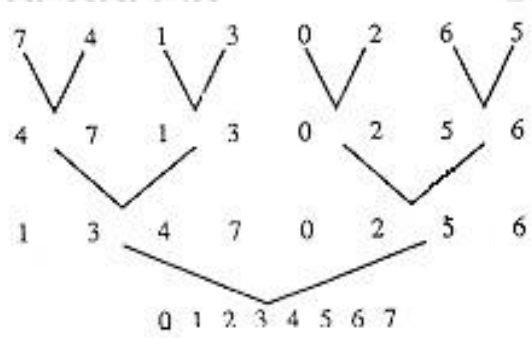


Figure 9.2: 2-way merge sort

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.

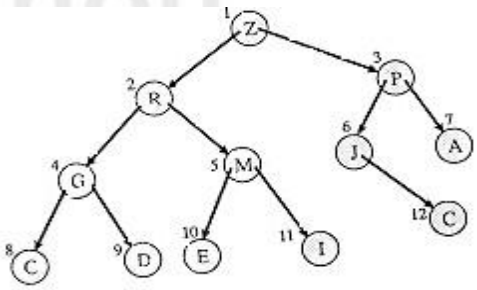


Figure 9.3: A Binary Tree

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.*

Index	1	2	3	4	5	6	7	8	9	10	11	12
Array[Index]	Z	R	P	G	M	J	A	C	D	E	I	C

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 $\lfloor J/2 \rfloor$.

Consider the node M. It is at position 5. Its parent node is, therefore, at position $\lfloor 5/2 \rfloor = 2$ i.e. the parent is R. Its children are at positions 2×5 & $(2 \times 5) + 1$, 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.*

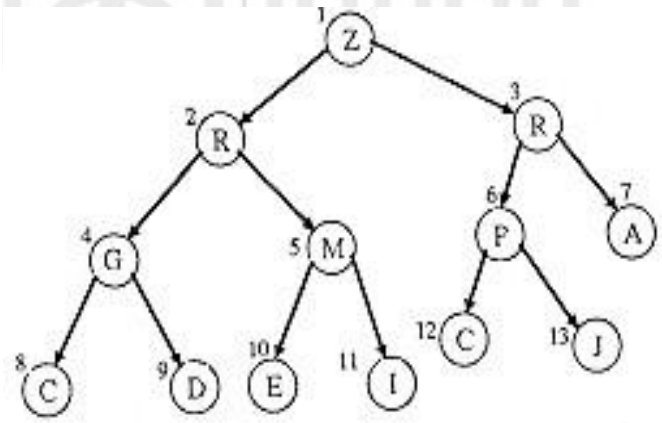


Figure 9.5: A Heap

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.

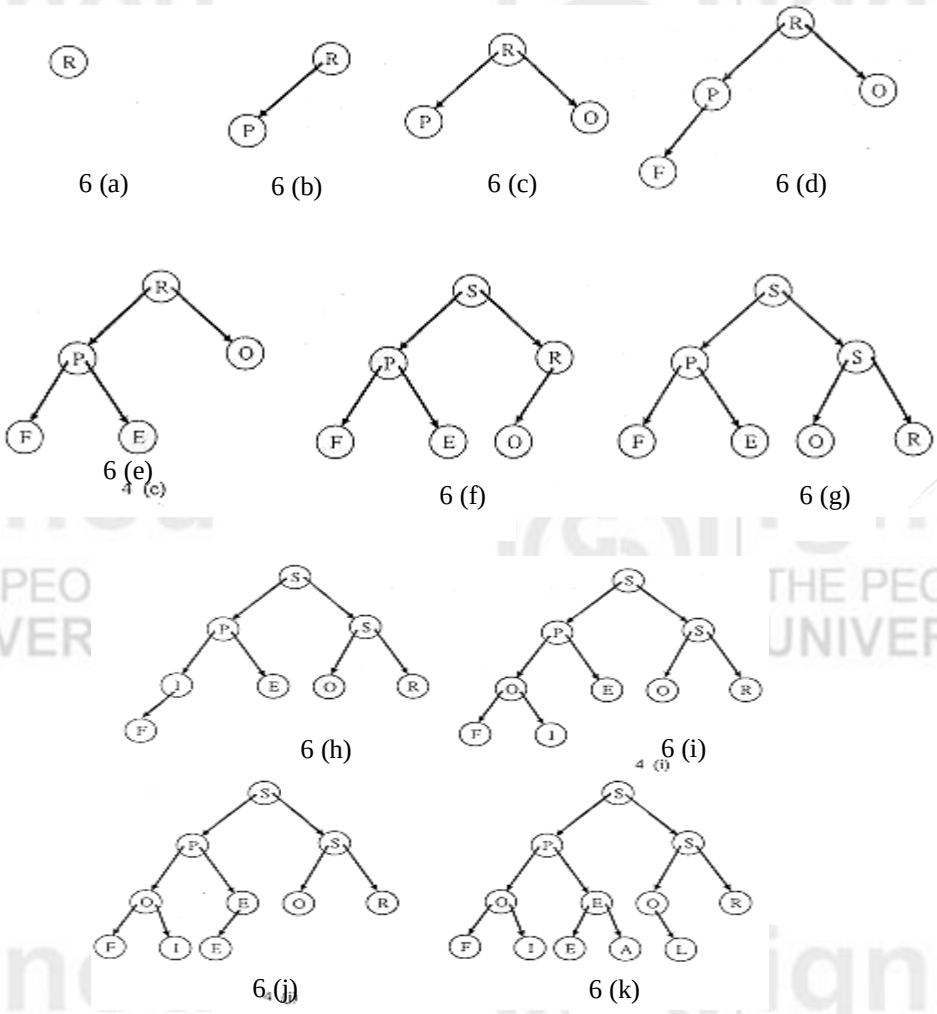


Figure 9.6: Heap Sort (Top down Construction)

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.

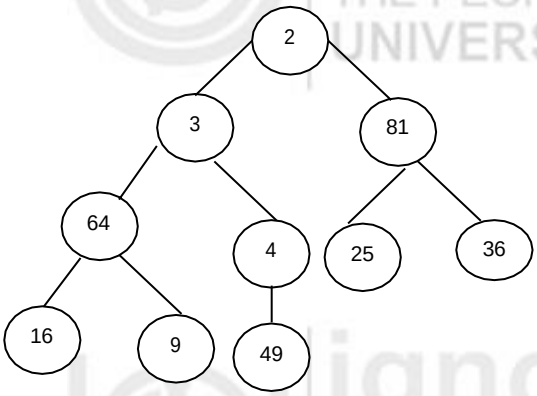


Figure 9.7: A Binary tree 9.7

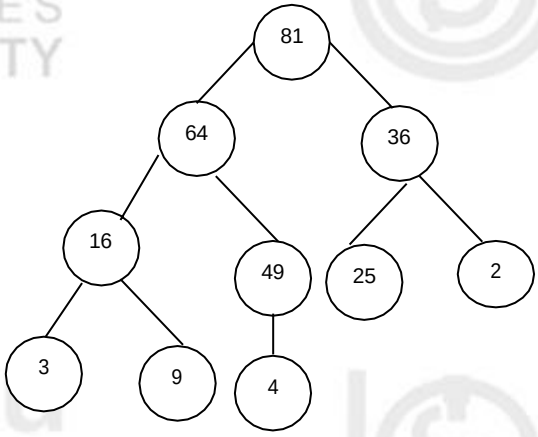
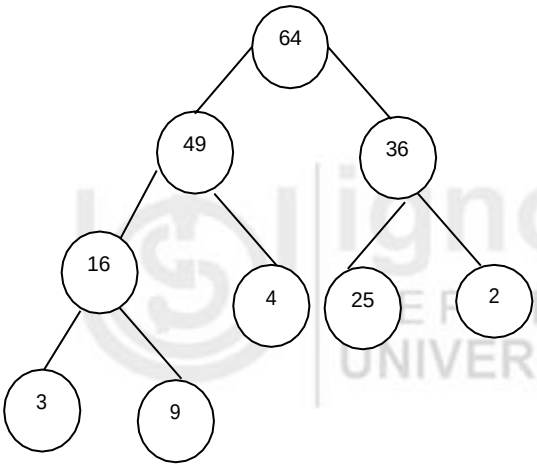
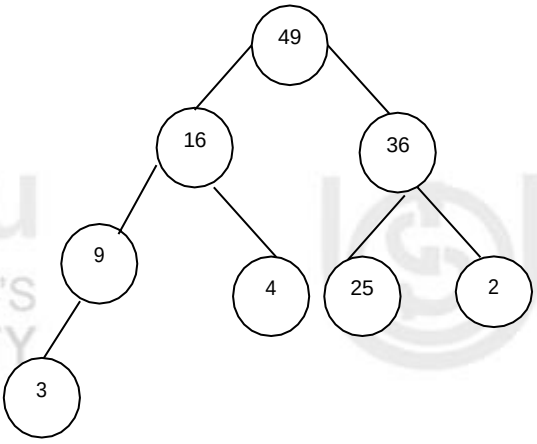


Figure 9.8: Heap of figure

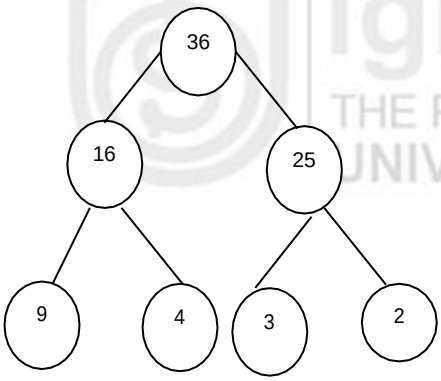
Figure 9.9 illustrates various steps of the heap of *Figure 9.8* as the sorting takesplace.



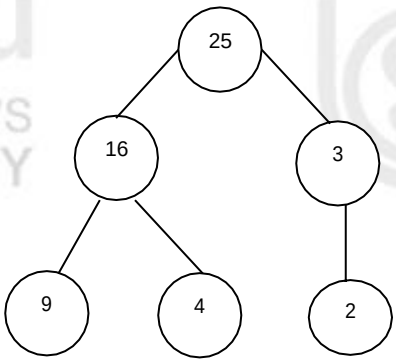
Sorted: 81
Heap size: 9



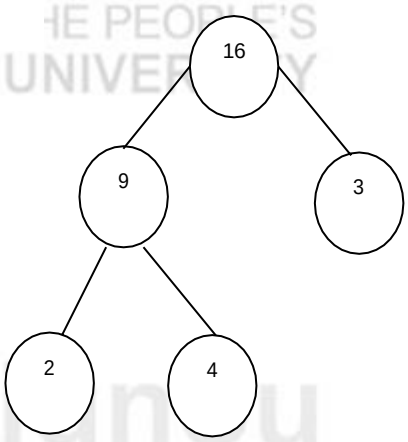
Sorted: 81,64
Heap size: 8



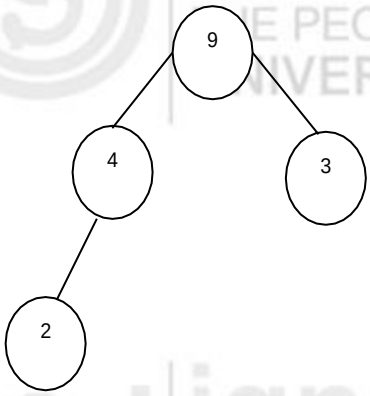
Sorted: 81,64,49
Heap size:7



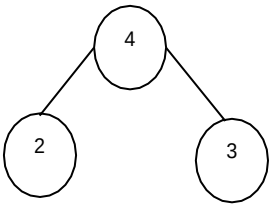
Sorted:81,64,49,36
Heap size:6



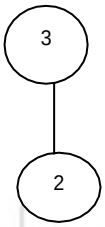
Sorted: 81, 64, 49, 36, 25
Size: 5



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