Batch Name	: OC07 Crash Course Batch
Module Name	: Data Structures

DS DAY-01:

- data structure is a programming concept

Q. What is a data structure?
It is a way to store data elements into the memory (i.e. into the main memory) in an oragnized manner, so that operations like addition, deletion, searching, sorting, traversal etc.... can be performed on it efficiently.

We want to store marks of 100 students:
int m1, m2, m3, m4, m5, ..., m100;//400 bytes – sizeof(int): 4 bytes – 32 bit compiler

int marks[100];//400 bytes

+ array: it is it is basic/linar data structure, which a collection/list of logically related similar type of data elements gets stored into the memory at contiguos locations.

- traversal on array => to visit each array element sequentially from first element max till last element.

```
for( index = 0 ; index < SIZE ; index++ ){
        printf("%4d", arr[ index ] );
}
```

- we want to store info about an employee:

empid	: int
emp_name	: char [ ]
salary	: float

+ structure: it is basic/linear data structure, which is a collection/list of logically related similar and disimmilar type of data elements gets stored into the memory collectively as a single record/entity.

- there are 2 types of data structures:
1. linear/basic data structures
2. non-linear/advanced data structures

- to learn data structure is not to learn any programming language, it is nothing but to learn an algorithms in it: operations that can be performed on data elements:

What is an algorithm? Algorithms => Human Beings/End User
- An algorithm is a set of finite no. of instructions written in human understandable language like english, if followed, acomplishesh given task.

Algorithm to do sum of array elements:
step-1: intially take sum as 0
step-2: start traversal of an array and keep adding each array element into the sum sequentially from first element max till last element.
step-3: return final sum

What is a Pseudocode? => special form of an algorithm=> Programmer User
- An algorithm is a set of finite no. of instructions written in human understandable language like english **with some programming constraints**, if followed, acomplishesh given task, this form of an algorithm is referred as a pseudocode.

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

        return sum;
}
```

Program is an implementation of an algorithm
OR
An algorithm is like a blue print of a program

What is a Program?   Program => Machine
- Program is a set of finite no. of instructions written in any programming language given to the machine to do specific task.

```c
#include<stdio.h>

int array_sum( int arr[ ], int n ){
        int sum = 0;
        int index;

        for( index = 0 ; index < n ; index++ ){
                sum += arr[ index ];
        }
        return sum;
}
```

```
int main(void)
{
	int arr[ 5 ] = {10,20,30,40,50};

	printf("sum = %d\n", array_sum(arr, 5);

	return 0;
}
```

- an algorithm is a solution of a given problem
- algorithm = solution
- one problem may has many solutions/algorithms, when one problem has many solutions one need to select an efficient solution/algo.
- to decide efficiency of an algorithm there is a need to do their analysis
- analysis of an algorithm 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 algorithm:
1. **time complexity** of an algorithm is the amount of time i.e. computer time it needs to run to completion.

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

Pune & Mumbai
if multiple paths exists between Pune & Mumbai then one need to select an optimized path
factors to decide optimized paths:
distance in km
time required to cover distance
mode of travelling
traffic conditions
etc....

Client => Manager (Not a Programmer)
Manager => Project Manager(Technical) => Developer(Programmer)
=> Program => Machine

searching: to search given key element in a collection/list of data elements.
- there are two searching algorithms:
1. linear search
2. binary search

1. Linear Search/Sequential Search:
step-1: accept key from the user which is to be search
step-2: start traversal of an array and compare value of key element with each array element sequentially from first element till either match is not found or max till the last element, if key is matches with any array element then return true otherwise return false.

```
Algorithm LinearSearch(A, n, key){
    for( index = 1; index <= n ; index++ ){
        if( key == A[ index ] )//compare value of key with each array ele
            return true;//key is found
    }

    return false;
}
```

best case: if key is found in an array at very first position: O(1)
if size of an array = 10, no. of comparisons = 1
if size of an array = 20, no. of comparisons = 1
if size of an array = 50, no. of comparisons = 1
.
.
if size of an array = n, no. of comparisons = 1

worst case: if either key is found at last pos or key is not found in an array: O(n).
if size of an array = 10, no. of comparisons = 10
if size of an array = 20, no. of comparisons = 20
if size of an array = 50, no. of comparisons = 50
.
.
if size of an array = n, no. of comparisons = n

best case time complexity = if an algo takes min amount of time to run to completion. ==> lower limit

worst case time complexity = if an algo takes max amount of time to run to completion. ==> upper limit

average case time complexity = if an algo takes neither min nor max amount of time to run to completion.

There are 3 notations used to represent time complexities:
1. Big Omega (Ω) - this notation is used to denote best case time complexity asymptotic lower bound.

2. Big Oh (O) - this notation is used to denote worst case time complexity asymptotic upper bound.

3. Big Theta (θ)- this notation is used to denote an average case time complexity
asymptotic tight bound.

Asymptotic analysis: it is a mathematical way to calculate time complexity of an algorithm without implementing it in any programming language.

Descrete Maths:
Rule: if running time of an algo is having any additive /substractive /divisive / multiplicative contant then it can be neglected.
e.g.
O(n + 3) => O(n)
O(n – 5) => O(n)
O(n / 2) => O(n)
O(n * 3) => O(n)

- if any algorithm follows divide-and-conquer approach we get time complexity of that algorithm in terms of log.

2. Binary Search:

after caculating mid pos big size array divided logically into 2 subarrays => left subarray & right subarray
for left subarray, value of left remains as it is, and right = mid-1
for right subarray, value of right remains as it is, and left = mid+1

- binary search algo is also called as logarithmic search/half-interval search

first node/element => root node => root position
node which is not having further child/s => leaf node => leaf position
node which is having further child/s => non-leaf node => non-leaf position

sorting: to arrange data elements in a collection/list of elements either in an ascending order or in a desceding order.

- sort collection/list => bydefault in an ascending order

sorting algorithms:
1. selection sort:

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

=> O((n^2 – n)) / 2) ... by using rule: if an algo is having divisive constant it can be neglected

=> O(n^2 – n)
=> O(n^2 – n) => <u>O(n^2)</u>

rule: if running time of any algo is having a polynomial then in ite time complexity we need to consider only leading term.
e.g.
O(n^3 + n + 2) => O(n^3)
O(n^2 + 5) => O(n^2)

# DS DAY-02:
DAY-01:
- introduction to data structures
- introduction to an algorithm & analysis of an algorithm
- searching algorithms: linear search & binary search
- sorting algorithm: selection sort

2. Bubble Sort:

total no. of comparions = (n-1) + (n-2) + (n-3) + ...
=> n (n – 1) / 2
=> O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

best case : if array elements are already sorted:

10 20 30 40 50 60

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 all pairs are already in order => there is no need of swapping => array ele’s are already sorted => no need to go to next iteration

total no. of comparions = (n-1)
=> $O(n – 1)$
=> $O(n)$ => $\Omega(n)$

```
for( i = 1 ; i < SIZE ; i++ ){//for loop is for iterations

        key = arr[ i ];
        j = i-1;

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

        //insert key into (left hand ele's) at its approriate position
        arr[ j+1 ] = key;
}
```

best case: if array elements are already sorted
iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

iteration-2:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

iteration-3:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

iteration-4:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

iteration-5:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

in best case in every iteration only 1 comparison takes place, and insertion sort requires max (n-1) no. iterations to sort array ele's.
Total no. of comparisons = 1 * (n – 1) = (n – 1) => O(n – 1) => O(n)

=> Ω(n).

int arr[100];

105 ele's
95 ele's – 5*4 bytes - wasted

int arr[ ] ;//not allowed – compile time error
int arr[ ] = { 1, 2, 3, 4 , 5 };

dynamic memory allocation => dynamically allocated block can be used as an array

linked list data structure has beed designed to overcome limitations of an array:
- linked list must be dynamic (no. of ele's in it should gets decided during runtime).
- addition & deletion operations should performed efficiently i.e. in O(1) time.

linked list:

what is a linked list:

linked list: it is a **basic/linear data structure**, which is a **collection/list of finite no. of logically related similar type of data elements** in which:
- an addr of first element always kept into a pointer variable referred as **head**
- each element contains actual data and an addr of (i.e. link) of its next element (as well as an addr its prev element).

- in a linked list elements are linked with each other this link need to maintain by the programmer explicitly.
- in a linked list element is also called as a **node.**

- basically there are two types of linked list:
1. singly linked list: it is type of linked list in which each element/node contains an addr of its next node in a list.
(in each element/node = no. of links = 1)
further there are 2 subtypes of singly linked list:
i. singly linear linked list
ii. singly circular linked list

2. doubly linked list: it is type of linked list in which each element/node contains an addr of its next node as well as an addr of its prev node in a list.
(in each element/node = no. of links = 2)
further there are 2 subtypes of doubly linked list:
i. doubly linear linked list
ii. doubly circular linked list

- total there are 4 types of linked list:
i. singly linear linked list
ii. singly circular linked list
iii. doubly linear linked list
vi. doubly circular linked list

i. singly linear linked list:

if(head == NULL) => list is empty
if(head != NULL) => list is not empty => either list contains only one node OR list may contains more than one nodes.

primitive data types/builtin data type/predefined data types: char, int, float, double, void

non-primitive data types/derived data types: pointer, structure, union, enum etc...

What is NULL?
NULL is a predefined macro whose value is 0 which typecasted into a void *

#define NULL ((void *)0)

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

```
struct node
{
	int data;//4 bytes
	struct node *next;//4 bytes
};
```

sizeof(struct node) = 8 bytes

- sizeof(type *) = 4 bytes on 32-bit compiler
- sizeof(type *) = 8 bytes on 64-bit compiler

- on linked list data structure we can perform basic 2 operations:
1. addition: to add/insert 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 specific position (in between pos)

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 which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at specific position (in between pos).

traversal on linked list: to visit each node in a linked list sequentially from first node max till last node.
- we can always start traversal from first node

i. add node into the linked list at last position (slll):
- we can add as many as we want no. of nodes into slll in **O(n)** time.

Best Case	**: Ω(1) - if list empty**
Worst Case	**: O(n)**
Average Case	**: θ(n)**

ii. add node into the linked list at first position (slll):
- we can add as many as we want no. of nodes into slll in **O(1)** time.

Best Case	**: Ω(1)**
Worst Case	**: O(1)**
Average Case	**: θ(1)**

iii. add node into the linked list at specific position (slll):
- we can add as many as we want no. of nodes into slll in **O(n)** time.

Best Case	**: Ω(1) => if pos == 1**
Worst Case	**: O(n)**
Average Case	**: θ(n)**

# DS DAY-03:
sorting algorithms: bubble sort & insertion sort
limitations of an array data structure
why linked list?

what is a linked list, its types
we can perform basic 2 operations on linked list: addition & deletion

i. delete node from the linked list which is at first position (slll):
we can delete delete node which is at first pos from the slll in O(1) time.

Best Case	: Ω(1)
Worst Case	: O(1)
Average Case	: θ(1)

ii. delete node from the linked list which is at last position (slll):
we can delete delete node which is at last pos from the slll in O(n) time.

Best Case	: Ω(1) => if list contains only one node
Worst Case	: O(n)
Average Case	: θ(n)

iii. delete node from the linked list which is at specific position (slll):
we can delete delete node which is at specific pos from the slll in O(n) time.

Best Case	: Ω(1) => if pos == 1
Worst Case	: O(n)
Average Case	: θ(n)

- all algorithms which we applied on slll, can be applied on scll as it is, only we need to always maintains next part of last node i.e. next part of last node must always points to first node.

- In scll linked list as we need to update/maintained next part of last node always while addlast, addfirst, deletelast & deletefirst, we have to traverse list till last node and hence all operations in scll takes O(n) time.

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

DLLL:

```
struct node
{
	struct node *prev;//4 bytes
	int data;//4 bytes
	struct node *next;//4 bytes
};
```

sizeof(struct node) = 12 bytes

- all algorithms which we applied on slll, can be applied on dlll as it is, only we need to take care about forward link as well as backward of each node i.e. we need to maintains prev part & next part of each node in all operations.

DCLL is the most efficient form of a linked list
linked list => DCLL

+ Stack: it is a **basic/linear data structure**, which is a **collection/list of logically related similar type of data elements** in which we can add as well as delete elements only from one end referred as **top end.**
- in this list, **element which was inserted last can only be deleted first**, so this list works in **last in first out** / **first in last out** manner, and hence stack is also called as **LIFO list / FILO list.**
- we can perform basic 3 operations on stack data structure in O(1) time:
1. Push : to insert/add an element onto the stack from top end
2. Pop   : to delete/remove an element from the stack which is at top end
3. Peek : to get the value of an element which is at top end (without push/pop)

- stack can be implemented by 2 ways:
1. static implementation of stack (by using an array)
2. dynamic implementation of stack (by using an linked list)

1. static implementation of stack (by using an array): static stack

```
#define SIZE 5

typedef struct
{
        int arr[ SIZE ];
        int top;
}stack_t;

stack_t s;
```

primitive data types/predefined/built in : char, int, float, double & void
non-primitive data types/derived : pointer, structure, union, array, enum etc...

arr : int [ ] - non-primitive data type
top : int - primitive data type

1. Push : to insert/add an element onto the stack from top end
step-1: check stack is not full (if stack is not full then only we can push ele into it).
step-2: increment the value of top by 1
step-3: insert an ele onto the stack at top end

2. Pop   : to delete/remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop ele from it).
Step-2: decrement the value of top by 1 [i.e. we are deleting an element from the stack].

3. Peek : to get the value of an element which is at top end (without push/pop)
step-1: check stack is not empty (if stack is not empty then only we can peek ele from it).
step-2: get the value of an element which is at top end (without increment / decrement top).

2. dynamic stack (linked list)

push : add_last( )
pop : delete_last( )

list is empty => stack is empty => last in first out manner => stack
head -> 10 20

OR

push : add_first( )
pop : delete_first( )

- in any program where collection/list of data elements should works in last in first out manner – stack can be used.
- stack data structure is used to implement expression conversion & expression evaluation algorithms:

Q. What is an expression?
An expression is a collection of operands & operators
- there are 3 types of expression:

1. infix expression	: a+b
2. prefix expression	: +ab
3. postfix expression	: ab+

# DS DAY-04:

- algo to convert infix expression to postfix
- algo to convert infix expression to prefix
- algo to convert prefix expression to postfix
- algo to evaluate psotfix evalution

+ Queue: it is a **basic/linear data structure**, which is a **collection/list of logically related similar type of data elements** in which, we can insert elements into it from one end referred **rear end**, whereas elements can be deleted from another end referred as **front end.**
- in this list **element which was inserted first can be deleted first**, so this list works **first in first out manner / last in last out** manner, hence queue is also called as **fifo list / lilo list.**
- we can perform basic 2 operations on queue data structures in O(1) time:
1. enqueue: to insert an element into the queue from rear end
2. dequeue: to delete an element from dequeue which is at front end

- there are 4 types of queue:
1. linear queue (fifo)

2. circular queue (fifo)

3. priority queue: it is a type of queue in which elements can be added into it from rear end randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

4. double ended queue (deque): it is a type of queue in which elements can be added as well deleted from both the ends.

1. linear queue (fifo):

it can be implemeneted by 2 ways:
static implementation of a linear queue (by using an array)
dynamic implementation of a linear queue (by using linked list)

static implementation of a linear queue (by using an array):

```
struct queue
{
        int arr[ 5 ];
        int rear;
        int front;
};
```

```
1. enqueue: to insert an element into the queue from rear end
step-1: check queue is not full (we can insert an ele into the queue only if it is not full).
step-2: increment the value of rear by 1
step-3: insert ele into the queue from rear end
step-4: if( front == -1 )
            front = 0
```

```
2. dequeue: to delete an element from dequeue which is at front end
step-1: check queue is not empty (we can delete ele from the queue only if it is not empty).
step-2: increment the value of front by 1 [ i.e. we are deleting an element from the queue ].
```

Circular Queue:

rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2
rear = 2, front = 3
rear = 3, front = 4

if front is at next pos of rear => cir q is full
front == rear + 1

0 == 4+1
0 == 5 => LHS != RHS

front == (rear + 1)%SIZE

for rear=0, front=1, front is at next pos rear => cir q is full
front == (rear + 1)%SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for rear=1, front=2, front is at next pos rear => cir q is full
front == (rear + 1)%SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

for rear=2, front=3, front is at next pos rear => cir q is full
front == (rear + 1)%SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir q is full

for rear=3, front=4, front is at next pos rear => cir q is full
front == (rear + 1)%SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir q is full

for rear=4, front=0, front is at next pos rear => cir q is full
front == (rear + 1)%SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full

increment => rear++;//rear = rear + 1;

to increment the value of rear by 1 [it should keep rotating in between : 0 TO SIZE-1].

rear = (rear+1)%SIZE

rear=0 => rear = (rear+1)%SIZE => (0+1)%5 = 1%5 = 1
rear=1 => rear = (rear+1)%SIZE => (1+1)%5 = 2%5 = 2
rear=2 => rear = (rear+1)%SIZE => (2+1)%5 = 3%5 = 3
rear=3 => rear = (rear+1)%SIZE => (3+1)%5 = 4%5 = 4
rear=4 => rear = (rear+1)%SIZE => (4+1)%5 = 5%5 = 0

+ dynamic queue (linked list: dcll):
enqueue : add_last( )
dequeue : delete_first( )

list is empty => queue is empty

FIFO

head => 40 50

OR
enqueue : add_first( )
dequeue : delete_last( )

head => 55 44

DFS (Depth First Search) Traversal : Stack
BFS (Breadth First Search) Traversal : Queue

+ introduction to an advanced data structure:
tree:
root element/node: first specially designated ele in a tree
parent node/father
child node/son
grand parent/grand father
grand child/grand son
siblings: child nodes of same parent
ancestors: all the nodes which are in the path from root node to that node are called as its ancestors.
- root node is an ancestor for all the nodes

descendents: all nodes which can be accessible from it
- all the nodes are descedents of root node

- depth of a tree = max level of any node in a given tree

- tree – by design tree is a dynamic data structure
as tree is dynamic – it can grow upto any level and any node can have any no. of child nodes, and due to operations like addition, deletion, searching etc... becomes inefficient, and hence restrictions can be applied on tree data structure due to it there are diff types of tree:

- binary tree: it is a tree in which each node can have max 2 child nodes
i.e. each node can have either 0 OR 1 OR 2 no. of child nodes.
OR tree in which degree of each node is either 0 OR 1 OR 2.

- set which contains 0 no. of ele’s => empty set/null set
- set which contains only 1 ele => singleton set
- set which contains more than 1 ele

- binary search tree: it is a binary tree in which left child is always smaller thaan its parent and right child is always greater or equal to parent.
- binary search tree BST, basically designed to achieve operations like addition, deletion and searching in O(log n) time.

- there are 2 tree traversal methods:
1. bfs (breadth first search) traversal:
- it is also called level wise traversal,
- in this traversal method, traversal starts from root node and all the nodes gets visited level wise from left to right.

2. dfs (depth first search) traversal

V – Visit
L – Left Subtree
R – Right Subtree

under dfs, further there are 3 ways:

1. inorder	: L V R
2. preorder	: V L R
3. postorder	: L R V

graph:

google map:

city = vertex = 100 cities = 100 vertices
path between cities => edges
info about cities => vertices
info about paths between cities => edges

graph

if vertices are adjancent entry between them in matrix = 1
if vertices are not adjancent entry between them in matrix = 0

CCAT => Min DS => Notes + PPTs + Black Book

doubts = whatsapp group / zoom channel