UNIT 10 SORTING

Structure **Page Nos.**

10.0 INTRODUCTION

Retrieval of information is made easier when it is stored in some predefined order. Sorting is, therefore, a very important computer application activity. Many sorting algorithms are available. Different environments require different sorting methods. Sorting algorithms can be characterised in the following two ways:

1. Simple algorithms which require the order of n^2 (written as $O(n^2)$)comparisons to sort n items.
2. Sophisticated algorithms that require the $O(n\log_2 n)$ comparisons to sort n items.

The difference lies in the fact that the first method moves data only over small distances in the process of sorting, whereas the second method moves data over large distances, so that items settle into the proper order sooner, thus resulting in fewer comparisons. Performance of a sorting algorithm can also depend on the degree of order already present in the data.

There are two basic categories of sorting methods: **Internal Sorting and External Sorting.** Internal sorting is applied when the entire collection of data to be sorted is small enough so that the sorting can take place within the main memory. The time required to read or write is not considered to be significant in evaluating the performance of internal sorting methods. External sorting methods are applied to larger collection of data which reside on secondary devices. Read and write access times are a major concern in determining sorting performances of such methods.

In this unit, we will study some methods of internal sorting. The next unit will discuss methods of external sorting.

10.1 OBJECTIVES

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

- list the names of some sorting methods;
- discuss the performance of several sorting methods, and
- describe sorting methods on several keys.

10.2 INTERNAL SORTING

In internal sorting, all the data to be sorted is available in the high speed main memory of 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

10.2.1 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 10.1*) before presenting the formal algorithm.

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

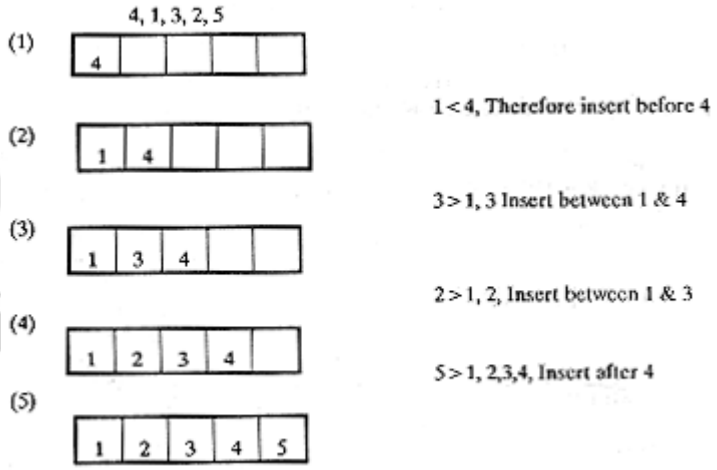


Figure 10.1: Insertion sort

Thus to find the correct position search the list till an item just greater than the target is 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.

10.2.2 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 process is 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 :

$= (N-1) +(N-2) \ldots + 2 + 1$

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

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

10.2.3 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. The basis of quick sort is the *divide and conquer* strategy i.e. Divide the problem [list to be sorted] 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 items have 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 found whose 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 10.1 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 10.1: 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.

10.2.4 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 merge sort 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 can be better understood by the following example. This is also called *Concatenate sort*. *Figure 10.2* depicts 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.

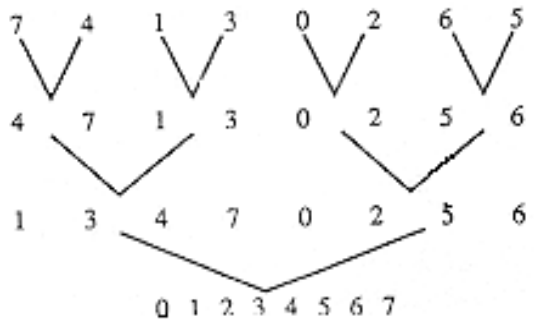


Figure 10.2: 2-way merge sort

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.

10.2.5 Heap Sort

We will begin by defining a new structure called *Heap. Figure 10.3* illustrates a Binary tree.

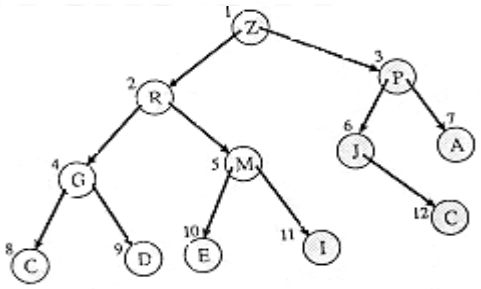


Figure 10.3: A Binary Tree

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, the corresponding array is depicted in *Figure 10.4*.

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 10.4: 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

$\lfloor 5/2 \rfloor = 2$ 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 be combined to generate a sorting algorithm.

The operations on a heap work in 2 steps.

1. The required node is inserted/deleted/or replaced.
2. 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 10.5*.

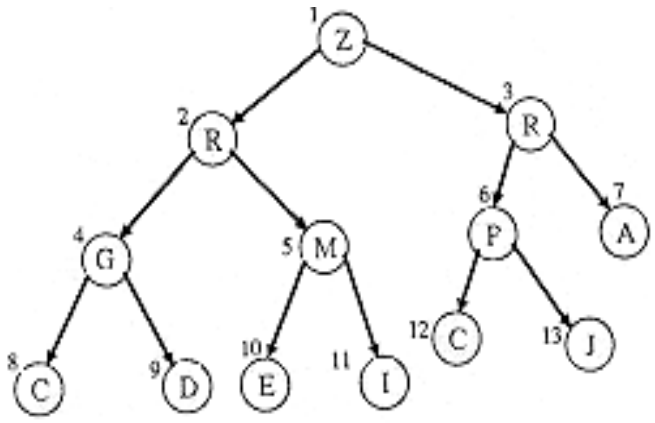


Figure 10.5: A Heap

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 the heap 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 10.6 shows different steps of the top down construction of the heap.

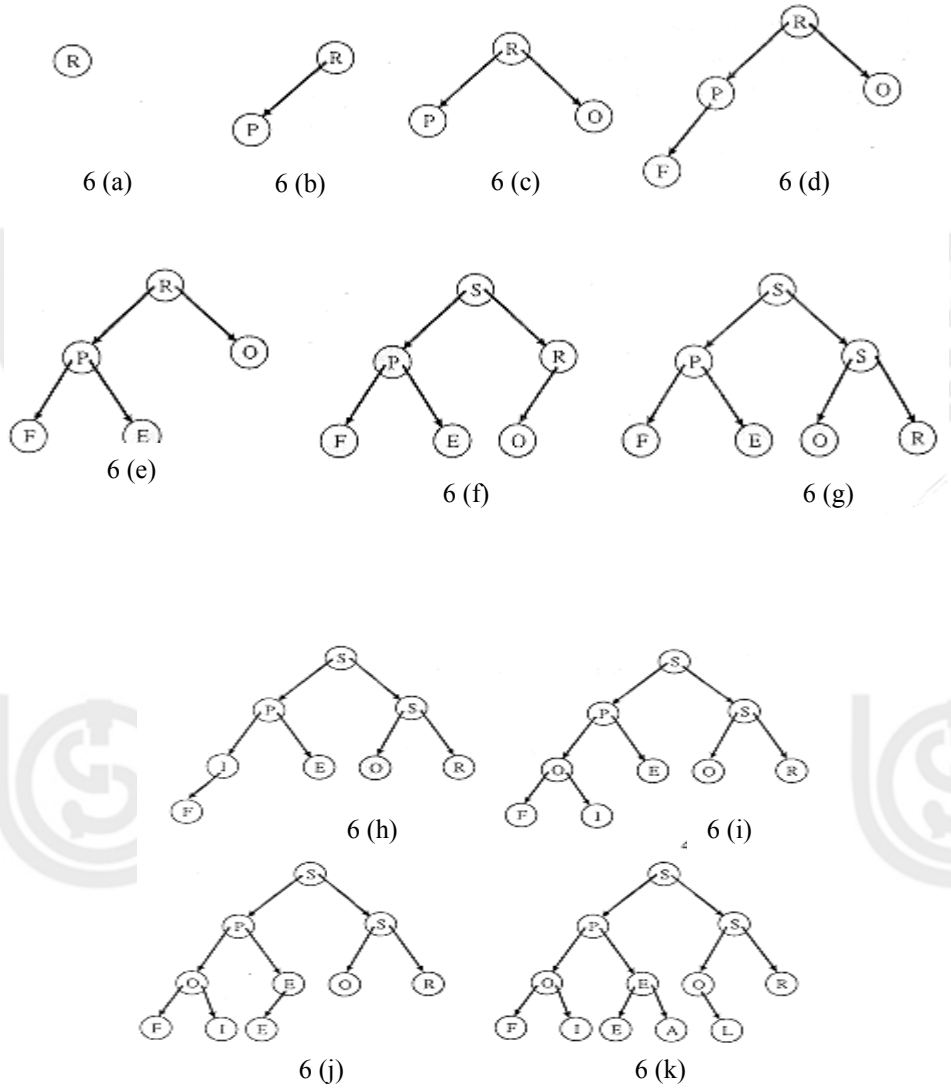


Figure 10.6: Heap Sort (Top down Construction)

Example: The input file is (2,3,81,64,4,25,36,16,9, 49). When the file is interpreted as a binary tree, it results in *Figure 10.7*. *Figure 10.8* depicts the heap.

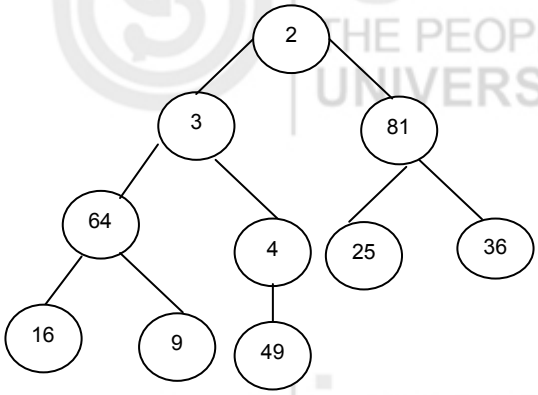


Figure 10.7: A Binary tree

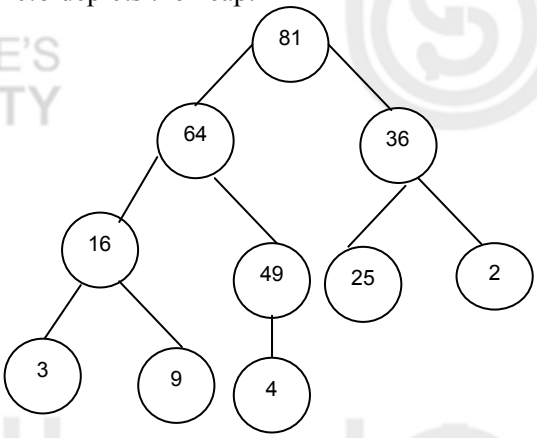
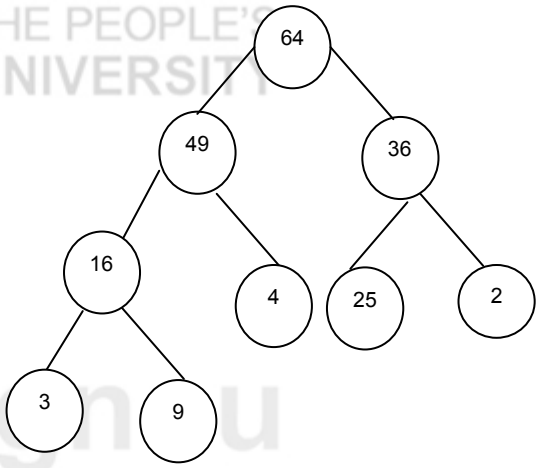
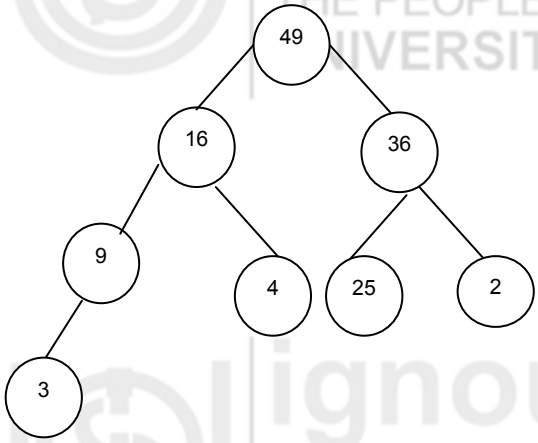


Figure 10.8: Heap of figure 10.7

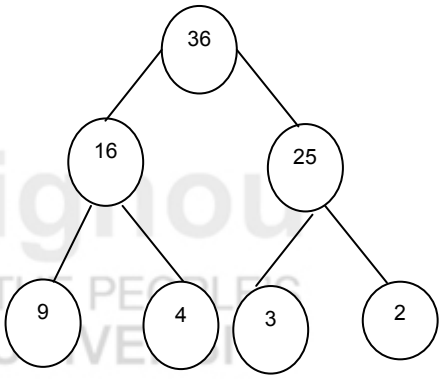
Figure 10.9 illustrates various steps of the heap of *Figure 10.8* as the sorting takes place.



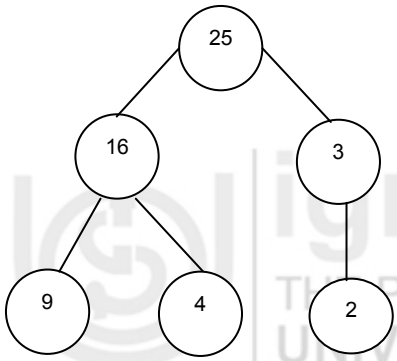
Sorted: 81
Heap size: 9



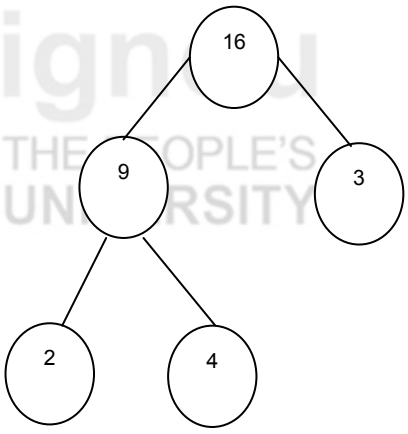
Sorted: 81,64
Heap size: 8



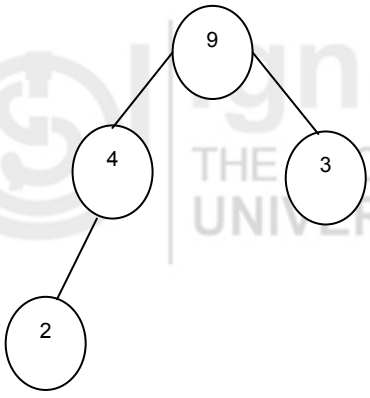
Sorted: 81,64,49
Heap size:7



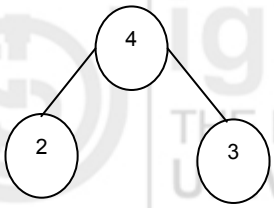
Sorted:81,64,49,36
Heap size:6



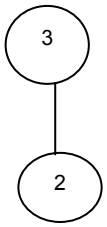
Sorted: 81, 64, 49, 36, 25
Size: 5



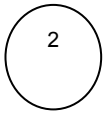
Sorted:81, 64, 49, 36, 25, 16
Size:4



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 10.9 : Various steps of figure 10.8 for a sorted file

10.3 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 from Ace, twos to king. This is thus a case of sorting on 2 keys.

Now, this can be done in 2 ways.

1
- Sort the 52 cards into 4 piles according to the suit.
- Sort each of the 4 piles according to face value of the cards.

2
- 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. Though they 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 1

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 _____

10.4 SUMMARY

Sorting is an important application activity. Many sorting algorithms are available. But, each is efficient for a particular situation or a particular kind of data. The choice of a sorting algorithm is crucial to the performance of the application.

In this unit we have studied many sorting algorithms used in internal sorting. This is not a conclusive list and the student is advised to read the suggested books for

exposure to additional sorting methods and for detailed discussions of the methods introduced here.

The following are the *three* most important efficiency criteria:

- use of storage space
- use of computer time
- programming effort.

10.5 SOLUTIONS/ANSWERS

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.

10.6 FURTHER READINGS

Reference Books

1. *Data Structures using C* by Aaron M.Tanenbaum, Yedidyah Langsam, Moshe J.Augenstein, PHI publications.

2. *Algorithms+Data Structures = Programs* by Niklaus Wirth, PHI publications.

Reference Websites

http://www.it.jcu.edu.au/Subjects/ cp2001/1998/LectureNotes/Sorting/

http://oopweb.com/Algorithms/Files/Algorithms.html