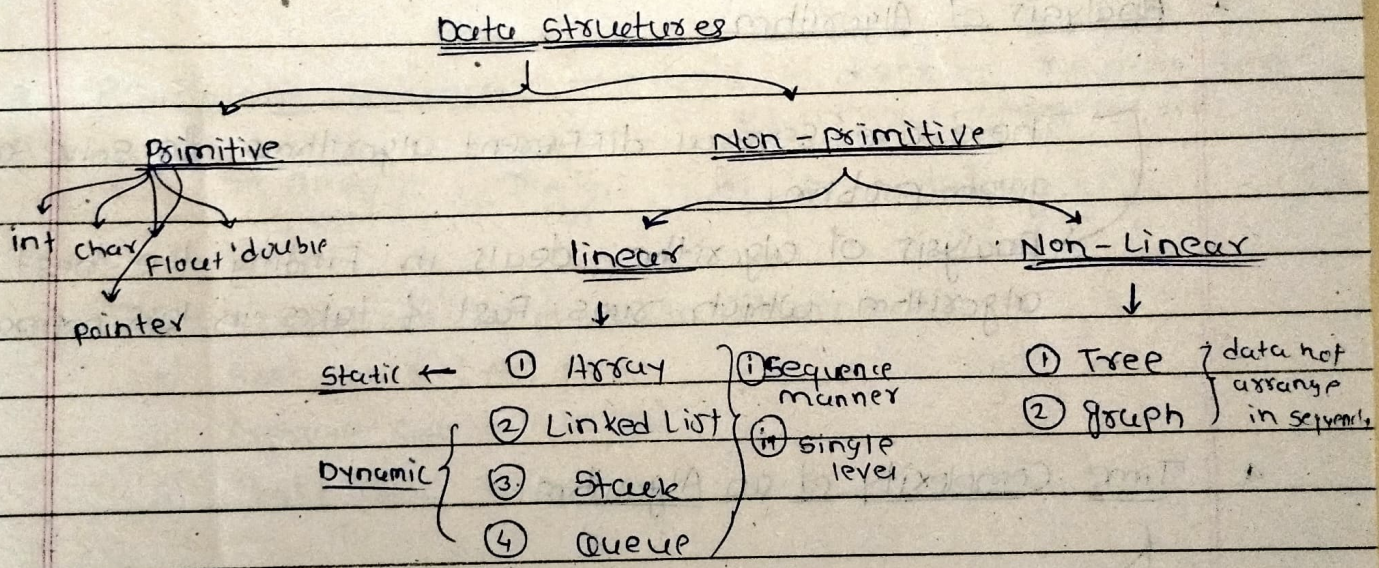
Pratik Boyade

Page No.:
Date: / /

* Data Structures
  ↳ It is a way to organize data.
  After organizing data it becomes easy to process it.

eg:-
organised book & unorganized books in library.

* Types of Data Structures



* Algorithms
  ↳ An Algorithm is a set of instruction to perform a task or to solve a given problem

eg:- ① A recipe book is a colleetion of recipes in which each recipe provides an step by step instruction to prepare food.
② let's say to prepare a tea
  (i) Boil water
  (ii) put tea in pot
  :
  (n) — — enjoy the tea. etc.

Page No.:
Date: / /

eg:- Print of 3 given numbers.

→ Let's write algorithm

① PerForm sum of 3 numbers
② Store it in a variable sum
③ Divide sum by 3
④ Store the value in variable avg
⑤ Print the value stored in avg.

code

```
Public Void FindAvg(int a,
                    int b, int c)
{
    int sum = a+b+c;
    int avg = sum/3
System.out.println(avg);
```

* Analysis of Algorithm

→ There are several different algorithms to solve a given problem.

→ Analysis of algorithm deals in Finding the best algorithm which runs Fast & takes in less memory.

* Time Complexity of an Algorithm

→ The amount of time taken by algorithm to run.

→ The input processed by an algorithm helps in determining the time complexity.

eg:-

```
public int findsum(int n)
{
    return n*(n+1)/2;
}
```

O(1)

It is Fast

```
Public int findsum(int n)
{
    int sum = 0
    For(int i=1; i<=n; i++)
    { sum = sum+1;
    }
}
```

It is slow

O(n)

* Space Complexity

→ The amount of memory or space taken by algorithm to run is called a Space Complexity.

→ The memory required to process the input by an algorithm helps in determining the Space Complexity.

* Asymptotic Notations → used to describe running time of algorithm (mathematical tool).
  - Omega (Ω), Theta (Θ) & Big 'O' (O) Notation.

These Notations helps us in determining
i) Best case (Ω) omega
ii) Average case Theta (Θ)
iii) worst case (Big O)

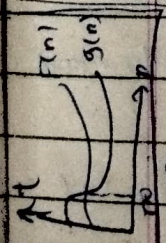
1] Omega (Ω) Notation → (Determines best case of an Algorithm)

Very rarely used

because we doesn't intrest to find the Best case

→ It is the Formal way to express the lower bound of an algorithm's running time.

→ Lower bound means For any given input this notation determines best amount of time an algorithm can take to complete.



eg:-

IF we say algorithm takes 100 sec as best amount of time. So, 100 sec will be lower bound of that algorithm.

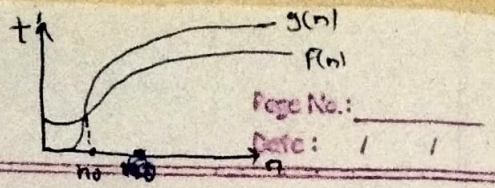
→ The algorithm can take more than 100 secs but it will not take less than 100 secs.

Shrilaxmi

$F(n) \leq C * g(n)$

C = constant

$n > n_0$



Page No.:

Date: / /

2] Big O (O) Notation → (Determine worst case of an Algorithm)

(Opposite of Ω)

→ It is the Formal way to express upper bound of an algorithm's running time

→ Upper bound means For any given input this notation determines longest amount of time an algorithm can take to complete.

For example :-

IF we say certain algorithm takes 100 sec as longest amount of time. So, 100 sec will be upper bound of that algorithm.

→ The algorithm can take less than 100 sec but it will not take more than 100 sec.

→ (Mostly used Algo) → we are very much intrested in Finding the max^m amount of time → so we can optimized algorithm to whiever time we want.

3] Theta (Θ) Notation → (Determine Average Case) (Upper + lower) Bound.

→ It is the Formal way to express the upper & lower bound of an algorithm's running time.

→ Determine average amount of time can take to comple

→ Very leastly used → because there are very rare used cases where we actually determine the average time.

Analysis Rules to calculate Big O Notation.

* It's a single processor
* It performs sequential execution of statements
* Assignment operation takes 1 unit of time (eg: int x=5;)
* Return statement takes 1 unit of time (eg: return x;)
* Arithmetical operation, Logical operation → 1 unit of time
* other small / single operation → 1 unit of time

1) Drop lower order terms

eg: $T = n^2 + 3n + 1$

lower order of non dominant term → Drop

$T = n^2$

Time Complexity = Big O(n^2)

2) Drop Constant multipliers

eg: $T = 3n^2 + 6n + 1$

$T = n^2 + 6n + 1$

$T = n^2$

Time Complexity = Big O(n^2)

3) Break the code into fragments

∴ Finally $T = T_1 + T_2 + T_3$

Page No.:
Date: / /

Time complexity of Constant Algorithm

eg:-

①
```
public int sum(int x, int y){
    int result = x+y;
    return result;
}
```

line no	operation	time
2	1+1+1+1	4
3	1+1	2
Total		6

$T = 4+2 = 6$ ← It always take the constant amount of time which is 6

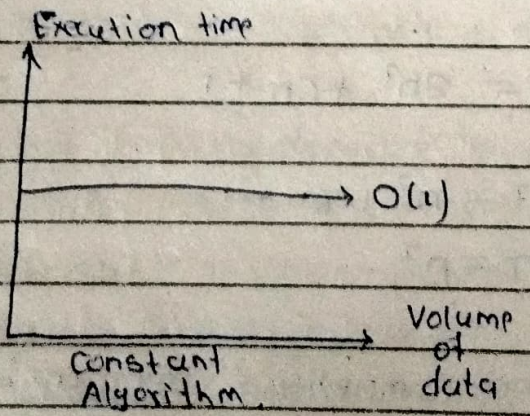
$T \approx C$ (constant)

So, Time complexity = Big O(1) ⇒ O(1)

②
```
public int get(int[], int i){
    return arr[i];
}
```

Time Complexity ⇒ O(1)

Graph



Time Complexity of Linear Algorithm

→ eg:

Find Sum of First n natural no.

```
Public Void FindSum(int n){
    int Sum = 0;        → (1 step) (k1)
    For (int i=1; i<=n; i++){
        Sum = Sum + i;  → (n step) (nk2)
    }
    return Sum;         → (1 step) (k3)
}
```

Time Complexity(T) = $T_1 + T_2 + T_3$

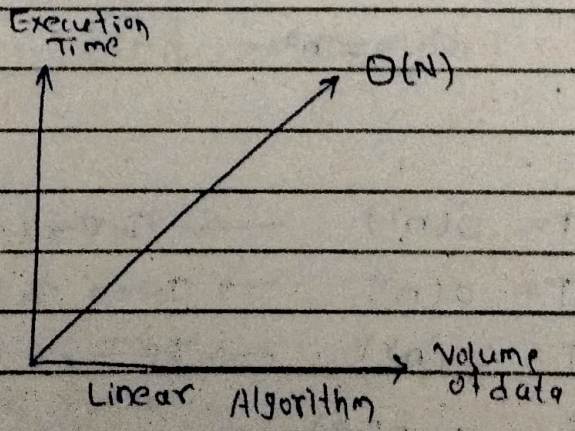
$= k_1 + nk_2 + k_3$

$= nk_2$

$T = n$

$\therefore \boxed{T = O(n)}$ ← Big $O(n)$

Graph



Page No.:
Date: / /

Time complexity of polynomial Algorithm

Eg:-

```
For(int i=1; i<=n; i++){
    For(int j=1; j<=n; j++){
        Sout("i=" + i + ", j=" + j);
    }
    Sout("End of inner loop");
}
Sout("End of outer loop");
}
```

↓

defines (For loop inside the For loop)

Time complexity (Tn) = $K_1 n + K_2 n$

$= n + n + n - (n-1)$

$= n(1 + 1 + (n-1))$

$= n^2 n$

$= n^2$

i

0 → 0 to n

1 → 1 to n

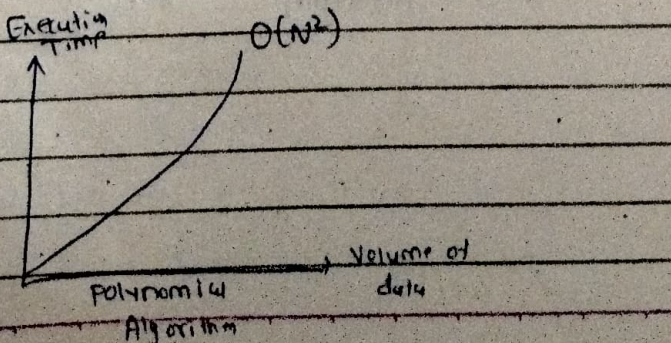
T = $O(n^2)$ → For → 2 For loop

T = $O(n^3)$ → IF → 3 For loop

T = $O(n^4)$ → IF → 4 For loop exist

Nested For Loop

Graph



* Practice Examples for Time Complexity

①
```
Void Fun(int array[], int length)
{
    int Sum = 0
    int Product = 1;                    } T1 = K1

    For(int i=0; i < length; i++)       } T2 = nK2
    {
        Sum += array[i];
    }

    For (int i=0; i < length; i++)      } T3 = nK3
    {
        Product *= array[i];
    }
}
```

→ $T = T_1 + T_2 + T_3$

$= K_1 + nK_2 + nK_3$

$= nK_2 + nK_3$

$= n(K_2 + K_3)$

$T = n$

$\boxed{T = O(n)}$ → (Big O of n)

2]
```
Void Fun(int n)
{
    int Sum = 0
    int Product = 1;                    } K1

    For(int i=0; i<n; i++) {  → 0 to n  } n+n+--(n-1)
      For (int j=0; j<n; j++){  → 0 to n }
        PrintF ("%d", i,j);
      }
    }
}
```

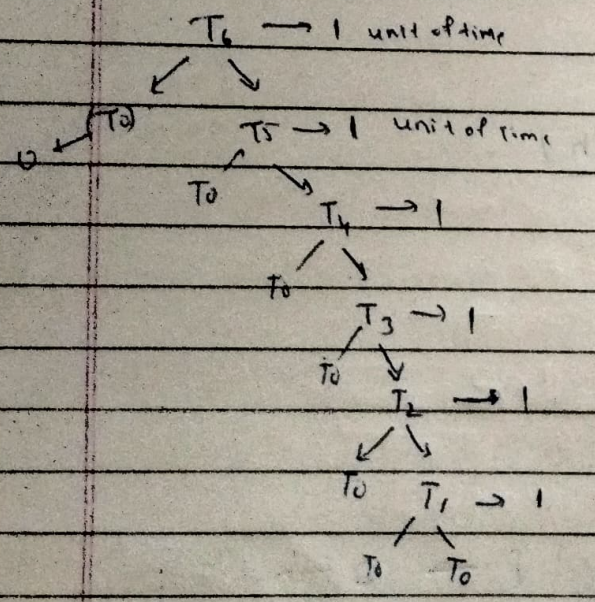
$T = K_1 + Kn^2$ $\quad T = O(n^2)$

Page No.:
Date: / /

(3) range (0, n), average Processing Time = T(n)
what is value of T(6) ?

```
int Function ( int n)
{
    int i ;                          ⟶ (K1)
IF (n<=0)
{   return 0;
}
else
{
    i = random (n-1) ;               ⟶ (1)
    Print F(" this in ") ;
    return Function (i) + Function (n-1-i) ;
}
}
```

⟶



~~6~~

T(6) = 6 ← no of calls

T = O(n)

④ which of the following are equivalent to O(N)? why?

i) O(N+P), where P ∈ N/9 → non dominant term

$T = O(N+P)$

$\boxed{T = O(N)}$

ii) O(9N − k)

$T = O(9N-K)$

$T = O(9N)$

$\boxed{T = O(N)}$

iii) O(N + 8 log N)

→ $T = O(N+8\log N)$

$\boxed{T = O(N)}$

iv) O(N + M²) → m is not constant

→

$T = O(N+M^2)$

$T = O(N+M^2)$

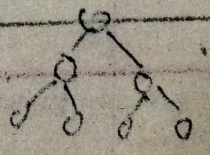
→ This is not O(N)

⑤ Following code for balanced binary search tree, what is runtime,

→

```
int sum(Node node)
{  if(node == NULL)
   {  return 0;
   }                                  } k1
   return sum(node.left) + node.value + sum(node.right);
}
```

$\boxed{T = O(N)}$ $\boxed{T = O(N)}$



Page No.:
Date: / /

Q Find the complexity of Following code which test whether given no is prime or not?

```
int isPrime(int n){
    if (n==1) {
        return 0;
    }
    For (int i=2; i*i<n; i++) {
        if (n%i==0)
            return 0;
    }
    return 1;
}
```

K_1

$K_2(\sqrt{n})$

→

Suppose $i*i < n$

$i^2 < n$ $\quad i = 2, 3, \cdots (n-1)$

$\boxed{i < \sqrt{n}}$

$i = \sqrt{n} - 1$

$\boxed{i = \sqrt{n}}$

So $T = T_1 + T_2$

$= k_1 + \sqrt{n}\, k_2$

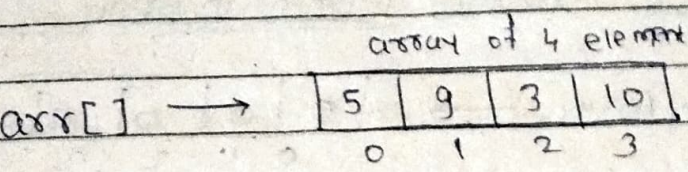
$T = \sqrt{n}$

$\boxed{T = O(\sqrt{n})}$

Array

Page No.:
Date: / /

* Array



* It's a collection of data elements of specified type
* All data have contigious memory locations.
* Each partition has two neighbours except First & last one
* Size of array is Fixed & cannot be modified once it is created
* Index starts at 0 and for (a 1D array) ends at (length - 1)

* Declaration (1D)

Syntax :-

```
DataType arrayName[];
DataType[] arrayName;
```

eg :- `int myArray[];` { use most for Java → mostly used
or `int[] myArray;`

* Initialization → It gives memory to array elements.

Syntax :-

```
arrayName = new dataType[size];
```

eg :- `myArray = new int[5];`

* → * Declaration & Initialization in one step (Java)

```
dataType[] arrayName = new dataType[size];
```

eg :- `int[] myArray = new int[5];` // `int myArray = {1,4,3,1,1};`

Adding & Updating element in Array:

myArray → | 0 | 0 | 0 | 0 | 0 |
0 1 2 3 4

when we initialize an array

```
Public void arrayDemo() {

    int[] myArray = new int[5];   → | 0 | 0 | 0 | 0 | 0 |
    myArray[0] = 5;               → | 5 | 0 | 0 | 0 | 0 |
    myArray[1] = 1;               → | 5 | 1 | 0 | 0 | 0 |
    myArray[2] = 8;               → 5 1 8 0 0
    myArray[3] = 2;               → 5 1 8 2 0
    myArray[4] = 10;              → 5 1 8 2 10
(update)→ myArray[2] = 9;         → 5 1 9 2 10
}
```

Print elements of array in Java

→

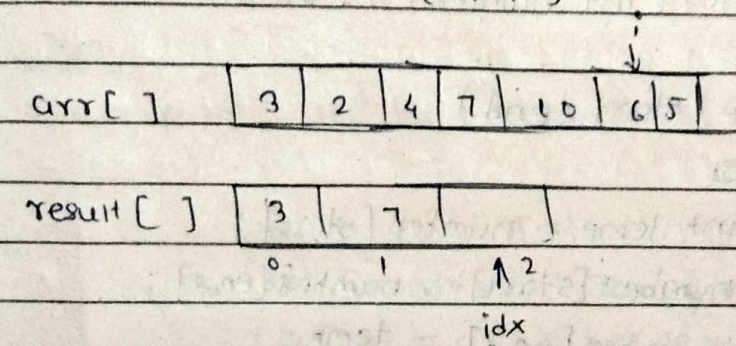
```
Public void printArray(int[] arr)
{
    int n = arr.length;
    For (int i = 0; i<n; i++)
    {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}
```

Removing integer from an array.

Q] Given an array of integers, return an array with even integers removed

Input : arr = {3, 2, 4, 7, 10, 6, 5}

Output : arr = {3, 7, 5}



(oddCount)

removeEven(arr);

```
public static
→ int[] removeEven(int[] arr)
  {
      int oddCount = 0;
      For(int i=0; i<arr.length; i++)
      {
          iF(arr[i] %.2 != 0)
          {
              oddCount++;
          }
      }
```

First we count how many odd numbers are present in oddCount

```
      int[] result = new int[oddCount];
      int idx = 0;
      For(int i=0; i<arr.length; i++)
      {
          ~~arr~~if(arr[i] %.2 != 0)
          {
              result[idx] = arr[i];
              idx++;
          }
      }
      return result;
  }
```

→ create another new array for passing value to it

↖ size

Page No.:
Date: / /

How to Reverse an Array in Java

Q] Given an array, the task is to reverse array or string

Given - {2, 11, 5, 10, 7, 8}
0 1 2 3 4 5

output :- {8, 7, 10, 5, 11, 2}

→

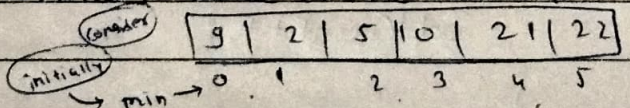
```
reverseArray (int numbers[ ], int start, int end)
{
    while (start < end)
    {
        int temp = numbers[start];
        numbers[start] = numbers[end];
        numbers[end] = temp;
        start ++;
        end --;
    }
}
```

At starting

start = 0
end = 5

To Find the minimum value of the Array.

→




```
Public int FindMinimum (int[ ] arr)
{
    int min = arr[0];          // initially consider First value is min
    For (int i ; i < arr.length; i++)
    {
        IF (arr[i] < min)
        {
            min = arr[i];
        }
    }
return min;
}
```

* Find the second maximum value in an Array.

→

arr[] | 13 | 34 | 2 | 34 | 33 | 1 |
0 1 2 3 4 5

initially second max & max value are take minimum.

Program

```
public int FindSecondMax(int[] arr)
{
    int max = Integer.MIN_VALUE
    int Secondmax = Integer.MIN_VALUE
    For(int i=0 ; i< arr.length ; i++)
    {
        IF(arr[i]> max)
        {
            Secondmax =max;
            max = arr[i];
        }
        else if( arr[i] > secondmax && arr[i]!=max)
        {
            Secondmax= arr[i];
        }
    }
    return secondmax;
}
```

Page No.:
Date: / /

Q How to move zeroes to the end of Array

→ arr[] | 8 | 1 | 2 | 1 | 0 | 0 | 3 |

output → | 8 | 1 | 2 | 1 | 3 | 0 | 0 |

- Program :-

```
public void movezeroes(int[] arr, int n)
{
    int j=0;
    for(int i=0; i<n; i++)
    { if(arr[i]!=0 && arr[j]==0)
      {
          int temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
      }
      if(arr[j]!=0)
      {
          j++;
      }
} }
```

Q Find the missing no in array of given range

(Imp) Q1→ Given an array of (n-1) distinct numbers in the range of 1 to n. Find the missing number in it.

| arr[] | 2 | 4 | 1 | 8 | 6 | 3 | 7 |

output :- 5

Explaination :- The missing From range 1 to 8 is 5.

Shrikrupa

math Formula to Find the Sum of n natural number

$$= \frac{n*(n+1)}{2}$$

Page No.:
Date: / /

Program :-

```
public int findMissingNumber(int[] arr)
{
    int n = arr.length +1;
    int sum = n*(n+1)/2;
    for (int num : arr)
    {
        sum = sum - num;        // Subtract one by one all values in array from sum
    }                           // we will get the missing no.
    return sum;
}
```

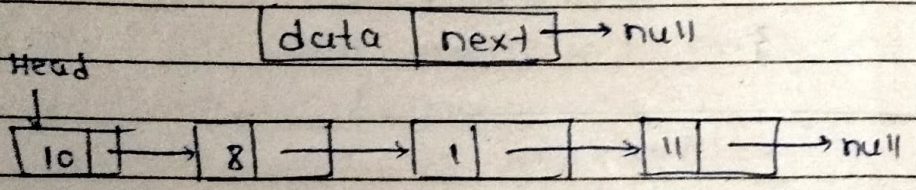
Singly Linked List

Page No.:
Date: / /

* Singly Linked List → Linear DS
  → Non-contiguous memory allocation

Singly linked list is a data structure used for storing collection of nodes (stores an object in node) & has following properties

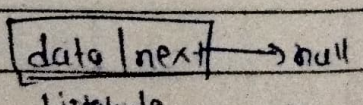
* It contains sequence of nodes
* A node has data and reference to next node in a list
* First node is the head node.
* Last node has data and points to null.



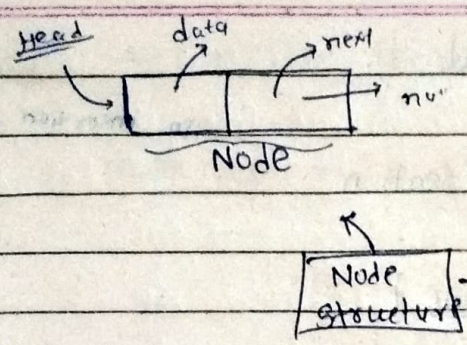
* Implementation of ListNode in SinglyLinked List

→
```
// Generic Type
public class ListNode <T>
{
    private T data;
    private ListNode <T> next;
}
```

```
// Integer Type
public class ListNode
{
    private int data;
    private ListNode next;
}
```



Rough Implementations



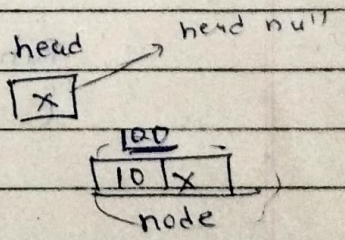

```
Static class Node
{   int data;
    Node next;

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

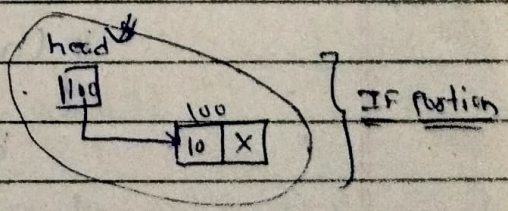
Cond^n

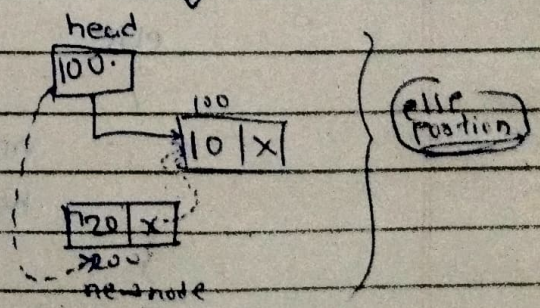
1. Linked list exist
2. Linked list does not exist

```
IF (head == null)
{
    head = node;
}
else
{
    head = new_node;
    new_node.next = head;
    head = new_node;
}
```



node = 100
node.data = 10
node.next = null





Page No.:
Date: / /

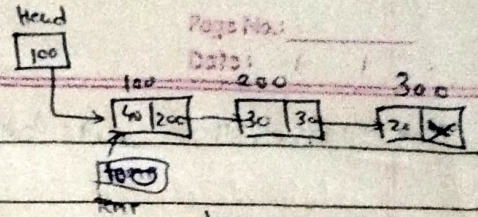
Creation of linked List

→ also insertion at begin

```
Public class Single_Creation
{
    Static void Node {
        int data;
        Node next;
        Node (int data)
        {
            this.data = data;
            this.next = null;
        }
    } Node head = null;
    Public void Creation()          → Insertion at begin (same as creation)
    {
        int data n;
        Scanner sc = new Scanner (System.in);
    do{ Sout ("Enter data");
        data = sc.nextInt();
        Node new-node = new Node (data);
      if (head == null)
        {
            head = new-node;
        }
      else
        {
            new-node.next = head;
            head = new-node;
        }
      Sout (" do you want to continue. If yes, press 1");
      n = sc.nextInt(1);
    }
    while(n==1); }
```

(assign data to new node)

new node → creation



For traversing → we use temp

so

Node temp = head;

temp creation

temp = 100

temp.data = 40

temp.next = 200

Full code with data continue

```
public void traverser()
{   Node temp = head;
    if (head == null)
    {
        Sout("LL does not exist");
    }
    else
    {
        while(temp != null)
        {
            Sout(temp.data);
            temp = temp.next;
        }
    }
}
public static void main(String args[])
{
    Single_Creation l = new Single_Creation();
    l1.creation();
    l1.traverser();
}
```

Page No.:
Date: / /

Insertion at begin, end & position of (Single LL)

At begin

we have to use new node as a ptr

```
new-node.next = head;
head = new_node;
```

At end

we have to use ptr ① temp

```
Node temp = head;
while (temp.next != null)
{
    temp = temp.next
}
temp.next = new_node;
```

At position

```
Scout (" Enter the po of node to be inserted ");
P = sc.nextInt();
```

we have to use two pointer ① new node ② temp1

```
Node temp1 = head;
For (int i = 0; i<(P-2); i++)
{
    temp1 = temp1.next;
}
new-node.next = temp1.next;
temp1.next = new-node;
```

i<P-2

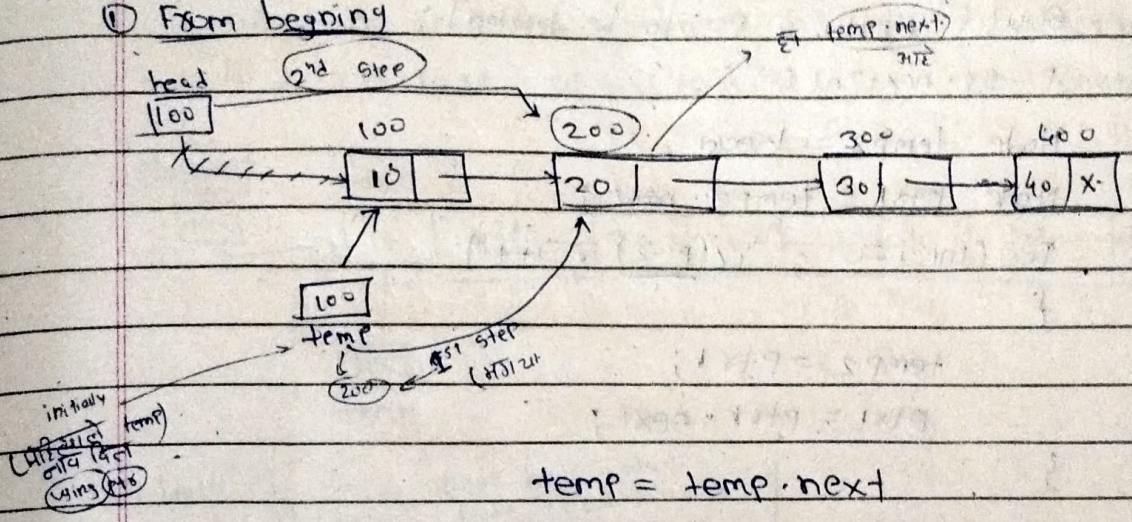
bb

Poutik Boyade (ABC)

Page No.:
Date: / /

* Deletion of linked list

① From beginning



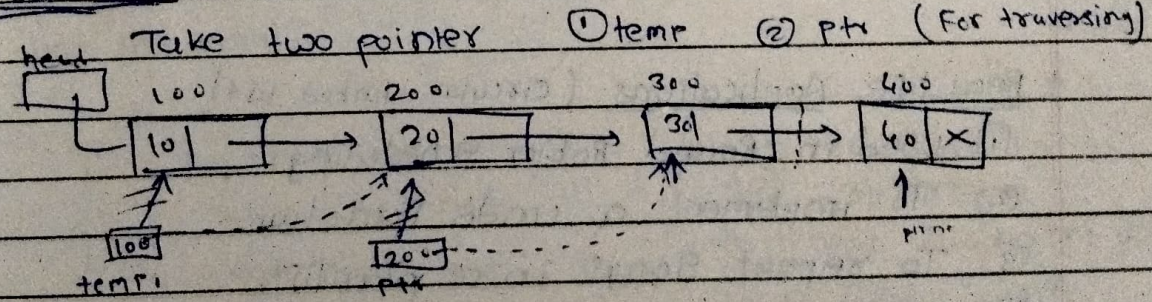
temp = temp.next

head = temp

Code

```
{
    Node temp = head;
    temp = temp.next;
    head = temp;
}
```

② From end



→

```
Node temp1 = head;
Node ptr = temp1.next;
while (ptr.next != null)
{   temp1 = ptr;
    ptr = ptr.next;
}
temp.next = null;
```

③ Delete From po

```
Sout (" Enter a po to be deleted");
p = sc.nextInt();
Node temp2 = head;
Node ptr1 = temp2.next;
For (int i=0 ; i<(p-2) ; i++)
{
    temp2 = ptr1;
    ptr1 = ptr1.next;
}
temp2.next = ptr1.next;
```

Circular linked List

The circular linked list is a linked list where all are connected to form a circle.

In a circular linked list, the first node and last node are connected to each other which forms a circle.

→ There is no null at the end.

Real life Applications (circular linked list)

① Used in Round Robin Scheduling
② To implement a undo Function
③ To repeat Songs in a play list.
④ To keep track of the turn in a multiplayer game.

Time complexity → ① For traversing in singly linked list = O(n)
② In circular linked list (For traversing) = O(1)

↑ Because in circular linked list, we have known the address of last node (tail)

Shrikrupa

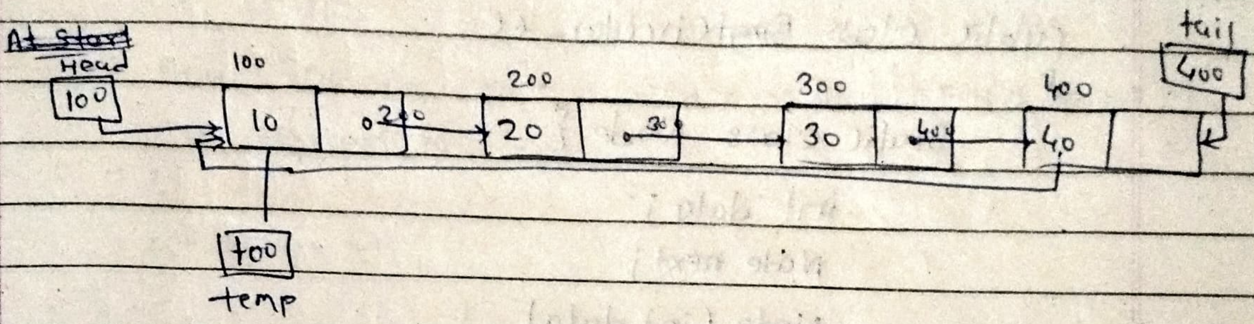
(Rough work)

Page No:
Date: / /

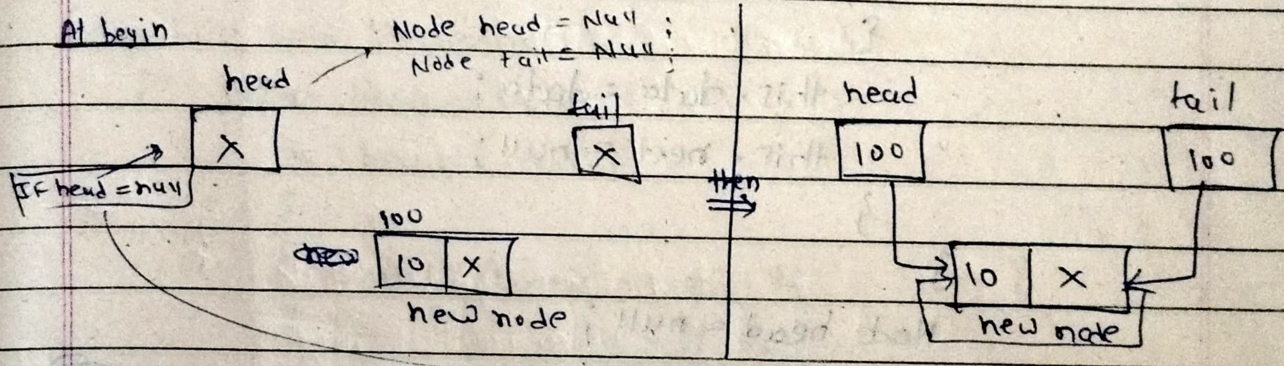
+ Creation

→ Creation is Same as an linear linked list, only we we have create extra point 'tail' at end.

→ And In last asign(tail = head). IF Insertion take plac.



At begin



head = newnode

tail = new-node

new-node.next = Head/tail/new_node

कोणतंपण घेऊ शकतो

IF link list exist

else part

newnode.next = head

head = new-node

tail.next = head;

* only tail part extra आहे आपल्याला tail.next हे starting ला join करावं लागतो.

Code :- Creation (at begin) & traversing

```
→ import java.util.Scanner;

public class Circular_ll
{
    static class Node {
        int data;
        Node next;
        Node (int data)
        {
            this.data = data;
            this.next = null;
        }
    }
        Node head = null;
① *     Node tail = null;
    public void creation ()
    {
        int data, n;
        Scanner sc = new Scanner(System.in);
        do
        { System.out.print ("Enter data");
          data = sc.nextINt();
          Node new_node = new Node(dala);
          IF (head = null)
          {
              head = new_node;
② *           tail = new_node;
③ *           new_node.next = Head;
          }
```

इथा Head, tail, new_node काहि पण चालते

```
        else
        {       new_node.next = head;
                head = new_node;
                tail.next = head;
        }
        Sout("Do you want to add more data. If yes press 1:");
        n = sc.nextInt()
    }
    while(n==1);
}

public void traverser()
{   Node temp = head;
    If (head==null)
    {
        Sout("LL does not exist");
    }
    else
    {
        do {
            Sout(temp.data + " ");
            temp = temp.next;
        } while (temp!=head);
    }
}

public static void main(String args[])
{
    Circular_ll a = new Circular_ll();
    a.creation();
    a.traverser();
}
}
```

④

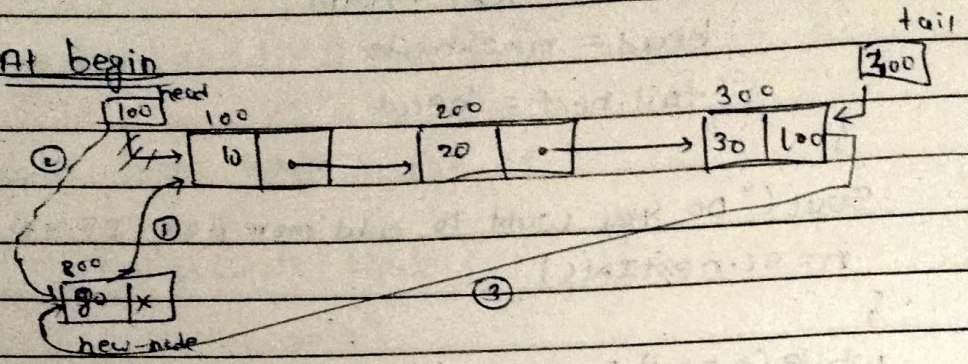
⑤

⑥ temp!=head
not null

Page No.:
Date: / /

Circular LL (Insertion)

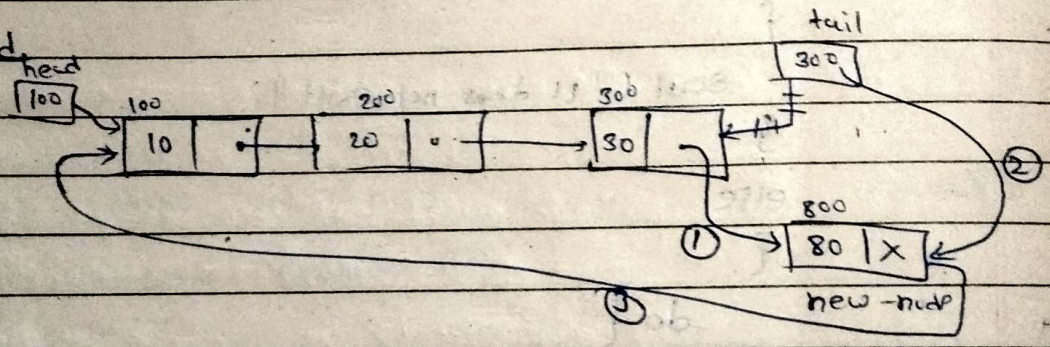
① At begin



Code

```
new-node.next = head;
head = new-node;
tail.next = head;
```

② At end



Code

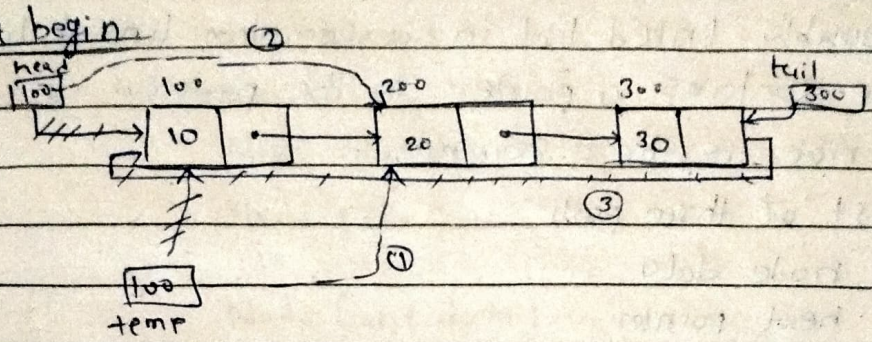
```
tail.next = new_node;
tail = new_node;
new_node.next = head;
```

③ Insert at Po

↳ It is same as Insertion of Linear linked list (no change).

Circular LL (Deletion)

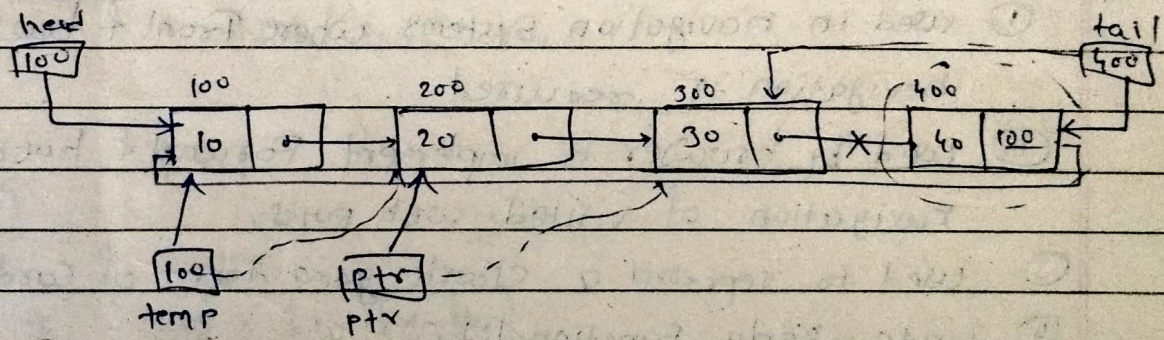
① At begin



code

```
temp = temp.next;
head = temp;
tail.next = head;
```

② At end



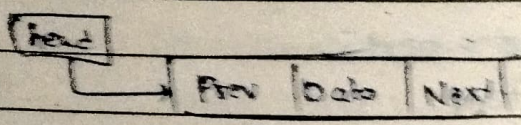
code

```
Node temp = head;
Node ptr = temp.next;
While (ptr.next != head)
 {
    temp = ptr;
    ptr = ptr.next;
 }
    temp.next = head
    tail = temp;
```

Doubly linked list

Doubly linked list is a complex linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

* Consist of three parts
  - node data
  - next pointer
  - previous pointer



Real life Applications

1. Used in navigation systems where front & backward navigation is required
2. Used by browser to implement forward & backward navigation of visited web pages.
3. Used to represent a classic game deck of cards.
4. undo - Redo Functionality.

Time Complexity

Insertion = O(1)

Traversal = O(n)

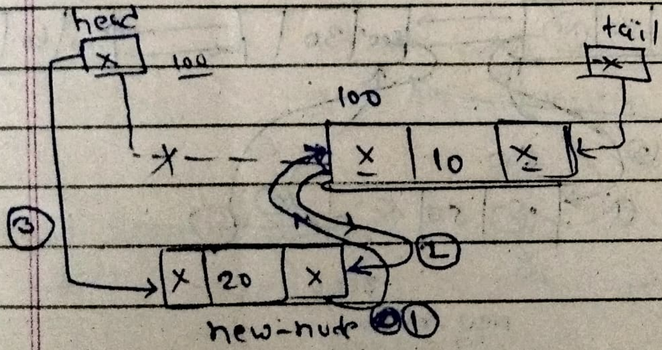
Space Complexity = O(1)

Page No.:
Date: / /

Structure

```
static class Node {
        int data;
        Node next;
        Node prev;

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



② head.prev = new-node ;

① new-node.next = head ;

③ head = new-node;

Creation

```
IF (head = null)
    {   head = new-node;
        tail = new-node;
    }
    else {  head.prev = new-node;
            new-node. next = head;
            head = new-node;
    }
```

आगे पर इसके

Note
↳ traversing is same as singly.

Page No.:
Date: / /

Insertion (doubly)

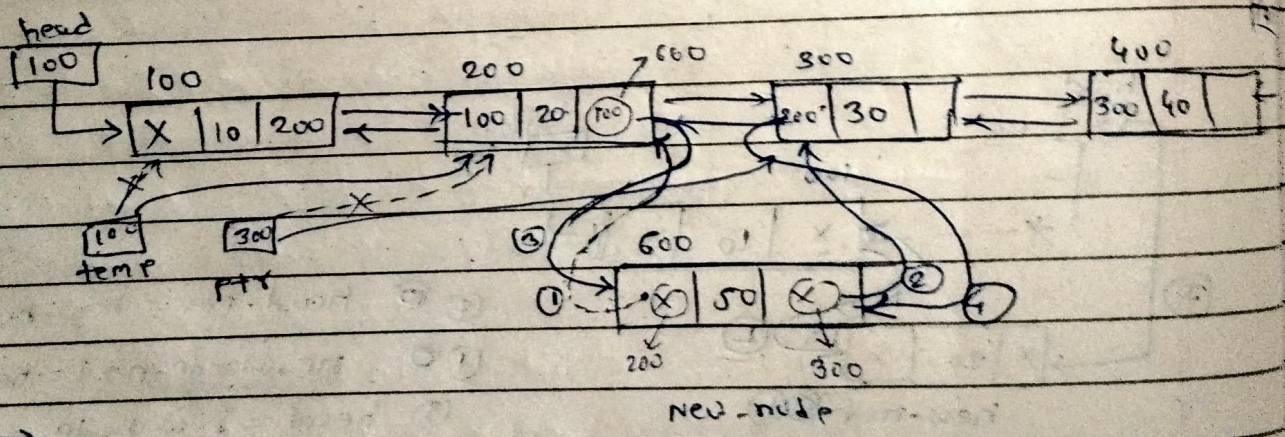
① At begin

```
new-node.next = head;
head.prev = new-node;
head = new-node;
```

② At end

```
tail.next = new-node
new-node.prev = tail;
tail = new-node;
```

③ Specific location

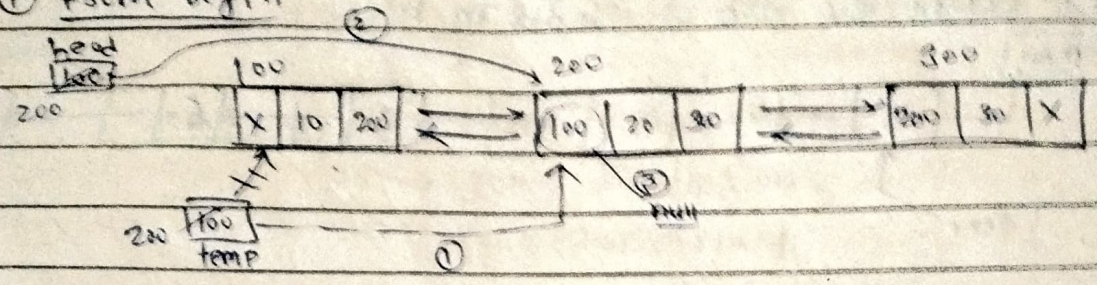


→

```
For(i=1; i<p-1; i++)
{ temp = ptr;
  ptr = ptr.next;
}
new_node.prev = temp;
new-node.next = ptr;
temp.next = newnode;
ptr.prev = newnode;
```

Deletion (Doubly)

① From begin




```
Node temp = head;
① temp = temp.next;
   head = temp;
   head.prev = null;
```

② From end

```
Node temp = tail;
temp = temp.prev;
temp.next = null;
tail = temp;
```

③ Specific location

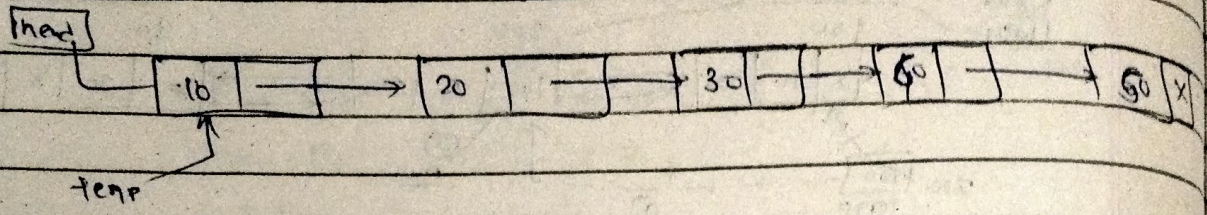
→

```
For (i=1; i<p-1; i++)
{   temp = ptr;
    ptr = ptr.next;
}
temp.next = ptr.next
ptr.next.prev = temp;
```

Page No.:
Date: / /

Practice Questions

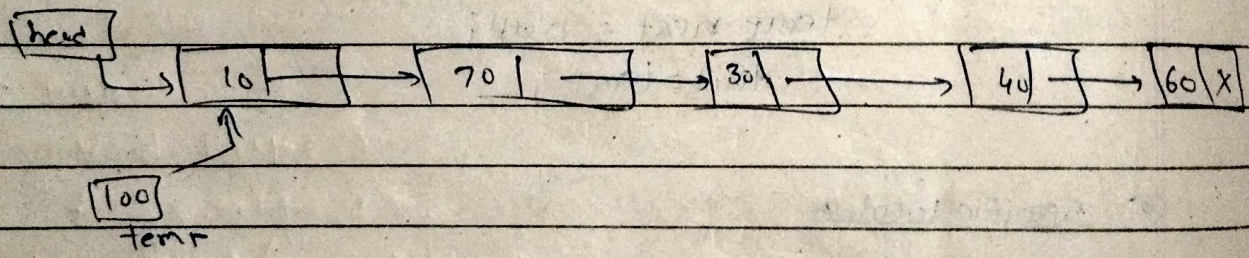
① Count the no of Nodes in LL



→

```
int c = 0;
Node temp = head;
while (temp != null)
{
    c++;
    temp = temp.next;
}
Sout (c);
```

② Find maximum element



→

```
int max = 0;
Node temp = head;
while ( temp != null)
{
    If (temp.data > max)
    {
        max = temp.data;
    }
    temp = temp.next;
}
```

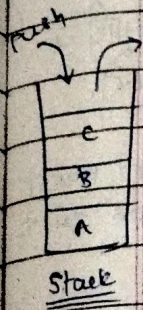
Shrikrupa

Stack

Page No.:
Date: / /

LIFO (Last in First out)

Stack :- → Stack is a linear data structure
→ Follows LIFO order for performing operations



Operations → Push = insertion
→ POP = Deletion.
→ Peak (top of stack)

* Real World Application of Stack

1. Used in Recursion
2. Undo operation
3. Web browser (history)
4. Tower of Hanoi (Algorithm of Stack)
5. Functions calls.

→ * Explain why Stack is a recursive data structure?

→

① Recursion :- It is a technique of solving any problem by calling same Function again & again until some breaking condition where recursion stops & it starts calculating the solution from there.

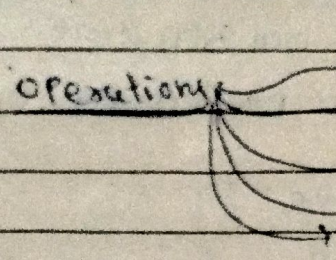
*→ ② Thus, In recursion last Function called needs to be completed first.

③ So, Stack is a LIFO data structure (Last in First out) & Hence, it is used to implement in recursion.

* Time & space complexity of Stack

	Best	Space
Push	O(1)	O(1)
POP	O(1)	O(1)
Peak	O(1)	O(1)
	↑ Time	↑ Space.

1 Implementation of Stack (using Array)



(top = -1)

Array मध्ये आपल्याला size define करायची होती.
Suppose A[10];
So, Size पेक्षा जास्त element ला करायला गेले overFlow cond"

* underFlow cond" → Check in case of (pop)
* overFlow cond" → Check in case of (push)

1] Void underFlow ()
{
  if (top == -1)
  {
    Sout ("underFlow cond");
  }

② Void overFlow
{
  if (top = (n-1)) (size of array)
  {
    Sout ("overflow"
  }

3) For Push

Void Push ( )
{ overFlow ( ) ; → First check overFlow cond" (IF part)
  Sout ("Enter data").;
  int i = sc.nextInt ( ) ; } else part
  top = top + 1 ;
  a [top] = i ;
}

(array we declare)

pop = top--
push = top++

Page No.:
Date: / /

4] For pop

```
Void pop()
{
  if (top == -1)
  {
    Sout("UnderFlow");
  }
  else
  {
    top = top+1;
  }
```

5] For Display

```
void display()
{  Sout("Items are");
   for(int i = top ; i >=0 ; i++)
   {
      Sout(a[i]);
   }
}
```

* Time Complexity

	Best	Worst
Push	O(1)	O(n)
pop	O(1)	O(n)
display	O(n)	O(n)

Page No.:
Date: / /

Implementation of Stack (using Linked list)

Note :- ① Stack चे Linked list मध्ये implementation करताना जो 'Head' असतो त्यालाच (Top) ठेवायचे.

② Insertion (Push) आणि Deletion (Pop) हे फक्त beginning ला होते.

③ Linked list मध्ये कधीच overflow condition येत नाही. कारण Stack सारख Linkedlist मध्ये size आधीच Define नाही करावी लागते.

④ Time complexity पण Linked list ची O(1) असते. कारण, आपल्याला फक्त Beginning ला operation करावे ल

① Underflow

```
if (top == -1)
{
    Sout("Underflow");
}
```

②
```
Void Push()
{
    Sout("Enter data");
    int data = sc.nextInt();          //Scanner class
    Node new_node = new Node(data);
    if (top == null)
    {
        top = new_node;
    }
    else
    {
        new_node.next = top;
        top = new_node;
    }
}
```

Page No:
Date: / /

③ void pop( )

```
{
    if (top == null)
    {
        Sout ("stack empty");
    }
    else
    {
        ~~temp = temp.next~~ top = top.next;
    }
}
```

→ इस आपण एक pointer घेऊन पण delete केला नाही.

④ void display( )

```
{
    Node temp = top;
    while (temp != null)
    {
        Sout (" temp.data");
        temp = temp.next;
    }
}
```

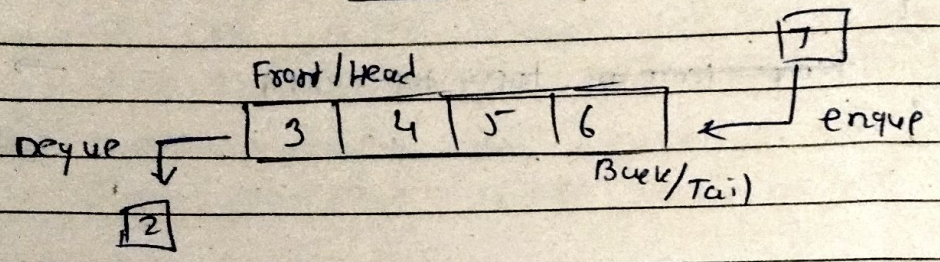
Time complexity	Best case	worst case
push( )	O(1)	O(1)
pop( )	O(1)	O(1)
display( )	O(n)	O(n)

Queue

Page No.:
Date: / /

* **Queue** :- Queue is a linear data structure that is open at both ends.
  - Follows First in First out order for perform operation (FIFO)

* Addition of element is done at one end & deletion is done at another end.



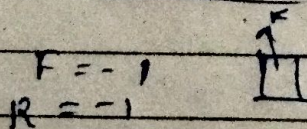
Queue Data Structure

* **Characteristic**
  * Queue can handle multiple data.
  * We can access both ends.
  * They are Fast & Flexible.

Operation

- Enque → Insertion → occur at end
- Deque → Delition → occur at start
- Underflow → check when underflow cond"
- Overflow → Check when enque cond"
- Display

① **Underflow cond"**




```
IF (F==-1 || R==-1)
{
    // underflow
}
```

② **Overflow**


```
if (R == (N-1))
{
    //overflow
}
```

Shrilaxmi

```
int f = -1 , r = -1;
int n = 10;
int q[ ] = new int[n];
```

Page No.:
Date: / /

③ Enque (using Array)

```
Void enque( )
{
  if(r=(n-1))
  {  Sout(" overflow");
  }
  else
  {  Sout("Enter data:")
     int i = sc.nextInt();
     if ( f== -1 && R == -1)
     {
        F = 0;
        r = 0;
        q[r] = i;
     }
     else
     {  r = r + 1;
        q[r] = i;
     }  }
}
```

④ Deque

```
Void Deque ( )
{
  if (f == -1 || r == -1)
  {
    Sout (" underflow");
  }
  else
  {
    F = F + 1;
  }
```

⑤ Display

```
void display ( )
{
  Sout ("Items are");
  For (int i=F ; i<=r ; i++)
  {
     Sout (q[i]);
  }
}
```

Node f = Null;
Node r = Null;

Page No.:
Date: / /

* Queue (using linked list)

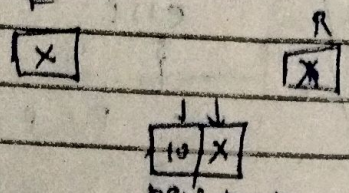
① Underflow

```
if (f == null & r == null)
{
    sout ("underflow")
}
```

② Overflow

```
if (new == null)
{ sout("overflow");
}
```

③ Enque



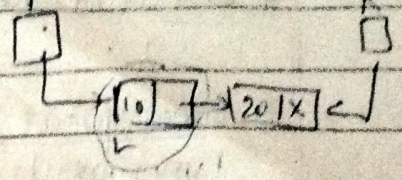

```
Public void enque ();
if (f == null)
{
    f = new-node;
    R = new-node;
}
else
```

④ Public void enque ()
{

```
Sout ("Enter data");
int data = sc.nextInt();
Node new-node = new Node(data);
if(f==null)
{
    F = new-node;
    R = new-node;
}
else
{
    R.next = new-node;
    R = new-node;
}
```

④ public void Deque ( )

```
{
    if (F == null)
    {
        Sout ("Underflow");
    }
    else
    {
        F = F.next;
    }
}
```



(F = F.next)

⑤ Display

```
Node temp = F;
while (temp != null)
{
    Sout(" ", temp.data);
    temp = temp.next
}
```

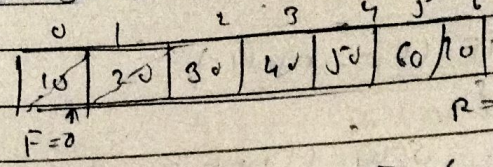
* Time Complexity (Queue)

Queue	Array	LL
Enque	O(1)	O(1)
Deque	O(1)	O(1)
Display	O(n) ↓ Fix(size)	O(n)

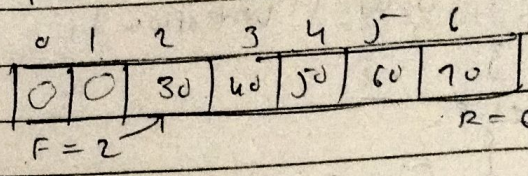
Page No.:
Date: / /

Circular Queue using Arrays

• Linear queue (using Array) →



(After deleting first two element)



(only exist in case of arrays) ⟶ Wastage of space occour in linear queue.

(The use of circular que can overcome this problem)

• Circular Queue

① Front end is connected to rear end.

* Cond" For the que to be circular que

① Front == 0 & rear = (n-1)

② Front = rear + 1

* Comparision

		Linear	Circular
①	Insertion & Deletion	Insertion is done at rear end, Deletion is done at Front end	Insertion & Deletion can take place at any end
②	Memory space	Require more space than circular que	require less space compared to linear que
③	Memory utilization	The use of memory is inefficient	The memory can be more efficiently utilized
④	order	Follows FIFO	no specific order for execution

Sirilarupa

Page No.:
Date: / /

Linear Queue

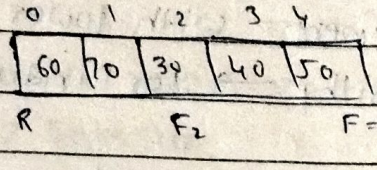
1] Enque ⇒ R = R + 1

2] Deque ⇒ F = F + 1

Circular Queue

1) Enque ⇒ R = (R+1) % max size

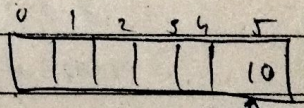
2) Deque ⇒ F = (F+1) % max size



R = (0+1) % 5 = 1 % 5 = 1

	Linear	Circular Queue
1] Overflow	f(R == (max size - 1))	f(F == (R+1) % max size)
2] Underflow	f(F == -1 && R == -1)	f(F == -1 && R == -1 || F == R)

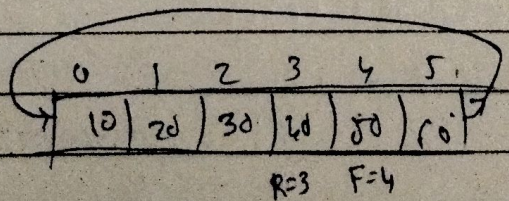
For (F=R) ⟶ means only one element is present in queue



F = R → so underflow exist

extra else if condn in Deque

③ Display




```
For (i=F; i!=R; i = (i+1) % n))
  {
    sout (q[i]);
  }
```

Page No.:
Date: / /

Priority Queue → implements min Heap

① A priority queue is a type of queue that arranges element based on their priority values.

② Element with higher priority values are typically retrieved before elements with lower priority values.

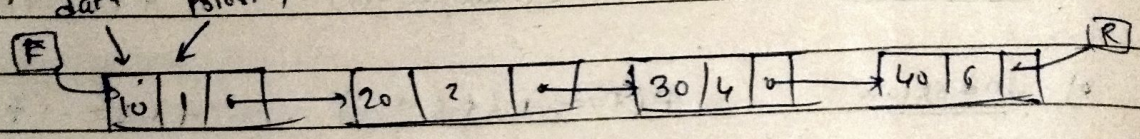
Application

③ used is Dikstra Algorithm → For Finding Shortest path

④ Prim's Algorithm

⑤ Heap Sort → (use in Implementation)

⑥ Operating system → we in OS For load balancing.



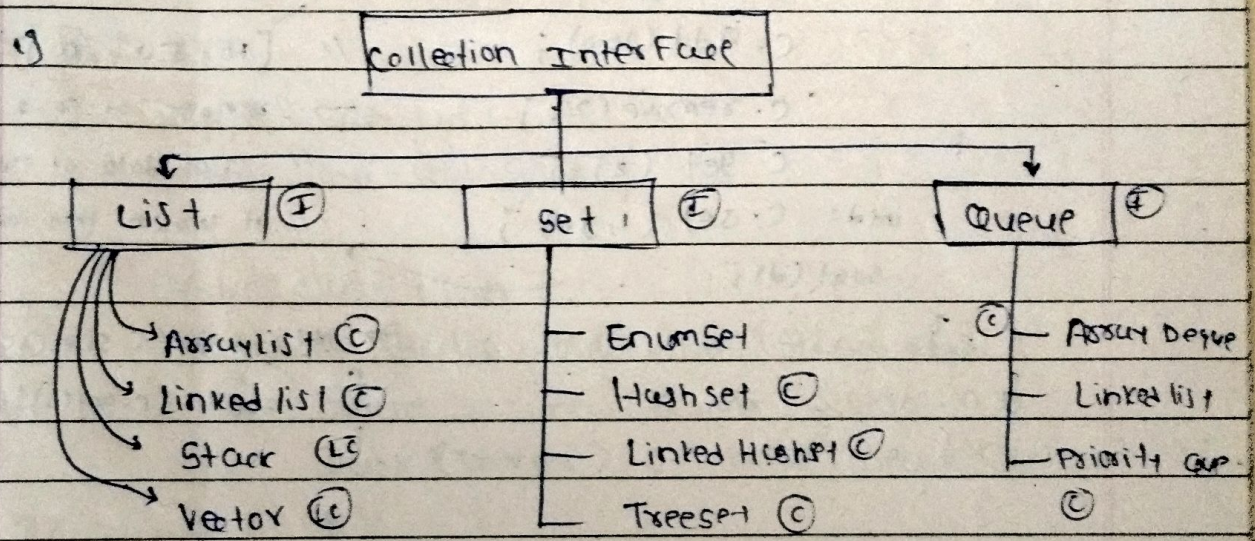
Java collection Framework

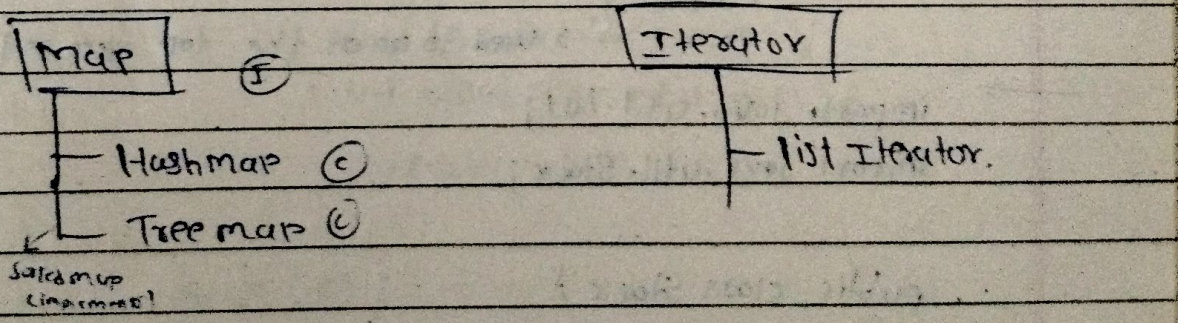
Page No.:
Date: / /

The Java Collections framework provides a set of interfaces & classes to implement various data structures & algorithms.

Three types of collection Framework

1] Collection interface — Java.util.Collection
2] Map interface — Java.util.Map
3] Iterator

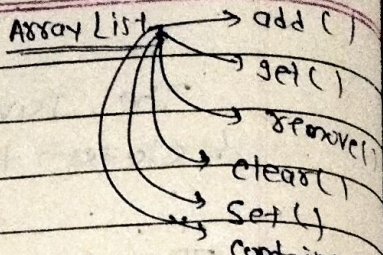




Collection Framework :- It is the set of predefined classes and interfaces which is used to store multiple data.

Page No.:
Date: / / Fri's

1) Array list




```
import java.util.ArrayList;

public class array_list{
    public static void main(String args[])
    {
        List<Integer> c = new ArrayList<>();
                  datatype
        c.add(10);
        c.add(50);
        c.add(100);
        c.remove(2);
        c.get(2);
        int d= c.set(1,90);
        sout(d);
    }
}
```

output

// [10, 50, 100]

→ // remove at FO 2 i.e 100

// Print data at position 2

// at location then insert 90

```
for(int i=0 ; i< c.size(); i++
{
    sout(c.get(i));
}
```

2) Stack → push()
→ pop()
→ peek()
} inbuilt functions

* search()
* isempty()

peek() → used to access the top element of stack

```
import java.util.List;
import java.util.Stack;

public class Stack {
    public static void main(String args[])
    {
        Stack<Integer> c = new Stack<Integer>();
```

```
        c.push(100);
        c.push(200);
        c.push(300);
        c.push(400);                    // [100, 200, 300, 400]
        int d = c.peek();               // 400
        int e = c.search(200);
        Sout(e);
    }
}
```

Input From user

→
```
import java.util.List;
import java.util.Stack;

Public class stack {
    public static void main (String args[])
    {
        Scanner sc = new Scanner (System.in);
        Stack <Integer> c = new Stack <Integer> ();

        For (int i=0; i<5; i++)
        {
            Sout ("Enter your number:");
            int value = sc.nextInt();
            c.Push(value);
        }
    Sout (c);

    if (!c.isEmpty())
    {
        c.pop();
    }
}
```

→ In Collection Framework Array & LL are same.

3] LinkedList

→ ~~import~~

```
import java.util.Linkedlist ;
   public class linklist {
       public static void main(String args[])
       {
           Linked list<Integer> c = new linkedList<>();
           c.add(10);
           c.add(50)
           c.add(100);
           c.add(60)
       →   c.remove(2)
           c.Clear();          → Clear all the data
  boolean d = c.contains(50)   → check whether, that perticular value is present or not in LL
       }
   }
```

---

* Queue using Linked List

Queue → Linked list ; → Priority Queue

* Linked list
  - → OFFER → used to enqueue
  - → Poll → used to dequeue ...
  - → peek

* Priority Queue
  - → OFFER
  - → Pull

* आपण Collection Framework use करत किंवा simple way ने program करत time complexity same राहते. Page No.: Date: / /

→
```
import java.util.Queue
import java.util.LinkedList;

public class queue {
    public static void main(String args[])
    {
        Queue<Integer> c= new LinkedList<>();
        c.offer(10);
        c.offer(20);
        c.offer(30);
        c.offer(60);        // [10,20,30,60]

        Sout(c.peek());     // 10
        c.poll();           // 60 is deleted
        c.poll();           // 30 is deleted
        Sout(c)             // [10,20]
    }
}
```

* Priority Queue

→
```
import java.util.PriorityQueue;
import java.util.Queue;
public class priority_queue {
    public static void main(String args[])
    {                                   → use min heap
        Queue<Integer> c= new PriorityQueue<>();
        c.offer(40);
        c.offer(89);
        Sout(c);
        c.poll();
        Sout(c);
```

Note :- For max heap ⇒ `Queue<Integer> c= new PriorityQueue<>(comparator.reverseOrder());`
`java.util.Comparator;`

Page No.:
Date: / /

Set Interface

- Set interface does not allow duplicate values because it's storing the data in hashes & hash table does not allows duplicate values.
- unordered set of items.

1] HashSet

- add()
- remove()
- Contains()
- Size()
- Clear

(unordered collection)

2] LinkedHashSet

- add()
- remove()
- Contains()
- Size()
- Clear

(ordered collection)

3] TreeSet

- add()
- remove
- Contains
- Size()
- clear()

(Sorted collection)

1] HashSet

```
import java.util.HashSet;
import java.util.Set;
public class Hashset {
    public static void main (String args[]){
    {
        Set<Integer> c = new Hashset<>();
        c.add(10);
        c.add(50);
        c.add(100);
        c.add(60);        // [50, 100, 10, 60]
        c.remove(50);     // [100, 10, 60]
        c.size() → return no items available
    }
}
```

→ unordered collection of data (main diff b/w arraylist & hashset)

2] LinkedHashSet → Display the data in sequence as we insert.

→ Same as hashset (but difference - the LinkHashset is implemented using LinkedList)

→ LinkHashset maintain the property of Hash set as well as linkedlist.

→ Set<Integer> c = new LinkedHashSet<>();

3] TreeSet → Display Data in sorted form.

→ maintaining the property of linked list & Binary tree.

→ Set<Integer> c = new Treeset<>();

Map Interface → A map contains values on the basis of key i.e key & value pair

HashMap
- Put()
- PutIFAbsent()
- isEmpty()
- remove()

TreeMap
- Put()
- PutIFAbsent()
- isEmpty()
- remove()

→ Each key & value pair is known as an entry.

→ A map contains unique keys.

Note → A map doesn't allow duplicate keys, but you can have duplicate values

→ HashMap & LinkedHashMap allow null keys & value but TreeMap doesn't allow any null key or value.

Uniqueness (key is unique)

1) HashMap

→ Map<Integer, String> c = new HashMap<>();
(Integer = key)

c.Put(1, 'A');
c.Put(2, 'B');
c.Put(3, 'C');
~~c.Put(4,~~
c.PutIfAbsent(4, 'D'); → data Absent असेल तर Order करतो, आणि असेल तर तसाच राहू देतो.

For(Integer d : C.keySet())
{
Sout(d); // For accessing the keys
}

~~For(Sout~~ For(String d : C.values())
{
Sout(d); // For only accessing the values.
}

2) TreeMap → Sorted order (used Binary sort concept)

Map<Integer, String> c = new TreeMap<>();

List Iterator → list iterator is used to traverse all the elements in the list

* Forward traversing → hasNext();
  → next();

* Backward traversing → hasPrevious()
  → Previous()

→ eg:

```
List<Integer> c = new ArrayList<>();
c.add(10);
c.add(20);
c.add(100);
c.add(60);

ListIterator<Integer> d = c.listIterator();
while(d.hasNext())
{
    Sout(d.next());
}
while(d.hasPrevious())
{
    Sout(d.Previous());
}
```

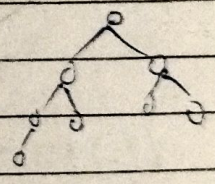
Non-Linear Data structures

Binary Tree

Types of Binary Tree

① Binary Tree ⟶ only two child are present.

② Complete BT ⟶ two child complete पाहिजे आणि last वाला complete असला पाहिजे, पण leftmost पाहिजे

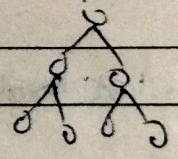


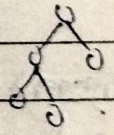
⟶ This is complete BT

③ Full BT / 2-Tree / Proper BT / Strictly BT

⟶ कोणत्या पण node ला 0 किंवा 2 child असतील

eg





⟹ Full BT

④ Perfect BT ⟹ जेव्हा levels ला completely Filled असतो

⟹ perfect BT

Formula

#

Page No.:
Date: / /

Formula

1] Max no of nodes possible at level i = 2^i

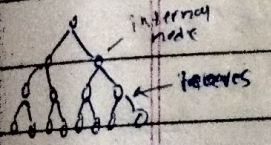
2] Max. no of nodes possible in a B.T of height h = $2^{h+1} - 1$

3] Min no of nodes possible in a B.T of height h = $(h+1)$ (h = height)

4] Max height of B.T having 'n' nodes = $\lceil \log_2(n+1) - 1 \rceil$

5] Min ~~no of nodes~~ height possible in B.T having 'n' nodes = $(n-1)$

* In full binary tree of height h



$l = 2^h$ leaves and $m = 2^h - 1$ internal nodes

∴ binary tree of height 3
8 leaves 4 + internal nodes.

Motivation → IF you want to be Programmer —then→ you do not need to learn recursive approach

But, IF you want to be good Programmer —then→ You must learn recursive app.

★ **Recursion** → efficient approach in order to solve the Problem

Recursion is the technique of making a Function call itself.

Note ★★

1. When a Function call's itself
2. Base condition is must (कही कोणत्या Problem मध्ये base condⁿ exist करत नसेल तर, आपण त्यांना recursive method ने use नाही करू शकत.)

(terminating condition)

eg:-

```
A(n)
{ sout ("Hi")
  A(n-1);
}
```

Function calls → recursion

⟹ Let n = 4

Hi n=(4-1) = 3
Hi n= (3-1)=2
Hi ~~n=(2-1)=1~~
Hi n= (1-1) = 0
H n= (0-1)= -1

} There is no termination condⁿ (Base condⁿ) on Program goes to infinite loop

↗ This is not used in recursion

For eg:-

```
A(n)
{ if(n==1)
  { return 1;
  }                 } Base condⁿ (terminating condⁿ)
  else
  {
    sout ("Hi");
    A(n-1);
  }
```

3. use to solve bigger problem after breaking it down in smaller Problem

eg:- Factorial of 50 ⟹ we break the Problem First we Find Factorial 1 then 2, 3 - 49 & then we get Factorial of 50

4. iterative problems ⇌ Recursive problem

Page No.:
Date: / /

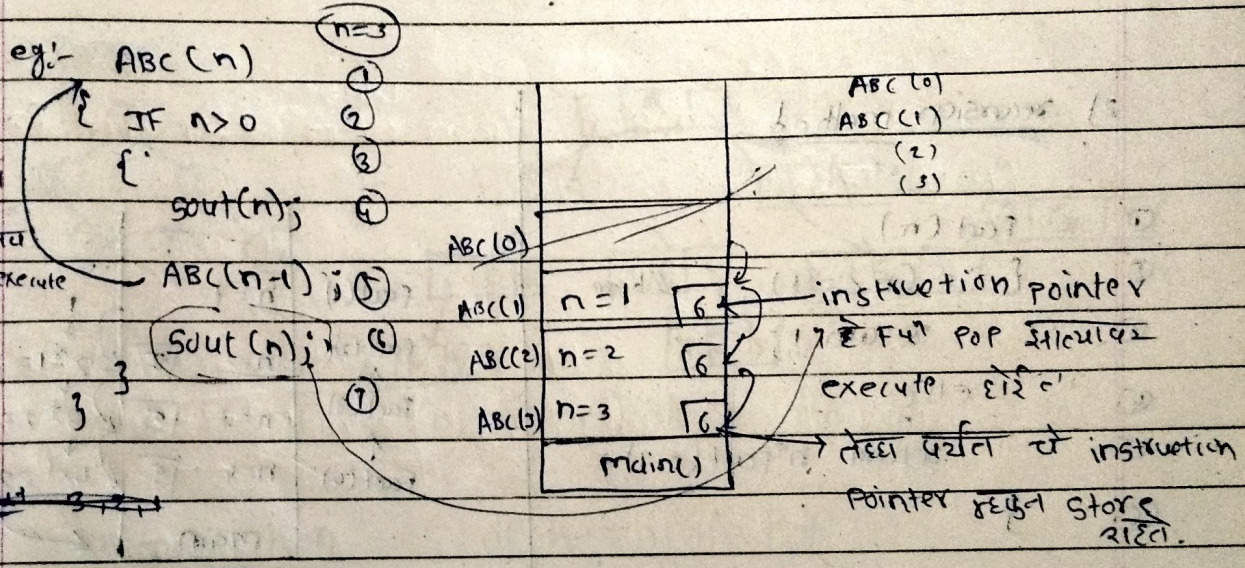
Recursion

Stack

Tree

1) Stack

instruction pointer :- The program counter, called instruction pointer, is a processor register that indicates the current address of program being executed.

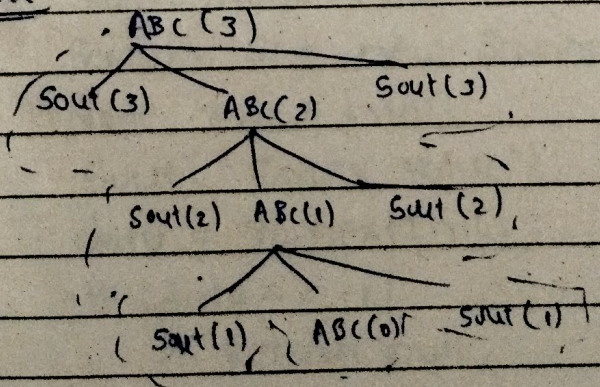


~~output 3,2,1~~

output :- 3, 2, 1, 1, 2, 3

pop झालेपासून execute होईल

2] Tree



⇒ 3, 2, 1, 1, 2, 3

Page No.:
Date: / /

Program to Find the Factorial

① Iterative method

```
F=1 ; n=4
for(i=1 ; i<=n ; i++
{
   F=F*i;
}
```

⇒

1*1 = 1

1*2 = 2

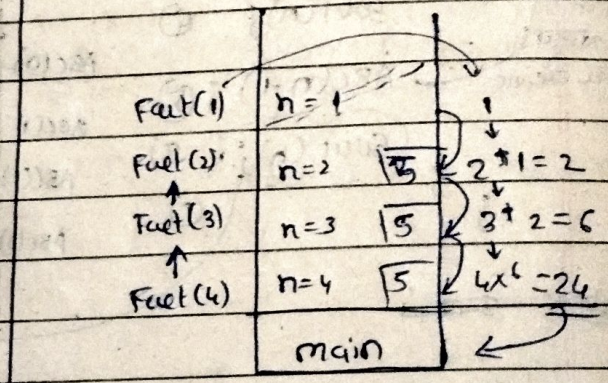
2*3 = 6

6*4 = 24;

So Fact(4) = 24

2) Recursion method

```
① Fact(n)
② { if (n<=1)        // base
③     return 1;
④   else
⑤     return n*Fact(n-1);
⑥ }
```



So Fact(4) = 24

Binary tree Implementation

- Array.
  - left child

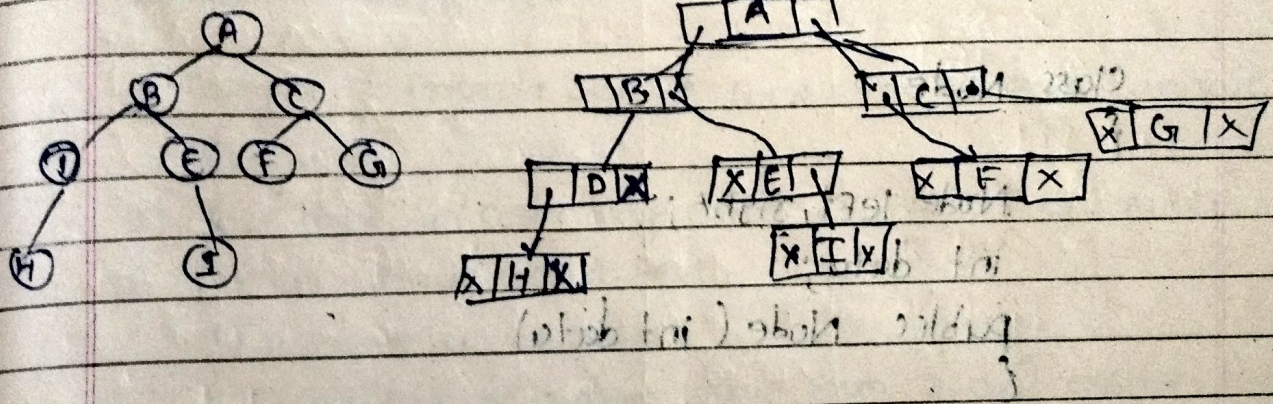
→ index (Array)

Left child = 2i+1
Right child = 2i+2

} Array मध्ये data या formula ने Store केला जातो i.e. tree मध्ये left आणि right

- using Linked list

* Concept



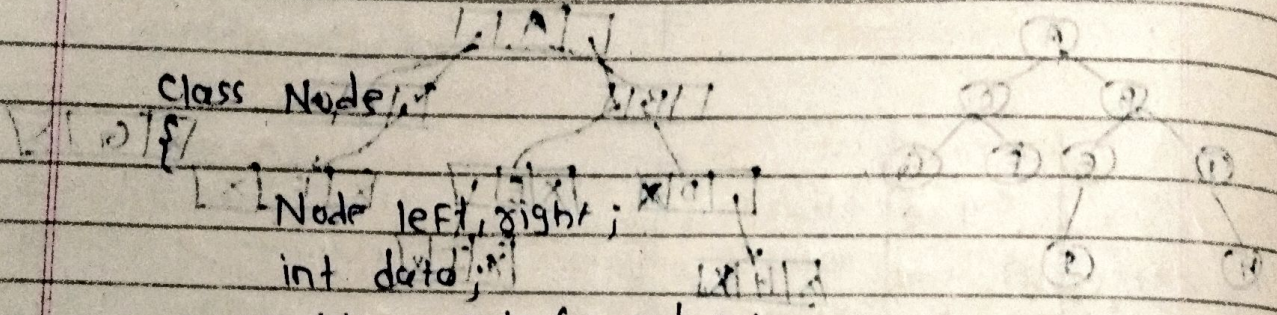
Implementation

```
public class tree_creation
{
    Static Node Create() {
    Scanner sc = new Scanner(System.in);
    Node root = null;
    Sout("Enter value");
    data = sc.nextInt();
    IF (data == -1)
        return null;
    root = new Node(data);
```

```
System.out.print("enter left child of" + root.data);
root.left = create();
System.out.print("enter right child of" + root.data);
root.right = create();
return root;
}

public static void main(String args[])
{
    Node root = create();
}
}

Class Node
{
    Node left, right;
    int data;
    public Node(int data)
    {
        this.data = data;
        left = null }
        right = null;
    }
}
```



Shelfscraps

O(nlogn) ← **Huffman Coding** → david Huffman

Page No.:
Date: / /

* Application of Binary Tree → Huffman coding
  ↳ used to compress the data
  ↓ Frequency occuring characters

Huffman Coding
⇓
Compress → Fixed length → simple
→ Variable length

→ Huffman coding is a technique of compressing data to reduce its size without losing any of the details.

Huffman Coding Algorithm

1) Create a Priority queue consisting of each unique character
2) Sort them in asending order of their frequencies.
3) For all unique characters
   i) Create a NewNode
   ii) extract minimum value from Q and assig to left child
   iii) extract minimum value from Q & assign to Right child of NewNode
   iv) Calculate the sum of these two minimum value and assign it to the value of NewNode
   v) Insert this NewNow into the tree
4) return rootNode.

eg:- ①

1	6	5	3
B	C	A	D

→ ① Calculate Frequency → From string

⇓ sort, in ~~Accen~~ increasing order (In Priority queue)

1	3	5	6
B	D	A	C

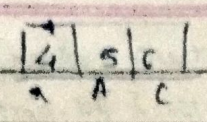
⇓
Add ~~lower child~~ & & make smaller to left

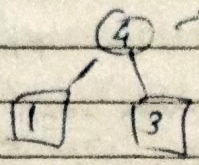
Huffman coding complexity

① Time complexity = $O(n \log n)$

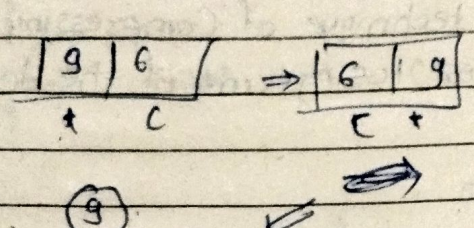
Page No.:
Date: / /

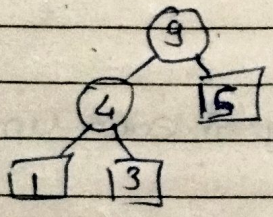
* Extracting minimum frequency from priority queue = $2*(n-1)$ times
* Complexity = $O(\log n)$

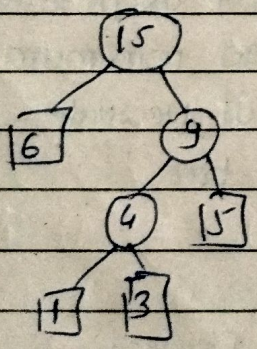




→ make lower as left node & higher as Right Node of tree







Character	Freq	Code	Size
A	5	11	5×2 = 10
B	1	100	1×3 = 3
C	6	0	6×1 = 6
D	3	101	3×3 = 9

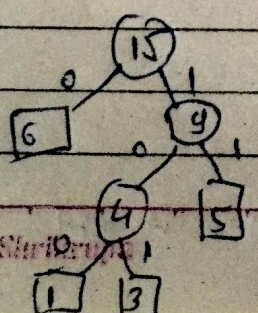
4×8 = 32 bits | 15 bits | 28 bits

without encoding total size = 120 bits

After encoding = 32+15+28 = 75 bits

Compress

Decode It



left child = 0
Right child = 1

Applications

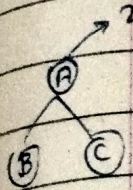
① used in conventional compression formats like GZIP, BZIP2, PKZIP etc

② For text & Fax transmission

Graph

Page No.:
Date: / /

Graph :- A graph is a nonlinear structure. like linear data structures (lists), it can be implemented with either an Indexed or a linked building data structure.



* Non-linear Data Structures

* A graph G(V, E) consist of

1] a set of objects $V = \{v_1, v_2, v_3 \dots v_n\}$ called vertices.

2] a set of $E = \{e_1, e_2, e_3, \dots e_n\}$ called Edges.

Applications

1) Facebook, twitter accounts → graphs suggest mutual friends
2) Telicommunication Network.
3) Google maps → used for building transportation system
4) Computer science → used to represent Flow of Computation.
5) world wide web →
6) operating system → Process & resources are consider as ~~edges~~ vertices of graph. & edges can drawn resources to allocated process.
7) mapping system → In GPS We also use graphs
8] microsoft Excel → use DAG means Directed Acyclic graphs
9] Dijkstra Algo → use graph, to find shortest path betw" Nodes
10] Social media sites → we use graph to track data of users.

Graph Representation in memory

1] Adjacency Matrix [Array]

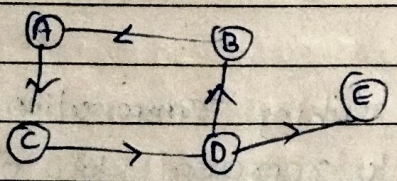
2] Adjacency List { Link list }

3] Multi List

1] Adjacency Matrix → using Array

Aij = 1, if there is an edge From Vi to Vj

Aij = 0, if there is no edge

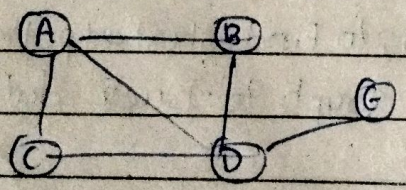


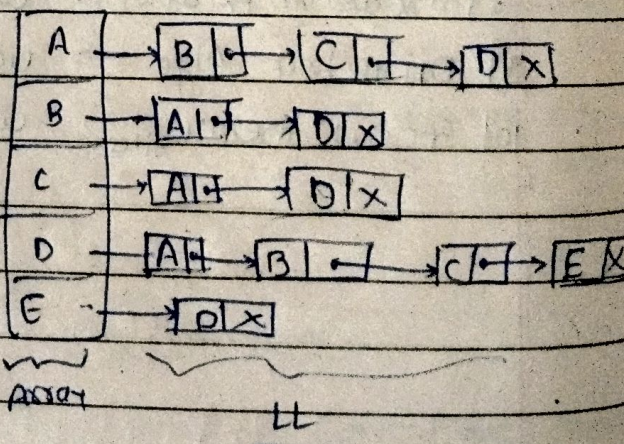
	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	0
C	0	0	0	1	0
D	0	1	0	0	1
E	0	0	0	0	0

Weighted graph असेल तर त्या ठिकाणी edge चा weight लिहायचा.

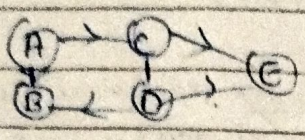
2] Adjacency List (using Linklist) → or combination of (Array + LL)

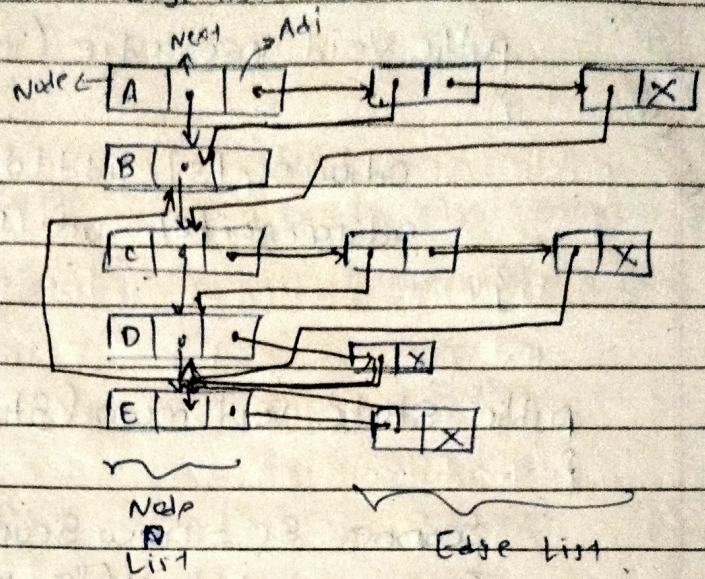
eg:-





3] MultiList (pure LL) → Node list
→ Edge list





* Graph Implementation using Adjacency List

* Program

```
import java.util.LinkedList;
import java.util.Scanner;

public class graph_implement {

    privute LinkedList <Integer> adjacency[];   → used Collection Framework
    public graph_implement(int v)
    {
        adjacency = new LinkedList[v];
        For(int i=0; i<v; i++)
        {
            adjacency[i] = new LinkedList<Integer>();
        }
    }
```

Only for implementation of graph
Traversing is done using BFS & DFS

```
public void insertedge (int s, int d)
{
    adjacency [s] . add (d);
    adjacency [d] . add (s);
}
```

sort (adjacency [s]);
sort (adjacency [d]);

```
public static void main (String args[])
{
    Scanner sc = new Scanner (System.in);
    System.out.print ("Enter Number of Vertices & Edges");
    int v = sc.nextInt();
    int e = sc.nextInt();

    graph_implement g = new graph_implement (v);
    System.out.print ("Enter edges");
    for (int i=0; i<e; i++)
    {
        int s = sc.nextInt();    // Source      } edges
        int d = sc.nextInt();    // Destination }
        g. insertedge (s,d);
    }
}
```

Page No.:
Date: / /

Graph Traversal

BFS (Breath First Search)

(Queue) → [FIFO]

↓

Slower

↓

Time complexity = O(V+E)

↓

require more memory space O(V)

DFS (Depth First Search)

(Stack) : [LIFO]

↓

Faster

↓

O(V+E)

V = Vertices
E = Edges

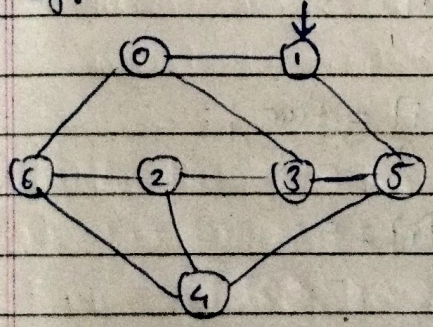
↓

less memory space require

+ **BFS** → uses Queue

BFS is a vertex-based technique for finding the shortest path in the graph.

eg:-



& 1 queue

we have to create 3 Arrays.

For visited arrays V[7] =

1	0	5	6	3	4	2
true	T	T	T	T	T	T

boolean type

P[7] =

1	0	5	6	3	4	2

Integer type

Queue =

1	0	5	6	3	4	2

After pushing from Que value is printed one by one = 1, 0, 5, 6, 3, 4, 2

→ one vertex is selected at a time when it is visited & marked then its adjacent are visited & stored in the queue.

Continue in previous program

Page No.:
Date: / /

Implementation BFS

insert from previous program

```
public void bFS(int source)
{
    boolean visited_node[] = new boolean[adjacency.length];
    int parent_nodes[] = new int[adjacency.length];
    Queue<Integer> q = new LinkedList<>();
    q.add(source);
    visited_nodes[source] = true;
    parent_nodes[source] = -1;
    while(!q.isEmpty())
    {
        int p=q.poll();
        System.out.print(p);
        For(int i; adjacency[p])
        {
            IF(visited_node[i]!=true)
            {
                visited_nodes[i] = true;
                q.add[i];
                parent_nodes[i] = p;
            }
        }
    }
}
```

Imp step

use in previous program

```
public static void main(String args[])
{ Sout("Enter Source For bfs traversal");
    int Sowe = Sc.nextInt();
    q.bFS(Source);
}
```

Page No.:
Date: / /

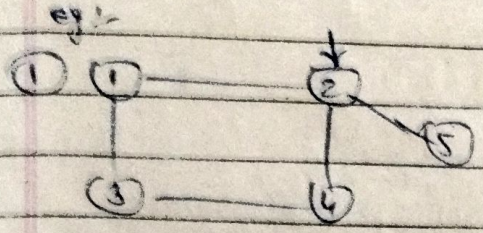
DFS (Depth First Search) → uses Stack → popped last element (LIFO)

The DFS is an edge - based technique

two stages

1. First visited vertices are pushed into the stack &
2. Second if there are no vertices then visited vertices are popped.

eg:-



Adjacent of (2)

	2	1	4	5	3
V[5]	true (1)	T	T	T	T

P[5]	2	5	4	3	1

parent node

Stack1: 2 | 1 | 4 | 5

see 2 adjacent node

Printing :→ 2, 5, 4, 3, 1

Stack2: 1 | 4 | ~~5~~

see 5 adjacent node

1 | ~~4~~

adjacent of 4 are 2, 3 → 2 is visited by 3 is no so add 3

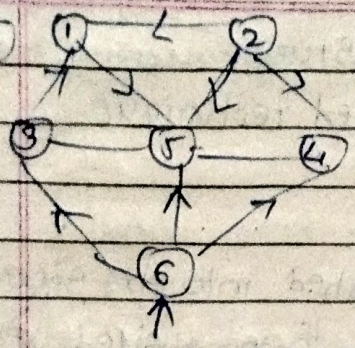
1 | ~~3~~

see adjacent of 3 ⟹ all are visited

~~1~~

Page No.:
Date: / /

eg:-



	6	3	5	4		
V[6] =	T	T	T	T	T	T

P[6] =	6	4	5	3		

Stack: | ~~6~~ | 3 | 5 | ~~4~~ |

S_2 | 3 | ~~5~~ |

Printing = 6, 4, 5, 3, 1

~~3~~

~~1~~

2 is not visited

↳ because their is no path to find 2

So, 2 cannot be traversed Hear.

Implementation DFS

```
public void DFS(int source)
{
    boolean visited_nodes[] = new boolean[adjacency.length];
    int parent_nodes[] = new int[adjacency.length];
    Stack<Integer> q = new Stack<>();
    q.add(source);
    visited_nodes[source] = true;
    parent_nodes[source] = -1;
    while(!q.isEmpty())
    {
        int p=q.pop();
        System.out.print(p);
```

```
        for (int i = adjacency[p])
        {
            if (visited_nodes[i] != true)
            {
                visited_node[i] = true;
                q.add(i);
                parent_nodes[i] = p;
            }
        }
    }
}

public static void main (String [] args)
{
    Sout (" Enter source for DFS traversal")
    int sources = sc.nextInt();
    q.bFS(source);
    q.dFS(source);
}
```

(add problem) →

(same) ↩

Page No.:
Date: / /

Spanning Tree (n-1) edges

A Spanning tree is a subset of the graph (subgraph) that does not have cycle, and it also cannot be disconnected. (which has all vertices covered with minimum possible no of edges).

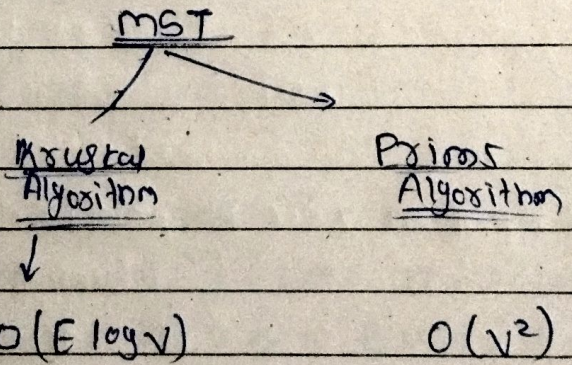
Applications

A Spanning tree used to find minimum path to connect all nodes of graph

1. Computer network routing
2. Civil network planning
3. Cluster Analysis

Minimum Spanning Tree (MST)

The MST is the one whose cumulative edge weights have the smallest value.



Shrikrupa

Page No.:
Date: / /

1. Relaxation → used in Dijkstra algorithm

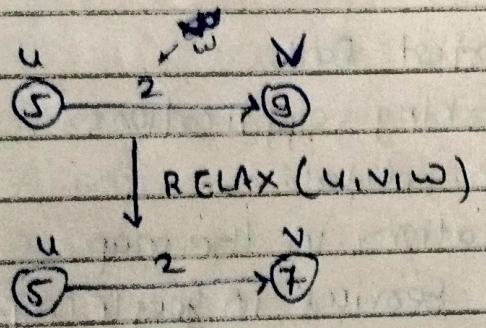
The single-source shortest path are based on a technique known as relaxation.

→ A method that repetedly decreases an upperbound on actual shortest path weight of each vertex until the upper bound equivalent the shortest-path weight.

Code Algo

```
RELAX (u, v, w)
1) IF d[v] > d[u] + w(u,v)
2). then d[v] ← d[u] + w(u,v)
3). π[v] ← u
```

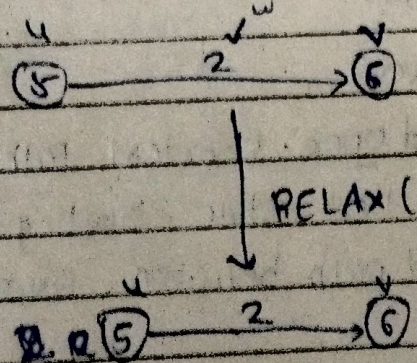
eg:- a]



u w(u,v) 5+2 = 7 & 7 is less than 9

so, we update 9 to 7 in (v)

b]



Here, 5+2 = 7

& 7 is greater than 6

so, no changes occur

Dijkstra's Algorithm → uses greedy approach

It is an algorithm for finding the shortest path between nodes in a weighted graph.

Complexity

- Time Complexity = $O(E \log V)$

Space Complexity = $O(V)$

where

E = no of edges

V = no of vertices

Applications

1) To find the Shortest Paths
2) In social networking applications
3) In a telephone network
4) To find the location in the map.
5) Digital mapping services in google maps
6) Flighting Agendy
7) Robotic Path → drones
8) Designate File Server
9) IP routing - to Find open shortest Path First (OSPF)
   ↳ "OSPF" is a link state routing protol → used to find the best path between Source & Dest?

Page No.:
Date: / /

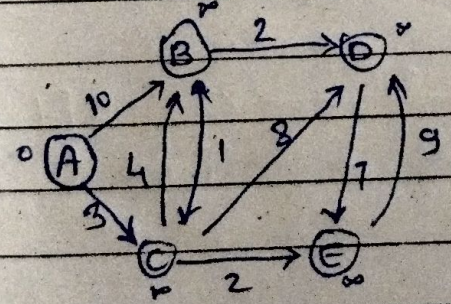
* Algorithm (Dijkstra's)

(Precond^n :- G = (V, ω) is a weighted graph with initial vertex V_0)

(~~Post cond^n~~)

1) Initialize the distance Field to 0 for V_0 & to ∞ for each of the other vertices.
2) Enqueue all the vertices into a priority queue Q with highest priority being the lowest distance Field value.
3) Repeat steps 4-9 until Q is empty
4) Deque the highest priority vertex into x
5) Do steps 7-10 for each vertex y that is adjacent to x & in the priority queue.
6) Let s be the sum of x's distances Field plus the weight of the edge from x to y.
7) If s is less than y's distance Field, do steps 8-9. otherwise go back to step 3
8) Assign s to y's distance Field
9) Assign x to y's back reference Field.

eg:-



→ Initialize - Single - source
→ relax( )
→ Extract min( )

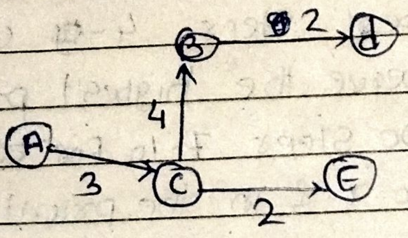
	A	B	C	D	E
Distn	0	∞	∞	∞	∞
	[0]	10	[3]	∞	∞
	[0]	(7)	[3]	11	[5]
	[0]	[7]	[3]	11	[5]
	[0]	[7]	[3]	[9]	[5]

updated value using relaxation (10 to 7)

जो small उसलाई लाई queue बाट Dequeue गरिन्छ

So, the shortest Distn for

A = 0
B = 7
C = 3
d = 9
E = 5



Shrilonga

Page No.:
Date: / /

* Leetcode #200 / Number of Islands

* Given an m x n 2D binary grid, which represent a map of 1's (land) & 0's (water), return the number of islands. An island is surrounded by water & is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Input -

1's	1	0	0	
1	0	0	0	land → 1
0	0	1	0	water → 0
0	0	0	1	

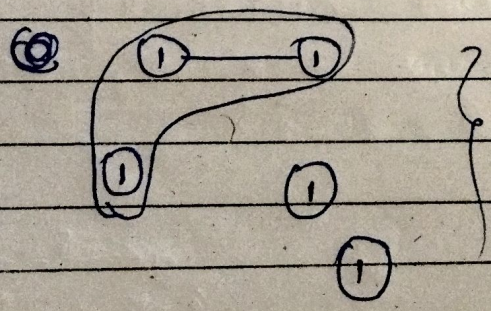
grid[][]

water so all 0 are surrounded

Output = 3 → upper 3(1's) form 1 land &

other lower two lands ⇒ Total = 3 lands

→



→ Algorithm to find the no of Islands are given by DFS.

Page No.:
Date: / /

Floyd Warshall Algorithm

→ All Source Shortest Path Algorithm

The problem is to find the shortest distance between every pair of vertices in a given edge-weighted directed Graph.

→ follows dynamic programming approach.

DP : Recursive Soln → memorization (DP) → Top-down approach (DP)

Page No.:
Date: /

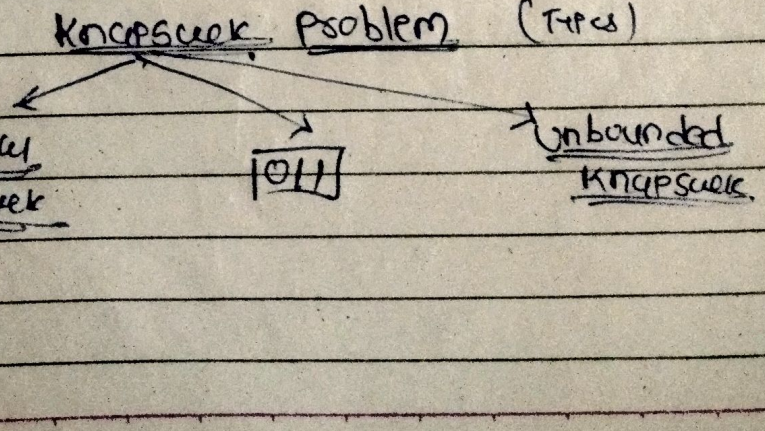
Dynamic programming ⟹ Recursion + Storage

Dynamic programming is a technique ~~that~~ for solving a complex problem by first breaking into a collection of simpler subproblems, solving each problem just once, & then storing their solutions to avoid repetitive computations.

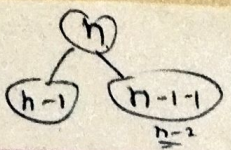
① It simply breaks the problems into sub-problems & saves the result for future purpose.
② Find the optimal solutions of these problems.
③ Stores the result of subproblem → memorization
④ It Reuses them so that same sub-problem is calculated more than once.

0/1 Knapsack Problem

1] Subset sum
2) Equal sum partition
3) Count of subset sum
4) minimum subset sum difference
5] Target sum
6] no of subsets of given difference



DP :-



Page No.:
Date: / /

Fractional Knapsack	0/1 Knapsack
① using greedy Approach	① using Dynamic programming approach
② This problem either takes ~~an item or not~~. !	② This problem either take an item or not. It does not take part of it.
② We can also take a Fraction of item	
③ It has an optimal Structure	③ It also has optimal structure.
④ eg:- we have 10 amount of space. & The Amount A = 5, B = 4 & C = 3	④
First we put A & then B ⇓ So 5+4 = 9 (A+B) So only 1 space is remain & C = 3. So what will D ⇓	→ 0/1 knap But the 0/1 knapsack dose not devide C. it add whole C or remain empty space (does not add C).
⑤ In this situation Fractional Knapsack divide C & Fill the space of 1 to 10	

0/1 Knapsack (DP) →

* How to Identify

weight array → wt[ ] = 2 3 4 5
value array → v[ ] = 1 4 5 7
} O/P → maxratio

Capacity → W = 7 kg

↓↓

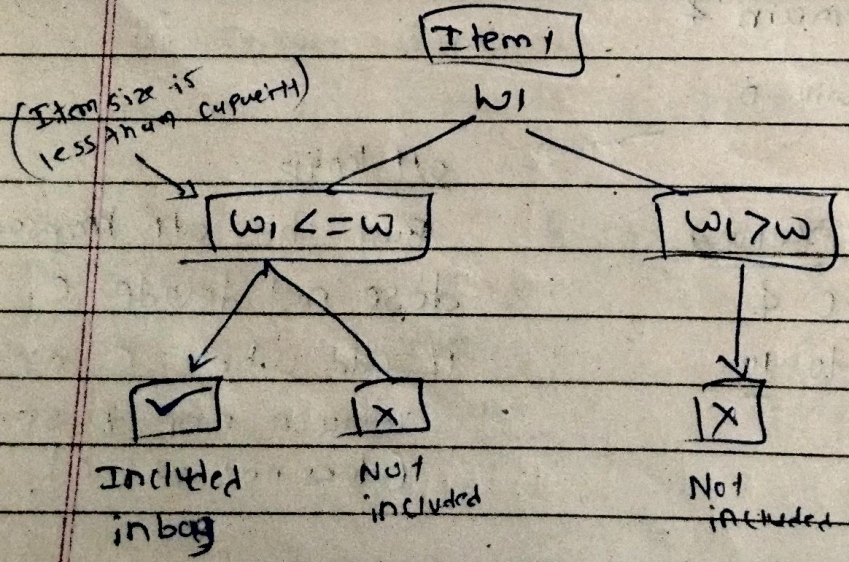
For DP

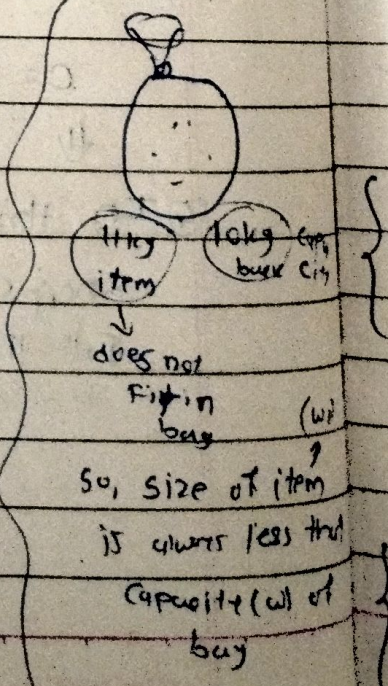
↓ Choice
→ Optimal
↳ max, min, largest, greatest etc.

0/1 knapsack Recursive code
↳ we have to return max^m profit

Recursive
↓
Choice

Choice Diagram





Page No.:
Date: / /

Program Approach → कैसे

return max profit ← `int KnapSack(int wt[], int val[], int W, int n)` ← Fun^n

{ I/P

Base Condn

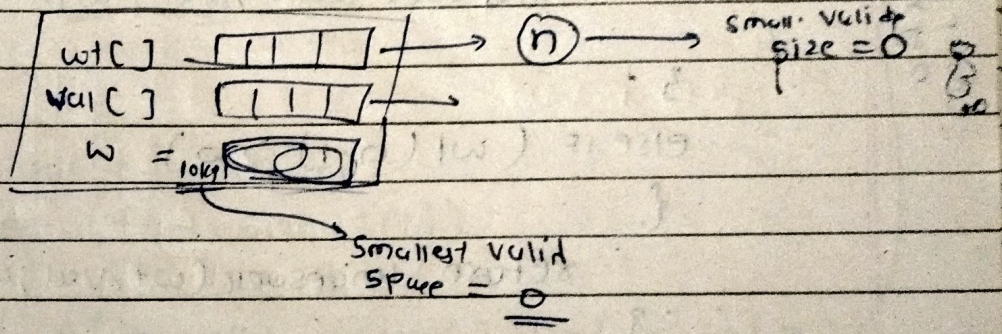
Choice diagram

}

* **Base Cond^n** → important in recursive approach

→ Think of the smallest valid I/P.

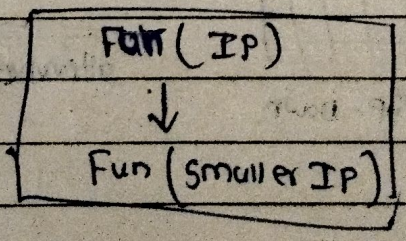
eg:-



So; `IF (n==0 || W==0) { return 0 }` } Base cond^n

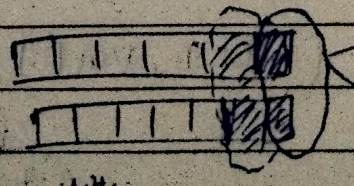
* **Recursive Fun^n**

{ Input को small करते जाओ } I/P - OP method ⇒



Fib(n)
↓
Fib(n-1)

eg:-



{ Answer को आसान कैसे करते जाओ }

`IF (wt(n-1) <= W`

`val[n-1] + knapSack(wt, val, W - wt`

Code :

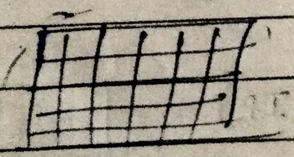
```
int knapsack (int wt[], int val[], int W, int n)
{
    if (n==0 || W==0)      } // Base condn
    { return 0 };

    if (Wt[n-1] <= W)
    {
        return max( val[n-1] + knapsack (wt, val,
                                   W-wt(n-1), n-1),
            knapsack (wt, val, W, n-1) );
    }
    else if (wt(n-1) > W)
    {
        return knapsack (wt, val, W, n-1);
    }
```

Choice Diagram

Memorization

Recursive code
↓
memorize



→ alternative → Memorization

knapsack (wt[], val[], W, n)

W → wt - wt(n-1)

n → n-1

इथे change होणार (फक्त याचीच matrix बनेल)

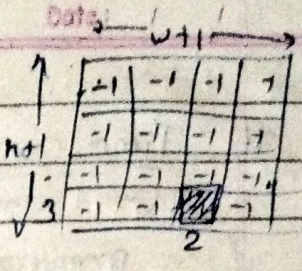
matrix name → int t [n+1] [W+1]

memset() → function in ~~2D~~ 2D array

memset(arr, 0, sizeof arr);

```
int t[n+1][wt+1]
memset(t, -1, sizeof(t))
```

↓ recursive Form



t[3][2] = recurForm(2,3)

→ कधी ह्या आधि value असाती ना तो return केली असती आणी (-1) असेल तर recursive fn

* Code changes (DP) (memorization)

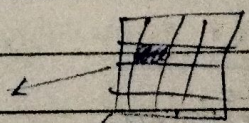
globally declare →

```
int t[
    int static t[102][1002];
    memset(t,-1, sizeof(t))

int knapsack(int wt[], int val[], int w, int n)
{
  if(n==0 || w==0)
  { return 0 }

→ if(t[n][w]!=-1)
  {
     return t[n][w]
  }

  if(wt[n-1]<=w)
  {
   return t[n][w]= max( ____sum as before____ )
  }
  else if( wt[n-1]>w)
  {
   return t[n][w] = knapsack(wt,val,w,n-1);
  }
```



Page No.:
Date: / /

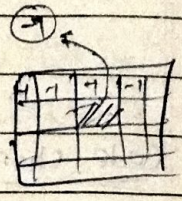
Time Complexity (knapsack)

(Two variation same)

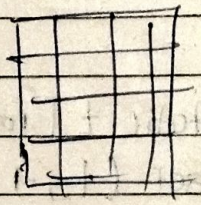
Memorization or Top Down = $O(n^2)$

Top-Down Approach (0/1 knapsack)

1] Recursive approach
2] Memorization ⟹ recursive +



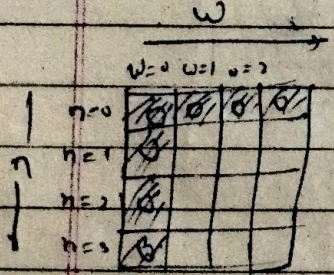
3] Top-Down ⟹



→ (without using recursive)

→ For changing recursive Fun^n, (First we remove base cond^n)

IF (n==0) || (w==0) return 0 ⟹ recursive Base cond^n converted into Top-Down Initilization



t[n+1][w+1]

→ (Initilization)

```
int
For (i=0; i<(n+1); i++)
{
  For(int j=0; j<(w+1); j++)
  {
    IF (i==0 || j==0)
      t[i][j]=0;
  }
}
```

Skillcraft

* Also, From Choice diagram code of memorization, we have to remove recursion.

t[n][w]

Val[n-1] + Knapsack(wt, val, w - wt[n-1], n-1)

Val[n-1] + t[n-1][w - wt[n-1]]

कारण फक्त W, आणि n change होतात

So, IF (wt[n-1] <= w)

{

t[n][w] = max(Val[n-1] + t[n-1][w - wt[n-1]] , t[n-1][w])

}

else

{

t[n][w] = t[n-1][w]

}

Choice Diagram code

Add this code with initialization

we get code of Top-Down approach

~~Dynamic Programming~~

of

0/1 Knapsack

Page No.:
Date: / /

* Program :- 0/1 knapsack (DP) → Top-Down

→

```
import java.util.*

Class Top-Down {

    Static int KnapSack ( int Wt, int val, int W, int n)
    {
        int[] t = new int[W+1];

        For (int i=1 ; i<(n+1); i++ )
        {
          For (int j=1 ; j<(w+1) ; j++)
          {
              iF (wt[i-1] <=j )

        t[i][j] = max ( val[i-1]+ t[i-1][j-wt[i-1]] ,
                                  t[i-1][j] )

              else
                {   t[i][j] = t[i-1][j]
                }
        return t[i][i]
    }

    Public Static void main ( String[] args)
    {   int Profit[] = {60, 100, 120};
        int weight[] = {10, 20, 30};
        int W = 50;
        int n =  Profit.length;
        Sout ( knapsack(W-1, val,
        Sout(knapsack (weight, Profit, W, n) ;
```

Pratik Bagade (2019...) To be continued...

Page No.:

Date:

* Subset Sum Problem