Batch Name :
Subject Name : Data Structure

DS DAY-01:

Q. Why there is a need of data structure in programming?

There is a need of data structures in programming to achive efficiency in operations.

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 **organized manner** so that operations like **addition, deletion, searching, sorting, traversal etc....** can be performed on it efficiently.

- we want to store marks 100 students:

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

- we want to arrange marks in a descending order => sorting

int arr[100];//400 bytes

+ Array:

An array is **linear/basic data structure**, which is a **collection/ list of logically related similar type of data elements** gets stored into the memory at **contiguos locations.**

int arr[5] ;

- in an array, to covert array notation into its eq pointer notation is done by the compiler,

i.e. to maintained link between array elements in an array is the job of compiler.

- we want to store records of 100 employees:

empid : int
name : char [ ] / string
salary : float

- in an array we can collect/combine logically related similar type of data elements only, and hence to overcome this limitation structure data structure has been designed.

SunBeam

+ Structure:
It is a **linear / basic 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 entity / record.**

```
struct employee
{
        int empid;//4 bytes
        char name[ 32 ];//32 bytes
        float salary;//4 bytes
};
```

sizeof structure = sum of size of all its members
sizeof(struct employee): 40 bytes

- there are 2 types of data types:
1. primitive/predefined/builtin data type: char, int, float, double, void
=> data types which are already known to the compiler

2. non-primitive / derived / user defined data type : pointer, array, structure, union, enum, function etc...
=> data types which are not already known to the compiler, i.e. need to define/derived by the programmer

user defined data type:
typedef

```
typedef int INT;//user defined => not a derived data type
typedef int bool_t;//user defined => => not a derived data type
typedef struct employee emp_t;//derived / user defined
int *iptr;//derived data type
```

- there are two types of data structures:

1. linear / basic data structures: data structures in which data elements gets stored into the memory in a linear manner/linearly and hence can be accessed lineary (i.e. one after another).
- array
- structure & union
- linked list
- stack
- queue

2. non-linear / advanced data structures: data structures in which data elements gets stored into the memory in a non-linear manner (e.g. hierarchical manner), and hence can be accessed non-linearly.
- tree
- graph
- binary heap
- hash table

etc....

SunBeam

structure is a derived data type =>

```
struct employee
{
        int empid;//4 bytes
        char name[ 32 ];//32 bytes
        float salary;//4 bytes
};
```
=> compiler

<data type> <var_name>;
int num;

struct employee emp;

- to learn data structures, is not to learn any programming language, it is nothing but to learn algorithms, data structure algorithms can be implemented in any programming langauge (C / C++ / Java / Python).

Q. What is a Program? => Machine
- A Program is a finite set of instructions written in any programming language (i.e. either in low level / high level) given to the machine to do specific task.

Q. What is an algorithm? => User/Human Beings
- An algorithm is a finite set of instructions written in any human understandable langauge like english, if followed, acomplishesh given task.

- Program is an implementation of an algorithm.
- An algorithm is like a blue print / design of a program on paper.

Blue-print => implementation => building

- Pseudocode is a special form of an algorithm for programmer user.
Q. What is a pseudocode? => Programmer User
- An algorithm is a finite set of instructions written in any human understandable langauge like english with some programming constraints, if followed, acomplishesh given task, this kind/form of an algorithm us referred as a pseudocode.
- an algorithm to do sum of array elements:
- traversal on array => to visit each array element sequentially from first element max till last element.

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

SunBeam

```
Pseudode:  => Programmer
Algorithm ArraySum( arr, n){//arr is an array of size n
        sum = 0;
        for( index = 1 ; index <= n ; index++ ){
                sum += arr[ index ];
        }
        return sum;
}


Program: => Machine
int array_sum( int arr[ ], int size ){

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

Example:
IT Industry:
Client (Algorithm i.e. Requirement) => Software Architect/Tech Manager
Software Architect (Pseudocode) => Software Developer
Software Developer (Program) => Machine.

- an algorithm is a solution of a given problem.
- an algorithm = solution
- one problem may has many solutions
e.g.
searching => to search/find an element (let say referred as key element), in a given collection/list/set of elements.
1. linear search
2. binary search

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

etc.....

- if one problem has many solutions, we need to select an efficient solution out of them, and to decide which solution/algorithm is an efficient one, we need to do their analysis.

SunBeam

- **analysis of an algorithm** is a work of calculating/determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.
- there are 2 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.

1. linear search / sequential search:
algorithm:
step-1: accept key from user (key = element which is to be search)
step-2: start traversal of an array from first element and compare value of key element with each array element sequentialy till match is not found or max till last element.
step-3: if the value of key is matches with any of the array element then return true other wise return false.

pseudocode:

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

        //if key do not matches with any of array element
        return false;//key is not found
}
```

best case occurs : **if key is found 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 = 100 => no. of comparisons = 1

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

SunBeam

worst case occurs : if either key is found at last position or key does not exists : **O(n).**

if size of array = 10 => no. of comparisons = 10
if size of array = 20 => no. of comparisons = 20
if size of array = 100 => no. of comparisons = 100
.
.
if size of array = n => no. of comparisons = n

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

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

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

+ Asymptotic analysis: it is a **mathematical** way to calculate time complexity and space complexity of an algorithm without implementing it in any programming language.
- in this kind of analysis, focus is on basic operation in that algorithm
e.g. searching => basic operation is comparison, and hence anlaysis can be done depends on no. of comparisons takes places in different cases.
sorting => basic operation is comparison, and hence anlaysis can be done depends on no. of comparisons takes places in different cases.
Addition of matrices => basic operation addition, and hence analysis can be done depends on number addition instructions.

- there are some notations and few assumptions that we need to follow:

there are 3 asymtoptic notations:
1. Big Omega (Ω) - this notation is used to represent **best case time complexity.**
- big omega is referred as asymtptic lower bound
- running of an algo should not be less than its **asymtotic lower bound.**

2. Big Oh (O) - this notation is used to represent **worst case time complexity**
- big oh is referred as asymtptic upper bound.
- running of an algo should not be greater than its asymtotic upper bound.

3. Big Theta (θ) - this notation is used to represent an **average case time complexity.**
- big theta is referred as asymtptic tight bound.

SuReam

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

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

DS DAY-02:

2. Binary Search
algorithm:
step-1: accept key from user
step-2:
- calculate mid pos by the formula => mid = (left+right)/2
- by means of calculating mid position, big size array gets divided logically into two subarray's => left subarray & right subarray
- left subarray is from left to mid-1, and right subarray is from mid+1 to right.

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

step-3: compare value of key with ele at mid pos, if key matches with ele at mid pos return true

step-4: if key do not matches then search key either into the left subarray or into the right subarray

step-5: repeat step-2, step-3 & step-4, till either key not found, or max till subaarray is valid, if subarray is invalid then return false indicates key not found.

if(left <= right) => subarray is valid
if(left > right) => subarray is invalid

- in a binary tree, any element is at either one of the following 3 positions:
1. root pos
2. leaf pos
3. non-leaf pos

root node/root pos => first pos
node which is not having further child node => leaf node
node which is having child node => non-leaf node

SunBeam

if key is found at root pos => best case => O(1)
no. of comparisons for input size array = 1
time complexity = Ω(1).

if key is found at non-leaf pos => average case
O(log n)
time complexity = θ(log n)

if either key is found at leaf pos or key is not found => worst case
O(log n)
time complexity = O(log n)

+ Sorting Algorithms:
Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.
- when we say sort array elements, bydeafult we need to sort array elements in an ascending order.

1. Selection Sort:

for size of an array is n,
in iteration-1: no. of comparisons = n-1
in iteration-2: no. of comparisons = n-2
in iteration-3: no. of comparisons = n-3
.
.

iterations (n-1):

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

T(n) = O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

rule/assumption: 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 + 4) => O(n^3)
O(n^2 + 5) => O(n^2)
O(n^3 + n^2 + n – 3) => O(n^3)

SunBeam

DS DAY-03:
2. Bubble Sort:

for size of an array is n,
in iteration-1: no. of comparisons = n-1
in iteration-2: no. of comparisons = n-2
in iteration-3: no. of comparisons = n-3
.
.

iterations (n-1):

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

T(n) = O((n^2 – n) / 2)
=> O(n^2 – n)
=> O(n^2)

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

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

best case:
input array => 10 20 30 40 50 60
flag= 0

for it=0
iteration-0:
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

for it=1
iteration-1: all next iterations will be skipped

- if all pairs are in order
=> if there is no need of swaaping in first iteration, it means all pairs are already in order => array is already sorted -> no need to go to next iteration and we are reducing no. of comparisons at an implementation level.

SunBeam

=> array ele's are already sorted

- in best case this algo takes only one iteration and
total no. of comparisons = n-1
T(n) = O(n - 1) => O(n)
best case time complexity = Ω(n).

3. Insertion Sort:

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

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

        //insert key into the left hand i.e. into an array at its appropriate pos
        arr[ j+1 ] = key;
}
```

best case : if array ele's are already sorted:

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

10 20 30 40 50 60

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

10 20 30 40 50 60

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

10 20 30 40 50 60

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

10 20 30 40 50 60

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

SunBeam

10 20 30 40 50 60

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

4. Merge Sort:
- this algo follows divide-and-conquer approach
- there are 2 steps in this algo:

step-1: divide big size array logically into smallest size subarray's i.e. subarray having size 1.

- in each iteration by means of calculting mid pos, we are dividing big size array logically into two subarray's, left subarray will be from left to mid, and right subarray will be from mid+1 to right.

for left subarray, value of left remains as it is and right = mid
for right subarray, value of right remains as it is and left = mid+1

step-2: merge two already sorted subarray's into a single array in a sorted manner.
- if subarray contains only 1 ele => subarray is already sorted.

 # DS DAY-04:
5. Quick Sort:
- this algo follows divide-and-conquer approach.
- the basic logic of this algo is partitioning

partitioning:
step-1: to select pivot element (in an array we can select either **leftmost** or rightmost or middlemost element as a pivot element)

step-2: shift all ele's which are smaller than pivot towards its left and shift all ele's which are greater than pivot towards its right, by means of this shifting pivot element gets fixed at its appropriate pos and big size array gets divided logically into two partitions, **left partition & right partition.**

step-3: apply partitioning on left partition as well as right partition till the size of partition is greater than 1.

SunBeam

worst case(quick Sort): if either array elements are exitsts in already sorted manner or exactly in a reverse order => O(n2)
 - rarely occurs.

SunBeam