UNIT-VI
Sorting

Sorting is a technique to rearrange the list of elements either in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified in two types:

a) Internal Sorting

If the data to be sorted remains in main memory and also the sorting is carried out in main memory it is called internal sorting. Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.

The following are some internal sorting techniques:
1. Insertion sort
2. Selection sort
3. Merge Sort
4. Radix Sort
5. Quick Sort
6. Heap Sort
7. Bubble Sort

b) External Sorting

If the data resides in secondary memory and is brought into main memory in blocks for sorting and then result is returned back to secondary memory is called external sorting.

External sorting is required when the data being sorted do not fit into the main memory (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

The following are some external sorting techniques:
1. Two-Way External Merge Sort
2. K-way External Merge Sort

Insertion Sort

Insertion Sort iterates through a list of data items. Each data item is inserted at the correct position within a sorted list composed of those data items already processed.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Insertion sort is a faster and more improved sorting algorithm than selection sort. In selection sort the algorithm iterates through all of the data through every pass whether it is already sorted or not. However, insertion sort works differently, instead of iterating through all of the data after every pass the algorithm only traverses the data it needs to until the segment that is being sorted is sorted.

Insertion Sort Algorithm

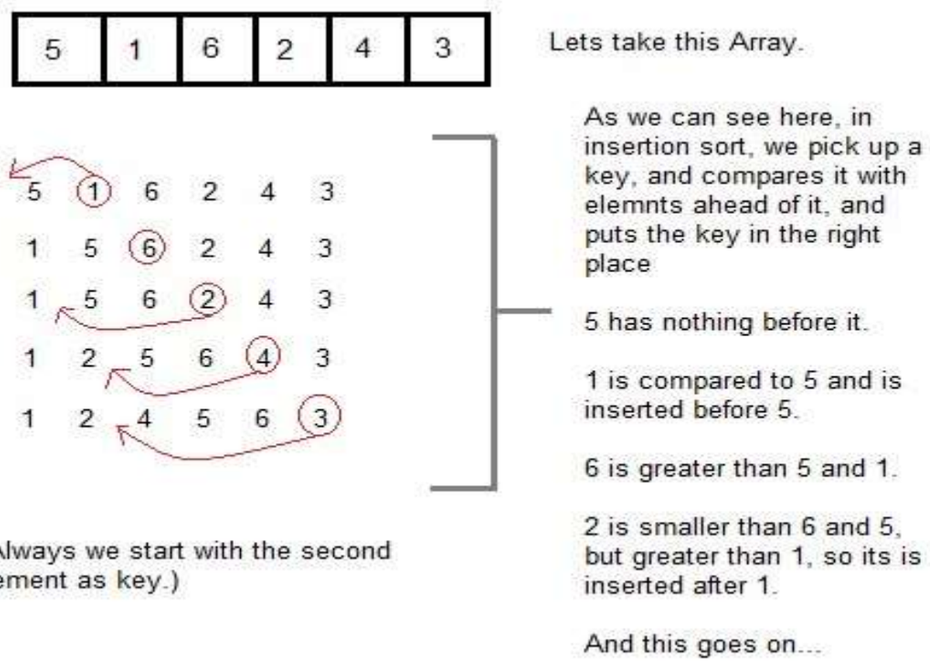
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Insertion Sort Psuedocode

```
template <class T>
void insertionsort (T a[], int n)
{
        int j;
        T temp;
        for (int j = 1; i < n; j++)
        {
                temp=a[j];
                insert(a, temp, j-1);
        }
}
```

```
template <class T>
void insert(T a[], T el, int i)
{
    a[-1]=el;
    if (el<a[i] )
     {
        a[i+1]=a[i];
        a--;
     }
     A[i-1]=temp;
}
```

Working and Examples of Insertion Sort



Time complexity of Insertion Sort

Time complexity, T(n) =1+2+3+4+5+.....+n-1
=n(n-1)/2
=O(n^2)

C++ Program to implement Insertion sort

```
#include <iostream.h>
template <class T>
void insert(T a[], T el, int i)
{
    a[-1]=el;
```

```
    if (el<a[i] )
     {
        a[i+1]=a[i];
        a--;
     }
     A[i-1]=temp;
}
template <class T>
void insertionsort (T a[], int n)
{
            int j;
            T temp;
            for (int j = 1; i < n; j++)
            {
                        temp=a[j];
                        insert(a, temp, j-1);
            }
}

void main()
{
   int a[20], i, j, k, temp, n;
   cout<<"enter no. of elements";
   cin>>n;
   cout<<"enter the elements\n";
   for (i = 0; i < n; i++)
      cin>>a[i];
   insertionsort(a,n);

   cout<<"sorted array\n"<<endl;
   for (i = 0; i < n; i++)
          cout<<a[i]<<endl;
}

Output
enter no. of elements
5
enter the elements
15 11 8 5 2
sorted array
2 5 8 11 15
```

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

In this method, the elements are divided into partitions until each partition has sorted elements. Then, these partitions are merged and the elements are properly positioned to get a fully sorted list.

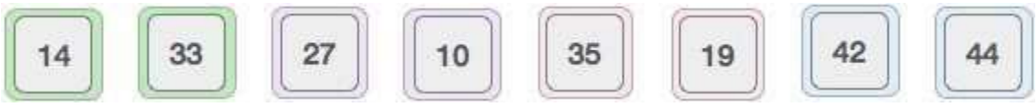
Working of Merge Sort

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into twoarrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Algorithm
Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Merge Sort Pseudocode

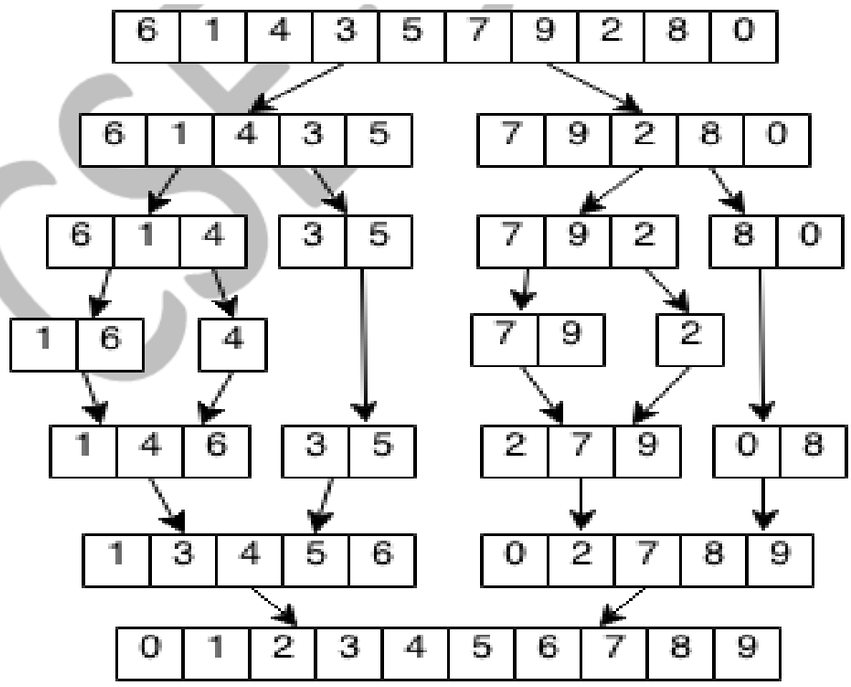
```
template <class T>
void MergeSort (T a[20], int low, int  high)
{
        if (low < high)
        {
                mid := (low + high)/2;
                MergeSort (a, low, mid);
                MergeSort (a, mid + 1, high);
                Merge (a, low, mid, high);
        }
}
```

```
template<class T>
```

```
void Merge (T a[20], int low, int mid, int high)
{
        k :=low; i :=low; j:=mid + 1;
        while ((i ≤ mid) && (j ≤ high))
        {
                if (a[i] ≤ a[j])
                {
                        b[k]:= a[i];
                        k :=k+1;
                        i:=i+1;
                }
                 else
                {
                        b[k] := a[j];
                        k=k+1;
                        j:=j +1;
                }
        }

          while(a[j]<=high)
        {
                        b[k]:= a[j];
                           k=k+1;
                           j := j +1;
        }
        while(a[i]<=mid)
        {
                        b[k] := a[i];
                           k=k+1;
                           i :=i + 1;
           }
        for ( i := low; i<=high; i++)
                        a[i] := b[i];
}
```

Example on Merge Sort



Time Complexity of Merge Sort

T(n)= O(nlogn)

C++ program to implement Merge Sort

```
#include <iostream.h>
template <class T>
void Merge(T a[20], int low, int mid, int high)
{
        int k :=low; i :=low; j:=mid + 1;
        while ((i ≤ mid) && (j ≤ high))
            {
                        if (a[i] ≤ a[j])
                        {
                                    b[k]:= a[i];
                        k :=k+1;
                        i:=i+1;
                   }
                        else
                        {
                                    b[k] := a[j];
                        k=k+1;
                        j:=j+1;
                        }
            }

        while(a[j]<=high)
            {
                                    b[k]:= a[j];
                        k=k+1;
                        j := j +1;
            }
            while(a[i]<=mid)
            {
                                    b[k] := a[i];
                        k=k+1;
                        i :=i + 1;
        }
            for ( i := low; i<=high; i++)
                  a[i] := b[i];
}
template<class T>
void Mergesort (T a[20], int low, int  high)
{
            if (low < high)
            {
                        mid := (low + high)/2;
                        MergeSort (a, low, mid);
                        MergeSort (a, mid + 1, high);
                        Merge (a, low, mid, high);
            }
}
void main()
{
    int a[20], i, j, k, temp, n;
    cout<<"enter no. of elements";
    cin>>n;
    cout<<"enter the elements\n";
    for (i = 0; i < n; i++)
        cin>>a[i]; Mergesort(a,0,n-
    1); cout<<"sorted
    array\n"<<endl; for (i = 0; i <
    n; i++)
            cout<<a[i]<<endl;
}
Output
enter no. of elements
5
enter the elements
6 1 4 3 5 7 9 2 8 0
sorted array
```

0 1 2 3 4 5 6 7 8 9

Quick Sort

In Quick sort, an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(nlogn), where n is the number of items.

Quick Sort Partition Algorithm

Step 1 − Choose the lowest index value has pivot
Step 2 − Take two variables i and j to point left and right of the list respectively
Step 3 – 'i' points to the low index
Step 4 – 'j' points to the high index
Step 5 − while value at a[i] is less than pivot move 'i' right
Step 6 − while value at a[j] is greater than pivot move 'j' left
Step 7 − if both step 5 and step 6 does not match swap a[i] and a[j]
Step 8 − if left ≥ right, swap pivot and a[j], where partition of the list occurs in such a way that all the elements in the left partition are less than pivot and all the elements of in the right partition are greater than pivot.

Quick Sort Algorithm

Step 1 − Make the left-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Psuedocode for Quick sort Partition

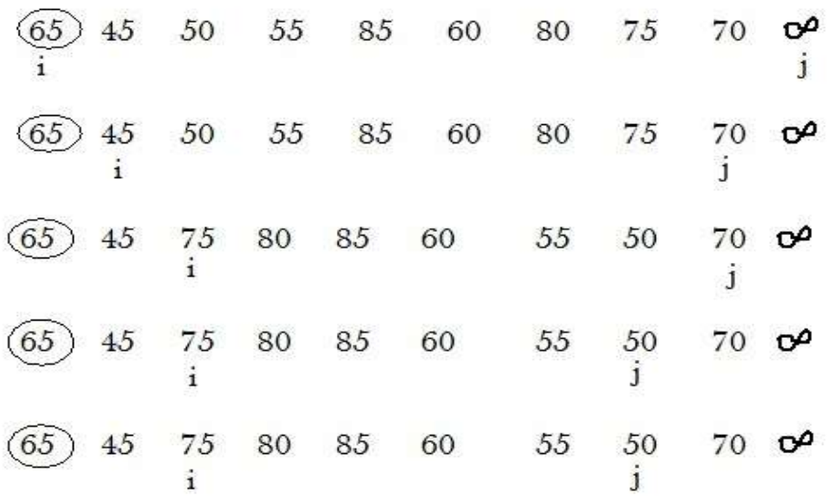
```
template<class T>
void Partition (T a, int l,int r)
{
        T v := a[l];
        int i := l; j := r+1;
        repeat
        {
        repeat
        i := i+1;
        until (a[i] ≤ v && i<=r);
        repeat  j :=j-1;
        until (a[j] ≥ v && j>=l);
        if (i < j) then interchange(a[i],a[j]);
        } until (i ≥  j);
        Interchange(v,a[j]);
        return j;
}
template<class T>
Algorithm Interchange (T a, T b)
{
        T t;
        t := a[i];
        a[i] := a[j];
        a[j]=t;
}
```

Psuedocode for Quick sort

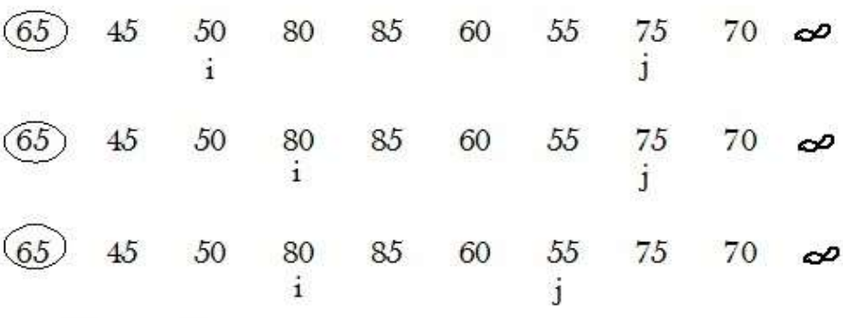
```
template<class T>
Algorithm QuickSort (T a[20], int l, int  r)
{
        if (p <  q)
        {
              j :=Partition(a, l, r);
              QuickSort (l, j - 1);
              QuickSort (j + 1, r);
        }
}
```

Working of Quick Sort

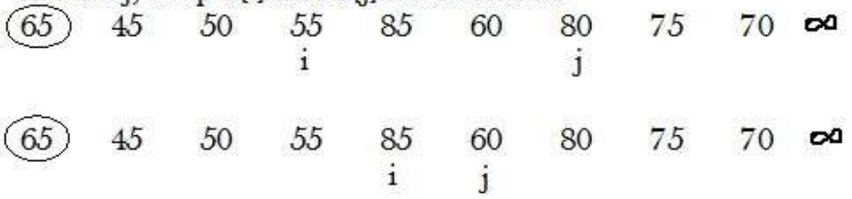
Consider the list



since i<j, swap a[i] and a[j] i.e. 75 and 50



since i<j, swap a[i] and a[j] i.e 80 and 55



Since i<j, swap a[i] and a[j] i.e 85 and 60

65 45 50 55 60 85 80 75 70 ∞
i j

65 45 50 55 60 85 80 75 70 ∞
j i

Since i>j, swap a[j] with pivot element i.e., 60 and 65 and now partition occurs

[60 45 50 55] 65 [85 80 75 70] ∞

List 1: 60 45 50 55 (Elements less than pivot)
List 2: 85 80 75 70 (Elements greater than pivot)
QuickSort is again applied for List1 and List2.

[60 45 50 55] 65 [85 80 75 70] ∞
i j

[60 45 50 55] 65 [85 80 75 70] ∞
i j

[60 45 50 55] 65 [85 80 75 70] ∞
i j

[60 45 50 55] 65 [85 80 75 70] ∞
i j

[60 45 50 55] 65 [85 80 75 70] ∞
j i

Since i>j, swap a[j] with pivot element i.e., 60 and 55 and now partition occurs

[55 45 50] 60 65 [85 80 75 70] ∞

List 1: 55 45 50
List 2: Empty
QuickSort is again applied for List1 and List2.

[55 45 50] 60 65 [85 80 75 70] ∞
i j

[55 45 50] 60 65 [85 80 75 70] ∞
i j

[55 45 50] 60 65 [85 80 75 70] ∞
i j

[55 45 50] 60 65 [85 80 75 70] ∞
ij

[55 45 50] 60 65 [85 80 75 70] ∞
j i

Since i>j, swap a[j] with pivot element i.e., 50 and 55 and now partition occurs

[50 45] 55 60 65 [85 80 75 70] ∞

List 1: 50 45
List 2: Empty
QuickSort is again applied for List1 and List2.

[50 45] 55 60 65 [85 80 75 70] ?
i j

[50 45] 55 60 65 [85 80 75 70] ?
i j

[50 45] 55 60 65 [85 80 75 70] ?
ij

[50 45] 55 60 65 [85 80 75 70] ?
j i

Since i>j, swap a[j] with pivot element i.e., 50 and 45 and now partition occurs

[45] 50 55 60 65 [85 80 75 70] ?

List 1:45
List 2: Empty
QuickSort is again applied for List1 and List2.
Since List 1 is single element, we return out from function

45 50 55 60 65 [85 80 75 70]

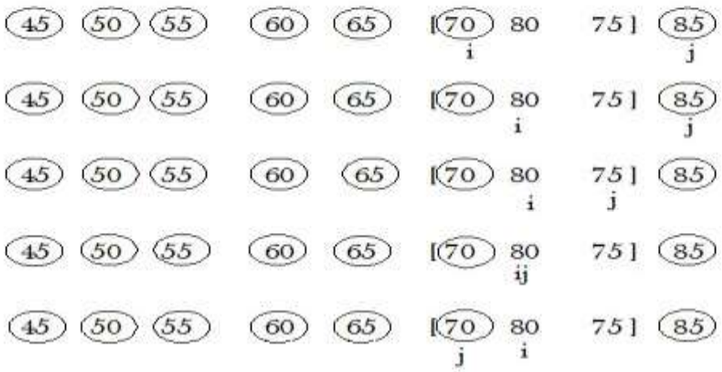
45	50	55	60	65	[85	80	75	70]	?
					i				j
45	50	55	60	65	[85	80	75	70]	?
						i			j
45	50	55	60	65	[85	80	75	70]	?
							i		j
45	50	55	60	65	[85	80	75	70]	?
								i	j
45	50	55	60	65	[85	80	75	70]	?
									ij
45	50	55	60	65	[85	80	75	70]	?
								j	i

Since i>j, swap a[j] with pivot element i.e., 85 and 70 and now partition occurs

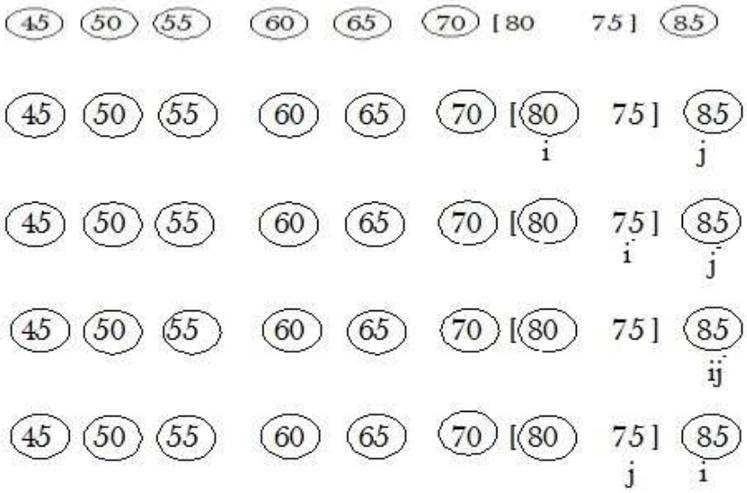
45 50 55 60 65 [70 80 75] 85

List 1: Empty
List 2: 80 75

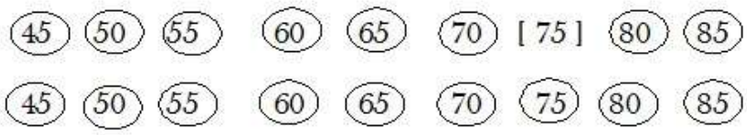
List 1: 70 80 75
List 2: Empty



Since i>j, swap a[j] with pivot element i.e., 70 and 70 and now partition occurs



Since i>j, swap a[j] with pivot element i.e., 75 and 80 and now partition occurs



Hence all the elements are sorted.

Time Complexity of Quick Sort:

Average Case: Let the average case value be $T_A(n)$.
Under the assumptions, the partitioning element v has an equal probability of being the ith smallest element, $1 \le i \le p-m$ in a[m:p-1]. Hence the two subarrays remaining to be sorted are a[m:j] and a[j+1:p-1] with probability $1/(p-m)$, $m \le j < p$.
From this recurrence obtained is

$$T_A(n) = (n+1)+1/n \sum_{1 \le k \le n} [T_A(k-1) + T_A(n-k)]$$

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

Best Case: Let the best case value be $T_B(n)$.
$T_B(n) = O(n \log n)$

Worst Case: Let the worst case value be $T_W(n)$. If all the elements are sorted, then worst case occurs,
$T_w(n) = O(n^2)$

Heap Sort

Heap Sort is one of the best sorting methods being in-place and with O(nlogn) as worst-case complexity.
Heap sort algorithm is divided into two basic parts :

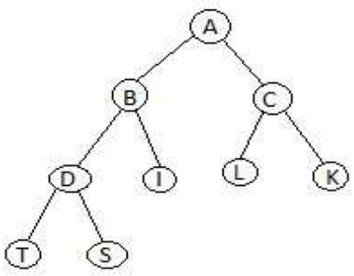
- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Heap sort is an in -place sorting algorithm: only a constant number of array elements are stored outside the input array at any time. `
Worst case running time of Heap sort is O(nlogn).

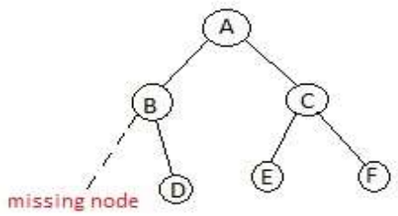
Heap

Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. Shape Property : Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

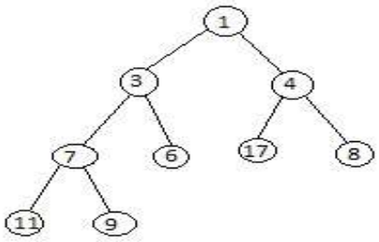


Complete Binary Tree



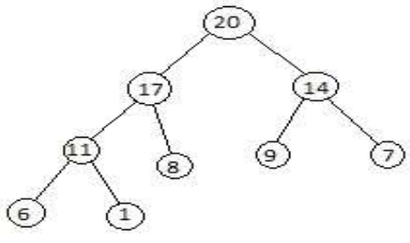
In-Complete Binary Tree

2. Heap Property : All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smalled than their child nodes, heap is called Min-Heap.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Algorithm

1. ~~Build a max heap from the input data.~~

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

Heap Sort Working and Psuedocode

Initially on receiving an unsorted list,

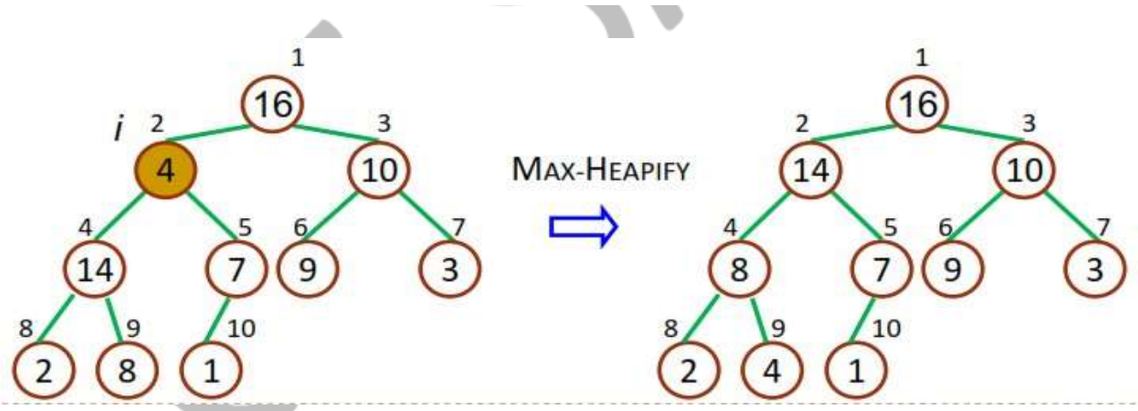
1. First step in heap sort is to build Max-Heap.
2. Repeat Second, Third and Fourth steps, until we have the complete sorted list in our array.
3. Second step- Once heap is built, the first element of the Heap is largest, so we exchange first and last element of a heap.
4. Third step- We delete and put last element(largest) of the heap in our array.
5. Fourth-Then we again make heap using the remaining elements, to again get the largest element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

Basic Procedures

1. The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
2. The BUILD-MAX-HEAP procedure, which runs in $O(n)$ time, produces a max-heap from an unordered input array.
3. The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.

The MAX-HEAPIFY procedure

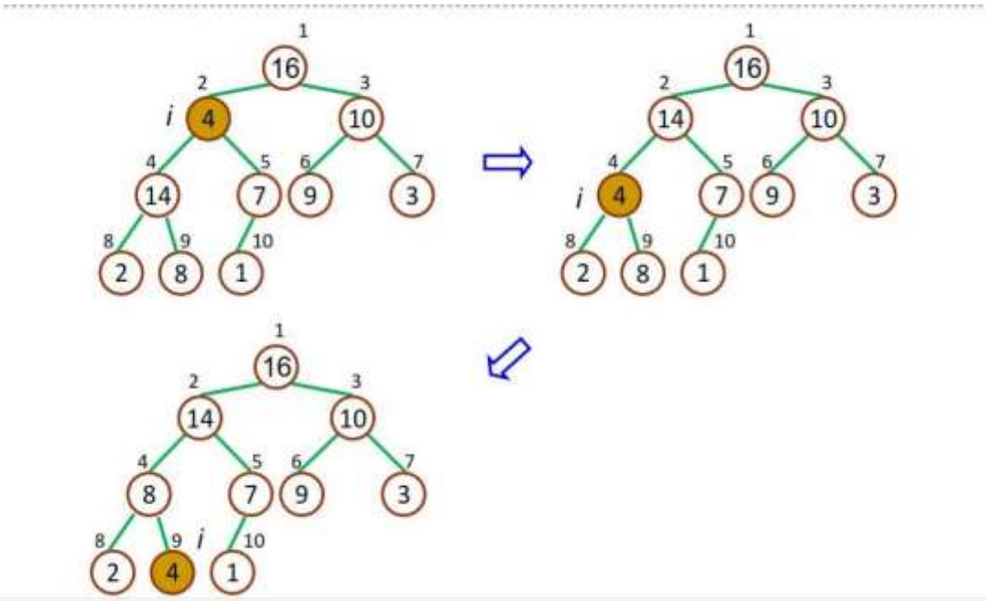
MAX-HEAPIFY is an important subroutine for manipulating max heaps.
Input: an array A and an index *i*
Output: the subtree rooted at index *i* becomes a max heap
Assume: the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but A[i] may be smaller than its children
Method: let the value at A[i] "float down" in the max-heap



The MAX-HEAPIFY pseudocode

```
Procedure MAX-HEAPIFY(A, i)
{
   l =LEFT(i)
   r =RIGHT(i)
   if l <=heap-size[A] and A[l] > A[i]
   then largest =l
   else largest =i
   if r<=heap-size[A] and a[r] > A[largest]
   then largest =r
   if largest ≠ i
   then exchange[ A[i], A[largest] ]
   MAX-HEAPIFY (A, largest)
}
```

An example of MAX-HEAPIFY procedure



Building a Heap

We can use the MAX-HEAPIFY procedure to convert an array A=[1..n] into a max-heap in a bottom-up manner.

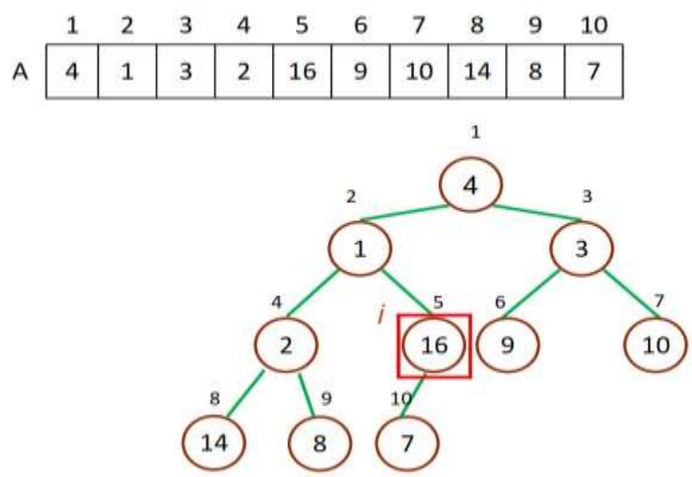
The elements in the subarray A[(⌊$n/2$⌋+1)...n] are all leaves of the tree, and so each is a 1-element heap.

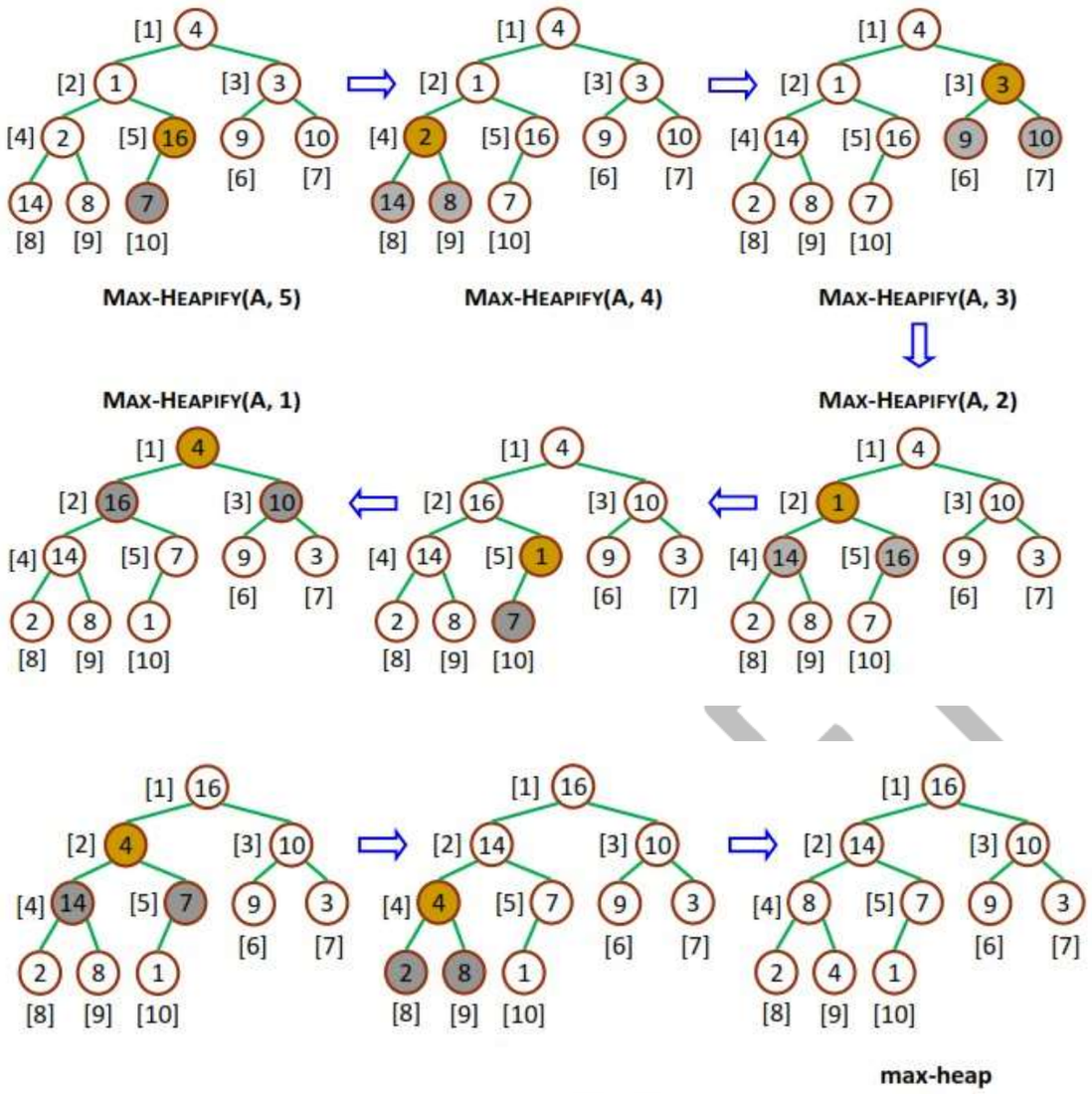
The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

BUILD_MAX_HEAP Psuedocode

```
Procedure BUILD-MAX-HEAP(A)
{
   heap-size[A] =length[A]
   for i  = ⌊length[A]/2⌋ downto 1
   do MAX-HEAPIFY(A,i)
}
```

An example





The heapsort algorithm

Since the maximum element of the array is stored at the root, A[1] we can exchange it with *A*[*n*].

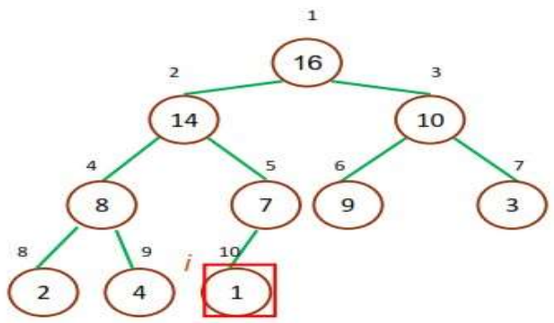
If we now "discard" *A*[*n*], we observe that *A*[1...(*n*- 1)] can easily be made into a max-heap.

The children of the root *A*[1] remain maxheaps, but the new root *A*[1] element may vio late the max-heap property, so we need to readjust the maxheap. That is to call MAX-HEAPIFY(*A*, 1).

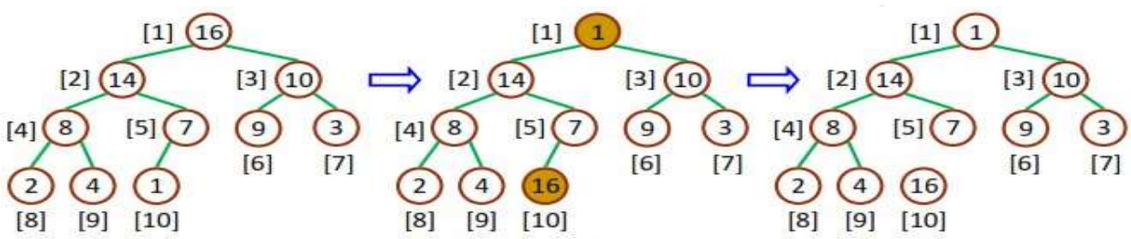
HeapSort Psuedocode

```
Procedure HEAPSORT(A)
{
   BUILD-MAX-HEAP(A)
   for i  =length[A]  downto 2 do
        exchange (A[1], A[i])
        heap-size[A] =heap-size[A] -1
        MAX-HEAPIFY(A, 1)
}
```

An example



	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16



Initial heap

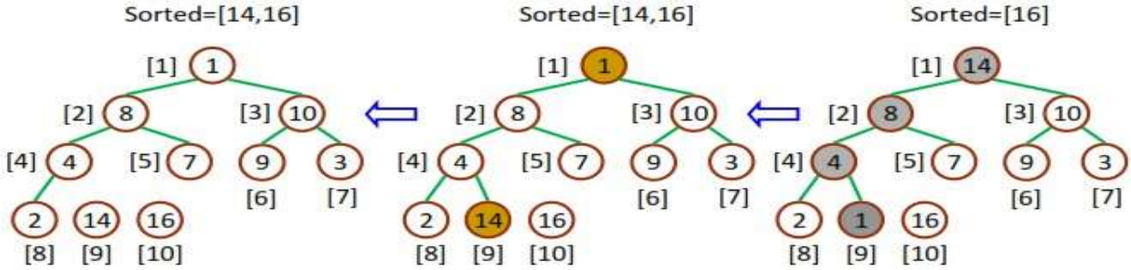
Exchange
Heap size = 10
Sorted=[16]

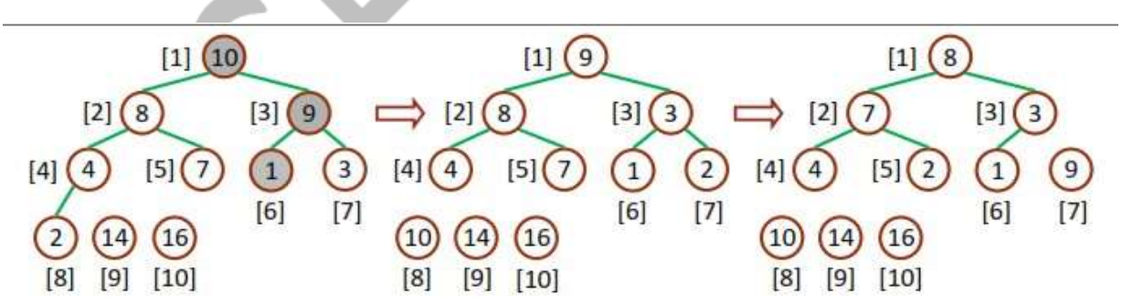
Discard
Heap size = 9
Sorted=[16]

Discard
Heap size = 8
Sorted=[14,16]

Exchange
Heap size = 9
Sorted=[14,16]

Readjust
Heap size = 9
Sorted=[16]





Readjust
Heap size = 8
Sorted=[14,16]

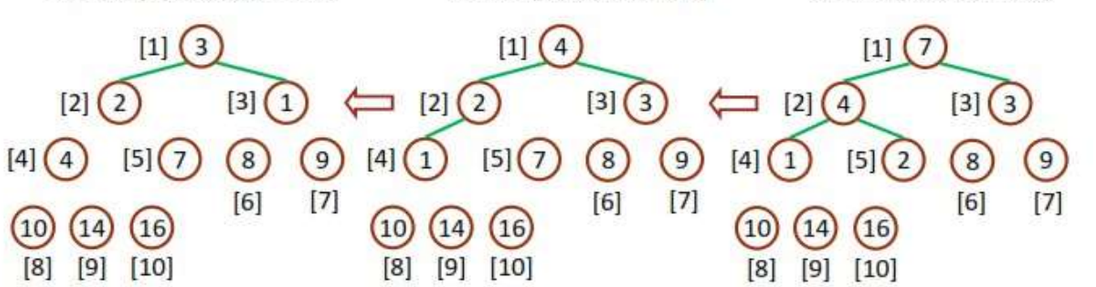
Heap size = 7
Sorted=[10,14,16]

Heap size = 6
Sorted=[9,10,14,16]

Heap size = 3
Sorted=[4,7,8,9,10,14,16]

Heap size = 4
Sorted=[7,8,9,10,14,16]

Heap size = 5
Sorted=[8,9,10,14,16]



Time complexity of Heapsort
The HEAPSORT procedure takes $O(n\log n)$ time
the call to BUILD-MAX-HEAP takes $O(n)$ time
each of the n- 1 calls to MAX-HEAPIFY takes $O(\log n)$ time

Hence, Heapsort procedure takes O(nlogn) time

Question:

Given Set of Elements { 25, 37, 12 , 4, 90}. Sort these elements using Heap Sort technique.

Sol: