Linear Search:

- Two possibilities in the linear search one is data is present and other is data is not present.
- If the data is present print the data.
- Searching start from start to end and traverse the each element.
-

0	1	2	3	4	5	6	7	8
15	5	20	11	7	19	42	8	50

n = 9

data = 42

```
for(int i=0 ; i<n ; i++){
    if(a[i]==data){ → only enter when data found.
        cout<<"Data Found"<<i ;
        break ; → only break the loop if value found.
    }
```

case 1

```
if(i==n){
    cout<<"Not Found";
}
}
```

case 2

→ create variable outside the loop and set to 1 if found +1.

```
if(found==0){
    cout<<"Not Found";
}
```

M.Khaqan Nasir

Time complexity:

- Best case $\Theta(1)$

Found value at the first position.

- Wrost case $\Theta(n)$

Found value at the Last position.

- Average case

Sum of all the cases and divide by the no. of cases.

$$\text{sum} \quad \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Scanned with CamScanner

Binary Search:

- The first condition of binary search is data is in sorted form.
- If the data is not sorted first sort the data and then apply binary search.
- Binary search divide the array into two sub arrays by finding the middle index.
- Find mid index by divide 2 after addition of first and Last index. $\frac{first + Last}{2}$
- 3 conditions in Binary Search.
  - (i) Data is on Mid. (data == mid)
  - (ii) Data is greater than Mid (data > Mid)
  - (iii) Data is Less than Mid. (data < mid)
- $\frac{l+r}{2} = \frac{0+9}{2} = 4$

0	1	2	3	4	5	6	7	8	9
10	20	30	35	40	45	60	70	80	90

l ← 10, r → 90, mid → 40

n = 10

- we divide array into two parts in every condition becomes false.
- If the left becomes greater than right index means value not found.

```
int BinarySearch (array, size, data) {
    left = 0, right = n-1;
  while (left < right) {  → Execute when data is found
      mid = (left + right)/2  → Calculate Mid
    if (data == array[mid]) {
        return mid;  → return value
    }
    else if (data < array[mid]) {
        right = mid-1;
    }
    else { left = mid +1;
        }  }
```

Scanned with CamScanner

return -1; → means data not found.

}

Time complexity:

we divide the time complexity into half in the binary search.

Worst case $\Theta(\log n)$

Best case $\Theta(1)$

- When in any case loops execution going to half means time complexity is $\log n$.

Bubble Sort:

- In the bubble sort we sort the data in the ascending and the descending order.
- compare first element with the second element If is not in the correct order than swap the elements and if ~~not~~ yes move to the next.

A

0	1	2	3	4
15	16	6	8	5

$\boxed{n=5}$

15 16 6 8 5
15 6 16 8 5
15 6 8 16 5
15 6 8 5 16 → Pass 1.

Pass 2 ←
15 6 8 5 16
6 15 8 5 16
6 8 15 5 16
6 8 5 15 16
6 8 5 15 16

- After pass 1 in the ascending order the largest data move to the right and same for descending but the smallar data move.
- After pass 1 start pass 2.

Pass 3 ←
6 8 5 15 16
6 8 5 15 16
6 5 8 15 16
6 5 8 15 16
6 5 8 15 16

Scanned with CamScanner

6	5	8	15	16	
5	6	8	15	16	
5	6	8	15	16	→ Pass 4
5	6	8	15	16	
5	6	8	15	16	

- To calculate how many passes are required is the $\boxed{n-1}$
- One loop for passes and the inner loop for comparison.

```
for (int i = 0 ; i <n-1 ; i++) {
    for (int j = 0 ; j<n-1 ; j++) {
        if ( A[j] > A[j+1]) {
            int temp = A [j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

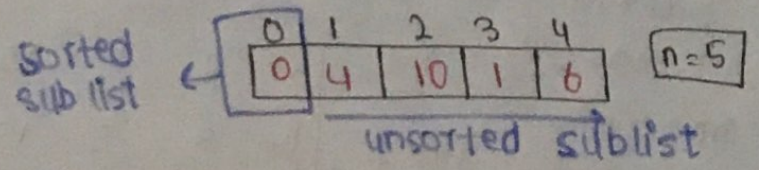
→ $j < n-1-i$

For avoid unnecessary comparison so use this condition.

- No. of comparisons depends on the outer loop.

Insertion Sort:

- In this we divide the array in into two sub list.
  (i) sorted sublist
  (ii) unsorted sublist
- We take one value from unsorted datalist and insert the value into sorted list.



- one by one take value from unsorted sublist and insert on sorted sublist on appropriate place.

Scanned with CamScanner

unsorted sublist → [4] → temp

sorted ← [4 | 5 | 10 | 1 | 1 | 6 | 2]

unsorted

• move one step to right.

```
for (int i = 1; i<n; i++) {        → 1 bcz first consider as sort
    temp = a[i];                   → stored in temp.
    j = i-1;
    while (j>=0 && a[j] > temp) {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = temp;
}
```

Time Complexity:

- If the array is given in descending order and you have to solve in the ascending order this is the worst case. $O(n^2)$
- In the best case the time complexity is $O(n)$, when the data is already given in ascending order.

Selection Sort:

- Same as the insertion sort the array is divided into sub arrays one is sorted array and second is unsorted array.
- But one change is that sorted array is empty and the remaining will be the unsorted.

	0	1	2	3	4	5
sorted	7	4	10	8	3	1

unsorted

$n=6$

- We find the minimum element in the unsorted subarray and then swap with the first element of the unsorted subarray.

minimum = 1

11	4	10	8	3	7

sorted — unsorted

Pass 1

Now do the same step for unsorted subarray again. find minimum and replace with the unsorted ~~sub~~ first element of unsorted subarray.

minimum = 3

1	3	10	8	4	7

sorted — unsorted

Pass 2

minimum = 4

1	3	4	8	10	7

sorted — unsorted

Pass 3

minimum = 7

1	3	4	7	10	8

sorted — unsorted

Pass 4

minimum = 8

1	3	4	7	8	10

sorted — unsorted

Pass 5

- No need to sort the last element. If we have $\boxed{n=6}$, then the passes required is $5 = \boxed{n-1}$.
- Two loops are required for selection sort, one the outer loop for passes and the inner loop for find minimum and then swap the value.

```
for (int i=0 ; i<n-1; i++) {
    int min=i;
    for (int j=i+1; j<n; j++) {
        if (a[j] < a[min]) {
            min=j;
        }
    }
    if (min != i) {
        swap (a[i], a[min]); }
}
```

No [redacted] 6

Time complexity:

- Time complexity of selection sort in $\Theta(n^2)$ both in worst and best case because every time loop go to n.

Quick Sort:

- Quick sort follow the divide and conquer technique.
- we can choose one element from the array as the pivot element. Its our choice to chose first, last or any random element as pivot element.
- divide and conquer means the complete array will be dividing into sub arrays and the process of dividing the array into subarray is partitioning.
- The method of partition is known as the backbone of Quick sort.

	0	1	2	3	4	5	6
A	10	5	1	2	9	16	11

pivot = 10

- ~~Sort~~ divide the array in a way in which all the elements are less than pivot element will be on left side and greater on right.
- The same element as pivot element depends on you to move on which side.
-

key

Partition 1	Pivot	Partition 2
values < pivot		values > pivot

4

2, 1, 5, 9	10	11, 16

not need to solve index 4

Now apply same process on 0-3 and 5-6 because 20 is on proper place.

Scanned with CamScanner

L.B									U.B
	7	6	10	5	9	2	1	15	7
	start →							← end	

pivot = 7

- If we found any element from left to right is lesser or equal to that element increment the start otherwise stop viceversa for end.
  - 7 == pivot increament start
  - 6 < pivot increment start
  - 10 > pivot stop start.

Now check end.

7 == pivot stop end

Now swap the start and end.

7	6	7	5	9	2	1	15	10
		start						end

After 1 swap.

- Again do the same steps.
  - 7 == pivot increment start
  - 5 < pivot increment start
  - 9 > pivot stop start.

Now check end.

- 10 > pivot decrement end
- 15 > pivot decrement end
- 1 < pivot stop end.

Now swap start and end.

7	6	7	5	1	2	9	15	10
				start		end		

After 2nd swap.

Scanned with CamScanner

- Now check again
  - 1 < pivot increment start
  - 2 < pivot increment start
  - 9 > pivot start stop.

→ Now check for the end
  - 9 > pivot decrement end
  - 2 < pivot end stop.

- In this case start cross the end so no need to swap. After this swap the pivot element with end.

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

↓

[5]

- Now check all the elements are less than 7 on the left side and greater than 7 on the right side.

→ Apply the same funda on [0-4] and [6-8]

```
int Partition (A, lb, ub) {
        pivot = a[lb];
        start = lb;
        end = up;
    while(start < end) {
            while (a[start] <= pivot) {
                start ++; }
            while (a[end] > pivot) {
                end --; }
            if (start < end) {
                swap; } }

            swap (a[lbB], a[end]);
                return end;
        }
```

Scanned with CamScanner

```
Quick sort (A, lb, ub) {
    if (lb < ub) {
        loc = Partition(A, lb, ub);
        Quicksort (A, lb, loc-1);
        Quicksort (A, loc+1, ub);
    }
}
```

Time complexity:

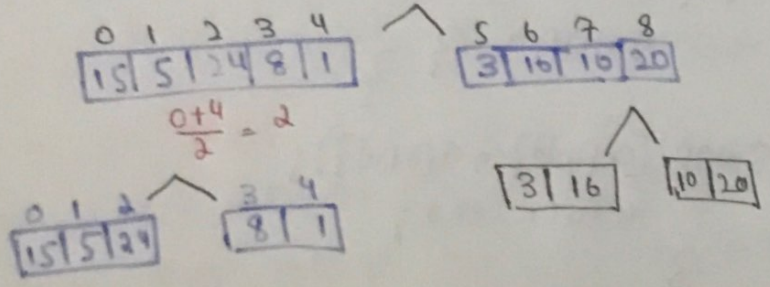
- In the worst case $\Theta(n^2)$.
- In the best and average case $\Theta(n \log n)$.

Merge Sort:

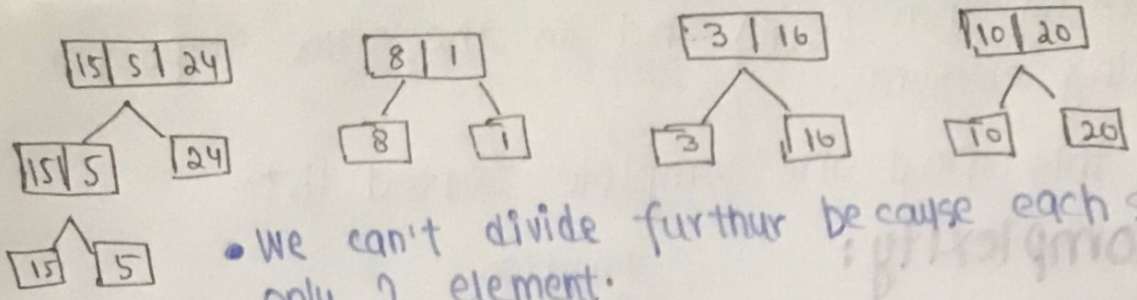
- This sort follow the principle of divide and conquer same as quick sort.
- In this technique array divide into n sublist and each list has 1 element.
- We can divide the list until found 1 element.
- After this merge the each list and found the new sorted list.
- Merging the sublists until we found a 1 complete list. (sorted).

0	1	2	3	4	5	6	7	8
15	5	24	8	1	3	16	10	20

Step 1. divide the list into two sublist by finding the mid index. $\frac{0+8}{2} = 4 \rightarrow$ mid.



Scanned with CamScanner

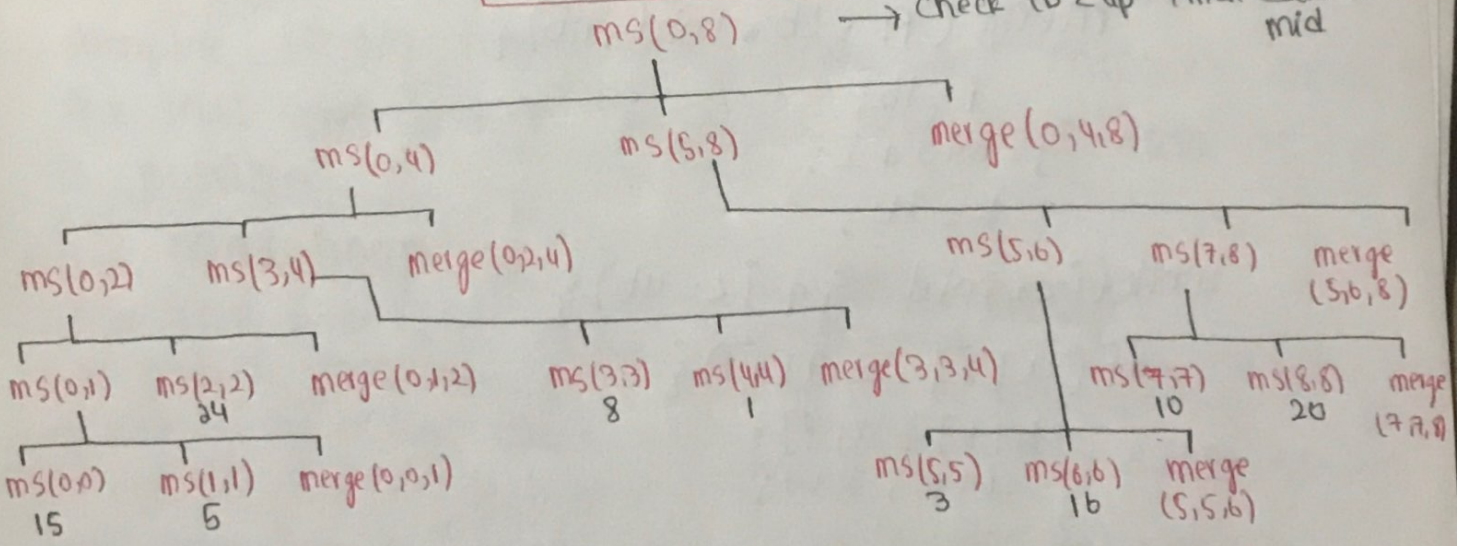


- We can't divide furthur because each sublist has only 1 element.

→ Step 2 :- merging the sublist.

```
MergeSort(A, lb, ub){
    if(lb < ub){
        mid = (lb+ub)/2;
    MergeSort(A, lb, mid);
    Merge Sort(A, mid, ub);
    merge(A, lb, mid, ub);
    }
}
```

Recursive Approach.



- First they go to (0,8) then (0,4) then (0,2) then (0,1) then (0,0) then (1,1) then merge (0,0,1) → After merge [5 | 15]
- then (2,2) then merge (0,1,2) → After merge [5 | 15 | 24]
- then (0,2) then (3,4) then (3,3) than (4,4) then merge (3,3,4) → After merge [1 | 8]
- then (3,4) then merge (0,2,4) → After merge [1 | 5 | 8 | 15 | 24]

Scanned with CamScanner

- then (0,4) then (5,8) and do the same steps for remaining elements.
- After this find one complete sorted list.

Time Complexity:

- Time complexity of merge sort (function) is $\Theta(m+n)$, m is the first list and n is the second list.
- We have 3 pointer variables one is i, j ank k. i for the first list, j for the second list and k for the 3rd list. We compare i with j found the smallest element and place on k.
- Increment the k and the list pointer in which element is smaller. (i/j).
- If one list is complete then put the remaining list element as it is.

```
merge (A, lb, mid, ub){
        i = lb;
      j = mid + 1;
        k = lb;
while(i <= mid && j <= ub){
    if(a[i] <= a[j]){
        b[k] = a[i];
        i++; k++;
    }
    else{ b[k] = a[j];
        j++; k++;
    } }
if (i > mid) {
        while (j <= ub) {
            b[k] = a[j];
            j++; k++; }
```

Scanned with CamScanner

```
else {
    while( i <= mid) {
        b[k] = a[i];
        i++; k++; }
    }
}
```

- After this copy the value of b to A and then close the merge function using for loop.

```
for (k= lb ; k <=ub ; k++) {
    a[k] = b[k];
}
}
```

- Both in best and worst case of merge sort the time complexity is $\Theta(n\log n)$.

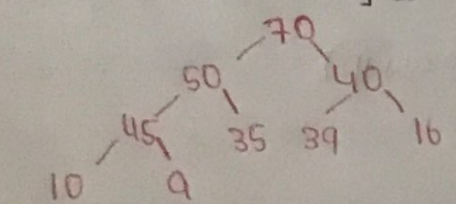
Heap:

- Tree based data structure in which the tree is complete binary tree. (All the levels are complete except the last level and in the last level the data as left as possible.)

Max heap

→ For every node i, the value of node is less than or equal to its parent value, except the root node bcz root node has no parent.

$A[Parent(i)] \geq A[i]$



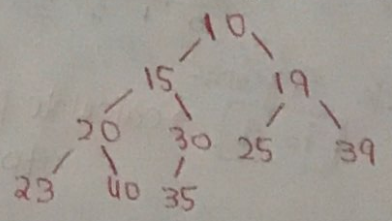
	1	2	3	4	5	6	7	8	9
A	70	50	40	45	35	39	16	10	9

- root is maximum

Min heap

→ For every node i, the value of node is greater than or equal to its parent values, except root node.

$A[parent(i)] \leq A[i]$



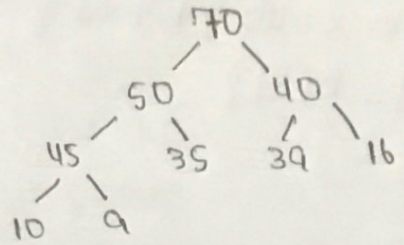
	1	2	3	4	5	6	7	8	9	10
A	10	15	19	20	30	25	39	23	40	35

- root is minimum.

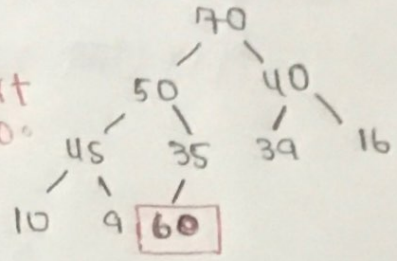
Scanned with CamScanner

Insert in Max heap:

- Insertion is heap sort start from leaf node and follow the complete binary tree principle and data as left as possible.
- Insert the value in the last position as left as possible and after inserting compare the value with the parent node if parent is small then swap the value.



→ insert 60.



70	50	40	45	35	39	16	10	9
0	1	2	3	4	5	6	7	8

70	50	40	45	35	39	16	10	9	60
1	2	3	4	5	6	7	8	9	10

→ Now compare the 60 with his parent if parent is smaller then swap the 60 with his parent.

calculate parent = $i=10 \rightarrow \left\lfloor \frac{i}{2} \right\rfloor = \frac{10}{2} = 5$

35 < 60 (swap)

0	1	2	3	4	5	6	7	8	9
70	50	40	45	60	39	16	10	9	35

→ Again check 60 with his parent. We check 60 until 60 is less than to his parents.

$\boxed{i=4}$ → calculate parent $\left\lfloor \frac{i-1}{2} \right\rfloor = 1$

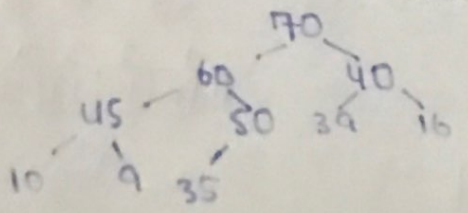
50 < 60 (swap)

0	1	2	3	4	5	6	7	8	9
70	60	40	45	50	39	16	10	9	35

→ check Again:-

$\boxed{i=1}$ → calculate parent $\left\lfloor \frac{i-1}{2} \right\rfloor = 0$

70 < 60 (NO need to swap)



Scanned with CamScanner

→ Time complexity in max heap to insert any element is $\Theta(\log n)$.

→ Because maximum no. of swaping are equal to height of the tree.

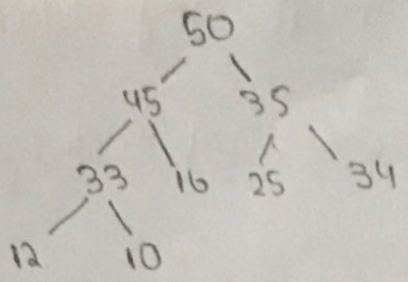
→ height of tree is always $(\log n)$ Θ.

```
insertHeap (A, n, value) {
    n=n+1;
    A[n]=value;
    i=n;
while (i>1) {
        parent = ⌊(i-1)/2⌋;
    if (A[parent] < A[i]) {
        swap(A[parent], A[i]); i=parent
        }
    else {return;
        }
    }
}
```

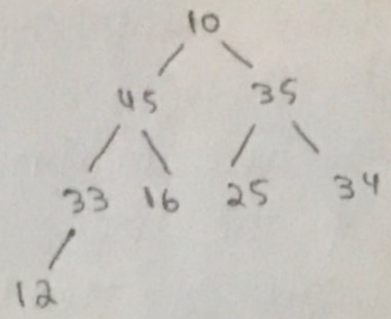
Deletion in Max Heap:-

- Only delete the root node (Rule).
- After delete the least element move to the root node place.
- After replace check the Max heap condition if not satisfy then compare with his both childs, swap with his greater child
- Do this process until the children will be smaller compared to his parent.

Scanned with CamScanner



→ Delete 50 and replace with his least node.



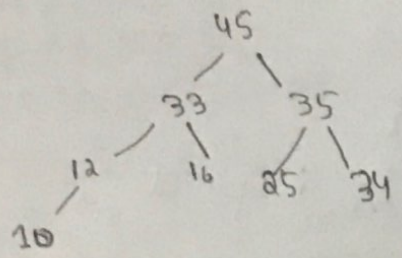
50	45	35	33	16	25	34	12	10

10	45	35	33	16	25	34	12	

n-1

- Now check the Max heap condition, if not satisfy than replace with his greater child.

10 ⊃ 45 , 10 ⊃ 33 , 10 ⊃ 12



left child = 2 x i = 2
Right child = (2xi)+1 = 3

45	33	35	12	16	25	34	10	

Empty space

So, use empty space and store 50

- Follow the same procedure and delete node one by one.
- After deleting all nodes your array is in sorted order.
- Both in best and worst case the time complexity of deletion in Max heap is (and average case also).

$\boxed{\Theta(n \log n)}$

Scanned with CamScanner