 # Module Name : Algorithms & Data Structures Using Java.

==========

 # DAY-01:

 # Introduction to data structures?

Q. Why there is a need of data structures?

- if we want to store marks of 100 students

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

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

- we want to store rollno, marks & name

```
rollno   : int
marks    : float
name     : char [] / String / String

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

struct student s1;

class Employee
{
    //data members
    int empid;
    String empName;
    float salary;
    //member functions/methods
};

Employee e1;
Employee e2;
```

=> to learn data structures is not learn any programming language, it is a programming concept i.e. it is nothing but to learn algorithms, and algorithms learned in data structures can be implemented by using any programming language.

 # algorithm to do sum of array elements => end user (human being)
step-1: initially take sum as 0.
step-2: traverse an array and add each array element into sum sequentially
from first element max till last element.
step-3: return final sum.

 # pseudocode => programmer user

```
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;
}
```

 # program => compiler => machine

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

Bank Manager => Algorithm => Project Manager => Software Architect
=> Pseudocode => Developers => Program => Machine

Problem : **"Searching"** => to search / to find an element (can be referred as a
key element) into a collection/list of elements.
1. Linear Search
2. Binary Search
Problem : **"Sorting"** => to arrange data elements in a collection / list of elements either 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....

- when one problem has many solutions we need to go for an efficient solution.

City-1:
City-2:

multiple paths exists => optimum path
distance, condition, traffic situation ....

- to traverse an array => to visit each array element sequentially from first element max till last element.

- there are two types of algorithms:
1. iterative approach (non-recursive)
2. recursive approach

- recursion
- recursive function
- recursive function call
- tail-recursive function
- non-tail recursive function

```
Class Employee
{
    int empid;
    String name;
    float salary;
```

```
};

- object e1 is an instance of Employee class
Employee e1(1, "sachin", 1111.11);
Employee e2(2, "nilesh", 2222.22);
Employee e3(3, "amit", 3333.33);
```

 # Space Complexity:

```
for size of an array = 5 => index = 0 to 5 => only 1 mem
copy of index = 1 unit
for size of an array = 10 => index = 0 to 10 => only 1
mem copy of index = 1 unit
.
.
for size of an array = n => index = 0 to n => only 1 mem
copy of index = 1 unit

for any input size array we require only 1 memory copy of
index var =>
simple var

+ sum:
for size of an array = 5 => sum => only 1 mem copy of sum
= 1 unit
for size of an array = 10 => sum => only 1 mem copy of
sum = 1 unit
.
.

for any input size array we require only 1 memory copy of
sum var => simple var


+ n = input size of an array -> instance characteristics
of an algo

for size of an array = 5 => if n = 5 => 5 memory copies
required to store 5 ele's => 5 units

for size of an array = 10 => if n = 10 => 10 memory
copies required to store 10 ele's => 10 units
```

for size of an array = 20 => if n = 20 => 20 memory copies required to store 20 ele's => 20 units

for size of an array = 100 => if n = 100 => 100 memory copies required to store 100 ele's => 100 units

size

- for any input size array no. of instructions inside an algo remains fixed i.e. space required for instructions i.e. code space for any size array will going to remains fixed or constant.

```
int sum( int n1, int n2 )//n1 & n2 are formal params
{
    int res;//local var
    res = n1 + n2;
    return res;

}
```

- When any function completes its excution control goes back into its calling function as an addr of next instruction to be executed in its calling function gets stored into FAR of that function as a **"return addr"**.

FAR contains:
local vars
formal params
return addr => addr of next instruction to be executed in its calling function
old frame pointer => an addr of its prev stack frame/FAR.
etc...

 # Linear Search:

```
for( index = 1 ; index <= n ; index++ ){
    if( key  == arr[ index ] )
        return true;
}

return false;
```

- In Linear Search best case "if key found at very first position"
for size of an array = 10, no. of comparisons = 1
for size of an array = 20, no. of comparisons = 1
for size of an array = 50, no. of comparisons = 1
.
.

for any input size array => no. of comparisons = 1
Running Time => O(1)

- In Linear Search worst case "if either key found at last first position or key does not exists"
for size of an array = 10, no. of comparisons = 10
for size of an array = 20, no. of comparisons = 20
for size of an array = 50, no. of comparisons = 50
.
.

for size of an array = n, no. of comparisons = n
Running Time => O(n)

Lab Work => Implement Linear Search => by non-rec as well rec way

 # DAY-02:
- linear search
- binary search
- comparison between searching algo
- sorting algorithms:
basic sorting algo's : selection, bubble & insertion

assumption-1:
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 - 2) => O(n)
O(n / 5) => O(n)
O(6 * n) => O(n)

Binary Search:

by means of calculating mid position big size array gets divided logically into two subarray's:
left subarray => left to mid-1
right subarray => mid+1 to right

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

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

best case occurs in binary search if key is found in very first iteration in only 1 comparison.
if size of an array = 10, no. Of comparisons = 1
if size of an array = 20, no. Of comparisons = 1
.
.
for any input size array, no. Of comparisons = 1
=> running time => O(1)

- in this algo, **in every iteration 1 comparison takes place and search space gets divided by half** i.e. array gets divided logically into two subarray's and in next iteration we will search key either into left subarray or into right subarray.

 # Substitution method to calculate time complexity of binary search:

Size of an array = 1000
1000 = n
500 = n/2
250 = n/4
125 = n/8
.
.
.

n/2 / 2 => n / 4
n/4 / 2 => n / 8
.
.

for iteration-1 input size of an array => n
after iteration-1: n/2 + 1 => $n / 2^1 + 1$
after iteration-2: n/4 + 2 => $n / 2^2 + 2$
after iteration-3: n/8 + 3 => $n / 2^3 + 3$
.
.
.
after iteration-k: $n / 2^k + k$ .... eq-I

$T(n) = n / 2^k + k$ .... eq-I

lets assume, $n = 2^k$
$n = 2^k$
$\log n = \log 2^k$ .... **[by taking log on both sides]**
$\log n = k \log 2$

log n = k .... **[as log 2 ~= 1]**
k = log n

put value of n = 2^k and k = log n in eq-I, we get
$T(n) = n / 2^k + k$
$=> T(n) = 2^k / 2^k + \log n$
$=> T(n) = 1 + \log n$
$=> T(n) = O(1 + \log n)$
$=> T(n) = O(\log n + 1)$
$=> T(n) = O(\log n)$.

1. Selection Sort:

total no. of comparisons = (n-1)+(n-2)+(n-3) + ...+ 1
=> n(n-1) / 2
hence
=> T(n) = O(n(n-1) / 2)
=> T(n) = O((n^2 - n) / 2)
=> T(n) = O(n^2 - n)
=> **T(n) = O(n^2)**

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

assumption:
if an algo contains a nested loops and no. Of iterations of outer loop and inner loop dont know in advanced then running time of such algo will be whatever time required for statements which are inside inner loop.

```
for( i = 0 ; i < n ; i++ ){

    for( j = 0 ; j < n ; j++ ){
        statement/s  => n*n no. Of times => O(n^2) times
    }

}
```

+ features of sorting algorithms:

1. inplace => if a sorting algo do not takes extra space (i.e. space required other than actual data ele's and constant space) to sort data elements in a collection/list of elements.

2. adaptive => if a sorting algorithm works efficiently for already sorted input sequence then it is referred as an adaptive.

3. stable => if in a sorting algorithm, relative order of two elements having same key value remains same even after sorting then such sorting algorithm is referred as stable.

Input array => 10 40 20 30 10' 50

After Sorting:
Output => 10 10' 20 30 40 50 => stable

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

 # Design & Analysis of an Algorithm By Coreman

2. Bubble Sort:

 # DAY-03:

3. Bubble Sort:

total no. of comparisons = (n-1)+(n-2)+(n-3) + ...+ 1
=> n(n-1) / 2
hence
=> T(n) = O(n(n-1) / 2)
=> T(n) = O((n^2 - n) / 2)
=> T(n) = O(n^2 - n) .... [as 2 is a divisive constant it can be negected]
=> **T(n) = O(n^2)** ....

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

for(pos = 0 ; pos < arr.length-1-it ; pos++)

Best Case : if array elements are already sorted

flag = false

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

if all pairs are in order => array is already sorted => no need of swapping => no need to goto next iteration

in best case only 1 iteration takes places and in iteration total (n-1) no. Of comparisons takes place
T(n) = O(n - 1)
T(n) = O(n).

3. Insertion Sort:

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

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

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

```
}

Best Case:
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 50

10 20 30 40 50 50

no. of comparisons = 1

Iteration-5:
10 20 30 40 50 50

10 20 30 40 50 50

no. of comparisons = 1
```

in best case total (n-1) no. of iterations are required and in each iteration only 1 comparison takes place
total no. Of comparisons = 1 * (n-1) => n-1
T(n) = O(n - 1)
T(n) = O(n) => Ω(n).

+ Linked List:

Q. Why Linked List ?

+ Limitations of an array data structure:
1. in an array we can collect/combine logically related only similar type of data element => to overcome this limitation **structure** data structure has been designed.

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

3. **addition & deletion** on an array are not efficient as it takes **O(n)** time.
- while adding ele into an array we need to shift elements towards its right hand side by one-one pos till depends on size of an array, whereas while deleting an ele from an array we need to shift elements towards its left hand side by one-one pos till depends on size of an array

- to overcome 2nd & 3rd limitations of an array data structure **linked list** data structure has been designed.

- linked list must be dynamic
- addition & deletion operations on linked list must be perfomed 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(ref) of first element in it always kept into a pointer/ref referred as head, and each element contains actual data and an addr/link of its next element (as well as link of its prev element) in it.

- Element in a Linked List is also called as a node.
- Basically there are 2 types of linked list:
1. singly linked list : it is a type of linked list in which each node in it contains link to its next node only i.e. in each node no. of links = 1
- there are 2 subtypes of sll:
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 in it contains link to its next node as well as link to its prev node i.e. in each node no. of links = 2
- there are 2 subtypes of dll:
i. doubly linear linked list
ii. doubly circular linked list

i. singly linear linked list =>
- on linked list we can perform basic 2 operations
1. addition : to add/insert node into the linked list
2. deletion : to delete/remove node from the linked list

1. addition : to add/insert 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 speficif position (in between position).

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

Best Case : Ω(1) - if list is empty
Worst Case : O(n)
Average Case : Θ(n)

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
- we can start traversal from first node and we get an addr/ref of first node always from head

ii. add node into the linked list at first position:
- we can add as many as we want number of nodes into the **sllll** at first position in **O(1) time.**

Best Case : Ω(1) - if list is empty
Worst Case : O(1)
Average Case : Θ(1)

iii. add node into the linked list at specific position:
- we can add as many as we want number of nodes into the **slll** at first position in **O(n) time.**

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

- In a Linked List Programming remember one rule:
make before break => always create new links (links which are associative with newly created node) first and then only break old links.

Lab Work : Convert SLLL program into a menu driven program.

DAY-04:

there are 2 types of programming langauges:
1. procedure oriented programming language
e.g. C : procedure => function
logic of a program gets divided into functions

maintainability =>

2. object oriented programming language
e.g. C++, Java :
logic of a program gets divided into classes

- we can delete node from linked list by 3 ways:
1. delete node at last position

2. delete node at first position
3. delete node at specific position

searchAndDelete():
- priority queue can be implemented using linked list
searchAndDelete()
- BST => deleteNode()

addition();

addLast()
addFirst()
addAtPos()

- whatever basic operations (i.e. addition & deletion) we applied on slll, all operations can be applied onto the scll as it is, execpt we need to maintained / take care about next part of last node always.

i. add node at last position
Lab Work:
ii. add node at first position
iii. add node at specific position
iv. delete node at first position
v. delete node at last position
vi. delete node at specific position

DS DAY-05:
+ Operations on slll:
 - delete node at first position
 - delete node at last position
 - delete node at specific position
 - search and delete : linear search
 - to display linked list in reverse order by using recursion
 - to reverse the linked list
 - limitations of slll
 - operations on scll:

+ limitations of scll:
- in scll, addLast(), addFirst(), deleteLast() & deleteFirst() operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward direction
- prev node of any node cannot be accessed from it

- to overcome limitations of singly linked list (slll & scll) doubly linked list has been designed.

"doubly linear linked list": it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
 i. data part : contains actual data of any primtive/non-primitive type
 ii. next part (ref) : contains reference of its next node
 iii. prev part (ref): contains reference of its prev node
- prev part of first node and next part of last node points to null.

- all operations that we performed on slll, can be applied as it is on dlll, only we need to maintained forward link as well as backward link of each node.

+ limitations of dlll:
- in dlll, addLast() and deleteLast() operations are not efficient as it takes O(n) time.
- we can start traversal only from first node in O(1) time.

- to overcome limitations of dlll, dcll linked list has been designed.

"doubly circular linked list": it is a type of linked list in which,
- head always contains an addr of first node if list is not empty
- each node has 3 parts:
 i. data part : contains actual data of any primtive/non-primitive type
 ii. next part (ref) : contains reference of its next node
 iii. prev part (ref): contains reference of its prev node
- prev part of first node points to last node and next part of last node points to first node.

Collection LinkedList => DCLL - generics

```
class Employee{
    //data members
    //methods
}

class LinkedList{

    static class Node{
        Employee data;//data part of a node is of type Employee class object
        Node next;
        Node prev;
    }
```

```
    .....
}
```

+ applications of linked list:
1. linked list is used to imnplement basic data structures like stack, queue, priority queue, deque etc...
2. linked list is used to imnplement advanced data structures like tree, graph & hash table.
3. linked list is used in an OS to implement kernel data structures lik job queue, ready queue, kernel linked list, iNode list (linked list of FCB's) etc....
4. undo & redo functionalities of an OS
etc...

+ difference between array & linked list:
1. array is "static", whereas linked list "dynamic"
2. addition and deletion operations on an array are not efficient as it takes O(n) time, whereas addition & deletion operations on linked list can be performed efficiently in O(1) time and are convenient as well.
3. array elements can be accessed by using "random access method" which is efficient than "sequential access method" used in linked list.
4. searching operation can be performed on an array efficiently as we can apply binary search on it, whereas on linked list we can perform only linear search.
5. to store "n" no. of elements in an array it takes less space than to store "n" no. of elements in a linked list as we have maintained link between element in it explicitly which takes extra space.
6. array ele's gets stored into memory in data section/stack section, whereas linked list elements gets stored into the memory in the heap section.
7. as array elements gets stored into the memory at contiguos locations, to maintain link between array elements is the job compiler, whereas to maintained link between elements is the job of programmer, we have to explicitly maintained link between nodes in a list.

+ "Stack": 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 as well as deleted from only one end referred as top end.

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

- We can perform basic 3 operations onto the Stack in O(1) time:
1. Push : to add/insert 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 of an element).

- Stack can be implemented by 2 ways:
1. static implementation of stack (by using an array) => static stack
2. dynamic implementation of stack (by using linked list) => dynamic stack

"adaptor": as stack can be static as well as dynamic, as it adopts the feature of data structure by using which we implement it.

1. static implementation of stack (by using an array) => static stack:

```
int arr[ 5 ];
int top;
```

arr : int [] - non-primitive data type
top : int - primitive data type

1. Push : to add/insert an element onto the stack from top end:
step-1: check stack is not full (if stack is not full then only element can be pushed onto the stack from top end).
step-2: increment the value of top by 1
step-3: insert an element onto the stack at top end

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 element can be pop from the stack which is at top end).
step-2: decrement the value of by 1 [by decrementing 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 of an element).
step-1: check stack is not empty (if stack is not empty then only element can be peeked from the stack which is at top end).
step-2: return/get the value of an element at top end [without increment/decrement top].

DAY-06:
- Operations SCLL, DLLL, DCLL
- Stack: concept & definition
- We can perform basic 3 operations on Stack in O(1) time: Push, Pop & Peek
- Stack can be implemented by 2 ways:
1. Static Stack (by using an array)
2. Dynamic Stack (by using linked list-dcll)

- there is no stack full condition in a dynamic stack
- if list is empty => stack is empty

Stack works in LIFO manner:
Push => addLast() - O(1)
Pop => deleteLast() - O(1)
Peek => get the data part of last node

if(head == null) => list is empty => stack is empty

head => 44 33 22 11

OR

Push => addFirst() - O(1)
Pop => deleteFirst() - O(1)
Peek => get the data part of first node

Lab Work => to implement dynamic stack

- stack is also used in / to implement expression conversion algorithms and evaluation algorithms.

What is an expression ?
Combination of operands and operators
- there are 3 types of expression
1. infix expression : a+b
2. prefix expression : +ab
3. postfix expression : ab+

infix expression => a*b/c*d+e-f*g+h

Lab Work : Implement algo to convert given parenthesized infix expression into its equivalent prefix expression.

Lab Work : Implement an algo for postfix evaluation (if an expression contains multi-digit operands).
- vector of strings to store infix & postfix expression
- stack of integers, while pushing an operand of type string we need to convert into its equivalent int.

+ Queue : it is a **linear/basic data structure** which is **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** and elements can be deleted from it which is at **front end.**

- in this collection/list, **element which was inserted first can be deleted first**, so this list works in **first in first out manner** or **last in last out manner**, hence queue is also called as **fifo list / lilo list.**
- On Queue data structure basic 2 operations can be performed in **O(1)** time:
1. Enqueue => insert/add an element into the queue from rear end.

2. Dequeue => delete/remove an element from the queue which is at front end.

- there are 4 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.

4. double ended queue (deque) => it is a type of queue in which elements can added as well as deleted from both the ends.
- on deque we can perform basic 4 operations in **O(1)** time:
i. push_back() => addLast()
ii. push_front() => addFirst()
iii. pop_back() => deleteLast()
iv. pop_front() => deleteFirst()
- deque can be implemented by using **DCLL.**
- further there are 2 types of deque:
1. input restricted deque => in this type of deque elements can be added into it only from one end, whereas elements can be deleted from both the ends.

2. output restricted deque => in this type of deque elements can be added into it from both the ends, whereas elements can be deleted only from one end.

1. Enqueue => insert/add 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).
step-2: increment the value of rear by 1
step-3: insert an element into the queue from rear end
step-4: if(front == -1)
front = 0

2. Dequeue => delete/remove 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 an element from it).
step-2: increment the value of front by 1 [by means of incrementing value of front by 1 we are achieving deletion of an element from queue].

DAY-07:
- queue can be implemented by 2 ways:
1. static implementation of queue (by using an array)
2. dynamic implementation of queue (by using an linked list).

1. static implementation of queue (by using an array):

```
int arr[ 5 ];
int front;
int rear;
```

```
arr   : int []
front : int
rear  : int
```

Circular Queue:
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2

in a cir quueue => if front is at next pos of rear => queue full
front == (rear + 1)%SIZE

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


rear++;
rear = rear + 1;

rear = ( rear + 1 ) % SIZE

for rear=0 => rear=(rear+1)%SIZE = (0+1)%5 = 1%5 = 1
for rear=1 => rear=(rear+1)%SIZE = (1+1)%5 = 2%5 = 2
for rear=2 => rear=(rear+1)%SIZE = (2+1)%5 = 3%5 = 3
for rear=3 => rear=(rear+1)%SIZE = (3+1)%5 = 4%5 = 4
for rear=4 => rear=(rear+1)%SIZE = (4+1)%5 = 5%5 = 0
```

2. dynamic implementation of queue (by using an linked list : DCLL).

- if list is empty => queue is empty
- there is no queue full condition for dynamic queue

Enqueue => addLast()
Dequeue => deleteFirst()

OR
head => 44

Enqueue => addFirst()
Dequeue => deleteLast()

Lab Work:
1. implement dynamic queue
2. implement priority queue by using dcll

Priority Queue by using Linked List:
searchAndDelete()

```
class Node{
    int data;
    Node next;
    Node prev;
    int priorityValue;
}
```

Basic Data Structures: comfortable
Advanced Data Structures:
- tree (can be implemented by using an array as well as linked list).
- binary heap (array implementation of a tree)
- graph (array & linked list)
- hash table (array & linked list)
- merge sort & quick sort

+ Tree:
+ tree terminologies:
root node
parent node/father
child node/son
grand parent/grand father
grand child/grand son
ancestors => all the nodes which are in the path from root node to that node

- further restrictions can be applied on binary tree to mainly to achieve addition, deletion and searching operations effciently expected in O(log n) time => binary search tree can be formed.

	Addition	Deletion	Searching
Array	O(n)	O(n)	O(log n)
Linked List	O(1)	O(1)	O(n)
BST	O(log n)	O(log n)	O(log n)

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

While adding node into the BST => We need to first find/search its appropriate position in a BST and after that we can add node at that position.

- there are basic two tree traversal methods:
1. bfs (breadth first search) traversal / levelwise traversal:
- traversal always starts from root node
- and nodes in a bst gets visited levelwise from left to right.

2. dfs (depth first search) traveral
- under dfs traversal further there are 3 ways by which tree can be traversed:
i. Preorder (V L R) :
- start traversal always from root node
- first visit cur node, then visit its left subtree and we can visit right subtree of any node only after either visitng its whole left subtree or left subtree is empty.
- in this traversal, root node always gets visited first, and this property remains recursively true for each subtree.

ii. Inorder (L V R) :
- start traversal always from root node
- in this traversal, we can visit any node only after visiting its whole left subtree or its left subtree is empty.
- in this traversal, all the nodes gets visited always in a an ascending sorted order.

iii. Postorder (L R V):
- start traversal always from root node
- in this traversal, we can visit any node only after visiting its whole left subtree as well as right subtree or its left subtree and right subtree are empty.
- in this traversal, root node always gets visited last, and this property remains recursively true for each subtree.

Lab Work : Implement nonRecPostOrder() algorithm/method.

Lab Work : implement recusrive addNode() function for BST.

 # DAY-08:
- queue : linear queue & circular queue
- tree : concept & definition
- tree terminologies
- addNode into the BST

inorder successor
inorder predecessor

- In a BST, inorder successor of any node usually exists/found at left end in its right subtree if it is is having right subtree.

- tree traversal algorithms:
preorder, postorder & inorder by using rec & non-rec way
dfs traversal & bfs traversal
searching on BST
deletion of a node in a BST.
calculation of height of BST.

Todays:
quick sort
m ort
tree concepts: complete binary tree, avl tree, balanced BST, balace factor, threaded binary treem multi-way tree, b-tree & b+ tree, binary heap
 # partitioning
i=left;
j=right;
pivot = arr[left];

/* shift ele's which are smaller than pivot towards left hand side, and ele's which are greater than pivot shift towards right side.*/

```
while( i < j ){

    while( i <= right && arr[ i ] <= pivot )
        i++;//goto the next ele

    while( arr[ j ] > pivot )
        j--;//goto the prev ele

    //if i & j have not crossed then swap them
    swap(arr[ i ], arr[ j ]);
}
//swap pivot ele with jth pos ele
swap( arr[ left ], arr[ j ] );
```

- in quick sort, worst case may occurs either array ele's are already sorted or present exactly in reverse order, this condition rarely occurs.

```
[ 60 50 40 30 20 10 ]
pass-1: [ 50 40 30 20 10 ] 60 [ RP ]
pass-2: [ 40 30 20 10 ] 50 [ RP ]
pass-3: [ 30 20 10 ] 40 [ RP ]
pass-4: [ 20 10 ] 30 [ RP ]
pass-5: [ 10 ] 20 [ RP ]
```

n * n => n^2

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

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

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

pass-4: pivot
[ 40 50 60 ]
```

n^2

if pivot element gets selected as mid pos ele, then there are very rare chances to occurs worst case.

```
[ 60 50 40 30 20 10 ]
pass-1: [ 10 20 30 ] 40 [ 60 50 ]
pass-2: [ 10 ] 20 [ 30 ]
pass-3: [ 50 ] 60 [ RP ]
```

log n + log n
2 (log n) => O(n log n)

merge sort on linked list : to merge two already sorted linked lists into a third list in a sorted manner

l1 => head =>

l2 => head =>

l3 => head => 10 -> 15 -> 20 -> 25 -> 30 -> 35 -> 40 -> 45 -> 50

google map app:
information about 1000's of cities and info between paths of those cities can be kept.

City

```
class City{
    String cityName;
    String cityCode;
    String pinCode;
    String state;
    ......
}
```

City class objects => vetices
Pune <====> Mumbai

Information of Path between Citie => Weight

```
class Path{
    String toCity;
    String fromCity;
    float distanceInKM;
    .....
}
```

 # DS_DAY-11:
+ Graph Traversal Algorithms:
1. dfs (depth first search) traversal - stack
2. bfs (breadth first search) traversal - queue

- graph can be a tree but tree cannot be a graph.
- subgraph of a graph which can be formed by means of removing one or more edges from it in such a way that it should remains connected and do not contains a cycle. Such a subgraph of a graph is referred as spanning tree of a given graph.

Hash table: