DSA

Date: _____

Objectives

To understand the design of fundamental data structures as well as algorithms that operate on them

To understand the fundamental tradeoffs in the design of the data structure

To introduce tools for analyzing the time and space complexity of DS.

To provide rigorous 'hand-on' experience with implementing different data structures in a programming language.

Data:

Data is defined as facts or figures, or information that's stored in or used by a computer.

Simply, anything that is stored or processed by a computer is called data.

Computer works on binary data.

Structure

Structure refers to the arrangement of items in a specific format & pattern.

Simply, a structure is something of many parts that is put together.

The arrangement of and relations between the parts or elements.

Data Structures:

is representation of the logical relationship existing between individual elements of data.

Data structure is a specialized formats for organizing and storing data in memory that considers not only the elements stored but also their relationship to eachother.

Algorithm: An algorithm is a sequence of instructions that one must perform in order to solve as well-formulated problem.

Algorithm is the method of translating the inputs into the output.

A sequence of instruction or step to perform a certain task.

An algorithm is a sequence of unambigeous instruction/operation for solving a problem i.e for obtaining required output for any legitimate input in a finate amount of time.

Date: _____

Characteristics of an Algorithm:
-) Well-Defined inputs, -) Well-Defined Outputs, -) Clear & Unambiguous.
-) Finite-ness, -) Language independent, -) Feasible.

Algorithms can be represented in number of ways:
Flowcharts, Pseudocode, Natural Language, Programming Language.

Algorithms refers to operations on different data structures and the set of instructions for executing them.
Example:
We want to find the shortest path from home to school.
We have three potential paths:
Home → shop → market → school
Home → market → school
Home → mosque → school

It seems very trivial problem for us but computer is just an efficient slave. It can't think on its own.

For this purpose we write a set of instructions that can be coded later
1) Find all the places you can go from home
2) From each of those places, find all paths
3) keep track of the distance you have travelled.
4) Repeat this process until you get to school.
5) Compare the distance you have travelled.
6) Determine the shortest path.

Correct	Incorrect
We say that algorithm is correct when it translates every input instance into the ~~algorithm~~ correct output	An algorithm is incorrect when there is atleast one input instance for which the algorithm gives an incorrect output.

Date: _____

Remarks:
There may be infinite ways to solve a problem, it means there may be infinite algorithms.
Which algorithm is best?
We mainly focus on two perimeters.
•) Time Complexity •) Space Complexity.

Optimal Algorithm: Among all possible solutions to a problem, the best and optimal solution is the one which: (1) Takes minimum computational steps to determine the output - (2) Consumes minimum memory space to solve the problem.

Why Data Structure?
They help to manage & organize data - They are essential ingredients in creating fast & powerful algorithms. They make code cleaner and easier to understand.
Data Structure is important cuz it is used in almost every program or software system - It helps to write efficient code, structures the code and solve problems. Data can be maintained more easily by encouraging a better design or implementation. DS is just a container for the data that is used to store, manipulate and arrange - It can be proccessed by algorithms.

Example: Slide 24 – 28.

Date: _____

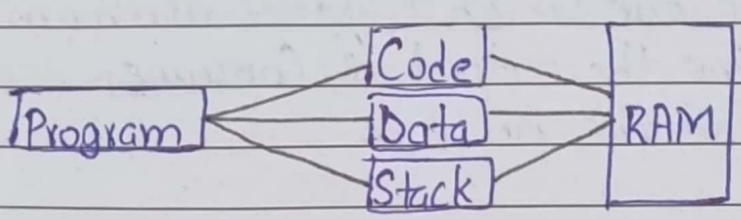
Classification of Data Structures:

Data Structures are normally divided into two broad categories.

Data Types: •) Primitive •) Non-Primitive.

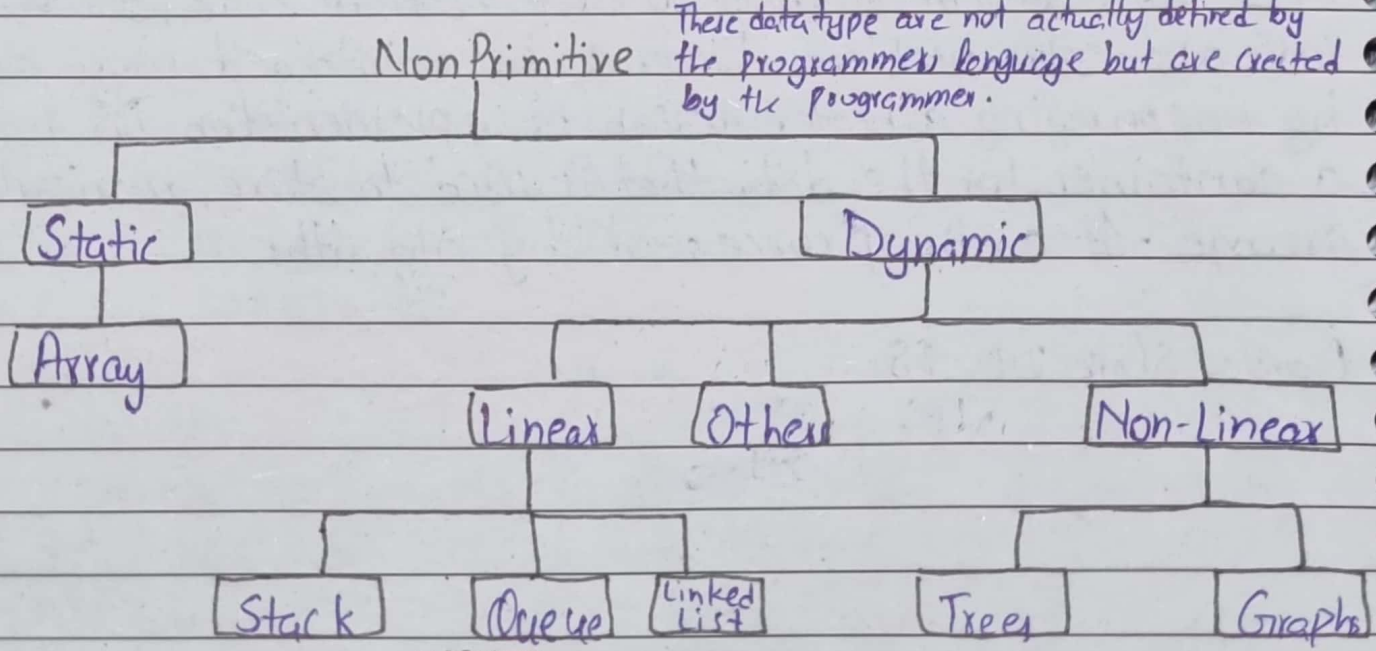
↳ integer, float, char, boolean, pointer, String.

Primitive DS: that are directly operated upon the machine level instructions are known as primitive data structures. Are those which are predefined way of storing data by the system. The set of operations that can be performed on these data are also predefined.



Non-Primitive DS:

There are more sophisticated data structure. The data structure that are derived from primitive data structures are called NonPremitive DS. The non primitive DS emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data.



Date: _____

How memory works?

You can imagine of a memory as a chest of drawers to store your thing Each drawer can hold one element. You want to store two thi so you ask for 2 drawers. This is basically how your computer's memory works - Your computer looks like a giant set of drawers and each drawer has an address - Each address is some hexadecimal number.

Each time you want to store an item in memory, you ask the computer for some space & it gives you an address where you can store your item. If you want to store multiple items ther are two basic way to do so: arrays and lists.

Array: Finite, Ordered collection of Homogeneous data element where,

-) Ordering is maintained by storing all element in
-) Continuous / Contiguous location of memory.

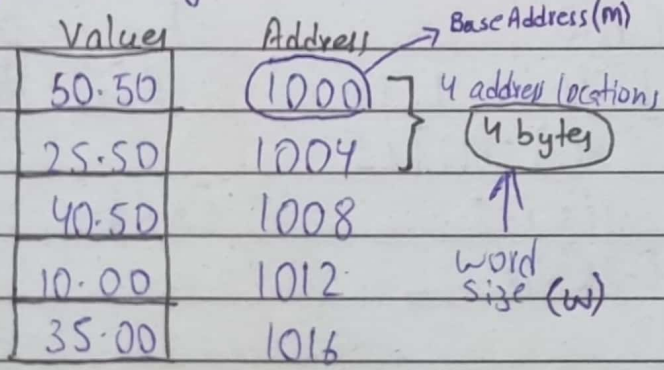
Properties:

-) Finite: Contains only a limited (finite) number of elements.
-) Ordered: All elements are stored one by one contineous location of comp mem
-) Homogeneous: All the elements of an array of the same data type.

Example:

An array of integers to store the roll number of all students in a class.

An array of strings to store the names of all villagers in village.



Row Major = L + [i x n + j] x W
Column Major = L + [j x m + i] x W

Array

Date: _____

Terminology:

- Size:
  - Number of elements in array
  - Also called: Length, Dimension.
  - Example: int A[5] → Size is 5.

int array[10] = {35, 33, 42, 5, 8, 10, 9, 21, 65, 31}

Name | Type | Size | Elements

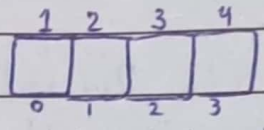
Index: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

(L) Lower bound (U) Upper Bound

- Type:
  - kind of data type it is meant for.
  - Example: int A[5] → Type is int

(W) Size → Word → of the datatype

- Base/Base Address (M)
  - Address of the memory location where the first element of the array is located.



index of element 3

0 + 3 - 1 = 2

Formulas/Equation

Size = U - L + 1

Index (i) = L + P - 1 ∴ P: Position of element in array.

Address = M + (i - L) * W ∴ W = Word size of array

Also called indexing formula

Could be expressed in the form of Position 'P' as Address = M + (P - 1) * W

Cuz (i - L) = (P - 1)

Find F[4], M = 5009 : 5009 + (4 - 0) * 4 ⇒ 5025 (F: Float data.)

Find Address of int data at P = 4, M = 1693

1693 + 2(4 - 1) * 2 ⇒ 1699

Algorithm: Traverse Array

Ques: Increment all values by 1.

Index	Array A
0	11
1	20
2	9
3	18
4	15

```
i = 0
While i <= 4, do
    A[i] = A[i] + 1
    i = i + 1
EndWhile
```

```
For i = 0 to 4
    A[i] = A[i] + 1
End for.
```

Date: _____

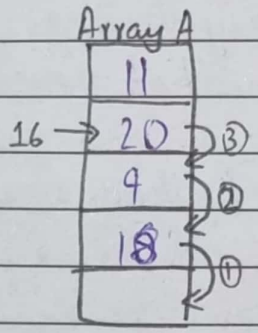
Index	Array A
-1	11
0	20
1	9
2	18

Trace Algorithm: Search Array

```
i = L, found = 0, location = NULL
While (i <= U) && (found == 0), do
  if (A[i] == key), then
    found = 1
    location = i
  Endif
  i = i + 1
EndWhile
IF (found == 0) then
  print ("Search unsuccessful")
else
  print ("Search successful")
endif
Return (location).
```

```
IF (A[U] != NULL)
  Print ("Array is Full");
else
```

Algorithm: insertarray Q: Insert an element '16' in this array



Steps:

```
A[4] = A[3]
A[3] = A[2]
A[2] = A[1]
A[1] = 16
```

Location → (1)

```
i = 4
While (i >= 2)
  A[i] = A[i-1]
  i = i - 1
EndWhile
A[1] = 16
```

Else

```
i = U
While (i > location)
  A[i] = A[i-1]
  i = i - 1
EndWhile
A[location] = key.
```

if (i == Null) then
Print (key not found)
else

Date:

Algorithm: Delete Array (Q): Delete an element from the array

Index	Array A	Steps		key → 20
0	10	i = SearchArray(A,20)	i = SearchArray(A,20)	i = SearchArray(A, key)
1	20	// i = 1	// i = 1	else
2	30	A[1] = A[2]	While (i <= 3), do	While (i < U), do
3	40	A[2] = A[3]	A[i] = A[i+1]	A[i] = A[i+1]
4	50	A[3] = A[4]	i = i+1	i = i+1
			End While	End While
		A[4] = NULL	A[4] = NULL	A[U] = NULL

Application of Arrays: Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables.
Many databases, small and large, consist of one dimensional arrays whose elements are records. Array are used to implement other data Structure Such as: Lists, Heaps, Hash Table, Queues, Stacks etc

Pros & Cons of Array: Fast, random access elements - Very memory efficient, very little memory is required other than that needed to store the content.
Slow deletion & insertion of elements - Size must be known when the array is created and is fixed (static).

Dynamic Data Structure: Most often we face situations in programming where the data is dynamic in nature - That is, the no. of data items keep changing during the execution of the program. Eg: consider a program for processing a list of customers of a corporation - The list grows when names are added & shrinks when names are deleted - When list grows we need to allocate more memory space to the list to accomodate additional data items.

Date: _____

Such situations handled using dynamic data structures in conjuction with dynamic memory management - Dynamic DS provide flexiblity in adding, deleting or rearranging data items at runtime. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at runtime, thus, optimizing the use of storage space.

Dynamic Array: are arrays that grow (or shrink) as required - In fact a new array is created when the old array becomes full by creating a new array object, copying over the values from the old array and then assigning the new array to the existing array reference variable - There is usually no limit on the size of such structures, other than the size of main memory.

(1) When growing array, how much to grow it?
Memory efficient? (1, x2, etc) · Time efficient (create, copy, insert, delete)

- Before every insertion, check to see if the array needs to grow - Growing by doubling works well in practice, because it grows very large very quickly 10, 20, 40, 80, 160, 320, 640, 1280 · Very few array re-sizing must be done To insert n items you need to do ≈ log(n) re-sizing.
- While the copying operation is expensive it does not have to be done oft
- When the doubling does happen it may be time-consuming - And, right aft a doubling half of the array is empty. Re-sizing at each insertion would be prohibitively slow. Similarly we can have blank array initially - As we add an element we can increase the size by 1 - Some may be considered for deletion; we decrease the size by 1 at each deletion
- Time vs Space Trade-off

Date: _____

Static Array:
Advantage: Fast random access of elements (take constant to access an element)
Efficient in terms of memory usage (almost no memory overhead)
Disadvantage: Has fixed size - Data must be shifted during insertion & deletion

Dynamic Array:
Advantage: Fast random access of element. Efficient in terms of memory usage
Disadvantage: Data must be shifted during insertion and deletion - Array may contain extra unused but reserved slots.

Why random access is fast in Array?
The elements in an array are stored at consecutive locations in the main memory - Therefore, if we know the address of the first element in the array and the size of each element, we can easily calculate the address of any element of the array in the main memory.

Algorithm Efficiency: An algorithm is said to be efficient and fast if it take less time to execute and consumes less memory space.

Performance Analysis: of an algorithm is the process of calculating space required by that algorithm & time required by that algorithm
The performance of an algorithm is measured on the basis of the following Properties:

•) Space Complexity: Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorith
•) Time Complexity: The time complexity of an algorithm is the total amount of time required by an algorithm to completes its execution.

Date: _____

Create data structure & algorithms to solve problems } Design

Prove algorithm work Buggy algos are worthless! Examine properties of algorithm Simplicity, running time, space needed	} Analysis

Growth of function: The order of growth of the running time of an algorithm, gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms.

Asymptotic Analysis: of an algorithm refers to defining the mathematical bound/framing of its run-time performance. Using asymptotic analysis, we ca very well conclude the best case, avg case & worst case scenario of algorith

Usually, the time required by an algorithms falls under three types.

→ Best Case Minimum time required for program execution.

→ Average Case Average time required for program execution.

→ Worst Case Maximum time required for program execution.

Asymptotic Notation:, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and igore least significant terms in the complexity of that algorith

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

-) Big Oh Notation, $O(n)$ Worst Case

-) Omega Notation, $\Omega(n)$ Best Case

-) Theta Notation, $\Theta(n)$ Avg Case

Date:

Asymptotic Notation Representation:
Constant: $O(1)$, Logarithmic: $O(\log n)$, Linear: $O(n)$, Linear logarithmic $O(n \log n)$
Quadratic: $O(n^2)$, Cubic: $O(n^3)$

(i) Constant Space Complexity:

```
int square(int a)   2 bytes
{
    return a*a      2 byte
}
```

4 bytes $\Theta(1)$

(ii) Linear Space Complexity

```
int sum(int[] A, int n)        // 2n, 2
{
    int sum = 0, i;            // 2, 2
    for(i=0; i<n; i++)
        sum = sum + A[i];
    return sum;                // 2
}
```

$2n+8$ $O(n)$

```
int[] ABC(int[]A, int[]B, int n)        // 2n, 2n, 2
{   int[]C = new int[n];                // 2n
    for(int i=0; i<n; i++)              // 2
    {
        C[i] = A[i] + B[i];
    }
    return C;                           // 2n
}
```

$6n+6$ $O(n)$

$8n+4$

Date: _____

(i) Constant Time Complexity:

```
int square(int a)
{
    return a*a;
}
```

It required same amount of time
i.e: 2 units. O(1)

```
Public void ABC()
{ int s = Input (1)
  s = s+5   (2)
  print(s)  (1)
}
```

4 → O(1)

Assignment a=5	(1)	Unit
Input	(1)	"
Output	(1)	"
Airthmatic a+b	(1)	"
logical a<b	(1)	"
return	(1)	"

```
for(int i=0; i<5; i++)        (1) (n) (n-1)
{
    Print("Hello") → n-1
}
```

4

```
for(int i=0; i<n; i++)
                n+1  n
{
    Print("Hello") n
}
```

(ii) Linear Time Complexity:

```
int sum(int[]A, int n)
{
    int sum=0, i;
    for(i=0; i<n; i++)
        sum = sum+A[i];
    return sum;
}
```

n=2

0<2 ● 1
1<2 n+1 2 n
2<2

Sum=0	1	1
i=0	1	1
i<n	1	n+1
i++	1	n
S=S+A[i]	1+1	n
return	1	1

4n+4
O(n)

		Repetition
i=0	1	1
i<5	1	n+1
i++	1	n
P	1	n

3n+2
O(n)

$n=2$ $\begin{bmatrix} 2 \geq 0 & i-- \\ 1 \geq 0 & 0 \\ 0 \geq 0 & 1 \\ -1 \geq 0 & 2 \end{bmatrix}$ $n+2$ $\rightarrow n+1$

Cost: is the amount of comp time required for a single operation in each line.
Repetition: is the amount of comp time required by each operation for all its repetition.
Total: is the amount of computer by each operation to execute.

Date: _____

```
int a=5
int b=6
for(int i=n; i≥0; i--)
{
    Print("Hello")
    Print(a+b)
}
Print(a)
Print(b)
```

	Cost	Repetition
a=5	1	1
b=6	1	1
i=n	1	1
i≥0	1	n+2
i--	1	n+1
P(H)	1	n+1
P(a+b)	1+1	n+1
P(a)	1	1
P(b)	1	1

$5n+11$ $O(n)$

```
for(int i=0; i<n; i++)
{
  for(int j=0; i<n; j++)
  {
    Print(i+j)
  }
}
```

	Cost	Repetition		Total
i=0	1	1		1
i<n	1	n+1		n+1
i++	1	n		n
j=0	1	1	(n)	n
j<n	1	n+1	(n)	n^2+n
j++	1	n	(n)	n^2
P(i+j)	1+1	n	(n)	$2n^2$

$4n^2+4n+2$ $O(n^2)$

```
Print("Hello")
for(int i=1; i≤n+1; i++)
{ int j=5
  while(j>n)
  {
    Print(j);
  }
  Print(n)
}
Print('Bye')
```

	Cost	Repetition
P(H)	1	1
i=1	1	1
i≤n+1		
i++		
j=5		
j>n		
P(j)		
P(n)		
P(bye)		

n=2: 1≤3, 2≤3, 3≤2, 4≤3

$3n^2+11n+12$

(iv) Logarithmic Time Complexity

```
int Method(int n)
{ int x=0;
  while(n>1)
  { n=n/2;
    x=x+1; }
  return x;
}
```

	Cost	Repetition	Total
int x=0;	1	1	1
while(n>1)	1	$\log_2(n)+1$	$\log_2(n)+1$
n=n/2;	1+1	$\log_2(n)+(1+1)$	$2\log_2(n)$
x=x+1; }	1+1	$\log_2(n)(1+1)$	$2\log_2(n)$
return x;	1	1	1
}			

$5\log_2(n)+5$

Date: _____

`for(i=0; i<n+1; i++)`	1	1	$n+2$	$1+n+2+n+1$
`{`				
`for(j=1; j<=n; j++`	1	$n+1$	$(n+1)$	$1(n+1)+(n+1)(n+1)+n(n+1)$
`{`		n	$(n+1)$	
`Print( );`	1		$n(n+1)$	$n(n+1)$
`}`				

$3n^2+7n+6$

$n=3$

$1<2$

$2<2$

$n=3$

$0\leq 3$

$1\leq 3$

$2\leq 3$

$3\leq 3$

$4\leq 3$

`for(i=1; i<n-1; i++)`	$i=1$	1	1	
`{`	$i<n-1$	1	$n-1$	
`print(i);`	$i++$	1	$n-2$	
`}`	$p(i)$	1	$n-2$	
`for(int j=0; j<=m; j++)`	$j=0$	1	$1 \quad (n-2)$	
`{`	$j\leq m$	1	$m+2 \quad (n-2)$	
`print(j)`	$j++$	1	$m+1 \quad (n-2)$	
`}`	$p(j)$	1	$m+1 \quad (n-2)$	

~~$1+n-1+n-2+n-2+1+m+2+m+1+m+1$~~

$3n+3m+1 \rightarrow O(n+m)$

if nested loop

$1+n-1+n-1+n-2+n-2+nm-2m+2n-4+nm-2m+n-2+nm-2m+n-2$

$8n-6m+3nm-14 \rightarrow O(nm)$

n=3

1 | 1<2 | 1<3
2<1 | 2<2
1<4
2<3
3<2

Date: _____

```
for(i=1; i<n-1; i++)
{
  for (int j=0; j<n; j++)
  {
    for(int j=0; j<n; j++)
    {
      for (int k=0; k<n; k++)
      {
        print ();
      }
    }
}
```

i=1	1	1
i<n-1	1	n-1
i++	1	n-2
j=0	1	1(n-2)
j<n	1	(n+1)(n-2)
j++	1	(n)(n-2)
k=0	1	1(n-2)(n)
k<n	1	(n+1)(n-2)(n) → (n²+n)(n-2) ⇒ n³-2n²+n²-
k++	1	(n)(n-2)(n)
print()	1	n(n-2)(n)

0 < 2
1 < 2
2 < 2

n³-2n

$1+n-1+n-2+n-2+n^2-2n+n-2+n^2-2n+n^2-2n+n^3-2n^2+n^2-2n+n^3-2n^2+$

$2n^3+n^2-6n-6 \rightarrow O(n^3)$

$3n^3-3n^2-3n-6$

Muhammad Anas

Assignment #2 3-C Date: _____

Q.1) (a)

a=0	1	1
i=0	1	1
i<n	1	n+1
i++	1	n
int j=0	1	1(n)
j<m	1	m+1(n)
j++	1	m(n)
a+=1	1+1	m(n)
Print(a)	1	m(n)

$1+1+n+1+n+nm+n+mn+mn+mn$

$4mn+4n+3$

$i<n \Rightarrow 0<2$
$1<2$
$2<2$

~~$1+1+n+n-1+n-1+mn-m+mn-m-n+1+mn-m-n+2+mn-m-n+1$~~

~~$4mn-4m+3$~~ ~~$5mn-n-5m+4$~~

$O(mn)$

$i<=n$
$0<=2$
$1<=2$
$2<=2$
$3<=2$

(b)

count=1	1	1
i=0	1	1
i<4	1	1
i++	1	1
j=0	1	1
j<i	1	1
j++	1	1
P(count)	1	1
count++	2	2

$10 \Rightarrow O(1)$

(c)

a=0	1
i=0	1
i<=n	n
i+=2	n-1
a++	2(n-1)

$1+1+n+2+n-1+2n-2$

$4n+1 \Rightarrow O(n)$

Date: _____

(d)

int i=0	1	1
i<n	1	n+1
i++	1	n
j=0	1	1(n)
j<m	1	m+1(n)
j++	1	m(n)
k=0	1	1(m)(n)
k<n	1	n+1(mn)
k++	1	n(mn)
l=0	1	$1(mn^2)$
l<m	1	$m+1(mn^2)$
l++	1	$m(mn^2)$
Print(i+l)	2	$2(m^2n^2)$

i<n
0<2
1<2
2<2

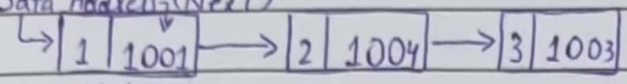
$1+n+1+n+n+m+n+mn+mn+mn^2+mn+$
$mn^2+mn^2+m^2n^2+mn^2+m^2n^2+2m^2n^2$
$4m^2n^2+4mn^2+3mn+m+4n+2$
$\boxed{O(m^2n^2)}$

Q.2) (a) $O(n^2)$ (b) $O(n^2)$ (c) $O(n^4)$

Date: _____

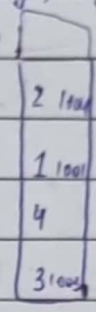
Linked List

Data Address (Next)



obj are created at runtime dynamically

How data stored in memory, dynamic.

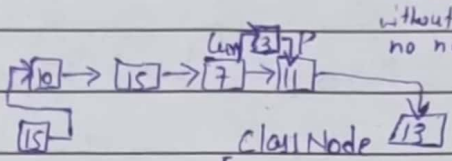


LinkList → abstract data type.

-) First node is a head node

LinkList & nodes having composition relation without linklist there will be no node.

List ADT: (Note)



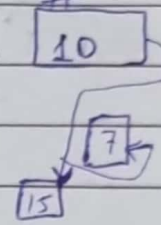

```
InsertAtFront(int data)
{ if (size>0)
  { Node n = new Node(data);
    n.next = Head;
    Head = n;

InsertMiddle(int data, int pos)
{ if (Pos ≤ sized)
  { Node n = new Node(data);
    if (Pos == 1)
      n.next = Head;
      Head = n;
else
int c=1;
while (c != Pos-1)
{ c++;
  curr = curr.next;
}
n.next = current.next;
curr.next = n;
}
```

```
class Node
{ int data;
  Node next;
}
SLL
{ Node Head
  int size
}
else if (Pos == Size+1)
{ while (curr.next != null)
  { curr = curr.next;
  }
  cur.next = n;
```

```
Insertion (At end)
head = null
size = 0
Insert(int data)
{ Node n = new Node(data)
  if (head == null)
  { head = n; }
  else
  { ~~head.next~~ }
  Node curr = Head;
  while (curr.next != null)
  { curr = curr.next;
  }
  curr.next = n;
  size++;
}
```

•) Phela n ko locate krein
gae connet with next equal to current ke next.
•) Then current ka next

#



Last node's next is alw
n

```
Traverse
Node curr = head
while (curr != null
{ print (curr.data
  curr = curr.next
}
```

Date: _____

```
public void InsertAt(int data, int pos)
{   if (pos <= size+1)
    {
        Node n = new Node(data)
    if (pos == 1)
    {
        n.next = head;
        head = n;
    }
```

Traverse:

```
Node curr = head;
while (curr != null)
{
    print (curr.data);
    curr = curr.next;
}
```

Search: (Head, key)

Curr=Head
Pos=1

```
While (Curr != null)
{   if (curr.data == key)
    {   return true; }
Endif  Pos++
    curr = curr.next;
}
return false;
```

Update → old data

curr.data = newdata.

Update (if (Pos <= Size))

```
Node curr = head;
int c = 1;
While (c != Pos)
{
    curr = curr.next;
}
curr.data = newval;
}
```

In LinkedList we can only do sequential search (Linear)
Because LL is not a random access data structure

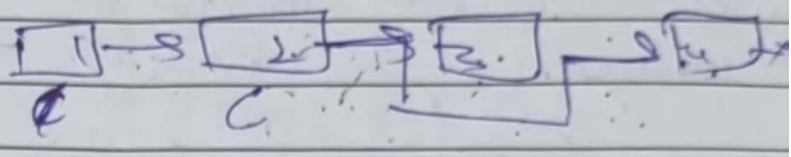
*) Pos wise delete:

*) Aik node pehla delete:

Date: _____

Deletion:

```
if (key == head.data)
{ head = head.next; }
else{
  Node curr = head;
  while (curr.next.data != key)
  { curr = curr.next; }
  if (curr.next.next == null)
  { curr.next = null; }
  else
  { curr.next = curr.next.next; }
```



```
Delete (int Pos){
if (head == null){
  Print (List empty)}
Node curr = head; Node Previous = null; int curPos
While (curr != null && curPos < Pos)
{ previous = current; current = current.next.
  currPos++; }
if (curr == null){ Print (not exist) }
else { previous.next = curr.next;
  Print (Position) }
```

① Odd position node data ↑ ki printing

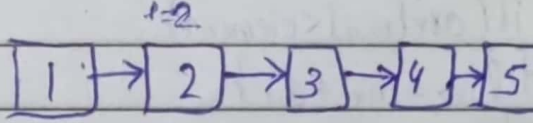
② Deletion, pos & count parameter (Pos=2, C=2) → kis position sae delete krna hai count btae ga kitni bar repeat krna

```
EvenPositionNodes(){
  Node Curr = head;
  int Pos = 1;
while (curr != null)
{ if (pos % 2 == 0)
  { Print (curr.data); }
  curr = curr.next;
  pos++; }
}
```

```
deleteElementAtPosition (int Pos, int C)
{ if (Pos <= 0 || count <= 0)
  { Print (Invalid Pos or C); return; }
Traverse
Node current = head;
for (int i=1; i<pos-1 && current != null; i++)
{ current = current.next; }
Check the range
if (Current == null || current.next == null)
{ Print (Invalid); return; }
Delete Node
Node temp = current.next;
for (int i=0; i<c && temp != null; i++)
{ current.next = temp.next;
  temp = null;
  temp = current.next; }
}
```

Traversal Code

```
insert (int data){
Node newNode = new Node(data
if (head == null)
{ head = newnode; }
else
{ Node last = head;
  while (last.next != null)
  { last = last.next; }
  last.next = newNode;
}
}
```



Date: _____

Linear Search: (Sorted or Unsorted) O(n)

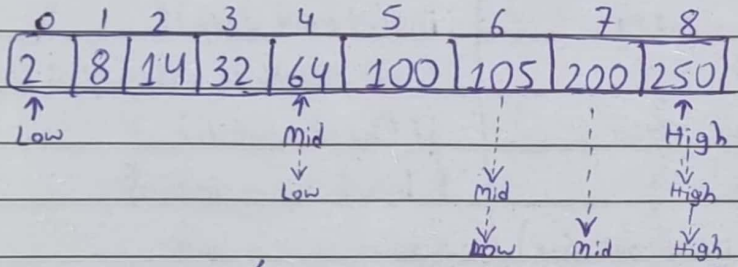
↳ Array traversal

```
int linearSearch(int arr[], int size, int element)
{   for (int i=0; i<size; i++)
    {  if (arr[i] == element){
        return i; }
    }
    return -1;
}
```

```
int main(){
  int arr[] = {1,5,8,6,9,4,10};
  int size = sizeof(arr)/sizeof(int);
  int element = 9;
  int searchindex = linearSearch(arr, size, element);
  print(element, searchindex);
} ret
```

Binary Search: (Sorted) O(log n)

↳ Take example of book contain 1000 pages so if we need to open page # 230 so we don't open each page turn by turn we open some random page might be mid i.e 500 then 250 this is how we reach to our required page #. Some way binary search works.



Low	High	Mid	Element
0	8	4	No
4	8	6	No
6	8	7	Yes

```
int binarySearch(int arr[], int size, int element)
{ int low=0, Mid, High=size-1;
while (low<=high)
{ mid = (low+high)/2;
  if (arr[mid] == element)
  { return mid; }
  if(arr[mid]<element)
  { low = mid+1; }
  else
  { high = mid-1; }
}
```

```
int main(){
int arr[] = {1,2,4,8,16,32};
int size
int element = 64;
int searchIndex = binarySearch(arr, size, element);
Print(element, searchIndex);
}
```

Date: _____

10	14	9	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

$m = (0+9)/2 \rightarrow 4.5$

$m = (5+6) \rightarrow 5$

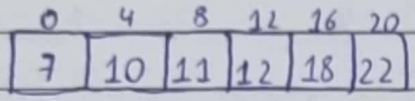
key = 31

32 < 33

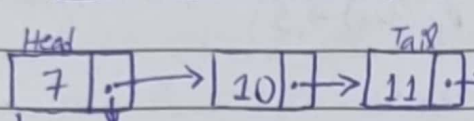
```
S = LB - 0
E = UB - 9
While ( S <= E)
{ m = (S+E)/2
  if (arr[m] == key)
  { return true; }
else if (key > arr[m])
  { S = m+1; }
else if (key < arr[m])
  { E = m-1; }
```

Linked Lists: Date: _____

↳ Linked lists are similar to arrays (Linear data structure)



⇒ Arrays elements are stored in contiguous memory locations.



⇒ In linked list elements are stored in non-contiguous memory loc.

Why Linked list? Advantage: Insertion & Deletion fast.

Memory & the capacity of an array remains fixed - In case of linked lists, we can keep adding & removing elements without any capacity contraints.

Drawbacks of Linked list:

→ Extra memory space for pointers is required (for every node 1 pointer is needed)

→ Random access not allowed as elements are not stored in contiguous memory location.

Insertion:

```
insertatfirst(int val){
  Node n = new Node(val);
  n.next = head;
  head = node;
};
```

```
display(){
  Node temp = head;
  while (temp != null){
    print(temp.value);
    temp = temp.next;}
}
```

```
insertatlast(int val){
  if (tail == null){
    insertFirst(val);
    return; }
  Node n = new Nod(val);
  tail.next = node;
  tail = node;
  size++;
}
```

```
insertPos(int val, int Index){
  if (index == 0){
    insertatfirst(val);
    return; }
  if (index == size){
    insertlast(val);
    return; }
```

Continue

```
  Node temp = head;
  for(int i=1; i<index; i++){
    temp = temp.next; }
  Node n = new Node(val, temp.next)
    temp.next = node;
  size++;
}
```

Date: _____

Deletion:

```
deleteFirst(){
int val = head.value;
head = head.next;
if (head == null){
   tail = null ; }
size--;
return val; }
```

```
Node get(int index){
Node node = head;
for(int i=0; i<index; i++){
   node = node.next; }
   return node;
}
```

```
deleteLast(){
if (size<=1){
 return deleFirst();}
Node secondlast = get(size-2);
int val = tail.value;
tail = secondlast;
tail.next = null;
return val; }
```

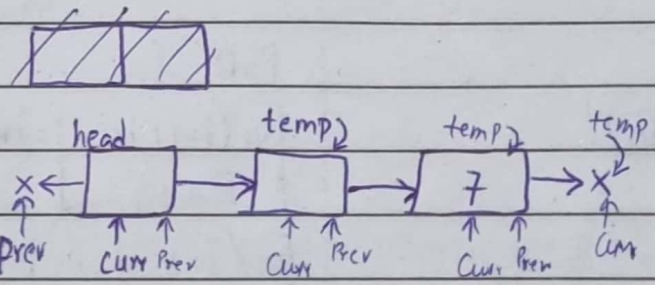
```
int deletePos(int index){
  if (index==0)
{ return deleteFirst();}
  if (index == size-1)
  { return deleteLast();}
Node prev = get(index-1);
  int val = prev.next.value;
  prev.next = prev.next.next;
```

From data find node (value)

```
Node find(int value)
{ Node n = head;
  while (n != null)
  { if (n.value == value)
    { return n;}
    n = n.next;
  }
  return null;
}
```

Reverse LL:

```
Node reverse(Node n)
{ Node curr = head;
  Node prev = null;
  while (curr != null)
  { Node temp = curr.next;
    curr.next = prev;
    prev = curr;
    curr = temp; }
  n = Prev;
  return n;
}
```



Time Complexity:

Date: _____

for (i=0; i<n; i++) —— O(n)

for (i=0; i<n; i=i+2) —— $\frac{n}{2}$ O(n)

for (i=n; i>1; i--) —— O(n)

for (i=1; i<n; i=i*2) — $O(\log_2 n)$

for (i=1; i<n; i=i*3) — $O(\log_3 n)$

for (i=n; i>1; i=i/2) — $O(\log_2 n)$

Questions:

```
P=0
for(i=1; P<=n; i++)
{
    P=P+i;
}
```

i	P
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
⋮	
k	1+2+3+4 ⋯ +k

Assume: P > n

$\because P = \frac{k(k+1)}{2}$ $O(\sqrt{n})$

$\frac{k(k+1)}{2} > n$

$k^2 > n$

$k \geq \sqrt{n}$

```
for(i=0; i*i<n; i++)
{ State.. }
```

i*i < n

Assume i*i >= n $O(\sqrt{n})$

$i^2 = n$

$i = \sqrt{n}$

```
for(i=1; i<n; i=i*2)
{ Stat }
```

Assume i >= n

$\therefore i = 2^k$

$2^k >= n$

$k = \log_2 n$ $O(\log n)$

i
1
1×2 = 2
2×2 = 2^2
$2^2 \times 2 = 2^3$
⋮
2^k

```
for(i=0; i<n; i++)
{ Stat } —— n

for(j=0; j<n; j++)
{ State.. } —— n

2n
O(n)
```

```
P=0
for(i=1; i<n; j=i*2)
{ P++; } —— P = log n

for(j=1; j<P; j=j*2)
{ State; } —— log P

O(log log n)
```

$O(\log(\log n))$

n = 8	
i	
1	] 3
2	
4	
8x	

$\log 8 = 3$

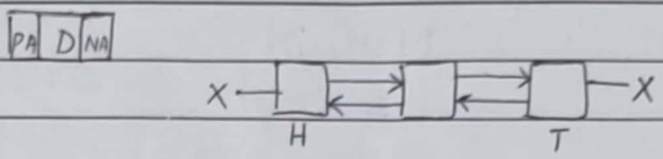
$\log 2^3$

$3 \log 2 = 3$

traverse, search, update same code as SLL

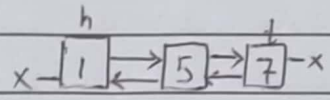
Date: _____

Double Linked List:



At End

```
Node n = new Node(data);
if(head == null)
{ head = n;
  tail = n;
} else {
  tail.next = n;
  n.prev = tail;
  tail = n; }
```



Reverse:

```
Node curr = tail;
while (curr != null)
{ print(curr.data)
  curr = curr.prev;
}
```

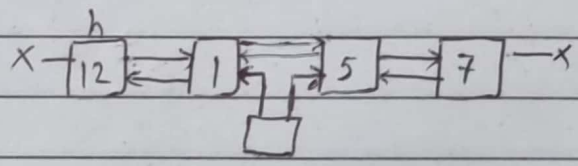
insertion by position

(Data, Pos)

```
if(pos <= size+1)
{
 if(pos == 1)
 { n.next = head;
   head.prev = n;
   head = n; }
 else if(pos == size+1)
 { tail.next = n;
   n.prev = tail;
   tail = n; }
 else
 { int c = 1;
   Node curr = head;
   while (c != pos)
   { curr.next;
     c++; }
   n.next = curr;
   n.prev = curr.prev;
   curr.prev.next = n;
   curr.prev = n;
```

if Pos-1

```
while (c1 = pos-1)
n.next = curr.next
n.pre = curr
curr.next.pre = n
curr.next = n
```



Deletion

```
if(Pos <= size)
{ if(pos == 1)
  { head = head.next;
    head.prev = null; }
  else if(pos == size)
  { tail = tail.prev;
    tail.next = null; }
```

Cont →

```
else
{ int c = 1;
  Node curr = head;
  while (c != Pos)
  { curr = curr.next;
    c++; }
  curr.prev.next = curr.next;
  curr.next.prev = curr.prev;
}
```

Q.) Circular LL Date: _____

```
DeleteValue = Dv
if(head.data==Dv)
{head=head.next;
 head.prev=null;}
else if(tail.data=Dv)
{ tail=tail.prev;
  tail.next=null;}
else
{ while(curr.next!=null)
  { if(curr.data!=Dv)
    { Curr=curr.next;}
  } curr.prev.next=curr.next;
   Curr.next.prev=curr.prev;
}
```

```
DFO(key){
if(head==null)
 return;
Node curr=head
Node prev=null;
do{
if(curr.data==key){
```

① Delete Num first occurance.
② last occurance
③ Delet all occurance

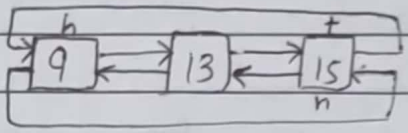
(2), 5, 6, 2, 3, 9, (2)
1 — All — Last

Date: _____

Circular linked List

Simple insert

```
if (head == null)
{ head = n;
  tail = n;
  head.prev = tail;
  tail.next = head; }
else
{ tail.next = n;
  n.prev = tail;
  tail = n;
  head.prev = tail;
  tail.next = head;
```




```
Node curr = head;
do
{ print(curr.data) }
curr = curr.next.
while (curr != head)
```

Delete By Data

```
if (DV == head.data)
  { same code }
elseif (DV == tail.data)
  { Same }
else
{ Node curr = head.next;
  do
  { if (curr.data == DV)
    { curr.Prev.next = curr.next
      curr.next.prev = curr.Prev }
    curr = curr.next;
  while (curr != head)
```

Insert

```
if (Pos == 1)
{ n.next = head;
  head.prev = n;
  head = n;
  head.prev = tail;
  tail.next = head; }
else if (pos == Size+1)
{ tail.next = n;
  n.prev = tail;
  tail = n;
  tail.next = head;
  hed.prev = tail; }
else
{ int c = 1;
  Node curr = head;
  while (c != Pos)
  { c++;
    curr = curr.next; }
  n.prev = curr.prev
  n.next = curr;
  curr.prev.next = n;
  curr.prev = n;
```

Delete At

```
{ if (Poss == 1)
  { head = head.next
    head.prev = tail;
    tail.next = head; }
  else if (Pos == Size)
  { tail = tail.prev;
    tail.next = head;
    head.prev = tail; }
  else
  { int c = 1;
    Node curr = head;
    while (c != Pos)
    { c++;
      curr = curr.next; }
    curr.prev.next = curr.next
    curr.next.prev = curr.prev. }
}
```

(Linear) Push Pop
•) We can place or remove from top only.

Underflow
If list is empty

LIFO

Stack Function
• Push → Return
• Pop → Delete
• Top → Pointer.
• Peak → top ki value

↳ is an ordered group of homogeneous item of element.

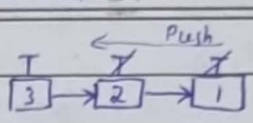
•) Push(X): insert X at top
•) PoP(): Remove topmost element.
•) Peak: examine the element at top of stack.

Date:

Stack: (Data Structure) Abstract Data Type (ADT)

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

```
class Stack{
Node top;
public Stack()
{top = null;}
public void Push(int data)
{ Node n = new Node(data);
if (top == null)
 { top = n;}
else
{n.next = top;
 top = n;}
}
```

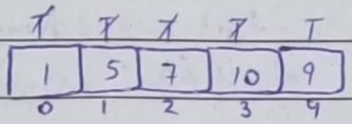


For insertion we insert from 1st position as we want LIFO.

```
Public int Pop()
{ if (top == null)
 { Print("Stack Underflow")}
  return -1; }
else
{int temp = top.data;
 top = top.next;
 return temp;}
}
```

→ we remove this line for Peak.

Using Array:



T = Size - 1

•) Its not legal to access any element other than the one that is at the top of stack!

```
class Stack{
int top, size;
int [] arr;
public Stack(int size)
{this.size = size;
arr = new int[size];
top - 1;}
```

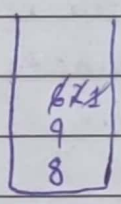
```
Public void Push(int data)
{ if (top == size-1)
  {Print(Stack overflow);}
else
 {arr[++top] = data;}
}
```

```
Public int Pop()
{ if (top == -1)
 {Print(stack underflow);}
else
return arr[top--];}
}
```

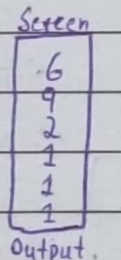
arr[top];
top--;

•) For Peak we just remove top-- we return top.

Q)
Push(8)
Push(9)
Push(6)
Pop()
Peek()
Push(2)
Pop()
Push(1)
Peek()
Peek()
Pop()



Screen



Output.

Operation on Collection

-) Add element
-) Remove element
-) Determine if collection is empty.
-) Determine the collection's size.

Use of Stack in computing.

•) Backtracking •) Browsers.
•) Word processor, editors
•) Markup languages
•) Stack Calculator •) Compiler.
•) Call Stack (Runtime stack)

Date: _____

Application of Stack:

- Stack data structure are used in bracketing problems.
- Many compilers use a stack for parsing the syntax of experssions.
- Stack is used to reverse a string or any other array.
- Stack is used to keep info about the active functions or subroutines
- keeping track of operations for managing the Undo function.
- Stack can be used for experession evaluation.
- Stack can be used to check parenthesis matching in an expression.
- Stack can be used for Memory Management.

(Q) What do we need to implement stack?

A data (container) to hold data element, something to indicate the top of the stack.

Practise Ques.

- Construct algo for basic stack operations if stack is implemented using DLL.
- Consider UP & DOWN pointers instead of Previous & Next for your convenience.

```
Push(data)
newNode = CreateNode(data)
newNode.Down = top
if top != null
    top.UP = newNode
top = newNode.
```

```
Pop()
if top == null
  return (Underflow)
data = top.data
top = top.DOWN
if(top != null)
  top.UP = null
return data.
```

```
Peak()
if top == null
  return (Underflow)
return top.data.
```

Infix → operand <operator> operand
Prefix → <operator> operand1, operand2
Postfix → oprand1 oprand2 <operator>

Postfix/Prefix:

Date: _____

$(2+3)*7/3$ → eqn is called infix, computer will solve it either using prefix & Postfix.

2+3
+23 → Prefix
23+ → Postfix

convertion → operations will be performed after convertion at evaluation stage.

+, - ⇒ same precedence / left associatively
*, / ⇒ " " " "
^ ⇒ Right associativity.

(-) 2+3-4

-
*

operands → on screen (2,3)
operators → in stack (+, -, *)

=) 2+3x4

x
+

234x+

multiply is high precedence so 4 will come out & will directly push.

23+4- Precedence of + & - is same so, first take + out then push -

=) 2x3+4

+
x

23x4+

LA
Same Precedence → First pop then push.
High in low out → First pop then push.
low in high out → directly in.

=) 2^3^4

^
^

234^^

→ Same preceding will go onto the last in RA

(-)(2+3)*7/3

/
x
+
(

23+7x3/

-) Opening bracket push in stack, when we get the closing bracket so pop everthing before closing.

=) A=B=Dx(E/F+BxC

x
+
x
x
/
(

AB=DEx=F/BCx+

↳ we will not consider this
→ Now only pop X

(-)(A+B)xC-(DxE)-(F+G)

x
C
-
x
x
/
(

AB+CxDE*-FG+-

(-)(a+(b-c))x((d-e)/(F+g-h))

abc-+de-Fg+h-/x

)
/
*
(
/
x
+
(
(
x
)
x
/
+
(
*
(

(-)(2+5)^3x7^4

^
x
x
+
(

25+3^74^x

= Paranthesis balance.

Date: _____

Prefix → write notation in reverse

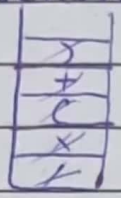
(-)(2+3)x 7/4 → reverse infix

4/7x(3+2)

47/32+x

x+23/74

↳ again reverse.

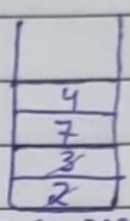


Evalution:

-)23+7x4/

2+3 7x4

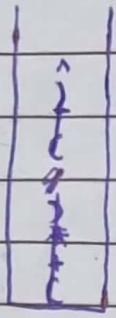
when you reach to operator so, pop 2 operands put the first pop after operator & second before operator.



Post Fix

(Q) (A+B*C)/(D-E)^F

ABC*+DE-F^/



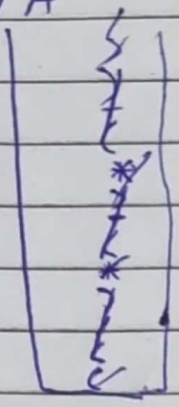
Pre Fix

Q) A/((B+C)*(D-E)*(F/G))

((G/F)*(DE-D)*(C+B))/A

GF/ED-*CB+*A/

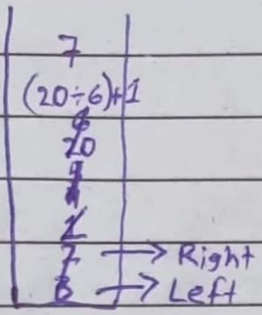
/A*+BC*-DE/FG



Post Fix Evaluation

(Q) 32-45*6/+7-

[(20÷6)+1]-7



first = right

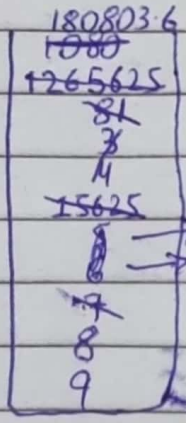
second = left

Pre Fix Evaluation

(Q) -+/*^34^56789

98765^43^*/+-

(180802.6)



First → left

Second → right

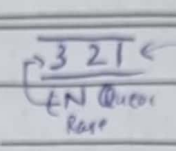
Unlike stack queue is open at both its ends
*) One end is used to insert (enqueue) → rear
and other is used to remove (dequeue) → front

Enqueue → insert at end

Stack & Queue both doesn't support random access

Queue: (linear) (FIFO)

Date:



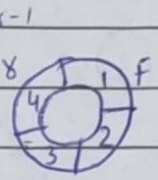
Peak:

```
class Queue                                  → int count,
{ int f, r;
  int []arr;
  int size;
public Queue(int size)                       → count = 0;
{ r = -1;
  f = 0;
  this.size = size;
  arr = new int[size]
}
```

```
Public void Enqueue(int data)
{ if (r == size-1)
  { Q is full }
  else
  { arr[++r] = data; }
}
```

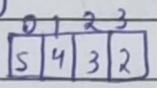
```
Public Dequeue
{ if (f == r+1)
  { Q is Empty; }
  else
  { temp = arr[f]
    f++;
    return temp; }
```

↓ with count
Circular Queue.



Front = rear = -1 → empty.

```
Public void Enqueue(int data)
{ if (count == size)
  { Q is full }
  else
  { r = (r+1) % size
    arr[r] = data;
    count++; }
}
```

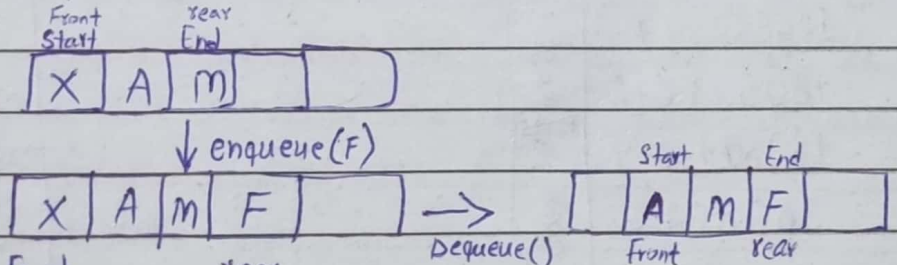


Size = 4
Count = 0, 1, 2, 3, 4
r = -1 f = 0
-1+1
0%4, 1%4, 2%4, 3%4

DQue
Count = 4, 3, 2, 1, 0
5, 4, 3, 2

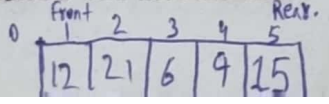
```
Public Dequeue
{ if (count == 0)
  { Q is empty }
  else
  { temp = arr[f]
    f = (f+1) % size;
    count--;
    return temp; }
}
```

Enque: operation to place a new item at the tail of the queue.
Dequeue: operation to remove the next item from the head of the queue.




```
Dequeue (value)
If r=0 then return null
value = Q[f]
for i = f to r-1
    Q[i] = Q[i+1]
r--
return value.
```

MIGHTY PAPER PRODUCT

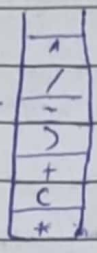


Assignment #03 CS221004 M.Anas Date: _____

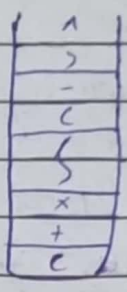
Q.1

(1) AB+CD^*E-

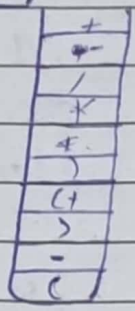
(2) ABC+*DEF^/-



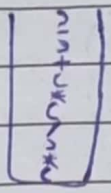
(3) ABCx+DE-F^/



(4) AB*CD-*EFG+/+



(5) AB*CDE+*F-/

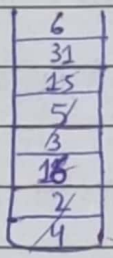


Q.3

(1) 4^2 . 3x5

31+6

37

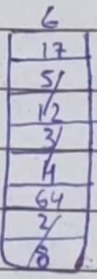


(2) 8^2

4x3, 17-6

64-11

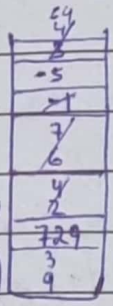
53



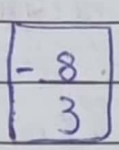
(3) 9^3 4^3

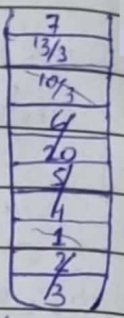
-5+64=59

729-59=670



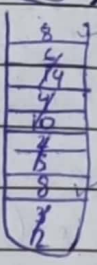
(4)





(5) 8x8

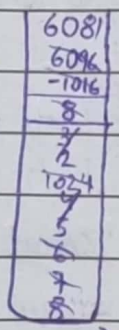
64



Q.4 (1) 87654^23^-*++

4^5 2^3 8-1024 1016x6

-6081

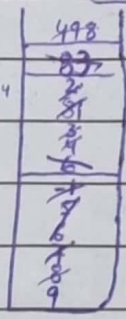


(2) 98765-+43^2+**-

83x6 3^4

498x8 = 3984

3984-9 = 3975

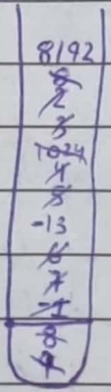


(3) 98-76+/54^32^*-

8x1024

8192-(-13)

8205



(4) 98765^43^*/+-

5^6

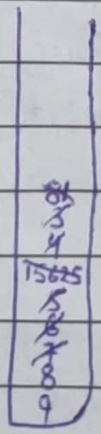
81x15625

1265625÷7

180803·6+8

180821·6-9

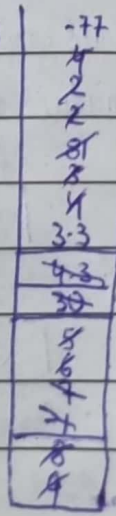
180802·6



(5) 98-765*/+43^22+-*

4-81 = -77x3·3

-254·1



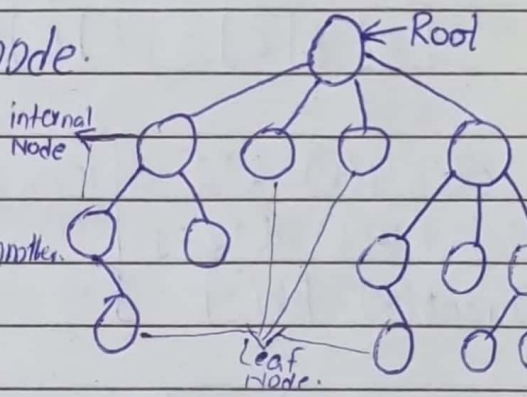
4) Print all leaf nodes

Tree:

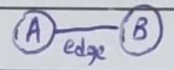
Date: _____

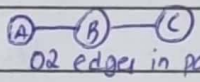
-) A tree is a nonlinear abstract data type that stores element in a hierarchy.
Tree is a set of elements that either, it is empty or it has a distinguished element called the root & zero or more tree (called subtree of the root)

Tree Terminology: Node: the elements in the tree.
Edges: Connection between nodes.
Root: the distinguished element that is the origin of the tree
-) There is only one root node in the tree.
Empty Tree: has no nodes & no edges
Parent or Predecessor: the node directly above another node in the hierarcy
-) A node can have only one parent.
Child or successor: a node directly below another node in the hierarcy.
Siblings: node that have the same parent.
Ancestor of a node: its parent, the parent of its parent, etc.
Decendants of a node: its children, the children of its children, etc.
Leaf node: a node without children.
Internal node: a node that is not a leaf node.



Height of Tree:



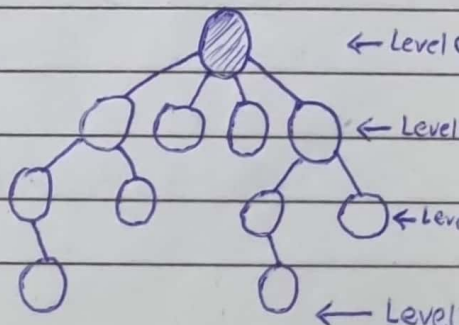


02 edges in path A to C

→ A path is a sequence of edges leading from one node to another.
length of path: number of edges on the path
Height of ~~path~~ (non-empty) tree: lenght of the logest path from the root to the leaf.
What is the height of tree that has only root node? 0

Height starts from 0 and goes on.
Height of empty tree is -1

Level of a node: Number of edges between root & the node
It can be defined recursively
-) Level of root is 0
-) level of node that is not root node is level of its parent + 1



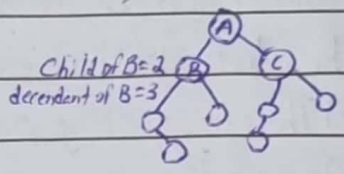
•) Back tracking is not possible in tree can't go from child to parent

Date: _____

Subtree: of a node consists of child node & all its descendants.
-) A subtree is itself a tree. -) A node may have many subtrees.

Degree or arity of a node: the number of children it has.
Degree or arity of a tree: the maximum of the degree of the tree's nodes.

Binary Tree: not more than 2 child
General Tree: a tree each of whose nodes may have any number of children.

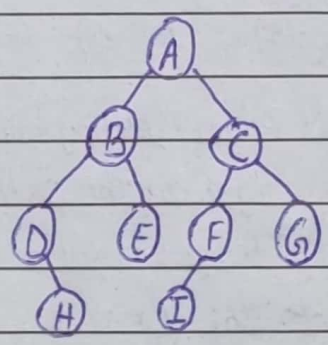


n-ary tree: a tree each of whose nodes may have no more than n children.
Binay Tree: a tree whose nodes may have no more than 2 children.
i.e: binary tree is a tree with degree (arity) 2.
The children (if present) are called the left child & right child.

Array based implementation of BT:

	1	2	3	4	5	6	7	8	9	10	11	12
Root →	A	B	C	D	E	F	G	Ø	H	Ø	Ø	I

Leftchild(k) = k*2
Rightchild(k) = (k*2)+1
Parent(k) = [k/2]
] if any index starts with 1

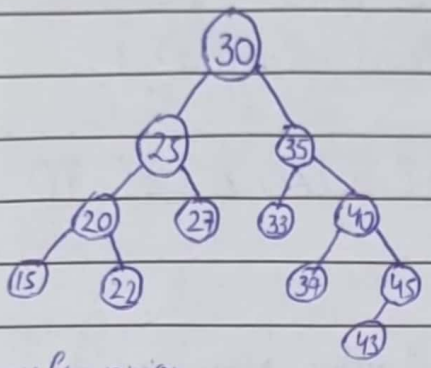


L.C = (k*2)+1
R.C = (k*2)+2
] index = 0

Date:

Tree traversal: BST : 2 child, node < Parent = left, node > Parent = right

30, 25, 35, 20, 15, 40, 45, 43, 27, 33, 39, 32



if less check, if left is null, if null insert else make curr = curr.left
if greater, check if right is null. if null insert, else make curr. = curr.right.

Array Conversion

30, 25, 35, 20, 27, 33, 40, 15, 22, -, -, -, -, 39, 45, ..., 43, -

A traversal of a tree requires that each node of the tree be visited once

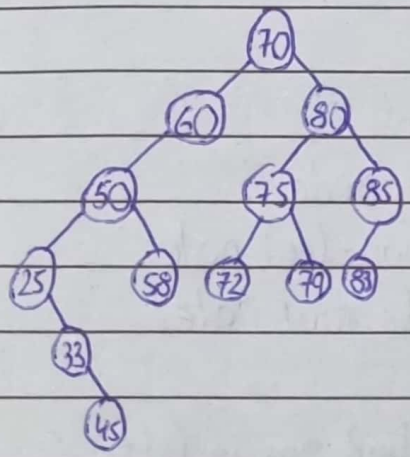
Standard traversal ordering:

Pre-order: ↳ Root, L, R

In-order ↳ L, root, R

Post Order ↳ L, R, Root

70, 80, 85, 60, 50, 58, 25, 33, 45



Pre-order

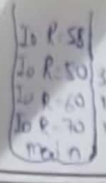
70, 60, 50, 25, 33, 45, 58, 80, 75, 72, 79, 85, 83

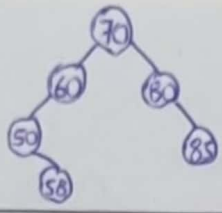
Post-order:

45, 33, 25, 58, 50, 60, 72, 79, 75, 83, 85, 80, 70

In-order (ascending order).

25, 33, 45, 50, 58, 60, 70, 72, 75, 79, 80, 83, 85





Date: _____

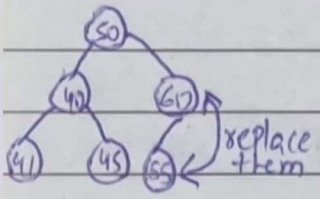
Inorder (node root)	PreOrder (node R)	PostOrder (node R)
if (Root != null)	if (R != null)	if (R != null)
{ Inorder (Root.left)	{ ~~Preorder~~	{ Postorder (R.left)
Print (Root.data)	Print (R.data)	Postorder (R.right)
Inorder (Root.right)}	Preorder (R.left)	Print (R.data) }
	Preorder (R.right)}	
50,58,60,70,80,85	70,60,50,58,80,85	58,50,60,85,80,70

Searching in tree:
Same Process as insertion
if num is less move to left otherwise move to right & check value if it matches then return true.

```
insert (int value, Node node)
{ if (node == null)
  { node = new Node(value);
    return node; }
  if (value < node.value)
  {node.left = insert(value, node.left)}
  if (value > node.value)
  {node.right = insert(value, node.right)}
  return node;
}
```

Deletion: two child, One child, leaf node

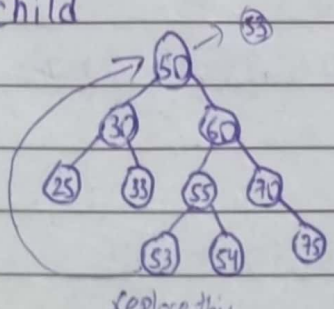
One Child



leaf node
leaf node → null

Deletion:
Case 1: Node leaf node
Case 2: Node is non-leaf node
Case 3: Node is the root node.
↳ Two child

two child



In order successor
★ Right · left most minimum
★ left · right most maximum

• Find Min in right	• Find max in left
• Copy the value in targetted node	• Copy the value in targetted node
• Delete duplicate from right subtree	• Delete duplicate from left subtree.

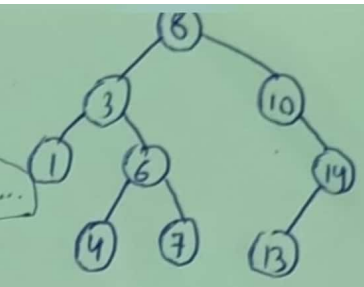
Case: 1

Step 1: Search the node
Step 2: Delete the node
↳ node = null.

Case: 2

Can replace with Inorder predecessor or successor
If we want to delete 6
So it will be replace with either 4 or 7.

Inorder: 1, 3, 4, 6, 7, ...



Inorder traversal
1, 3, 4, 6, 7, 8, 10, 13, 14

Case 3:

Search for node
Search for Ipre & Ipost
keep doing this until the tree has no empty nodes.

Case 3:
Alway prefer leaf node for replace
•) we can replace 8 with 7 or 10
7 eazy
Simply replace.
If we want to replace with 10
So we need to place 13 in place of 10
as 13 is inorder processor of 10, to fill 10 node's place.

```
Search (int key, Node root)
{ if (root == null)
    return null;
  if (root.data == key)
    return root;
  else if (root.data > key)
    return search(root.left, key)
  else
    return Search(root.right, key)
}
```

```
Deletion (root, key)
{ if (root == null)
    return null;
//Search for node to be deleted
 if (key < root.data)
    root.left = deletion (root.left, key)
 else if (key > root.data)
    root.right = deletion (root.right, key)
 else
// Node with only one child or no child.
  if (root.left == null)
     return root.right
  else if (root.right == null)
     return root.left.
// Node with two children
 minValueNode = findMin (root.right)
 root.data = minValueNode.data.
 root.right = deletion (root.right, minValue
 return root }
```

It will be replaced with the smallest node in its right subtree.
& Smallest node will be deleted from right subtree.

```
findMin (node)
{while (node.left != null)
    node = node.left
 return node }
```

BST : log(n) → insert, delete, Search → only for balanced otherwise O(n)

Date: _____

Q) Count no. of leaf nodes.

```
if (Root.null)
{ Inorder (R.left) }
if (R.right == null && R.left==null)
  { count++ }
 Inorder (R.right)
```

Q) Print data of all leaf nodes.

```
LeafNode (n)
 if (n != null)
   if(n.left==null && n.right==nul
      print (n.data)
 LeafNode (n.left)
 LeafNode (n.right)
```

x——x——x——x

```
findHeight(root)
if( root == null)
 return 0; //Height of empty tree is 0
else{ int leftheight = findHeigh(root.le
 int rightheight = findHeight(root.righ
return (leftheight > rightheight)? leftHeight+
                                  or rightheight++
```

Height of Tree : Traverse & count roots.

Why Balancing: To get log(n) complexity.

BF = -1, 0, 1 so tree is balanced.

BF = Height(LST) - Height(RST)

Types of Balanced Tree:

Perfect: A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes & all the leaf nodes are at same level.

Full: A full binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

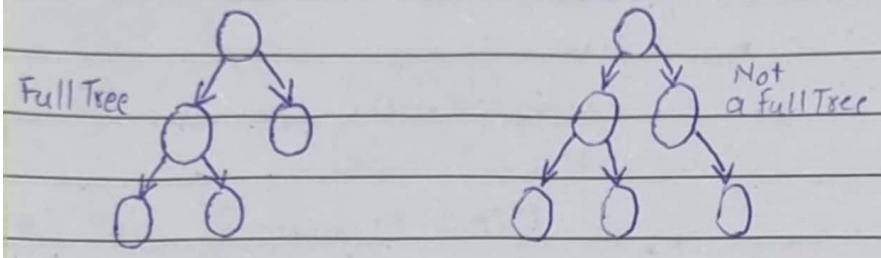
Complete: A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

Balanced: A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

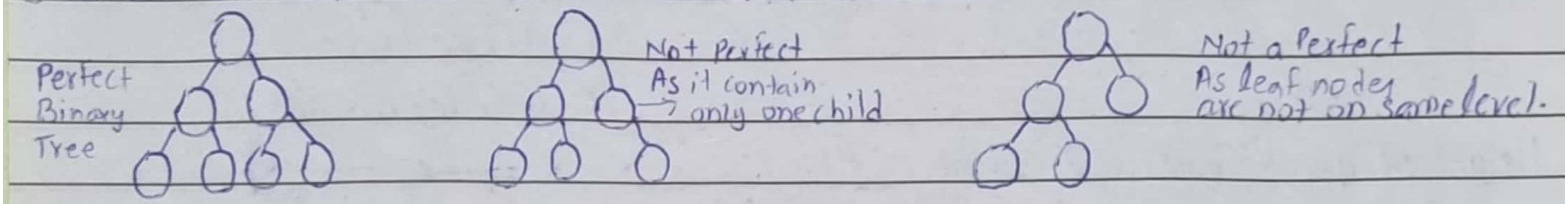
LST height - RST height <= 1

Date: _____

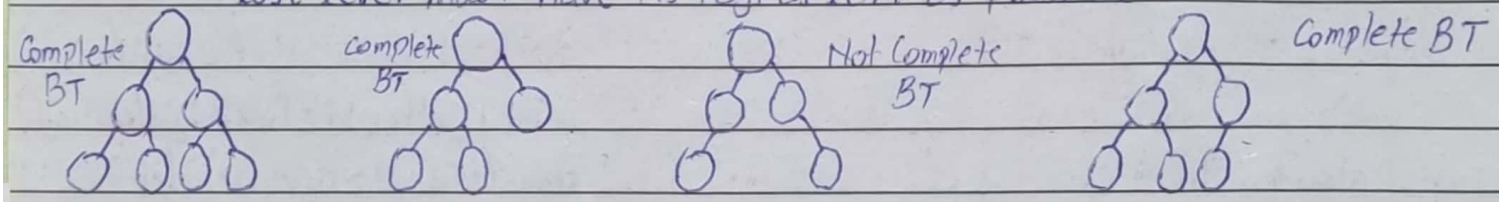
① Full : All node have either 0 or 2 children



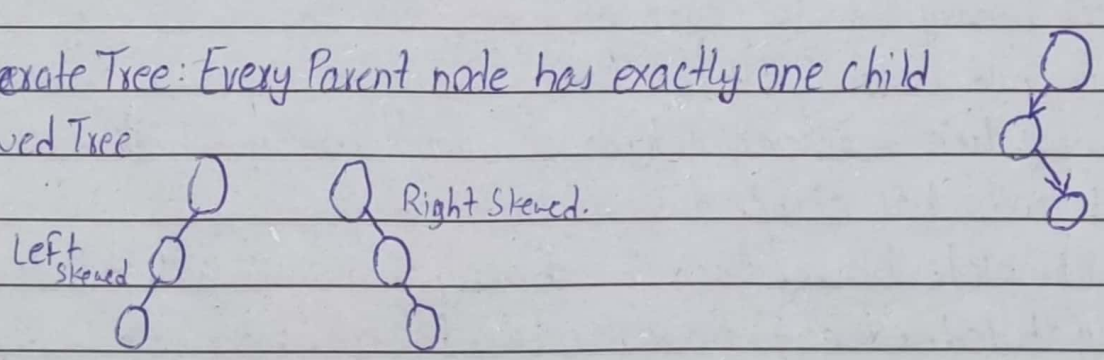
② Perfect : Internal nodes have 2 children + all leaf nodes are on same level.



③ Complete : All levels are completely filled except possibly the last level
last level must have its keys as left as possible.



④ Degenerate Tree : Every Parent node has exactly one child
↳ Skewed Tree



Graph

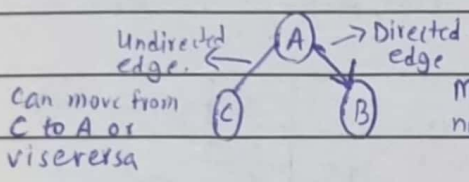
Date: _____

↳ Non-linear Data Structure. (Group DS) No level, No heirarchy.

A graph is a collection of nodes (vertex) connected through edges.

Used to model Paths in a city, social networks, websites backlinks, internal employee netw

If two vertices are connected they are called neighbours

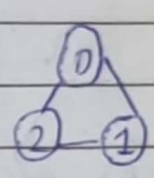


Indegree: No of edges going out of the node

Outdegree: No of edg

Adjacency List: Mark the nodes with the list of its neighbou

0 – 1→2
1 – 0→2
2 – 0→1

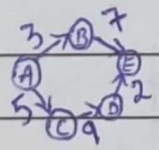


Adjacency Matrix: Aij = 1 for an edge between i and j, 0 otherwise

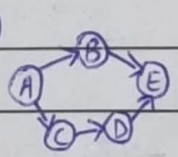
	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

Type of Graph: ①Directed Weighted ③Undirected Weighted
②Directed Unweighted ④Undirected Unweighted.

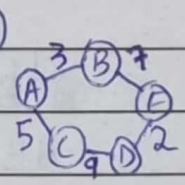
5 vertex so 5×5 matrix

① 

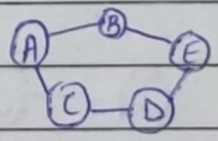
	A	B	C	D	E
A	0	3	5	0	0
B	0	0	0	0	7
C	0	0	0	9	0
D	0	0	0	0	2
E	0	0	0	0	0

② 

	0 A	1 B	2 C	3 D	4 E
0 A	0	1	1	0	0
1 B	0	0	0	0	1
2 C	0	0	0	1	0
3 D	0	0	0	0	1
4 E	0	0	0	0	0

③ 

	A	B	C	D	E
A	0	3	5	0	0
B	3	0	0	0	7
C	5	0	0	4	0
D	0	0	9	0	2
E	0	7	0	2	0

④ 

	0 A	1 B	2 C	3 D	4 E
0 A	0	1	1	0	0
1 B	1	0	0	0	1
2 C	1	0	0	1	0
3 D	0	0	1	0	1
4 E	0	1	0	1	0

BFS → Same → Level order traversal

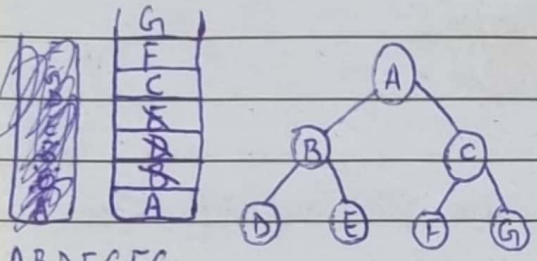
DFS → Prolly Same → Pre-Order.

Date: _____

Two Algorithms of Graph Traversal are:

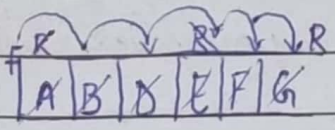
-> Breadth First Search (BFS) → Queue

-> Depth First Search (DFS) → Stack



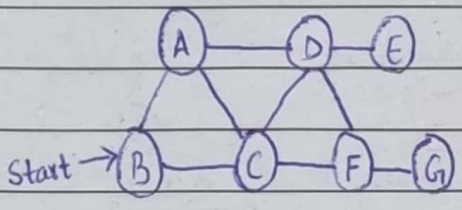
ABDECFG

↳DFS

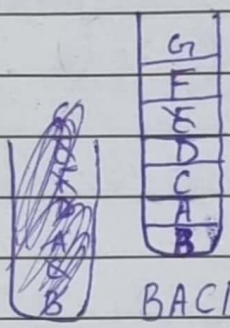


ABCDEFG → BFS

*) Once we enqueue so we mark that vertex as visited.



BACDFEG → BFS



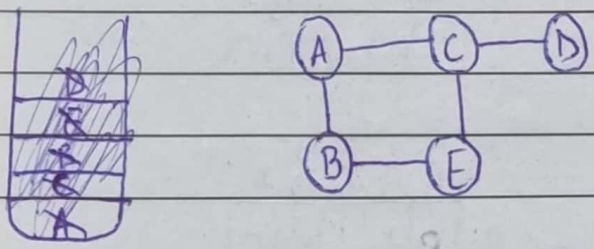
BACDEFG → DFS

DFS Process: ① Start by putting First of the graph's vertex on top of a stack.

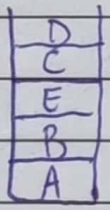
② Take the top item of the stack & add it to the visited list.

③ Add ones which aren't in the visited list to the top of the stack.

④ Keep repeating steps 2 & 3 until the stack is empty.



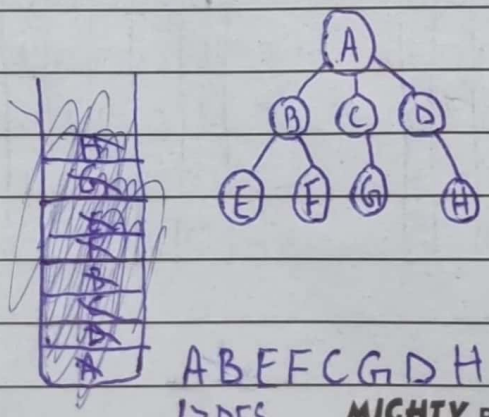
ABCED → BFS



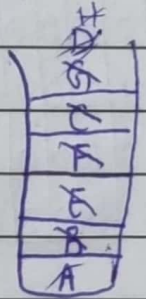
DFS → A, B, E, C, D

ABECD

↳DFS



ABCDEFGH → BFS



ABEFCGDH

↳DFS

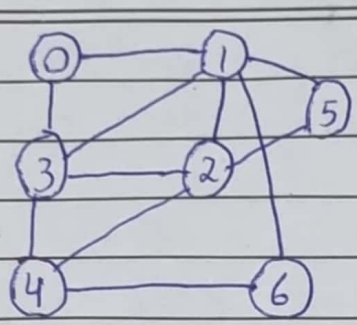
DFS → A, B, E, F, C, G, D, H

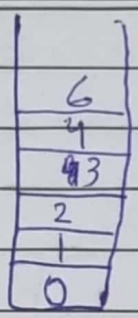
MIGHTY PAPER PRODUCT

Simply take any one child of start node insert it in stack & print it

Repeat until deadend then from top of

Date: _____

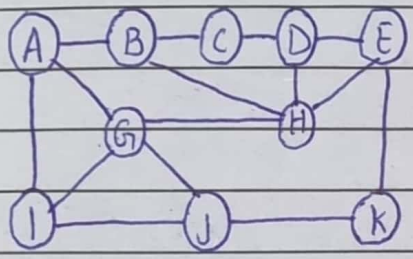




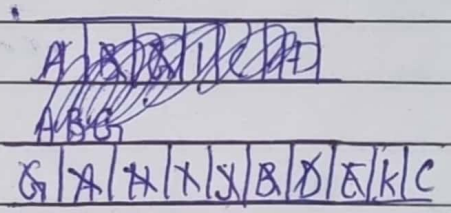
0	1	3	2	5	6	4

0,1,3,2,5,6,4 → BFS

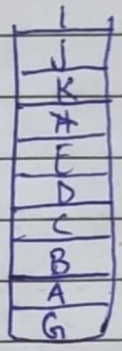
DFS → 0, 1, 2, 3, 4, 6, 5



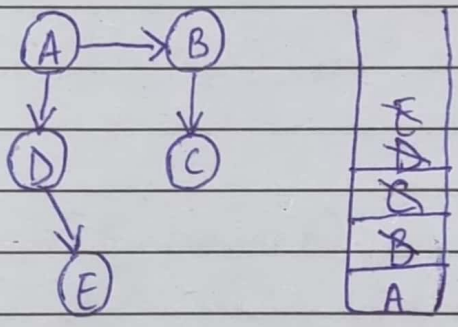
Start vertex is G



BFC → G, A, H, I, J, B, D, E, k, C



GABCDEHKJI → DFS



A	B	D	C	E

ABDCE → BFS

ABCDE → DFS