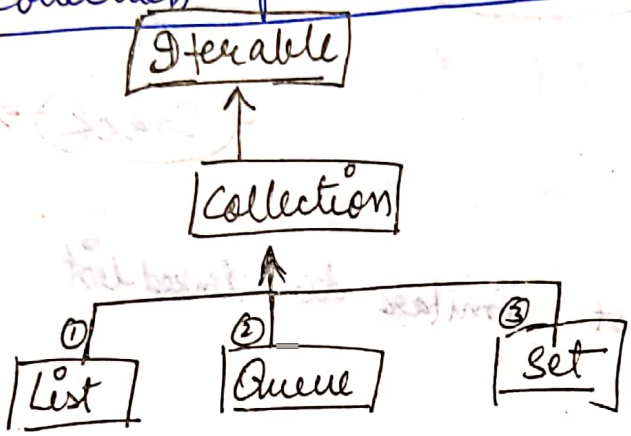
Java DSA (Part - 2)

Lecture - 23

Java Collection Frame Work

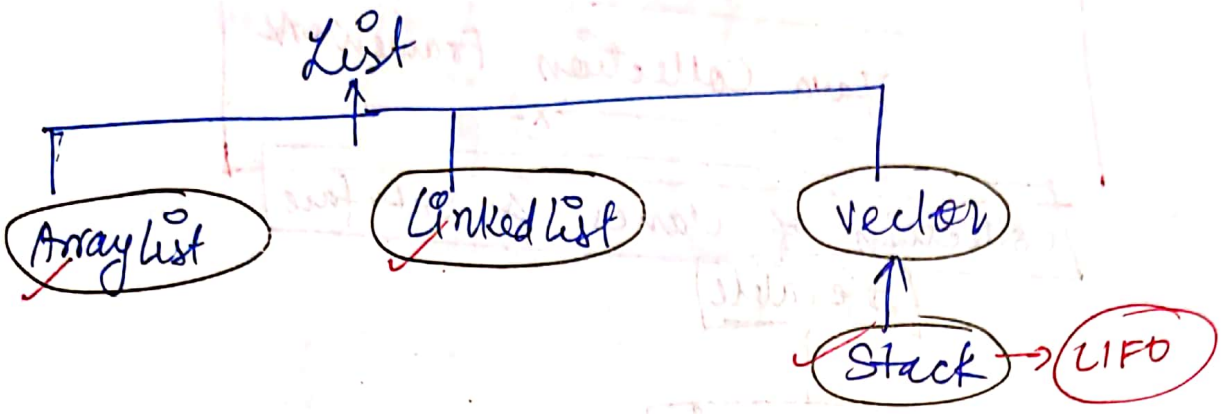
Collection of classes & Interface



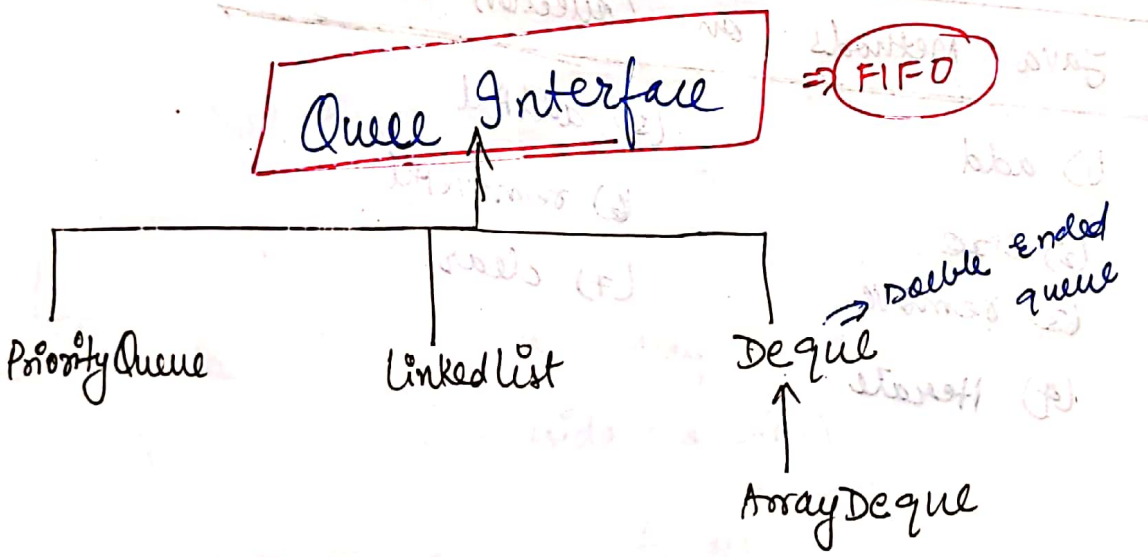
* Java Methods on Collection Frame Work *

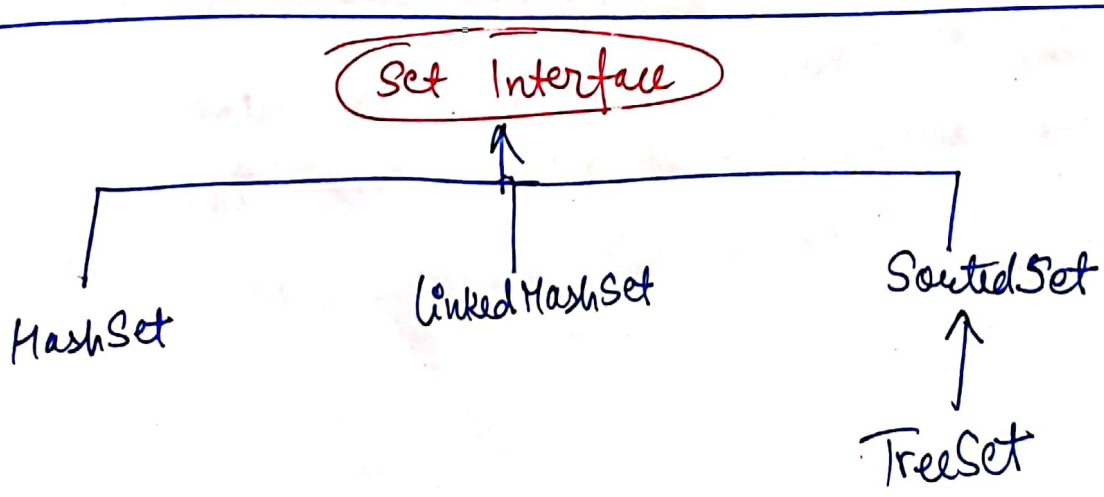
(1) add
(2) size
(3) remove
(4) iterate
(5) addAll
(6) removeAll
(7) clear

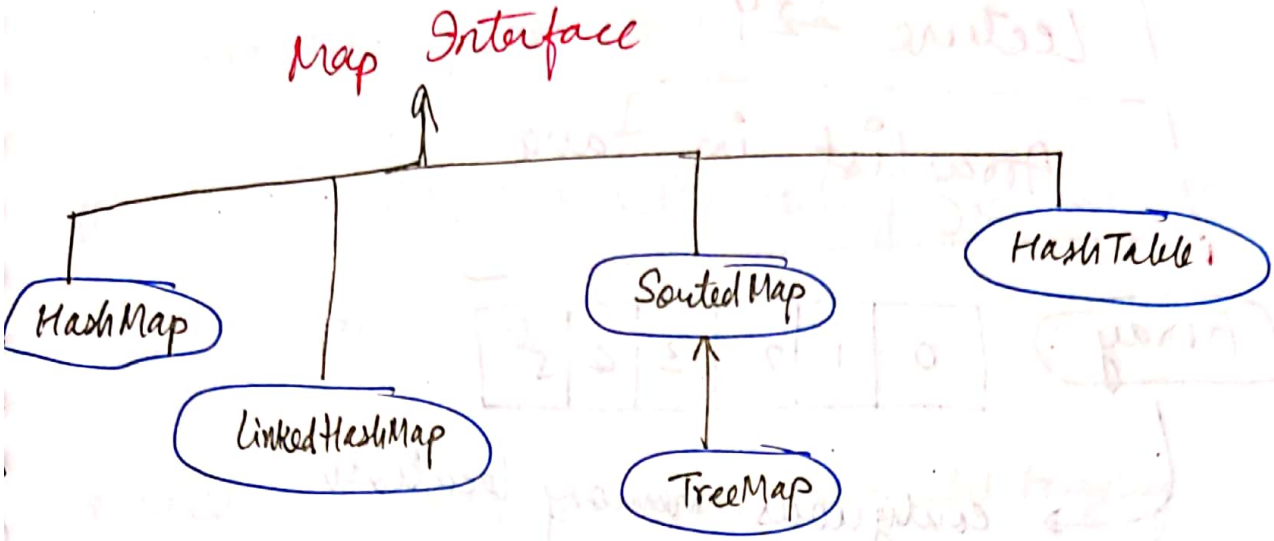
List Interface



* Vector ≈ almost similar to linked list (adv. java)

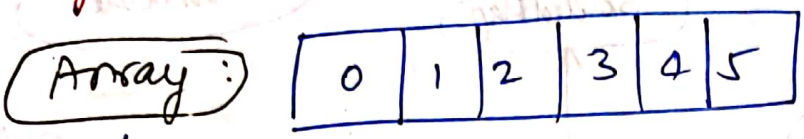






Lecture -24

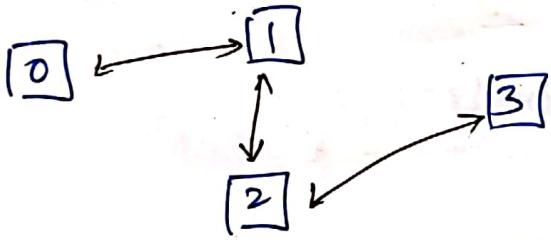
Array list in Java



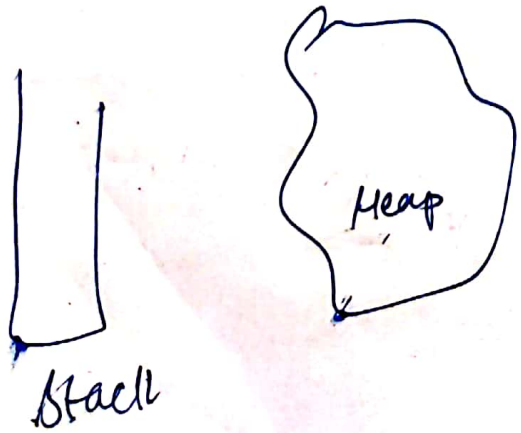
- contiguous memory location
- fixed size
- stores primitive (int, float)
- stores objects

ArrayList

- size - variable
- non-contiguous memory locations



- size can be unlimited until over system memory gets fulled
- can only store objects
- created in heap memory



Operations on Array list

① Add ② Get ③ Modify ④ Remove

⑤ Iterate/operation

* Need a package ⇒ import java.util.ArrayList;

* Array list can't store primitive data-type (int, float ...)

↓

so to store integer number in our Array list we use "Integer" - class

Note:- for classes ⇒ (Integer, String, Float, Boolean) ⇒ class

---

Code

```
import java.util.ArrayList;
import java.util.Collections;

class Arraylistts
{
    public static void main(String args[])
    {
```

```
ArrayList<Integer> list = new ArrayList<Integer>();
ArrayList<String> list2 = new ArrayList<String>();
ArrayList<Boolean> list3 = new ArrayList<Boolean>();

//add elements
list.add(1);
list.add(3);
list.add(5);
sout(list);

// to get an element
int element = list.get(0); // 0th index
sout(element);

// add element in between
list.add(1,2);
        (1 index par 2 add)
sout(list);

//set element
list.set(0,0); -> (0 index par 0)
sout(list);

// delete element
list.remove(0); //0th index)
```

```
//size of list
int size = list.size();
sout( size);

//loop on list
for(int i=0; i< list.size(); i++)
    {
      sout(list.get(i)+" ");
    }

//sorting
Collections.sort(list);
Sout(list);
}//main
}//class
```

---

* Arraylist gets stored in the memory dynamically.

→ To insert any element in between, we have to shift all other elements a step backword

middle insert ⇒ $T_n = O(n)$

Search ⇒ $T_n = O(1)$

Lecture -25

Linked List in Java

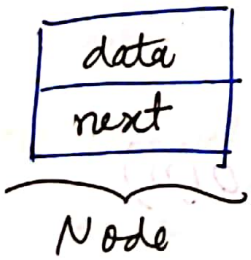
	Arraylist	Linked List
Insert :	$O(n)$	$O(1)$
Search :	$O(1)$	$O(n)$

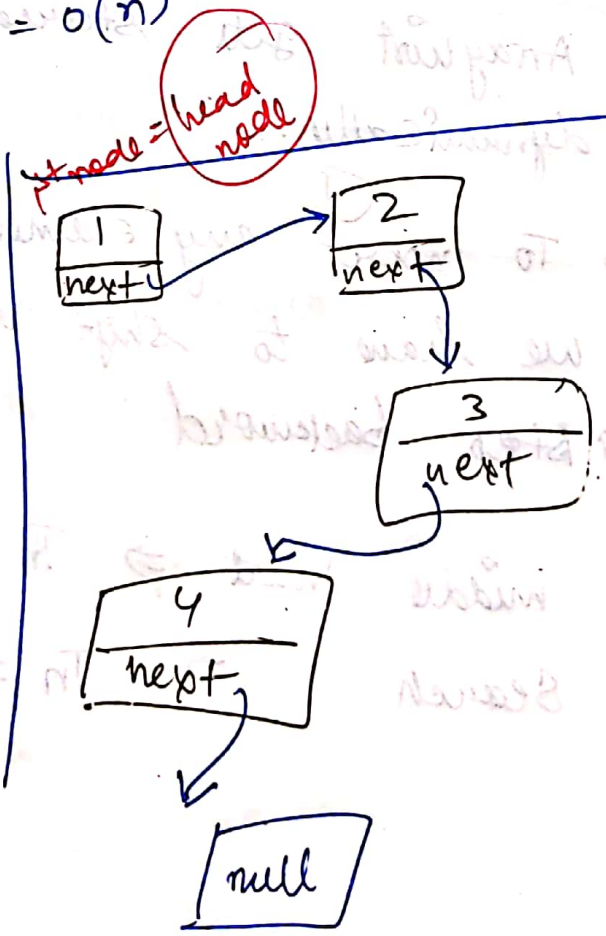
Linked List

* Variable size
* Non-contiguous Memory
* Insert → $T_n = O(1)$
* Search → $T_n = O(n)$

Basic Structure

⇒ Node

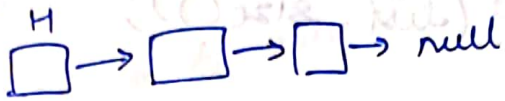




* Head → 1st node
* tail → last node

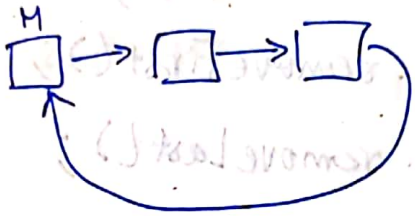
Types of L.L

(1) Singular



(2) Double

(3) Circular



Code

Linked List

L.L. Using Collection Framework

```
Import java.util.*;

class LL
{
    public static void main(String args[])
    {
        LinkedList<String> list = new LinkedList
                                  <String>();

        //add
        list.add("is"); → add to last
        list.add("a");
        list.addLast("list"); → last
        list.addFirst("this"); → first
        list.add(3, "linked"); → pass 'S'
        Sout(list);
```

```
//get value
sout(list.get(0));
sout(list.get(3));
sout(list.size());

//delete
  list.remove(3);
  list.removeFirst();
  list.removeLast();

  sout(list);

  //iterate
  for(int i=0; i<list.size(); i++)
    {
        sout(list.get(i)+"  ");
    }
    sout("null");

}//main

}//class
```

Linked List Scratch Implementation

```
class LL
{
    Node head;

    private int size;   → to calculate size of L.L.

    LL() // class constructer
    {
        size = 0
    }

    // Node class
    public class Node
    {
        String data;
        Node next;

        //constructor
        Node(String data)
        {
            this.data = data;
            this.next = null;
            size++;
        }
    } // class
```

// Add

```
public void addFirst(String data)
{
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
} //fn
```

```
public void addLast(String data)
{
    Node newNode = new Node(data);
    if(head == null)
    {
        head = newNode;
        return;
    }
    Node lastNode = head;
    while(lastNode.next != null)
    {
        lastNode = lastNode.next;
    }
    lastNode.next = newNode;
} //fn
```

```
// Print display
public void display()
{
    Node currNode = head;
    while (currNode != null)
    {
        sout(currNode.data + " ");
        currNode = currNode.next;
    }
} //fn
```

```
if (head == null)
{
    sout("Empty");
}
```

```
// Delete Element
public void removeFirst()
{
    if (head == null)
    {
        sout("Empty");
        return;
    }
    head = this.head.next;
    size--;
} //fn
```

```
public void removeLast()
{
    if (head == null)
    {
        sout("Empty");
        return;
    }
```

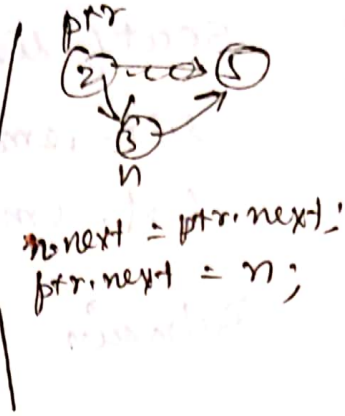
```
size--;
if (head.next == null)
{
    head = null;
    return;
}

Node currNode = head;
Node lastNode = head.next;

while (lastNode.next != null)
{
    currNode = currNode.next;
    lastNode = lastNode.next;
}

currNode.next = null;
}

//size
public int getSize()
{
    return size;
}

//Main fn
{
    LL list = new LL();
    list.addLast("is");
    list.addFirst("this");
    list.display();
```

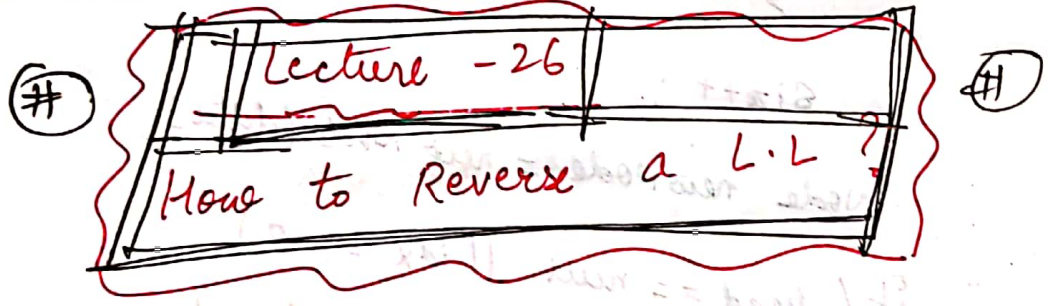
```
sout( list.getSize() );
list.removeLast();
list.removeFirst();
}//main
```

// insert in the middle of LL (at index 'i')

```
public void addInMiddle(int idx, String data)
{
    if(idx > size || idx < 0)
    {
        sout("Invalid");
        return;
    }
    size++;
    Node newNode = new Node(data);
    if(head == null || idx == 0)
    {
        newNode.next = head;
        head = newNode;
        return;
    }
    Node curr = head;
    for(int i=1; i<size; i++)
    {
        if(i == idx)
        {
```

```
Node nextNode = currNode.next;
currNode.next = newNode;
newNode.next = nextNode;
break;
}
currNode = currNode.next; // ptr = ptr.next;
}
}//fn
```





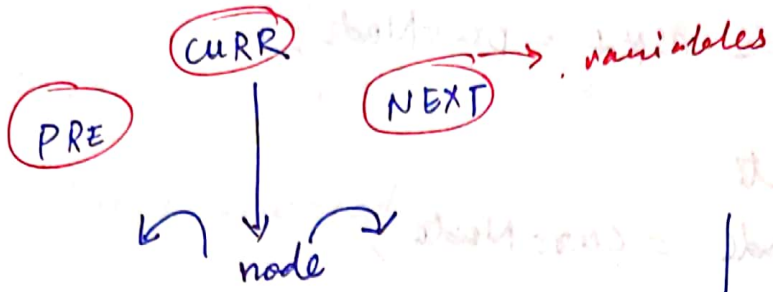
I/p

o/p

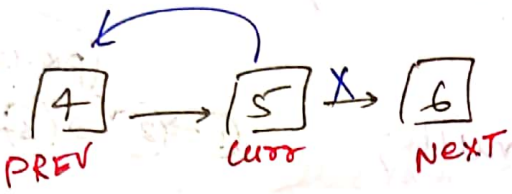
* no extra memory to be used

$T_n = O(n)$ (space complexity)

Iterative Approach

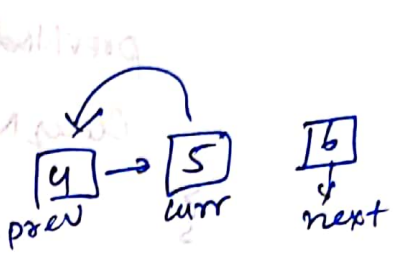


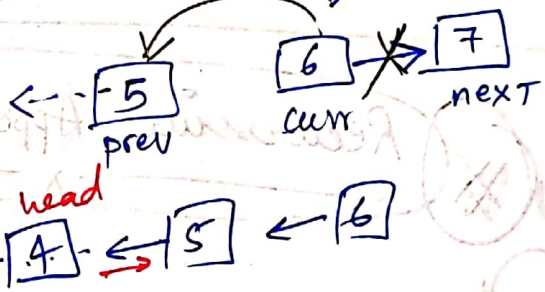
Eg:-



① curr.next = prev

②

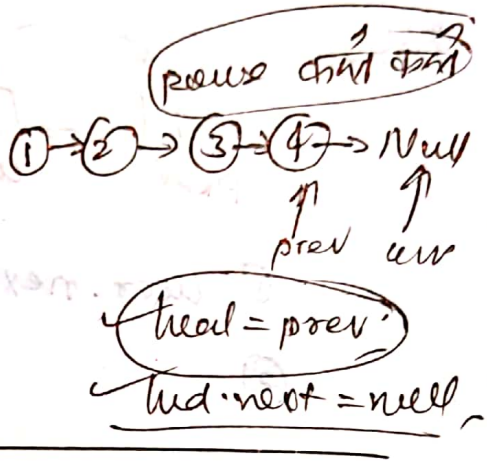




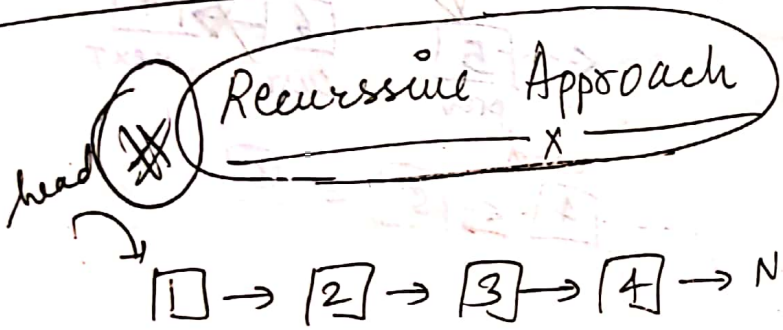
Code

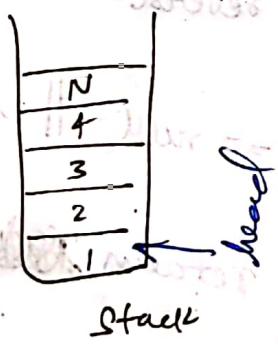
```
public void reverseIterate()
{
    if(head == null || head.next == null)
    {
        return;
    }
    Node prevNode = head;
    Node currNode = head.next;
    while(currNode != null)
    {
```

```
Node nextNode = currNode.next;
currNode.next = prevNode;

//update
prevNode = currNode;
currNode = nextNode;
}
head.next = null;
head = prevNode;
}//fn
```



Recursive Approach





Code

```
public Node reverseRecursive(Node head)
{
//Base Case
if(head == null || head.next == null)
{
    return head;
}
```

```
Node newHead = reverseRecursive(head.next);
    head.next.next = head;
    head.next = null;

    return newHead;
} //fn
```

Using Collection framework

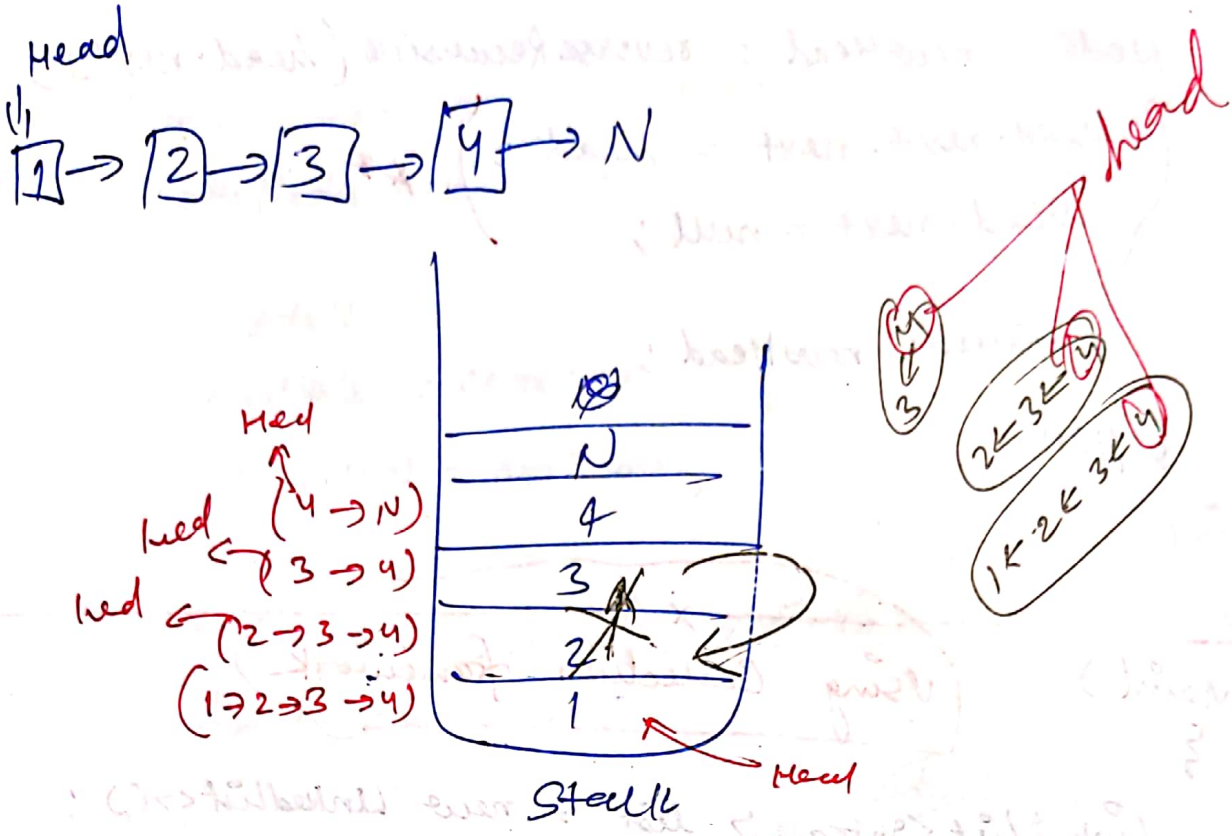
```
main()
{
    LinkedList<Integer> list = new LinkedList<>();
        list.add(1);
        list.add(2);
        Collection.reverse(list);
        sout(list);
} //main
```

---

op

1 → 2 → 3 → 4 → null (old)

4 → 3 → 2 → 1 → null (new)



(1) head.next.next = head;
(2) (3) (2)

(2) head.next = null;
(2) (3) → connection null,

(3) return head
(Upar se jo purana wala head aayega wahi yaha/hamesa head rahega)

Most Imp. Linked List Ques-

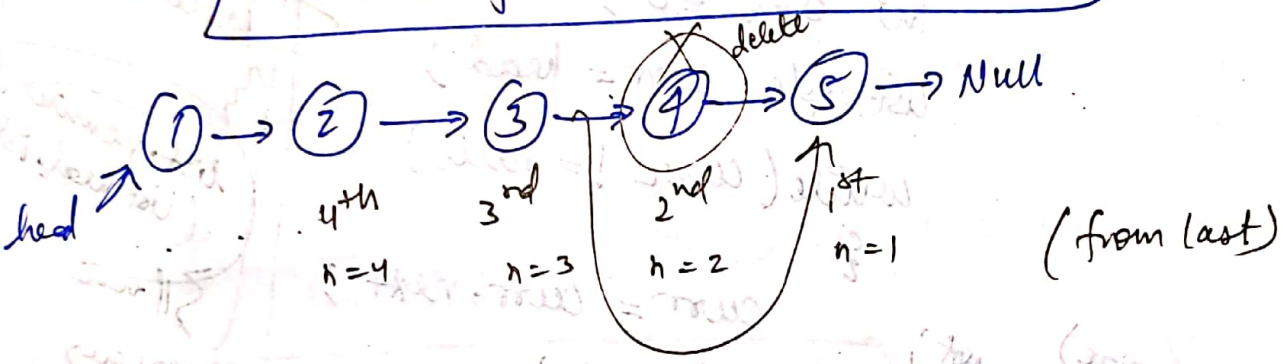
(Q1) Find the n^{th} node from the end & remove it.

$T_n = O(n)$ → time complexity

$O(1)$ → Space complexity

SM

n^{th} node from last + delete it



(i) Count size of L.L.
(traverse karke)

n (distance)
(n^{th} node from last)

(size − n + 1)

~~(n^{th} node from~~
(same from beg.)

(size − n)
(node less than prev wala jise find + delete karna hai)

Code

```
public ListNode removeNthFromEnd(ListNode head, int n)
{
    if (head.next == null)
    {
        return null;
    }

    //size
    int size = 0;
    ListNode curr = head;
    while (curr != null)
    {
        curr = curr.next;
        size++;
    }

    if(n == size)
    { return head.next;
    }

    int index = size - n;
    ListNode prev = head;
    int i = 1;
    while(i < index)
    {
        prev = prev.next;
        i++;
    }
    prev.next = prev.next.next;
    return head;
}//full
```

```
#ListNode = Node
main()
{
    list.remove(list.head.val);
}//main
}//class
```

(Ques 2) Check if a L.L. is a pallindrome ?

Sol^n)

Pallindrome

POP

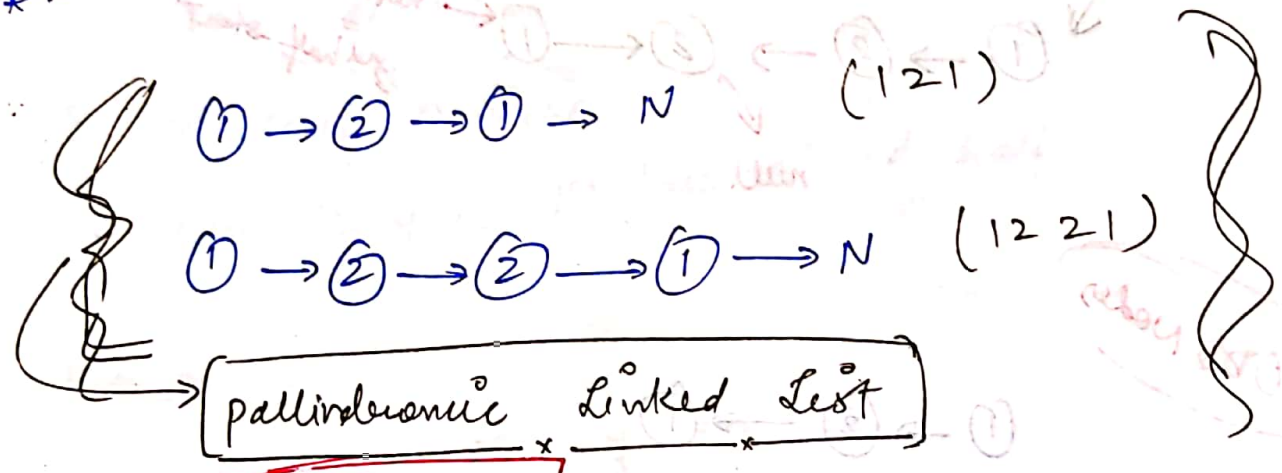
1221

141

BOB

start se ya last se read karne pe same aati ho

**



Method to do this

① with help of Array

② with help of Arraylist

value store karalo then check Array is pallindrome or not

⇓

But issues is that

⇓

Space - extra use kar rahe hai

⇓

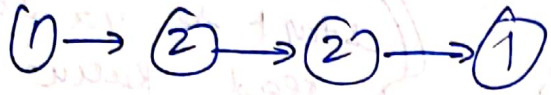
logic use Karo

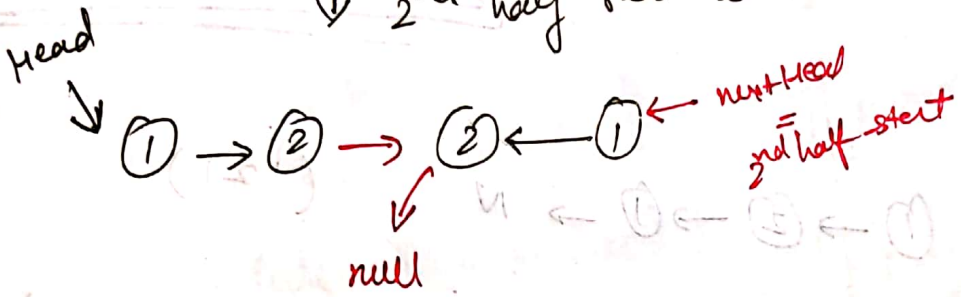
but computer ki space use hoti kam jada

* Divide the L.L into 2 equal parts

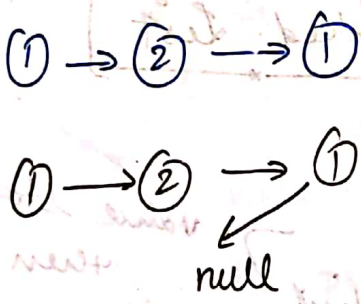
* Choose any 1 half (either 1st half || 2nd half) and make it reverse

Even Node:





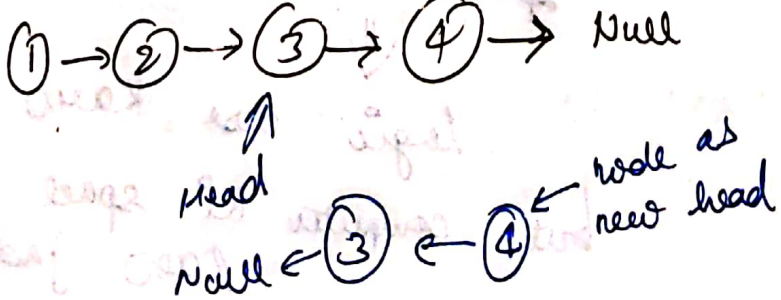
Odd Nodes

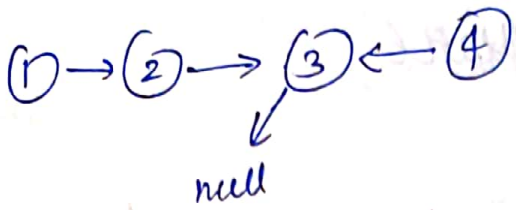


* after Reverse → Compare them.

1st half L.L → Compare → 2nd half L.L

** Note





* just change the head to the mid value jaha se reverse karna hai L-L-ko

new head = mid of L-L (while reversing our L-L)

Steps

① middle of L-L

② 2nd Half Reverse

③ check/compare 1st half & 2nd half.

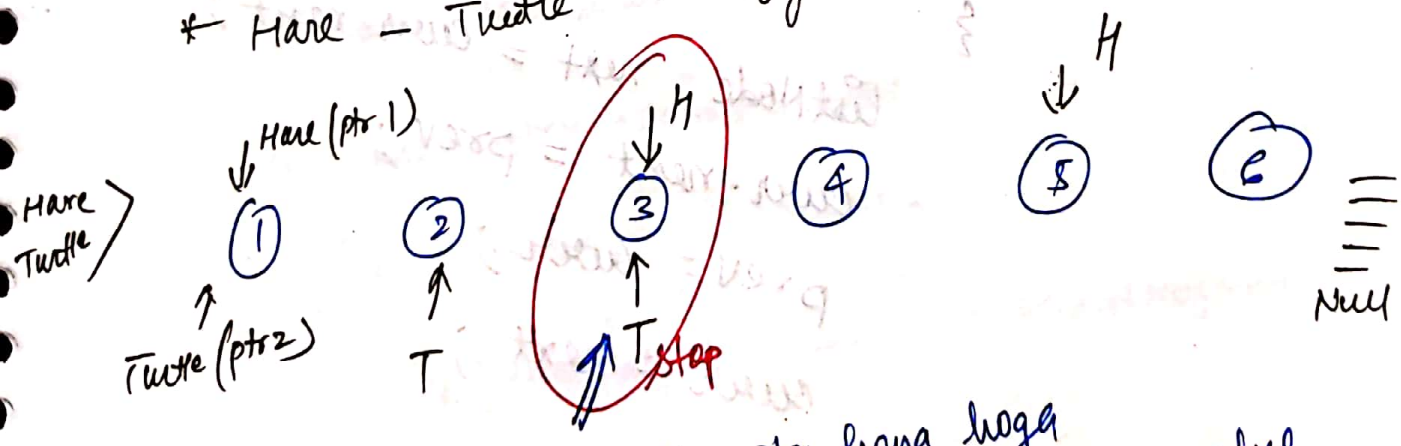
* **Code** *

To get mid value of L-L

(i) step-I : Calculate size
=> size/2

(ii) step-II :

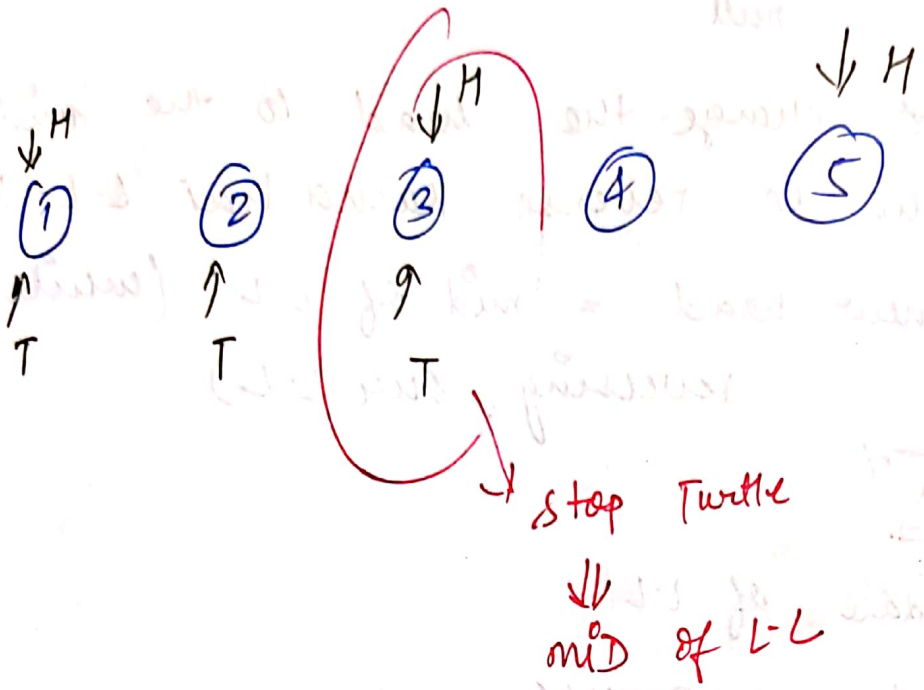
* Hare - Turtle strategy *



Turtle ko stop karna hoga
↓
gives us middle value

⇒ stop to save Hare.

odd



Code Java

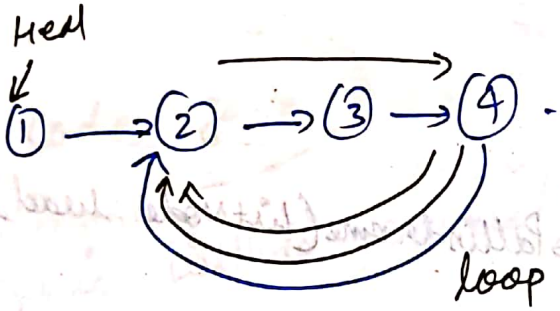
```
// reverse
public ListNode reverse(ListNode head)
{
    ListNode prev = null;
    ListNode curr = head;
    while(curr != null)
    {
        ListNode next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

3/10/21

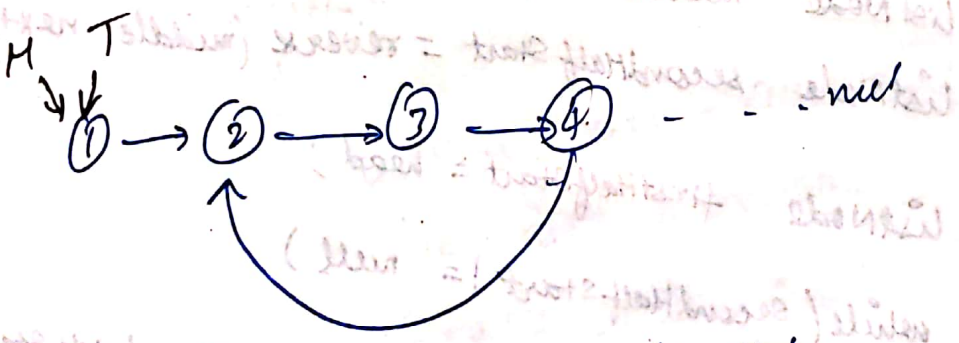
```
// get middle element
public ListNode findMiddle(ListNode head)
{
    ListNode hare = head; // fast
    ListNode Turtle = head; // slow

    while(hare.next != null && hare.next.next != null)
    {
        hare = hare.next.next;
        turtle = turtle.next;
    }
    return turtle;
}

//check pallindrome
public boolean isPallindrome(ListNode head)
{
    if(head == null || head.next == null)
    {
        return true;
    }

    ListNode middle = findMiddle(head);
    ListNode secondHalfStart = reverse(middle.next);

    ListNode firstHalfStart = head;

    while(secondHalfStart != null)
    {
        if(firstHalfStart.val != secondHalfStart.val)
        {
            return false;
        }
```

```
firstHalfStart = firstHalfStart.next;
secondHalfStart = secondHalfStart.next;
}// while
return true;
}// th
```

(Ques 3) Detecting Loop in a Linked List

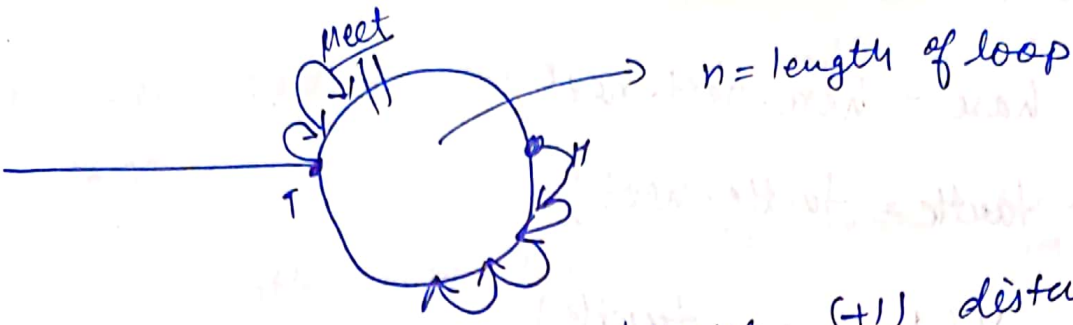


* Floyd's Algorithm of detecting loops
or Hare Turtle pointer



hare distance = 2 * (turtle)

x
x+1
x+2



Jo Toti ka beech hamesha (+1) distance ka gap hai & that will keep on increasing w.r.t time

x
x+1
x+2
.
.
.
n → total length of loop

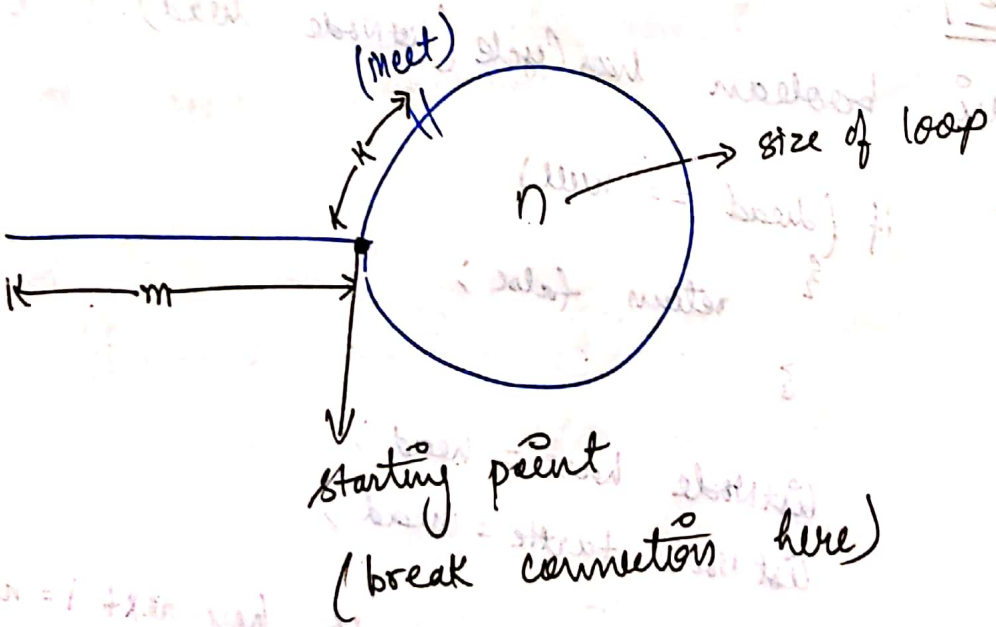
* when if M & T meet at any point then loop exists in linked list.

Code

```
public boolean hasCycle (ListNode head)
{
    if (head == null)
    {
        return false;
    }

    ListNode hare = head;
    ListNode turtle = head;

    while (hare != null && hare.next != null)
    {
```

```
hare = here.next.next;
tortle = tortle.next;
if( hare == tortle)
  {
     return true;
  }
}while
return false;
}if
```

After detecting this cycle,
Remove this cycle from L-L



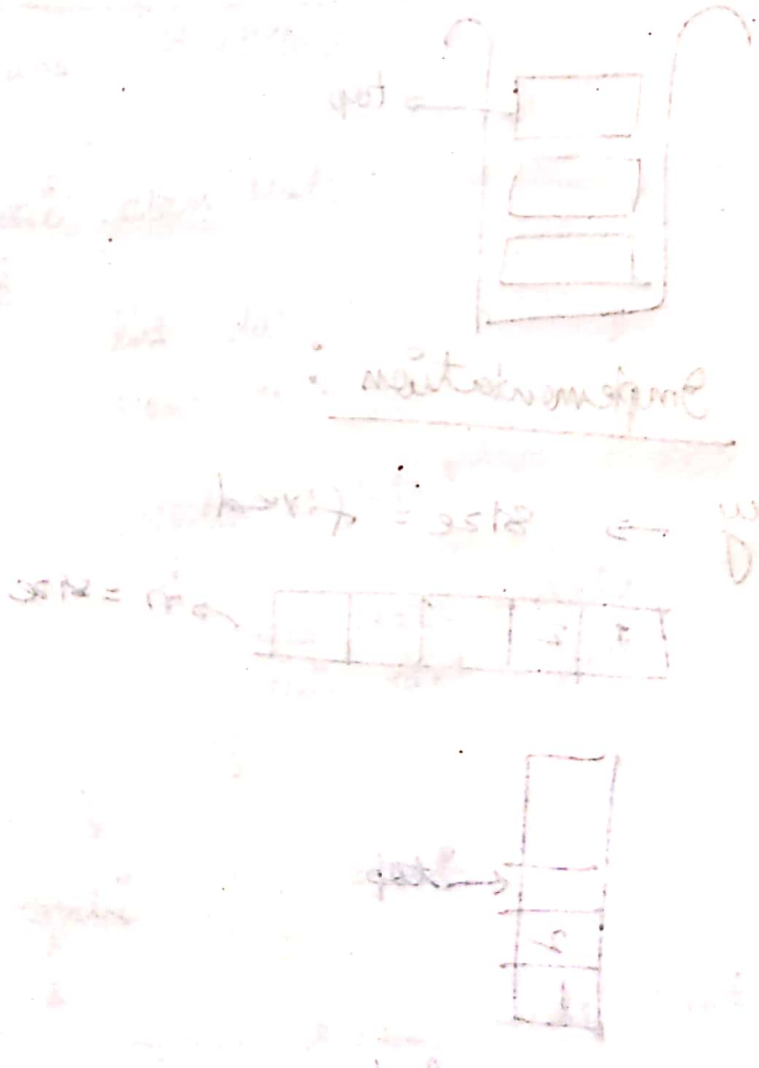
* when hare & turtle meet at a common point ;

① turtle ⟹ head (ki taraf lekar jao)

② Hare ko 1 step move karo

⇓

jaha para again meet karege wo starting point hoga.



* Lecture - 27 *

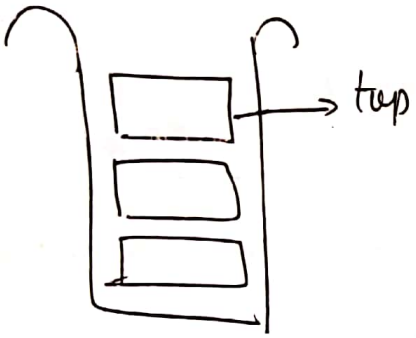
* Stack Data Structure in Java *

→ Push O(1) → insert at top

→ Pop O(1) → remove from top

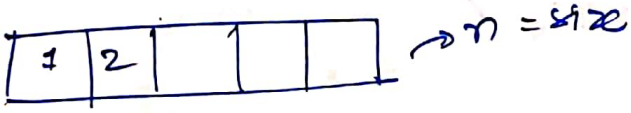
→ Peek O(1) → top element ko nikalna

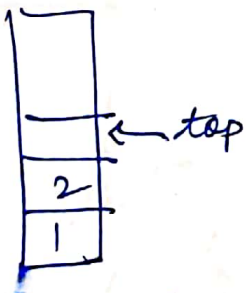
* LIFO (last in first out)



Implementation :

(1) Array → size = fixed

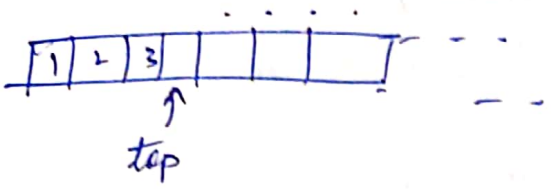




1-) Stack full

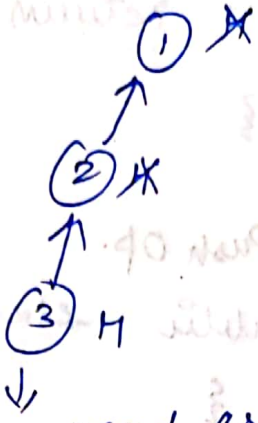
② Array List

Size = variable



③ Linked List

Size = variable



Treat Head as top

∴ Tn = O(1)

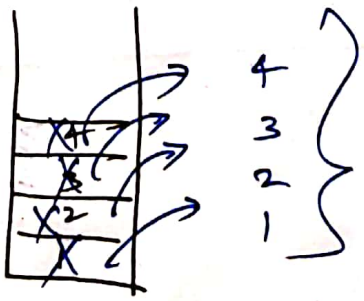
otherwise Tn > O(1)

Code → Using Linked List

```
public class StackClass
{
    static class Node
    {
        int data;
        Node next;

        Node(int data)
        {
            this.data = data;
            this.next = null;
        }
    }
    static class Stack
    {
        public static Node head;

        // check for empty
```

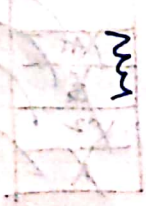
```
public static boolean isEmpty()
{
    return head == null;
}

//Push Op.
public static void push(int data)
{
    Node newNode = new Node(data);
    if(isEmpty())
    {
        head = newNode;
        return;
    }
    newNode.next = head;
    head = newNode;
}

//Pop. Operation
public static int pop()
{
    if(isEmpty())
    {
        return -1;
    }
    int top = head.data;
    head = head.next;
    return top;
}
```

```
//Peek Op.
public static int peek()
{
    if (isEmpty())
    {
        return -1;
    }
    return head.data;
}//fn
}//class

//main fn
void main()
{
    Stack s = new Stack();

    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    while (!s.isEmpty())
    {
        sout(s.peek());
        s.pop();
    }
}//main
}//class
```



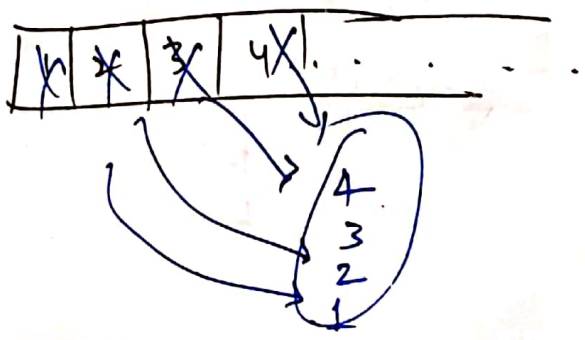
Code – Using Arraylist

```
import java.util.*;
public class StackClass
{
  static class Stack
    {
      Arraylist<Integer> list = new Arraylist<>();
      //check for empty
      public static boolean isEmpty()
      {
        return list.size == 0;
      }
      //Push Op:
      public static void push(int data)
      {
        list.add(data);
      }
      //Pop. Op:
      public static int pop()
      {
        int top = list.get(list.size()-1);
        list.remove(list.size()-1);
        return top;
      }
```



```
// Peek Op.
Public static int peek()
{
    if(isEmpty())
    {
        return -1;
    }
    return list.get(list.size()-1);
}
}
main()
{
    Stack s = new Stack();
    s.push(1);
    s.push(2);
    s.push(3);
    //display
    while(!s.isEmpty())
    {
        Sout(s.peek());
        s.pop();
    }
}//main
}//class
```

Array List

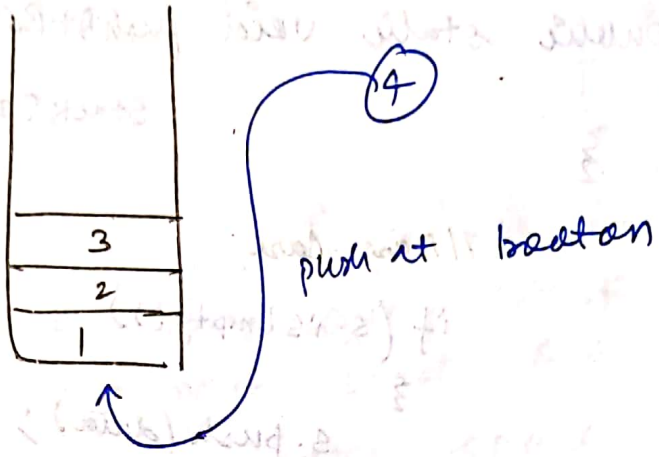


Code → Using Collection Frame Work

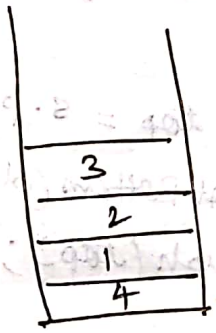
```
import java.util.*;
public class StackClass
{
    main()
    {
        Stack<Integer> s = new Stack<>();
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);

        while(!s.isEmpty())
        {
            sout(s.peek);
            s.pop();
        }
    }//main
}//class
```

(Ques 1) To push an element at the bottom of a stack

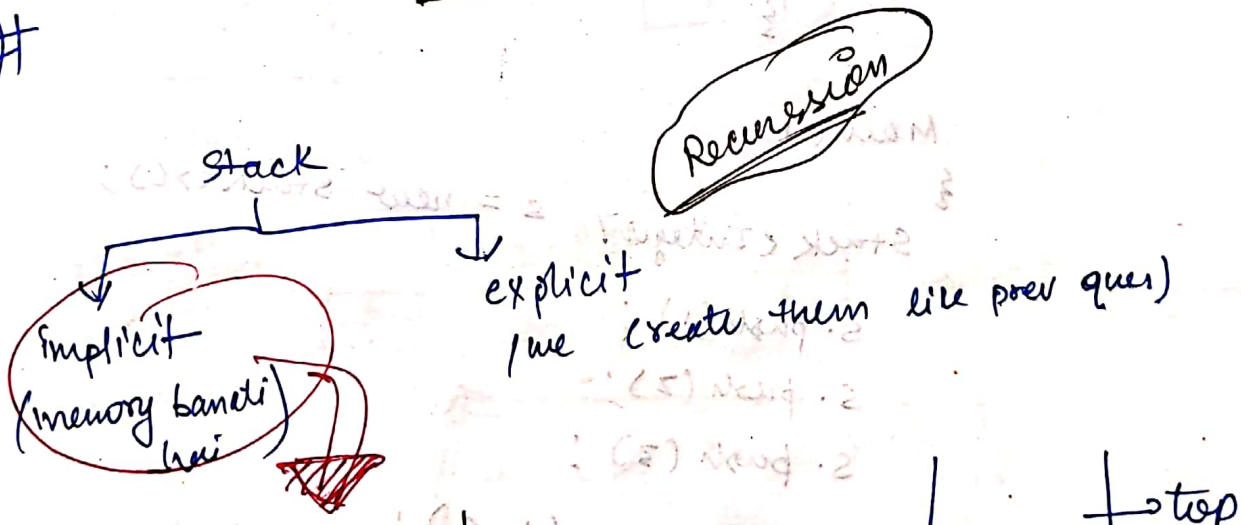
I/p :

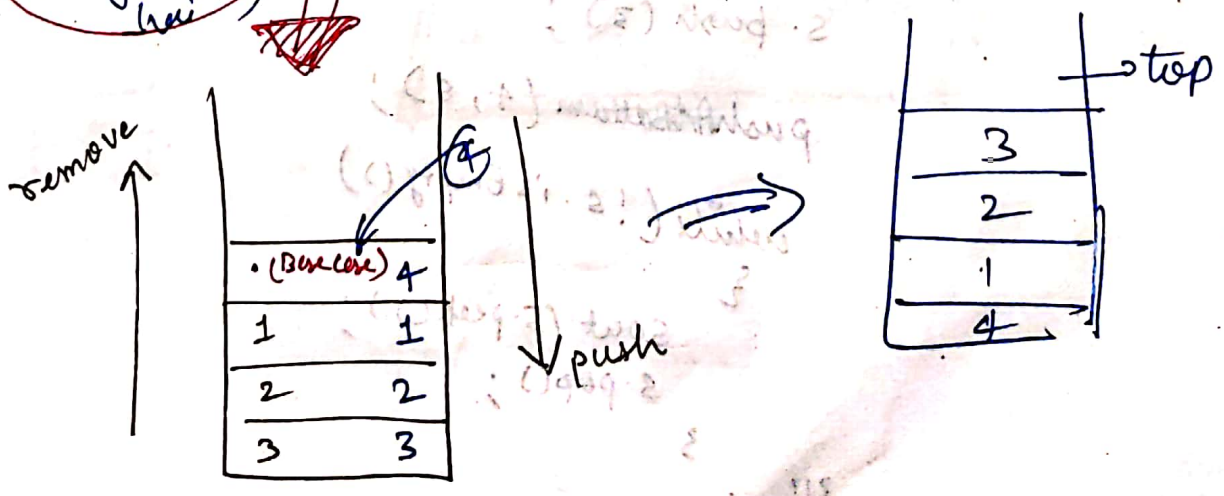


O/p :



#

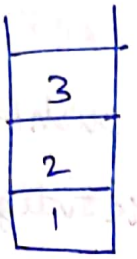




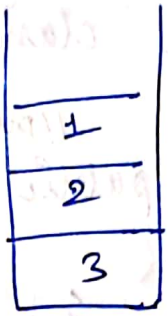
```
import java.util.*;

public class StackClass
{
    public static void pushAtBottom(int data,
                                     Stack<Integer> s)
    {
        //Base Case
        if(s.isEmpty())
        {
            s.push(data);
            return;
        }
        int top = s.pop();
        pushAtBottom(data, s);
        s.push(top);
    }

main()
{
    Stack<Integer> s = new Stack<>();
    s.push(1);
    s.push(2);
    s.push(3);

    pushAtBottom(4, s);

    while(!s.isEmpty())
    {
        sout(s.peek());
        s.pop();
    }
}//main
}// class
```

Ques 2) Reverse a Stack

I/P : (Given)



O/P :

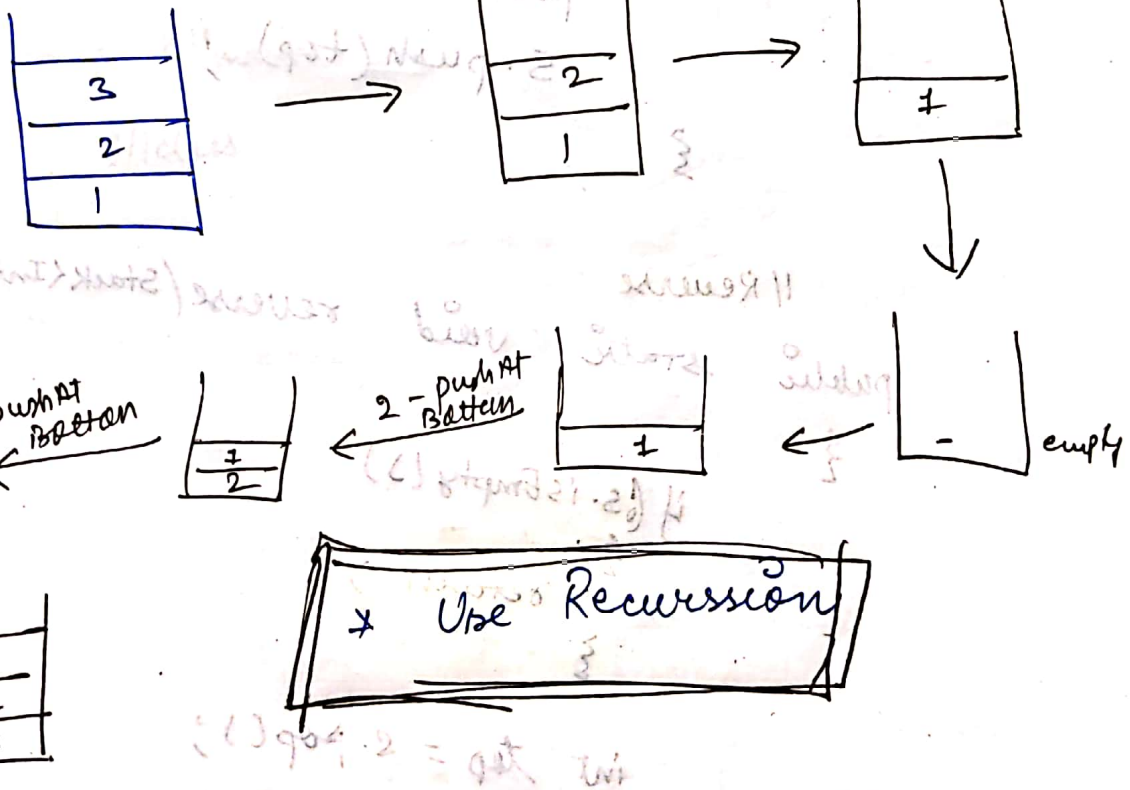


Method - 1

⇒ pop element from given stack & store them in a new stack one by one.

→ but will req. extra space

Method - 2



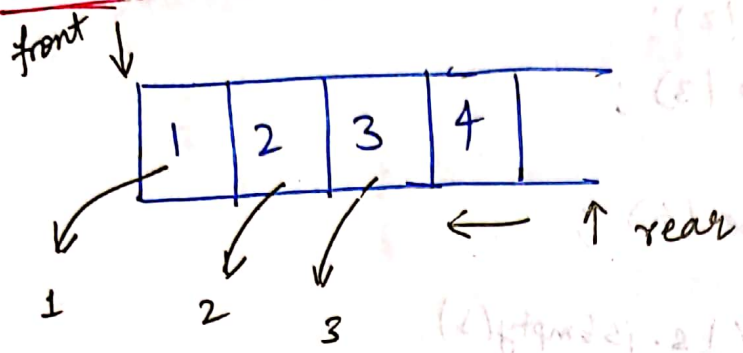
Code

```
public class StackClass
{
    //push at Bottom
    public static void pushAtBottom(int data,
                                    Stack<Integer> s)
    {
        if (s.isEmpty())
        {
            s.push(data);
            return;
        }
        int top = s.pop();
        pushAtBottom(data, s);
        s.push(top);
    }

    //Reverse
    public static void reverse(Stack<Integer> s)
    {
        if (s.isEmpty())
        {
            return;
        }
        int top = s.pop();
        reverse(s);
        pushAtBottom(top, s);
    }
}
```

```
main()
{
    Stack<Integer> s = new Stack<>();
    s.push(1);
    s.push(2);
    s.push(3);

    reverse(s);

    while(!s.isEmpty())
    {
        sout(s.peek());
        s.pop();
    }
}//main
}//class
```

Lecture - 28

Queue Data Structure in Java



FIFO (First In First Out)

Operations

1. Enque → Add → adding elements
2. Dequeue → Remove → removing element
3. Front → Peek → front element ko dekna hai bas.

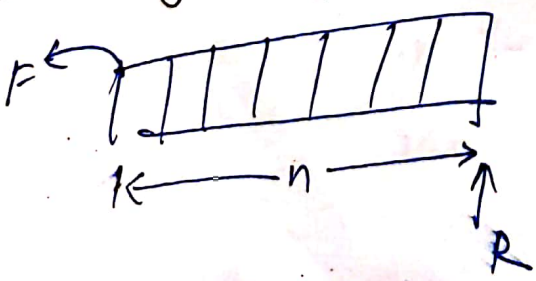
Implementation

1. Using Array (size fixed)

size = n

F => 0

R => idx last



* initially:

F = -1
R = -1
} when Queue is Empty

as we add 1 elemt to empty Que,

the F = +1, R = +1

Now as we keep on adding elements to Queue then F = 1 (const) but R = ( )

→ value keeps on incresing

Code - Using Array

```
public class QueueClass
{
    //class Queue
    static class Queue
    {
        static int arr[];
        static int size;
        static int rear = -1;

        // Constructor
        Queue(int n)
        {
            arr = new int[n];
            this.size = n;
        }

        // Empty Check
        public static boolean isEmpty()
        {
            return rear == -1;
        }
```

```
//enqueue
public static void add(int data)
{
    if(rear == size-1)
    {
        sout("full Queue");
        return;
    }
    rear++;
    arr[rear] = data;
}

//dequeue
public static int remove()
{
    if(isEmpty())
    { sout("Empty Queue");
      return -1;
    }
    int front = arr[0];
    for(i=0; i<rear; i++)
    {
        arr[i] = arr[i+1];
    }
    rear--;
    return front;
}
```

$T_n = O(1)$ → add

$T_n = O(n)$ → remove + peek } using Array

```
//Peek Op.
public static int peek()
{
    if(isEmpty())
    {
        sout("Empty");
        return -1;
    }
    return arr[0];
}

main()
{
    Queue q = new Queue(5);
    q.add(1);
    q.add(2);
    q.add(3);

    while(!q.isEmpty())
    {
        sout(q.peek());
        q.remove();
    }
}//main

}//class
```

Implementation - 2

Circular Queue Using Array

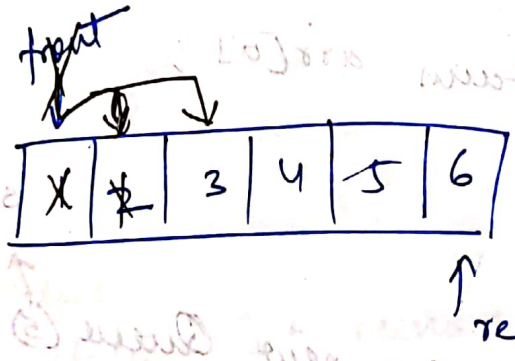
Queue (Using Array) **Circular Queue**

add → O(1)

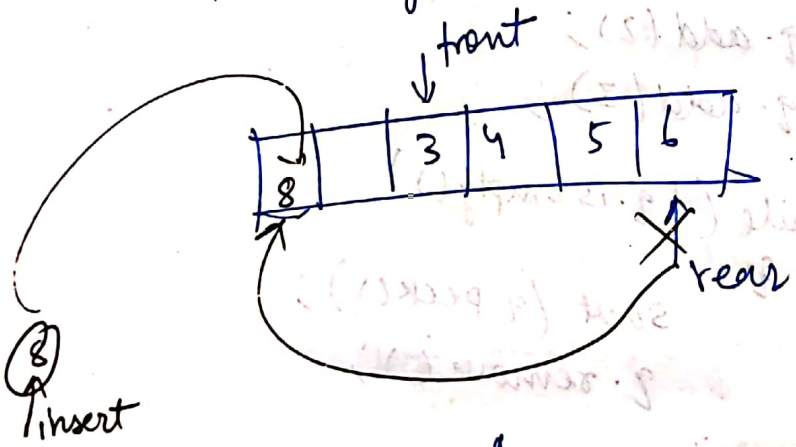
peek + remove → O(n)

add + remove + peek → O(1)

Note:



Move front by 1 after deleting 1 element.



↳ but rear to back no space

↓

rear will incre by 1 and move to start making a circle

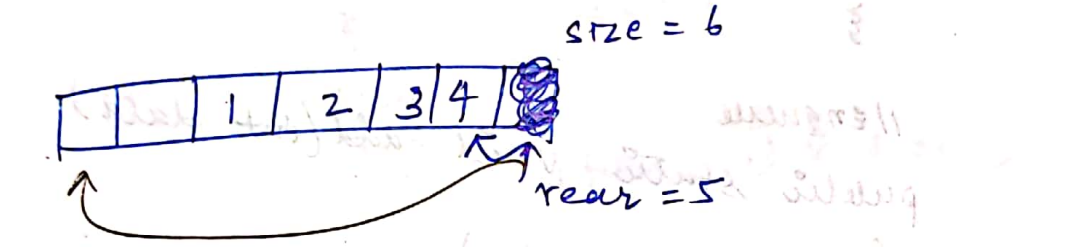
** Rear + 1 = front (Queue full)

Rear = -1
Front = -1
} Initially (beg.)

Now, instead of rear++

⇓

rear = (rear + 1) % size ;



(5+1) % 6 = 6 % 6 = 0

(Rear + 1) % size == front → Queue full condition

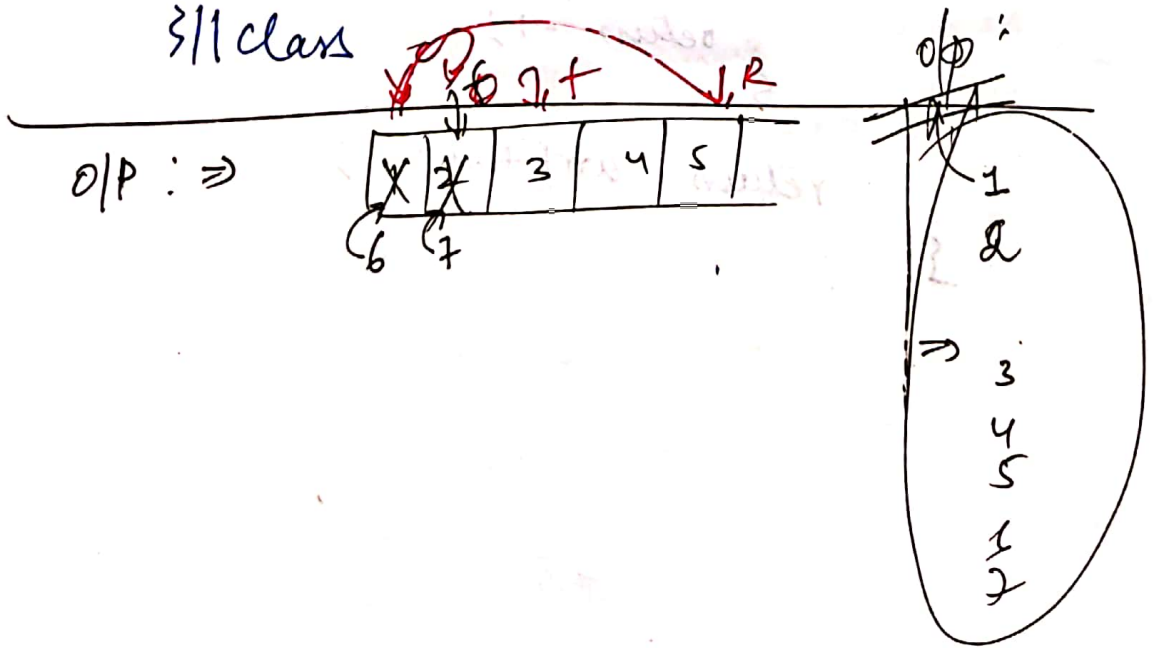
Code

```
public class QueueClass
{
    static class Queue
    {
        static int arr[];
        static int size;
        static int rear = -1;
        static int front = -1;
        Queue(int n)
        {
            arr = new int[n];
        } this.size = n;
```

```
//Empty check
public static boolean isEmpty()
{
    return rear == -1 && front == -1 ;
}

//full check
public static boolean isFull()
{
    return (rear +1) % size == front ;
}

//enqueue
public static void add(int data)
{
    if(isFull())
    {
        sout("Full");
        return ;
    }
    if(front == -1)
    {
        front = 0;
    }
    rear = (rear +1)% size ;
    arr[rear] = data ;
}
```

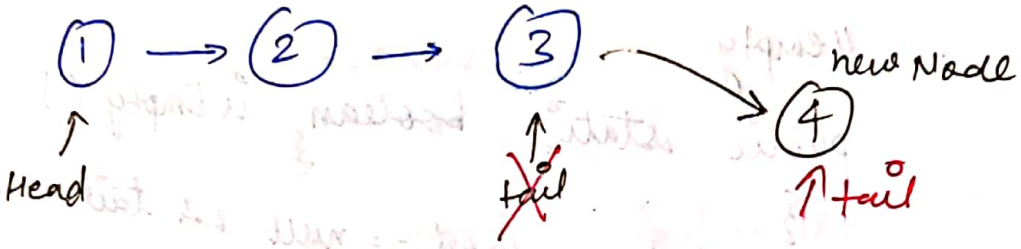
```
//dequeue
public static int remove()
{
    if(isEmpty())
    {
        sout(" Empty");
        return -1;
    }

    int result = arr[front];
    //single element
    if(front == rear)
    {
        front = rear = -1;
    }
    else
    {
        front = (front + 1) % size;
    }

    return result;
}

//Peek
public static int peek()
{
    if(isEmpty())
    {
        sout("E.Q");
        return -1;
    }
    return arr[front];
}
```

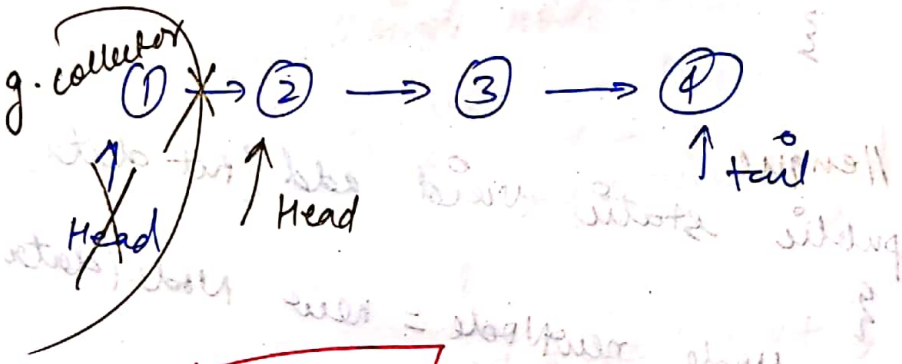
```
main()
{
    Queue q = new Queue();
    q.add(1);
    q.add(2);
    q.add(3);
    q.add(4);
    q.add(5);

    sout(q.remove());
    q.add(6);
    sout(q.remove());
    q.add(7);

    sout("\n\n");
    while(!q.isEmpty())
    {
        sout(q.remove());
    }
}//main
}//class
```



Implementation - 3

Using Linked List





Code

```
public class QueueClass
{
    static class Node
    {
        int data;
        Node next;

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

```
Static class Queue
{
    Static Node head = null;
    Static Node tail = null;

    //empty
    public static boolean isEmpty()
    {
        return head == null && tail == null;
    }

    //enque
    public static void add(int data)
    {
        Node newNode = new Node(data);
        if(isEmpty())
        {
            tail = head = newNode;
        }
        else
        {
            tail.next = newNode;
            tail = newNode;
        }
    }//fn
```

```
// dequeue Op.
public static int remove()
{
    if(isEmpty())
    {
        sout("E.Q");
        return -1;
    }
    int front = head.data;
    // single node
    if(head == tail)
    {
        tail = null;
    }
    head = head.next;
    return front;
}//fn

// Peek Op.
public static int peek()
{
    if(isEmpty())
    {
        sout("E.Q");
        return -1;
    }
    return head.data;
}//fn
}//class
```

```
Main()
{
    Queue q = new Queue();
      q.add(1);
      q.add(2);
      q.add(3);

      //display
      while(!q.isEmpty())
       {
          sout(q.peek());
          q.remove();
       }
}//main
}//class
```

Implementation - 4

Using Java Collection

Note:

+ Stack is an interface in Java Collection
+ We can't make Object with Interface so, we need other → LinkedList, → ArrayDeque

to use Queue interface.

Queue<Integer> q = new LinkedList();

or Queue<Integer> q = new ArrayDeque();

Code

```
import java.util.*;
public class QueueClass
{
    main()
    {
    Queue<Integer> q = new LinkedList();
     q.add(1);
     q.add(2);
     q.add(3);
      // empty
     while(!q.isEmpty())
     {
         sout(q.peek());
         q.remove();
     }
}// main
}// class
```

** Queue Using 2 stacks

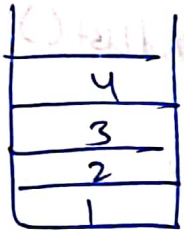
Que

Add / Push
O(n)

Pop / Peek + Remove
O(n)

Implementing Queue Using 2 Stacks

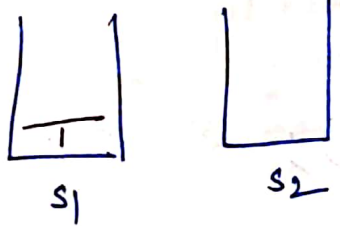
Stack
**



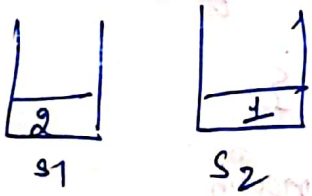
pop : ⇒ 4 3 2 1
(LIFO principle)

2 Stack

Insert : 1 2 3 4

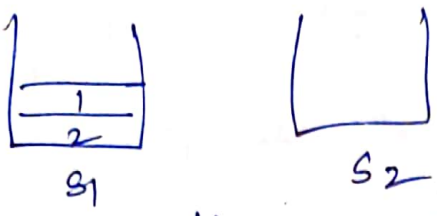


remove all elemt from S1 & put them in S2 then add new elemt in S1

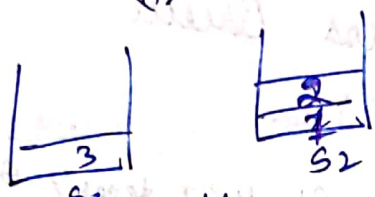


put S2 elemet in S1

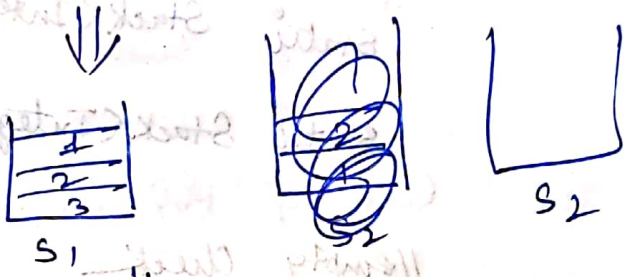
⇒



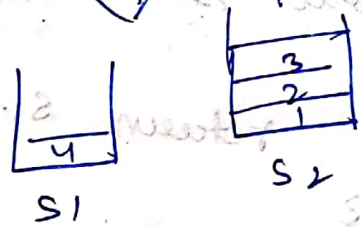
Remove S_1 element to S_2 + insert 3 in S_1



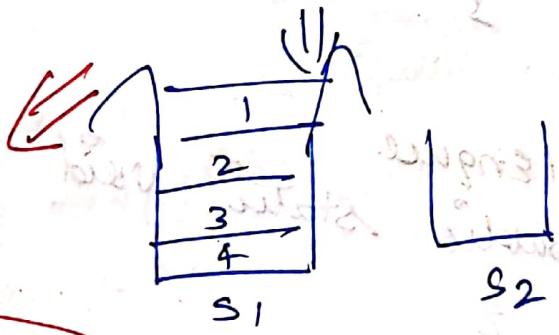
put S_2 elent in S_1



keep $S_1 \rightarrow S_2$



pop



1 2 3 4

FIFO principle

Queue. (implemented)

Code

```
import java.util.*;
public class QueueClass
{
    static class Queue
    {
        static Stack<Integer> s1 = new Stack<>();
        static Stack<Integer> s2 = new Stack<>();

    //empty Check
    public static boolean isEmpty()
    {
        return s1.isEmpty();
    }

    //Enque
    public static void add(int data)
    {
        while(!s1.isEmpty())
        {
            s2.push(s1.pop());
        }
        s1.push(data);
```

```
while (s2.isEmpty())
{
    s1.push(s2.pop());
}

//dequeue
public static int remove()
{
    return s1.pop();
}

//peek
public static int peek()
{
    return s1.peek();
}

Main()
{
    Queue q = new Queue();
    q.add(1);
    q.add(2);
    q.add(3);

    while (!q.isEmpty())
    {
        sout(q.peek())
        q.remove();
    }
}//Main
}//class
```

Lecture - 29

Hashset in Java

* Set → Unique elements

Hashset → Data Structure → very efficient data structure

⇓

Bzc it's Time Complexity

* Insert / Add → O(1)
* Search / Contains → O(1)
* Delete / Remove → O(1)

Data Structure → / Operation	Hashset	Array	Sorted array	B.S. Tree Best	B.S. Tree worst
+ Insert/Add	O(1)	O(1)	O(n)	O(h)	(O(n))
* Search / Contains	O(1)	O(n)	O(log n)	O(h)	O(n)
* Delete / Remove	O(1)	O(n)	O(n)	O(h)	O(n)

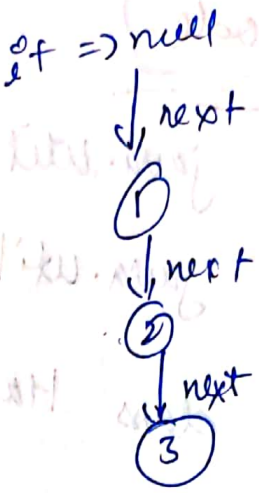
O(log n) → BST

* Contains Tn → makes DSA efficient

Constant Tn for Hashset

working of iterator

Set = [1, 2, 3]



* set is a black box
  ⇓
  order not matter

it.next() ⇒ 1
it.next() → 2
it.next() → 3

it.hasnext() → true
→ false

```
while (it.hasNext())
{
    sout(it.next())
}
```

⇒ order doesn't remains constant overtime.

Code

```
import java.util.HashSet;
import java.util.Iterator;

public class Hashing
{
    void main()
    {
        HashSet<Integer> set = new HashSet<>();

        //Add
        set.add(1);
        set.add(2);
        set.add(3);
        set.add(2);

        //size
        sout(set.size());

        //Search
        if(set.contains(1))
        {
            sout("Present");
        }
        if(!set.contains(6))
        {
            Sout("Not..!");
        }
```

```
// delete
set.remove(1);
if(!set.contains(1))
{
    sout("Absent");
}

//print all elements
sout(set);
```

** //Iterator -- Hashset doesn't have any cart. order

```
set.add(0);
Iterator it = set.iterator();
while(it.hasNext())
{
    sout(it.next() + " ");
}

//isEmpty
if(!set.isEmpty())
{
    sout("set is Empty");
}
}//main
}//class
```

Lecture -30

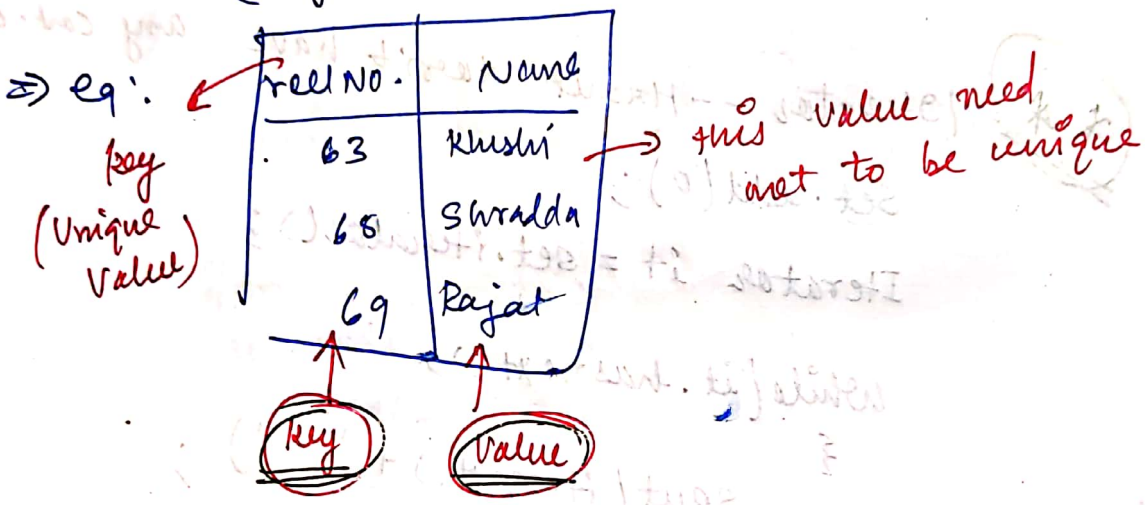
HashMap in Java

HashMap

⇒ Used to store pair

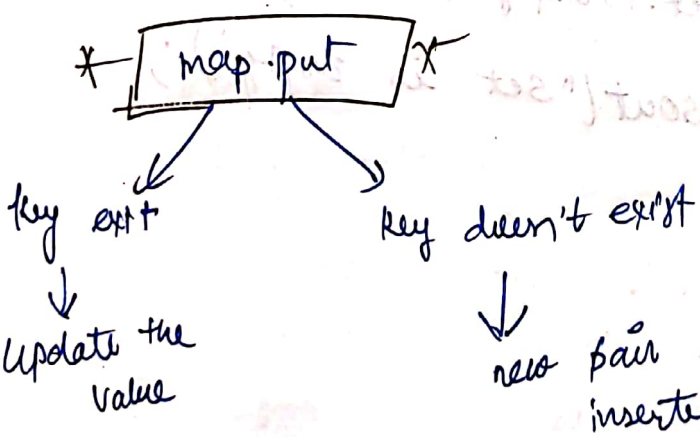
(key ↔ value)

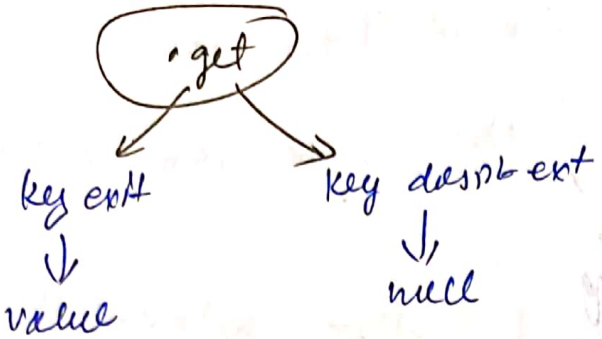
⇒ eg:

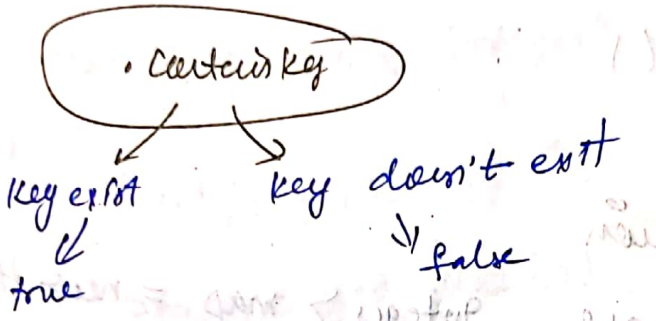


⇒ Unordered map → sequence of (key, value)

may change with time.

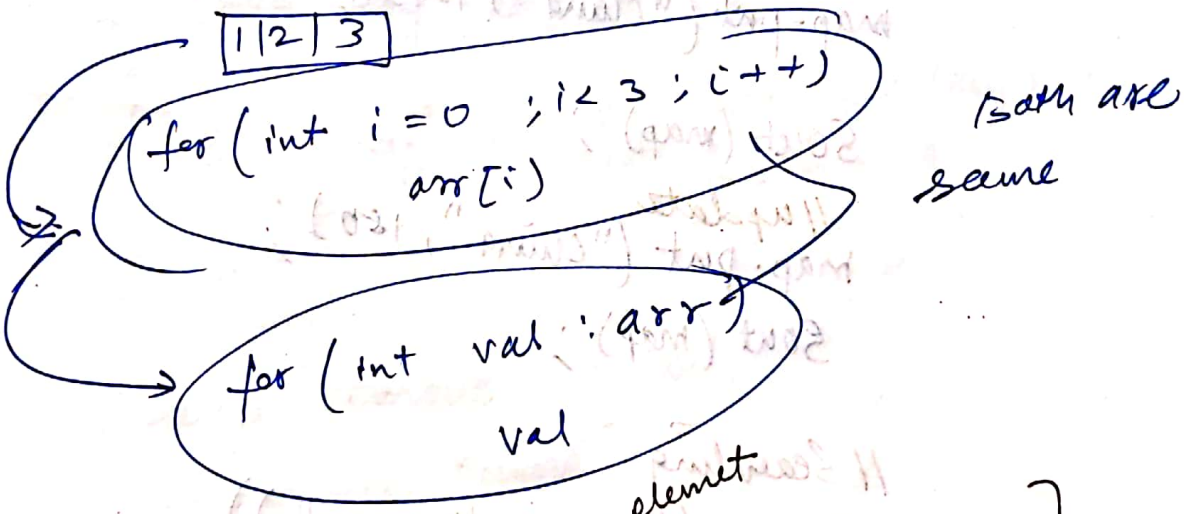






Iteration in HashMap

for (val : collection)



for (Map.Entry<Integer, Integer> e : map.entrySet())

e → element

e.getKey() → key

e.getValue() → value

Code

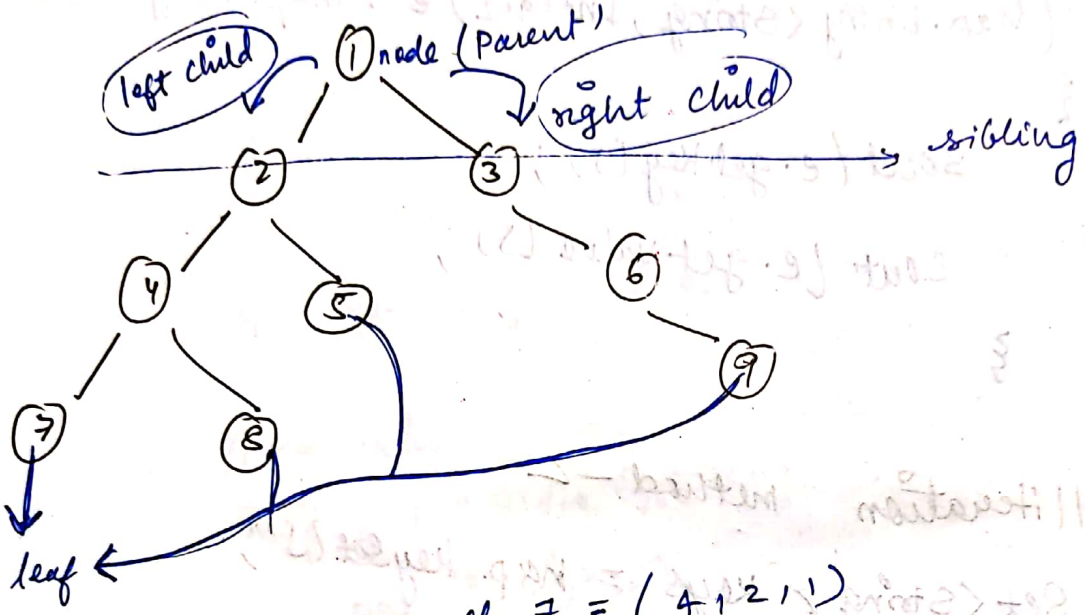
```
import java.util.*;

public class Hashing
{
    ..main()
    {
      // creation
    HashMap<String, Integer> map = new HashMap<>();

    //insertion
    .map.put("India", 120);
    map.put("US", 30);
    map-put("China", 150);

     Sout(map);
     //update
    map.put("China", 150);
     Sout(map);

    // Searching
    if(map.containsKey("Indo"))
      {
         Sout("Key Present");
      }
    else
      {
         Sout("Key Absent");
      }
```

```
sout(map.get("China")); // key exist
sout(map.get("Indo")); // key not exist

// iteration method-1
for(Map.Entry<String, Integer> e : map.entrySet())
{
    sout(e.getKey());
    sout(e.getValue());
}

// iteration method-2
Set<String> keys = map.keySet();
for(String key : keys)
{
    sout(key + " - " + map.get(key));
}

// delete / remove
map.remove("China");
sout(map);

}//main
}//class
```

Lecture - 31

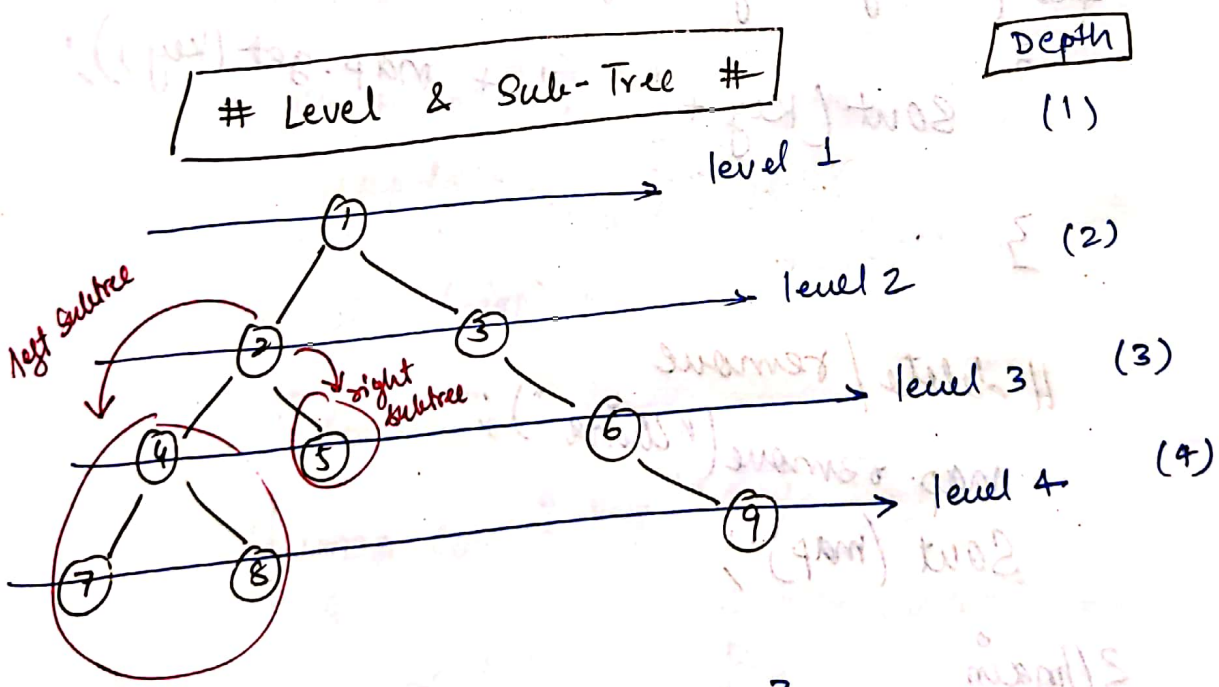
Binary Tree

* At max → 2 Children



Ancestors of 7 = (4, 2, 1)

Level & Sub-Tree



Level of node = 5 ⇒ L-3

Depth of node = 5 ⇒ 3

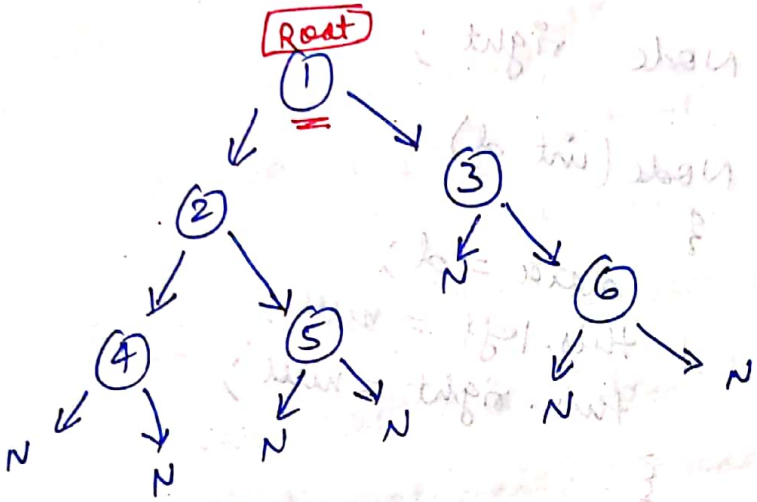
Build Tree → Preorder Sequence

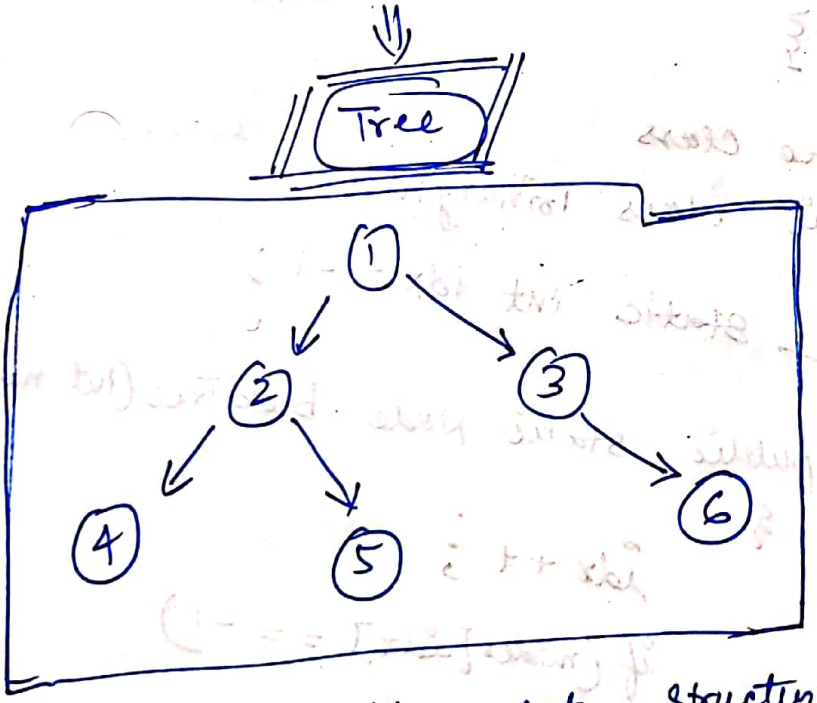
⇒ (1, 2, 4, -1, -1, 5, -1, -1, 3, -1, 6, -1, -1)

(Pre-order sequence given).

* -1 ⇒ represent Null node (N = null)

1st ⇒ root Node





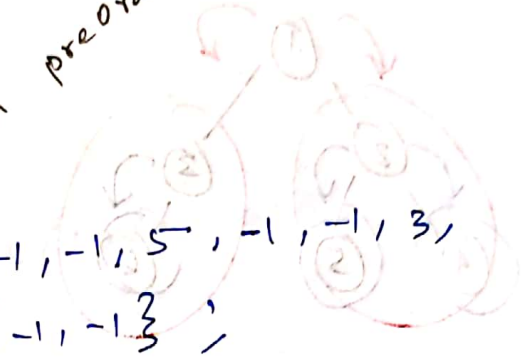
* Tree data structure *

Code :→ To Create a Binary Tree

```
class BinaryTreeYT
{
    // node class
    static class Node
    {
        int data;
        Node left;
        Node right;

        Node(int d)
        {
            data = d;
            this.left = null;
            this.right = null;
        }
    }

    // Tree class
    static class BinaryTree
    {
        static int idx = -1;

        public static Node buildTree(int nodes[])
        {
            idx++;
            if(nodes[idx] == -1)
            {
                return null;
            }
```

to travel at each index ←

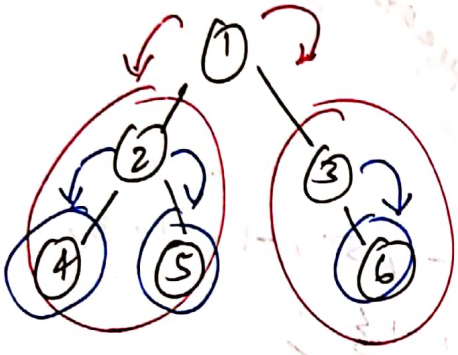
```
Node newNode = new Node(nodes[idx]);
newNode.left = buildTree(nodes); // left subtree
newNode.right = buildTree(nodes); // right subtree
return newNode; // root node will be returned at the end

} // fn
} // class

main()
{
    int nodes[] = {1,2,4,-1,-1,5,-1,-1,3,-1,6,-1,-1};   // data in preorder sequence
    BinaryTree tree = new BinaryTree();
    Node root = tree.buildTree(nodes);
    sout(root.data); // root of tree
} // main
} // class
```



Traversal in Binary Tree

① Pre-Order

Root → left subtree → right subtree



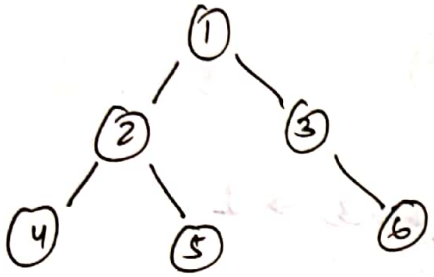
1 → (2 → 4 → 5) → (3 → 6)

Code:

```
public static void preOrder(Node root)
{
    //base case
    if (root == null)
    {
        return;
    }
    sout (root.data + " ");
    preOrder (root.left);
    preOrder (root.right);
}//fun

Main()
{
    preOrder(root);
}
```

② In Order

Left subtree → root → right subtree



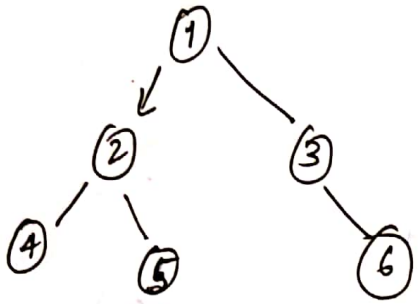
(4 → 2 → 5) → 1 → (3 → 6)

Code :-

```
public static void inOrder(Node root)
{
    if (root == null)
    {
        return ;
    }
    inOrder(root.left);
    sout(root.data + " ");
    inOrder(root.right);

} //fn

main()
{
    inOrder(root);

}
```

(3) Post-Order

left subtree → right subtree → root

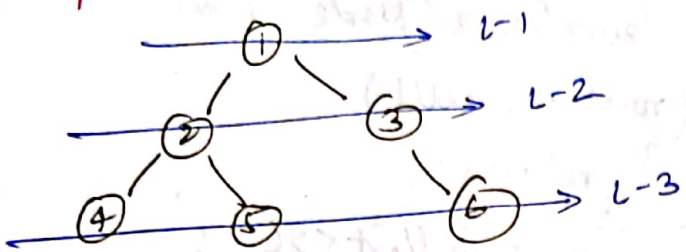


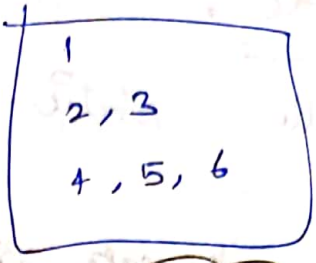
(4 → 5 → 2) → (6 → 3) → 1

Code:

```
public static void postOrder(Node root)
    {
        if (root == null)
            {
                return;
            }
        postOrder(root.left);
        postOrder(root.right);
        sout(root.data + "  ");
    }//fn

main()
{
    postOrder(root);
}
```

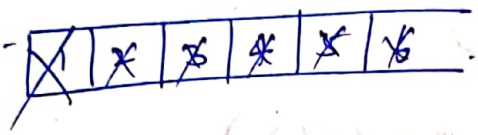
(5) Level-Order





FIFO property use kar

Use Queue Data structure



* O/P: (1 2 3 4 5 6)

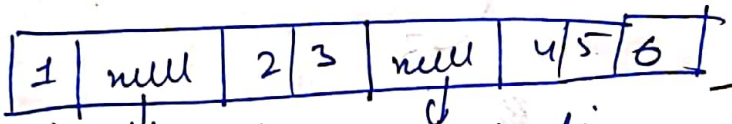
Store 1 → when level 1 over

remove 1

print (1)

store all children of (1) - node after print (1)

so.. on & so... forth ...



→ store null in Queue after storing Queue next node to

Queue

print (1) → store 2 3

null → next line

bahar way

lekin again put null in Queue

→ O/P : { 1 / 2 3 / 4 5 6 }

Note :-

Tree

(1) BFS → O(n) → level order

(2) DFS → O(n) → pre, post, in order

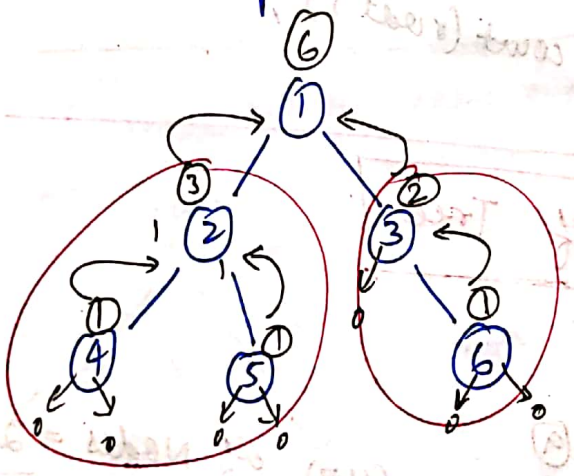
Code :

```
public static void levelOrder(Node root)
{
    if(root == null)
    {
        return;
    }
    // create Queue
    Queue<Node> q = new LinkedList<>();

    q.add(root);
    q.add(null);

    while(!q.isEmpty())
    {
        Node currNode = q.remove();
        if(currNode == null)
        {
            sout("\n");
            if(q.isEmpty())
            {
                break;
            }
        }
        else
        {
            q.add(null);
        }
    }//if

    else
    {
        sout(currNode.data + " ");
        if(currNode.left != null)
        {
            q.add(currNode.left);
        }
```

```
if (curNode.right != null)
{
    q.add(curNode.right);
}
}//else
}//tn.

main()
{
    levelOrder(root);
}
```

Count of Nodes

→ Use of Recursion



count / no. of nodes = 6

→ node = (x + y + 1)

LSC — left subtree (x)

RSC (right subtree count) — right subtree (y)

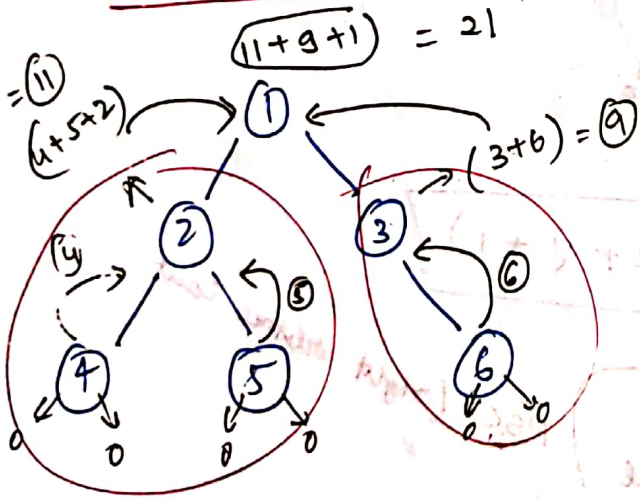
* x = no. of node

* y = no. of nodes

Code :

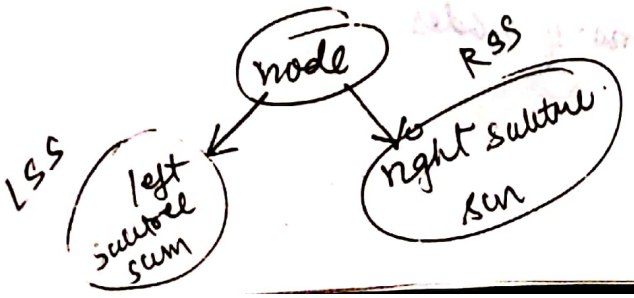
```
public static int count(Node root)
{
    if(root == null)
    {
        return 0;
    }
    int leftNode = count(root.left);
    int rightNode = count(root.right);
    return (leftNode + rightNode + 1);
}

main()
{
    sout(count(root));
}
```

Sum of Nodes of Tree



Sum of Nodes = 21

$T_n = O(n)$



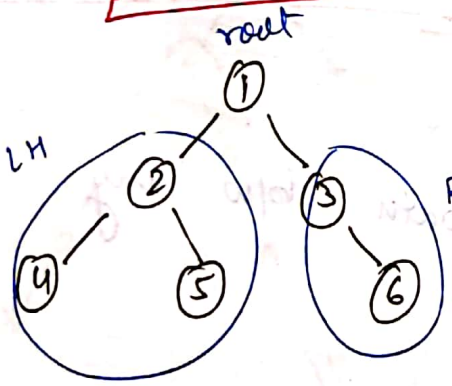
⇒ (LSS + RSS + node.data)

Code :

```
public static int sumOfNodes (Node root)
{
    if (root == null)
    {
        return 0;
    }
    int leftSum = sumOfNodes (root.left);
    int rightSum = sumOfNodes (root.right);
    return (leftSum + rightSum + root.data);
}

main()
{
    sout(sumOfNodes (root));
}
```

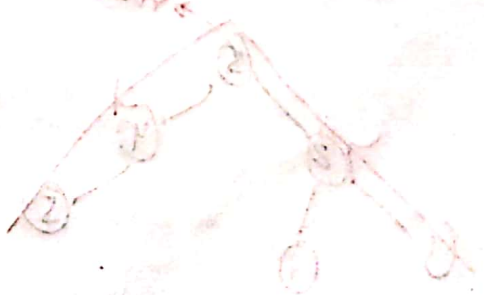
Height of Tree ⟹ Root to deepest leaf

$T_n = O(n)$



height of Tree $= \left[\max(LH, RH) + 1 \right]$

max of these



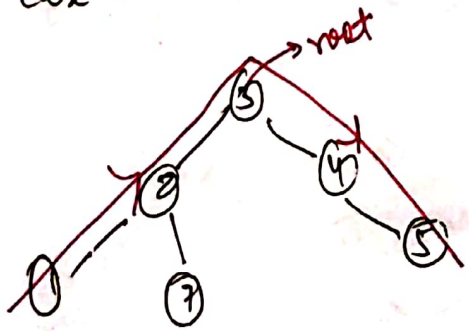
Code:

```
public static int height (Node root)
{
    if (root == null)
    {
        return 0;
    }

    int leftHeight = height (root.left);
    int rightHeight = height (root.right);

    return ( Math.max (leftHeight, rightHeight) + 1 );
}//th

main
{ sout (height (root));
}
```

Diameter of a Tree

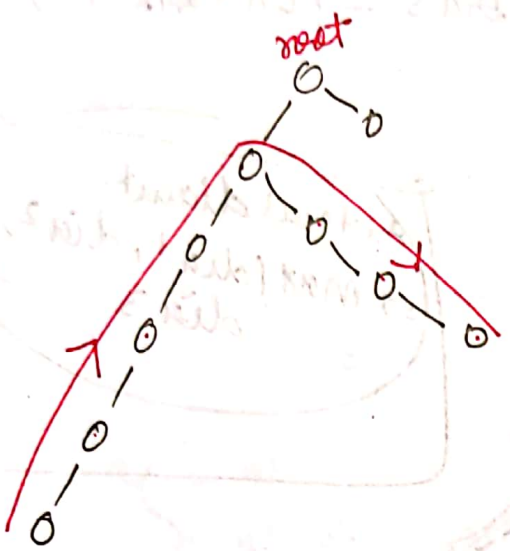
⇒ No. of nodes in longest path b/w any 2 nodes.

Case 1: When diameter passes through Root



diameter = 5

Case 2 :- Diameter (Not) passes through root node



diameter = 8

Approach - 1

$T_n = O(n^2)$

Case 1 :

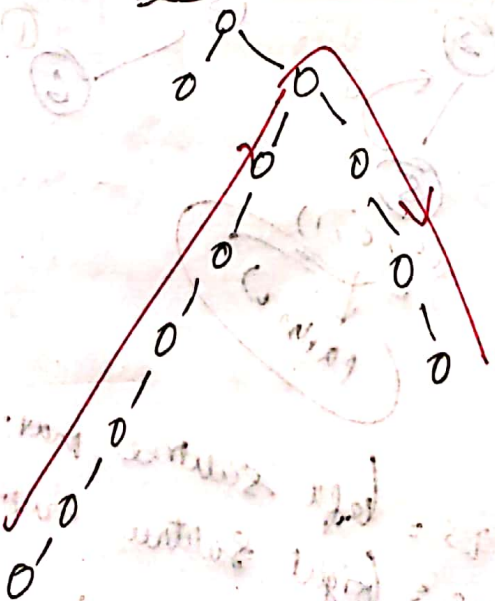
Diameter present → left subtree

Dia 1 = LS

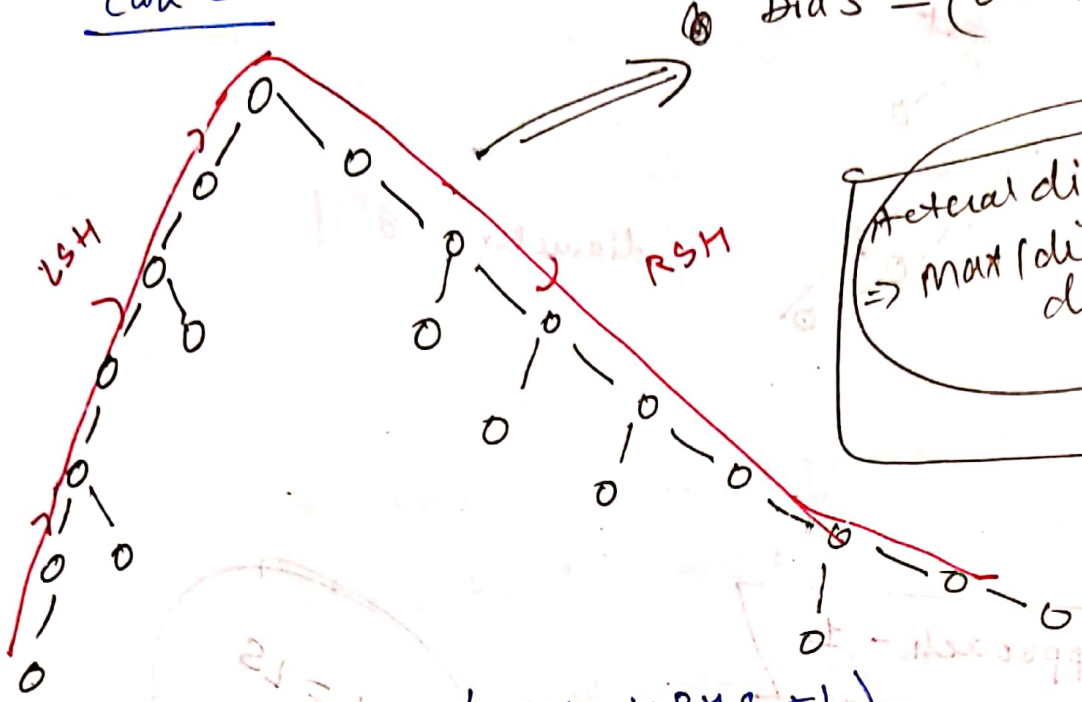
Dia 2 = RS

Case 2 :

Diameter present → right subtree



Case 3:

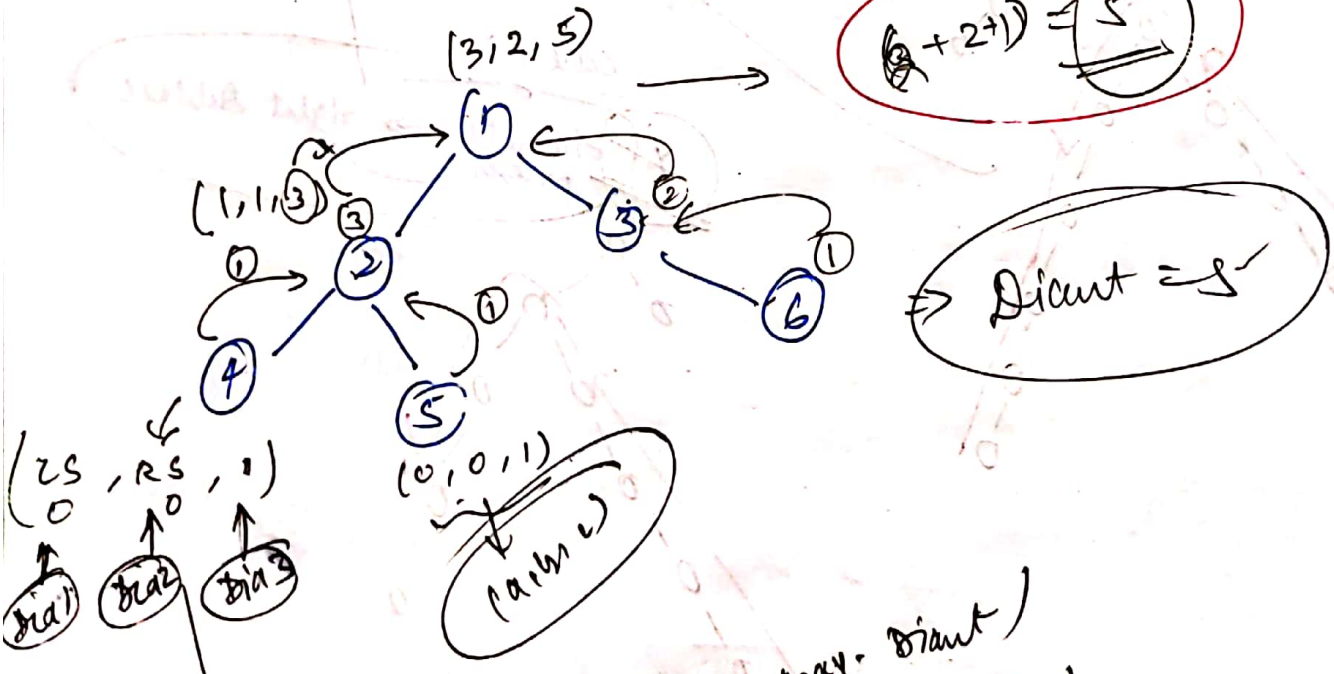


Dia 3 = (LH + RH + 1)

Actual diamt
⇒ max (dia 1, dia 2, dia 3

diamtr = (LSH + RHS + 1)

* max. 3 case possible
⇓
actual diameter of tree

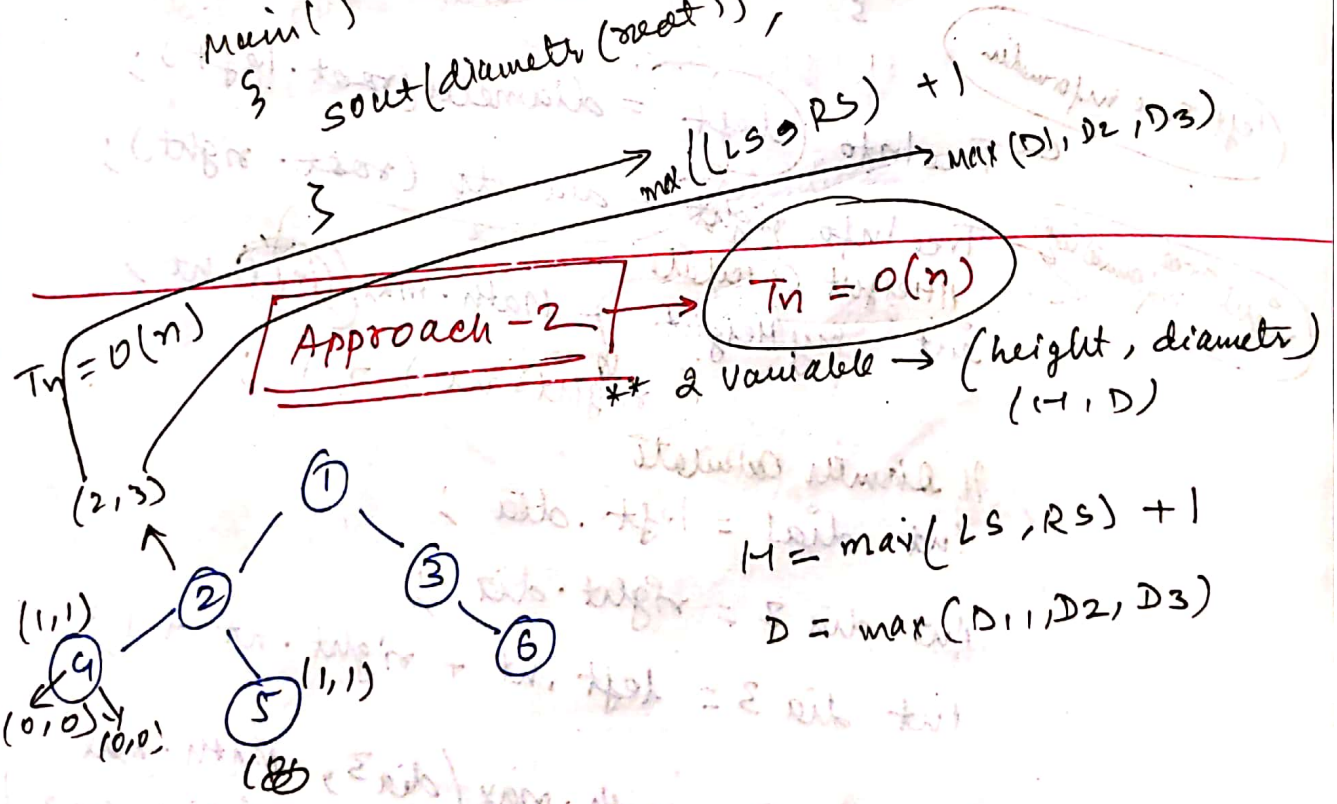


LS = (left subtree max. diamt)
RS = (right subtree max. diamtr)
dia 3 = (LH + RH + 1)

Code :

```
public static int diameter(Node root)
{
    if (root == null)
    {
        return 0;
    }
    a  int dia1 = diameter(root.left);
    b  int dia2 = diameter(root.right);
    c  int dia3 = height(root.left) + height
                  (root.right) + 1;

    return Math.max(dia3, Math.max(dia2, dia3));
}
```

```
main()
{
    sout(diameter(root));
}
```



```
//class
static class TreeInfo
{
    int ht, dia;
    //constructor
    TreeInfo(int ht, int dia)
    {
        this.ht = ht;
        this.dia = dia;
    }
}//class
```

* Create a class where we'll store 2 information
  - height
  - Diameter
* Har node ko apna height & diameter sabse, me return karni hoi

```
//fn
public static TreeInfo diameter(Node root)
{
    // Base case
    if (root == null)
    {
        return new TreeInfo(0,0);
    }

    TreeInfo left = diameter(root.left);
    TreeInfo right = diameter(root.right);
    //Height Calculate
    int myHeight = Math.max(left.ht, right.ht) + 1;

    // Diameter Calculate
    int dia1 = left.dia;
    int dia2 = right.dia;
    int dia3 = left.ht + right.ht + 1;

    int myDia = Math.max(dia3, Math.max(dia1, dia2));
```

left Tree information

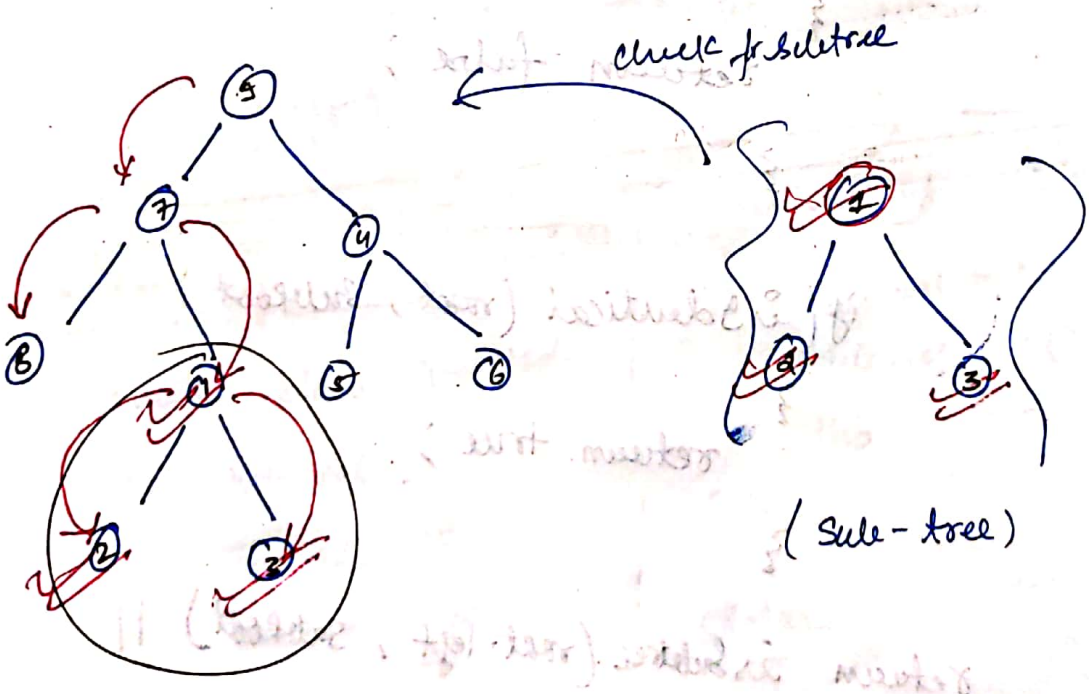
right Tree information

```
TreeInfo myInfo = new TreeInfo(myHeight, myDia);
return myInfo;
}//fn

main()
{
    cout(diameter(root).dia);
}
```

Qus: Subtree of Another Tree

→ sample tree given hoga

→ find out whether our tree is a subtree of that sample tree



Steps :-

① Compare with each node
if match == found, then start comparing
its left & right node.

Code

```
public boolean isSubtree(TreeNode root,
                         TreeNode subRoot)
{
    // subtree → no elemt present.
    if (subRoot == null)
    {
        return true;
    }
    // main tree → empty
    if (root == null)
    {
        return false;
    }
    if(isIdentical(root, subRoot))
    {
        return true;
    }
    return isSubtree(root.left, subRoot) ||
           isSubtree(root.right, subRoot);
}
```

```
public boolean isIdentical(TreeNode root, TreeNode subRoot)
{
    if(root == null && subRoot == null)
    {
        return true;
    }
    if(root == null || subRoot == null)
    {
        return false;
    }
    if(root.val == subRoot.val)
    {
        return isIdentical(root.left, subRoot.left)
            && isIdentical(root.right, subRoot.right);
    }
    return false;
}//fn
```

Given in the Ques:-

```
public class TreeNode
{
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode()
    {
    }

    TreeNode(int val, TreeNode left,
             TreeNode right)
    {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}//class
```

Lecture - 32

Binary Search Tree (B.S.T)

* Binary Tree → $T_n = O(n)$ { n = no. of nodes in tree }

* Binary Search Tree → $T_n = O(H)$ { H = height of tree }

Properties

(1) all that of Binary Tree

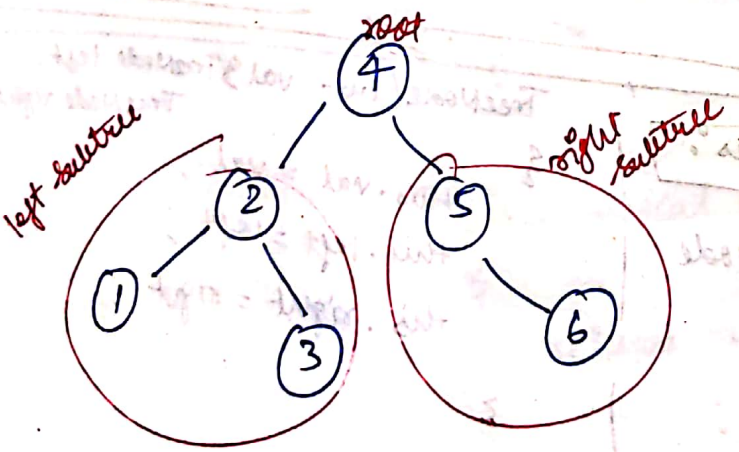
(2) At max 2 children

(3) left subtree node value < Root value

(4) right Subtree node value > Root value

(5) left & right subtree are also B.S.T with no duplicates

(6) No duplication of node value allowed

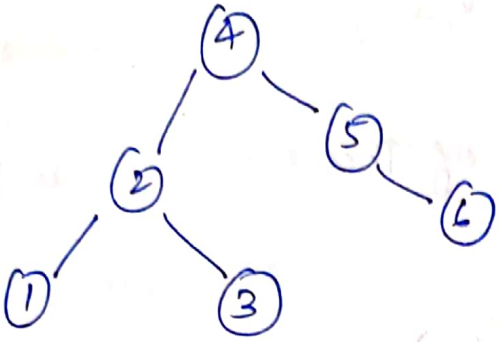


inorder ⇒ [1 2 3 4 5 6]

(6) inorder traversal gives sorted data.

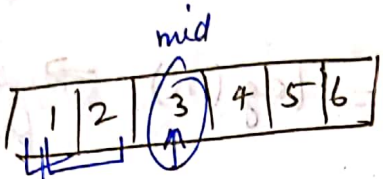
↳ (left → root → right) ⇒ (increasing sorted sequence)

B.S.T - Search



* key = 3 (search) *

Binary Search
↓
sorted array :



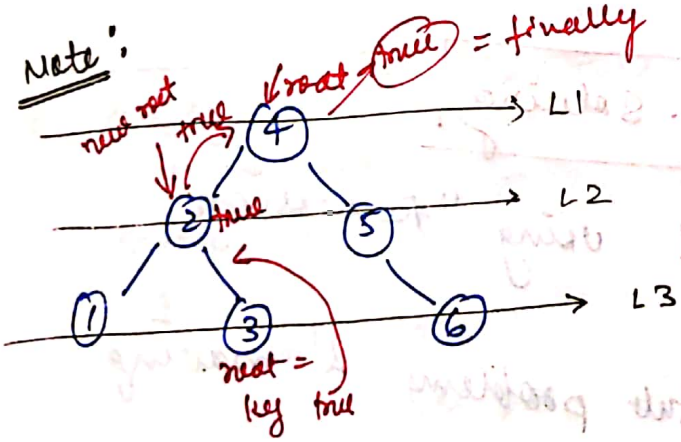
key = 1

key = 1 → return true ;

Tree :

Key (mid) = root value
↓
compare
↙ ↘
left subtree | right subtree

Note :



(key = 3)

* **Worst Case** :→ when key element present at the leaf node

$T_n = O(H)$

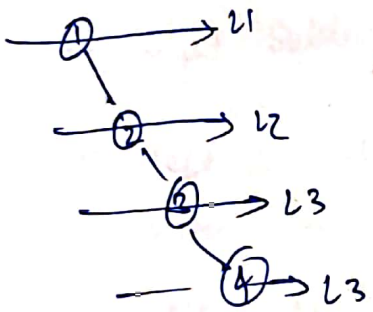
H = Height of Tree

$H = 3$

$\therefore T_n = O(H) = O(3)$

$T_n = O(H) = \log(n)$ ⇒ only applicable for perfectly balanced Tree

* for skewed tree



key = 4

$T_n = O(N)$

$T_n = O(4)$

* B.S.T makes our search efficient.

Strategy for Prob. Solving

* Most problems solved using "Recursion"

↓

by dividing into sub problems & making recursive calls on subtree

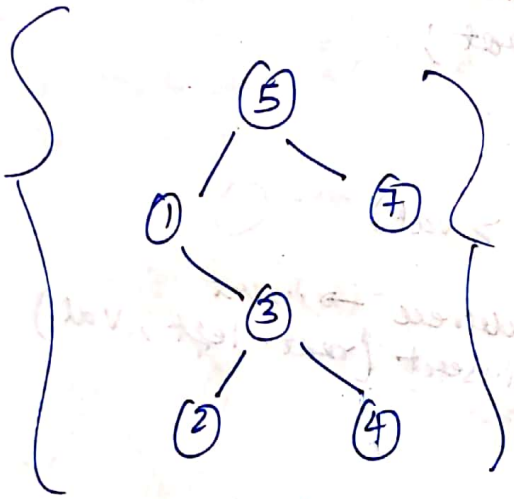
(#) Construct a B.S.T

value E] = {5, 1, 3, 4, 2, 7}

* define a root node (initilize with null value)

```
class Node
{
    int data;
    Node left;
    Node right;
}
```

Root = null
5
compare next value with root → left / → right
comparision to get its position



B.S.T

inorder sequence
⇒ [1 2 3 4 5 7]

Code # ⇒ (Construct Binary Tree)

```
static class Node
{
    int data;
    Node left;
    Node right;

    Node(int data)
    {
        this.data = data;
```

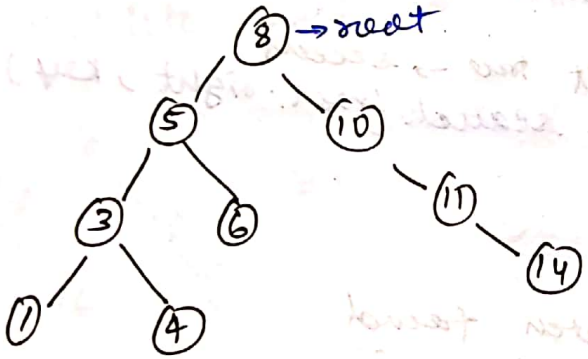
```
left = null;
right = null;

}// constructor

}// class-static

// fn to insert element after comparison   return type → Node
                                           // recursive add value
public static Node insert(Node root, int val)
{
    // base case
    if (root == null)
    {
        root = new Node(val);
        return root;
    }

    if (root.data > val)
    {  // left subtree → insert
    root.left = insert(root.left, val)
    }

    if(root.data < val)
    {  // right subtree → insert
    root.right = insert(root.right, val)
    }

    return root;

}// fn
```

```
main()
{
    int value[] = {5, 1, 3, 4, 2, 7};
    Node root = null;
    for(int i = 0; i < value.length; i++)
    {
        root = insert(root, value[i])
    }
}//main
```

(t1)

* make/check inOrder traversal to verify whether tree is BST or not

Search in a B.S.T



* (key = 1) *

Root > Key → left subtree

Root = Key → found

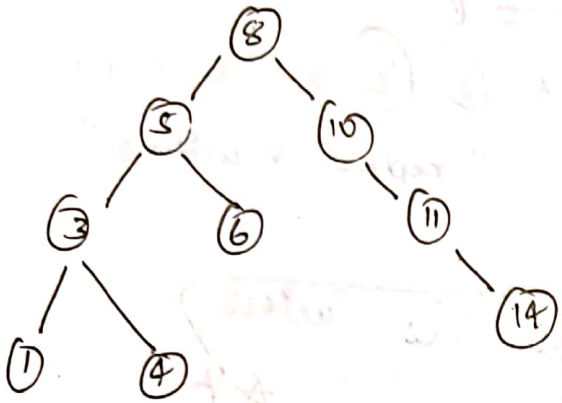
Root < Key → right subtree

∴ $T_n = O(H)$

$= O$ (height of tree)

```
// fn to search element in BST
public static boolean search(Node root, int key)
{
    // Base Case
    // if(root.data > key)
    if (root == null)
    {
        return false;
    }
    if (root.data > key)
    { // left tree -> search
        return search(root.left, key);
    }
    else if (root.data < key)
    { // right tree -> search
        return search(root.right, key);
    }
    else
    { // match found
        return true;
    }
} // fn

main()
{
    t1 [ Construct tree wala part ]

    if(search(root, 1))
    { sout("found");
    }
    else { sout("Not found"); }
} // main
```

Deleting a Node in BST

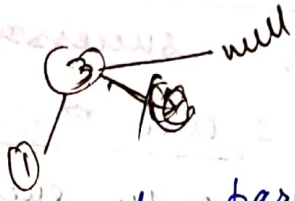


3 case Possible

(i) No child (leaf node)

(ii) 1 child

(iii) 2 child

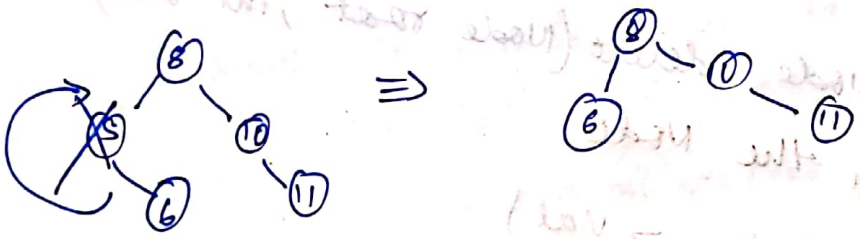


Case I: Leaf Node

⇒ delete node & return null to parent

Case II: 1 child

⇒ delete node & replace with singly child node

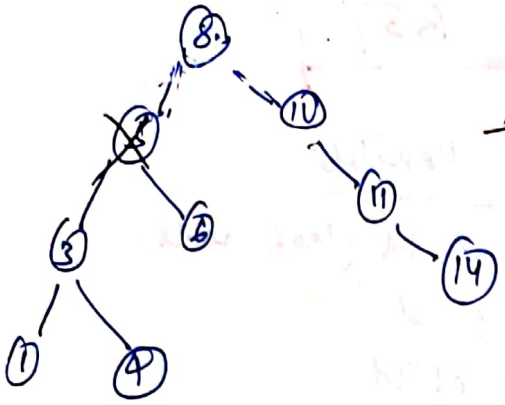


⇒ in java; our garbage collector simply removes unused node

⇒ it will be done automatically

Case III: 2 child wala node

⇒ delete that node & replace value with inorder successor



→ inorder seq: 1 3 4 5 6 8 10 11 14

→ lies in case 1 or case 2

inorder succ of 5 ⇒ 6

"replace 5 with 6"

inorder successor always lies with either 1 child or no child.

inorder successor ko ya to case 1 ya case 2 delete kar dega

How to search for inorder successor

us node ke right subtree me tab tak chalo jab tak usme left most node na mil jaye

BST ⇒ (left most node) in right subtree

Code :-

```
public static Node delete(Node root, int val)
{
    // search the Node
    if (root.data > val)
    {
        // left subtree
        root.left = delete(root.left, val);
    }
    else if (root.data < val)
    {
        // right subtree
        root.right = delete(root.right, val);
    }
    else // root.data == val
    {
```

```
// case 1 : leaf node
if (root.left == null && root.right == null)
{
    return null;
}

// case 2 : 1 child -> node
if (root.left == null) -> root -> right exist
{
    return root.right;
}
else if (root.right == null)
{
    return root.left;
}

// case 3 : 2 child -> node
Node is = inOrderSucc (root.right) ; // get inorder succ.
root.data = is.data ; // root data = inorder data
root.right = delete (root.right, is.data);
} // else - block
return root;

} // fn
```

delete inorder succ

uses : case 1 or case 2

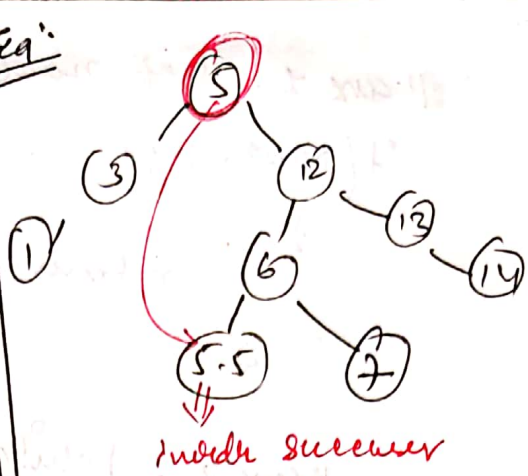
```
// inorder successor -> calculate
public static Node inOrderSucc (Node root)
{
    while (root.left != null)
    {
        root = root.left;
    }
    return root;
}
```

```
Main()
{
```

[t]

```
delete (root, 4);
inOrder(root);

}//main()
```

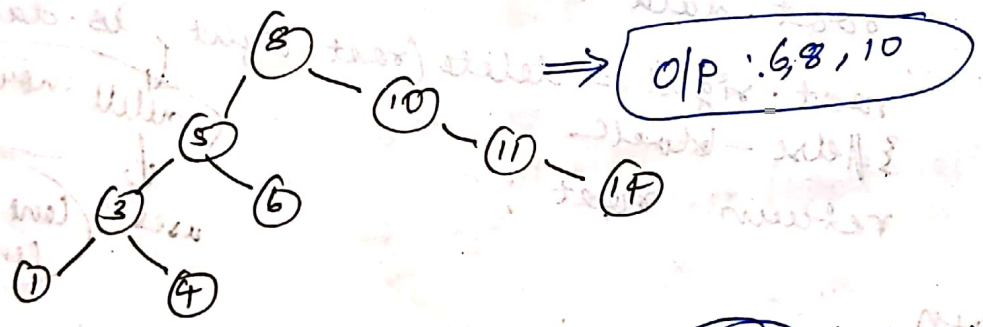
Eg:



(#) **Print in Range**

x = 6 & y = 10 (including range)

=> print all NO => [6, 10] in b/w range of 6 to 10 which are present in your B.S.T.



=> for each root compare (x & y) -> range

$$x \leq root \leq y$$

⇓

print => (root.data)

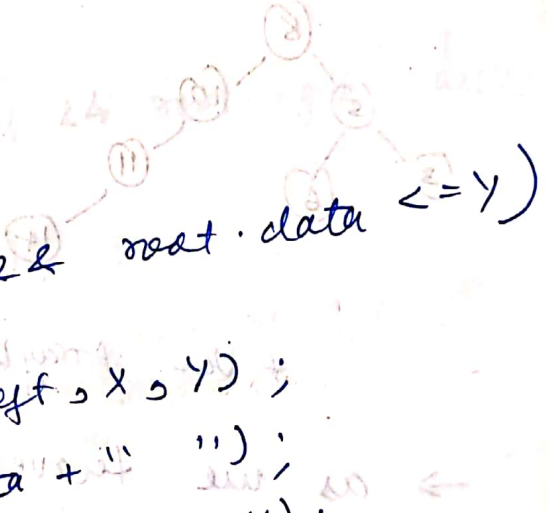
(1) $x \le root \le y$
- → left
- → right

(3) $y < root$
↳ left

(2) $x > root$
↳ right

Code:

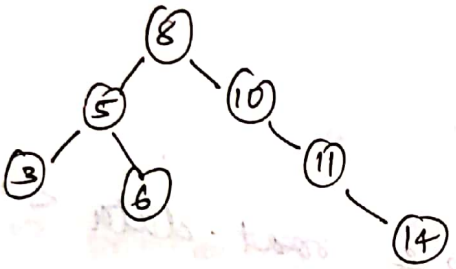
```
public static void printRange(Node root,
                              int x, int y)
{
    //base case
    if (root == null)
    {
        return;
    }
    if (root.data >= x && root.data <= y)
    {
        printRange(root.left, x, y);
        sout(root.data + " ");
        printRange(root.right, x, y);
    }
    else if (root.data >= y)
    {
        printRange(root.left, x, y);
    }
    else
    {
        printRange(root.right, x, y);
    }
}//fn
```



main()
{
[I]
printRange(root, 6, 10);
}//main

Root to Leaf Paths

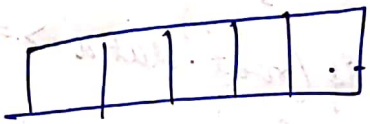
* Print all those path which start from root and points towards leaf.



O/P:
i) 8-5-3
ii) 8-5-6
iii) 8-10-11-14

3 Poss. Path

* Use ArrayList



⇒ path

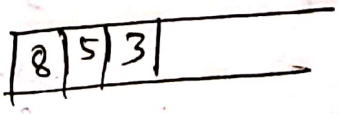
→ as we traverse, store our value in arraylist.

⇒ PreOrder Traversal type algo

* Root (add)
* Left Subtree traverse
* Right Subtree traverse
* Remove (root)

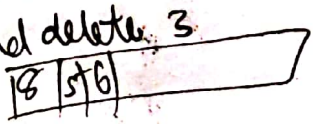
Algo:

① * as we traverse, we keep on adding it to our ArrayList



② * leaf node pe pahuch kar path ko print karo do

③ * Back track karo and delete 3



....

```
public static void printRootLeaf (Node root,
                                  ArrayList<Integer> path)
{
    if(root == null)
    {
        return;
    }
    path.add(root.data);
    //leaf node
    if(root.left == null && root.right == null)
    {
        printPath(path);
    }
    else //non-leaf node
    {
        printRootLeaf(root.left, path);
        printRootLeaf(root.right, path);
    }
    path.remove(path.size()-1);
}//fn
```

last index of our ArrayList
// to delete last element present in our ArrayList

```
//fn to print path
public static void printPath(ArrayList<Integer> path)
{
    for(int i=0; i<path.size(); i++)
    {
        sout(path.get(i)+" ");
    }
    sout("\n");
}//fn

//main
main()
{
```


```
    printRootLeaf(root, new ArrayList<>());
}//main
```