Batch Name : PreCAT OM19
Subject : Data Structures

DS DAY-01:

- to store marks of 100 students
int m1, m2, m3, m4, ..., m100;//400 bytes

i want to sort marks in a descending

int marks[100];//400 bytes
operations can be performed on data ele's efficiently.

Array: it is a collection/list of logically related similar type of elements in which data ele's gets stored into the memory at contiguos location.

We want to store rollno, name & marks of student/record

Structure: it is a collection/list of logically related similar and disimmilar type of data elements in which data ele's gets stored into the memory collectively as a sinlge entity(record).

Class: it is a basic/linear data structure which is a collection of logically related similar and disimmiar type of data elements as well as functions.

typedef struct employee emp_t;

struct empolyee e1;//abstraction – abtract data type
emp_t e2;

```
class student
{
	//data  members
	private:
		int rollno;
		String name;
		float marks;
	public:
	//member functions
		student();
		~student();
		//getter functions
		//setter functions
		//facilitators
};
```

student s1;//abstract data type - abstraction

student s2;- reusability

struct employee e1;

to scan an array/traversal on an array: to visit each array element sequentially from first element till max last element.

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

Q. What is a Program?
Program is a finite set of instructions written in any programming language (like C, C++, Java, Python, Assembly etc....), given to the machine to do specific task.

Program --> Machine

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.

Algorithm --> Programmer User

- A Program is an implementation of an algorithm.
- An algorithm is a like blueprint of a program.

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 algorithm is referred as a pseudocode.

Algorithm to do sum of array elements: --> End User/Programmer User
Step-1: initially take value of sum variable as 0.
Step-2: start traversal of an array and keep adding each element into the sum variable one by one.
Step-3: return the final value of sum.

Pseudocode: its a special form of an algorithm --> Programmer User

```
Algorithm ArraySum( A, size ){
	sum = 0;
	for( index = 1; index <= size ; index++ ){
		sum += A[ index ];
	}

	return sum;
}
```

Program: --> Machine

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

Code <==> Program
Source Code – program written in any programming language.

- An algorithm is a solution of a given problem.
- algorithm = solution
- we can have many solutions for a the same problem, in this case one need to select an efficient solution/algo.

e.g.
searching: to search a given key element in a collection/list of data elements.
1. linear search
2. binary search

sorting: to arrange data elements in a collection/list of data elements either in an ascending order or in a descending order.
1. selection sort
2. bubble sort
3. insertion sort
4. quick sort
5. merge sort

Pune --> Mumbai
- multiple paths/routes may exists between two cities
- when we know multiple paths between 2 cities --> an optimized/efficient one
parameters/measures: distance, cost, status, traffic situation, time .....

searching:
+ Analysis of an algorithm:
- to decide efficiency of an algo's, we need to do their analsis.
- analysis of an algo, is nothing but to calculate how much time i.e. computer time and space i.e. memory it needs to run to completion.
- there are 2 measures of analysis of 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.

- space ==> memory required to store variables, constants & instructions in a program/algo.

- Linear search:
step-1: accept value of key element (which is to search) from user
step-2: start traversal an array from first element and compare value of key with each array ele sequentially till match is found or max till the last element. If match found the n return true, otherwise return false.

Best case: If key is found at first position --> O(1) – constant time complexity
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 position or key is not exists --> 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

- We need to follow certains rules and we have to use some notations:

Asymptotic Notations:
1. Big Omega (Ω) - this notation is used
2. Big Oh (O)
3. Big Theta (θ)

- descrete mathemetics
Rule:
1. 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(n/2) => O(n)
O(n*2) => O(n)

- if it is not mentioned specifically then (bydefault) we have to consider an average case time complexity for all algorithms.
- usually magnitudes of time complexities of an algo's are same in average case and worst case.

Q. What is the time complexity of a linear search?
Average Case = θ(n)

Q. What is the best case time complexity of a linear search?
Best Case = Ω(1)

Q. What is the average case time complexity of a linear search?
Average Case = θ(n)

space =

Prerequisite C Programming Language Topics: do revise following topics
1. storage classes
2. pointers
3. functions
4. structure

DS DAY-02:
- introduction to DS:
Why there is a need of data structure?
What is data structure? Types of data structure
- introduction to an algorithm, analysis of an algo:
- linear search:
best case : Ω(1)
worst case : O(n)
average case : θ(n)

binary search:
- by menas of calculating mid pos, big size array has been divided into two subarrays – left subarray & right subarray.

For left subarray value of left remains as it is, right = mid-1
For right subarray value of right remains as it is, left = mid+1

subarray is valid till (left <= right)
subarray becomes invalid as soon as left > right

if size of an array 1000
[0.....1000] ==> [0...499] 500 [501 .... 1000]

LSA => [0 ... 499] ==> [0.....249] 250 [251 ..... 500]
LSA => [0... 249] ==> [0...124] 125 [126.....249]

after iteration-1: n/2
after iteration-2: n/4
after iteration-3: n/8

after every iteration search space is getting reduced by half

for size of an array = n
for ietration search space = n
after iteration-1: $n/2 + 1 \Rightarrow n/2^1 + 1$
after ietartion-2: $n/4 + 2 \Rightarrow n/2^2 + 2$
after ietartion-3: $n/8 + 3 \Rightarrow n/2^3 + 3$
.
.
after k iterations: $n/2^k + k$

lets assume, $n = 2^k$
$\Rightarrow \log n = \log 2^k$ (by taking log on both sides)
$\Rightarrow \log n = k \log 2$
$\Rightarrow \log n = k$ (log 2 = 1)
$\Rightarrow k = \log n$

$T(n) = n/2^k + k$
put $n = 2^k$ in above equation, $k = \log n$
$T(n) = 2^k/2^k + \log n$
$T(n) = 1 + \log n$
$T(n) = O(1 + \log n)$
$T(n) = O(\log n + 1) \Rightarrow O(\log n)$

- algorithm which follows divide-and-conquer approach, we get time complexity in terms of log

- when we compare two algo's for deciding efficiency we need to consider an average case time complexities.

Rule:
if running time of an algo is having a polynomial, then only leading term gets considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^4 + n^2 + 2) \Rightarrow O(n^4)$

1. selction sort:

Total no. of comparisons = $n(n-1)/2 \Rightarrow (n^2 - n)/2$

$T(n) \Rightarrow O(n^2 - n/2) \Rightarrow O(n^2 - n) \Rightarrow$ **$O(n^2)$**

- C programming language: procedure oriented programming language
procedure/function: in C we need to divide logic of a program into functions

- C++ programming language: object oriented programming language

a = 20
b = 10

temp=10

temp = a;
a = b;
b = temp;

- bubble sort is also called as sinking sort

home work: implement bubble sort

for it = 0 => pos = 0,1,2,3,4
for it = 1 => pos = 0,1,2,3
for it = 2 => pos = 0,1,2
for it = 3 => pos = 0,1

for(pos = 0 ; pos < SIZE-1-it ; pos++)

if array is already sorted:

flag = 0

10 20 30 40 50 60 -> all pairs are in order -> array ele's are already sorted

best case: if array ele's are already sorted
total no. of comparisons = n-1
T(n) = O(n-1) => **O(n)**

DS DAY-03:
- binary search: algorithm, analysis & implementation
- sorting algo's:
selection sort:
bubble sort

3. Insertion Sort:

```
for( i = 1 ; i < SIZE ; i++ ){
	key = arr[ i ];
	j = i-1;
	while( j >= 0 && key < arr[ j ] ){
		arr[ j+1 ] = arr[ j ];//shift ele towards its right
		j--;
	}
	arr[ j+1 ] = key;
}
```

best case in insertion sort:
10 20 30 40 50 60 70

- in every iteration only 1 comparison takes place, and in insertion sort max n-1 no. of iterations are there
total no. of comparisons = n-1
T(n) = O(n-1) => O(n) => Ω(n).

- in any sorting algo, if relative order of two ele's having same key value remains same even after sorting then such sorting algo is referred as stable.

10 10' 20 30 40

Key = 10'

arr[1] = 10
arr[4] = 10

- to convert program into a menu driven program

10 20 30 40 50 60 70 80
[LP] 10 [20 30 40 50 60 70 80]
[LP] 20 [30 40 50 60 70 80]
[LP] 30 [40 50 60 70 80]

as in above case in every pass array is not getting divided equally into partitions and hence time complexity in this case do not get in terms of log n **worst case – $O(n^2)$.**

 # DS DAY-04
- Merge Sort is not inplace on an array, but if we apply merge sort on linked list it is inplace.

- Array
+ lmitations of array data structure:
1. in an array we can collect/combine only logically related similar type of elements – to overcome this limitation structure data structure has been designed.
2. **array is static** i.e. size of an array cannot grow or shrink during runtime.
int arr[100];

3. **addition and deletion operations on an array are not efficient as it takes O(n) time.**
int arr[100];

90 ele's
if we want to add/insert an ele into an array at 10^{th} position – we need to shift all ele's from cur 10^{th} pos till 90^{th} pos towards its right by one one position whereas while deletion we need to shift ele's towards left by one one position

Q. Why Linked List?
- to overcome limitations of an array linked list data structure has been designed.
- linked list must be dynamic and addition & deletion operations should gets peeformed efficiently.

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

- 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 a type of linked list in which each node contains an addr of its next node.
- there are two type 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 previous node.
- there are two type of doubly linked list:
    1. doubly linear linked list
    2. doubly circular linked list

- total there 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: it is a type of linked list in which
- head always contains an addr of first element/node, if list is not empty
- each node has two parts:
i. data part: it contains actual data of any primitive/non-primitive type
ii. pointer part (next): it contains an addr of its next node
- last node points to NULL, i.e. next part of last node contains NULL.

primitive: int, char, float, double, void
non-primitive: array, structure, union, pointer, function, enum ....
- we can maintained linked list of any type of elements

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

#define NULL ((void *)0)

'\0' - null character
it is an escape sequence char whose ascii value is 0.

each node has two parts:

1. data : int
2. next : ?? *

- to store an addr of int var -> (int *)
- to store an addr of char var -> (char *)
.
.
- to store an addr of struct type var -> (struct type *)

```
struct node
{
	int data;
	struct node *next;//self referential pointer
};
```

- when we define structure, we defined data type – derived data type
- we can perform basic 2 operations on linked list data structure:

1. addition : to add/insert node into the linked list
- we can add node into the linked list by 3 ways

1. add node into the linked list at last position
2. add node into the linked list at first position
3. add node into the linked list at specific (in between) position

2. deletion : to delete/remove node from the linked list
- we can delete node from the linked list by 3 ways

1. delete node from the linked list which is at first position
2. delete node from the linked list which is at last position
3. delete node from the linked list which is at specific (in between) position

1. add node into the singly linear linked list at last position: O(n)
- we can add as many as we want no. of nodes into the linked list at last pos in O(n) time.

Best Case: Ω(1) – if list is empty
Worst Case: O(n)
Average Case: θ(n).

2. add node into the singly linear linked list at first position:
- we can add as many as we want no. of nodes into the linked list at first pos in O(1) time.

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

rule: make before break i.e. always creates a new links (links associated with newly created node) and then break old links.

exit value:
0 : successfull termination
>0 (1) : erroneuos termination – due an error
<0 (-1) : abnormal termination – e.g. divide-by-zero error

- traversal of a linked list: to visit each node in a linked list sequentially from first node max till last node.

DS DAY-05:

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

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

- whatever algo's/operations i.e. addition & deletion we performed on slll all operations/algo's can be performed on scll exactly as it is, just we need to take care about next part of last node.

- whatever algo's/operations i.e. addition & deletion we performed on slll all operations/algo's can be performed on dlll exactly as it is, just we need to take care about forward link (next) as well as backward link (prev).

Linked List = Doubly Circular Linked List

+ Stack: it is a basic/linear data structure, which is a collection/list of logically related similar type of elements, into which elements can be added as well as deleted from only 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 this list is also called as **LIFO/FILO list.**

- On stack data structure we can perform basic 3 operations 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 pos
3. Peek: to get the value of topmost element (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 linked list)

- stack can be considered as an **adaptive data structure**, as adopts faetures of data structure by using which we implement it.

 # DS DAY-05

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

```
struct stack
{
	int arr[ 5 ];
	int top;
};
```

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

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 onto the stack)
step-2: increment the value of top by 1
step-3: push/insert an element onto the stack at top position

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

3. Peek: to get the value of topmost element (without Push/Pop).
step-1: check stack is not empty (if stack is not empty then only we can peek element from the stack).
step-2: return the value of an element which is at top position (without increment/decrement of top).

2. dynamic implementation of stack (by using linked list)

Push : add_last( )
Pop : delete_last( )

OR

Push : add_first( )
Pop : delete_first( )

head -> 33 22 11

dynamic stack empty – list is empty
dynamic stack full – there is no stack full condition

+ applications of stack data structure:

+ stack application algorithms:
1. conversion of given infix expression into its eq postfix expression
2. conversion of given infix expression into its eq prefix expression
3. conversion of given prefix expression into its eq postfix expression
4. postfix evalution

Q. what is an expression?
An expression is combination of an operands and opeartors
- there are 3 types of expression:
1. infix expression : a+b
2. postfix expression : ab+
3. prefix expression : +ab

DS DAY-07

Stack:
- concept & definition
- push, pop & peek – O(1) time
- implementation of stack by using an array as well as linked list
- stack application algo's:
1. to convert given infix expression into its eq postfix
2. to convert given infix expression into its eq prefix
3. to convert given parenthesized infix expression into its eq postfix
4. to convert given prefix expression into its eq postfix
5. to evaluate postfix expression

+ Queue: it is a basic/linear data structure which is a collection/list of logically related similar type of data elements in which elements can be added from one end referred as rear end, and elements can be deleted from another end referred as front end.

In this list, element which was inserted first can be deleted first, this list works in first in first out manner/last in last out manner, and hence it list is also called as FIFO List/LILO List.
- on queue data structure basic 2 operations can be performed in O(1) time:
1. enqueue: to insert/add/push an element into the queue from rear end
2. dequeue: to delete/remove/pop an element from the queue 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 randomly (i.e. without checking priority) from rear end, 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 as deleted from both the ends.
- basic 4 operations can be performed on deque:
i. push_back : add_last( )
ii. push_front : add_first( )
iii. pop_back : delete_last( )
iv. pop_front : delete_first( )
- deque can be implemented by using dcll/dlll (with head & tail pointer)
- there are further 2 types of deque:
1. input restricted deque: it is a type of deque in which elements can be added only from one end, and elements can be deleted from both the ends.

2. output restricted deque: it is a type of deque in which elements can be added from both the ends, but elements can be deleted only from one end.

1. implementation of linear queue (fifo) (by using an array):

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

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

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

Limitation of a linear queue
vacant places cannot be reutilized – memory wastate is there and hence to overcome this limitation circular queue has been designed

2. circular queue (fifo)

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

if front is at next position of rear --> cir queue full

queue full : front == (rear + 1)%SIZE

for rear=0, front=1, front is at next pos of rear – cir queue is full
=> front == (rear + 1)%SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS ==> Cir Queue is Full

for rear=1, front=2, front is at next pos of rear – cir queue is full
=> front == (rear + 1)%SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS ==> Cir Queue is Full

for rear=2, front=3, front is at next pos of rear – cir queue is full
=> front == (rear + 1)%SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS ==> Cir Queue is Full

for rear=3, front=4, front is at next pos of rear – cir queue is full
=> front == (rear + 1)%SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS ==> Cir Queue is Full

for rear=4, front=0, front is at next pos of rear – cir queue is full
=> front == (rear + 1)%SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS ==> Cir Queue is Full

rear++;
=> rear = rear + 1

rear = (rear+1)%SIZE

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

2. implementation dynamic queue (by using linked list)
enqueue : add_last( )
dequeue : delete_first( )

head -> 30 40 50

OR

enqueue : add_first( )
dequeue : delete_last( )

dfs traversal in a tree & graph : stack
bfs traversal in a tree & graph : queue

FCFS (First Come First Served) CPU Scheduling Algorithm

tree terminolgies:
* root node
* 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
- root node is an ascenstor of all the nodes
* descendents: all the nodes which can be accessible from it
- all the nodes are descendents of root node
* degree of a node = no. of child node/s having it
* degree of a tree = max degree of any node in a given tree
* leaf node/terminal node/external node: node which is not having any number child nodes

OR node having degree 0

*** non-leaf node/non-terminal/internal node:** node which is having any number child node/s
OR node having non-zero degree.

*** level of any node = level of its parent node + 1**
- if we assume level of root node = 0
*** level of a tree = max level in a given tree**
depth of a tree = max level in a given tree