Data Structures:
- Section B: 7 Questions
- All questions are concept/algorithm oriented.

From Monday:
Lecture Time: 4 TO 7
4 TO 6:30 : Lecture Session: First 10 mins quiz
6:30 TO 7:00 : Lab Session & Doubts solving.

==========

Q. Why is there is need of data structure?

Q. What is Computer?
- Computer is a machine/hardware/digital device, can be used to do differnt tasks efficiently and accurately.
- there are four basic functions of computer:
1. data storage: secondary storage devices
2. data processing: main memory
3. data movement:
4. control:

program.c - HDD
$./program.out -> main memory

int m1, m2, m3, ..., m100; -> yes
4*100 = 400 bytes

sort all marks in descending order

int marks[100]; -> 400 bytes

- there is a need of data structure to achieve three things in programming:
1. efficiency
2. abstraction
3. reusability

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

- there are two types of data structure:
1. "linear data structure"/"basic data structure": data ele's gets stored into the memory in a linear manner and hence can be accessed in a linear manner.
- array
- structure
- union
- class

- linked list
- stack
- queue

2. "non-linear data structure"/"advanced data structure": data ele's gets stored into the memory in a non-linear manner and hence can be accessed in a non-linear manner.
- tree (heirarchical manner)
- graph
- hash table (associative manner)
- binary heap

etc...

+ "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 memory locations.

+ "structure": it is a collection/list of logically related similar and disimilar type of ele's gets stored into the memory collectively i.e. as a single entity/record.

+ "union": it is a collection/list of logically related similar and disimilar type of ele's gets stored into the memory collectively i.e. as a single entity/record, but memory gets shared among all its members. (for effective memory utilization instead of structure union data structure can be used in a programming).

+ "class": it is a collection/list of logically related similar and disimilar type of data elements as well as functions which performs operations on that data ele's, i.e. inside class we can combine data members as well as member functions.

```
struct sample
{
        int n1;
        int n2;
        int n3;
        int n4;
};
```

sizeof(struct sample): 16 bytes
- we know in advanced that, out of 4 members we will going to use only one member at a time.

```
int marks[100];
```

- information of 100 students:
roll_no: int
name : char []

marks : int

```
struct student
{
	int roll_no;//4
	char name[32];//32
	int marks;//4
};
```

```
struct student s;//40 bytes
struct student s1; abstract data type

<data type> <var_name>;
int n1;
char ch;
struct student s3;

struct employee e1;
```

```
class student
{
	private:
		int roll_no;
		char name[32];
		int marks;
	public:
		student();//initialization
		~student();
		accept_student_record();
		display_student_record();
		..
		..
};
```

```
student s1;
student s2;
student s[10];
```

- "reusability"

```
int factorial(int n)
{
	...
```

```
        ...
}


int main(void)
{
        res = factorial(5);
        res = factorial(10);



}
```

Q. What is an algorithm?
- an algorithm is a set of finite no. of instructions written in human understandable language (like an english), if followed, acomplishses given task.

- an algorithm is a set of finite no. of instructions written in human understandable language (like an english) with some programming contraints, if followed, acomplishes given task, such algo is referred as "pseudocode".

Q. What is Program?
- set of instructions written in any programming language (with its own syntax) given to the machine to do specific task.

- Algorithm is a blue print and Program is nothing but an implementation of an algorithm.

```
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;
}
```

- an algorithm is a nothing but a solution of a given problem.
- algorithm = solution
- Problem: Sorting - to arrange data ele's in a collection/list of ele'e either in an ascending order or in a descending order.
- there are many sorting algorithms:
    1. selection sort
    2. bubble sort
    3. insertion sort
    4. quick sort
    5. merge sort

6. heap sort
7. radix sort
8. shell sort
etc....

- one problem may has many solutions
- when one problem has many solutions/algo's, we need to select an efficient solution out of it, and to decide which algo/solution is an effcient one, there is 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 memmory it needs to run to completion.
- there are two measures of analysis of an algorithm:
1. "time complexity": time complexity of an algo is the amount of computer time it needs to run to completion.

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

+ Array:
# Searching Algorithms:
+ Searching: to search/find given key element in a collection/list of data ele's

1. Linear Search: In this algorithm key element gets compared with each element in a list/collection of elements sequentially from the first element either match is not found or maximum till last element, if match is found it returns true and if match not found it returns false.

```
Algorithm LinearSearch(A, n, key)//A is an array of size "n" & key to be search
{
        for( index = 1 ; index <= n ; index++ )
        {
                if( key == A[ index ] )
                        return true;
        }

        return false;
}
```

best case : O(1) : Big Omega(1)
worst case : O(n) : Big Oh(n)
average case : O(n/2) : Big Theta(n)

+ "asymptotic analysis:" it is a mathematical way to calculate time complexity and space complexity of an algorithm without implementing it in any programming language.
- e.g. in linear search algo, sorting algo we will going to do analysis depends on no. of comparisons.

- there are three asymptotic notations can be used to represent magnitudes of time complexity & space complexity:
1. Big Omega ( ): best case time complexity
2. Big Oh (O) : worst case time complexity
3. Big Theta ( ): average case time complexity

- if running time of an algo is having any additive/substractive/multiplicative or divisive constant then it can be neglected.
e.g.

O(n+2) ==> O(n)
O(n-1) ==> O(n)
O(2*n)==> O(n)
O(n/3) ==> O(n)

2. Binary Search:

- this algo works on "divide-and-conquer" stratergy
- for this algorithm, prerequisite is that array ele's must be already sorted (ascending order).

step1: consider first element in an array/subarray is at "left" position and the last element is at "right" position, then calculate "mid" position by the formula, mid = (left+right)/2 and by means of calculating mid position we are dividing big size array logically into two subarrays:
1. left subarray : from left position till mid-1, and
2. right subarray : from mid+1 till right position.
compare key element with an element which is at mid position, if key element matches with mid position element then return true, if in very first iteration only by doing one comparison, then it is considered as the best case in a binary search and hence running time of an algorithm in this case is O(1).

step2: if the key element does not matches with mid position element then either key may exists in left subarray or it may exists into the right subarray and hence we need to search key element either into left subarray or into right subarray by skipping whole left subarray or right subarray:

```
if( key < A[ mid ] )
{
        search element into only in a left subarray by skipping whole right
        subarray
}
else
{
        search element into only in a right subarray by skipping whole left
        subarray
}
```

step3: repeat step1 & step2 till subarray is valid

```
Algorithm BinarySearch(A, size, key)
{
        left = 0;
```

```
        right = size;

        //while array or subarray is valid
        while( left <= right )
        {
            mid = (left+right)/2;

            if( key == A[ mid ] )
                return true;


            if( key < A[ mid ] )
                right = mid-1;
            else
                left = mid+1;
        }

        return false;
    }
```

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

for left=0 & right = 9 => mid = (0+9)/2 = 9/2 = 4
for left=5 & right = 9 => mid = (5+9)/2 = 14/2 = 7
for left=8 & right = 9 => mid = (8+9)/2 = 17/2 = 8
for left=9 & right = 9 => mid = (9+9)/2 = 18/2 = 9

best case : O(1) : Big Omega(1)
worst case : O(log n) : Big Oh(log n)
average case: O(log n) : Big Theta(log n)

- when we compare two algo, i.e. while doing comparison and decide efficiency we must considered their worst case time complexities.

O(n) > O(log n)

O(1) < O(log n) < O(n)

- usually, magnitudes of time complexities in an average case and worst case is same.

+ Sorting Algorithms:
Algorithm & Implementation
1. "Selection Sort":
    - in this algorithm, in first iteration the first position gets selected and

element which is at selected position gets compared with remaining all its next position elements, if in any comparison element which is at selected position is found greater than element at any other position then they gets swapped, and by this way in first iteration the smallest element in an array gets settled at the first position
- in second iteration second position gets selected and element which is at selected position gets compared with all its next position elements and as per condition mentioned above and second smallest element gets settled at second position and so on, and hence in max (n-1) no. of iterations all elements in an array gets arranged in a sorted manner.

Best case : O(n^2)
Worst case : O(n^2)
Average case: O(n^2)

2. "Bubble Sort":
- in this algorithm elements which are at two consecutive positions gets compared, if they are not in order (i.e. if prev position element is greater than next position element) then swapping takes place otherwise no need of swapping.
- by applying above logic, in the first iteration largest element gets settled at last position, in second iteration second largest element gets settled at second last position and so on, in max (n-1) no. of iterations all array elements gets arranged in a sorted manner in an ascending order.

10 20 30 40 50 60

Best Case : Big Omega(n) -> it occures only if all array ele's are already sorted
Worst Case : Big Oh(n^2)
Average Case: Big Theta(n^2)

- this algo is also called as "sinking sort".
- selection sort & bubble sort algo's are simple but these algo's are not efficient for larger input size array.

3. Insertion Sort:

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

        while( j >= 0 && arr[j] > key )
        {
                arr[j+1] = arr[j];
                j--;
        }
```

```
        arr[j+1] = key;
}
```

Best Case : Big Omega(n) -> it occures only if all array ele's are already sorted
Worst Case : Big Oh(n^2)
Average Case: Big Theta(n^2)

- insertion sort algorithm is an efficient for already sorted input sequence by design, and hence insertion is an efficient sorting algo for smaller input size array (if size of an array is smaller then insertion sort is efficient than quick sort as well).

Only Algorithm:
4. Quick sort:

partitioning:
- we need to select pivot element
- we need to shift ele's which are smaller than pivot towards its left side as possible, and ele's which are greater than pivot need to shift towards as right as possible
- ele's which are at left side of pivot will be referred as "left partition", whereas ele's which are at right side of pivot will be referred as "right partition", and pivot ele gets settled at its appropriate position

- apply partitioning further on left partition as well on right partition, till size of subarray is greater than 1.

Best Case : Big Omega(n (log n))
Worst Case : Big Oh(n^2)
Average Case : Big Theta(n (log n))

```
pivot = arr[left];

while( i <= j )
{
        while( i < right && arr[i] <= pivot )
                i++;

        while( arr[j] > pivot )
                j--;

        if( i <= j )
                SWAP(arr[i], arr[j]);
}

SWAP(arr[j], arr[left]);
```

5. Merge Sort

Best Case : Big Omega(n (log n))
Worst Case : Big Oh(n (log n))
Average Case : Big Theta(n (log n))

O(1) < O(log n) < O(n) < O(n log n) < O(n^2)

==========

 # DAY-04:

+ Limitations of an array data structure:

- array is "static" i.e. size of an array cannot be either shrink or grow during runtime.
- addition & deletion operations on an array are not efficient as these operations takes O(n) time.
- and hence to overcome these limitations of array data structure "linked list" data structure has been designed.

+ "linked list": it is a collection/list of logically related similar type of elements
in which:
- address of first element always gets stored into a pointer variable referred as "head" pointer, and
- each element contains data (of any primitive/non-primitive type) and contains addr of its next (as well as prev) element in that list.

- 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 addr of its next node in a list.

i. singly linear linked list
ii.singly circular linked list

2. doubly linked list: it is a type of linked list in which each node contains addr of its next node as well addr of its prev node in that list.

i. doubly linear linked list
ii.doubly circular linked list

---

i. singly linear linked list
ii. singly circular linked list
iii. doubly linear linked list
iv. doubly circular linked list

i. "singly linear linked list": it is a linked list in which
- head always contains addr of first node, if list is not empty
- each node has two parts:
1. data part: contains actual data of any primitive/non-primitive type
2. pointer part(next): contains addr of its next node
- last node's next part points to NULL.

* "traversal" of a linked list: to visit each node in a list atmost once sequentially

from first node till last node.
* "traversal" on an array -- to scan array ele's sequentially from first position to last position.

- basically we can perform addition and deletion operations onto the linked list
1. addition: we can add node into the singly linear linked list by 3 ways:
    1. add node into the list at last position
    2. add node into the list at first position
    3. add node into the list at specific position

1. add node into the list at last position:

sizeof(int *) = 4 bytes	scale factor(int *): 4 bytes
sizeof(char *) = 4 bytes	scale factor(char *): 1 byte
sizeof(double *) = 4 bytes	scale factor(float *): 4 bytes
sizeof(float *) = 4 bytes	scale factor(double *): 8 bytes
sizeof(node_t *) = 4 bytes	scale factor(node_t *): 8 bytes

sizeof(type *) = 4 bytes - 32 bit compiler
type may be any primitive/non-primitive