SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V -Data Structures – SCSA1203

UNIT 5 SEARCHING AND SORTING TECHNIQUES

Basic concepts - List Searches using Linear Search - Binary Search - Fibonacci Search - Sorting Techniques - Insertion sort - Heap sort - Bubble sort - Quick sort - Merge sort - Analysis of sorting techniques.

SEARCHING

Searching and sorting are fundamental operations in computer science. Searching refers to the operation of finding the location of a given item in a collection of items. Sorting refers to the operations of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data.

Searching : *Searching is an operation which finds the location of a given element in a list. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not.*

The two standard searching techniques are:

- *Linear search*
- *Binary search.*

LINEAR SEARCH

Linear search is the simplest method of searching. In this method, the element to be found is sequentially searched in the list (Hence also called sequential search). This method can be applied to a sorted or an unsorted list. Hence, it is used when the records are not stored in order.

Principle : *The algorithm starts its search with the first available record and proceeds to the next available record repeatedly until the required data item to be searched for is found or the end of the list is reached.*

Algorithm :

```
Procedure LINEARSEARCH( A, N, K, P )

// A is the array containing the list of data items
// N is the number of data items in the list
// K is the data item to be searched
// P is the position where the data item is found

P ∈ -1
Repeat For I = 0 to N -1 Step 1
        If A( I ) = K
        Then
                P ∈ I
                Exit Loop
        End If
End Repeat
End LINEARSEARCH
```

In the above algorithm, A is the list of data items containing N data items. K is the data item, which is to be searched in A. P is a variable used to indicate, at what position the data item is found. If the data item to be searched is found then the position where it is found is stored in P. If the data item to be searched is not found then -1 is stored in P, which will indicate the user, that the data item is not found.

Initially P is assumed -1. The data item K is compared with each and every element in the list A. During this comparison, if K matches with a data item in A, then the position where the data item was found is stored in P and the control comes out of the loop and the procedure comes to an end. If K does not match with any of the data items in A, then P value is not changed at all. Hence at the end of the loop, if P is -1, then the number is not found.

Example:

K ∈ Number to be searched : **40**

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[0]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[1]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[2]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[3]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[4]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[5]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K ≠ A[6]

45	**56**	**15**	**76**	**43**	**92**	**35**	**40**	**28**	**65**

K = A[7] ∈ P = 7 : Number found at position 7

Advantages:

1. Simple and straight forward method.
2. Can be applied on both sorted and unsorted list.

Disadvantages:

1. Inefficient when the number of data items in the list increases.

BINARY SEARCH

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order. Binary search cannot be applied on an unsorted list.

Principle: *The data item to be searched is compared with the approximate middle entry of the list. If it matches with the middle entry, then the position is returned. If the data item to be searched is lesser than the middle entry, then it is compared with the middle entry of the first half of the list and procedure is repeated on the first half until the required item is found. If the data item is greater than the middle entry, then it is compared with the middle entry of the second half of the list and procedure is repeated on the second half until the required item is found. This process continues until the desired number is found or the search interval becomes empty.*

Algorithm:

```
Procedure BINARYSEARCH( A, N, K, P )

// A is the array containing the list of data items
// N is the number of data items in the list
// K is the data item to be searched
// P is the position where the data item is found

Lower ← 0, Upper ← N – 1, P ← -1

While Lower ≤ Upper

        Mid ← ( Lower + Upper ) / 2
        If K = A[Mid]
        Then
                P ← Mid
                Exit Loop
        Else
                If K < A[Mid]
                Then
                        Upper ← Upper -1
                Else
                        Lower ← Lower + 1
                End If
        End If

End While
End BINARYSEARCH
```

In Binary Search algorithm given above, A is the list of data items containing N data items. K is the data item, which is to be searched in A. P is a variable used to indicate, at what position the data item is found. If the data item to be searched is found then the position where it is found is stored in P. If the data item to be searched is not found then -1 is stored in P, which will indicate the user, that the data item is not found.

Initially P is assumed -1, lower is assumed 0 to point the first element in the list and upper is assumed as upper -1 to point the last element in the list. The mid position of the list is calculated and K is compared with A[mid]. If K is found equal to A[mid] then the value mid is assigned to P, the control comes out of the loop and the procedure comes to an end. If K is found lesser than A[mid], then upper is assigned mid – 1, to search only in the first half of the list. If K is found greater than A[mid], then lower is assigned mid + 1, to search only in the second half of the list. This process is continued until the element searched is found or the search interval becomes empty.

Example:

K ∈ Number to be searched : **40**
U ∈ Upper
L ∈ Lower
M∈ Mid

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = 4 U = 9
K < A[4] ∈ U = 4 – 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = 1 U = 3
K > A[1] ∈ L = 1 + 1 = 2

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M = 2 U = 3
K > A [2] ∈ L = 2 + 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M, U = 3
K = A[3] ∈ P = 3 : Number found at position 3

Advantages:

1. Searches several times faster than the linear search.
2. In each iteration, it reduces the number of elements to be searched from n to n/2.

Disadvantages:

1. Binary search can be applied only on a sorted list.

Fibonacci Search

Given a sorted array arr[] of size n and an element x to be searched in it. Return index of x if it is present in array else return -1.

Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

Similarities with Binary Search:

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has Log n time complexity.

Differences with Binary Search:

- Fibonacci Search divides given array into unequal parts
- Binary Search uses a division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when the input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Algorithm:

Let the searched element be x. The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of the given array. Let the found Fibonacci number be fib (m'th Fibonacci number). We use (m-2)'th Fibonacci number as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for sub array after i, else we recur for sub array before i. Below is the complete algorithm. Let arr[0..n-1] be the input array and the element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the array has elements to be inspected:
    1. Compare x with the last element of the range covered by fibMm2
    2. If x matches, return index
    3. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
    4. Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

SORTING

Practically, all data processing activities require data to be in some order. Ordering or sorting data in an increasing or decreasing fashion according to some linear relationship among data items is of fundamental importance.

Sorting : *Sorting is an operation of arranging data, in some given order, such as increasing or decreasing with numerical data, or alphabetically with character data.*

Let A be a list of n elements A_1, A_2,…A_n in memory. Sorting A refers to the operation of rearranging the contents of A so that they are increasing in order (numerically or lexicographically), that is,

$$A_1 \leq A_2 \leq A_3 \leq \ldots \leq A_n$$

Sorting methods can be characterized into two broad categories:

- *Internal Sorting*
- *External Sorting*

Internal Sorting : *Internal sorting methods are the methods that can be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory.*

The key principle of internal sorting is that all the data items to be sorted are retained in the main memory and random access in this memory space can be effectively used to sort the data items.

The various internal sorting methods are:

1. *Bubble Sort*
2. *Selection Sort*
3. *Insertion Sort*
4. *Quick Sort*
5. *Merge Sort*
6. *Heap Sort*

External Sorting : *External sorting methods are the methods to be used when the list to be sorted is large and cannot be accommodated entirely in the main memory. In this case some of the data is present in the main memory and some is kept in auxiliary memory such as hard disk, floppy disk, tape, etc.*

The key principle of external sorting is to move data from secondary storage to main memory in large blocks for ordering the data.

√ **Criteria for the selection of a sorting method.**

The important criteria for the selection of a sorting method for the given set of data items are as follows:

1. Programming time of the sorting algorithm
2. Execution time of the program
3. Memory space needed for the programming environment

√ **Objectives involved in design of sorting algorithms.**

The main objectives involved in the design of sorting algorithm are:

1. Minimum number of exchanges
2. Large volume of data block movement

This implies that the designed and desired sorting algorithm must employ minimum number of exchanges and the data should be moved in large blocks, which in turn increase the efficiency of the sorting algorithm.

INTERNAL SORTING

Internal Sorting: *These are the methods which are applied on the list of data items, which are small enough to be accommodated in the main memory.*

There are different types of internal sorting methods. The methods discussed here sort the data in ascending order. With a minor change we can sort the data in descending order.

BUBBLE SORT

This is the most commonly used sorting method. The bubble sort derives its name from the fact that the smallest data item bubbles up to the top of the sorted array.

Principle : *The bubble sort method compares the two adjacent elements starting from the start of the list and swaps the two if they are out of order. This is continued up to the last element in the list and after each pass, a check is made to determine whether any interchanges were made during the pass. If no interchanges occurred, then the list must be sorted and no further passes are required.*

Algorithm:

```
Procedure BUBBLESORT( A, N )

// A is the array containing the list of data items
// N is the number of data items in the list

Last ∈ N – 1

While Last > 0
        Exch ∈ 0

        Repeat For I = 0 to Last Step 1
                If A[I] > A[I+1]
                Then
                        A[I] ↔ A[I+1]
                        Exch ∈ 1
                End If
        End Repeat

        If Exch = 1
        Then
                Exit Loop
        Else
                Last ∈ Last – 1
        End If
End While

End BUBBLESORT
```

In Bubble sort algorithm, initially *Last* is made to point the last element of the list and *Exch* flag is assumed 0. Starting from the first element, the adjacent elements in the list are compared. If they are found out of order then they are swapped immediately and *Exch* flag is set to 1. This comparison is continued until the last two elements are compared. After this pass, the *Exch* flag is checked to determine whether any exchange has taken place. If no exchange has taken place then the control comes out of the loop and the procedure comes to an end as the list is sorted. If any exchange has taken place during the pass, the *last* pointer is decremented by 1 and the next pass is continued. This process continues until list is sorted.

Example:
N = 10 € Number of elements in the list
L € Points to last element (Last)

Pass 1

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**
Out of order € Swap									L=9
23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**
Out of order € Swap									L=9
23	**42**	**11**	**74**	**65**	**58**	**94**	**36**	**99**	**87**
Out of order € Swap									L=9
23	**42**	**11**	**65**	**74**	**58**	**94**	**36**	**99**	**87**
Out of order € Swap									L=9
23	**42**	**11**	**65**	**58**	**74**	**94**	**36**	**99**	**87**
Out of order € Swap									L=9
23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**99**	**87**
Out of order € Swap									L=9

Pass 2

23	**42**	**11**	**65**	**58**	**74**	**36**	**94**	**87**	**99**
Out of order € Swap								L=8	
23	**11**	**42**	**65**	**58**	**74**	**36**	**94**	**87**	**99**
Out of order € Swap								L=8	

23	11	42	58	65	74	36	94	87	99

Out of order €Swap L=8

23	11	42	58	65	36	74	94	87	99

Out of order €Swap L=8

Pass 3

23	11	42	58	65	36	74	87	94	99

Out of order €Swap L=7

23	11	42	58	65	36	74	87	94	99

Out of order €Swap L=7

Pass 4

23	11	42	58	36	65	74	87	94	99

Out of order €Swap L=6

11	23	42	58	36	65	74	87	94	99

Out of order €Swap L=6

Pass 5

11	23	42	36	58	65	74	87	94	99

Out of order €Swap L=5

Pass 6

Adjacent numbers are compared up to L=4. But no swapping takes place. As there was no swapping taken place in this pass, the procedure comes to an end and we get a sorted list:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Simple and works well for list with less number of elements.

Disadvantages:

1. Inefficient when the list has large number of elements.
2. Requires more number of exchanges for every pass.

INSERTION SORT

The main idea behind the insertion sort is to insert the i^{th} element in its correct place in the i^{th} pass. Suppose an array A with n elements A[1], A[2],…A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2],..A[K-1].

Principle: *In Insertion Sort algorithm, each element A[K] in the list is compared with all the elements before it (A[1] to A[K-1]). If any element A[I] is found to be greater than A[K] then A[K] is inserted in the place of A[I}. This process is repeated till all the elements are sorted.*

Algorithm:

```
Procedure INSERTIONSORT(A, N)

// A is the array containing the list of data items
// N is the number of data items in the list

Last ∈ N – 1

Repeat For Pass = 1 to Last Step 1
        Repeat For I = 0 to Pass – 1 Step 1
                If A[Pass] < A[I]
                Then
                        Temp ∈ A[Pass]
                        Repeat For J = Pass -1 to I Step -1
                                A[J +1] ∈ A[J]
                        End Repeat
                        A[I] ∈Temp
                End If
        End Repeat
End Repeat

End INSERTIONSORT
```

In Insertion Sort algorithm, *Last* is made to point to the last element in the list and *Pass* is made to point to the second element in the list. In every pass the *Pass* is

incremented to point to the next element and is continued till it reaches the last element. During each pass A[Pass] is compared all elements before it. If A[Pass] is lesser than A[I] in the list, then A[Pass] is inserted in position I. Finally, a sorted list is obtained.

For performing the insertion operation, a variable temp is used to safely store A[Pass] in it and then shift right elements starting from A[I] to A[Pass-1].

Example:

N = 10 ∈ Number of elements in the list
L ∈ Last
P ∈ Pass

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=1 *A[P] < A[0] ∈ Insert A[P] at 0* L=9

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=2 L=9

A[P] is greater than all elements before it. Hence No Change

23	**42**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

P=3 *A[P] < A[0] ∈ Insert A[P] at 0* L=9

11	**23**	**42**	**74**	**65**	**58**	**94**	**36**	**99**	**87**

P=4 L=9

A[P] < A[3] ∈ Insert A[P] at 3

11	**23**	**42**	**65**	**74**	**58**	**94**	**36**	**99**	**87**

P=5 L=9

A[P] < A[3] ∈ Insert A[P] at 3

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**

P=6 L=9

A[P] is greater than all elements before it. Hence No Change

11	**23**	**42**	**58**	**65**	**74**	**94**	**36**	**99**	**87**

P=7 L=9

A[P] < A[2] ∈ Insert A[P] at 2

11	23	36	42	58	65	74	94	99	87

P=8 L=9

A[P] is greater than all elements before it. Hence No Change

11	23	36	42	58	65	74	94	99	87

P, L=9

A[P] < A[7] ∈ Insert A[P] at 7

Sorted List:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Sorts the list faster when the list has less number of elements.
2. Efficient in cases where a new element has to be inserted into a sorted list.

Disadvantages:

1. Very slow for large values of n.
2. Poor performance if the list is in almost reverse order.

QUICK SORT

Quick sort is a very popular sorting method. The name comes from the fact that, in general, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small lists than one larger one. The basic strategy of quick sort is to divide and conquer. Quick sort is also known as ***partition exchange sort.***

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and moves an item a great distance in one move.

Principle: *A pivotal item near the middle of the list is chosen, and then items on either side are moved so that the data items on one side of the pivot element are smaller than the pivot element, whereas those on the other side are larger. The middle or the pivot element is now in its correct position. This procedure is then applied recursively to the 2 parts of the list, on either side of the pivot element, until the whole list is sorted.*

Algorithm:

Procedure QUICKSORT(A, Lower, Upper)

```
// A is the array containing the list of data items
// Lower is the lower bound of the array
// Upper is the upper bound of the array

If Lower ≥ Upper
Then
        Return
End If

I = Lower + 1
J = Upper + 1

While I < J
        While A[I] < A[Lower]
                I ∈ I + 1
        End While

        While A[J] > A[Lower]
                J ∈ J – 1
        End While

        If I < J
        Then
                A[I] ↔ A[J]
        End If
End While

A[J] ↔ A[Lower]
```

QUICKSORT(A, Lower, J – 1)
QUICKSORT(A, J + 1, Upper)

End QUICKSORT

In Quick sort algorithm, *Lower* points to the first element in the list and the *Upper* points to the last element in the list. Now I is made to point to the next location of *Lower* and J is made to point to the *Upper*. A[Lower] is considered as the *pivot* element and at the end of the pass, the correct position of the *pivot* element is fixed. Keep incrementing I and stop when A[I] > A[Lower]. When I stops, start decrementing J and stop when A[J] < A[Lower]. Now check if I < J. If so, swap A[I] and A[J] and continue moving I and J in the same way. When I meets J the control comes out of the loop and A[J] and A[Lower] are swapped. Now the element at position J is at correct position and hence split the list into two partitions: (A{Lower] to A[J-1] and A[J+1] to A[Upper]). Apply the Quick sort algorithm recursively on these individual lists. Finally, a sorted list is obtained.

Example:

N = 10 € Number of elements in the list
U € Upper
L € Lower

i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=0 U, J=9
Initially I=L+1 and J=U, A[L]=42 is the pivot element.

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=2 J=7 U=9
A[2] > A[L] hence I stops at 2. A[7] < A[L] hence J stops at 7
I < J € Swap A[I] and A[J]

42	**23**	**36**	**11**	**65**	**58**	**94**	**74**	**99**	**87**

L=0 J=3 I=4 U=9
A[4] > A[L] hence I stops at 4. A[3] < A[L] hence J stops at 3
I > J € Swap A[J] and A[L]. Thus 42 go to correct position.

The list is partitioned into two lists as shown. The same process is applied to these lists individually as shown.

€ List 1 € € List 2 €

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L, J=0 I=1 U=2
(applying quicksort to list 1)

11	23	36	42	65	58	94	74	99	87

L, J=1 U, I=2

11	23	36	42	65	58	94	74	99	87

L=4 J=5 I=6 U=9

(applying quicksort to list 2)

11	23	36	42	58	65	94	74	99	87

L=6 I=8 U, J=9

11	23	36	42	58	65	94	74	87	99

L=6 J=8 U, I=9

11	23	36	42	58	65	87	74	94	99

L=6 U, I, J=7

Sorted List:

11	23	36	42	58	65	74	87	94	99

Advantages:

1. Faster than any other commonly used sorting algorithms.
2. It has a best average case behavior.

Disadvantages:

1. As it uses recursion, stack space consumption is high.

MERGE SORT

Principle: *The given list is divided into two roughly equal parts called the left and the right subfiles. These subfiles are sorted using the algorithm recursively and then the two subfiles are merged together to obtain the sorted file.*

Given a sequence of n elements A[1], ….A[N], the general idea is to imagine them split into two sets A[1],…A[N/2] and A[(N/2) + 1],…A[N]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of N elements. Thus this sorting method follows Divide and Conquer strategy.

Algorithm:

```
Procedure MERGE(A, low, mid, high)

// A is the array containing the list of data items

I ∈ low, J ∈ mid+1, K ∈ low
While I Y mid and J Y high
        If A[I] < A[J]
        Then
                Temp[K] ∈A[I]
                I ∈I + 1
                K ∈ K+1
        Else
                Temp[K] ∈A[J]
                J ∈J + 1
                K ∈K + 1
        End If
End While
```

```
If I > mid
Then
        While J Y high
                Temp[K] ∈A[J]
                K ∈ K + 1
                J ∈J + 1
        End While
Else
        While I Y mid
                Temp[K] ∈A[I]
                K ∈ K + 1
                I ∈I + 1
        End While
End If

Repeat for K = low to high step 1
        A[K] ∈ Temp[K]
End Repeat
End MERGE
```

```
Procedure MERGESORT(A, low, high)

// A is the array containing the list of data items

If low < high
Then
        mid ∈ (low + high)/2
        MERGESORT(low, high)
        MERGESORT(mid + 1, high)
        MERGE(low, mid, high)
End If
End MERGESORT
```

The first algorithm MERGE can be applied on two sorted lists to merge them. Initially, the index variable I points to low and J points to mid + 1. A[I] is compared with A[J] and if A[I] found to be lesser than A[J] then A[I] is stored in a temporary array and I is incremented otherwise A[J] is stored in the temporary array and J is incremented. This comparison is continued until either I crosses mid or J crosses high. If I crosses the mid first then that implies that all the elements in first list is accommodated in the temporary array and hence the remaining elements in the second list can be put into the temporary array as it is. If J crosses the high first then the remaining elements of first list is put as it is in the temporary array. After this process we get a single sorted list. Since this method merges 2 lists at a time, this is called 2-way merge sort.

In the MERGESORT algorithm, the given unsorted list is first split into N number of lists, each list consisting of only 1 element. Then the MERGE algorithm is applied for first 2 lists to get a single sorted list. Then the same thing is done on the next two lists and so on. This process is continued till a single sorted list is obtained.

Example:

Let L ∈ low, M∈ mid, H ∈ high

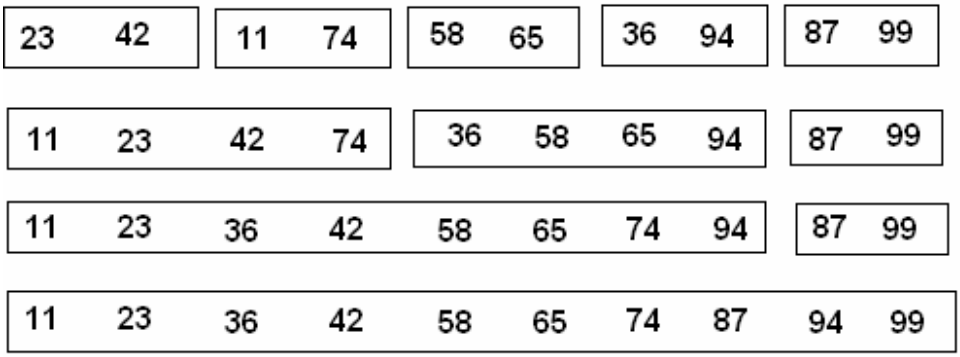
i = 0 i =1 i = 2 i = 3 i = 4 i = 5 i = 6 i = 7 i = 8 i = 9

42	23	74	11	65	58	94	36	99	87

U M H

In each pass the mid value is calculated and based on that the list is split into two. This is done recursively and at last N number of lists each having only one element is produced as shown.

Now merging operation is called on first two lists to produce a single sorted list, then the same thing is done on the next two lists and so on. Finally a single sorted list is obtained.



Advantages:

1. Very useful for sorting bigger lists.
2. Applicable for external sorting also.

Disadvantages:

1. Needs a temporary array every time, for storing the new sorted list.

HEAP SORT

Heap: *A Heap is a compete binary tree with the property that the value at each node is at least as large as (or as small as) the values at its children (if they exist). If the value at the parent node is larger than the values on its children then it is called a* ***Max heap*** *and if the value at the parent node is smaller than the values on its children then it is called the* ***Min heap****.*

- If a given node is in position I then the position of the left child and the right child can be calculated using **Left (L) = 2I** and **Right (R) = 2I + 1.**
- To check whether the right child exists or not, use the condition **R = N.** If true, Right child exists otherwise not.
- The last node of the tree is **N/2.** After this position tree has only leaves.

Principle: The Max heap has the greatest element in the root. Hence the element in the root node is pushed to the last position in the array and the remaining elements are converted into a max heap. The root node of this new max heap will be the second largest element and hence pushed to the last but one position in the array. This process is repeated till all the elements get sorted.

Algorithm:

Procedure WALKDOWN(A, I, N)

// A is the list of unsorted elements
// N is the number of elements in the array
// I is the position of the node where the walkdown procedure is to be applied.

```
While I Y N/2
        L € 2I, R € 2I + 1
If A[L] > A[I]
Then
        M € L
Else
        M € I
End If
If A[R] > A[M] and R Y N
Then
        M € R
End If
If M [ I
Then
        A[I] ↔ A[M]
        I € M
Else
        Return
End If
End While

End WALKDOWN
```

```
Procedure HEAPSORT(A, N)

// A is the list of unsorted elements
// N is the number of elements in the array

Repeat For I = N/2 to 2 step -1
        WALKDOWN(A, I, N)
End Repeat
Repeat For J = N to 2 Step -1
        WALKDOWN(A, 1, J)
        A[1] ↔ A[J]
End Repeat
End HEAPSORT
```

The WALKDOWN procedure is used to convert a subtree into a heap. If this algorithm is applied on a node of the tree, then the subtree starting from that node will be converted into a max heap. In the above given WALKDOWN algorithm, the element at the given node is compared with its left and right child and is swapped with the maximum of that. During this process the element at the given node may walkdown to its correct position in the subtree. The procedure stops if the element at I reaches a leaf or it reaches its correct position.

In the HEAPSORT algorithm there are two phases. In the first phase the walkdown procedure is applied on each node starting from the last node at N/2 to node at position 2. The root node is not disturbed. During this first phase, the subtrees below the root satisfy the max heap property.

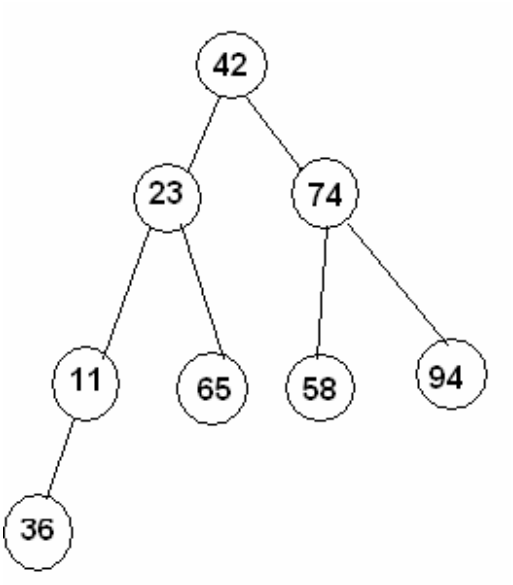
In the second phase of the sorting algorithm, the walkdown procedure is applied on the root node. After this pass the entire tree becomes a heap. The root node element and the last element are swapped and the last element is now not considered for the next pass. Thus the tree size reduces by one in the next pass. This process is repeated till we obtain a sorted list.

Example:

Given a list A with 8 elements:

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**

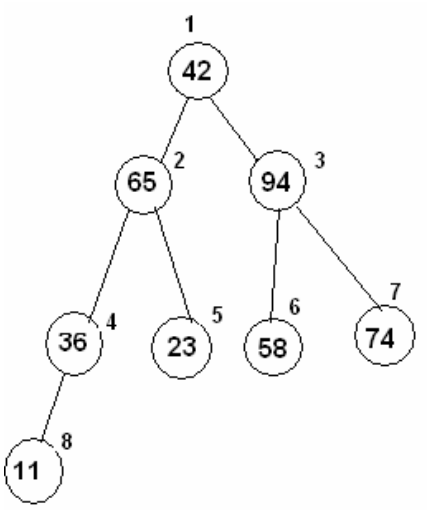
The given list is first converted into a binary tree as shown.



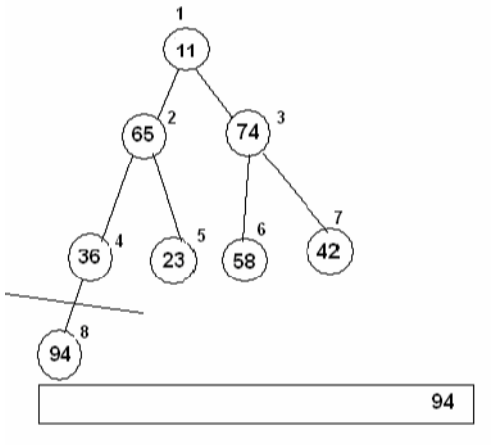
Binary tree

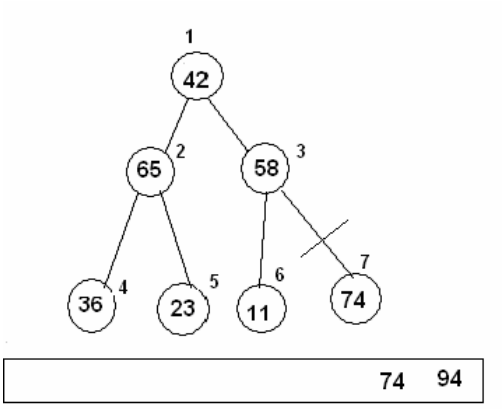
Phase 1:

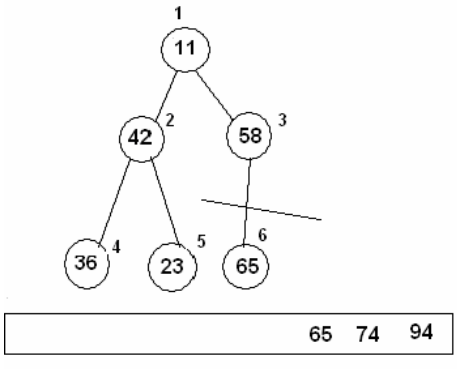
The rearranged tree elements after the first phase is

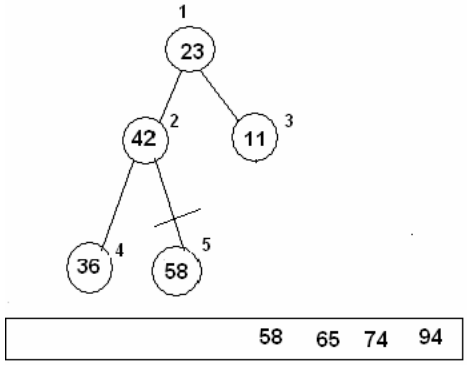


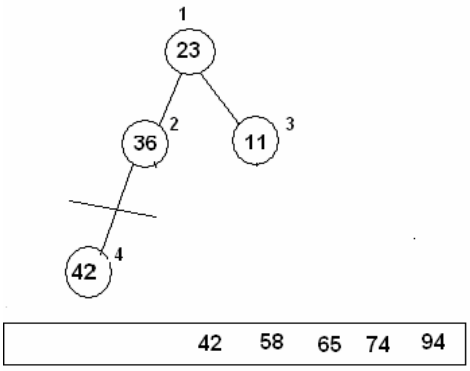
Phase 2:

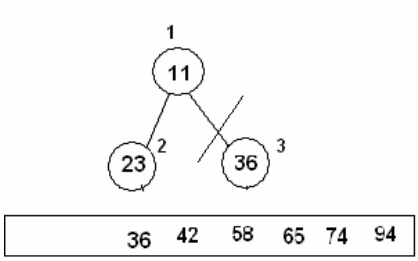


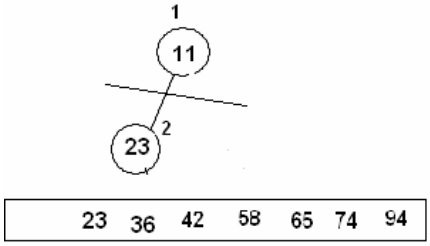


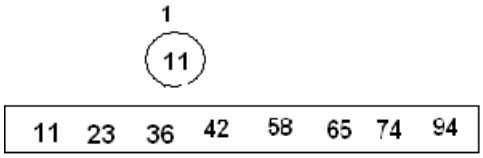












ANALYSIS OF SORTING TECHNIQUES

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)