Batch Name : PreCAT OM25
Subject Name : Data Structures

Q. Why there is a need of data structure?

Q. What is a data structure?
- It is 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 efficiently.

- we want to store marks of 100 students

int m1, m2, m3, m4, m5, ..., m100;//400 bytes sizeof(int)=4 bytes

we want sort marks in descending order:

int marks[100];//400 bytes sizeof(int)=4 bytes

- we want to store information of 100 students
rollno : int
name : char [ ]
marks : float

word => it is unit of the memory from system point of view
word length = 8 bits / 16 bits / 32 bits / 64 bits
- on few machines word length is fixed, whereas on few machines word length may vary.

#pragma pack(1) => word length = 8 bits
#pragma pack(2) => word length = 16 bits

.
.

structure member alignment: word alignment/ word
word length =>

- there are two types of data types:
1. primitive/predefined/in built data types : char, int, float, double and void
2. non-primitive/derived data types : array, structure, union, pointer, enum, function etc...
typedef => user defined data type
typedef int INT;
- by using typedef we can give another name to an existing data type

```
typedef struct employee
{
	int empid;
	char name[ 32 ];
	float salary;
}emp_t;


typedef struct employee emp_t;

typedef unsigned int  size_t;
```

- function pointer => it is pointer variable in which we can store an addr of function.
- function name itself is an addr of the function i.e. an addr of first instruction in a function definition.

```
<return_type> <function_name>( arguments list);

unsigned int

unsigned int var;

typedef unsigned int size_t;

size_t var;
```

- to learn data structures is not learn any programming language, it is nothing but to learn algorithms, and these algorithms can be implemented in any programming language.

Q. What is an algorithm?
Q. What is a program?

Program => Machine
Algorithm => Human Beings/User

array scanning/traversal of an array => to visit each array element sequentially from first element max till last element.

- flowchart => it is a digramatic representation of an algorithm

- algorithm to do sum of array elements: => human being

```
step-1: initially take sum as 0
step-2: traverse an array and add each array element into the sum sequentially.
step-3: return final sum
```

pseudocode : to do sum of array elements: => programmer user

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

        return sum;
}
```

C Program => Compiler => Machine

```c
int array_sum(int arr[ ], int size)
{
        int sum = 0;
        int index;

        for( index = 0 ; index < size ; 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;
}
```

client => algorithm => manager => soft arch
pseudocode => developer => program => machine

- an algorithm is nothing but a solution of a given problem
- algorithm = solution

1. linear search / sequential search :

```
Algorithm LinearSearch(A, size, key)
{
        for( index = 1 ; index <= size ; index++ ){
                if( key == A[ index ] )
                        return true;
        }

        return false;

}
```

best case => if key is found in an array at 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 in an array at last position or key does 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

rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected.
e.g.
O(n + 3) => O(n)
O(n – 5) => O(n)
O(n / 2) => O(n)
O(4 * n) = O(n)

# DS DAY-02:
2. Binary Search:

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

iteration-1 => search space = 1000 = n
[0 1 2 ..... 1000]
[0 1 2 3 ..499] 500 [501 ... 1000]

iteration-2 => search space = 500 = n/2
[0 1 2 3 ..499]
[0 1 2 249] 250 [251 ... 499]

iteration-3 => search space = 250 = n/4
[0 1 2 249]
[0 ... 124] 125 [126 ... 249]

iterarion-4 => search space = 125 = n/8

1. Selection Sort:

iteration-1: no. of comparisons = n-1
iteration-2: no. of comparisons = n-2
iteration-3: no. of comparisons = n-3
.
.
.
iteration-(n-1) :

total no. of comparisons = (n-1)+(n-2)+.....
total no. of comparisons = n(n – 1) / 2
$T(n) = O(n(n – 1) / 2)$
$T(n) = O((n^2 – n) / 2)$
$T(n) = O(n^2 – n)$ .... as divisive constant can be neglected
$T(n) = O(n^2)$ ... in a polynomial leading is considered

rule : if running time of an algo is having a polynomial then in its time complexity only leading term will be considered.
e.g.
$O(n^3 + n + 1) => O(n^3)$
$O(n^2 – n) => O(n^2)$
$O(n^3 – n^2 + n – 5) => O(n^3)$

# DS DAY-03:
2. Bubble Sort:
iteration-1: no. of comparisons = n-1
iteration-2: no. of comparisons = n-2
iteration-3: no. of comparisons = n-3
.
.
.
iteration-(n-1) :

total no. of comparisons = (n-1)+(n-2)+.....
total no. of comparisons = n(n – 1) / 2
T(n) = O(n(n – 1) / 2)
T(n) = O((n^2 – n) / 2)
T(n) = O(n^2 – n) .... as divisive constant can be neglected
T(n) = O(n^2) ... in a polynomial leading is considered

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

```
                          pos < 3
for( pos = 0 ; pos < SIZE-it-1 ; pos++ )//inner for loop
```

best case :
flag = false

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 => not a single time control comes inside if block
=> array is already sorted

in best case => no. of iterations = 1 => no. of comparisons = n-1
T(n) = O(n - 1) => O(n)

rule => if any algo follows divide-and-conquer approach then we get time complexity in terms of log.

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 side by 1 pos
                j--;//goto the prev pos ele
        }

        //insert key at ist appropriate pos
        arr[ j+1 ] = key;
}
```

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

iteration-1:
10 20 30 40 50 60
key = 20
10 20 30 40 50 60
no. of comparisons = 1

iteration-2:
10 20 30 40 50 60
key = 30
10 20 30 40 50 60
no. of comparisons = 1

iteration-3:
10 20 30 40 50 60
key = 40
10 20 30 40 50 60
no. of comparisons = 1

iteration-4:
10 20 30 40 50 60
key = 50
10 20 30 40 50 60
no. of comparisons = 1

iteration-5:
10 20 30 40 50 60
key = 60
10 20 30 40 50 60
no. of comparisons = 1

in best case in each iteration only one comparison takes place and insertion sort algo takes max (n-1) no. of iterations
total no. of comparisons = 1*(n-1) = n-1
T(n) = O(n – 1) => O(n) => Ω(n).

# DS DAY-04:
- introduction to data structure
- algorithms
- searching algorithms: linear search & binary search
- sorting algorithms: selection sort, bubble sort, insertion sort and merge sort

- Quick Sort :

worst case: if either array elements are exitsts in already sorted manner or exactly in a reverse order => O(n^2) - rarely occurs.

10 20 30 40 50 60

pass=1: [10 20 30 40 50 60]
[LP] 10 [20 30 40 50 60]

pass=2: [20 30 40 50 60]
[LP] 20 [30 40 50 60]

pass=3: [30 40 50 60]
[LP] 30 [40 50 60]

- array data structure:
array is static i.e. size of an array cannot grow or shrink during runtime.
int arr[100];

- addition & deletion operations on an array are not efficient as it takes O(n) time.

Q. Why Linked List?
Linked List has been designed to overcome limitations of an array
1. array static => linked list must be dynamic
2. addition & deletion operations on an array are not efficient : O(n)
=> in linked list addition & deletion operations : O(1) time.

Q. What is a Linked List?

1. singly linear linked list:

if(head == NULL) => list is empty

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

#define NULL ((void *)0)

node has 2 parts:

```
struct node
{
        int data;//4 bytes
        struct node *next;//self referential pointer – 4 bytes
};
```

sizeof(struct node) = 8 bytes
sizeof(struct node *) = 4 bytes
scale factor(struct node *) = 8 bytes.

to store an addr of int type of var => int *
to store an addr of char type of var => char *
to store an addr of float type of var => float *
to store an addr of double type of var => double *
to store an addr of struct emp type of var => struct emp *
to store an addr of struct node type of var => struct node *

- on linked list data structure we can perform basic 2 operations:
1. addition : to add node into the linked list
2. deletion : to delete node from 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 spepcific position (inbetween position)

- we can delete node from the linked list by 3 ways:
1. delete node from the linked list at first position
2. delete node from the linked list at last position
3. delete node from the linked list at specific position (inbetween position)

1. add node into the singly linear linked list at last position:
- we can add as many as we want number of nodes into the slll at last position in **O(n)** time.

Best Case	**: if list is empty => O(1)**	**: Ω(1)**
Worst Case	**: if list is not empty => O(n):**	**O(n)**
Average Case	**: if list is not empty => O(n):**	**θ(n)**

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