# Batch Name   :   PreCAT OM22 – Fast Track Batch
# Module Name :   Data Structures

Q. What is a data structure?

We want to store marks of 100 students
int m1, m2, m3, m4, ..., m100;//400 bytes
- we want to sort marks of 100 students in a descenfding order
sorting

int marks[100];//400 bytes

we want to store info of 100 students:

rollno:	int
name :	char [ ]
marks:	float

struct employee emp[100];

- to learn data structure is not to learn any programming language, it is nothing but to learn **algorithms.**

Q. What is a Program?
- a program is a finite set of instructions written in any programming language given to the machine to do specific task.

Q. What is an algorithm?
- an algorithm is a finite set of instructions written in human understandable language like english, if followed, acomplishesh given task.

- A Program is an impementation of an Algorithm
- An Algorithm is like a blueprint of a Program.

Algorithm ==> User / Pseudocode ==> Programmer User
Program ==> Machine

Q. What is a Pseudocode?
- an algorithm is a finite set of instructions written in human understandable language like english with some **programming constraints**, if followed, acomplishesh given task, such algo is also called as pseudocode.

- pseudocode is a special form of an algo traversal on an array/to scan array => to visit each array element sequentially from first element max till last element.

Algorithm : to do sum of array elements:
step-1: initially take sum as 0.
step-2: start traversal of an array and add each array element into the sum sequentially.
Step-3: return final sum

Pseudocode/Special form of an algo: ==> Programmer User

```
Algorithm ArraySum( A, n)//A is an array having size n
{
        sum = 0;
        for( index = 1 ; index <= n ; index++ ){
                sum += A[ index ];
        }

        return sum;
}
```

Program: ==> Machine

```
int array_sum( int arr[ ], int size ){
        int sum = 0;
        for( int index = 0 ; index < size ; index++ ){
                sum += arr[ index ];
        }

        return sum;
}
```

- An algorithm is a solution of a given problem
- algorithm = solution

- Problem: can we have multiple solutions for single problem

Pune => Mumbai
multiple paths exists between Pune & Mumbai

efficient/optmized path ==>
    distance in km
    cost required for
    medium
    traffic conditions

- One problem may has multiple solutions
Searching: to search given key element into a collection/list of elements
1. lienar search
2. binary search

Sorting: to arrange data elements in a collection/list of elements either in an ascending order (or in a desceding order).
1. bubble sort
2. selection sort
3. insertion sort
4. merge sort
5. quick sort
6. heap sort
7. radix sort
8. shell sort
etc....

- When we have multiple solustions/algo's for a single problem, we need to select an efficient solution/algo out of them, and to decide efficiency of an algo's we need to do their analysis.

- **analysis of an algo** is a work of calculating how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- there are two measures of analysis of an algo:
1. time complexity of an algo is the amount of time i.e. computer time it needs to run to completion.

2. space complexity of an algo is the amount of space i.e. computer memory it needs to run to completion.

Linear Search:

Best case : occurs if key is found at first position in only 1 comparison: O(1)
for size of an array = 10 => no. of comparisons = 1
for size of an array = 20 => no. of comparisons = 1
for size of an array = 30 => no. of comparisons = 1
.
.
for size of an array = 50 => no. of comparisons = 1
for size of an array = 100 => no. of comparisons = 1

for size of an array = n => no. of comparisons = 1

Worst case : occurs if either key is found at last position or key is not found O(n).

for size of an array = 10 => no. of comparisons = 10
for size of an array = 20 => no. of comparisons = 20
for size of an array = 30 => no. of comparisons = 30
.
.
for size of an array = 50 => no. of comparisons = 50
for size of an array = 100 => no. of comparisons = 100

for size of an array = n => no. of comparisons = n

best case: if an algo takes min amount of time to run to completion
worst case: if an algo takes max amount of time to run to completion
average case: if an algo takes neither min nor max amount of time to run to completion

for size of an array = 10 => 20 bytes/40 bytes ==> 10 units
for size of an array = 20 => 40 bytes/80 bytes ==> 20 units

for size of an array = n ==> n units
Space Complexity = O(n).

+ Rule: if running time of an algo is having any additive/substractive/multiplicative/divisive constant then it can be neglected.
e.g.
O(n + 3) => O(n)
O(n – 4) => O(n)
O(n / 3) => O(n)
O(2 * n) => O(n)

+ Binary Search:

by means of calculating mid pos big size array gets divided logically into two subarrays: left subarray & right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

if size of an array = 1000
iteration-1: [0 1 2 3 ..... 1000] => mid -> 1 comparison => 500
[0.. 499] 500 [501 .... 1000]

iteration-2: [501 .... 1000] => mid = 750 => 1 comparison => 250
[501......... 749] 750 [751 .... 1000]

iteration-3: [501 .... 750] 1 comparins => 125

after iteration-1: no. of cmp = 1, n/2 => $T(n/2^1) + 1$
after iteration-2: no. of cmp = 2, n/4 => $T(n/2^2) + 2$
after iteration-3: no. of cmp = 3, n/8 => $T(n/2^3) + 3$
.
.

let us assume k no. of iterations takes
after iteration-k: no. of cmp = k, $n/2^K$ => $T(n/2^K) + K$

# DS DAY-02:
Rule: if running time of an algo is having a polynomial, then in its time complexity only leading gets considered.
e.g.
O(n^3 + n + 5) => O(n^3)

Sorting Algorithm:
1. Selection Sort

total no. of comparisons = (n-1) + (n-2) + (n-3) + ...

total no. of comparisons = n(n – 1) / 2
=> (n^2 – n) / 2
=> O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

iteration-1 => 10 20 30 40 50 60 => no. of comparisons = 5
iteration-2 => 10 20 30 40 50 60 => no. of comparisons = 4
.
.
.
n-1 no. of iterations takes place

Best Case:

iteration-1:

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

if there is no need of swapping for any pair => if all pairs are in order => array is already sorted => no need to goto next iteration

in best only 1 iteration is required, and
total no. of comparisons = n-1
T(n) = O(n – 1)
T(n) = O(n)
Time Complexity = Ω(n)

- time complexities in an ascending order:
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$

3. Insertion Sort:

```
for( i = 1 ; i < SIZE ; i++ ){
        key = arr[ i ];
        j = i-1;

        //if pos is valid && key < arr[ j ]
        while( j >= 0 && key < arr[ j ] ){
                arr[ j+1] = arr[ j ];//shift ele towards its right by 1
                j--;//goto prev element
        }

        //insert key at its appropriate position
        arr[ j+1 ] = key;
}
```

total no. of iterations = n-1
in every iteration max n no. of comparisons takes place
= n(n-1)/2

=> $O(n^2)$

iteration-1:
10 20 30 40 50
=> 10 20 30 40 50 => no. of comparisons = 1

iteration-2:
=> 10 20 30 40 50

=> 10 20 30 40 50 => no. of comparisons = 1

iteration-3:

=> 10 20 30 40 50

=> 10 20 30 40 50 => no. of comparisons = 1

Best Case: if array is already sorted, then in every iteration only 1 comparison takes place, and insertion sort algo takes max (n-1) no. of iterations to sort all array ele’s
total no. of comparisons = 1 * (n-1) = (n-1)
T(n) = O(n-1) => O(n) ==> Ω(n).

Rule: if any algo follows **divide-and-conquer** approach, we get time complexity of that algo in terms of log.

# DS DAY-03:

+ Quick Sort

Worst Case occurs in a quick sort if either array ele's are already sorted or are exists in exactly in a reverse order.

iteration-1: 10 20 30 40 50 60
[LP] 10 [20 30 40 50 60]

iteration-2: [20 30 40 50 60]
[LP] 20 [30 40 50 60]

iteration-3: [30 40 50 60]
[LP] 30 [40 50 60]

int arr[1000];
900 ele's are there
if we want to insert/add an element into an array at 100th location

Q. Why Linked List? to overcome limitations of an array
Linked List must has following 2 features:
1. linked must be dynamic – no. of elements that we can add into it should not be fixed
2. addition & deletion operations should gets performed on linked list efficiently i.e. in O(1) time.

Q. What is a Linked List?
- Linked list is a **basic/linear data structure**, which is **a collection/list of logically related similar type of elements** in which,
- an addr of first element in a list always gets stored into a pointer variable referred as **head,** and each element contains actual data and an addr of (link of) its next element (as well as an addr/link of its previous element).

- in a linked list data structure an element is also called as a **node.**

- there basic 2 types of linked list:
1. singly linked list: it is a type of linked list in which each node contains an addr of its next node only.
(no. of links in each node = 1)
- there are 2 types of singly linked list:
1. singly linear linked list
2. singly circular linked list

2. doubly linked list: it is a type of linked list in which each node contains an addr of its next node as well as an addr of its prev node.
(no. of links in each node = 2)
- there are 2 types of doubly linked list:
1. doubly linear linked list
2. doubly circular linked list

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

if (head == NULL) ==> list is empty
if (head != NULL) ==> list is not empty

each node has 2 part
1. data
2. pointer (next)

```
to store an addr of int var      => int *
to store an addr of char var   => char *
to store an addr of float var   => float *
to store an addr of struct student var => struct student *
```

```
struct node
{
	int data;//4 bytes
	struct node *next;//self referential pointer -> 4 bytes (32-bit compiler)
};
```

sizeof(struct node): 8 bytes (32-bit compiler)

- we can perform 2 basic operations on a linked list data structure
1. addition: to add/insert a node into the linked list
2. deletion: to delete/remove node from the linked list

1. addition: to add/insert a node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at any specific position (in between)

2. deletion: to delete/remove node from the linked list
- we can delete node from the linked list by 3 ways:
i. delete node from the linked list at first position
ii. delete node from the linked list at last position
iii. delete node from the linked list at any specific position

to traverse linked list: to visit each node in the linked list sequentially from first node till max last node.
- we can always start traversal from first node, as we get addr of first node from head.

i. add node into the linked list at last position (slll):
- we can add as many as we want no. of nodes into the slll – dynamic in O(n) time:
best case : Ω(1) – if list is empty
worst case: O(n) – if list is not empty we need to traverse the list till last node
average case: θ(n)

- rule: in a linked list always creates new links (links associated newly created node) first and then only break old links.

ii. add node into the linked list at first position (slll):
- we can add as many as we want no. of nodes into the slll (dynamic) in O(1) time:
best case : Ω(1)
worst case: O(1)
average case: θ(1)

# DS DAY-04:
iii. Add node into the linked list at specific position (in between position): O(n) time.

best case : Ω(1) – if pos == 1
worst case: O(n) – if pos == nodes_cnt + 1
average case: θ(n) – if pos is any between pos

- deletion:
1. delete node from the linked list which is at first pos position: O(1)
best case : Ω(1)
worst case: O(1)
average case: θ(1)

2. delete node from the linked list which is at last pos position: O(n)
best case : Ω(1) – if list contains only one node
worst case: O(n)
average case: θ(n)

3. delete node from the linked list which is at specific pos position: O(n)
best case : Ω(1) – if list contains only one node
worst case: O(n)
average case: θ(n)

+ limitation of slll:
- prev node of any node cannot be accessed from it

SCLL:
if(head == NULL) ==> list is empty
if(head != NULL) ==> list is not empty
if(head == head->next) ==> list contains only one node
if(head != head->next) ==> list contains more one node

- Whichever algo’s we applied on SLLL, all algo’s can be applied exactly as it is on SCLL as well (only we need to take care about next part of last node always).

Add last : O(n)
Add first: O(n)

delete_last( ) : O(n)
delete_first( ) : O(n)

DLLL:
- Whichever algo's we applied on SLLL, all algo's can be applied exactly as it is on DLLL as well (only we need to take care about forward link as well as backward link).

DCLL:
- Whichever algo's we applied on SLLL, all algo's can be applied exactly as it is on DCLL as well (only we need to take care about forward link as well as backward link and we need to maintain prev part of first and next part of last node always).

Linked List ~= DCLL
- DCLL is the most efficient form/type of linked list
- Linked List dynamic
- addition & deletion operations are efficient as it takes O(1) time.

Q. What is the time complexity to add & delete node into the linked list(dcll)?
O(1)