6 Sorting Techniques

The operation of sorting is the most common task performed by computers today. Sorting is used to arrange names and numbers in meaningful ways. For example; it is easy to look in the dictionary for a word if it is arranged (or sorted) in alphabetic order .

Let A be a list of *n* elements A1, A2, ... An in memory. Sorting of list A refers to the operation of rearranging the contents of A so that they are in increasing (or decreasing) order (numerically or lexicographically); A1 < A2 < A3 < ...... < An.

Since A has *n* elements, the contents in A can appear in *n*! ways. These ways correspond precisely to the *n*! permutations of 1,2,3, ... *n*. Each sorting algorithm must take care of these *n*! possibilities.

For example

Suppose an array A contains 7 elements, 42, 33, 23, 74, 44, 67, 49. After sorting, the array A contains the elements as follows 23, 33, 42, 44, 49, 67, 74. Since A consists of 7 elements, there are 7! =.5040 ways that the elements can appear in A.

The elements of an array can be sorted in any specified order *i.e.*, either in ascending order or descending order. For example, consider an array A of size 7 elements 42, 33, 23, 74, 44, 67, 49. If they are arranged in ascending order, then sorted array is 23, 33, 42, 44, 49, 67, 74 and if the array is arranged in descending order then the sorted array is 74, 67, 49, 44, 42, 33, 23. In this chapter all the sorting techniques are discussed to arrange in ascending order.

Sorting can be performed in many ways. Over a time several methods (or algorithms) are being developed to sort data(s). Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, Binary sort, Shell sort and Radix sort are the few sorting techniques discussed in this chapter.

It is very difficult to select a sorting algorithm over another. And there is no sorting algorithm better than all others in all circumstances. Some sorting algorithm will perform well in some situations, so it is important to have a selection of sorting algorithms. Some factors that play an important role in selection processes are the time complexity of the algorithm (use of computer time), the size of the data structures (for Eg: an array) to be sorted (use of storage space), and the time it takes for a programmer to implement the algorithms (programming effort).

For example, a small business that manages a list of employee names and salary could easily use an algorithm such as bubble sort since the algorithm is simple to implement and the data to be sorted is relatively small. However a large public limited with ten thousands of employees experience horrible delay, if we try to sort it with bubble sort algorithm. More efficient algorithm, like Heap sort is advisable.

6.1. COMPLEXITY OF SORTING ALGORITHMS

The complexity of sorting algorithm measures the running time of n items to be sorted. The operations in the sorting algorithm, where A1, A2 ….. An contains the items to be sorted and B is an auxiliary location, can be generalized as:

(*a*) Comparisons- which tests whether A*i* < A*j* or test whether A*i* < B

(*b*) Interchange- which switches the contents of A*i* and A*j* or of A*i* and B

(*c*) Assignments- which set B = A and then set A*j* = B or A*j* = A*i*

Normally, the complexity functions measure only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

6.2. BUBBLE SORT

In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the positions of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Let A be a linear array of *n* numbers. Sorting of A means rearranging the elements of A so that they are in order. Here we are dealing with ascending order. *i.e.*, A[1] < A[2] < A[3] < …… A[*n*].

Suppose the list of numbers A[1], A[2], ……….. A[*n*] is an element of array A. The bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in the (or desired) ascending order, so that A[1] < A[2].that is if A[1] is greater than A[2] then interchange the position of data by swap = A[1]; A[1] = A[2]; A[2] = swap. Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Continue the process until we compare A[N – 1] with A[N].

Note: Step1 contains *n* – 1 comparisons *i.e.*, the largest element is “bubbled up” to the *n*th position or “sinks” to the *n*th position. When step 1 is completed A[N] will contain the largest element.

Step 2: Repeat step 1 with one less comparisons that is, now stop comparison at A [*n* – 1] and possibly rearrange A[N – 2] and A[N – 1] and so on.

Note: in the first pass, step 2 involves n–2 comparisons and the second largest element will occupy A[*n*-1]. And in the second pass, step 2 involves *n* – 3 comparisons and the 3rd largest element will occupy A[*n* – 2] and so on.

Step *n* – 1: compare A[1]with A[2] and arrange them so that A[1] < A[2]

After *n* – 1 steps, the array will be a sorted array in increasing (or ascending) order.

The following figures will depict the various steps (or PASS) involved in the sorting of an array of 5 elements. The elements of an array A to be sorted are: 42, 33, 23, 74, 44

FIRST PASS

33 swapped	33	33	33
42	23 swapped	23	23
23	42	42 no swapping	42
74	74	74	44 swapped
44	44	44	74

```
 23 swapped      23              23
33               33 no swapping  33
42               42              42 no swapping
44               44              44
74               74              74
```

```
23 no swapping   23
33               33 no swapping
42               42
44               44
74               74
```

```
23 no swapping
33
42
44
74
```

Thus the sorted array is 23, 33, 42, 44, 74.

ALGORITHM

Let A be a linear array of *n* numbers. Swap is a temporary variable for swapping (or interchange) the position of the numbers.

```
1. Input $n$ numbers of an array A
2. Initialise $i = 0$ and repeat through step 4 if $(i < n)$
3. Initialize $j = 0$ and repeat through step 4 if $(j < n - i - 1)$
4. If $(A[j] > A[j + 1])$
(a) Swap = $A[j]$
(b) $A[j] = A[j + 1]$
(c) $A[j + 1]$ = Swap
5. Display the sorted numbers of array A
6. Exit.
```

PROGRAM 6.1

```
//PROGRAM TO IMPLEMENT BUBBLE SORT USING ARRAYS
//STATIC MEMORY ALLOCATION
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>
```

```
#define MAX 20
void main()
{
        int arr[MAX],i,j,k,temp,n,xchanges;
        clrscr();
        printf (“\nEnter the number of elements : ”);
        scanf (“%d”,&n);
        for (i = 0; i < n; i++)
        {
                printf (“E\nnter element %d : ”,i+1);
                scanf (“%d”,&arr[i]);
        }
        printf (“\nUnsorted list is :\n”);
        for (i = 0; i < n; i++)
                printf (“%d ”, arr[i]);
         printf (“\n”);

        /* Bubble sort*/
        for (i = 0; i < n–1 ; i++)
        {
                xchanges=0;
                for (j = 0; j <n–1–i; j++)
                {
                        if (arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                                xchanges++;
                        }/*End of if*/
                }/*End of inner for loop*/
                if (xchanges==0) /*If list is sorted*/
                        break;
                printf(“\nAfter Pass %d elements are :  ”,i+1);
                for (k = 0; k < n; k++)
                        printf(“%d ”, arr[k]);
                printf(“\n”);
        }/*End of outer for loop*/
        printf(“\nSorted list is :\n”);
        for (i = 0; i < n; i++)
                printf(“%d ”, arr[i]);
        getch();
}/*End of main()*/
```

PROGRAM 6.2

```
//PROGRAM TO IMPLEMENT BUBBLE SORT
//USING DYNAMIC MEMORY ALLOCATION
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>

//this function will bubble sort the input
void bubblesort(int *a,int n)
{
        int i,j,k,temp;
        for(i=1;i < n;i++)
        for(j=0;j < n–1;j++)
                if (a[j] > a[j+1])
                {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                }
}

void main()
{
        int *a,n,*l,*temp;
        clrscr();
        printf (“\nEnter the number of elements :”);
        scanf (“%d”,&n);

        //allocating the array of memory dynamically
        a=((int*)malloc(n*sizeof (int)));
        temp=a;//storing the memnory address
        l=a+n;
        printf (“\nEnter the elements\n”);
        while(a < l)
        {
                scanf (“%d”,a);
                a++;
        }
        bubblesort(temp,n);
        printf ("\nSorted array is : ");
        a=temp;
        while(a < l)
        {
                printf ("  %d",*a);
                a++;
        }
        getch();
}
```

TIME COMPLEXITY

The time complexity for bubble sort is calculated in terms of the number of comparisons $f(n)$ (or of number of loops); here two loops (outer loop and inner loop) iterates (or repeated) the comparisons. The number of times the outer loop iterates is determined by the number of elements in the list which is asked to sort (say it is n). The inner loop is iterated one less than the number of elements in the list (*i.e.*, n-1 times) and is reiterated upon every iteration of the outer loop

$$f(n) = (n - 1) + (n - 2) + \ldots\ldots + 2 + 1$$
$$= n(n - 1) = O(n2).$$

BEST CASE

In this case you are asked to sort a sorted array by bubble sort algorithm. The inner loop will iterate with the 'if' condition evaluating time, that is the swap procedure is never called. In best case outer loop will terminate after one iteration, *i.e.*, it involves performing one pass, which requires n–1 comparisons

$$f(n) = O(n)$$

WORST CASE

In this case the array will be an inverted list (*i.e.*, 5, 4, 3, 2, 1, 0). Here to move first element to the end of the array, n–1 times the swapping procedure is to be called. Every other element in the list will also move one location towards the start or end of the loop on every iteration. Thus n times the outer loop will iterate and n (n-1) times the inner loop will iterate to sort an inverted array

$$f(n) = (n(n - 1))/2 = O(n2)$$

AVERAGE CASE

Average case is very difficult to analyse than the other cases. In this case the input data(s) are randomly placed in the list. The exact time complexity can be calculated only if we know the number of iterations, comparisons and swapping. In general, the complexity of average case is:

$$f(n) = (n(n–1))/2 = O(n2).$$

6.3. SELECTION SORT

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on.

Let A be a linear array of '*n*' numbers, A [1], A [2], A [3],...... A [*n*].

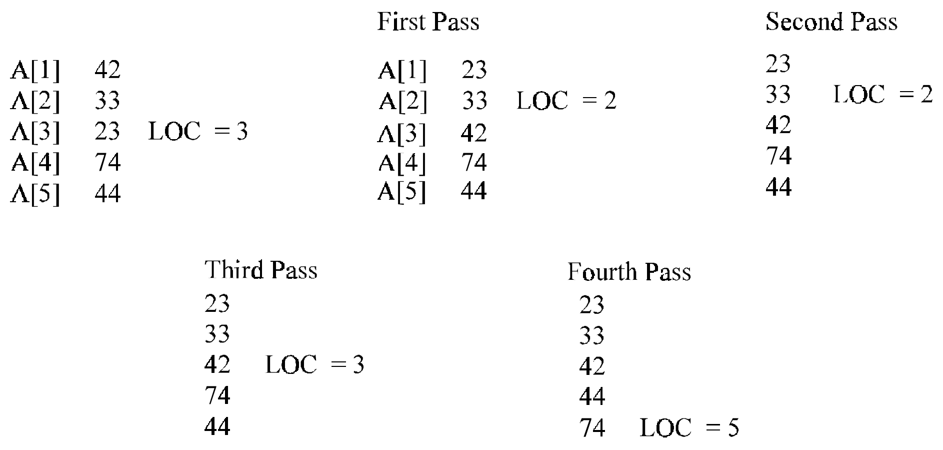
Step 1: Find the smallest element in the array of *n* numbers A[1], A[2], ... A[*n*]. Let LOC is the location of the smallest number in the array. Then interchange A[LOC] and A[1] by swap = A[LOC]; A[LOC] = A[1]; A[1] = Swap.

Step 2: Find the second smallest number in the sub list of *n* – 1 elements A [2] A [3] ...... A [*n* – 1] (first element is already sorted). Now we concentrate on the rest of the elements in the array. Again A [LOC] is the smallest element in the remaining array and LOC the corresponding location then interchange A [LOC] and A [2].Now A [1] and A [2] is sorted, since A [1] less than or equal to A [2].

Step 3: Repeat the process by reducing one element each from the array

Step n – 1: Find the *n* – 1 smallest number in the sub array of 2 elements (*i.e.*, A(*n*–1), A (*n*)). Consider A [LOC] is the smallest element and LOC is its corresponding position. Then interchange A [LOC] and A(*n* – 1). Now the array A [1], A [2], A [3], A [4],..........A [*n*] will be a sorted array.

Following figure is generated during the iterations of the algorithm to sort 5 numbers 42, 33, 23, 74, 44 :



			First Pass			Second Pass	
A[1]	42		A[1]	23		23	
A[2]	33		A[2]	33	LOC = 2	33	LOC = 2
A[3]	23	LOC = 3	A[3]	42		42	
A[4]	74		A[4]	74		74	
A[5]	44		A[5]	44		44	

Third Pass		Fourth Pass	
23		23	
33		33	
42	LOC = 3	42	
74		44	
44		74	LOC = 5

ALGORITHM

Let A be a linear array of *n* numbers A [1], A [2], A [3], ... A [k], A [k+1], ... A [n]. *Swap* be a temporary variable for swapping (or interchanging) the position of the numbers. *Min* is the variable to store smallest number and *Loc* is the location of the smallest element.

```
1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step5 if (i < n – 1)
   (a) min = a[i]
   (b) loc = i
```

```
3. Initialize $j = i + 1$ and repeat through step 4 if ($j < n – 1$)
4. if (a[$j$] < min)
    (a) min = a[$j$]
    (b) loc = $j$
5. if (loc ! = i)
    (a) swap = $a[i]$
    (b) $a[i] = a$[loc]
    (c) $a$[loc] = swap
6. display "the sorted numbers of array A"
7. Exit
```

PROGRAM 6.3

```
//PROGRAM TO IMPLEMENT SELECTION SORT
//USING STATIC MEMORY ALLOCATION, THAT IS USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>

#define MAX 20

void main()
{
        int arr[MAX], i,j,k,n,temp,smallest;
        clrscr();
        printf ("\nEnter the number of elements : ");
        scanf ("%d", & n);
        for (i = 0; i < n; i++)
        {
                printf ("\nEnter element %d : ",i+1);
                scanf ("%d", &arr[i]);
        }
        printf ("\nUnsorted list is : \n");
        for (i = 0; i < n; i++)
                printf ("%d ", arr[i]);
        printf ("\n");
        /*Selection sort*/
        for (i = 0; i < n – 1 ; i++)
        {
                /*Find the smallest element*/
                smallest = i;
                for(k = i + 1; k < n ; k++)
                {
                        if (arr[smallest] > arr[k])
                                smallest = k ;
                }
                if ( i != smallest )
                {
                        temp = arr [i];
                        arr[i] = arr[smallest];
                        arr[smallest] = temp ;
                }
                printf (“\nAfter Pass %d elements are :  ”,i+1);
                for (j = 0; j < n; j++)
                        printf (“%d ”, arr[ j]);
                printf (“\n”);
        }/*End of for*/
        printf (“\nSorted list is : \n”);
        for (i = 0; i < n; i++)
                printf (“%d ”, arr[i]);
        getch();
}/*End of main()*/
```

PROGRAM 6.4

```
//PROGRAM TO IMPLEMENT SELECTION SORT
//USING DYNAMIC MEMORY ALLOCATION
//CODED AND COMPILED USING TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>

//this function will sort the input elements using selection sort
void selectionsort(int *a,int n)
{
        int i,j,temp;
        for(i=0;i< n–1;i++)
        for(j=i+1;j < n;j++)
        if (a[i]>a[j])
        {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
        }
}

void main()
{
        int *a,n,*l,*temp;
        clrscr();
        printf ("\nEnter the number of elements\n");
        scanf ("%d",&n);

        //dynamically allocate the memory array block
        a=((int*)malloc(n*sizeof (int)));
        temp=a;
        l=a+n;
        printf ("\nEnter the elements\n");
        while(a < l)
        {
                scanf ("%d",a);
                a++;
        }

        //calling the selection sort fucntion
        selectionsort(temp,n);
        printf ("\nSorted array :  ");
        a=temp;
        while(a < l)
        {
                printf (" %d",*a);
                a++;
        }
        getch();
}
```

TIME COMPLEXITY

Time complexity of a selection sort is calculated in terms of the number of comparisons $f(n)$. In the first pass it makes $n - 1$ comparisons; the second pass makes $n - 2$

comparisons and so on. The outer *for loop* iterates for (n - 1) times. But the inner loop iterates for $n*(n-1)$ times to complete the sorting.

$$\begin{aligned} f(n) &= (n-1) + (n-2) + \ldots\ldots + 2 + 1 \\ &= (n(n-1))/2 \\ &= O(n_2) \end{aligned}$$

Readers can go through the algorithm and analyse it for different types of input to find their (efficiency) Time complexity for best, worst and average case. That is in best case how the algorithm is performed for sorted arrays as input.

In worst case we can analyse how the algorithm is performed for reverse sorted array as input. Average case is where we input general (mixed sorted) input to the algorithm. Following table will summarise the efficiency of algorithm in different case :

Best case	*Worst case*	*Average case*
$n - 1 = O(n)$	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n)$

6.4. INSERTION SORT

Insertion sort algorithm sorts a set of values by inserting values into an existing sorted file. Compare the second element with first, if the first element is greater than second, place it before the first one. Otherwise place is just after the first one. Compare the third value with second. If the third value is greater than the second value then place it just after the second. Otherwise place the second value to the third place. And compare third value with the first value. If the third value is greater than the first value place the third value to second place, otherwise place the first value to second place. And place the third value to first place and so on.

Let A be a linear array of n numbers A [1], A [2], A [3], ... A[n]. The algorithm scan the array A from A [1] to A [n], by inserting each element A[k], into the proper position of the previously sorted sub list. A [1], A [2], A [3], ... A [$k - 1$]

Step 1: As the single element A [1] by itself is sorted array.

Step 2: A [2] is inserted either before or after A [1] by comparing it so that A[1], A[2] is sorted array.

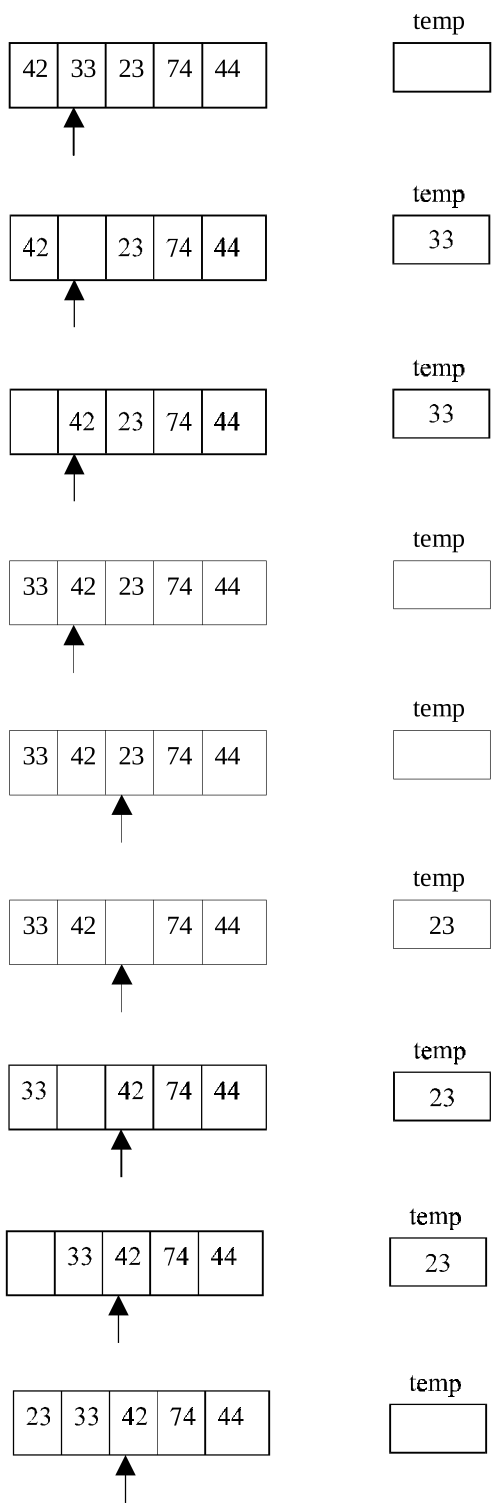
Step 3: A [3] is inserted into the proper place in A [1], A [2], that is A [3] will be compared with A [1] and A [2] and placed before A [1], between A [1] and A [2], or after A [2] so that A [1], A [2], A [3] is a sorted array.

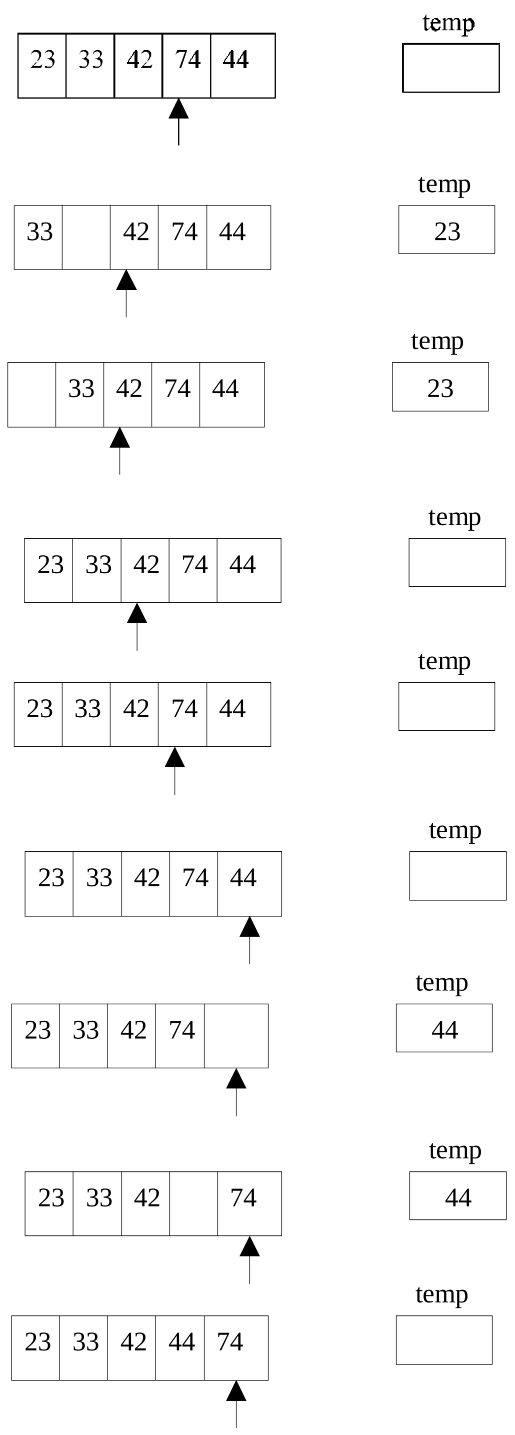
Step 4: A [4] is inserted in to a proper place in A [1], A [2], A [3] by comparing it; so that A [1], A [2], A [3], A [4] is a sorted array.

Step 5: Repeat the process by inserting the element in the proper place in array

Step n : A [n] is inserted into its proper place in an array A [1], A [2], A [3], ... A [n – 1] so that A [1], A [2], A [3], ... ,A [n] is a sorted array.

To illustrate the insertion sort methods, consider a following array with five elements 42, 33, 23, 74, 44 :


42 33 23 74 44
temp
42 23 74 44
temp 33
42 23 74 44
temp 33
33 42 23 74 44
temp
33 42 23 74 44
temp
33 42 74 44
temp 23
33 42 74 44
temp 23
33 42 74 44
temp 23
23 33 42 74 44
temp


23 33 42 74 44 temp
33 42 74 44 temp 23
33 42 74 44 temp 23
23 33 42 74 44 temp
23 33 42 74 44 temp
23 33 42 74 44 temp
23 33 42 74 temp 44
23 33 42 74 temp 44
23 33 42 44 74 temp

ALGORITHM

Let A be a linear array of n numbers A [1], A [2], A [3], ... ,A [n]......Swap be a temporary variable to interchange the two values. Pos is the control variable to hold the position of each pass.

```
1. Input an array A of n numbers
2. Initialize i = 1 and repeat through steps 4 by incrementing i by one.
    (a) If (i < = n – 1)
    (b) Swap = A [I],
    (c) Pos = i – 1
3. Repeat the step 3 if (Swap < A[Pos] and (Pos >= 0))
    (a) A [Pos+1] = A [Pos]
    (b) Pos = Pos-1
4. A [Pos +1] = Swap
5. Exit
```

PROGRAM 6.5

```
//PROGRAM TO IMPLEMENT INSERTION SORT USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>

#define MAX 20

void main()
{
        int arr[MAX],i,j,k,n;
        clrscr();
        printf (“\nEnter the number of elements : ”);
        scanf (“%d”,&n);
        for (i = 0; i < n; i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”, &arr[i]);
        }
        printf (“\nUnsorted list is :\n”);
        for (i = 0; i < n; i++)
                printf (“%d ”, arr[i]);
        printf (“\n”);
        /*Insertion sort*/
        for(j=1;j < n;j++)
        {
                k=arr[j]; /*k is to be inserted at proper place*/
                for(i=j–1;i>=0 && k<arr[i];i--)
                        arr[i+1]=arr[i];
                arr[i+1]=k;
                printf ("\nPass %d, Element inserted in proper place: %d\n",j,k);
                for (i = 0; i < n; i++)
                        printf ("%d ", arr[i]);
                printf ("\n");
        }
        printf ("\nSorted list is :\n");
        for (i = 0; i < n; i++)
                printf ("%d ", arr[i]);
        getch();
}/*End of main()*/
```

TIME COMPLEXITY

In the insertion sort algorithm $(n - 1)$ times the loop will execute for comparisons and interchanging the numbers. The inner while loop iterates maximum of $((n - 1) \times (n - 1))/2$ times to computing the sorting.

WORST CASE

The worst case occurs when the array A is in reverse order and the inner while loop must use the maximum number $(n - 1)$ of comparisons. Hence

$$\begin{aligned} f(n) &= (n - 1) + \ldots\ldots\ 2 + 1 \\ &= (n\ (n - 1))/2 \\ &= O(n^2). \end{aligned}$$

AVERAGE CASE

On the average case there will be approximately $(n - 1)/2$ comparisons in the inner while loop. Hence the average case

$$\begin{aligned} f\ (n) &= (n - 1)/2 + \ldots\ldots + 2/2 + 1/2 \\ &= n\ (n - 1)/4 \\ &= O(n^2). \end{aligned}$$

BEST CASE

The best case occurs when the array A is in sorted order and the outer for loop will iterate for $(n - 1)$ times. And the inner while loop will not execute because the given array is a sorted array

i.e., $f\ (n) = O(n)$.

6.5. SHELL SORT

Shell Sort is introduced to improve the efficiency of simple insertion sort. Shell Sort is also called *diminishing increment sort.* In this method, sub-array, contain *k*th element of the original array, are sorted.

Let A be a linear array of *n* numbers A [1], A [2], A [3], ... A [n].

Step 1: The array is divided into *k* sub-arrays consisting of every *k*th element. Say *k* = 5, then five sub-array, each containing one fifth of the elements of the original array.

Sub array 1 → A[0]	A[5]	A[10]
Sub array 2 → A[1]	A[6]	A[11]
Sub array 3 → A[2]	A[7]	A[12]
Sub array 4 → A[3]	A[8]	A[13]
Sub array 5 → A[4]	A[9]	A[14]

Note : The ith element of the *j*th sub array is located as A [(*i*–1) × *k*+*j*–1]

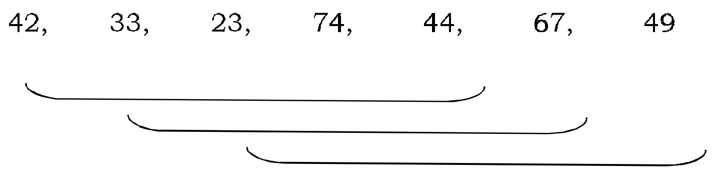
Step 2: After the first *k* sub array are sorted (usually by insertion sort) , a new smaller value of *k* is chosen and the array is again partitioned into a new set of sub arrays.

Step 3: And the process is repeated with an even smaller value of *k*, so that A [1], A [2], A [3], ... A [*n*] is sorted.

To illustrate the shell sort, consider the following array with 7 elements 42, 33, 23, 74, 44, 67, 49 and the sequence K = 4, 2, 1 is chosen.

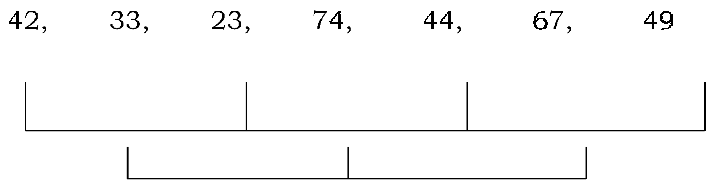
Pass = 1

Span = *k* = 4



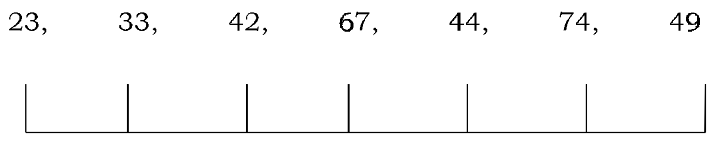
Pass = 2

span = *k* = 2



Pass = 3

Span = *k* = 1



23, 33, 42, 44, 49, 67, 74

ALGORITHM

Let A be a linear array of *n* elements, A [1], A [2], A [3], ... A[*n*] and *Incr* be an array of sequence of span to be incremented in each pass. X is the number of elements in the array *Incr*. *Span* is to store the span of the array from the array Incr.

```
1. Input n numbers of an array A
2. Initialise i = 0 and repeat through step 6 if (i < x)
3. Span = Incr[i]
4. Initialise j = span and repeat through step 6 if ( j < n)
    (a) Temp = A [ j ]
5. Initialise k = j-span and repeat through step 5 if (k > = 0) and (temp < A [k ])
    (a) A [k + span] = A [k]
6. A [k + span] = temp
7. Exit
```

PROGRAM 6.6

```
//PROGRAM TO IMPLEMENT SHELL SORT USING ARRAYS
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>

#define MAX 20

void main()
{
        int arr[MAX], i,j,k,n,incr;
        clrscr();
        printf (“\nEnter the number of elements : ”);
        scanf (“%d”,&n);
        for (i=0;i < n;i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”,&arr[i]);
        }
        printf (“\nUnsorted list is :\n”);
        for (i = 0; i < n; i++)
                printf (“%d ”, arr[i]);
        printf (“\nEnter maximum increment (odd value) : ”);
        scanf (“%d”,&incr);
        /*Shell sort algorithm is applied*/
        while(incr>=1)
        {
                for (j=incr;j<n;j++)
                {
                        k=arr[ j];
                        for(i = j–incr; i > = 0 && k < arr[i]; i = i–incr)
                                arr[i+incr]=arr[i];
                        arr[i+incr]=k;
                }
                printf (“\nIncrement=%d \n”,incr);
                for (i = 0; i < n; i++)
                        printf(“%d ”, arr[i]);
                printf (“\n”);
                incr=incr–2; /*Decrease the increment*/
        }/*End of while*/
        printf (“\nSorted list is :\n”);
        for (i = 0; i < n; i++)
                printf (“%d ”, arr[i]);
        getch();
}/*End of main()*/
```

TIME COMPLEXITY

The detailed efficiency analysis of the shell sort is mathematically involved and beyond the scope of this book. The time complexity depends on the number of elements in the array increment (*i.e.*, number of spans) and on their actual values. One requirement is that the elements of increments should be relatively prime (*i.e.*, no common divisor other than 1). This guarantees that after successive iteration of the sub arrays, the entire array is almost sorted when the span = 1 (*i.e.*, in the last iteration). If an appropriate sequence of increments is classified, then the order of the shell sort is

$$f(n) = O(n (\log n)^2)$$

6.6. QUICK SORT

It is one of the widely used sorting techniques and it is also called the partition-exchange sort. Quick sort is an efficient algorithm and it passes a very good time complexity in average case. This is developed by C.A.R. Hoare. It is an algorithm of the divide-and-conquer type.

The quick sort algorithm works by partitioning the array to be sorted. And each partitions are internally sorted recursively. In partition the first element of an array is chosen as a key value. This key value can be the first element of an array. That is, if A is an array then key = A (0), and rest of the elements are grouped into two portions such that, (*a*) One partition contains elements smaller than key value

(*b*) Another partition contains elements larger than the key value

Two pointers, up and low, are initialized to the upper and lower bounds of the sub array. During execution, at any point each element in a position above up is greater than or equal to key value and each element in a position below low pointer is less than or equal to the key. Up pointer will move in a decrement and low in an increment fashion.

Let A be an array A [1], A [2], A [3], ... ,A [n] of n number to be sorted. Then,

Step 1: Choose the first elements of the array (or sub-array) as the key *i.e.*, key = A(1).

Step 2: Place the low pointer in second position of the array and up pointer in the last position of the array *i.e.*, low = 2 and up = *n*.

Step 3: Repeatedly increase the pointer low by one position until (A [low] > key).

Step 4: Repeatedly decrease the pointer up by one position until (A [up] < = key).

Step 5: If up > low, interchange A [low] with A [up], Swap = A [low]; A [low] = A [up]; A [up] = Swap.

Step 6: Repeat the step 3, 4, and 5 until the condition in step 5 fails (*i.e.*, up < = low), then interchange A [up] with key.

NOTE: Now the given array is partitioned into two sub-arrays. The sub-array A [1], A [2], A [3], ... ,A [*k*−1) is less than A [*k*] (*i.e.*, key) and the second sub array A [*k* + 1], A [*k* + 2], A [k + 3], ... ,A [*n*) which is greater than the key value A (*k*). We can repeatedly apply this procedure on each of these sub-arrays until the entire array is sorted.

To illustrate the quick sort algorithm, consider the following array with 7 elements 42, 33, 23, 74, 44, 67, 49. Select the first value of the array as key, here the key = 42. and locate the low and up pointers.

```
       low→                     up
42,    33 ,   23,   74,   44,   67,   49
```

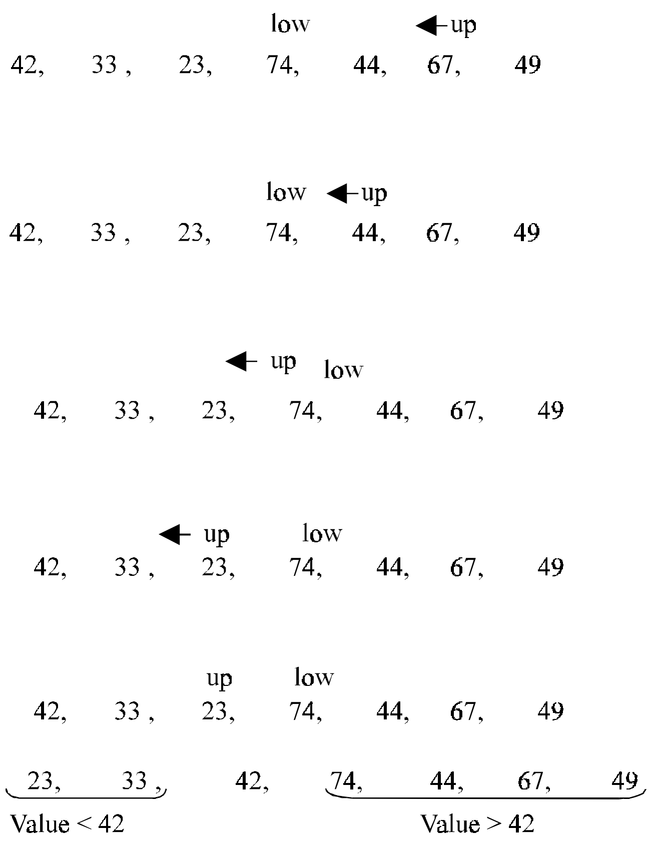
Move the low pointers by repeatedly incrementing the pointer by one position until (A[low] > key).

```
              low→              up
42,    33 ,   23,   74,   44,   67,   49
```

```
                    low→         up
42,    33 ,   23,   74,   44,   67,   49
```

Here (A [low] > key) *i.e.*, 74 > 42. Now decrease the pointer up by one position until (A [up] < = key).

```
                    low               ←up
42,    33 ,   23,   74,   44,   67,   49
```



ALGORITHM

Let A be a linear array of *n* elements A (1), A (2), A (3)......A (*n*), low represents the lower bound pointer and up represents the upper bound pointer. Key represents the first element of the array, which is going to become the middle element of the sub-arrays.

```
1. Input n number of elements in an array A
2. Initialize low = 2, up = n , key = A[(low + up)/2]
3. Repeat through step 8 while (low < = up)
4. Repeat step 5 while(A [low] > key)
5. low = low + 1
6. Repeat step 7 while(A [up] < key)
7. up = up–1
8. If (low < = up)
   (a) Swap = A [low]
   (b) A [low] = A [up]
   (c) A [up] = swap
   (d) low=low+1
   (e) up=up–1
 9. If (1 < up) Quick sort (A, 1, up)
10. If (low < n) Quick sort (A, low, n)
11. Exit
```

PROGRAM 6.7

```
//PROGRAM TO IMPLEMENT QUICK SORT
//USING ARRAYS RECURSIVELY
//CODED AND COMPILED IN TURBO C

#include<conio.h>
#include<stdio.h>

#define MAX 30

enum bool {FALSE,TRUE};

//Function display the array
void display(int arr[],int low,int up)
{
        int i;
        for(i=low;i<=up;i++)
                printf (“%d ”,arr[i]);
}

//This function will sort the array using Quick sort algorithm
void quick(int arr[],int low,int up)
{
        int piv,temp,left,right;
        enum bool pivot_placed=FALSE;
        //setting the pointers
        left=low;
        right=up;
        piv=low; /*Take the first element of sublist as piv */

        if (low>=up)
                return;
        printf (“\nSublist : ”);
        display(arr,low,up);
        /*Loop till pivot is placed at proper place in the sublist*/
        while(pivot_placed==FALSE)
        {
                /*Compare from right to left */
                while( arr[piv]<=arr[right] && piv!=right )
                        right=right–1;
                if ( piv==right )
                      pivot_placed=TRUE;
                if ( arr[piv] > arr[right] )
                {
                        temp=arr[piv];
                        arr[piv]=arr[right];
                        arr[right]=temp;
                        piv=right;
                }
                /*Compare from left to right */
                while( arr[piv]>=arr[left] && left!=piv )
                        left=left+1;
                if (piv==left)
                        pivot_placed=TRUE;
                if ( arr[piv] < arr[left] )
                {
                        temp=arr[piv];
                        arr[piv]=arr[left];
                        arr[left]=temp;
                        piv=left;
                }
        }/*End of while */

        printf (“-> Pivot Placed is %d -> ”,arr[piv]);
        display(arr,low,up);
        printf ("\n");
        quick(arr,low,piv–1);
        quick(arr,piv+1,up);
}/*End of quick()*/

void main()
{
        int array[MAX],n,i;
        clrscr();
        printf (“\nEnter the number of elements : ”);
        scanf (“%d”,&n);
        for (i=0;i<n;i++)
        {
                printf ("\nEnter element %d : ",i+1);
                scanf ("%d",&array[i]);
        }

        printf ("\nUnsorted list is :\n");
        display(array,0,n–1);
        printf ("\n");

        quick (array,0,n–1);
        printf ("\nSorted list is :\n");
        display(array,0,n–1);
        getch();
}/*End of main() */
```

TIME COMPLEXITY

The time complexity of quick sort can be calculated for any of the following case. It is usually measured by the number $f(n)$ of comparisons required to sort n elements.

WORST CASE

The worst case occurs when the list is already sorted. In this case the given array is partitioned into two sub arrays. One of them is an empty array and another one is an array. After partition, when the first element is checked with other element, it will take n comparison to recognize that it remain in the position so as $(n - 1)$ comparisons for the second position.

$$\begin{aligned} f(n) &= n + (n - 1) + \ldots\ldots + 2 + 1 \\ &= (n(n + 1))/2 \\ &= O(n^2) \end{aligned}$$

AVERAGE CASE

In this case each reduction step of the algorithm produces two sub arrays. Accordingly :

(*a*) Reducing the array by placing one element and produces two sub arrays.

(*b*) Reducing the two sub-arrays by placing two elements and produces four sub-arrays.

(*c*) Reducing the four sub-arrays by placing four elements and produces eight sub-arrays.

And so on. Reduction step in the kth level finds the location at 2^{k-1} elements; however there will be approximately $\log_2 n$ levels at reduction steps. Furthermore each level uses at most n comparisons,

so $\quad f(n) = O(n \log n)$

BEST CASE

The base case analysis occurs when the array is always partitioned in half, That key = A [(low+up)/2]

$$\begin{aligned} f(n) &= Cn + f(n/2) + f(n/2) \\ &= Cn + 2f(n/2) \\ &= O(n) \qquad \text{where C is a constant.} \end{aligned}$$

6.7. MERGE SORT

Merging is the process of combining two or more sorted array into a third sorted array. It was one of the first sorting algorithms used on a computer and was developed by John Von Neumann. Divide the array into approximately $n/2$ sub-arrays of size two and set the element in each sub array. Merging each sub-array with the adjacent sub-array will get another sorted sub-array of size four. Repeat this process until there is only one array remaining of size n.

Since at any time the two arrays being merged are both sub-arrays of A, lower and upper bounds are required to indicate the sub-arrays of a being merged. l1 and $u1$ represents the lower and upper bands of the first sub-array and l2 and $u2$ represents the lower and upper bands of the second sub-array respectively.

Let A be an array of n number of elements to be sorted A[1], A[2] ...... A[n].

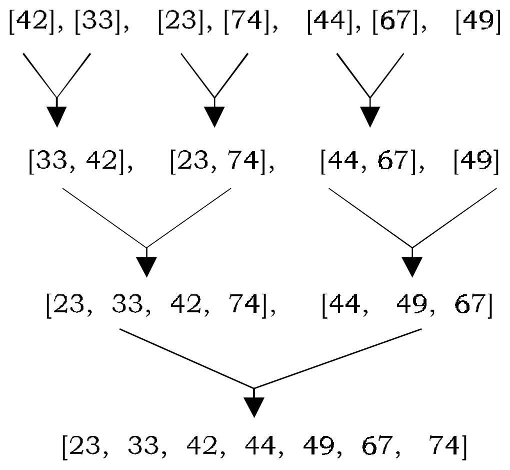
Step 1: Divide the array A into approximately $n/2$ sorted sub-array of size 2. *i.e.*, the elements in the (A [1], A [2]), (A [3], A [4]), (A [k], A [k + 1]), (A [n – 1], A [n]) sub-arrays are in sorted order.

Step 2: Merge each pair of pairs to obtain the following list of sorted sub-array of size 4; the elements in the sub-array are also in the sorted order.

(A [1], A [2], A [3], A [4]),...... (A [k – 1], A [k], A [k + 1], A [k + 2]),
...... (A [n – 3], A [n – 2], A [n – 1], A [n].

Step 3: Repeat the step 2 recursively until there is only one sorted array of size n.

To illustrate the merge sort algorithm consider the following array with 7 elements [42], [33], [23], [74], [44], [67], [49]



ALGORITHM

Let A be a linear array of size *n*, A [1], A [2], A [3] …… A [*n*], l1 and *u*1 represent lower and upper bounds of the first sub-array and l2 and *u*2 represent the lower and upper bound of the second sub-array. *Aux* is an auxiliary array of size *n*. *Size* is the *sizes* of merge files.

```
1. Input an array of n elements to be sorted
2. Size = 1
3. Repeat through the step 13 while (Size < n)
    (a) set l1 = 0; k = 0
4. Repeat through step 10 while ((l1+Size) < n)
    (a) l2 = l1+Size
    (b) u1 = l2–1
5. If ((l2+Size–1) < n)
        (i) u2 = l2+Size–1
    (b) Else
        (i) u2 = n-1
6. Initialize i = l1; j = l2 and repeat through step 7 if (i <= u1) and ( j < = u2)
7. If (A [i] < = A[j])
        (i) Aux [k] = A [i++]
    (b) Else
        (i) Aux [k] = A [j++]
8. Repeat the step 8 by incrementing the value of k until (i < = u1)
    (a) Aux [k] = A [I++]
9. Repeat the step 9 by incrementing the value of k until ( j < = u2)
    (a) Aux [k] = A [ j++]
10. L1=u2+1
11. Initialize I = l1 and repeat the step 11 if (k < n) by incrementing I by one
    (a) Aux [k++] = A[I]
12. Initialize I=0 and repeat the step 12 if (I < n) by incrementing I by one
    (a) A [i] = A [I]
13. Size = Size*2
14. Exit
```

PROGRAM 6.8

```
//PROGRAM TO IMPLEMENT MERGING OF TWO SORTED ARRAYS
//INTO A THIRD SORTED ARRAY
//CODED AND COMPILED IN TURBO C
```

```
#include<conio.h>
#include<stdio.h>

void main()
{
        int arr1[20],arr2[20],arr3[40];
        int i,j,k;
        int max1,max2;
        clrscr();

        printf (“\nEnter the number of elements in list1 : ”);
        scanf (“%d”,&max1);
        printf (“\nTake the elements in sorted order :\n”);
        for (i=0;i<max1;i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”,&arr1[i]);
        }
        printf (“\nEnter the number of elements in list2 : ”);
        scanf (“%d”,&max2);
        printf (“\nTake the elements in sorted order :\n”);
        for (i=0;i<max2;i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”,&arr2[i]);
        }
        /* Merging */
        i=0;    /*Index for first array*/
        j=0;    /*Index for second array*/
        k=0;    /*Index for merged array*/

        while( (i < max1) && (j < max2) )
        {
                if ( arr1[i] < arr2[j] )
                        arr3[k++]=arr1[i++];
                else
                        arr3[k++]=arr2[j++];
        }/*End of while*/
        /*Put remaining elements of arr1 into arr3*/
        while( i < max1 )
                arr3[k++]=arr1[i++];
        /*Put remaining elements of arr2 into arr3*/
        while( j < max2 )
                arr3[k++]=arr2[j++];

        /*Merging completed*/
        printf ("\nList 1 :  ");
        for (i=0;i<max1;i++)
                printf ("%d ",arr1[i]);
        printf ("\nList 2 :  ");
        for (i=0;i<max2;i++)
                printf("%d ",arr2[i]);
        printf ("\nMerged list : ");
        for( i=0;i<max1+max2;i++)
                printf ("%d ",arr3[i]);
        getch();
}/*End of main()*/
```

PROGRAM 6.9

```
//PROGRAM TO IMPLEMENT MERGE SORT WITHOUT RECURSION
//CODED AND COMPILED IN TURBO C
#include<stdio.h>
#include<conio.h>

#define MAX 30

void main()
{
        int arr[MAX],temp[MAX],i,j,k,n,size,l1,h1,l2,h2;
        clrscr();
        printf ("\nEnter the number of elements : ");
        scanf ("%d",&n);
        for (i=0;i<n;i++)
        {
                printf ("\nEnter element %d : ",i+1);
                scanf ("%d",&arr[i]);
        }
        printf ("\nUnsorted list is : ");
        for ( i = 0 ; i<n ; i++)
                printf("%d ", arr[i]);

        /*l1 lower bound of first pair and so on*/
```

```
for (size=1; size < n; size=size*2 )
{
        l1 = 0;
        k = 0;  /*Index for temp array*/
        while( l1+size < n)
        {
                h1=l1+size–1;
                l2=h1+1;
                h2=l2+size–1;
                if ( h2>=n ) /* h2 exceeds the limlt of arr */
                        h2=n-1;
                /*Merge the two pairs with lower limits l1 and l2*/
                i=l1;
                j=l2;
                while(i<=h1 && j<=h2 )
                {
                        if ( arr[i] <= arr[ j] )
                                temp[k++]=arr[i++];
                        else
                                temp[k++]=arr[ j++];
                }
                while(i<=h1)
                        temp[k++]=arr[i++];
                while( j<=h2)
                        temp[k++]=arr[ j++];
                /**Merging completed**/
                l1 = h2+1; /*Take the next two pairs for merging */
        }/*End of while*/

        for (i=l1; k<n; i++) /*any pair left */
                temp[k++]=arr[i];

        for(i=0;i<n;i++)
             arr[i]=temp[i];

        printf (“\nSize=%d \nElements are : ”,size);
        for ( i = 0 ; i<n ; i++)
                printf (“%d ”, arr[i]);
}/*End of for loop */
printf (“\nSorted list is :\n”);
for ( i = 0 ; i<n ; i++)
        printf (“%d ”, arr[i]);
        getch();
}/*End of main()*/
```

PROGRAM 6.10

```
//PROGRAM TO IMPLEMENT MERGE SORT THROUGH RECURSION
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>

#define MAX 20

int array[MAX];
//Function to merge the sub  files or arrays
void merge(int low, int mid, int high )
{
        int temp[MAX];
        int i = low;
        int j = mid +1 ;
        int k = low ;

        while( (i < = mid) && (j < =high) )
        {
                if (array[i] < = array[ j])
                        temp[k++] = array[i++] ;
                else
                        temp[k++] = array[ j++] ;
        }/*End of while*/
        while( i <= mid )
                temp[k++]=array[i++];
        while( j <= high )
                temp[k++]=array[j++];

        for (i= low; i < = high ; i++)
                array[i]=temp[i];
}/*End of merge()*/

//Function which call itself to sort an array
void merge_sort(int low, int high )
```

```
{
        int mid;
        if ( low ! = high )
        {
                mid = (low+high)/2;
                merge_sort( low , mid );
                merge_sort( mid+1, high );
                merge(low, mid, high );
        }
}/*End of merge_sort*/

void main()
{
        int i,n;
        clrscr();
        printf (“\nEnter the number of elements : ”);
        scanf (“%d”,&n);
        for (i=0;i<n;i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”,&array[i]);
        }

        printf (“\nUnsorted list is :\n”);
        for ( i = 0 ; i<n ; i++)
                printf (“%d ”, array[i]);

        merge_sort( 0, n–1);

        printf (“\nSorted list is :\n”);
        for ( i = 0 ; i<n ; i++)
                printf (“%d ”, array[i]);
        getch();
}/*End of main()*/
```

TIME COMPLEXITY

Let $f(n)$ denote the number of comparisons needed to sort an n element array A using the merge sort algorithm. The algorithm requires almost log n passes, each involving n or fewer comparisons.

In average and worst case the merge sort requires $O(n \log n)$ comparisons.

The main drawback of merge sort is that it requires $O(n)$ additional space for the auxiliary array.

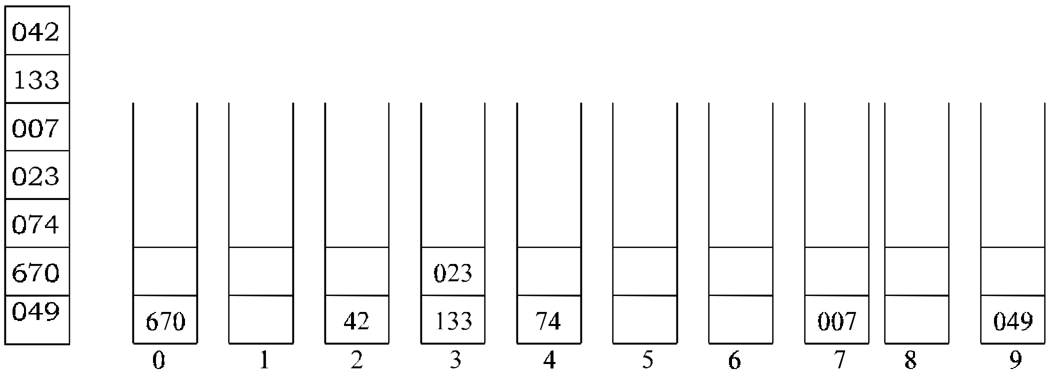
6.8. RADIX SORT

Radix sort or bucket sort is a method that can be used to sort a list of numbers by its base. If we want to sort list of English words, where radix or base is 26, then 26 buckets is used to sort the words.

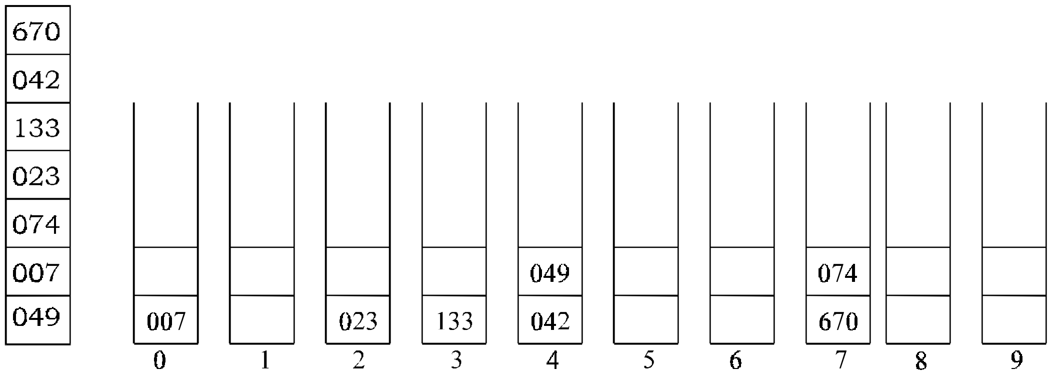
To sort an array of decimal numbers, where the radix or base is 10, we need 10 buckets and can be numbered as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Number of passes required to have a sorted array depends upon the number of digits in the largest element. To illustrate the radix sort, consider the following array with 7 elements :

42, 133, 7, 23, 74, 670, 49

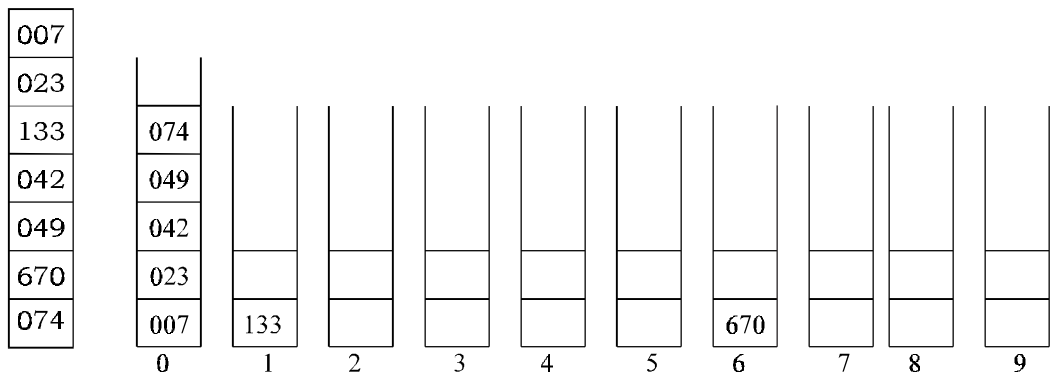
In this array the biggest element is 670 and the number of digit is 3. So 3 passes are required to sort the array. Read the element(s) and compare the first position (2 is in first position of 42) digit with the digit of the bucket and place it.



Now read the elements from left to right and bottom to top of the buckets and place it in array for the next pass. Read the array element and compare the second position (4 is in second position of the element 042) digit with the number of the bucket and place it.



Again read the element from left to right and from bottom to top to get an array for the third pass. (0 is in the third position of 042) Compare the third position digit in each element with the bucket digit and place it wherever it matches.



Read the element from the bucket from left to right and from bottom to top for the sorted array. *i.e.*, 7 23, 42, 49, 74, 133, 670.

ALGORITHM

Let A be a linear array of *n* elements A [1], A [2], A [3],...... A [*n*]. Digit is the total number of digits in the largest element in array A.

```
1. Input n number of elements in an array A.
2. Find the total number of Digits in the largest element in the array.
3. Initialise i = 1 and repeat the steps 4 and 5 until (i <= Digit).
4. Initialise the buckets j = 0 and repeat the steps (a) until (j < n)
   (a) Compare ith position of each element of the array with bucket number and
       place it in the corresponding bucket.
5. Read the element(s) of the bucket from 0th bucket to 9th bucket and from first
   position to higher one to generate new array A.
6. Display the sorted array A.
7. Exit.
```

PROGRAM 6.11

```
//PROGRAM TO IMPLEMENT RADIX SORT USING LINKED LIST
//CODED AND COMPILED IN TURBO C

#include<stdio.h>
#include<conio.h>
#include<malloc.h>

struct  node
{
        int info ;
        struct node *link;
}*start=NULL;

//Display the array elements
void display()
{
        struct node *p=start;
        while( p !=NULL)
        {
                printf (“%d ”, p->info);
                p= p->link;
        }
        printf (“\n”);
}/*End of display()*/

/* This function finds number of digits in the largest element of the list */
int large_dig(struct node *p)
{
        int large = 0,ndig = 0 ;

        while (p != NULL)
        {
                if (p ->info > large)
                        large = p->info;
                p = p->link ;
        }
        printf (“\nLargest Element is %d , ”,large);
        while (large != 0)
        {
                ndig++;
                large = large/10 ;
        }

        printf (“\nNumber of digits in it are %d\n”,ndig);
        return(ndig);
} /*End of large_dig()*/

/*This function returns kth digit of a number*/
int digit(int number, int k)
{
        int digit, i ;
        for (i = 1 ; i <=k ; i++)
        {
                digit = number % 10 ;
                number = number /10 ;
        }
        return(digit);
}/*End of digit()*/

//Function to implement the radix sort algorithm
void radix_sort()
{
        int i,k,dig,maxdig,mindig,least_sig,most_sig;
        struct node *p, *rear[10], *front[10];

        least_sig=1;
        most_sig=large_dig(start);

        for (k = least_sig; k <= most_sig ; k++)
        {
                printf ("\nPASS %d : Examining %dth digit from right   ",k,k);
                for(i = 0 ; i <= 9 ; i++)
                {
                        rear[i] = NULL;
                        front[i] = NULL ;
                }
                maxdig=0;
                mindig=9;
                p = start ;
                while( p != NULL)
                {
                        /*Find kth digit in the number*/
                        dig = digit(p->info, k);
                        if (dig>maxdig)
                                maxdig=dig;
                        if (dig<mindig)
                                mindig=dig;

                        /*Add the number to queue of dig*/
                        if (front[dig] == NULL)
                                front[dig] = p ;
                        else
                                rear[dig]->link = p ;
                        rear[dig] = p ;
                        p=p->link;/*Go to next number in the list*/
                }/*End while */
                /* maxdig and mindig are the maximum amd minimum
                   digits of the kth digits of all the numbers*/
                printf ("\nmindig=%d   maxdig=%d\n",mindig,maxdig);
                /*Join all the queues to form the new linked list*/
                start=front[mindig];
                for i=mindig;i<maxdig;i++)
                {
                        if (rear[i+1]!=NULL)
                                rear[i]->link=front[i+1];
                        else
                                rear[i+1]=rear[i];
                }
                rear[maxdig]->link=NULL;
                printf ("\nNew list : ");
                display();
        }/* End for */

}/*End of radix_sort*/

void main()
{
        struct node *tmp,*q;
        int i,n,item;
        clrscr();
        printf ("\nEnter the number of elements in the list : ");
        scanf ("%d", &n);

        for (i=0;i<n;i++)
        {
                printf ("\nEnter element %d : ",i+1);
                scanf ("%d",&item);

                /* Inserting elements in the linked list */
                tmp=(struct node*)malloc(sizeof(struct node));
                tmp->info=item;
                tmp->link=NULL;

                if (start==NULL) /* Inserting first element */
                        start=tmp;
                else
                {
                        q=start;
                        while(q->link!=NULL)
                                q=q->link;
                        q->link=tmp;
                }
        }/*End of for*/

        printf ("\nUnsorted list is :\n");
        display();
        radix_sort();
        printf ("\nSorted list is :\n");
        display ();
        getch();
}/*End of main()*/
```

TIME COMPLEXITY

Time requirement for the radix sorting method depends on the number of digits and the elements in the array. Suppose A is an array of n elements A1, A2..........An and let r denote the radix (for example r = 10 for decimal digits, r = 26 for English letters and r = 2 for bits). If Ai is the largest number then Ai can be represented as

$$Ai = a_{i\,s} \;\; a_{i\,s-1} \ldots\ldots a_{i\,k} \ldots\ldots a_{i\,2} \;\; a_{i\,1}$$

Then radix sort algorithm requires s passes. In pass k, $a_{i\,k}$ of the each element is compared with the bucket element. So radix sort requires the total comparison $f(n)$ of:

$$f(n) <= r \times s \times n$$

WORST CASE

In the worst case $s = n$ so

$$f(n) = O(n^2)$$

BEST CASE

In the best case $s = \log_d n$

So $f(n) = O(n \log n)$

AVERAGE CASE

In the average case, it is very hard to define the time complexity. Because it will depend on the choice of the radix r and also the number of digits on the largest element (*i.e.*, number of passes) But on an average ($\log_d n$) comparison is required. So

$$f(n) = O(n \log n)$$

In other words, radix sort performs well only when the number s of digits in the representation of the A_i is small. The main disadvantage of radix sort is that, it need $d \times n$

memory location to store bucket information. However this drawback may be minimized to 2 × *n* memory locations by using linked list rather than arrays.

6.9. HEAP

A heap is defined as an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value of the father. It can be sequentially represented as

$$A[j] <= A[(j - 1)/2]$$

for $0 <= [(j - 1)/2] < j <= n - 1$

The root of the binary tree (*i.e.*, the first array element) holds the largest key in the heap. This type of heap is usually called descending heap or mere heap, as the path from the root node to a terminal node forms an ordered list of elements arranged in descending order. Fig. 6.1 shows a heap.

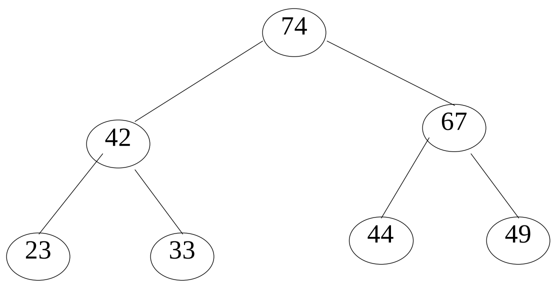


Fig. 6.1. Heap representation

74	42	67	23	33	44	49

Fig. 6.2. Sequential representation

We can also define an ascending heap as an almost complete binary tree in which the value of each node is greater than or equal to the value of its father. This root node has the smallest element of the heap. This type of heap is also called min heap.

6.9.1. HEAP AS A PRIORITY QUEUE

A heap is very useful in implementing priority queue. The priority queue is a data structure in which the intrinsic ordering of the data items determines the result of its basic operations. Primarily queues can be classified into two types:

1. Ascending priority queue
2. Descending priority queue

An ascending priority queue can be defined as a group of elements to which new elements are inserted arbitrarily but only the smallest element is deleted from it. On the other hand a descending priority queue can be defined as a group of elements to which new elements are inserted arbitrarily but only the largest element is deleted from it.

The implementation of the Heap as a priority queue is left to the readers.

6.9.2. HEAP SORT

A heap can be used to sort a set of elements. Let H be a heap with *n* elements and it can be sequentially represented by an array A. Inset an element *data* into the heap H as follows:

1. First place *data* at the end of H so that H is still a complete tree, but not necessarily a heap.

2. Then the data be raised to its appropriate place in H so that H is finally a heap.

To understand the concept of insertion of data into a heap is illustrated with following two examples:

INSERTING AN ELEMENT TO A HEAP

Consider the heap H in Fig. 6.1. Say we want to add a data = 55 to H.

Step 1: First we adjoin 55 as the next element in the complete tree as shown in Fig. 6.2. Now we have to find the appropriate place for 55 in the heap by rearranging it.

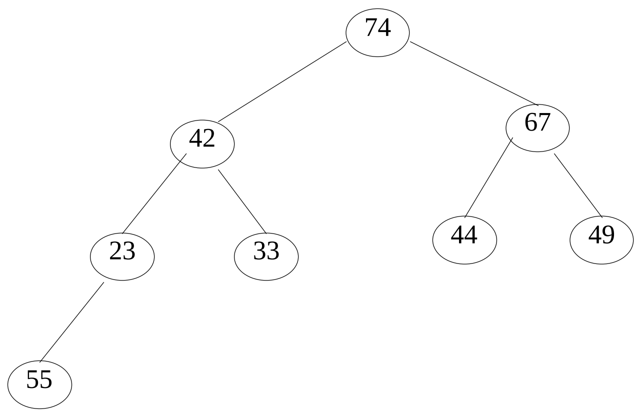


Fig. 6.2.

Step 2: Compare 55 with its parent 23. Since 55 is greater than 23, interchange 23 and 55. Now the heap will look like as in Fig. 6.3.

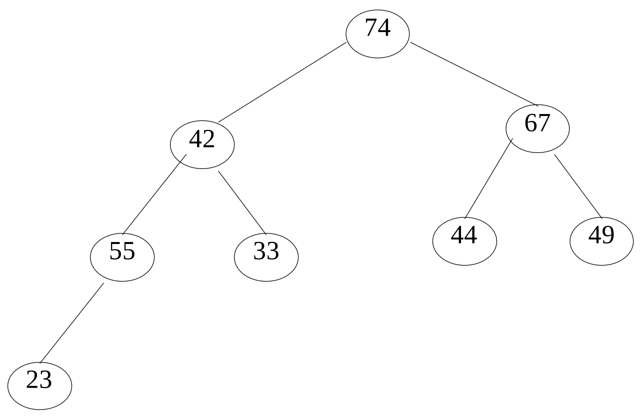


Fig. 6.3

Step 3: Compare 55 with its parent 42. Since 55 is greater than 42, interchange 55 and 42. Now the heap will look like as in Fig. 6.4.

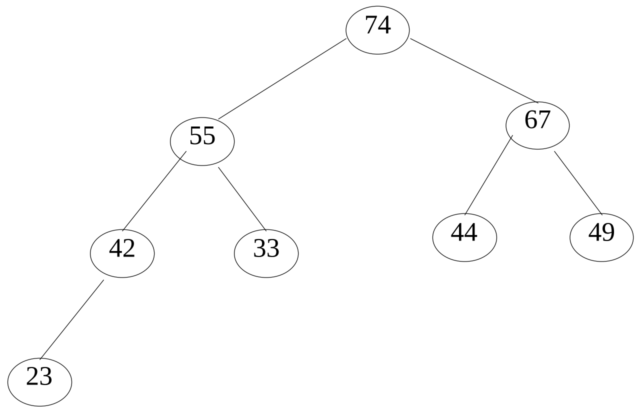


Fig. 6.4

Step 4: Compare 55 with its new parent 74. Since 55 is less than 74, it is the appropriate place of node 55 in the heap H. Fig. 6.4 shows the final heap tree.

CREATING A HEAP

A heap H can be created from the following list of numbers 33, 42, 67, 23, 44, 49, 74 as illustrated below :

Step 1: Create a node to insert the first number (*i.e.*, 33) as shown Fig 6:5

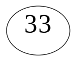


Fig. 6.5

Step 2: Read the second element and add as the left child of 33 as shown Fig. 6.6. Then restructure the heap if necessary.

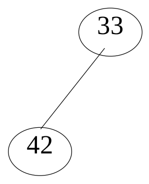


Fig. 6.6

Compare the 42 with its parent 33, since newly added node (*i.e.*, 42) is greater than 33 interchange node information as shown Fig. 6.7.

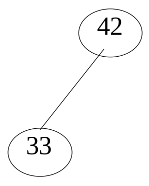


Fig. 6.7

Step 3: Read the 3rd element and add as the right child of 42 as shown Fig. 6.8. Then restructure the heap if necessary.

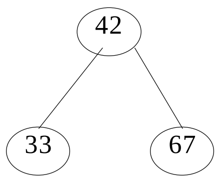


Fig. 6.7

Compare the 67 with its parent 42, since newly added node (*i.e.*, 67) is greater than 42 interchange node information as shown Fig. 6.8.

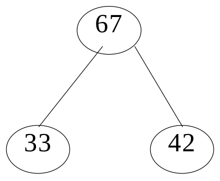


Fig. 6.8

Step 4: Read the 4th element and add as the left child of 33 as shown Fig. 6.9. Then restructure the heap if necessary.

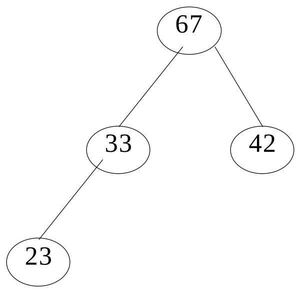


Fig. 6.9

Since newly added node (*i.e.*, 23) is less than its parent 33, no interchange.

Step 5: Read the 5th element and add as the right child of 33 as shown Fig. 6.10. Then restructure the heap if necessary.

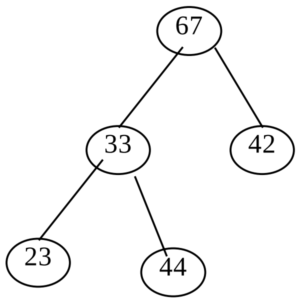


Fig. 6.10

Compare the 44 with its parent 33, since newly added node (*i.e.*, 44) is greater than 33 interchange node information as shown Fig. 6.11.

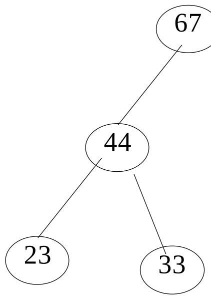


Fig. 6.11

Step 6: Read the 6th element and add as the left child of 42 as shown Fig. 6.12. Then restructure the heap if necessary.

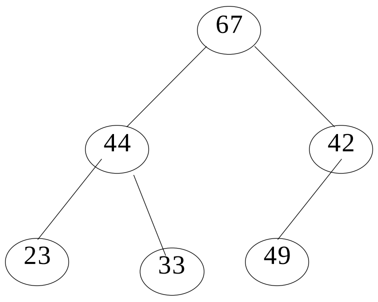


Fig. 6.12

Compare the (newly added node) 49 with its parent 42, since newly added node (*i.e.*, 49) is greater than 42 interchange node information as shown Fig. 6.13.

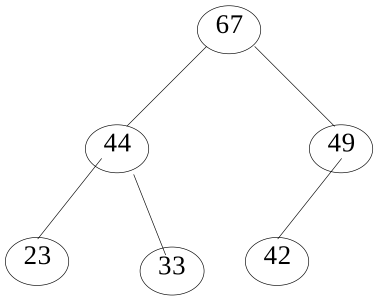


Fig. 6.13

Step 7: Read the 7th element and add as the left child of 49 as shown Fig. 6.14. Then restructure the heap if necessary.

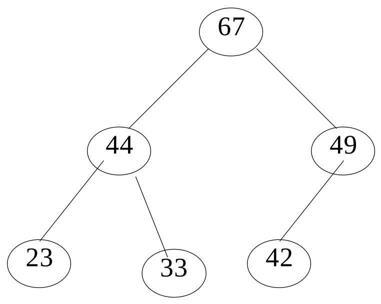


Fig. 6.14

Compare the (newly added node) 74 with its parent 49, since newly added node (*i.e.*, 74) is greater than 49 interchange node information as shown Fig. 6.15.

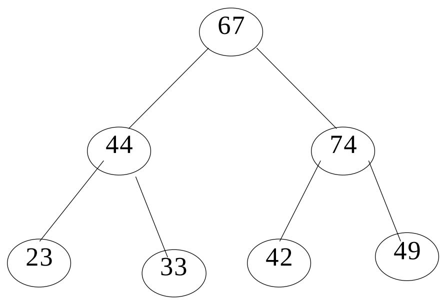


Fig. 6.15

Compare the recently changed node 74 with its parent 67, since it is greater than 67 interchange node information as shown Fig. 6.16.

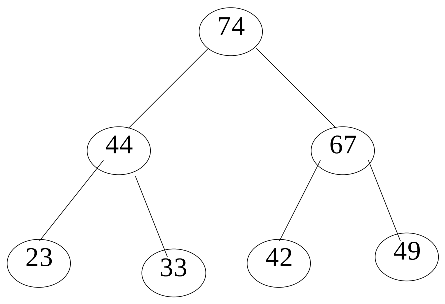


Fig. 6.16

ALGORITHM

Let H be a heap with *n* elements stored in the array HA. This procedure will insert a new element *data* in H. LOC is the present location of the newly added node. And PAR denotes the location of the parent of the newly added node.

```
1. Input n elements in the heap H.
2. Add new node by incrementing the size of the heap H: n = n + 1 and LOC = n
3. Repeat step 4 to 7 while (LOC < 1)
4. PAR = LOC/2
5. If (data <= HA[PAR])
(a) HA[LOC] = data
(b) Exit
6. HA[LOC] = HA[PAR]
7. LOC = PAR
8. HA[1] = data
9. Exit
```

DELETING THE ROOT OF A HEAP

Let H be a heap with *n* elements. The root R of H can be deleted as follows:

(*a*) Assign the root R to some variable data.

(*b*) Replace the deleted node R by the last node (or recently added node) L of H so that H is still a complete tree, but not necessarily a heap.

(*c*) Now rearrange H in such a way by placing L (new root) at the appropriate place, so that H is finally a heap.

Consider the heap H in Fig. 6.4 where R = 74 is the root and L = 23 is the last node (or recently added node) of the tree. Suppose we want to delete the root node R = 74. Apply the above rules to delete the root. Delete the root node R and assign it to *data* (*i.e.*, data = 74) as shown in Fig. 6.17.

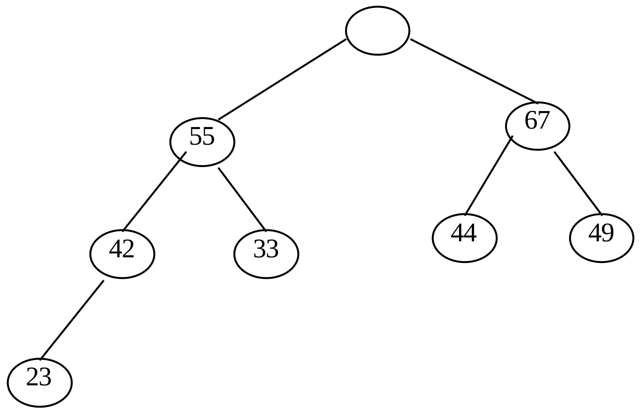


Fig. 6.17

Replace the deleted root node R by the last node L as shown in the Fig. 6.18.

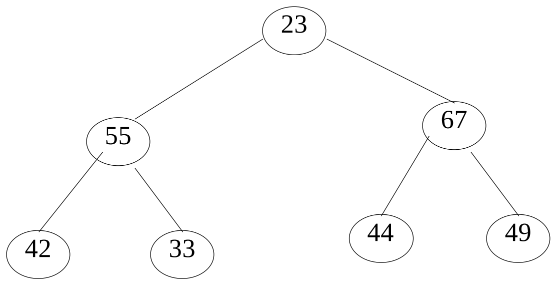


Fig. 6.18

Compare 23 with its new two children 55 and 67. Since 23 is less than the largest child 67, interchange 23 and 67. The new tree looks like as in Fig. 6.19.

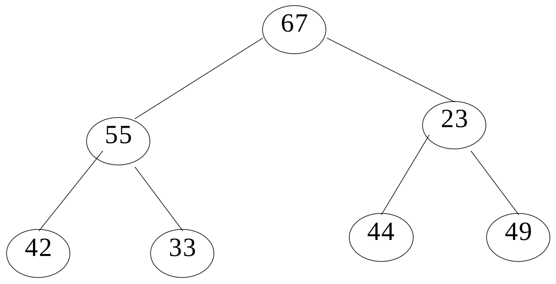


Fig. 6.19

Again compare 23 with its new two children, 44 and 49. Since 23 is less than the largest child 49, interchange 23 and 49as shown in Fig. 6.20.

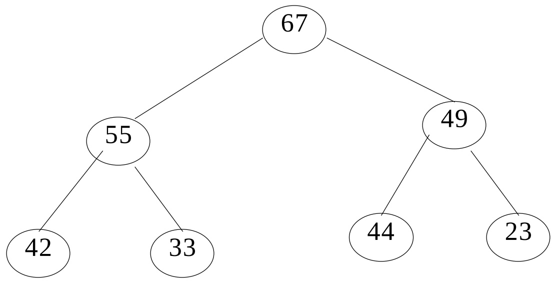


Fig. 6.20

The Fig. 6.20 is the required heap H without its original root R.

ALGORITHM

Let H be a heap with *n* elements stored in the array HA. *data* is the item of the node to be removed. *Last* gives the information about last node of H. The LOC, left, right gives the location of Last and its left and right children as the Last rearranges in the appropriate place in the tree.

```
 1. Input n elements in the heap H
 2. Data = HA[1]; last = HA[n] and n = n – 1
 3. LOC = 1, left = 2 and right = 3
 4. Repeat the steps 5, 6 and 7 while (right <= n)
 5. If (last >= HA[left]) and (last >= HA[right])
      (a) HA[LOC] = last
      (b) Exit
 6. If (HA[right] <= HA[left])
          (i) HA[LOC] = HA[left]
         (ii) LOC = left
     (b) Else
          (i) HA[LOC] = HA[right]
         (ii) LOC = right
 7. left = 2 × LOC; right = left +1
 8. If (left = n ) and (last < HA[left])
      (a) LOC = left
 9. HA[LOC] = last
10. Exit
```

PROGRAM 6.12

```
//PROGRAM TO IMPLEMENT HEAP SORT USING ARRAYS
//CODED AND IMPLEMENTED IN TURBO C

#include<conio.h>
#include<stdio.h>

int arr[20],n;

//Function to display the elements in the array
void display()
{      int i;
        for(i=0;i<n;i++)
                printf ("%d  ",arr[i]);
        printf ("\n");
}/*End of display()*/

//Function to insert an element to the heap
void insert(int num,int loc)
{
        int par;
        while(loc>0)
        {
                par=(loc–1)/2;
                if (num<=arr[par])
                {
                        arr[loc]=num;
                        return;
                }
                arr[loc]=arr[par];
                loc=par;
        }/*End of while*/
        arr[0]=num;
}/*End of insert()*/

//This function to create a heap
void create_heap()
{
        int i;
        for(i=0;i<n;i++)
                insert(arr[i],i);
}/*End of create_heap()*/

//Function to delete the root node of the tree
void del_root(int last)
{
        int left,right,i,temp;
        i=0; /*Since every time we have to replace root with last*/
        /*Exchange last element with the root */
        temp=arr[i];
        arr[i]=arr[last];
        arr[last]=temp;

        left=2*i+1; /*left child of root*/
        right=2*i+2;/*right child of root*/

        while( right < last)
        {
                if ( arr[i]>=arr[left] && arr[i]>=arr[right] )
                        return;
                if ( arr[right]<=arr[left] )
                {
                        temp=arr[i];
                        arr[i]=arr[left];
                        arr[left]=temp;
                        i=left;
                }
                else
                {
                        temp=arr[i];
                        arr[i]=arr[right];
                        arr[right]=temp;
                        i=right;
                }
                left=2*i+1;
                right=2*i+2;
        }/*End of while*/
        if ( left==last–1 && arr[i]<arr[left] )/*right==last*/
        {
                temp=arr[i];
                arr[i]=arr[left];
                arr[left]=temp;
        }
}/*End of del_root*/

//Function to sort an element in the heap
void heap_sort()
{
        int last;
        for(last=n–1; last>0; last--)
           del_root(last);
}/*End of del_root*/

void main()
{
        int i;
        clrscr();
        printf (“\nEnter number of elements : ”);
        scanf (“%d”,&n);
        for(i=0;i<n;i++)
        {
                printf (“\nEnter element %d : ”,i+1);
                scanf (“%d”,&arr[i]);
        }
        printf ("\nEntered list is :\n");
        display();

        create_heap();

        printf ("\nHeap is :\n");
        display();

        heap_sort();
        printf ("\nSorted list is :\n");
        display();
        getch();
}/*End of main()*/
```

TIME COMPLEXITY

When we calculate the time complexity of the heap sort algorithm, we need to analyse the two phases separately.

Phase 1: Let H be a heap and suppose you want to insert a new element data in H. Then few comparisons are required to locate the appropriate place, and it cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by log m where m is the number of elements in H. Then

$$f(n) = O(n \log n)$$

Note that the number of comparison in the worst case is $O(n \log n)$

In the second phase we analyse the complexity of the algorithm to delete a root element from the heap H with n elements.

Phase 2: Suppose H is a complete tree with $n - 1 = m$ elements, and suppose the left and right sub tree of H are heaps and L is the root of H. Rearranging the node L will take four comparisons to move one step down in the tree H. Since the depth of H does not exceed $\log_2 m$, rearranging will take at most $4 \log_2 m$ comparisons to find the appropriate place of L in the tree H.

$$f(n) = 4n \log_2 n$$

6.10. EXTERNAL SORT

In the previous section, we discussed different internal sorting algorithms and its importance. Now we will discuss about external sorting. These are methods employed to sort elements (or items), which are too large to fit in the main memory of the computer. That is any sort algorithm that uses external memory, such as tape or disk, during the sort is called external sort. Since most common sort algorithms assume high-speed random access to all intermediate memory, they are unsuitable if the values to be sorted do not fit in main memory. Internal sorting refers to the sorting of an array of data which is in RAM. The main concern with external sorting is to minimize external disk access since reading a disk block takes about a million times longer than accessing an item in RAM.

To study external sorting, we need to study the various external memory devices in addition to external sorting algorithms. The involvement of external storage device makes sorting algorithms more complex because of the following reasons:

1. The cost of accessing an item is much higher than any computational cost.
2. Different procedures and methods have to be implemented and executed for different external storage devices.

In this section, we will discuss some data storage devices and sorting algorithms for external storage devices. A cards reader (or punch) can be considered as a primitive external storage. However, this section deals with devices that allow more rapid data transfer and more convenient storage medium than punch cards. External storage devices can be categorized into two types based on the access method. They are:

- Sequential Access Devices (*e.g.*, Magnetic tapes)
- Random Access Devices (*e.g.*, Disks)

6.10.1. MAGNETIC TAPES

The principle behind magnetic tapes is similar to audio tape recorders. Magnetic tape is wound on a spool. Tracks run across the length of the tape. Usually there are 7 to 9 tracks across the tape width and the data is recorded on the tape in a sequence of bits. The number that can be written per inch of track is called the tape density — measured in bits per inch.

Information on tapes is usually grouped into blocks, which may be of fixed or variable size. Blocks are separated by an inter-block gap. Because request to read or write blocks do not arrive at a tape drive at constant rate, there must be a gap between each pair of blocks forming a space to be passed over as the tape accelerates to read/write speed. The medium is not strong enough to withstand the stress that it would sustain with instantaneous starts and stops. Because the tape is not moving at a constant speed, a gap cannot contain user data. Data is usually read/written from tapes in terms of blocks. This is shown in following Fig. 6.21 :

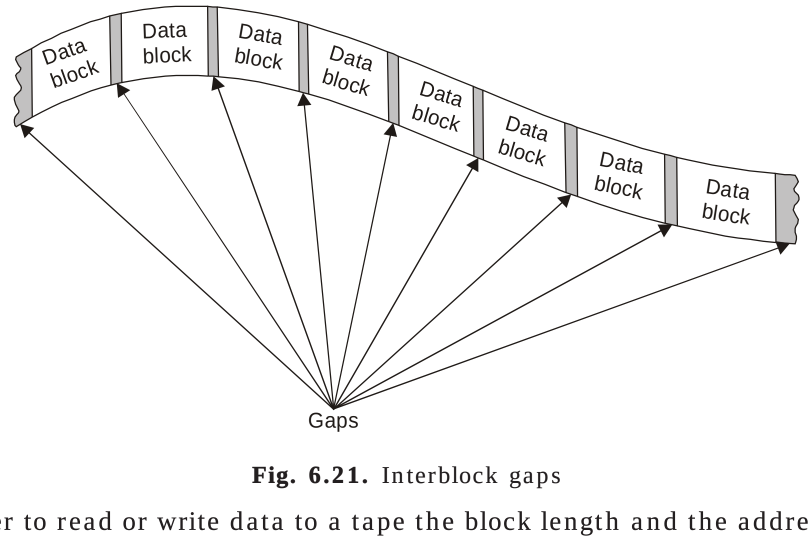


Fig. 6.21. Interblock gaps

In order to read or write data to a tape the block length and the address in memory to/from which the data is to be transferred must be specified. These areas in memory from/to which data is transferred will be called *buffers*. The block size (or length) will respond to its buffer size. And it is a crucial factor in tape access. A large block size is preferred because of the following reasons:

1. Consider a tape with a tape density of 600 bdp and an inter block gap of ¾ inch; generally this gap is enough to write 450 characters. With a small block size, the number of blocks per tape length will increase. This means a larger number of inter block gaps, *i.e.*, bits of data, which cannot be utilized for data storage, and thus tape utilization decreased. Thus the larger the block size, fewer the number of blocks, fewer the number of inter block gaps and better the tape utilization.
2. Larger block size reduces the input/output time. The delay time in tape access is the time needed to cross the inter block gap. This delay time is larger when a tape starts from rest than when the tape is already moving. With a small block size the number of halts in a read are considerable causing the delay time to be incurred each time.

6.10.2. DISKS

Disks are an example of direct access storage devices. In contrast to the way information is recorded on a gramophone record, data are recorded on disk platter in concentric tracks. A disk has two surfaces on which data can be recorded. Disk packs have several such disks or platters rigidly mounted on a common spinder. Data is read/written to the disk by a read/write head. A disk pack would have one such head per surface.

Each disk surface has a number of concentric circles called tracks. In a disk pack, the set of parallel tracks on each surface is called a cylinder. Tracks are further divided into sectors. A sector is the smallest addressable segment of a track.

Data is stored along the tracks in blocks. Therefore to access a disk, the track or cylinder number and the sector number of the starting block must be specified. For disk packs, the surface must also be specified. The read/write head moves horizontally to position itself over the correct track for accessing disk data.

This introduces three time components into disk access :

Seek time: The time taken to position the read/write head over the correct cylinder.

Latency time: The time taken to position the correct sector under head.

Transfer time: The time taken to actually transfer the block between main memory and the disk.

Having seen the structure of data storage on disks and tapes and the methods of accessing them, we now turn to specific cases of external sorting. Sorting data on disks and sorting data on tapes. The general method for external sorting is the merge sort. In this, file segments are sorted using a good internal sort method. These sorted file segments, called runs, are written out onto the device. Then all the generated runs are merged into one run.

6.10.3. EXTERNAL SORTING ALGORITHMS

Perhaps the simplest form of external sorting is to use a fast internal sort with good locality of reference (which means that it tends to reference nearby items, not widely scattered items). Quicksort is one sort algorithm that is generally very fast and has good locality of reference. If the file is too huge, even virtual memory might be unable to fit it. Also, the performance may not be too great due to the large amount of time it takes to access data on disk.

Merge sort is an ideal candidate for external sorting because it satisfies the two criteria for developing an external sorting algorithm. Merge sort can be implemented either top-down or bottom-up. The top-down strategy is typically used for internal sorting, whereas the bottom-up strategy is typically used for external sorting.

The top-down strategy works by:

1. Dividing the data in half
2. Sorting each half
3. Merging the two halves

Merge sort typically break a large data file into a number of shorter, sorted runs. These can be produced by repeatedly reading a section of the data file into RAM, sorting it with ordinary quicksort, and writing the sorted data to disk. After the sorted runs have been generated, a merge algorithm is used to combine sorted files into longer sorted files. The simplest scheme is to use a 2-way merge: merge 2 sorted files into one sorted file, and then merge 2 more, and so on until there is just one large sorted file.

One example of external sorting is the external mergesort algorithm. For the sake of clarity, let us assume that 900 megabyte of data needs to be sorted using only 100 megabytes of RAM.

1. Read 100 MB of the data in main memory and sort by some conventional method (usually quicksort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is sorted in chunks of 100 MB. Now you need to merge them into one single sorted output file.
4. Read the first 10 MB of each sorted chunk (call them input buffers) in main memory (90 MB total) and allocate the remaining 10 MB for output buffer.
5. Perform a 9-way merging and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10 MB of its associated 100 MB sorted chunk or otherwise mark it as exhausted if there is no more data in the sorted chunk, do not use it for merging.

Let us analyse how the merge sort algorithm responds when it is practically applied to run using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements. If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes.
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes.
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes.
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes.

5. Repeat until you have one chunk containing all the data, sorted --- that is, for log *n* passes, where *n* is the number of records.

On tape drives that can run both backwards and forwards, you can run merge passes in both directions, avoiding rewind time. For the same reason it is also very useful for sorting data on disk that is too large to fit entirely into primary memory.

The above described algorithm can be generalized by assuming that the amount of data to be sorted exceeds the available memory by a factor of K. Then, K chunks of data need to be sorted and a K-way merge has to be completed. If X is the amount of main memory available, there will be K input buffers and 1 output buffer of size X/(K+1) each. Depending on various factors (how fast the hard drive is, what is the value of K) better performance can be achieved if the output buffer is made larger (for example twice as large as one input buffer).

Note that you do not want to jump back and forth between 2 or more files in trying to merge them (while writing to a third file). This would likely produce a lot of time-consuming disk seeks. Instead, on a single-user PC, it is better to read a block of each of the 2 (or more) files into RAM and carry out the merge algorithm there, with the output also kept in a buffer in RAM until the buffer is filled (or we are out of data) and only then writing it out to disk. When the merge algorithm exhausts one of the blocks of data, refill it by reading from disk another block of the associated file. This is called buffering. On a larger machine where the disk drive is being shared among many users, it may not make sense to worry about this as the read/write head is going to be seeking all over the place anyway.

```
mergesort(int a[], int left, int right)
       {
         int i, j, k, mid;

         if (right > left) {
            mid = (right + left) / 2;
            mergesort(a, left, mid);
            mergesort(a, mid+1, right);
            /* copy the first run into array b */
            for (i = left, j = 0; i <= mid; i++, j++)
               b[ j] = a[i];
            b[j] = MAX_INT;
            /* copy the second run into array c */
            for (i = mid+1, j = 0; i <=right; i++, j++)
               c[ j] = a[i];
            c[ j] = MAX_INT;
            /* merge the two runs */
            i = 0;
            j = 0;
            for (k = left; k <= right; k++)
```

```
                    a[k] = (b[i] < c[ j]) ? b[i++] : c[ j++];
                }
            }
```

The bottom-up strategy for merge sort works by:

1. Scanning through data performing 1-by-1 merges to get sorted lists of size 2.
2. Scanning through the size 2 sub-lists and perform 2-by-2 merges to get sorted lists of size 4.
3. Continuing the process of scanning through size *n* sub-lists and performing n-by-*n* merges to get sorted lists of size 2*n* until the file is sorted (*i.e.*, 2*n* >= N, where N is the size of the file).

6.10.4. Mergesort Performance

Mergesort has an average and worst-case performance of O(n log n). In the worst case, merge sort does about 30% fewer comparisons than quicksort does in the average case; thus merge sort very often needs to make fewer comparisons than quicksort. In terms of moves, merge sort's worst case complexity is O(*n* log *n*); the same complexity as quicksort's best case, and merge sort's best case takes about half as much time as the worst case.

However, merge sort performs 2*n* – 1 method calls in the worst case, compared to quicksort's *n*, thus has roughly twice as much recursive overhead as quicksort. Mergesort's most common implementation does not sort in place, that is memory size of the input must be allocated for the sorted output to be stored in. Sorting in-place is possible but requires an extremely complicated implementation.

Although merge sort can sort linked list, it is also much more efficient than quicksort if the data to be sorted can only be efficiently accessed sequentially. Unlike some optimized implementations of quicksort, merge sort is a stable sort, as long as the merge operation is implemented properly.

More precisely, merge sort does between [*n* log *n* – *n* + 1] and [*n* log *n* – 0.914·*n*] comparisons in the worst case.

SELF REVIEW QUESTIONS

1. Explain the method of external sorting with disks. [*MG - MAY 2004* (*BTech*)]
2. Write and explain insertion sort algorithm. What is the complexity of the algorithm? [*MG - MAY 2004* (*BTech*), *MG - NOV 2004* (*BTech*)]
3. Explain quick sort algorithm. What is the complexity of your algorithm? [*CUSAT - JUL 2002* (*MCA*), *MG - MAY 2004* (*BTech*) *KERALA - MAY 2001* (*BTech*)]
4. Explain the method of external sorting with tapes? [*KERALA - DEC 2003* (*BTech*), *MG - NOV 2004* (*BTech*) *KERALA - MAY 2001* (*BTech*)]
5. Explain merging of sequential files. [*MG - MAY 2003* (*BTech*)]

6. What is external sorting methods?
[*KERALA - DEC 2003* (*BTech*), *KERALA - JUN 2004* (*BTech*)
MG - NOV 2003 (*BTech*), *MG - MAY 2000* (*BTech*)
ANNA - DEC 2004 (*BE*), *ANNA - MAY 2004* (*MCA*)]
7. Write the Quick sort algorithm. [*MG - NOV 2003* (*BTech*)]
8. With suitable example, explain radix sort.
[*CUSAT - APR 1998* (*BTech*), *MG - NOV 2002* (*BTech*)]
9. Explain bubble sort with example. Construct Heap sort for the initial key set 42 23 74 11 65 58 94 36 99 87. [*Calicut - APR 1995* (*BTech*)]
10. Explain Heap sort with an example. [*Calicut - APR 1997* (*BTech*)]
11. Explain quicksort algorithm? Write an iterative program fragment for quicksort?
[*KERALA - JUN 2004* (*BTech*), *CUSAT - NOV 2002* (*BTech*)]
12. Write an algorithm for merging two sorted list of numbers represented as linked lists. No new memory space may be used. The merged list should be also sorted.
[*CUSAT - DEC 2003* (*MCA*)]
13. For the following input list explain how Merge sort works. What is the time complexity involved in the algorithm? [*CUSAT - JUL 2002* (*MCA*)]
14. Differentiate between heap sort and Radix sort. [*ANNA - MAY 2004* (*MCA*)]
15. Distinguish between internal sorting and external sorting.
[*KERALA - DEC 2004* (*BTech*), *ANNA - MAY 2004* (*MCA*)
KERALA - DEC 2002 (*BTech*), *KERALA - MAY 2003* (*BTech*)]
16. How many key comparisons and interchanges are required to sort a file of size *n* using bubble sort? [*ANNA - DEC 2004* (*BE*)]
17. What is an external storage device? Explain in detail about any two devices.
[*ANNA - DEC 2004* (*BE*)]
18. Explain Internal sorting Methods.
[*KERALA - DEC 2003* (*BTech*), *KERALA - JUN 2004* (*BTech*)]
19. Write an algorithm for merge sort method. [*KERALA - JUN 2004* (*BTech*)]
20. Write a procedure for bubble sort method with example.
[*KERALA - DEC 2002* (*BTech*), *KERALA - DEC 2003* (*BTech*)]
21. Write an algorithm to sort elements by partition exchange method.
[*KERALA - MAY 2003* (*BTech*)]
22. Compare merge sort and insertion sort methods. [*KERALA - MAY 2001* (*BTech*)]
23. When is the bubble sort better than quick sort? [*KERALA - MAY 2002* (*BTech*)]
24. Write a function to implement the queue operation using two stacks.
[*KERALA - NOV 2001* (*BTech*)]
25. Write an algorithm for selection sort method. [*KERALA - NOV 2001* (*BTech*)]