Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

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

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
	int rollno;
	char name[32];
	float marks;
};
```

<data type> <var_name>;
- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

e.g.

```
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

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

- Algorithm to do sum of array elements: => Any User
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 value of sum.

- Pseudocode to do sum of array elements: => Programmer User

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

- Program to do sum of array elements: => Machine

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

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    time
    distance
    cost
    traffic condition
    status of path
    etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

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

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

```
Sp = space required for simple vars + space required for constants
+ space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n  => n units => instance var


index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like -
traversal methods in tree.


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


- for any input size array, no. of instructions will be remain
same, hence compilation time also remains same for any input size
array.


Asymptotic Analysis:

Searching => to search/find key element in a given collection/list
of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

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

     return false;
}
```

Best Case : if key is found in an array at first position
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 = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

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 = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

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

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided logically into two subarray's:
left subarray and right subarray

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

best case : if key is found in an array in very first iteration
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 = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = O(1)
time complexity of binary search in best case => Ω(1).

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

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => T(1) = O(1)

if size of an array > 1 i.e. size of an array = n

$T(n) = T(n) + 1$

after iteration-1:
$T(n) = T(n / 2) + 1 \Rightarrow T(n / 2^1) + 1$

after iteration-2:
$T(n) = T(n / 4) + 2 \Rightarrow T(n / 2^2) + 2$

after iteration-3:
$T(n) = T(n / 8) + 3 \Rightarrow T(n / 2^3) + 3$

.
.

lets assume, k no. of iterations takes place
after k iterations:
$T(n) = T(n / 2^k) + k$

lets assume kth iteration is the last iteration
assume => $n \Rightarrow 2^k$

=> $n \Rightarrow 2^k$
=> $\log n = \log 2^k$ ...... by taking log on both sides
=> $\log n = k \log 2$
=> $\log n = k$ ..... [$\log 2 \approx 1$]
=> $\underline{k = \log n}$

$T(n) = T(n / 2^k) + k$
substitute values of $n = 2^k$ & $k=\log n$ in above equation, we get
$T(n) = T(2^k / 2^k) + \log n$
$T(n) = T(1) + \log n$
$T(n) = \log n$
$T(n) = O(\log n)$

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = $(n-1) + (n-2) + (n-3) + (n-4) +... + 1$
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^2 + 5) \Rightarrow O(n^2)$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace – if sorting algo do not takes extra space

2. adaptive – if a sorting algo works efficiently for already sorted input sequence

3. stable – if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
<u>$T(n) = O(n^2)$</u>

```
n = size of an array / arr.lenght = 6

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

for( pos = 0 ; pos < n-1-itr ; pos++ )


input array => 10 20 30 40 50 60

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

10 20 30 40 50 60


in first iteration,
if there is no need of swapping even single time if all pairs are
already in order => array is already sorted


best case: if array elements are already sorted
total no. of comparisons = n-1
T( n ) = O( n - 1)
T( n ) = O( n ) ... [ 1 is a substractive constant, it can be
neglected ].


3. Insertion Sort:


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

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

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

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list 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 link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 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:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add 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 (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from 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 speecific position (in between position).

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

Best Case : $O(1)$: $\Omega(1)$- if list is empty
Worst Case : $O(n)$: $O(n)$
Average Case : $O(n)$: $\Theta(n)$

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

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

Best Case : $O(1)$: $\Omega(1)$- if list is empty
Worst Case : $O(1)$: $O(1)$
Average Case : $O(1)$: $\Theta(1)$

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

Best Case : $O(1)$: $\Omega(1)$- if pos = 1
Worst Case : $O(n)$: $O(n)$ - if pos = last pos
Average Case : $O(n)$: $\Theta(n)$

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.