UNIT 2 SORTING TECHNIQUES - 1

Structure

2.0 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. Differing environments require differing sorting methods. Sorting algorithms can be characterized 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 are applied when the entire collection of data to sorted is small enough that the sorting can take place within 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 time are major concern in determine sort performances.

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

2.1 OBJECTIVES

After going through this unit you will be able to :

List internal sorting methods

Discuss and analyze the performance of several sorting methods

Describe sorting methods on several keys

2.2 INTERNAL SORTING

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

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

2.2.1 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 before (figure 1) presenting the formal algorithm.

Example 1: Sort the following list using the insertion sort method:

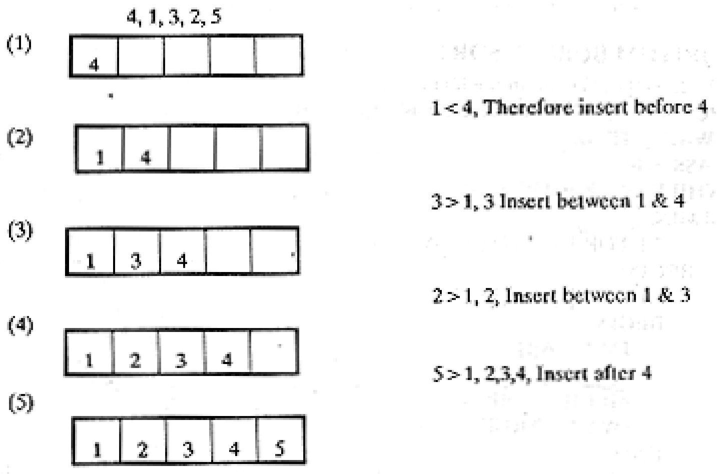

4, 1, 3, 2, 5
(1)
4
1 < 4, Therefore insert before 4
(2)
1 4
3 > 1, 3 Insert between 1 & 4
(3)
1 3 4
2 > 1, 2, Insert between 1 & 3
(4)
1 2 3 4
5 > 1, 2,3,4, Insert after 4
(5)
1 2 3 4 5

Figure 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.

We now present the algorithm for insertion sort.

ALGORITHM: INSERT SORT

INPUT: LIST[ ] of N items in random order.

OUTPUT: LIST[ ] of N items in sorted order.

```
1       BEGIN,
2.      FOR I = 2 TO N DO
3.      BEGIN
4.      F LIST[I] LIST[I-1]
5.      THEN BEGIN
6.      J = I
7.      T = LIST[I] /*STORE LIST[I]*/
8.      REPEAT /* MOVE OTHER ITEMS DOWN THE LIST*/
9:      J = J-1
10.     LIST [J + 1] =LIST [J];
11.     IFJ = 1THEN
12.     FOUND =TRUE
13.     UNTIL (FOUND = TRUE)
14.     LIST [I] = T
15.     END
16.     END
17.     END
```

2.2.2 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, they are swapped. This process is carried on till the list is sorted.

The detailed algorithm follows:

ALGORITHM BUBBLE SORT

INPUT: LIST [ ] of N items in random order.

OUTPUT: LIST [ ] of N items sorted in ascending order.

```
1.      SWAP = TRUE

        PASS 0/

2.      WHILE SWAP = TRUE DO

        BEGIN

        2.1 FOR.1 = 0 TO (N-PASS) DO
        BEGIN

        2.1.1 IF A[I] A [I + 1]

        BEGIN

                TMP = A[I]

                A[I] A[I+1]

                A[I+ 1] TMP

                SWAP = TRUE
        END

                ELSE

        SWAP = FALSE

        2.1.2 PASS = PASS + 1

        END

END
```

Total number of comparisions in Bubble sort are

= (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.

2.2.3 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
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. A[0], A[1] .. A[I-1] in sub list 1

2. A[I]

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

Repeat steps 1 & 2 for sublist & sublist2 till A[ ] is a sorted list.

As can be seen, this algorithm has a recursive structure.,

Step 2 or the 'divide' procedure is of utmost importance in this algorithm. This is usually implemented as follows:

1. Choose A[I] 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 sublist 1 & sublist2 are ready.

7. Now do the same for each of sublist 1 & sublist2.

We will now give the implementation of Quicksort and illustrate it by an example. Quicksort (int A[], int X, int 1)

```
        {

        int L, R, V 1.
1.      If (IX)
        {

2.      V = A[1], L = X-1, R = I; 3.

3.      For (;;)

4.      While (A[ + + L] V);

5.      While (A[- -R] V);

6.      If (L = R) /* left & right ptrs. have crossed */

7.      break;

8.      Swap (A, L, R) /* Swap A[L] & A[R] */ }

9.      Swap (A, L, I)

10.     Quicksort (A, X, L-1)

11.     Quicksort (A, L + 1, I) } }
```

Quick sort is called with A, I, N to sort the whole file.

Example: Consider the following list to be sorted in ascending order. 'ADD YOUR MAN'. (Ignore blanks)

N = 10

	0	1	2	3	4	5	6	7	8	9
A[ ] =	A	D	D	Y	O	U	R	M	A	N

Quicksort (A, 0, 9)

1. 9 > 0
2. V = [9] = 'N'
   L = 1-1 = 0
   R = I= 9

4. A[3] = 'Y' > V; There fore, L = 3
5. A [8] = 'A' > V; There fore, R = 8
6. L < R
8. SWAP (A,3,8) to get

	0	1	2	3	4	5	6	7	8	9
A[ ] =	A	D	D	A	O	U	R	M	Y	N

4 A [4] = 'O' > V, There fore, L =4
5. A[7] = 'M' < V, There fore, R = 7
6. L < R
7. SWAP (A,4,7) to get

	0	1	2	3	4	5	6	7	8	9
A [ ] =	A	D	D	A	M	U	R	O	Y	N

4 A [5] = 'U' > V;.. L = 5
5 A[4] = 'M' < V;R = 4
6 L < R ,.. break
9 SWAP (A,5,9) to get.

	0	1	2	3	4	5	6	7	8	9
A[ ] =	A	D	D	A	M	N	R	O	Y	U

at this point 'N' is in its correct place.

A[5], A[0] to A[4] constitutes sub list 1.

A[6] to A[9] constitutes sublist2. Now

10. Quick sort (A, 0, 4)
11. Quick sort (A, 5, 9)

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 comparitively small.

2. Use a better dividing element I in the implementations. We have always used A[N] as the dividing element. A useful method for the selection of a dividing element is the Median-of three method.

Select any3 elements from the list. Use the median of these as the dividing element.

2.2.4 2-WAY MERGE SORT

Merge sort is also one of the 'divide and conquer' class of algorithms. The basic idea into this is to divide the list into a number of sub lists, sort each of these sub lists and merge them to get a single sorted list. The recursive implementation of 2- way merge sort divides the fist into 2 sorts the sub lists and then merges them to get the 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.

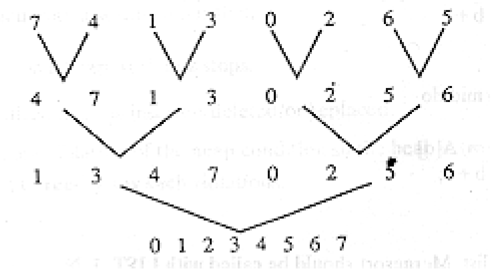


Figure 2 : 2-way.merge sort

We give here the recursive implementation of 2 Way Merge Sort.

```
Mergesort (int List [ ], int low, int high)

{
    int mid,
1.  Mid = (low + high)/2;
2.  Mergesort (LIST, low, mid);
3.  Mergesort (LIST, mid + 1, high);
4.  Merge (low, mid, high, List, FINAL)
}

Merge (int low, int mid, int high, int LIST[], int FINAL)

{

int a, b, c, d;

a = low, b = low, c = mid + 1

While (a < = mid and c < = high) do
{
If ( LIST [a] < = LIST [c] ) then
{
```

```
            FINAL [b] =LIST [a]
            ++a;
}
else
{
            FINAL [b] = LIST [c]
            ++ c;

}

            ++b
}
If (a mid) then

For (d = c; d< = high; ++d)

{
            B[b] = LIST [d]
            ++b;

}
Else
For (d = a; d<= mid; ++d)
{
            B[b] = A[d]
            ++b;
 }
}
```

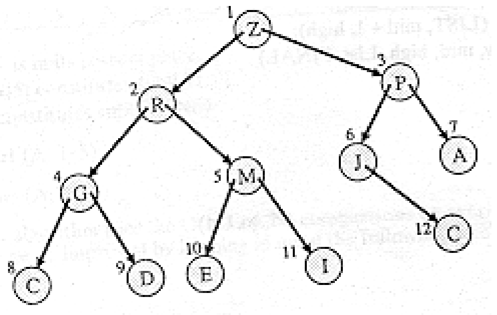
To sort the entire list, Mergesort should be called with LIST,1, N.

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 type for the merge phase. That is, to sort and list of size n, it needs space for 2n elements.

2.2.5 HEAP SORT

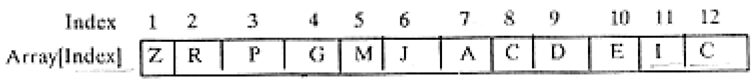
We will begin by defining a new structure the heap. We have studied binary trees in BLOCK 5, UNIT 1. A binary tree is illustrated below.



3(a) : Heap 1

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 would be



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

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 the position 5. Its parent node is, therefore, at position [5/2| = 2 i.e. the parent is R. Its children are at positions 2x5 & (2x5) + 1, i.e.10 + 11 respectively i.e. E & I are its children. We see from the pictorial representation that these relationships are correct.

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. 1 may cause violation of the heap condition so the heap is traversed and modified to rectify any, such violations.

Examples

Insertion

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.

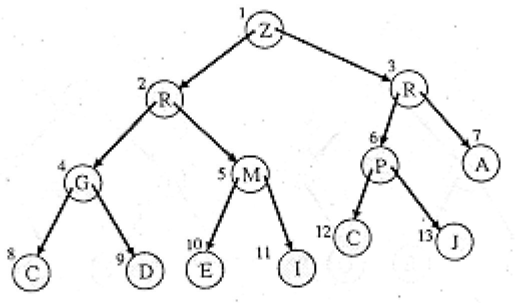
2. But R J, the heap condition is violated.

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.

6. The heap condition is now satisfied by all nodes to get.



3(b) : Heap 2

Deletion Consider the deletion of M from heap 2.

1. The larger of M's children is promoted to 5.
2.

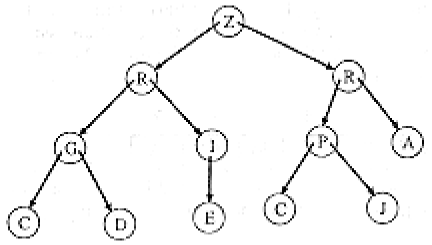


Figure 3(c) : Heap 3

An efficient sorting method based on the heap construction and node removal from the heap in order. This algorithm is guaranteed to sort n elements in N log N steps.

We will first see 2 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, keeping the heap condition inviolate 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 both methods of construction.

PROFESSIONAL

Top down construction

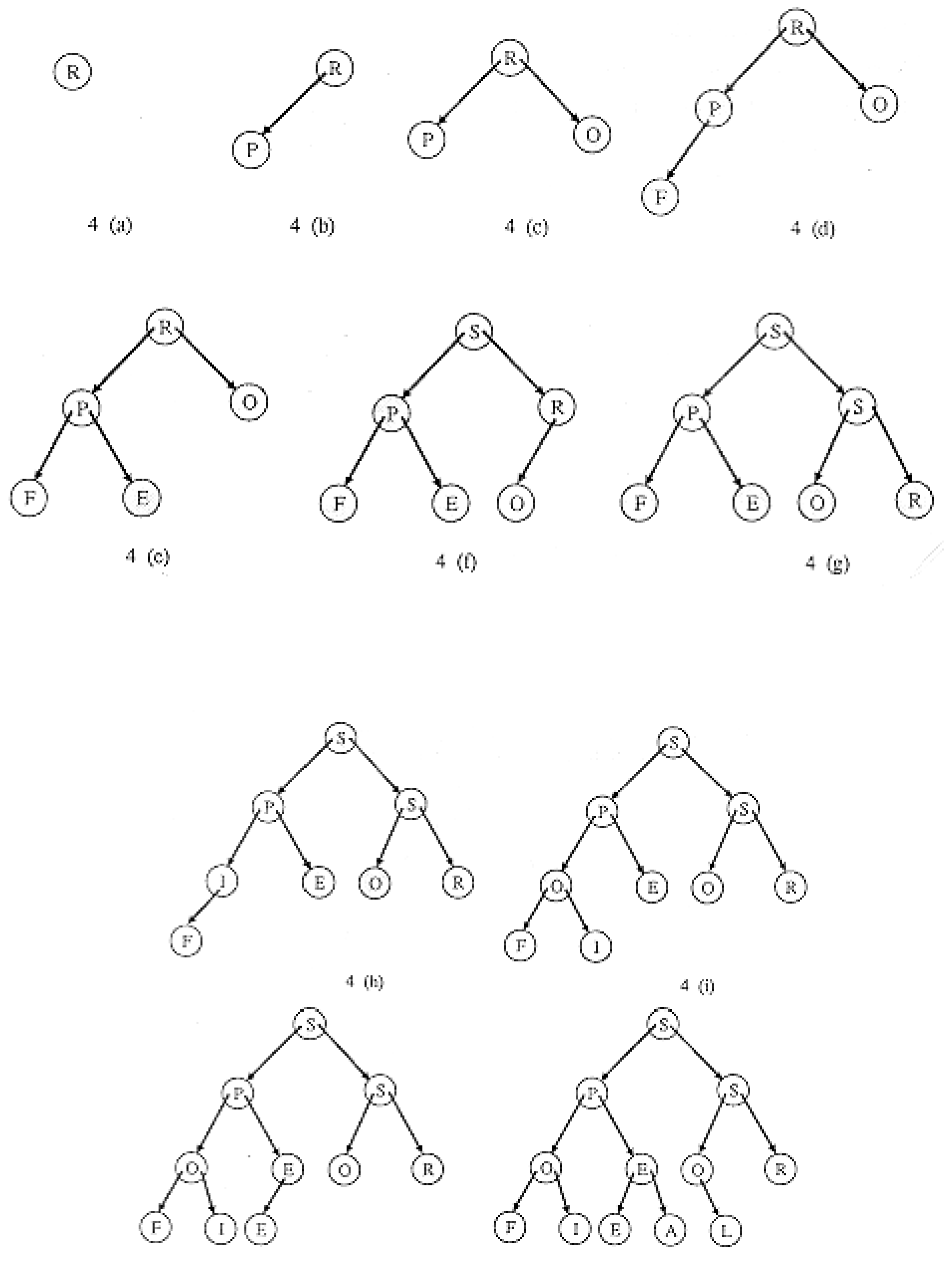


Figure 4: Heap Sort (Top down Construction)

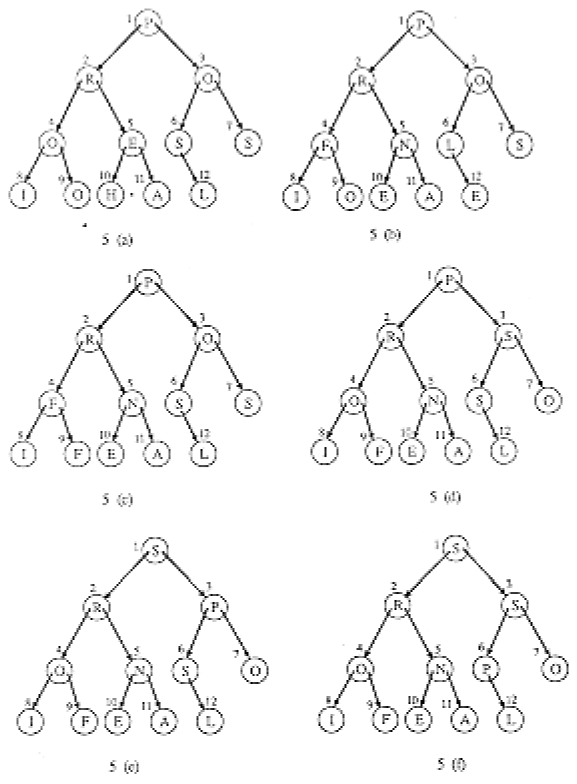
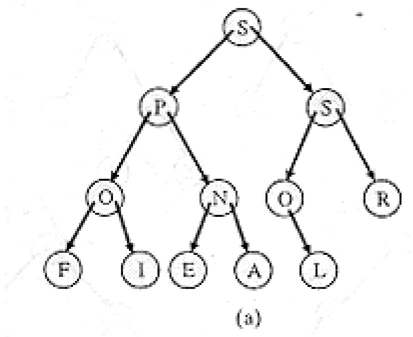
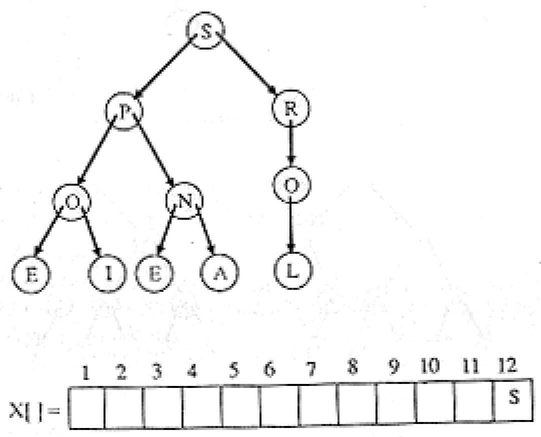


Figure 5: Heap Sort by bottom-up approach

We will now see how sorting takes place using the heap built by the top down approach. The sorted elements will be placed in X [ ] and array of size 12.

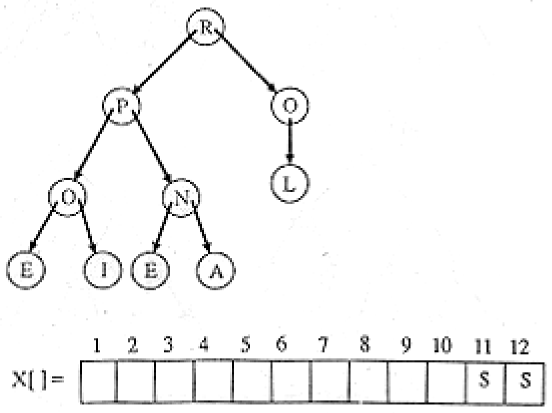


1. Remove S and store in X [12)



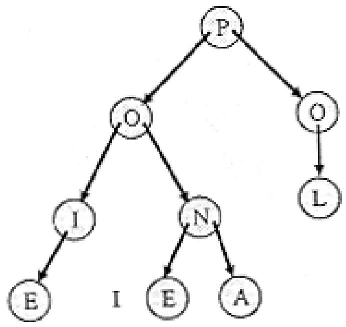
(b)

2. Remove S and store in X [11]



(c)

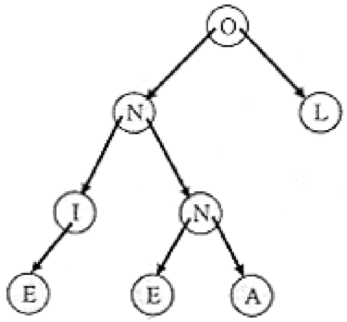
3. Remove R and store in X[10]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ]=										R	S	S

;(d)

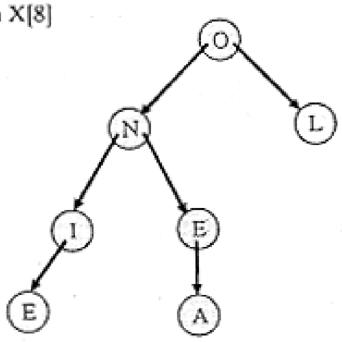
4. Remove P and store in X[a]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ]=									P	R	S	S

(e)

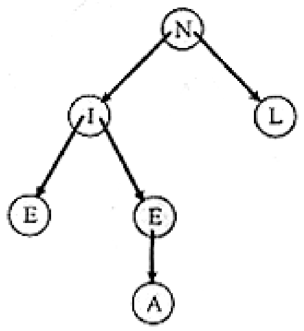
5. Remove O and store in X[8]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ]=								O	P	R	S	S

(f)

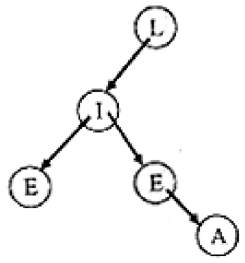
6. Remove O and store in X[7]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ] =							O	O	P	R	S	S

(g)

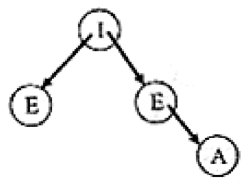
7. Remove N and store in X[6]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ] =						N	O	O	P	R	S	S

(h)

8. Remove L and store in X[5]



	1	2	3	4	5	6	7	8	9	10	11	12
X[ ] =						N	O	O	P	R	S	S

(i)

9. Similarly the remaining 5 nodes are removed and the heap modified, to get the sorted list.

A E E I L N 0 0 P R S S

Figure 6 : Sorting process through Heap

2.3 SORTING ON SEVERAL KEYS

So far we have been considering sorting based on single keys. But in real life application 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 (1) Sort the 52 cards into 4 piles according to the suit.
  (2) Sort each of the 4 piles according to face value of the cards.
2 (1) Sort the 52 cards into 13 piles according to face value.
  (2) 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 here can be said to stand for key. Though they are called sorting methods, MSD and LSD sort only decide the 'order' of sorting. The actual sorting could be done by any of the sorting methods discussed in this unit.

2.4 SUMMARY

Sorting is an important application activity. Many sorting algorithms are available, each the most 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 volumes for exposure to additional sorting methods and for detailed discussions of the methods introduced here.

The three important efficiency interia are

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

In the next unit we will discuss internal sorting.

2.5 REVIEW QUESTIONS

1. Formulate an algorithm to perform insertion sort on a linked list.

2. What initial order of data will produce the maximum number of comparisons in insertion sort.

3. Modify the insertion sort algorithm to use the binary search method to determine the position of insertion.

4. Implement bubble sort in Pascal.

5. Define 'divide & conquer' with relation to sorting.

6. Modify the quicksort algorithm to use the Median of three dividing method.

7. Modify the mergesort algorithm to deal with linked lists.

8. Describe a heap.

9. Formulate algorithm for the heap operations insert and delete.

2.6. PROGRAMMING PROJECTS

1. Implement quicksort using contiguous lists and linked lists and compare the performance.

2. A sorting procedure is said to be 'stable' if whenever two items which have the same value will be in the same order in the sorted list as in the unsorted one. Determine which of the methods discussed here are stable.

3. Write a program in C which will allow a user to

   a. enter the data to be sorted.

   b. specify the storage method - contiguous list or linked list.

   c. specify the sorting method (with an additional inputs required for a particular method).

   Implement the above.

2.7 SUGGESTED READING

1. FUNDAMENTALS OF COMPUTER ALGORITHMS

   HOROWITZ & SAHNI

2. FUNDAMENTALS OF DATA STRUCTURES IN PASCAL

   HOROWITZ & SAHNI

3. DATA STRUCTURES & PROGRAM DESIGN

   ROBERT L. KRUSE

4. THE ART OF PROGRAMMING VOLUME 3

   SORTING & SEARCHING
   DONALD. E. KNUTI-1