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 => $\Omega(1)$.

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 => $n \Rightarrow 2^k$

$\Rightarrow n \Rightarrow 2^k$
$\Rightarrow \log n = \log 2^k$ ..... by taking log on both sides
$\Rightarrow \log n = k \log 2$
$\Rightarrow \log n = k$ ..... [$\log 2 \sim= 1$]
$\Rightarrow$ <u>$k = \log n$</u>

$T(n) = T(n/2^k) + k$
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 = $(n-1) + (n-2) + (n-3) + (n-4) + \ldots + 1$
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>

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
$T(n) = O(n^2)$

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.

DAY-04:
sorting algorithms:
- bubble sort: algorithm, analysis and implementation
- insertion sort : algorithm, analysis and implementation
- comparisons of sorting algorithms
- limitations of an array data structure
- why linked and what is linked list
- types of linked list
- operations on slll: addlast, addfirst, addatpos

- deletion of a node from linked list:
i. delete node from the list at first position
ii. delete node from the list at last position
iii. delete node from the list at specific position

i. delete node from the list at first position:
- we can delete node from slll at first position in O(1) time.

Best Case	: O(1)	: Ω(1)
Worst Case	: O(1)	: O(1)
Average Case	: O(1)	: Θ(1)

ii. delete node from the list at last position:
- we can delete node from slll at last position in O(n) time.

Best Case	: O(1)	: Ω(1) - if list contains only one node
Worst Case	: O(n)	: O(n)
Average Case	: O(n)	: Θ(n)

iii. delete node from the list at specific position:
- we can delete node from slll at specific position in O(n) time.

Best Case	: O(1)	: Ω(1) - if pos=1
Worst Case	: O(n)	: O(n) - if pos=last pos
Average Case	: O(n)	: Θ(n)

example:
system calls:

system call API => write( ) => _write( )sys call is used to write data into the file
write() sys call API is like a wrapper function which internally makes call to actual _write() sys call

in parameters => parameters which we pass to the function as an input
out parameters => parameters which we pass to the function as an output i.e. to get output.

=> priority queue => can be implemenetd by using linked list with search_and_delete() function.

```
Class Node{
     private <type> data;
     private int priority_value;
     Node next;

}
```

=> priority queue is a type of queue in which elements can be added into it randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

=> delete() function in a binary search tree

+ limitations of slll:
- add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- we can traverse slll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only
- we cannot revisit any node in a slll => to overcome this limitation scll has been designed.

SCLL:
- We can perform all the operations that we applied on SLLL as it is on SCLL, except we need to take care about/ maintained next part of last node always.

- add at position in slll & scll remains exactly same.

- deletion of a node at first position:

H.W. Implement SCLL functionalities by using menu driven program. addition & deletion

DAY-04:

operations on sinlgy linear linked list:
- deletion: deleteFirst, deleteLast & deleteAtPosition
- display list in reverse order by recursive method
- reverse linked list
- search and delete
- limitations of slll => scll
- operations on scll => adddition & deletion

DAY-05:
+ limitations of slll:
- addLast, addFirst, deleteFirst & deleteFirst operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only

- to overcome limitations of singly linked list (slll & scll), doubly linked list has been designed.
- there are two types of doubly linked list
i. doubly linear linked list
ii. doubly circular linked list
- we can perform all operations on dlll which we applied on slll exactly as it is, except in dlll we need to maintained forward link as well as backward link of each node.

H.W.
Menu driven: SCLL, DLLL, DCLL
Implement deleteFirst => DCLL

Linked List => DCLL

Java Collection => Linked List

Applications of Linked List:
- Linked List data structure is used to store logically related similar type of data elements, if we dont know in advance size of the list/collection.
- Linked List is used to implement basic data structures like stack & queue.
- Linked List is used to implement kernel data structures like e.g. ready queue (list of PCB's), job queue (list of PCB's), iNode list (list of iNodes) => linked list implementation of queue
- Linked List is used to implement advanced data structures like tree, graph, hash table etc... and its algorithms can also be implemented by uing Linked List.

+ Stack: It is a basic/linear data structure, which is collection/list of logically related similar type of data elements, in which elements can be added as well as deleted from only one end referred as top end.

- in this list elements which was added last can only be deleted first, so this list works in last in first out manner, hence stack is also called as LIFO List/FILO List.

- we can apply basic 3 operations on stack data structure 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 end

3. Peek : to get the value of an element which is at top end 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)

- data structures can also be catagoriesed into two catagories:
1. static data structures: array, structure, class

2. dynamic data structure: linked list, tree, graph, hash table etc....

stack & queue
- Adaptive Data Structure => stack & queue data structure adapts feature of data structure by using which we implement it.
- If we implement stack by using an array it becomes static, whereas if we implement stack by using linked list it becomes dynamic.

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

```
Class Stack{
     private int [] arr;
     private int top;
     .
     .
     .

}
```

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

2. Pop : to delete/remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop element from it).
step-2: decrement the value of top by 1.
(by means of decrementing the value of top by 1 we are achieving deletion of an element from the stack).

3. Peek : to get the value of an element which is at top end without Push/Pop.
step-1: check stack is not empty (if stack is not empty then only we can peek element from it).
step-2: get the value of an element which is is at top end
[without incrementing/decrementing top].

DAY-06:
- Linked List
+ Stack :
 - concept and definition
- implementation of the stack by using an array => static stack

+ implementation of the stack by using linked list (dcll) => dynamic stack:

- stack works in last in first out manner
push : addLast( )
pop : deleteLast( )

peek

dynamic stack:
head => 22 11

OR
push : addFirst( )
pop : deleteFirst( )

H.W. Implement dynamic stack by using linked list.

+ Applications of Stack:
- undo & redo functionalities in an OS as well as in applications like editor, ms excel etc.... stack is used.
- stack is used to implement advanced data structure algorithms like dfs traversal in tree & grap, inorder, preorder and postorder traversal techniques in tree.
- stack is used by an OS to control flow of an execution of programs by maintaining FAR's onto the stack.
- to reverse a string
- stack is used in compilers for parenthesis balancing
- stack is also used in an algorithm to convert given infix expression into its prefix or postfix expression and in an algo to evaluate expressions.

+ expression conversion and evalution algorithms and their implementation.

Q. What is an expression ?
- An expression is a combination of an operands and an operators.

- there are 3 types of expressions:
1. infix expression   : a+b
2. prefix expression   : +ab
3. postfix expression   : ab+

1. algorithm to convert given infix expression into its equivalent postfix:

Lab Work: Implement Postfix evaluation algorithm for operands having multiple digits.

+ 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 into it from one end referred as rear end, whereas elements can be deleted from the queue from another end referred as front end.

- in this list element which was inserted first can be deleted first, so this list works in first in first out manner, hence it is also called as FIFO list/LILO list.

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
2. dequeue : to delete an element from the queue which is at front end.

- there are diff 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 from rear end randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.
- priority queue can be implemented by using linked list (searchAndDelete( ) function).

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.
- there are two types of deque:
i. input restricted deque : it is a type of deque in which elements can be inserted into it only from one end, whereas elements can be deleted from both the ends.

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

- we can perform basic 4 operations on deque in O(1) time:
1. push front : addFirst( )
2. push back : addLast( )
3. pop front : deleteFirst( )
4. pop back : deleteLast( )

- deque can be implemented by using dcll

HW: Priority queue & Dequeue

+ Implementation of Linear Queue & Circular Queue:
- FIFO Queue can be implented by two ways:
1. static implementation of fifo queue by using array => static queue
2. dynamic implementation of fifo queue by using linked list => dynamic queue.

1. static implementation of fifo queue by using array => static queue:

```
class Queue{
     private int [] arr;
     private int rear;
     private int front;

     Queue( ){
          arr = new int[ 5 ];
          rear = -1;
          front = -1;

     }
}
```

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert 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 from rear end).
step-2: increment the value of rear by 1.
step-3: insert an element into the queue at rear end
step-4: if(front == -1)
          front = 0;

2. dequeue : to delete 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 element from it from front end).
step-2: increment the value of from by 1.
[by means of incrementing the value of front by 1 we are achieving deletion of an element from the queue].

Circular Queue

```
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2
rear = 2, front = 3
rear = 3, front = 4

if front is at next pos of rear => cir queue is full

if( front == (rear + 1) % SIZE )
     cir queue is full

for => rear = 0, front = 1 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for => rear = 1, front = 2 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

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

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

for => rear = 4, front = 0 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full
```

- increment the value of front by 1.
=> front++;
=> front = front + 1;

to increment value of front by 1
front = (front+1)%SIZE
for front=0
=> front = (front+1)%SIZE
=> front = (0+1)%5
=> front = 1%5
=> front = 1

for front=1
=> front = (front+1)%SIZE
=> front = (1+1)%5
=> front = 2%5
=> front = 2

for front=2
=> front = (front+1)%SIZE
=> front = (2+1)%5
=> front = 3%5
=> front = 3

for front=3
=> front = (front+1)%SIZE
=> front = (3+1)%5
=> front = 4%5
=> front = 4

for front=4
=> front = (front+1)%SIZE
=> front = (4+1)%5
=> front = 5%5
=> front = 0

DAY-06:
- applications of stack data structure
- stack application algorithms: expression conversion and evaluation algorithms and their implementation.
- queue: concept, definition and types of queue implementation of linear queue and circular queue by using an array.

Queue can be implemented by using an array as well as by using linked list.

- dynamic queue by using linked list(dcll):

enqueue => addLast( )
dequeue => deleteFirst( )

head => 40 50
OR
enqueue => addFirst( )
dequeue => deleteLast( )
Lab Work => Implement dynamic queue by using linked list.

+ Array:
arr[0] = 10 &arr[0] = 1000
arr[1] = 20 &arr[1] = 1004
arr[2] = 30 &arr[2] = 1008
.
.
.

10, 20, 30, 40., ... => int type data elements gets stored into the memory at contiguos locations => contiguos as well as a linear - array ele's can be accessed linearly i.e. sequentially and as each array ele has index so we can access by random access method as well.

Linked List:
head [1000] => | 10 : 2000 | => | 20 : 3000 | => | 30 : 4000 | => | 40 : null |

linked list elements do not gets stored into the memory in a contiguos manner, but as we link them in a linear manner hence it can be accessed linearly/sequencially

1st node is at 1000
2nd node is at 1008
3rd node is at 1016

 # DAY-07:
- Advanced Data Structures:

+ tree: it is a non-linear/advanced data structure, which is a collection/list of logically related similar type of finite no. of data elements in which there is a first specially designated element referred as root element and remaining all elements are connected to the root element in a hierarchical follows parent-child relatioship.

- root node
- parent node/father
- child node/son
- grand parent node / grand father
- grand child node / grand son
- ancestors: all the nodes which are in the path from root node to that node.
- root node is an anscestor of all the nodes
- descendents: all the nodes which can be accessible from that node.
As all the nodes can be accessed from root node, hence all the nodes are descedents of root node.
- degree of a node = no. of child node/s having that node.
- degree of a tree = max degree of any node in a given tree
- leaf node is also called as terminal node/external node
- non-leaf node is also called as non-terminal node/internal node

- by concept tree is a dynamic data structure
- as tree is a dynamic data structure, so any node can have any no. of child nodes and it can grow upto any level, but due to this feature, operations on it like specially addition, deletion & searching becomes inefficient, so restrictions can be applied on tree, hence there are different types of tree:

- binary tree : it is a type of tree in which each node can have max 2 no. of child nodes.
OR each node can have degree either 0 OR 1 OR 2
- it is a tree in which max degree of any node is 2.

empty set / null set = 0 no. of elements
singleton set = contains only 1 element
set = more than 1 elements

- further restrictions can be applied on binary tree to achieve operations like addition, deletion and searching efficiently i.e. expected in O(log n) time, hence binary search tree has been designed.

- binary search tree => it is a binary tree in which left child is always smaller than its parent and right child is always greater or equal to its patent.

- we can add as many as we want number of nodes into the BST in O(log n) time, as to find an appropriate position for a newly created node takes O(log n) time.

- to traverse a tree/BST => to visit each node in a tree at once.
- in a tree, we can start traversal always from root node only.
- there are basic two tree traversal methods:
1. bfs (breadth first search) traversal
2. dfs (depth first search) traversal

1. bfs (breadth first search) traversal:
- bfs traversal is also called as level-wise traversal
- in this traversal method, traversal starts from root node and nodes gets visited level wise from left to right.

2. dfs (depth first search) traversal:
- in this traversal method, traversal starts from root node.
- under dfs, further bst can be traversed by 3 ways:

i. Preorder (VLR)
ii. Inorder (LVR)
iii. Postorder(LRV)

L => Left Subtree
R => Right Subtree
V => visit

i. Preorder traversal (VLR):
- in this traversal method, traversal starts from root node.
- very first we can visit cur node, then we can visit its left subtree, and then only we can visit right subtree of cur node i.e. we can visit right subtree of any node only after visiting its whole left subtree or left subtree is empty.
- in this type of traversal, root node always gets visited at first, this property remains same recursively for each subtree.

ii. Inorder traversal (LVR):
- in this traversal method, traversal starts from root node.
- very first we can visit left subtree of cur node then we can visit that node and then only we can visit its right subtree.
- we can visit any node only after visiting its whole left subtree or its left subtree is empty and then only we can visit right subtree of that node.

- in this type of traversal, all the nodes always gets visited in ascending order.

iii. Postorder traversal (LRV):
- in this traversal method, traversal starts from root node.
- first we need to visit whole left subtree of cur node, then we can visit its right subtree and then only we can visit cur node i.e. we can visit any node either only after visiting its left subtree as well as a right subtree or left subtree and right right subtree are empty.

- in this type of traversal, root node always gets visited at last, this property remains same recursively for each subtree.