Sorting Techniques

0 Criteria for Analysis:

1. Number of comparisions.
2. No. of Swaps
3. Adaptive ⟶ If the list is already sorted, it should take lesser time than usual.
4. Stable ⟶ If there are any duplicate elements in original list, then in the sorted list there order must be preserved.
5. Extra memory

Ex:

A	B	C	D	E	F	G	H
5	9	6	3	6	4	2	7

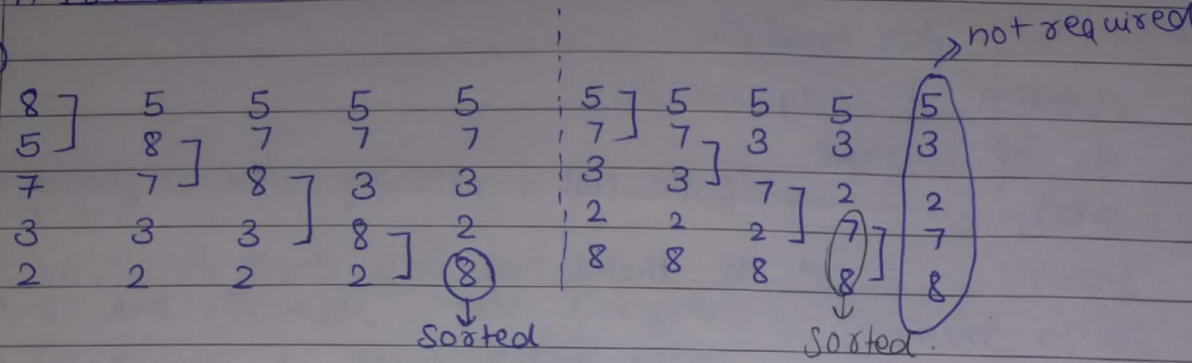
↳ This should come first than the other 6.

1. Bubble	$O(n^2)$	Comparision based sort	Worst case performance.
2. Insertion			
3. Selection			
4. Heap sort	$O(n \log n)$		
5 Merge sort			
6 Quick sort			
7 Tree sort			
8 Shell sort	$O(n^{3/2})$		
9 Count Sort	$O(n)$	Index Based Sort	
10 Bucket/Bin Sort			
11 Radix Sort			

Bubble Sort

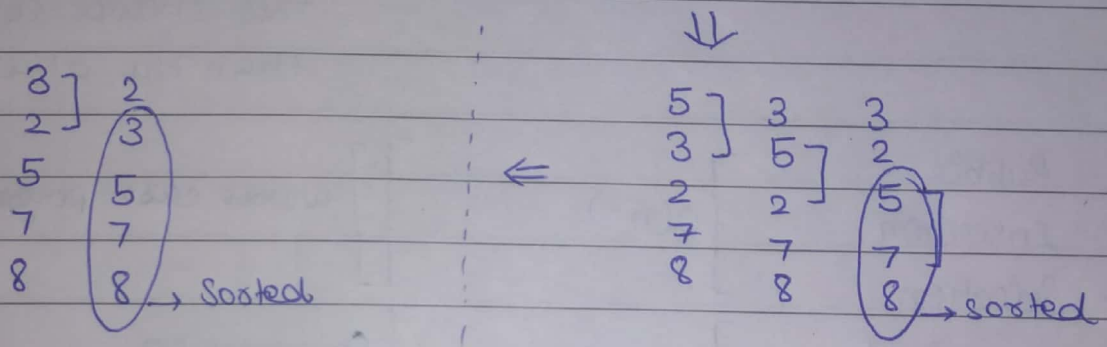
A [8|5|7|3|2]

(1 Pass)



5 ellements : 4 comp.
4 swap.

3 comp.
3 swap (Not actual but maximum swaps)



1 comp.
1 swap.

2 comp.
2 swap

ellements
↳ (n = 5) No. of passes: 4 → (n-1) passes

No. of comp: 4+3+2+1

for n: $(n-1)+(n-2)+\dots 2+1$

$= \frac{n(n-1)}{2} = O(n^2)$

No. of swaps: 4+3+2+1

Same = $\frac{n(n-1)}{2} = O(n^2)$

- time complexity depends on No. of comparision

here = $O(n^2)$

Algo: (Optimized code)

```
void BubbleSort (int A[], int n)
{   int flag;
    for (i=0; i<n-1; i++)  // passes
    {   flag=0;
        for (j=0; j<n-1-i, j++)  // [j<n-1-i]
        {   if A[j] > A[j+1]
                {Swap (A[j], A[j+1]);}
                flag=1;
        } if (flag==0)
            break; // to check if there are no swaps done.
    }
}
```

Bcz in each pass, one comp is reduced.

∴ If no swaps done it means the list is already sorted. and it is adaptive bcz it is taking lesser time when it is sorted.

2	2	2	2
3	3	3	3
5	5	5	5
7	7	7	7
8	8	8	8

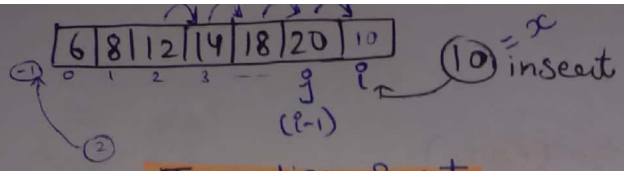
n-1 comp.
0 swaps.
Time complex. = O(n)

minimum time → O(n) → Best
maximum time → O(n²) → worst

Stable ✓

8	8
8	8
13	13
15	15

Recurrence relation: $T(n) = T(n-1) + n$



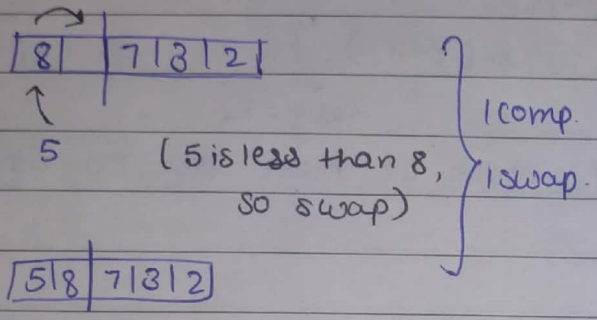
Date / /
Page No.

Insertion Sort

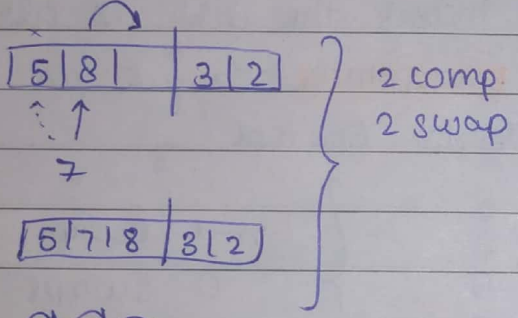
A [8|5|7|3|2] (n=5)

→ (Here assume 1st element is already sorted, so start with 2nd element.

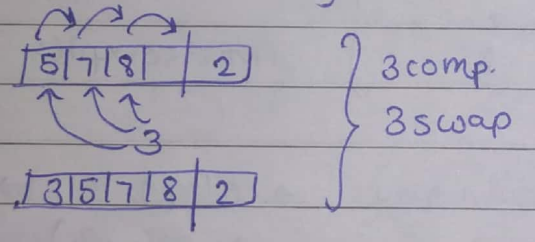
1st pass:



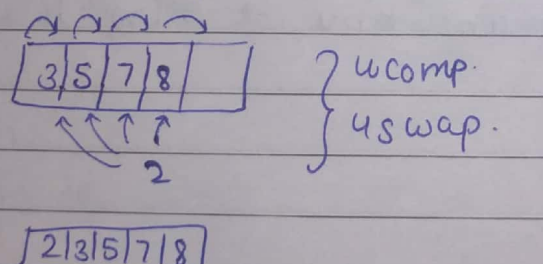
II pass:



III pass



IV pass



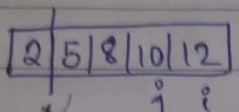
- No. of passes: (here) = 4, n = 5
  ↳ (n-1)
- No. of comp: 1+2+3+4..
  ↳ 1+2+3+4+...n-1 = $\frac{n(n-1)}{2}$ = n^2 → Complexity
- No. of swap: 1+2+3+4
  ↳ n^2

Algorithm

```
void insertion(int A[], int n)        // n: No. of elements
{
  for(i=1; i<n, i++)                  // passes.
  { j=i-1;
    x=A[i];
    while(j>-1 && x<A[j])
    {A[j+1]=A[j];
     j--;
    }
    A[j+1]=x;
  }
}
```

It is better for sorting of linked list bcz here we don't have to shift the element or create a new linklist.

- **Adaptive:** (By nature it is Adaptive) ✓



No of comp: $(n-1) \rightarrow O(n) \rightarrow$ For Best case.
No of swap: $0 \rightarrow O(1)$

- Best case: (When the list is already in Ascending order) $\rightarrow O(n)$
- Worst Case: (When it is in descending order.
  ↳ $O(n^2)$

- **Stable** ✓

Comparision between Bubble sort and Insertion sort

	Bubble sort	Insertion sort
Min comp.	$O(n)$	$O(n) \rightarrow$ Already in Ascending
Max comp.	$O(n^2)$	$O(n^2) \rightarrow$ Descending
min swap	$O(1)$	$O(1) \rightarrow$ Ascending
max swap	$O(n^2)$	$O(n^2) \rightarrow$ Descending
Adaptive	✓	✓] → only these are adaptive
Stable	✓	✓] → These and merge sort
Linked list	NO	Yes
K passes advantage	Yes ↳ gives K largest element	NO

Reccurence Relation:

$$T(n) = \begin{cases} O(1), & \text{if } n = 1 \\ T(n-1) + O(n), & \text{if } n > 1 \end{cases}$$

Date / /
Page No.

Selection Sort

- We work here on position or index not element.
- → good for less no. of swaps.
- → gives the K smallest element on K passes.

	A
0	8
1	6
2	3
3	10
4	9
5	4
6	12
7	5
8	2
9	7

i, K ← 0; i → till here i will move

j → scan and find minimum element and store it in K then swap it with A[i]

Algorithm

```
void selectionsort(int A[], int n)
{
  for(i=0; i<n-1; i++)  // Passes.
  { for(j=K=i, j<n, j++)
    {
      if(A[j] < A[K])
        { K=j; }
    }
    swap(A[i], A[K]);
  }
}
```

- No. of passes : (n-1)
- No. of comp: $1+2+3+\dots(n-1) = O(n^2)$
- No. of swap: 1 swap in each pass.
  - ↳ So (n-1) pass = (n-1) swap
    $= O(n)$.
  - ↳ Least no. of swap. (only algorithm)

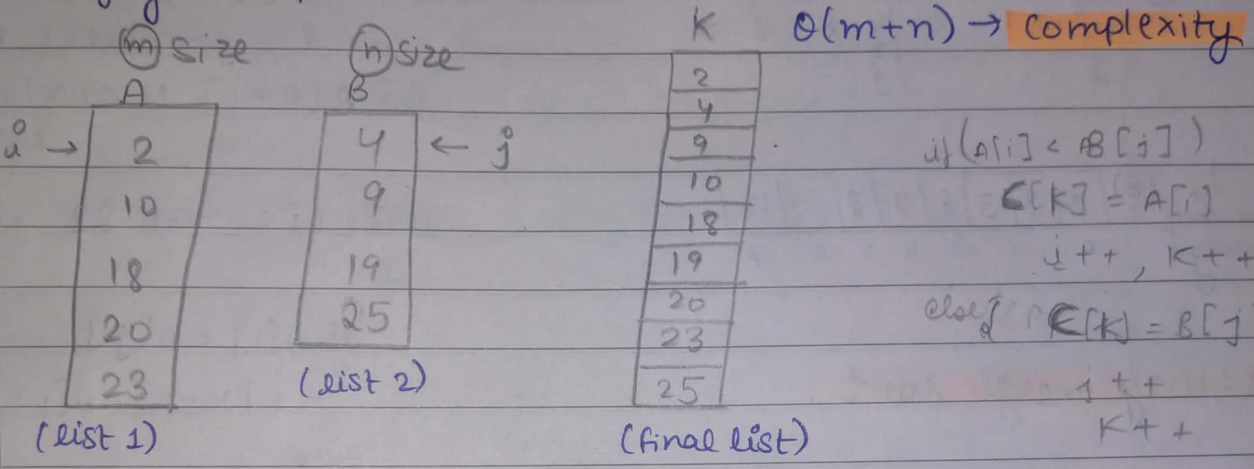
Imp: Bubble, Selection] on K passes → (K largest element), (K smallest element)

- Not Adaptive. [Even if the list is sorted it will take $O(n^2)$ complexity in all cases] → Worst, Best, Average
- Not Stable

Reccurence Relation: $T(n) = T(n-1) + n$

Merge Sort

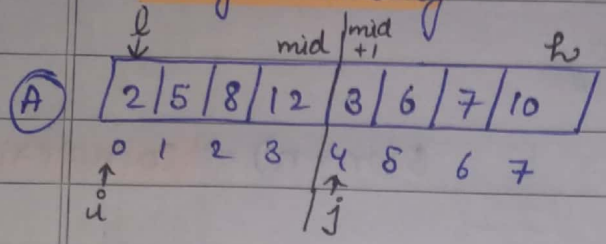
- **Merging** → combing 2 sorted list into a single list.
  - → merging 2 list
  - → merging 2 list in single array
  - → merging multiple list.



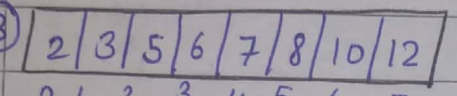
Algorithm:

```
Void merge (int A[], int B[], int m, int n)
{
  int i, j, K;
     i = j = K = 0;
  while (i<m && j<n)  //Stoping condition
   {
     if(A[i] < B[j])
          C[K++] = A[i++];
     else
          C[K++] = B[j++];
   }
  for( ; i<m ; i++)        // for remaing ellements.
       C[K++] = A[i];
  for( ; j<n ; j++)        //   "     "     "
       C[K++] = B[j];
}
```

merging 2 list from a single array:



Auxilary array

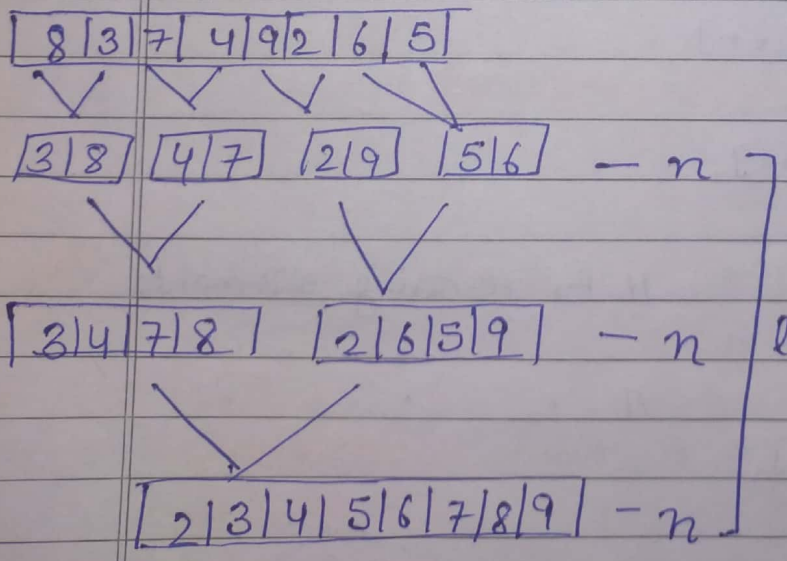


2-way merge sort:

merge sort involves recursively the array into 2 parts, sorting and finally merging them.

↳ follows Divide and Conquer.

Iterative merge sort



Time complexity: $n \log n$ (all cases)

Algorithm:

```
void merge(int A[], int l, int mid
           int h)
{ int i, j, k;
  i = l; j = mid+1, k = l;
  int B[h+1] // auxilary
                array
while (i<= mid && j<=h)
 {
   if A[i] < A[j]
      B[k++] = A[i++];
   else
      B [k++] = A[j++];
 }
for ( ; k<= mid, i++)
      B[k++] = A[i];
for ( ; j <= h, j++)
      B[k++] = A[j];
for (i=l ; i<=h, i++)
      A[i] = B[i];
}
```

Date / /
Page No.

Iterative algorithm:

```
Void IMergeSort(int A[], int n)
{
  int p, l, h, mid, i;
  for(p=2; p<=n, p=p*2) // passes (sets of 2)
  {
    for(i=0; i+p-1<n; i=i+p)
    {
      l=i;
      h=i+p-1;
      mid=(l+h)/2;
      Merge(A, l, mid, h);
    }
  }
}
```

n = 8

p = 2

i + p - 1

0 + 2 - 1

① < 8

i = 0 + 2

i = 2

2 + 2 - 1

③ < 8, i =

2 + 2

④

Recursive algorithm:

```
void RMergeSort(int A[], int l, int h)
{
  int mid;
  { mid=(l+h)/2; if (l<h)
    RMergeSort(A, l, mid);
    RMergeSort(A, mid+1, h);
    Merge(A, l, mid, h);
  }
}
```

Reccurence Relation:

$$\begin{cases} O(1) & , n=1 \\ 2T\left(\frac{n}{2}\right)+n & , n>1 \end{cases}$$

Quick Sort → Divide and conquer

- Pick 1 element as pivot

↳ [6|5|8|9|3|10|16|12|16]

pivot ↑ i = low

j = high ↑

↳ i searches for greater element and stops when it is found.

↳ j searches for lower element than pivot and stop when it is found.

→ then swap them.

→ repeat this till (i < j)

after : return j.

Algo:

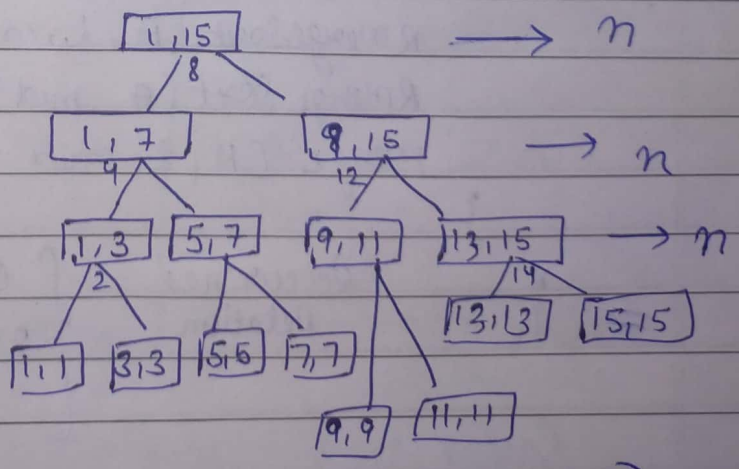
```
Quicksort(l, h)
{ if (l < h)
    { j = partition(l, h);
      Quicksort(l, j);
      Quicksort(j+1, h);
    }
}
```

Algorithm:

```
Partition(l, h)
{ pivot = A[l];
  i = l ; j = h ;
  while(i < j)
  {
    do
    { i++;
    } while (A[i] ≤ pivot)
    do
    { j--;
    } while(A[j] > pivot);
    if(i < j)
      swap(A[i], A[j]);
  }
  swap(A[l], A[j]);
  return j;
}
```

Analysis:

(Suppose the partioning is done in middle in every stage:



O(n log n))

Date / /
Page No.

Complexity:

When first element is selected as pivot:

- Best case → If partitioning is in middle
  Time: $O(n \log n)$
- Worst case → if partitioning is any end. (already sorted list)
  Time: $O(n^2)$
- Average case → $O(n \log n)$

When middle element is pivot:

- Best case → sorted list $O(n \log n)$
- Worst Case → partitioning at the end $O(n^2)$

NOTE:

- In selection sort, we selecti a position and the find the element.
- In Quick sort, we select a element and find its position.

So, Quick sort is also called:

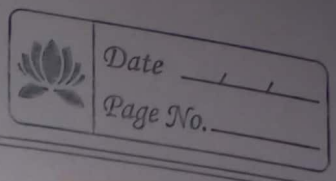
→ Selection Exchange Sort
→ Partition Exchange Sort

Reccurence Relation:

Worst case:
$$T(n) = \begin{cases} T(1), & n=1 \\ T(n-1)+n, & n>1 \end{cases}$$
↳ (n^2) complex

Best case:
$$T(n) = \begin{cases} T(1), & n=1 \\ 2T(N/2), & n>1 \end{cases}$$
↳ $O(n \log n)$

Count Sort



A

6	3	9	10	15	6	8	12	3	6
0	1	2	3	4	5	6	7	8	9

Step → 1

Take another of Size of (max. element + 1) in array A.

Initialize it to 0

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Step → 2

B

0	0	0	2	0	0	3	0	1	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Step → 3

A

3	3	6	6	6	8	9	10	12	15
0	1	2	3	4	5	6	7	8	9

Step → 4

Algorithm:

```
void CountSort (int A[], int n)
{  int max, i;
   int *c;
   max = findmax (A, n);   // max. element
   c = new int [max+1]
   for (i=0; i<max+1; i++)
       c[i] = 0            // initialize it to 0
   for (i=0; i<n; i++)
     { c[A[i]]++; }
   i=0; j=0;
   while (i<max+1)
   {  if (c[i]>0
      {  A[j++] = i;
         c[i]--;
      }
      else
         i++;
   }
}
```

Complexity

O(n).

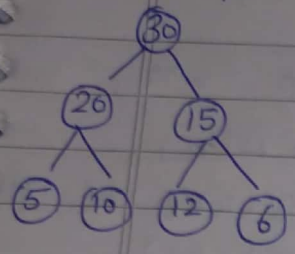
→ consumes a lot of space.

→ space consuming algorithm.

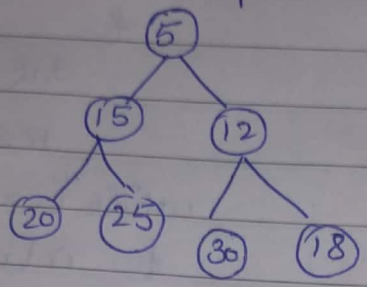
→ only efficient if the input data is not greater than the no. of objects to be sorted.

Heap

Max Heap



Min Heap



(2) Every node should have greater / smaller than or equal to its child node.

Node at index i
Left child at 2*i;
Right child at 2*i+1;

(1) What is Heap
(2) Insert in a Heap
(3) Deleting from Heap
(4) Heap sort
(5) Heapify

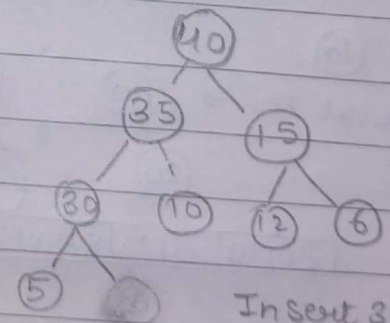
Conditions:

(1) Heap must be a complete Binary Tree. (there should not be any gaps in an array).

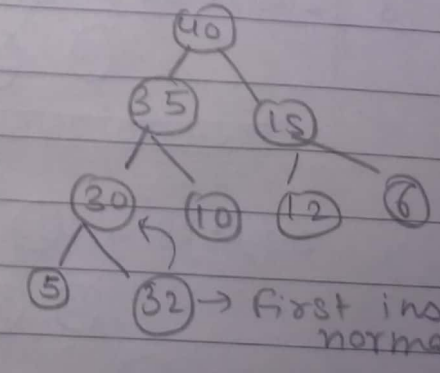
(3) Heap is a complete binary tree. So its height will always be logn.

Insertion:

(1) Insert an element in an binary order and then compare it with ancestor.
(2) If the element is greater than ancestor move them at its place.

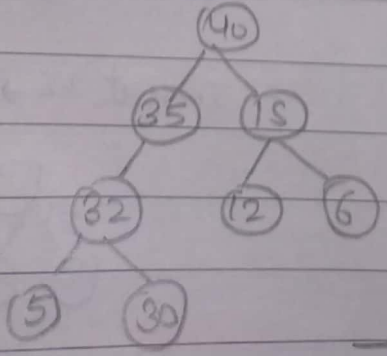


Insert 32



→ first insert normaly

Rearrange



n = index of last pointing element.

Date / / Page No.

(A) | 40 | 35 | 15 | 30 | 10 | 12 | 6 | 5 | 20 | 6 |
1 2 3 4 5 6 7 8 9 10

i n

① temp = A[n] or temp = 36

② Compare while ((temp > i/2) && (i > 1))
{ A[i] = A[i/2]
i = i/2
}

← Final tree

Algorithm (Inserting)

```
void Insert (int A[], int n)
{
    int temp, i = n;
    temp = A[n];
    while ((i > 1) && (temp > A[i/2]))
    {   A[i] = A[i/2];
        i = i/2;
    }
    A[i] = temp;
}
```

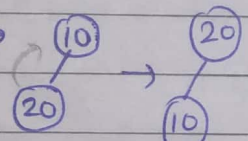
Complexity:

Depends on Height of Binary tree.
⇒ Θ(log n).

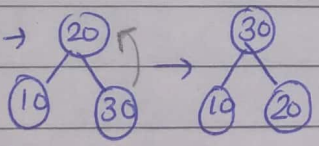
- **Creating a Heap:**

A | 10 | 20 | 30 | 25 | 5 | 40 | 35 |

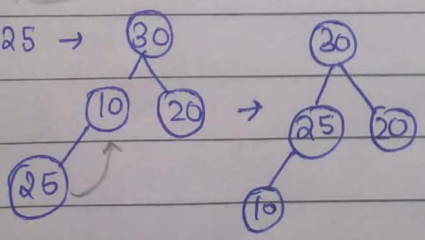
1. (10)

2. Insert 20 → 

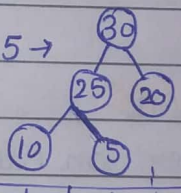
| 20 | 10 | 30 | 25 | 5 | 40 | 35 |

3. Insert 30 → 

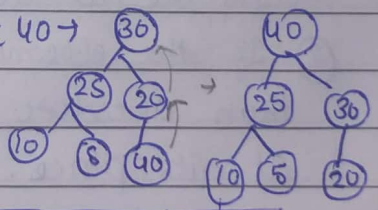
| 30 | 10 | 20 | 25 | 5 | 40 | 35 |

4. Insert 25 → 

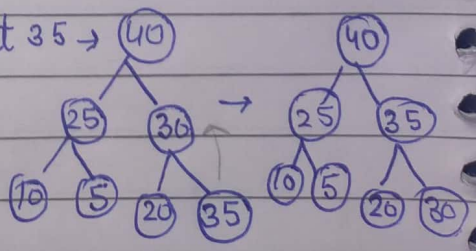
| 30 | 25 | 20 | 10 | 5 | 40 | 35 |

5. Insert 5 → 

| 30 | 25 | 20 | 10 | 5 | 40 | 35 |

6. Insert 40 → 

| 40 | 25 | 30 | 10 | 5 | 20 | 35 |

7. Insert 35 → 

| 40 | 25 | 35 | 10 | 5 | 20 | 30 | ✓

Algorithm:

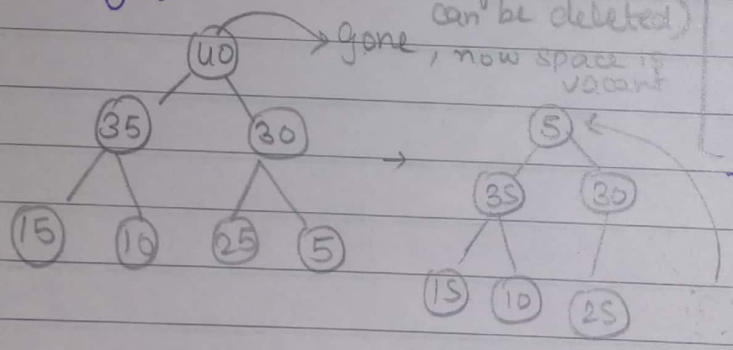
```
void Create(
{ int A[] = {                    }
  int i;
  for (i=2, i<=7, i++)
  {
     Insert(A, i);
  }
}
```

(Creating Heap)

Complexity:

- element (1) - log n
- element (n) → n log n

- **Deleting from Heap:** (only greatest / last elem can be deleted)



Heap sort

n log n (1) Create Heap of 'n' element
n log n (2) Delete n elem 1 by 1.
2n log n

Complexity of Heap sort
= O(n log n)

Algorithm for deleting:

```
void delete (int A[], int n)
{ int x, i, j; val,
  val = A[1]
  x = A[n],
  A[1] = A[n] // last elem is copied at 1st place
  A[n] = val,
i=1, j=2*i, // i=1 left
            j = child of i
while (j < n-1)
{ if (A[j+1] > A[j]) // right child is greater than left
    j=j+1; // j becomes right child
if (A[i] < A[j])
  { swap(A[i], A[j]
    i=j; //
    j=2*j; // move j to left child
else }
    break; // if root is greater than break
}
```

copy last element to first
↓ Now rearrange so that it becomes a max Heap.
↳ (a) compare children of new node root
(b) Swap the greater children with new root (if root is smaller).