Heap Sort

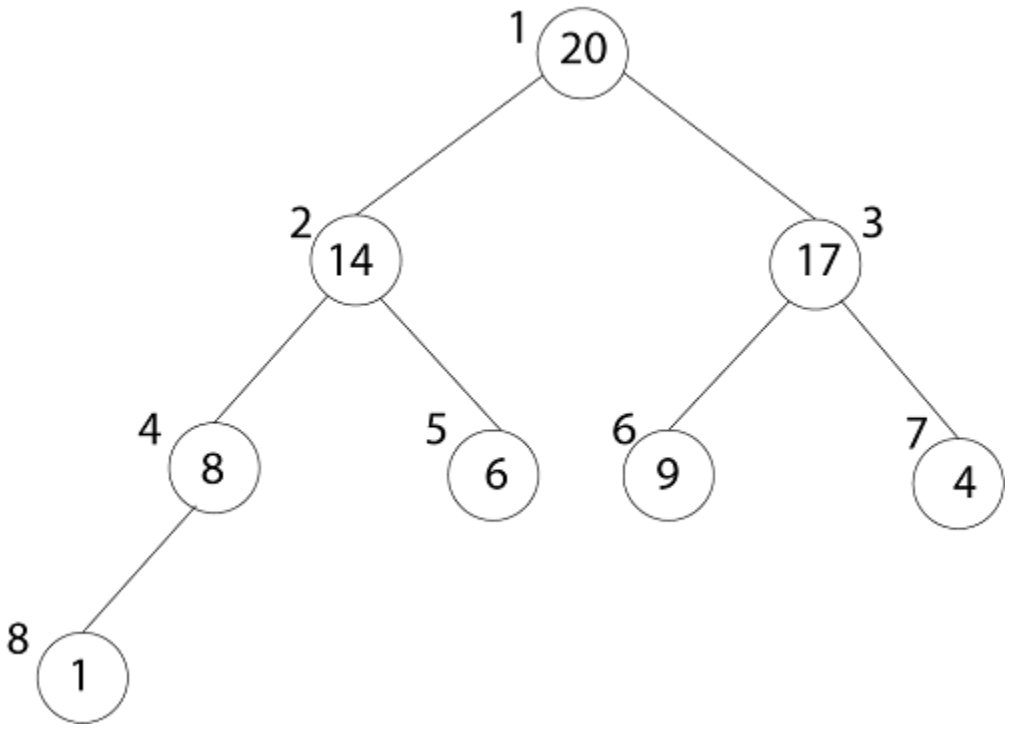
Binary Heap:

Binary Heap is an array object can be viewed as Complete Binary Tree. Each node of the Binary Tree corresponds to an element in an array.

1. Length [A],number of elements in array
2. Heap-Size[A], number of elements in a heap stored within array A.

The root of tree A [1] and gives index 'i' of a node that indices of its parents, left child, and the right child can be computed.

1. PARENT (i)
2. Return floor (i/2)
3. LEFT (i)
4. Return 2i
5. RIGHT (i)
6. Return 2i+1



Representation of an array of the above figure is given below:

1	2	3	4	5	6	7	8	9
20	14	17	8	6	9	4	1	

The index of 20 is 1

To find the index of the left child, we calculate 1*2=2

This takes us (correctly) to the 14.

Now, we go right, so we calculate 2*2+1=5

This takes us (again, correctly) to the 6.

Now, 4's index is 7, we want to go to the parent, so we calculate 7/2 =3 which takes us to the 17.

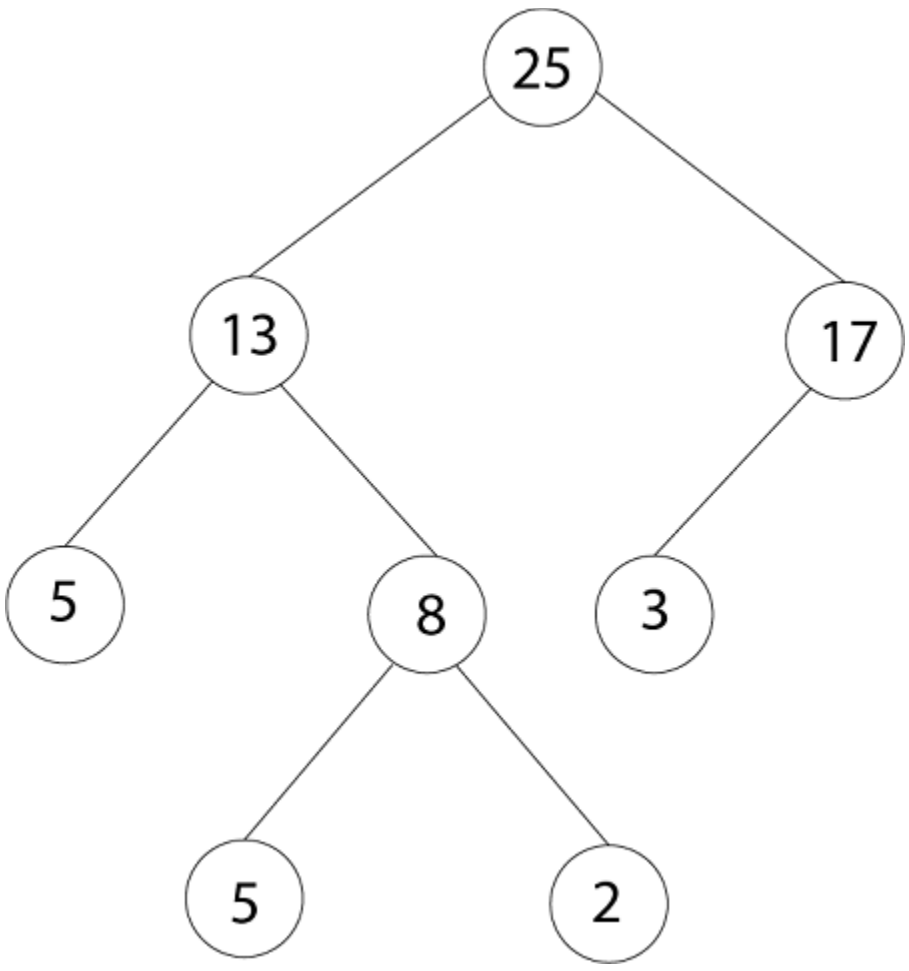
Heap Property:

A binary heap can be classified as Max Heap or Min Heap

1. Max Heap: In a Binary Heap, for every node I other than the root, the value of the node is greater than or equal to the value of its highest child

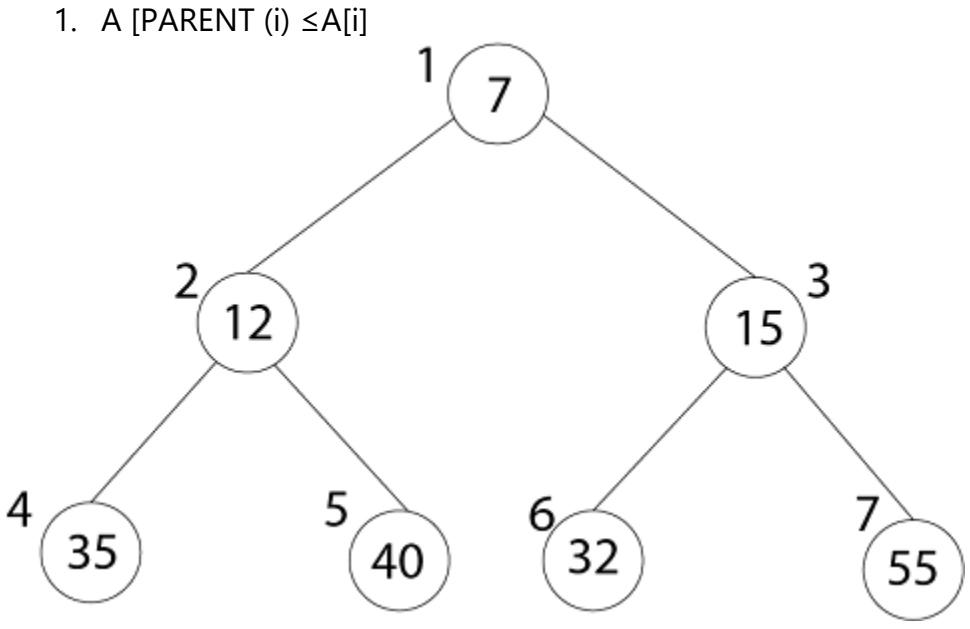
1. A [PARENT (i) ≥A[i]

Thus, the highest element in a heap is stored at the root. Following is an example of MAX-HEAP



2. MIN-HEAP: In MIN-HEAP, the value of the node is lesser than or equal to the value of its lowest child.

1. A [PARENT (i) ≤A[i]



Heapify Method:

1. Maintaining the Heap Property: Heapify is a procedure for manipulating heap Data Structure. It is given an array A and index I into the array. The subtree rooted at the children of A [i] are heap but node A [i] itself may probably violate the heap property i.e. A [i] < A [2i] or A [2i+1]. The procedure 'Heapify' manipulates the tree rooted as A [i] so it becomes a heap.

MAX-HEAPIFY (A, i)

```
1. l ← left [i]
2. r ← right [i]
3. if l≤ heap-size [A] and A[l] > A [i]
4. then largest ← l
5. Else largest ← i
6. If r≤ heap-size [A] and A [r] > A[largest]
7. Then largest ← r
8. If largest ≠ i
9. Then exchange A [i]    A [largest]
10. MAX-HEAPIFY (A, largest)
```

Analysis:

The maximum levels an element could move up are Θ (log n) levels. At each level, we do simple comparison which O (1). The total time for heapify is thus O (log n).

Building a Heap:

```
BUILDHEAP (array A, int n)
```

```
1 for i ← n/2 down to 1
2 do
3 HEAPIFY (A, i, n)
```

HEAP-SORT ALGORITHM:

HEAP-SORT (A)

```
1. BUILD-MAX-HEAP (A)
2. For I ← length[A] down to Z
3. Do exchange A [1] ⟷ A [i]
4. Heap-size [A] ← heap-size [A]-1
5. MAX-HEAPIFY (A,1)
```

Analysis: Build max-heap takes O (n) running time. The Heap Sort algorithm makes a call to 'Build Max-Heap' which we take O (n) time & each of the (n-1) calls to Max-heap to fix up a new heap. We know 'Max-Heapify' takes time O (log n)

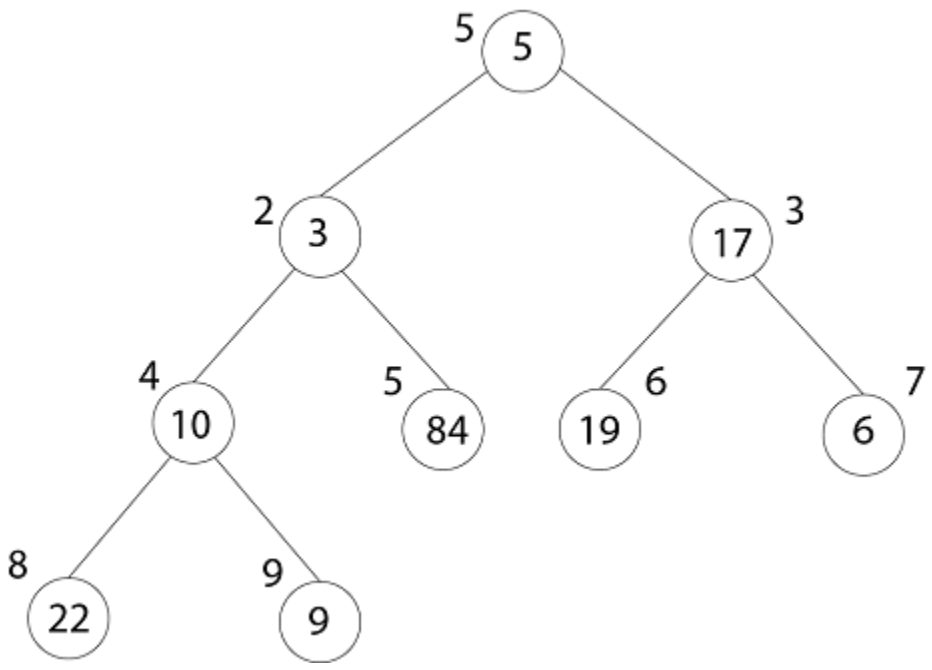
The total running time of Heap-Sort is O (n log n).

Example: Illustrate the Operation of BUILD-MAX-HEAP on the array.

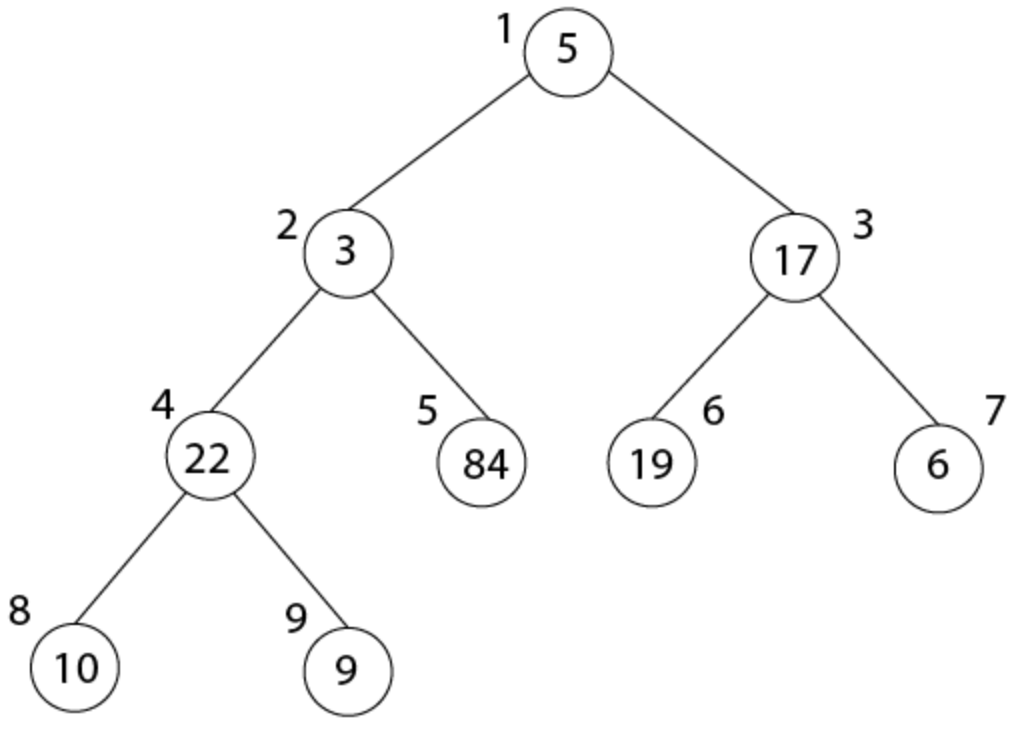
1. A = (5, 3, 17, 10, 84, 19, 6, 22, 9)

Solution: Originally:

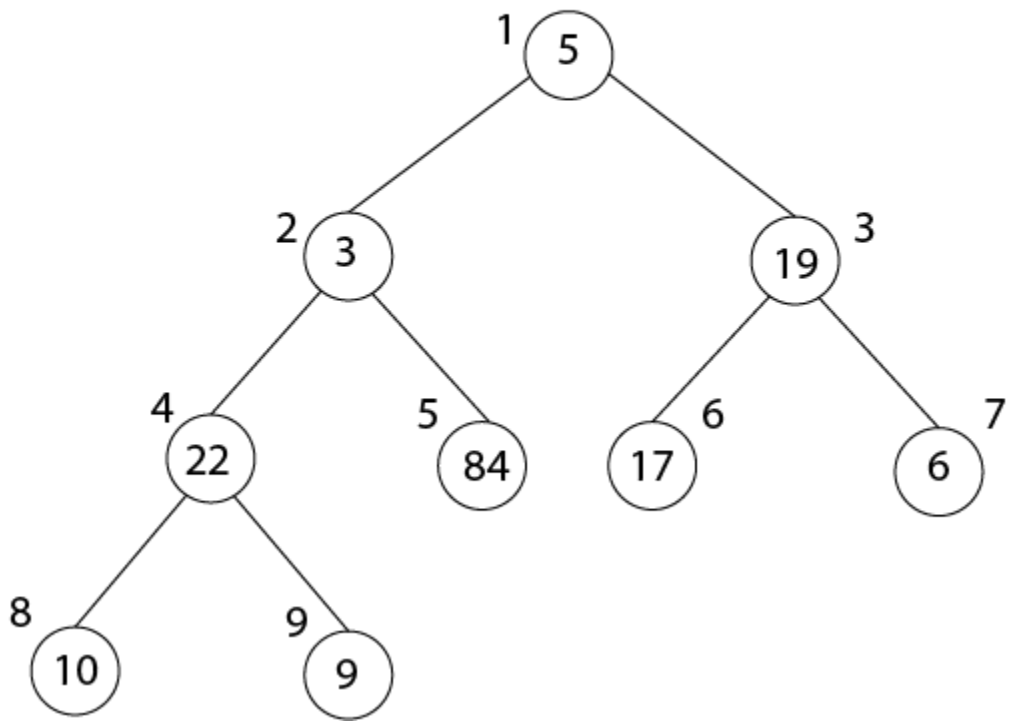
1. Heap-Size (A) =9, so first we call MAX-HEAPIFY (A, 4)
2. And I = 4.5= 4 to 1



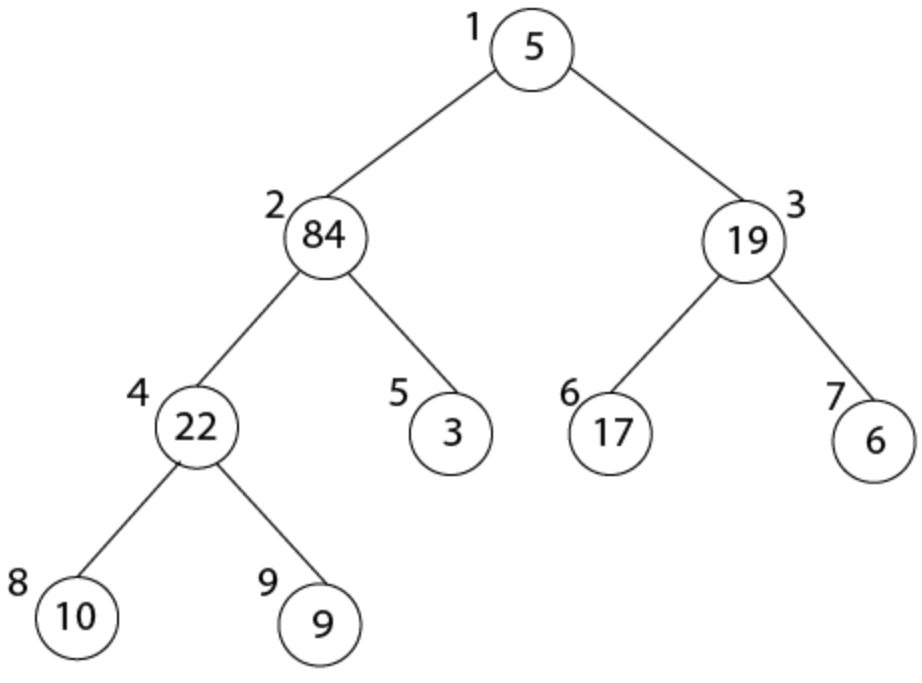
1. After MAX-HEAPIFY (A, 4) and i=4
2. L ← 8, r ← 9
3. l≤ heap-size[A] and A [l] >A [i]
4. 8 ≤9 and 22>10
5. Then Largest ← 8
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 9≤9 and 9>22
8. If largest (8) ≠4
9. Then exchange A [4] ←→ A [8]
10. MAX-HEAPIFY (A, 8)



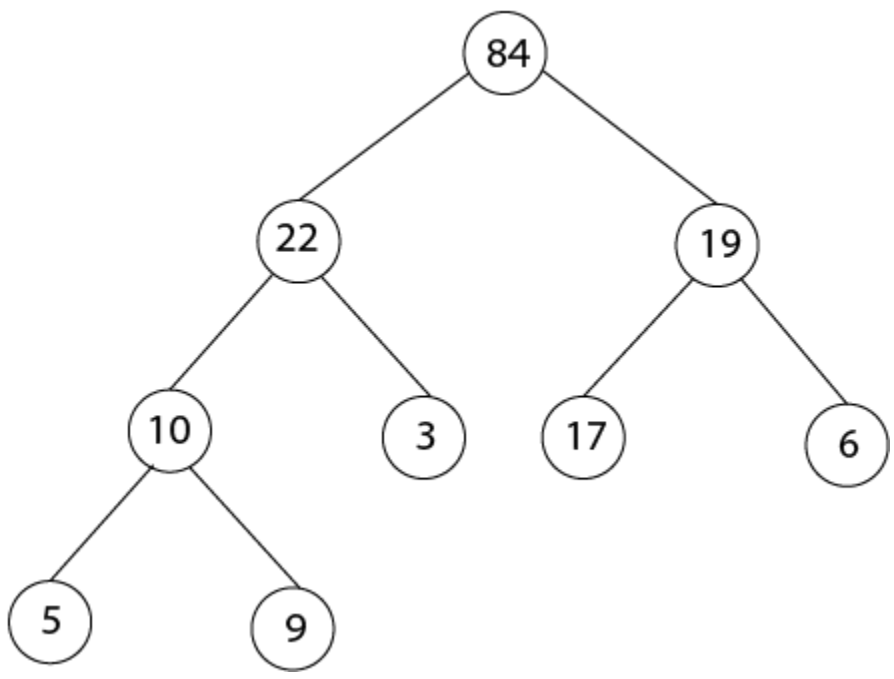
1. After MAX-HEAPIFY (A, 3) and i=3
2. l← 6, r ← 7
3. l≤ heap-size[A] and A [l] >A [i]
4. 6≤ 9 and 19>17
5. Largest ← 6
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 7≤9 and 6>19
8. If largest (6) ≠3
9. Then Exchange A [3] ←→ A [6]
10. MAX-HEAPIFY (A, 6)



1. After MAX-HEAPIFY (A, 2) and i=2
2. l ← 4, r ← 5
3. l≤ heap-size[A] and A [l] >A [i]
4. 4≤9 and 22>3
5. Largest ← 4
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 5≤9 and 84>22
8. Largest ← 5
9. If largest (4) ≠2
10. Then Exchange A [2] ←→ A [5]
11. MAX-HEAPIFY (A, 5)



1. After MAX-HEAPIFY (A, 1) and i=1
2. l ← 2, r ← 3
3. l≤ heap-size[A] and A [l] >A [i]
4. 2≤9 and 84>5
5. Largest ← 2
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 3≤9 and 19<84
8. If largest (2) ≠1
9. Then Exchange A [1] ←→ A [2]
10. MAX-HEAPIFY (A, 2)



Priority Queue:

As with heaps, priority queues appear in two forms: max-priority queue and min-priority queue.

A priority queue is a data structure for maintaining a set S of elements, each with a combined value called a key. A max-priority queue guides the following operations:

INSERT(S, x): inserts the element x into the set S, which is proportionate to the operation S=S∪[x].

MAXIMUM (S) returns the element of S with the highest key.

EXTRACT-MAX (S) removes and returns the element of S with the highest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is considered to be at least as large as x's current key value.

Let us discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM consider the MAXIMUM operation in θ (1) time.

HEAP-MAXIMUM (A)

1. return A [1]

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop of Heap-Sort procedure.

HEAP-EXTRACT-MAX (A)

```
1 if A. heap-size < 1
2 error "heap underflow"
3 max ← A [1]
4 A [1] ← A [heap-size [A]]
5 heap-size [A] ← heap-size [A]-1
6 MAX-HEAPIFY (A, 1)
7 return max
```

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index i into the array identify the priority-queue element whose key we wish to increase.

HEAP-INCREASE-KEY.A, i, key)

```
1 if key < A[i]
2 errors "new key is smaller than current key"
3 A[i] = key
4 while i>1 and A [Parent (i)] < A[i]
5 exchange A [i] with A [Parent (i)]
6 i =Parent [i]
```

The running time of HEAP-INCREASE-KEY on an n-element heap is O (log n) since the path traced from the node updated in line 3 to the root has length O (log n).

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new item to be inserted into max-heap A. The procedure first expands the max-heap by calculating to the tree a new leaf whose key is - ∞. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its right value and maintain the max-heap property

MAX-HEAP-INSERT (A, key)

```
1 A. heap-size = A. heap-size + 1
2 A [A. heap-size] = - ∞
3 HEAP-INCREASE-KEY (A, A. heap-size, key)
```

The running time of MAX-HEAP-INSERT on an n-element heap is O (log n).

Example: Illustrate the operation of HEAP-EXTRACT-MAX on the heap

1. A= (15,13,9,5,12,8,7,4,0,6,2,1)

Fig: Operation of HEAP-INCREASE-KEY

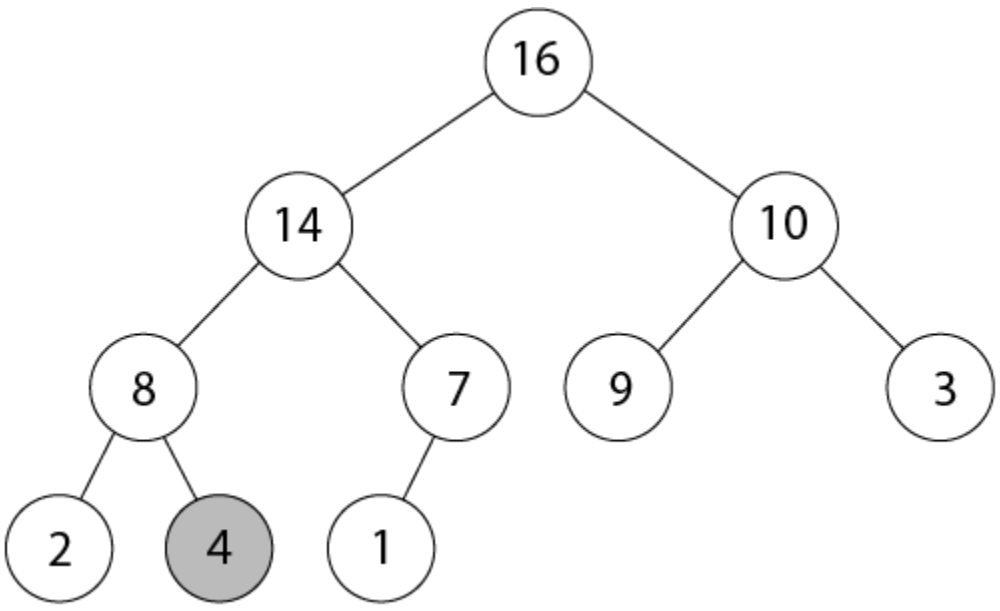


Fig: (a)

In this figure, that max-heap with a node whose index is 'i' heavily shaded

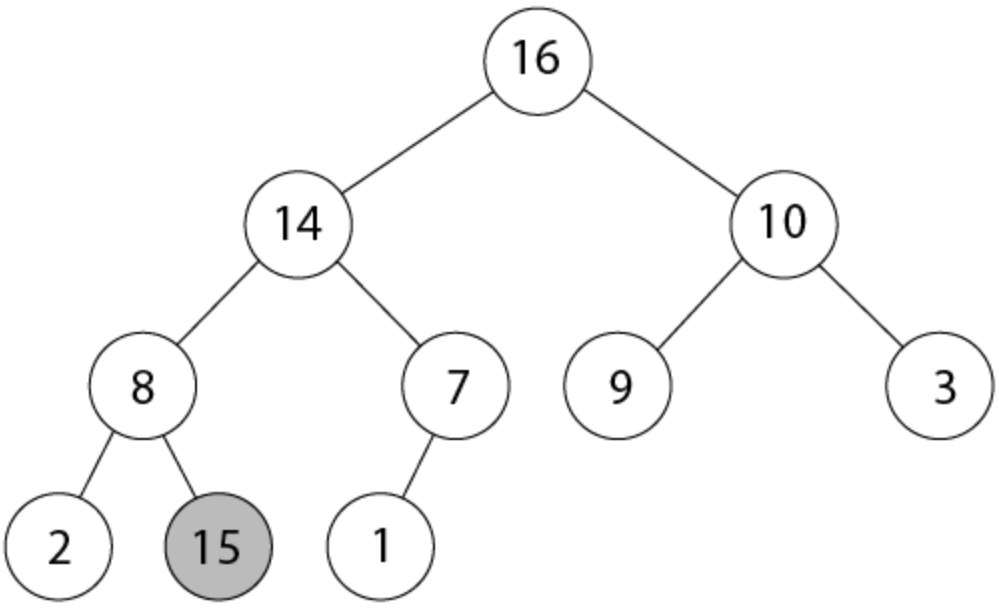


Fig: (b)

In this Figure, this node has its key increased to 15.

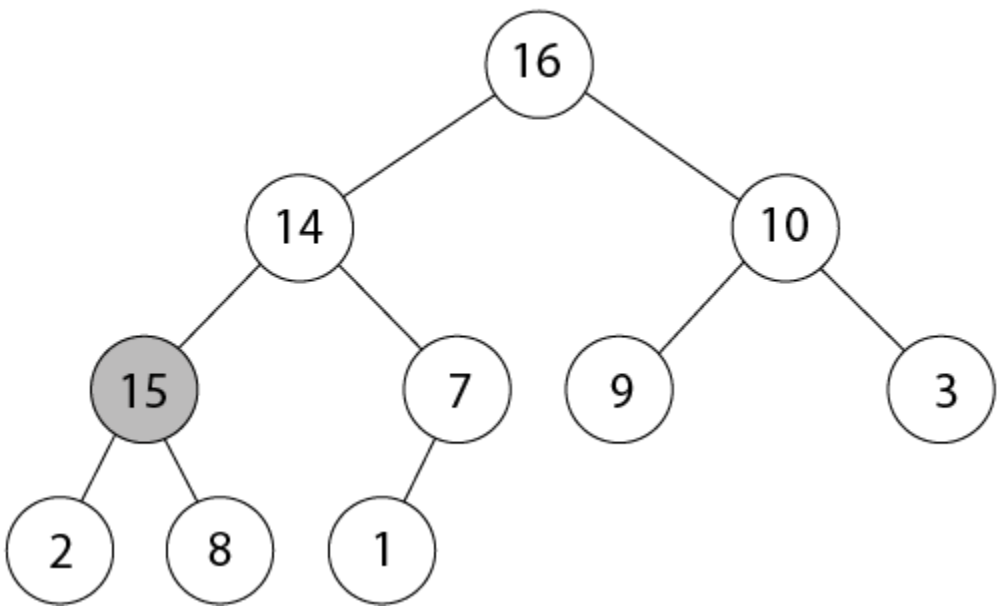


Fig: (c)

After one iteration of the while loop of lines 4-6, the node and its parent have exchanged keys, and the index i moves up to the parent.

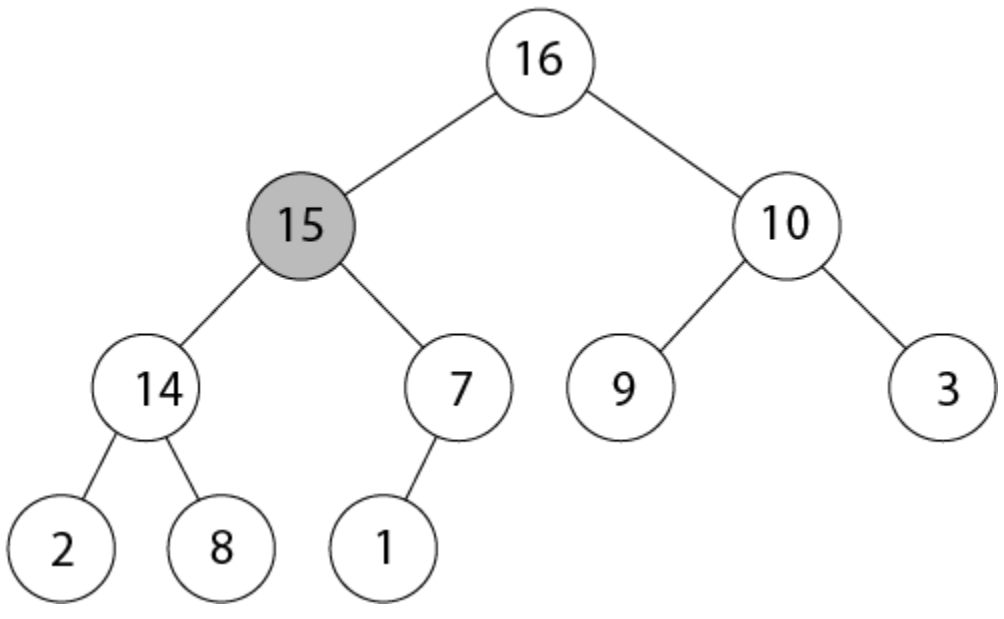


Fig: (d)

The max-heap after one more iteration of the while loops, the A [PARENT (i) ≥A (i)] the max-heap property now holds and the procedure terminates.

Heap-Delete:

Heap-DELETE (A, i) is the procedure, which deletes the item in node 'i' from heap A, HEAP-DELETE runs in O (log n) time for n-element max heap.

HEAP-DELETE (A, i)

```
1. A [i] ← A [heap-size [A]]
2. Heap-size [A] ← heap-size [A]-1
3. MAX-HEAPIFY (A, i)
```

Quick sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

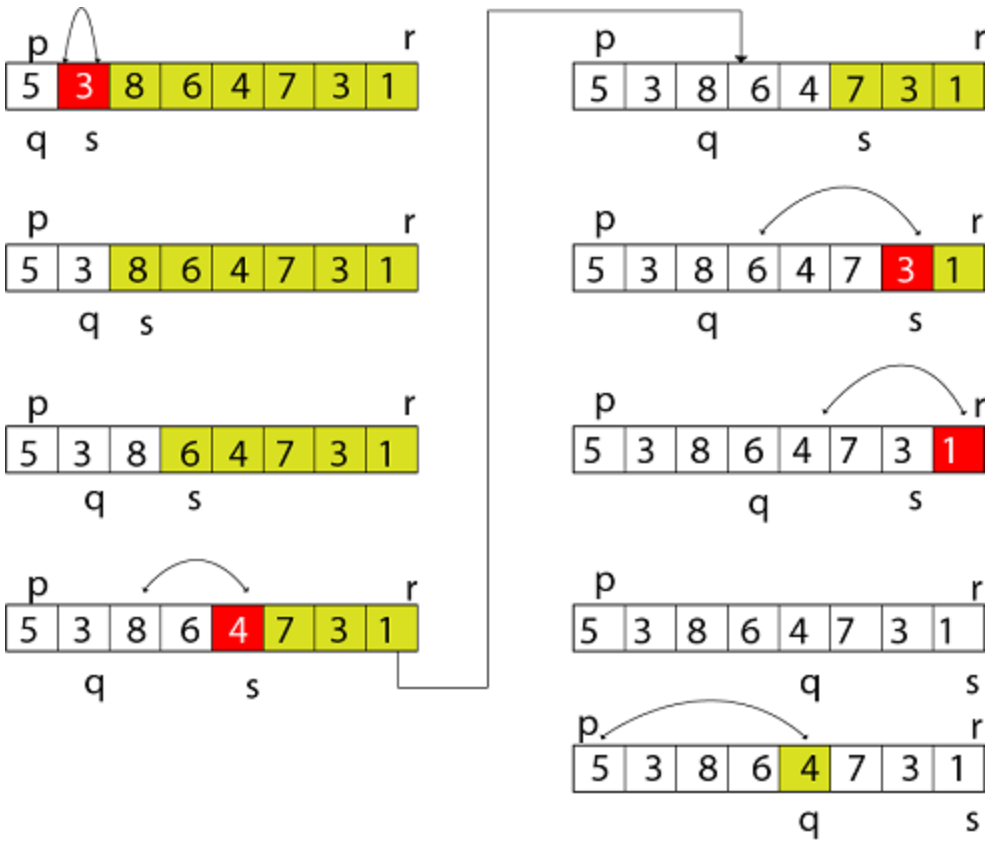
1. QUICKSORT (array A, **int** m, **int** n)
2. 1 **if** (n > m)
3. 2 then
4. 3 i ← a random index from [m,n]
5. 4 swap A [i] with A[m]
6. 5 o ← PARTITION (A, m, n)
7. 6 QUICKSORT (A, m, o - 1)
8. 7 QUICKSORT (A, o + 1, n)

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

1. PARTITION (array A, **int** m, **int** n)
2. 1 x ← A[m]
3. 2 o ← m
4. 3 **for** p ← m + 1 to n
5. 4 **do if** (A[p] < x)
6. 5 then o ← o + 1
7. 6 swap A[o] with A[p]
8. 7 swap A[m] with A[o]
9. 8 **return** o

Figure: shows the execution trace partition algorithm



Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list

88 77 60 **55** **90** 99

Sublist3 **Sublist4**

55 77 60 **88** 90 99

Sorted

55 **77** **60**

55 60 77

Sorted

Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

SORTED LISTS

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.

Equation:

1. T (n) =T(1)+T(n-1)+n

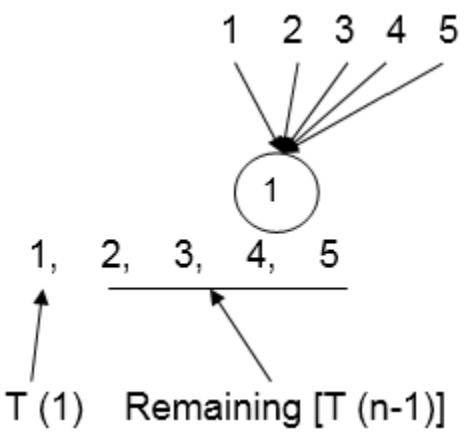
T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items.



Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n…… (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

Put T (n-1) in equation1

> By putting (n-1) in place of n in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n……(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

Put T (n-2) in equation (ii)

> By putting (n-2) in place of n in equation1

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

> By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n……(iii)

Note: for making T (n-4) as T (1) we will put (n-1) in place of '4' and if We put (n-1) in place of 4 then we have to put (n-2) in place of 3 and (n-3) In place of 2 and so on.

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n

T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+.....n

T (n) = (n-1) T (1) +T (1) +2+3+4+.....+n+1-1

[Adding 1 and subtracting 1 for making AP series]

T (n) = (n-1) T (1) +T (1) +1+2+3+4+..... + n-1

T (n) = (n-1) T (1) +T (1) + $\frac{n(n+1)}{2}$ -1

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

T (n) = (n-1) (0) +0+ $\frac{n(n+1)}{2}$ -1

T (n) =$\frac{n^2+n-2}{2}$

> Avoid all the terms expect higher terms n^2

T (n) =O (n^2)

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

Randomized Quick Sort [Average Case]:

Generally, we assume the first element of the list as the pivot element. In an average Case, the number of chances to get a pivot element is equal to the number of items.

1. Let total time taken =T (n)
2. For eg: In a given list
3. p 1, p 2, p 3, p 4.....pn
4. If p 1 is the pivot list then we have 2 lists.
5. I.e. T (0) and T (n-1)
6. If p2 is the pivot list then we have 2 lists.
7. I.e. T (1) and T (n-2)
8. p 1, p 2, p 3, p 4.....pn
9. If p3 is the pivot list then we have 2 lists.
10. I.e. T (2) and T (n-3)
11. p 1, p 2, p 3, p 4.....p n

So in general if we take the **Kth** element to be the pivot element.

Then,

$$T(n) = \sum_{K=1}^{n} T(K-1) + T(n-k)$$

Pivot element will do n comparison and we are doing average case so,

$$T(n) = n+1 + \frac{1}{n}\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)$$

N comparisons — Average of n elements

So Relational Formula for Randomized Quick Sort is:

$$T(n) = n+1 + \frac{1}{n}\left(\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)\right)$$

$$= n+1 + \frac{1}{n}(T(0)+T(1)+T(2)+...T(n-1)+T(n-2)+T(n-3)+...T(0))$$

$$= n+1 + \frac{1}{n}x2\ (T(0)+T(1)+T(2)+...T(n-2)+T(n-1))$$

1. $n\,T(n) = n(n+1) + 2\ (T(0)+T(1)+T(2)+...T(n-1)$.....eq 1

Put n=n-1 in eq 1

1. $(n-1)\,T(n-1) = (n-1)n + 2\,(T(0)+T(1)+T(2)+...T(n-2)$.....eq2

From eq1 and eq 2

$n\,T(n) - (n-1)\,T(n-1) = n(n+1)-n(n-1)+2\,(T(0)+T(1)+T(2)+?T(n-2)+T(n-1))-2(T(0)+T(1)+T(2)+...T(n-2))$

$n\,T(n) - (n-1)\,T(n-1) = n[n+1-n+1]+2T(n-1)$

$n\,T(n) = [2+(n-1)]T(n-1)+2n$

$n\,T(n) = n+1\,T(n-1)+2n$

$$\frac{n}{n+1}T(n) = \frac{2n}{n+1} + T(n-1) \qquad \text{[Divide by n+1]}$$

$$\frac{1}{n+1}T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n} \qquad \text{[Divide by n] .....eq 3}$$

Put n=n-1 in eq 3

$$\frac{1}{n}T(n-1) = \frac{2}{n} + \frac{T(n-2)}{n-1}\text{.....eq4}$$

Put 4 eq in 3 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1}\text{.....eq5}$$

Put n=n-2 in eq 3

$$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2}\text{.....eq6}$$

Put 6 eq in 5 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2}\text{.....eq7}$$

Put n=n-3 in eq 3

$$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3}\text{.....eq8}$$

Put 8 eq in 7 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3} \ldots\ldots\ldots\ldots \text{eq9}$$

2 terms of $\frac{2}{n+1} + \frac{2}{n} = T(n-2)$

3 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} = T(n-3)$

4 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2}$ =T(n-4)

From 3eq, 5eq, 7eq, 9 eq we get

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots + \frac{2}{3} + \frac{T(n-(n-1))}{n-(n-2)} \ldots\ldots\ldots \text{eq10}$$

From 3 eq $\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$

Put n=1

$$\frac{T(1)}{2} = \frac{2}{2} + \frac{T(0)}{1}$$

$$\frac{T(1)}{2} = 1$$

From 10 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + \frac{T(1)}{2}$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 1$$

Multiply and divide the last term by 2

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 2 \times \frac{1}{2}$$

$$= 2\left[\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots\ldots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$= 2 \sum_{2 \le k \le n+1}^{n} \frac{1}{k} = 2 \int_{2}^{n+1} \frac{1}{k}$$

Multiply & divide k by n

$$= 2 \int_{2}^{n+1} \frac{1}{\frac{k.n}{n}}$$

Put $\frac{k}{n} = x$ and $\frac{1}{n} = dx$

$$\frac{T(n)}{n+1} = 2 \int_{2}^{n+1} \frac{1}{x}\, dx$$

$$= 2 \log x \Big|_{2}^{n+1}$$

=2[log (n+1)-log2]

T (n) =2(n+1) [log (n+1)-log2]

Ignoring Constant we get

T (n) =n log n

T (n) =O (n log n)

NOTE: $\int \frac{1}{x} dx = \log x$

Is the average case complexity of quick sort for sorting n elements.

3. Quick Sort [Best Case]: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition	Complexities
1.Quick Sort[Worst Case]	T(n)=T(n-1)+T(0)+n	T(1)=0	$T(n)=n^2$
2.Quick Sort[Average Case]	$T(n)=n+1+\frac{1}{n}(\sum_{K=1}^{n} T(k-1)+T(n-k))$		T(n)=nlogn

Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

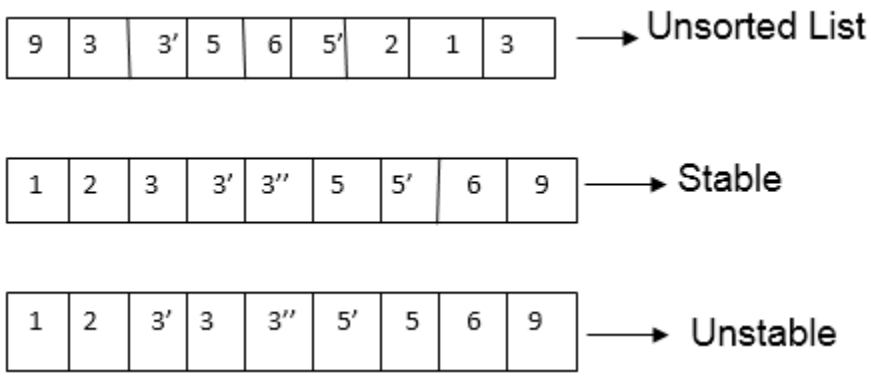
Some Sorting Algorithm is stable by nature like Insertion Sort, Merge Sort and Bubble Sort etc.

Sorting Algorithm is not stable like Quick Sort, Heap Sort etc.

Another Definition of Stable Sorting:

A Stable Sort is one which preserves the original order of input set, where the comparison algorithm does not distinguish between two or more items. A Stable Sort will guarantee that the original order of data having the same rank is preserved in the output.

In Place Sorting Algorithm:



1. An In-Place Sorting Algorithm directly modifies the list that is received as input instead of creating a new list that is then modified.
2. In this Sorting, a small amount of extra space it uses to manipulate the input set. In other Words, the output is placed in the correct position while the algorithm is still executing, which means that the input will be overwritten by the desired output on run-time.
3. In-Place, Sorting Algorithm updates input only through replacement or swapping of elements.
4. An algorithm which is not in-place is sometimes called not-in-Place or out of Place.
5. An Algorithm can only have a constant amount of extra space, counting everything including function call and Pointers, Usually; this space is O (log n).

Note:

1. Bubble sort, insertion sort, and selection sort are in-place sorting algorithms. Because only swapping of the element in the input array is required.
2. Bubble sort and insertion sort can be applying as stable algorithms but selection sort cannot (without significant modifications).
3. Merge sort is a stable algorithm but not an in-place algorithm. It requires extra array storage.
4. Quicksort is not stable but is an in-place algorithm.
5. Heap sort is an in-place algorithm but is not stable.