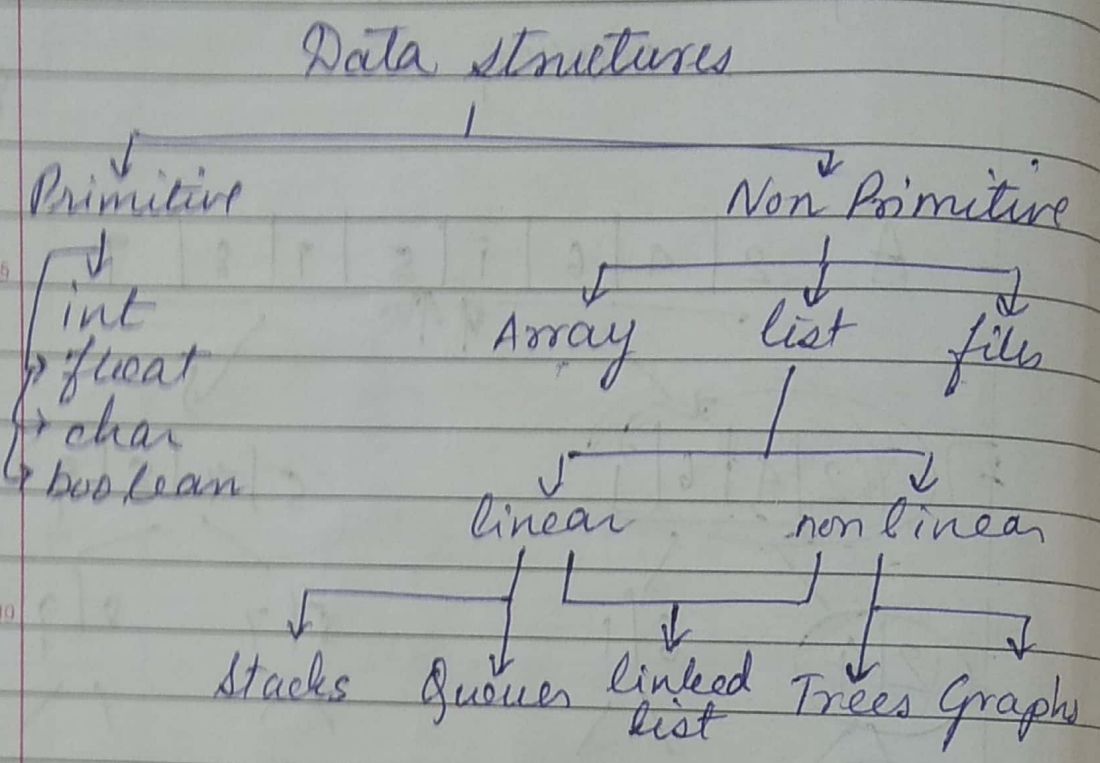
DS = Specialised format for orgn & storing data / coll^n of orgnized data



Primitive : built in OS, already available
eg int, float, char, boolean
Non primitive : Not available as built in
formed using built in datatypes

Array : Sequential form of data
single type only.
Files : Stores info, used for read & write Data = small records or lines
List : Sequential form of data, like DA Array
a) Linear : Stored sequentially
1) Stack 2) Queue
b) Non linear : non sequentially
1) Trees 2) Graphs
Linked Lists can be in linear or non linear form

Bubble Sort

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

Complexity of bubblesort is $O(n^2)$ in both best case & worst case.

Selection Sort

```
for i ← 0 to n-2
{
    min ← i
    for j ← i+1 to n-1
    {
        if A[j] < A[i] then
            min ← j
    }
    if min != i then
    {
        t = A[i];
        A[i] = A[min];
        A[min] = t;
    }
}
```

Complexity of Selection sort is $O(n^2)$ in both best & worst case.

sort(A)
n ← Camlin Page
Date
0 → n-1
n

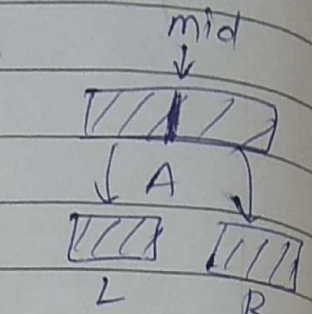
Merge Sort.

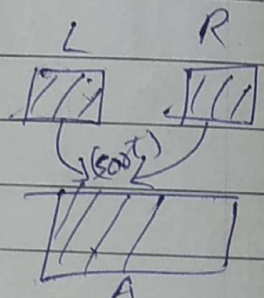
```
sort(A)
{
    n ← length(A)                          } base con^n
    if (n<2)                               } constant time C1
        return;
    mid = n/2;
    length of left = mid;
    length of right = n - mid;

    for i = 0 to mid-1                     } C2 n times
        L[i] ← A[i];
    for i = mid to n-1
        R[i] ← A[i];

    sort(L);                               } T(n/2)
    sort(R);                               } T(n/2)
    merge(L,R,A);                          } C3 n+C4
}

merge(L,R,A)
{   nL = length(L)
    nR = length(R)
    i=0, j=0, k=0;
    while(i<nL && j<nR)
    {   if(L[i] < R[j])
        {   A[k] ← L[i];
            k++; i++; }
        else
        {   A[k] ← R[j];
            k++; j++; }
    }
```





$T(n) = 2T(n/2) + (c_2 + c_3)n + (c_1 + c_4)$ if $n > 1$

c, if $n = 1$

$2T(n/2) + cn + cn$ if $n > 1$

```
while (i<nL)
{       A[K] ← L[i];
        k++; i++; }
while (j<nR)
{       A[K] ← R[j];
        k++; j++;
}
```

Complexity: ~~Worst case : O(n²)~~

Worst & Best case : O(nlogn)

Quick Sort

```
Qsort (A, Start, End)
{       if (Start < end)
        {   pIndex ← partition (A, S, E);
            sort (A, Start, pIndex-1);
        }   sort (A, pIndex, end);
}
```

```
Partition (A, Start, End)
{       pivot ← End; pIndex ← Start;
        for i ← start to end-1
        {   if (A[i] <= A[pivot])
            {   swap (A[i], A[pIndex])
                pIndex++; }
        }
        swap(A[pIndex], A[End]);
        return pIndex;
}
```

Complexity: W = $O(n^2)$, B = $O(n \log n)$

Bubble Sort = Compare two adjacent els. If in wrong order, swap

Selection Sort = Take 1 el, select largest element or least el & bring to top by swap.

Merge Sort = Divide 1 problem into sub problems. Reduces time complexity. Sort smaller groups & then merge.

Quick Sort = Divides the problem & uses recursion. Based on pivot el, divided into two parts, less than and greater than. Repeat until we have single element.

SEARCHING

eg.

2	5	8	13	9	32	81	55	76
0	1	2	3	4	5	6	7	8

If we want to search x = 81.
Output will be position/index of el if it exists.
Here Output should be 6.
If doesn't exist, Output is -1.

Two methods of Searching are:
1) Linear Search.
Starts from beginning,

```
Search (A, n, x)
{
    for i ← 0 to n-1
    {
        if (A[i] == x)
        {
            return i;
        }
    }
    return -1;
}
```

Here, worst case is O(n) time complexity.
If best case, O(1).

2) Binary Search.

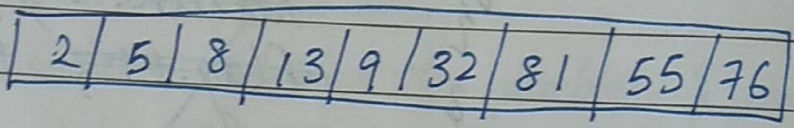
Divides array into two parts. Continuously divides these & searches till we get single el w/c we want.

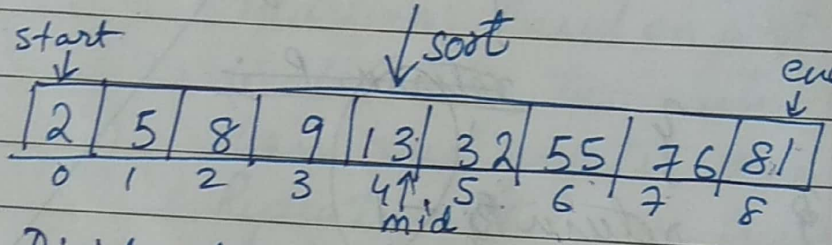
Main reqmt : Sorted array.

Time complexity is O(log n).

Comparison reduced 3/4th.

eg.

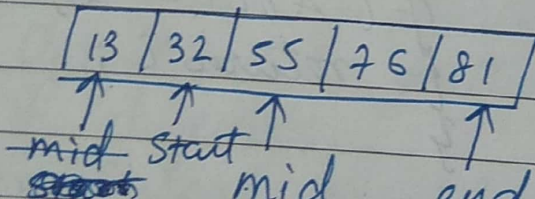




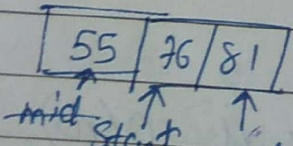
Divide element into two parts. May take floor or ceil in case of odd no.

Compare with mid element.

If less, we can neglect the ones less than these.



Again compare.



(A, n, x)

And finally we get [81]

```
Search (A, n, x)
{       start = 0
        end = n-1
        while ( start <= end)
        {
            mid = (start + end) / 2
            if ( x == A[mid] )
            {       return mid }
            else if ( x < A[mid] )
            {       end = mid - 1 }
            else
            {       start = mid + 1 }
        }
        return (-1)
}
```