Course Name : eDMC May-2021
Module Name : Algorithms & Data Structures (Using Java)

Lecture Hrs: 32 + Lab Hrs: 28 = Total 60 Hrs.
Theory Exam : 40 %
Lab Exam : 40 %
Internal Exam : 20 %
Total Evaluation : 100

DAY-01
+ Introduction:
Q. Why there is a need of data structures?
- there is a need of data structures to achieve 3 things in programming:
1. efficiency
2. abstraction
3. reusability

Q. What is 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, sorting, searching, traversal etc... can be performed efficiently.
- data structures is a programming concept which is followed in all programming languages.

- we want to store marks of 100 students:
int m1, m2, m3, m4, m5, ..., m100;//sizeof(int)=4 bytes => 100*4=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 data elements gets stored into the memory at contiguos locations.

Array notation
=> compiler converts array notation into the pointer notation:

arr[0] => *(arr + 0) => *(100 + 0) => *(100) => 10
arr[1] => *(arr + 1) => *(100 + 1) => *(104) => 20
.
.
arr[i] => *(arr + i)
- to maintain link between array elements is the job of compiler.

```
int main( void )
{
```

```
	int arr[ 100 ];

	accept_array_element( arr );
}
```

- we want to store information/records of an employee:
int emp_id
char emp_name[32]
float salary

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

structure: it is a basic/linear data structure, which is a collection/list of logically related similar and disimmilar type of data elements gets stored into the memory collectively as single entity/record.

- there are 2 types of data types:
1. primitive: char, int, float, double & void
2. non-primitive: array, structure, pointers, enum, union, boolean etc...
structure/class => abstract data type

- as we canot combine/collect data members and member functions inside a structure, hence class has been designed.

```
class employee{
private:
	//data members
	int empid;
	String name;
	float salary;

public:
	//member functions: methods
	employee( );
	//getter functions
	//setter functions
	//facilitators
	~employee( );
};
```

employee e1;
employee e2;

struct employee e2;
struct student s1;

abstract 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 and hence can be accessed linearly.
- array
- structure & union
- class
- 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. heirarchical manner) and hence can be accessed non-linearly.
- tree (hierachical manner)
- graph
- binary heap
- hash table (associative manner).

- to learn data structures is not to learn any programming language, it is nothing but to learn an algorithms.

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

Q. What is an algorithm?
An algorithm is a finite set of instructions written in any human understandable language like english, if followed, acomplishesh given task.

- An algorithm is a template of a program.
- Program is an implementation of an algorithm.

Q. What is a Pseudocode?
An algorithm is a finite set of instructions written in any human understandable language like english with some programming constraints, if followed, acomplishesh given task.
- pseudocode is a special form of an algorithm.

Problem: to do sum of array elements

Algorithm:

to do sum of array elements: => human user

step-1 : intially take sum as 0.

step-2 : start traversal of an array sequentially from first element max till last element and add each array element into the sum.

step-3 : return final sum

Pseudocode: => programmer users

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

- there are 2 types of algorithms:

1. iterative algorithm

Example:

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

	return sum;
}
```

Example:

```
for( exp1 ; exp2 ; exp3 ){
        statement/s
}
```

- iterative approach:

exp1 => initialization

exp2 => termination codition

exp3 => modification

2. recursive algorithm

Example:

```
Algorithm RecArraySum( A, size, index ){
        //base condition
        if( index == size )
                return 0;

        return ( A[ index ] + RecArraySum( A, size, index+1 ) );
}
```

Q. What is a Recursion?

- it is a process in which we can call any function from inside the same function OR function calls itself within itself, such function is called as recursive function.
- recursion is a process can be defined in terms of itself repeatedely.

Example:

```
public class Array{

        public static int factorial(int num){
                //base condition
                if( num == 0 )
                        return 0;

                return ( num * factorial(num-1) );
        }

        public static void main( String [] args ){
                res = factorial(5);//initialiazation
                //factorial:
                //calling function => main()
                //called function => factorial()

                printf("factorial = "+res);
                return 0;
        }
}
```

- factorial() function is a recursive function/algorithm:
- while writing recursive function/algo, we need to take care about 3 things:
1. **initializtion** – we have to take care about it at the time of first time function calling to the recursive function.

2. **base condition** – it decides when recursive function calling should gets stop.
- we need to take care about it at the begining of definition of recusrive function.

3. **modification** – we need to take care about modification at the time recursive function call
(recursive function call => it is a function call for which calling function & called function are same).

- there are 2 types of recusrive functions:
1. **tail recursive:** it is a type of recursive function in which recursive function call is the last executable statement in it.

```
void fun(int n)
{
	statement/s
	fun(--n);//recursive function call
}
```

2. **non- tail recursive:** it is a type of recursive function in which recursive function call is not the last executable statement in it.

```
void fun(int n)
{
	statement/s
	fun(--n);//recursive function call
	statement/s
}
```

Points:
- recursive algorithms are easy to implement than iterative algorithms
- recursive algortihms takes extra space

- An algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many (i.e. more than one) solutions
Searching => to search given element i.e. key element into a collection/list of elements.
There are 2 solutions/algorithms for searching:
1. linear search
2. binary search

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

There are more than one solutions/algorithms for sorting:
1. selection sort
2. bubble sort
3. insertion sort
4. merge sort
5. quick sort
etc....

- When one problem has many solutions/algorithms, there is need to decide an efficient one, and to decide an efficiency of algorithms we need to do their analysis.
- **analysis of an algorithm** is work of calculating/determining the total amount of **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 algorithms:
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.

Pune ===> Mumbai
- mulitple paths may exists between 2 cities, in this case we need to select an efficient/optimized path, and to decide which path is optimum there are some factors:
- distance in km
- travelling time
- mode of travel
- traffic status
- speed
etc...

Q. What is a Function Activation Record / Stack Frame?
- When any function gets called, its entry gets created onto the stack referred as **function activation record / stack frame.**
- per function call stack frame gets created/pushed onto the stack and upon function completion stack frame of that function gets popped/destroyed from the stack.
- FAR/Stack Frame contains: formal parameters, local variables, return addr, old frame pointer etc....

Example:

```
int addition( int n1, int n2)//n1 & n2 => formal paramters
{
        int res;//local variable
        res = n1 + n2;
        return res;
}
```

return addr: addr of next instruction in its calling funtion, and hence when any function completes its execution control of execution gets tranfers to its calling function.

old frame pointer contains an addr prevoius stack frame.

 # Space Complexity:

Space complexity of an algorithm = **code space** (i.e. space required for instructions) + **data space** (i.e. space required for constants, simple variables & instance vars) + **stack space** (space required for function activation records) **stack is applicable/considered only in a recursive algorithm.**

- Space complexity has two components:

1. fixed component : it contains **code space** and **space required for constants & simple variables.**

2. variable component: it contains **space required for instance variables and stack space.**

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

Example:

Iterative Algorithm

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

        return sum;
}
```

S = Code Space + Data Space

if size of an array = 5 => no. of instructions in an algo going to remain fixed
if size of an array = 10 => no. of instructions in an algo going to remain fixed
if size of an array = 100 => no. of instructions in an algo going to remain fixed

if size of an array = **n** => no. of instructions in an algo going to remains fixed/ constant.
code space = fixed component.

Data space (Sp) = space required for constants + space required for simple vars + space required for instance vars
Sp = **2 units** (1 unit for 0 & 1 unit for 1) + **3 units** (1 unit = A, 1 unit = sum & 1 unit = index) + **n units.**
= 5 + n => n + 5

if size of an array = 5 => 10 bytes / 20 bytes => **5 units**
if size of array = 10 => 20 bytes / 40 bytes => 10 units

if size of array = **n** => **n units**
size => instance variable => value of this variable, no. of units of memory required for this ar gets decided during runtime depends on size of an array,.

S = C (Code Space) + Sp (Data Space)
S >= Sp
S >= (n+5)
S >= n
S ~= O(n)

=> space required for an algo depends on size of an array.

Class - state
Object -- instance/behaviuor

employee emp1, emp2, emp3;

To calculate space complexity of recursive algorithm:
Example:

```
Algorithm RecArraySum( A, size, index ){
        //base condition
        if( index == size )
                return 0;

        return ( A[ index ] + RecArraySum( A, size, index+1 ) );
}
```

S = Code Space + Data Space + Stack Space
Code Space => as for any input size array no. of instructions in an algo are going to remains fixed => constant

Data Space = Sp
1 unit (for constant 0) + 3 units (for simple vars) => **4 units**

recursive function gets called **(n+1) no. of times and hence (n+1) function activation records gets created onto the stack**
Stack Space = 4 * (n+1) = 4n + 4

S = C + Sp + Stack Space
S >= 4n + 4
S >= 4n

 # Time Complexity:
Time Complexity = Compilation Time + Execution Time
- time complexity has two components:
1. fixed component: compilation time
2. execution time: variable component

- **execution time** of an algorithm depends on **instance characteristics** of it i.e. **input size of an array.**

Compiler => it is an application program that reads high level programming language code at once from first line till last line and it converts it into low level programming language code.

Iterative Algorithm

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

        return sum;
}
```

for any input size array in this algo compilation time going to remains fixed.

If size of an array = 5 , statement/s inside for loop executes 5 no. of times.
If size of an array = 10 , statement/s inside for loop executes 10 no. of times.
.
.
If size of an array = **n** , statement/s inside for loop executes **n** no. of times.

Scenario-1:
Machine-1: Pentium-4 => array size i.e. n = 100
Machine-2: Core i5 => array size i.e. n = 100

Scenario-2:

Machine-1: Core i5 => array size i.e. n = 100 : system is fully loaded with other processes as well

Machine-2: Core i5 => array size i.e. n = 100: system is not fully loaded with other processes

- execution time is not only depends on input size of an array it also depends on hardware configuration on which algo is running as well as an environment in which it is running, and hence to calculate time complexity calculation of compilation time and excution time is not a good practice, and hence time complexity and space complexity of an algorithms can be calculated **mathematically** without implementing algorithms in any programming langauges, this kind of analysis is referred **asymptotic analysis.**
- this type of analysis is used to do analysis of an algorithms roughly and we can decide which algo is an efficient one.

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

- time complexity of any algo **cannot be less than its best case time complexity,** hence best case time complexity is also referred as **asymptotic lower bound.**

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

- time complexity of any algo **cannot be more than its worst case time complexity,** hence worst case time complexity is also referred as **asymptotic upper bound.**

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

- time complexity of any algo **cannot be less than its best case time complexity and more than its worst case time complexity i.e. it is tighly bounded between lower bound and upper bound** hence average case time complexity is also referred as **asymptotic tight bound.**

Traversal on array => to visit each array element sequentially from first element max till last element.
=> traversal of an array is also called as to scan array

- In asymptotic analysis we have to use some notations and follow some rules:

- There are 3 asymptotic noations:
1. Big Omega (Ω) - this notation used to represent best case time complexity of an algorithm i.e. asymptotic lower bound.

2. Big Oh (O) - this notation used to represent worst case time complexity of an algorithm i.e. asymptotic upper bound.

3. Big Theta (θ) - this notation used to represent an average case time complexity of an algorithm i.e. asymptotic tight bound.

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

 # Searching Algorithms:
1. Linear Search/Sequetial Search:
step-1: accept key from user
step-2: traverse an array and compare value of the key with each array element sequentiall till key is matches with any array element or max till last element. If key is matches with any of array element returns true otherwise return false.

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

        return false;//key is not found
}
```

 # best case occurs <u>if key is found at very first position:</u> O (1)
if size of an array = 10 => no. of comparisons are = 1
if size of an array = 20 => no. of comparisons are = 1
if size of an array = 50 => no. of comparisons are = 1
.
.
if size of an array = n => no. of comparisons are = 1

 # worst case occurs <u>if either key is found at last position or key does not exists:</u> O (n)
if size of an array = 10 => no. of comparisons are = 10
if size of an array = 20 => no. of comparisons are = 20
.
.
if size of an array = n => no. of comparisons are = n

DAY-01: Lab Work
1. Implement linear search algorithm (in java) by using recursive and non-recusrsive approaches.

DAY-02:
Implementation of Linear Search
Binary Search: algorithm, analysis and implementation
Sorting Algorithms: Selection Sort, Bubble Sort & Insertion Sort.

2. Binary Search:
- to apply binary search, prerequisite is array elements must be sorted.
- by means of calculating mid position, big size array gets divided logically into two subarray's => left subarray & right subarray
in every iteration:
for left subarray => value of left remains as it is, right = mid-1
for right subarray => value of right remains as it is, left = mid+1

n = 1000

iteration-1: search space = n : 1000,
[1 2 3 ..... 1000]
mid = 500
[1 2 3 ..... 499] 500 [501..... 1000]

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

iteration-3: search space = n / 4 : 250
[1 .... 249]
mid = 125
[1.... 124] 125 [126.... 249]

iteration-4: search space = n / 8 : 125

.
.
.
- in this algo, in each iteration, 1 comparison takes place and search space is getting reduces by half.
T(n) = T(n / 2) + 1 ..... equation

for n = 1,
T(n) = O(1) .....trivial case

for n > 1,
$T(n) = T(n/2) + 1$ .....(I) => $T(n) = T(n/2^1) + 1$

we are using **substitution method** to calculate time complexity of binary search.
- to get the value of $T(n/2)$, put $n = n/2$ in equation-I, we get

$T(n/2) = T((n/2)/2) + 1$

=> $T(n/2) = T(n/4) + 1$ ..... (II)

by substitutoing value of T(n/2) in eq – I we get,
$T(n) = [T(n/4) + 1] + 1$

$T(n) = T(n/4) + 2$ ..... (III) => $T(n) = T(n/2^2) + 2$

- to get the value of n/4, put n = n/2 in eq -II
$T((n/2)/2) = T((n/2)/4) + 1$
$T(n/4) = T(n/8) + 1$

$T(n/4) = T(n/8) + 1$ ..... (IV)

by substituting value of T(n/4) in equation III, we get

$T(n) = [T(n/8) + 1] + 2$
$T(n) = T(n/8) + 3$

$T(n) = T(n/8) + 3$ ..... (V) => $T(n) = T(n/2^3) + 3$

let say after k iterations:
$T(n) = T(n/2^k) + k$ ..... (VI)

lets assume that it is the last iteration,
assume => $n = 2^k$

=> $n = 2^k$
=> $\log n = \log 2^k$ ..... by taking log on both sides
=> $\log n = k \log 2$
=> $\log n = k$ .....(log 2 ~= 1)
=> $k = \log n$

by substituting value $n = 2^k$ and $k = \log n$ in equation VI we get,
=> $T(n) = T(n/2^k) + k$
=> $T(n) = T(2^k/2^k) + \log n$
=> $T(n) = T(1) + \log n$
=> $T(n) = O(\log n)$

Rule: if any algorithm follows divide-and-conquer approach, then time complexity of that algo gets in terms of log.

Lab Work: to implement binary using recursive approach

Sorting Algorithms:
+ Sorting: to arrange data elements in a collection/list either in an ascending order or in a descending order.
1. Seclection Sort
2. Bubble Sort
3. Insertion Sort

Rule: if running time of an algorithm is having a polynomial, then in its time complexity only leading term gets considered.

1. Seclection Sort:

total no. of comparisons = (n-1) + (n-2) + (n-3) + ... => n(n-1) / 2
total no. of comparions are = n(n-1) / 2
T(n) => n(n-1) / 2
T(n) => (n^2 – n) / 2
T(n) => O ((n^2 – n) / 2)
T(n) => O (n^2 – n) ... by neglecting divisive constant.
T(n) => O (n^2)

prime number => i = 2 to n/2 => O(n)

```
for( i = 1 ; i <= 5 ; i++ ){
        statement/s
}
```
=> O(1)

```
for( i = 1 ; i <= n ; i++ ){//n = instance var
        statement/s
}
```

=> O(n)

```
for( i = 1 ; i <= n ; i += 2 ){//n = instance var
        statement/s
}
```

OR

```
for( i = n ; i > 0 ; i -= 2 ){//n = instance var
        statement/s
}
```

```
n=20        n=40
i=20        i=40
i=18        i=38
i=16        i=36
.           .
.           .
i=0         .
            .
            .
            i=0
```

=> O(n)

```
for( i = 1 ; i <= n ; i++ )//outer for loop
{
        //n times
        for( j =  1 ; j <= n ; j++ )//inner for loop
        {
                //statement/s      => n( n ) times => n^2
        }
}
```

for each iteration of outer for loop all iterations of inner for loop takes place.
Let say n = 5
for i = 1 => j =1,2,3,4,5,6
for i = 2 => j =1,2,3,4,5,6
for i = 3 => j =1,2,3,4,5,6
for i = 4 => j =1,2,3,4,5,6
for i = 5 => j =1,2,3,4,5,6
for i = 6 => outer for loop gets breaked
- if any algorithm contains nested loops, then time complexity of that algo will be the time taken by statement/s which are inside innermost loop.

```
for( i = 1 ; i <= n ; i++ )//outer for loop
{
        //n times
        for( j =  1 ; j <= n ; j++ )//inner for loop
        {
                for( k = 1 ; k <= n ; k++ )
                {
                        //statement/s      => n*n*n times => n^3
                }
        }
}
```

Lab Work => convert array operations (searching & sorting) program as a menu driven program.

- if specific case of time complexity of any algo is not mentioned explicitly, bydefault we need to consider an avg case.

+ features of sorting algorithms:
1. inplace => if sorting algo do not takes extra space to sort n no. of elements in a collection/list then it is referred as inplace.

2. adaptive => if sorting algo works efficiently for already sorted input sequence then it is reffered as adaptive.

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

Example:
10 40 20 30 10' 50

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

OR
after sorting:
10' 10 20 30 40 50 => not stable

Lab Work => to check sorting algorithms are stable or not on paper only with different examples.

2. Bubble Sort:

total no. of comparisons = (n-1) + (n-2) + (n-3) + ... => n(n-1) / 2
total no. of comparions are = n(n-1) / 2
T(n) => n(n-1) / 2
T(n) => (n^2 – n) / 2
T(n) => O ((n^2 – n) / 2)
T(n) => O (n^2 – n) ... by neglecting divisive constant.
T(n) => O (n^2)

for it=0 => pos = 0, 1, 2, 3, 4, 5
for it=1 => pos = 0, 1, 2, 3, 4
for it=2 => pos = 0, 1, 2, 3
.
.

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

best case: if array elements are already sorted

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

if all pairs are in order => array elements are already sorted
if all pairs are in order => there is no need of swapping for a single pair => array elements are already sorted => no need to go to second iteration.

only one iteration takes place and total no. of comparisons = (n-1)
T(n) = O(n – 1)
T(n) = O(n) => Ω(n).

DAY-03:
3. Insertion Sort:

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

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

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

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

iteration-1:
10 20 30 40 50 60
key=20

10 20 30 40 50 60 => no. of comparisons = 1

iteration-2:
10 20 30 40 50 60
key=30

10 20 30 40 50 60 => no. of comparisons = 1

iteration-3:
10 20 30 40 50 60
key = 40

10 20 30 40 50 60 => no. of comparisons = 1

iteration-4:
10 20 30 40 50 60

10 20 30 40 50 60 => no. of comparisons = 1

iteration-5:
10 20 30 40 50 60
key=60

10 20 30 40 50 60 => no. of comparisons = 1

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

T(n) = Ω(n).

Linked List:

Q. Why Linked List ?

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

2. **array is static** i.e. size of an array is fixed, we cannot either grow or shring size of an array during runtime.

3. addition & deletion operations on an array are not efficient as it takes O(n) time
- while adding ele into an array we need to shift elements towards right side 1-1 pos, whereas while deleting ele from an array we need to shift elements towards left side 1-1 pos

- to overcome last two limitations of an array data structure, **linked list** data structure has been designed.

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

- in a linked list element is also called as a node.
- there are 2 types of linked list
1. **singly linked list:** it is a type of linked list in which each element/node contains actual data and an addr of its next element/node.
(each node contain only one/single link => singly linked list).
- further there are 2 types of linked list:
i. singly linear linked list
ii. singly circular linked list

2. doubly linked list: it is a type of linked list in which each element/node contains actual data and an addr of its next element/node as well as an addr its prev node/element.
(each node contain two links => doubly linked list).
- further there are 2 types of linked list:
i. doubly linear linked list
ii. doubly circular linked list

- total 4 types of linked list are there:
i. singly linear linked list
ii. singly circular linked list
iii. doubly linear linked list
vi. doubly circular linked list

i. singly 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 two parts:
1. data part: contains actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
- last node points to null i.e. next part of last node contains null.

```
class Node{
	//data members
	int data;//4 bytes
	Node next; //null => 4 bytes

	Node( int data ){
		this.data = data;
		this.next = null;
	}
}
```

sizeof Node class object => 8 bytes

- we can apply/perform basic 2 operations on linked list data structure:
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 specific (inbetween) position

2. deletion : to delete/remove node from the linked list
- we can delete node from the linked list by 3 ways:
i. delete node from the linked list at first position
ii. delete node from the linked list at last position
iii. delete node from the linked list at spepcifc (in between) position

- traversal on a linked list => to visit each node in a linked list sequentially from first node max till last node.

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

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

In Linked List => make before break => always creates new links (links which are associated with newly created node) first and then only break old links.

ii. add node into the linked list at first position (slll):
- we can add as many as we want number of nodes into the slll at first position (i.e. linked is dynamic) in O(1) time.

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

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

Best Case : Ω(1) => if pos is the first position
Worst Case : O(n) => if pos is the last position
Average Case : θ(n) => if pos > 1 && pos < nodes_cnt

if(head == null) ==> list is empty
if(head != null) ==> list is not empty

- as head always referes to the first node in a list
head.data => data part of first node
head.next => next part of first node

if(head.next == null) => list contains only one node
if(head.next != null) => list contains more than one node

i. delete node from the linked list (slll) at first position
- we can delete node from slll which is at first position in O(1) time
Best Case : Ω(1)
Worst Case : O(1)
Average Case : θ(1)

ii. delete node from the linked list (slll) at last position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if list contains only one node
Worst Case : O(n)
Average Case : θ(n)

iii. delete node from the linked list (slll) at specific position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if pos == 1
Worst Case : O(n) => if pos == nodes_cnt
Average Case : θ(n) => if pos is in between position

+ limitations of slll:
- prev node of any node cannot be accessed from it
- list can be traverse only in a forward direction
- addLast( ) & deleteLast( ) operations are not efficient as it takes O(n) time.
- any node cannot be revisited => to overcome limitations of slll, scll has been designed.

Lab Work :
1. Implement deletNodeAtSpecificPosition( ) function in SLLL.
2. Implement SLLL with head as well tail pointers.
Head pointer always contains addr of first node, whereas tail pointer always contains an addr of last node.

Wrapper function => function can be used only for calling another function.
- Which contains only logic to give call to another function

- Reverse the SLLL:
1. we can traverse linked list only once
2. we can traverse slll only in a forward direction
3. we can start traversal of slll only from first node
4. prev node of any node cannot be accessed from it

Lab work: implement reverse linked list functionality by using recursion.

searchAndDelete( )
LinkedList : Collection => remove(element)
removeFirst( )
removeLast( )
remove(element); => searchAndDelete(element)

searchAndDelete( ) => this function is also used to implement priority queue by using linked list.

SCLL:
ii. singly 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 two parts:
1. data part: contains actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
- last node points to first node i.e. next part of last node contains an addr of first node.

- we can apply/perform basic 2 operations on linked list (scll) data structure:
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 specific (inbetween) position

2. deletion : to delete/remove node from the linked list
- we can delete node from the linked list by 3 ways:
i. delete node from the linked list at first position
ii. delete node from the linked list at last position
iii. delete node from the linked list at spepcifc (in between) position

- all the operations/algorithms which we applied on slll can be applied on scll as it is, except we need to maintain next part of last node always.

i. add node into the linked list at last position (scll):
- we can add as many as we want number of nodes into the slll at last position (i.e. linked is dynamic) in O(n) time.

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

ii. add node into the linked list at first position (scll):
- we can add as many as we want number of nodes into the slll at first position (i.e. linked is dynamic) in O(n) time.

Best Case : Ω(1)
Worst Case : O(n)
Average Case : θ(n)

iii. add node into the linked list at specific (in between) position (scll):
- we can add as many as we want number of nodes into the slll at specific position (i.e. linked is dynamic) in O(n) time.

Best Case : Ω(1) => if pos is the first position
Worst Case : O(n) => if pos is the last position
Average Case : θ(n) => if pos > 1 && pos < nodes_cnt

scll:
if(head == null) ==> list is empty
if(head != null) ==> list is not empty

- as head always referes to the first node in a list
head.data => data part of first node
head.next => next part of first node

if(head.next == head) => list contains only one node
if(head.next != head) => list contains more than one node

i. delete node from the linked list (scll) at first position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1)
Worst Case : O(n)
Average Case : θ(n)

ii. delete node from the linked list (scll) at last position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if list contains only one node
Worst Case : O(n)
Average Case : θ(n)

iii. delete node from the linked list (scll) at specific position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if pos == 1
Worst Case : O(n) => if pos == nodes_cnt
Average Case : θ(n) => if pos is in between position

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

- to overcome limitations of singly linked list (i.e. slll as well as scll), doubly linked list has been designed.

iii. 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 three parts:
1. data part: contains actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
3. pointer part (prev): contains an addr of its prev node
- next part of last node and prev part of first node points to null i.e. it contains null.

- we can apply/perform basic 2 operations on linked list (dlll) data structure:
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 specific (inbetween) position

2. deletion : to delete/remove node from the linked list
- we can delete node from the linked list by 3 ways:
i. delete node from the linked list at first position
ii. delete node from the linked list at last position
iii. delete node from the linked list at spepcifc (in between) position

- all the operations/algorithms which we applied on slll can be applied on dlll as it is, except we need to maintain forward link as well as backward link of each node (i.e. we need to maintains next part as well as prev part of each node while addition & deleltion operation.)

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

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

ii. add node into the linked list at first position (dlll):
- we can add as many as we want number of nodes into the dlll at first position (i.e. linked is dynamic) in O(1) time.

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

iii. add node into the linked list at specific (in between) position (dlll):
- we can add as many as we want number of nodes into the slll at specific position (i.e. linked is dynamic) in O(n) time.

Best Case : Ω(1) => if pos is the first position
Worst Case : O(n) => if pos is the last position
Average Case : θ(n) => if pos > 1 && pos < nodes_cnt

dlll:
if(head == null) ==> list is empty
if(head != null) ==> list is not empty

- as head always referes to the first node in a list
head.data => data part of first node

head.next => next part of first node

if(head.next == null) => list contains only one node
if(head.next != head) => list contains more than one node

i. delete node from the linked list (dlll) at first position
- we can delete node from slll which is at first position in O(1) time
Best Case : Ω(1)
Worst Case : O(1)
Average Case : θ(1)

ii. delete node from the linked list (dlll) at last position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if list contains only one node
Worst Case : O(n)
Average Case : θ(n)

iii. delete node from the linked list (dlll) at specific position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if pos == 1
Worst Case : O(n) => if pos == nodes_cnt
Average Case : θ(n) => if pos is in between position

DAY-05:
+ advantages of dlll:
- doubly linear linked list can be traversed in both forward as well as in a backward direction.
- prev node of any node can be accessed from it.

+ limitations of dlll:
- addLast() & deleteLast() operations are not efficient as it takes O(n) time.
- traversal can be start only from first node.
To overcome limitations of dlll, dclll has been designed.

iv. 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 **three** parts:
1. data part: contains actual data of any primitive/non-primitive type
2. pointer part (next): contains an addr of its next node
3. pointer part (prev): contains an addr of its prev node
- next part of last node contains an addr of first node and prev part of first node contains an addr of last node.
i.e. last node's next part points to first node and first node's prev part points to last node.

if(head == null) => list is empty
if(head != null) => list is not empty

if(head == head.next) => list contains only one node
if(head != head.next) => list contains more than one nodes

- all the operations/algorithms which we applied on dlll can be applied on dcll as it is, except we need to maintain next part of last node and prev part of first node.

i. add node into the linked list at last position (dcll):
- we can add as many as we want number of nodes into the dcll at last position (i.e. linked is dynamic) in O(1) time.

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

ii. add node into the linked list at first position (dcll):
- we can add as many as we want number of nodes into the dlll at first position (i.e. linked is dynamic) in O(1) time.

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

iii. add node into the linked list at specific (in between) position (dcll):
- we can add as many as we want number of nodes into the slll at specific position (i.e. linked is dynamic) in O(n) time.

Best Case : Ω(1) => if pos is the first position
Worst Case : O(n) => if pos is the last position
Average Case : θ(n) => if pos > 1 && pos < nodes_cnt

i. delete node from the linked list (dcll) at first position
- we can delete node from slll which is at first position in O(1) time
Best Case : Ω(1)
Worst Case : O(1)
Average Case : θ(1)

ii. delete node from the linked list (dcll) at last position
- we can delete node from slll which is at first position in O(1) time
Best Case : Ω(1) => if list contains only one node
Worst Case : O(1)
Average Case : θ(1)

iii. delete node from the linked list (dcll) at specific position
- we can delete node from slll which is at first position in O(n) time
Best Case : Ω(1) => if pos == 1
Worst Case : O(n) => if pos == nodes_cnt
Average Case : θ(n) => if pos is in between position

Lab Work:
- addNodeAtFirstPosition( )
- deleteNodeAtFirstPosition( )
- deleteNodeAtLastPosition( )

=> dcll is the most efficient form/type of a linked list

LinkedList => DCLL

+ Methods of LinkedList Collection:
1. addFirst(element) - add element into the list at first position
2. addLast(element) -
3. add(n, element) - add element/node into the list at n^{th} position
4. add( ) - add node into the list at last position
5. removeFirst( )
6. removeLast( )
7. remove(n) - delete n^{th} element/node from the linked list
8. remove(element) - remove element from the linked list – like search and delete
9. size( ) - get the size of the linked list i.e. no. of nodes in the linked list
10. get(n) - get the value of an n^{th} element/node
11. element( ) - get the value of head element i.e. first element in a linked 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, this list works in last in first out manner / first in last out manner, and hence stack is also called as LIFO list/FILO list.
- we can perform basic 3 oeprations onto the 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 from the stack which is at top end (without Push/Pop).

- Stack data structure can be implemented by 2 ways:
1. Static Stack (by using an array)
2. Dynamic Stack (by using linked list)

Array => Static
Linked List => Dynamic
Stack => Static/Dynamic => As Stack data structure can be static as well as dynamic depends on by using which data structure i.e. either an array or linked list it is implemented => Adaptor Classes

Static Stack (by using an array):

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 ele onto it).
step-2: increment the value of top by 1
step-3: insert ele onto the stack from 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 we can pop ele from it).
Step-2: decrement the value of top by 1 (by means of 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 from the stack which is at top end (without Push/Pop).
step-1: check stack is not empty (if stack is not empty then only we can pop ele from it).
step-2: get the value of an element from the stack which is at top end (without incrementing/decrementing top).

 # Dynamic Stack (by using linked list: dcll)
push => addLast( )
pop => removeLast( )

LIFO

head => 10 20 30 40

OR
push => addFirst( )
pop => removeFirst( )

 # Applications of an Array:
- in any program, if we want to collect logically related similar type of data element and we know in advance size of that list/collection, in such case we can go for an array
- an array is used to implemet basic data structures like stack & queue
- an array is also used to implement advanced data structures like tree, binary heap, graph etc...

 # Applications of Linked List:
- in any program, if we want to collect logically related similar type of data element and we dont know in advance size of that list/collection, in such case we can go for linked list.
- linked list is used to implemet basic data structures like stack & queue
- linked list is also used to implement advanced data structures like tree, graph, hash table etc...

 # Applications of Stack data structure:
- in any program, if collection/list of elements should works in a last in first out manner then we can go for stack.
- to control flow of an execution of a program, an OS internally maintains stack
- in recursion internally OS maintains stack
- undo & redo functionalities of an OS has been implemented by using stack
- stack is used to implement expression conversion & expression evalution algorithms
- stack is used to implement advanced data structure algorithms like **dfs (depth first search) traversal** in a tree & graph.

 # Expression Conversion & Evalution Algorithms:
1. to convert given infix expression into its equivalent postfix
2. to convert given infix expression into its equivalent prefix
3. to evaluate postfix expression

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

 # DAY-06:
- postfix expression evaluation

+ Queue: It is a **basic/linear data structure**, which is **a collection/list of logically related similar type of data elements** in which element can be added into it from one end referred as **rear end**, and elements can be deleted from another end referred as **front end.**

- in this list/collection, element which was inserted first can be deleted first, so this list usually works in **first in first out / last in last out** manner, hence queue is also called as **fifo list / lilo list.**
- on queue data structure we can perform basic 2 operations in O(1) time:
1. enqueue: to insert an element into the queue from rear end
2. dequeue: to delete/remove an element from the queue which is at front end

- there are different 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 can be implemented by using LinkedList:
by using searchAndDelete( ) function

example:

head => | 10 : 3 | <=> | 15 : 1 | <=> | 20 : 4 | <=> | 30 : 2 |

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

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.
- on deque we can perform basic 4 operations in **O(1)** time

1. push_back : addLast( )
2. push_front : addFirst( )
3. pop_back : removeLast( )
4. pop_front : removeFirst( )

=> deque can be implemented by using **doubly circular linked list (LinkedList) / doubly linear linked list with head & tail pointers.**
- there are 2 types of deque:

1. input restricted deque : it is a type of deque in which elements can be deleted from both the ends, whereas elements can be added into it only from one end.

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

 # Implementation of a Linear queue & Circular Queue:
- queue can be implemented by 2 ways
1. static implementation (by using an array)
2. dynamic implementation (by using linked list)
=> stack & queue are called adaptor classes

1. static implementation (by using an array):

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

2. dequeue: to 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 element from it from front end).
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].

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

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

for rear=0, front = 1 => front is at next pos 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 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 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 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 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;
to increment the value of rear by 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 a queue (by using linked list: dcll):
enqueue : addFirst( )
dequeue : removeLast( )

OR

enqueue : addLast( )
dequeue : removeFirst( )

Applications of Queue:
- queue is used in a program, if in a collection/list of elements it should works in first in first out manner.
- queue is used to implement (os) kernel data structure like ready queue, job queue, waiting queue, message queue etc...
- queue is also used to implement os algorithms like fcfs cpu scheduling algo, priority cpu scheduling algo, fifo page replacement algo etc...
- queue is used to implement advanced data structure algorithms like bfs (breadth first search) traversal in tree & graph.

DS-PART-01 => Basic Data Structures

DS-PART-02 => Advanced Data Structures
- to implement advanced data structures and algorithms in it, prerequisite is one must be expert in basic data structures. [Array, Structure, Class, LinkedList, Stack, Queue].

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

+ tree terminologies:
- root node
- parent node/father
- child node/son
- grand parent/grand father
- grand child/grand son
- siblings: child nodes of same parent
- ancestors: all the nodes which are in the path from root node to that node are called as its ancestors.
- root node is an ancestor of all the nodes

- descedents: all the nodes which can be accessible from that node
- 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 – node which is not having any child/s
OR node having degree zero
- non-leaf node – node which is having any number of child node/s
Or node having non-zero degree

- level of a node = level of its parent node + 1
- if we assume level of root node = 0
- depth of a tree - max level of any node in a given tree is called as depth of a tree

- as tree is a dynamic data structure, tree can grow upto any level and any node can have any number of child nodes, due to this feature operations on it mainly like addition, deletion & searching becomes inefficient, and hence to achieve operations efficiently on tree, some restrictions can be applied on it and due to this there are different types of tree data structure.

- binary tree : it is a type of tree in which each node can have max two child nodes i.e. each node has either 0 OR 1 OR 2 no. of child node/s.

OR binary tree is a tree in which each node has degree either 0 or 1 or 2.

- binary tree is a set of finite number of elements has 3 subsets
1. root element
2. left subtree (may be empty)
3. right subtree (may be empty)

set = 0 no. of ele's => empty set/null set
set = 1 no. of ele => singular/singleton
set = >1 no. of ele's

- further restrictions can be applied on binary tree to achieve addition, deletion and searching operations effciently i.e. expected in O(log n) time, is referred as binary search tree.

- binary search tree (bst) : it is binary tree in which left child is always smaller than its parent and right child is always greater or equal to parent.

```
class Node{
	Node left;//it contains ref of left child
	type data;//actual data of any primitive/non-primitive type
	Node right;//it contains ref of right child
}
```

example:

```
class Node{
	Node left;//it contains ref of left child
	int data;
	Node right;//it contains ref of right child
}
```

size of Node class object = 12 bytes

- We can traverse tree (bst) by 2 ways
1. bfs (breadth first search) traversal/level wise traversal
2. dfs (depth first search) traversal
- further there 3 ways under dfs:
1. preorder (V L R):
2. inorder   (L V R):
3. postorder (L R V)

V => Visit cur node
L => Left Subtree
R => Right Subtree

1. preorder (V L R):
- start traversal of a tree from the root node
- first visit a node (root node of subtree), then visit its whole left subtree and then visit its right subtree.
- in this type of traversal, root node always gets visited first and this property remains same recursively for every subtree.

2. inorder   (L V R):
- start traversal of a tree from the root node
- in this traversal, first visit left subtree of a node, then visit the node and then only visit its right subtree.
- in this traversal, any node can be visited only if either its left subtree is already visited or empty.
- in this type of traversal, nodes gets visited in an ascending order

3. postorder (L R V)
- start traversal of a tree from the root node
- first visit left subtree a node, then visit its right subtree and then only visit that node.
 - in this traversal, any node can be visited only if either its left subtree and right subtree are already visited or empty.
- in this traversal, root node always gets visited last and this property remains same recursively for every subtree.

- BFS traversal is also called as level-wise traversal
- traversal starts from root node and all the nodes in a BST gets visited levelwise from left to right.

# Queue Collection Functions:
Poll => dequeue
offer => enqueue
isEmpty( ) => to check queue is empty or not

- bfs traversal
- dfs traversal
- preorder – recursive & non-recursive
- inorder – recursive & non-recursive
- postorder – recursive & non-recursive

addition : we need to first find appropriate pos to add node
deletion : we need to first search node and then we can delete it
searching : binary search => O(log n)

- addition, deletion & searching operations in bst takes O(log n) time.

height of a node = max(ht. of its left subtreem, ht. of its right subtree) + 1.
height of a tree = max ht. Of any node in a given tree
OR height of a root node.

Max height of bst = O(n); n = no. of elements/nodes in it.
Min height of bst = O(log n)

- if a bst is having min height then only operations like addition, deletion & searching can be perform on it efficiently i.e. in O(log n) time.
- bst having max height i.e. O(n) for n no. of elements in it referred as imbalaced bst.
- bst having min height i.e. O(log n), for given n no. of elements is referred as balaced bst => as addition, deletion & searching operations can be performed on it efficiently.

- **Balanced BST:** if all the nodes in bst are balanced then it is referred as balanced bst.
- if balance factor of a node is either -1 OR 0 OR +1 then we can say node is balanced.
- **balance factor of a node** = ht. Of its left subtree - ht. Of its right subtree.

- **if balance factor of a node < -1** => node is left imbalanced and hence bst is left imbalanced => **left rotation**

- **if balance factor of a node > +1** => node is right imbalanced and hence bst is right imbalanced.
- to achieve operations like addition, deletion & searching efficiently i.e. O(log n) time in bst, it must be balanced.
- if bst is not balanced then there is need to balanced it by applying left rotations and right rotations as per requirement.
- instead of it, if at the time of addition and deletion of a node into and from bst, if it makes sure at that time itself bst remains balanced => **self balanced bst**, this concept was designed by two mathematicians **adelson velsinki** & **lendis** and hence self balanced bst is also called as **avl tree.**

- **complete binary tree** is also called as **binary heap**
- complete binary tree is a type of binary tree which is also called as binary heap, which can be represented by using an array.
- elements in a complete binary tree/binary heap can be stored into an array.

DAY-08:
Graph:
What is a Graph?
- Graph is a non-linear/advanced data structure, which is a collection/list of logically related similar and disimmilar type of elements which contains,
- finite set of data elements referred as vertices also called as nodes, and
- finite set of ordered/unordered pairs of vertices referred as an edges also called as an arcs, whereas edges may contains weight/cost/value and it may be -ve.

Example:
google map: app – s/w -> programs

information about cities and path between them is stored

info about city => city_name, state, country, picode etc....

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

100 cities => 100 city class objects

information about path between cities also stored

```
class Path{
	String src_city;
	String dest_city;
	float dist_km;
	String time
	.....
}
```

Cities => vertices
Paths => Edges

- graph data structure is used to reprsent network
facebook : app
millions of users =>

```
class Person{

        ....
}
```

if two person’s are friends => they are connected => edge

```
class Edge{
        common info
        int mutual_firends;

        .
        .
        .
}
```

example=3:

to design PCB => Printed Cicuit Board -> matlab software
Points
Paths

-ve voltage

- adjancent vertices : if there exists a direct edge between two vertices

- if edges in graph carries weight/cost/value it is referred as weighted graph, otherwise regerred as unweighted graph.
- if edges in a graph are unordered pairs of vertices then such a graph is referred as undirected graph, if edges in a graph are ordered pairs of vertices then it is referred as directed graph/di-graph.

- graph can be implemented by two ways/there are two graph representation methods:
1. adjancency matrix (by using 2-d array i.e. matrix)
2. adjancency list (by using array of linked lists).

1. adjancency matrix (by using 2-d array i.e. matrix):
size of the 2-d array/matrix = V * V, V= no. of vertices in it

unweighted graph:
- if two vertices are adjancent => entry between them in a matrix = 1
- if two vertices are not adjancent => entry between them in a matrix = 0

weighted graph:
- if two vertices are adjancent => entry between them in a matrix = weight
- if two vertices are not adjancent => entry between them in a matrix = INF

2. adjancency list (by using array of linked lists).
- no. of linked lists in graph = V, V = no. of vertices in a graph

unweighted graph:
- for each vertex (from vertex/src vertex) one linked list is maintained and it contains nodes with to vertices/dest vertices).

- java programming => “data structure algorithms”

Q. Lab Work => to implement adjancecy list representation for undirected weighted graph.

graph algorithms:
1. dfs traversal
2. bfs traversal
3. dfs spanning tree
4. bfs spanning tree
5. to check connectedness of a graph
6. to find pathlength

1. dfs traversal:
- in tree data structure we can start traversal from only root node, in a graph we can start traversal from any node.

DAY-09
1. dijsktra's algorithm: this algo is used to find shortest distance of all vertices from the given source vertex.

We are having 2 sets of vertices:
graph G1 => V { 0, 1, 2, 3, 4, 5, 6, 7, 8 }

Subgraph/MST Set of vertices
graph G2 => MST { }

- in every pass this algo, selects one vertex from set of vertices which is having min key value and which is not yet added into the MST set of vertices and add it into MST set.

- update distance of all its adjacent but unmarked vertices to the min distance

enter source vertex : 0
MST SET OF VERTICES: { 0 1 7 6 5 2 8 3 4 }
MST SET OF VERTICES = { 0, 1, 7, 6 , 5, 2, 8, 3, 4 }

- this algo is also used to find shortest distance of all vertices from all vertices

greedy approach

2. prim's algorithm: to find minimum spanning tree
(spanning tree of a given graph having min weight.)
- prim's finds MST by maintaining MST set of vertices

3. kruskal's algorithm: to find minimum spanning tree
(spanning tree of a given graph having min weight.)
- kruskal's finds MST by maintaining MST set of edges.