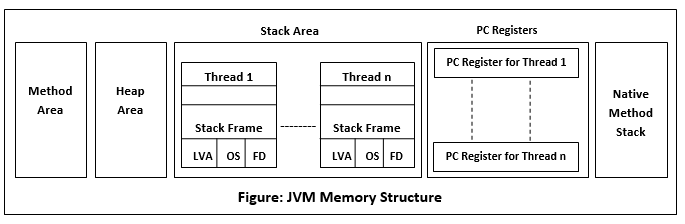
**Data Structures – Foundation**

**Motivation:** How can you be a better coder, if you don’t know how good your code is? 😊

**Memory Areas:** The below diagram describes the different memory areas in **Java**:



Important points to remember:

**Method Area:** Global variables are stored in Method Area.

**Heap Area:** Objects are stored in Heap.

**Stack Area:** Local variables are stored in Stack.

**Native method stack:** Stack stores the data of the methods in non-java language. ( also called as C-Stack)

**PC Register:**

* For non – native methods: JVM thread has a program counter (PC) associated with it. PC stores the available JVM instructions.
* For Native methods: PC value is undefined. PC Register stores the return address of the native pointer.

**Simple code:**

**public** **class** Employee {

**int** empId;

}

**public** **class** Test {

**static** String *Company*;

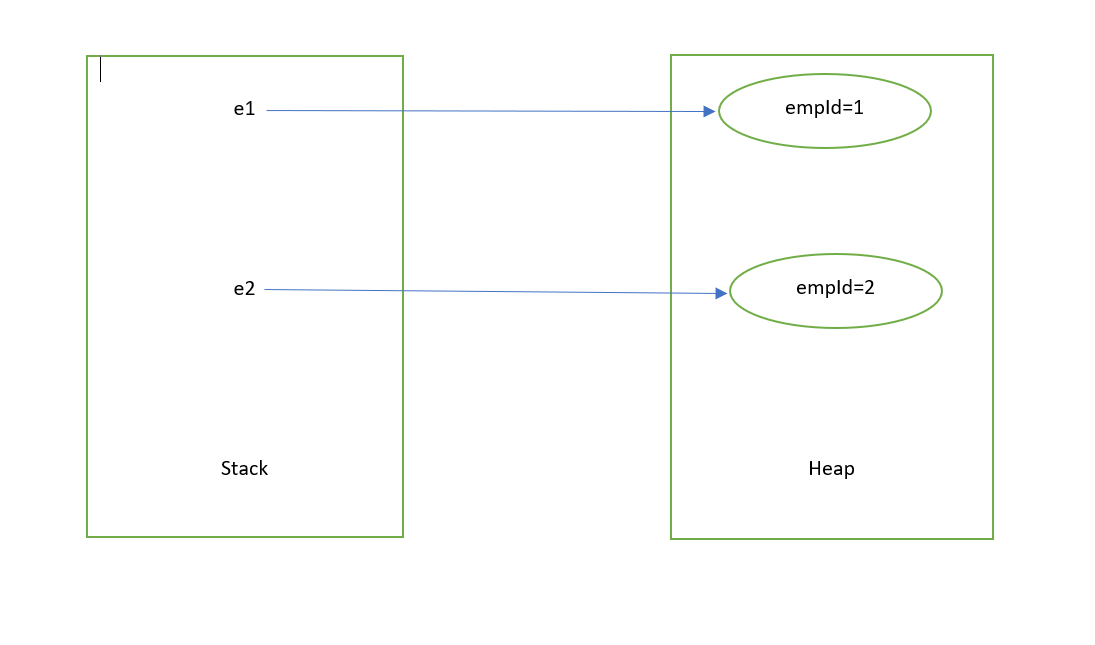
**public** **static** **void** main(String[] args) {

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

}

}



Tweaking the above code to see the change in the memory organization:

**public** **class** Test {

**static** String *Company*;

**public** **static** **void** main(String[] args) {

Employee e1 = **new** Employee();

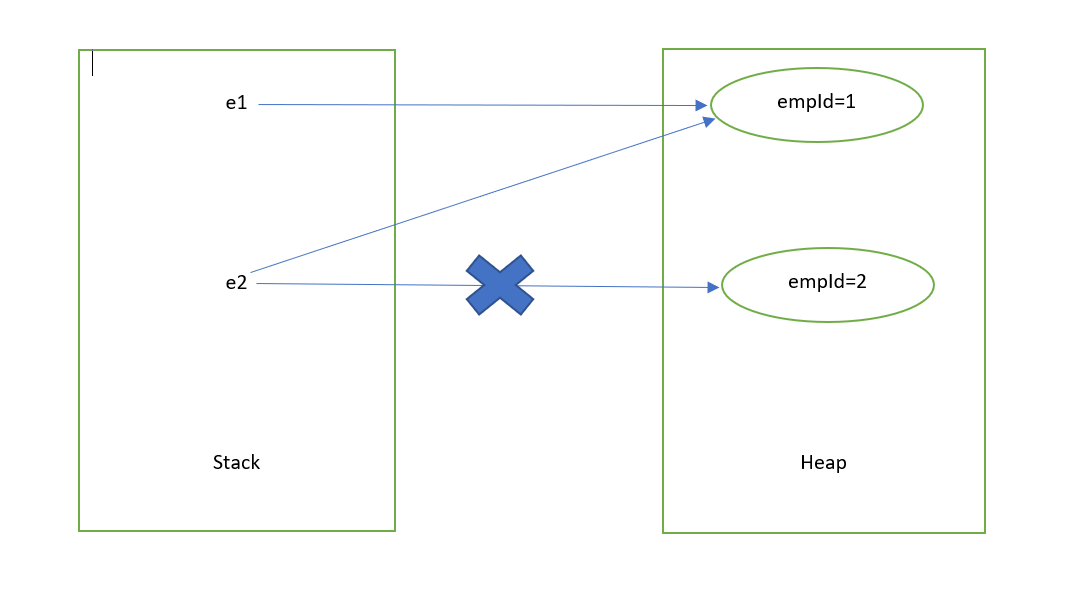
Employee e2 = **new** Employee();

e1.empId = 10;

e2 = e1;

}

}



**Important points to remember:**

1. For every method call, there will be an activation records that gets created.
2. Activation record gets destroyed, once the method call gets completed
3. The stack gets destroyed when the associated thread gets get destroyed.

**Sample code:**

**public** **class** MethodCallExample {

**public** **static** **void** main(String[] args) {

*m1*();

}

**private** **static** **void** m1() {

**int** a;

*m2*();

}

**private** **static** **void** m2() {

**int** b;

*m3*();

*m4*();

}

**private** **static** **void** m3() {

**int** c;

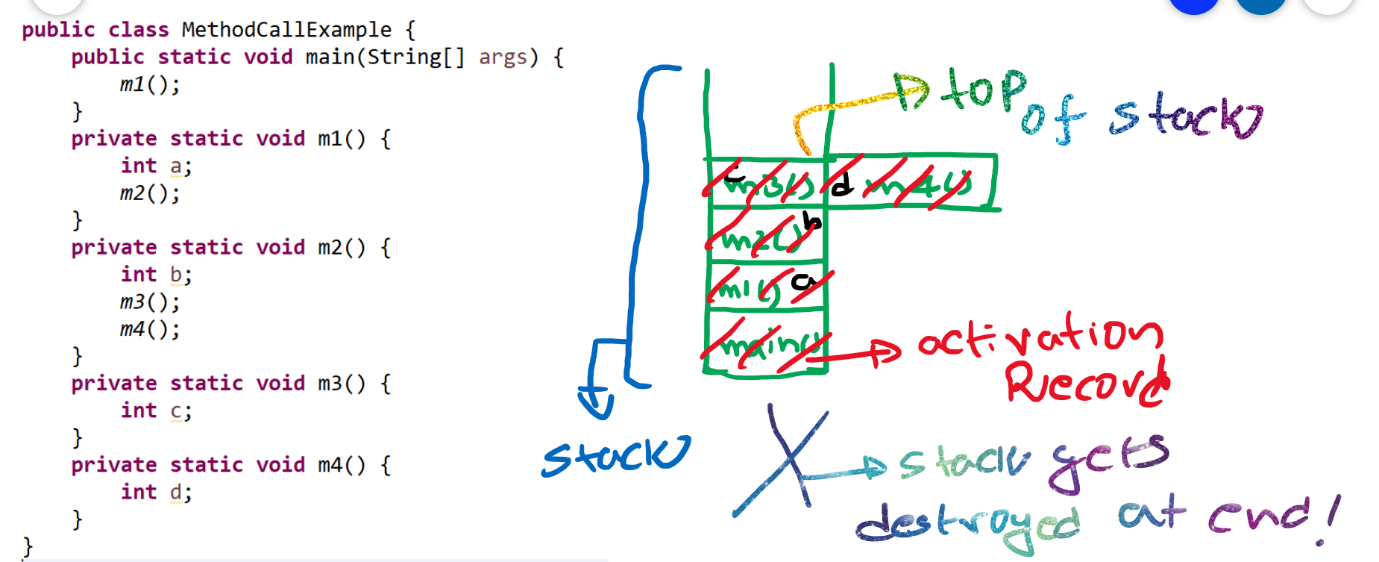
}

**private** **static** **void** m4() {

**int** d;

}

}



What are the different types of Data Structures to learn?

1. LinkedList
2. Stacks
3. Queues
4. Trees
5. Graphs

Logical DS

Physical DS

1. Stacks
2. Queues
3. Trees
4. Graphs
5. Arrays
6. Linked Lists

Data Structures

**What is Data Structure?**

It is nothing but the way the data is organized inside the memory. For example, in a linked list, the data is organized in form of nodes that are interconnected with each other.

**Real time usages of various Data Structures: -**

* **LinkedList**: - Music player.
* **Stack**: - Recently opened web pages inside a browser.
* **Queue**: - Job scheduling in Operating systems.
* **Trees**: - Folder structures in Operating Systems.
* **Graphs**: - Google Maps.
* **Trie**:- Red lines in a editor.

**What is an Algorithm?**

An Algorithm is nothing but a step by step process of achieving something.

**Example: -**

How to make a phone call?

1. Get your phone
2. Unlock it
3. Search for a contact
   1. If the contact exists?
      1. Do you have balance?
         1. if yes, make a call.
         2. If no, terminate?
   2. If contact does not exit
      1. Do you have the number to call?
         1. If no, terminate.
         2. If yes, make a call.
4. So on….

**Example: -** add two numbers

* Take two numbers in two variables, say a,b.
* Take a new variable sum.
* Add a,b and assign the value to sum.
* Return sum.

**private** **int** sum(**int** a, **int** b) {

**int** sum = a+b;

**return** sum;

}

There are different ways to achieve something. Like, if you want to travel from one place to other, you can do that using different ways, like you take a flight, train, bus, car or even you can walk. Every approach has its own pros and cons. Like taking a flight is faster but not cost efficient, and this may not be applicable every time. You cannot take a flight to travel to next street.

**How to compare Algorithms?**

Algorithm comparison should be independent of programming language and external factors like hardware. Ideal way to compare two algorithms is to compare their growth rate.

Say, you need to send a file to your friend. How can you do that?

1. By sending an email or via FTP.
2. By taking that file in a hard drive and by travelling by yourself to you friend and handing it over.

In case 2, the time would be constant and it is irrespective of the file size. Like even if it is 1TB file, the travel time taken is the same.

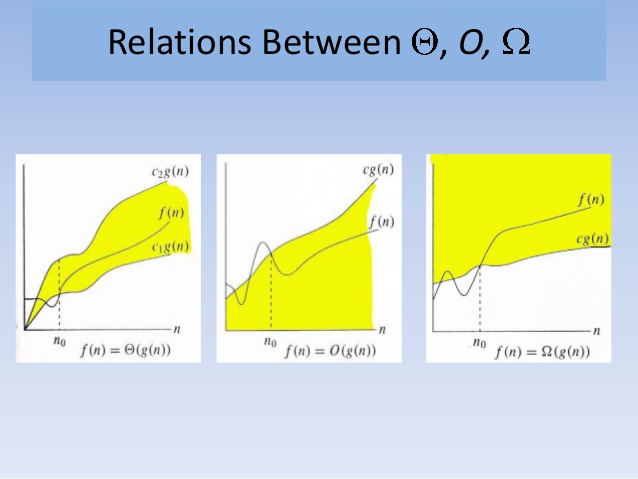
But, in case 1, if it’s a small text file, it is sent over internet in no time. But, if the file is larger, say 1 TB, it takes days to transfer that file electronically. So the growth rate depends on the input size of your file.

Case 2 has constant Time Complexity. We denote it using O(1).

**Types of Analysis: -**

1. **Best Case** − Minimum time required for program execution. **Ω (Big-Omega)**
2. **Average Case** − Average time required for program execution. **ϴ (Theta)**
3. **Worst Case** − Maximum time required for program execution. **Ο (Big-O)**

**Diagrammatic notation: -**



An algorithm can be represented in form of an expression.

**Example: -**

f(n)=3n+2

3n+2<=4n for all n>=3

Hence 3n+2 = O(n)

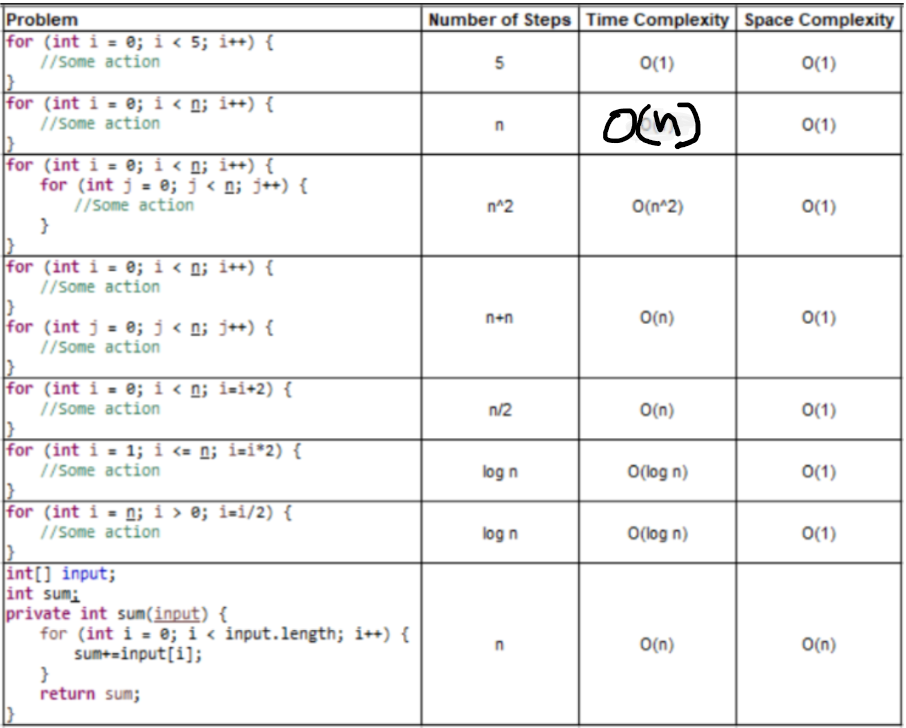
Say, you went to buy a pair of Shoes and Socks. Your friend came and asked you what you are buying. You simply say that you came to buy pair of shoes. You simply ignore socks, even if you had already bought them. The reason why you ignore is that the value of socks is negligible in comparison to that of shoes.

Similarly,

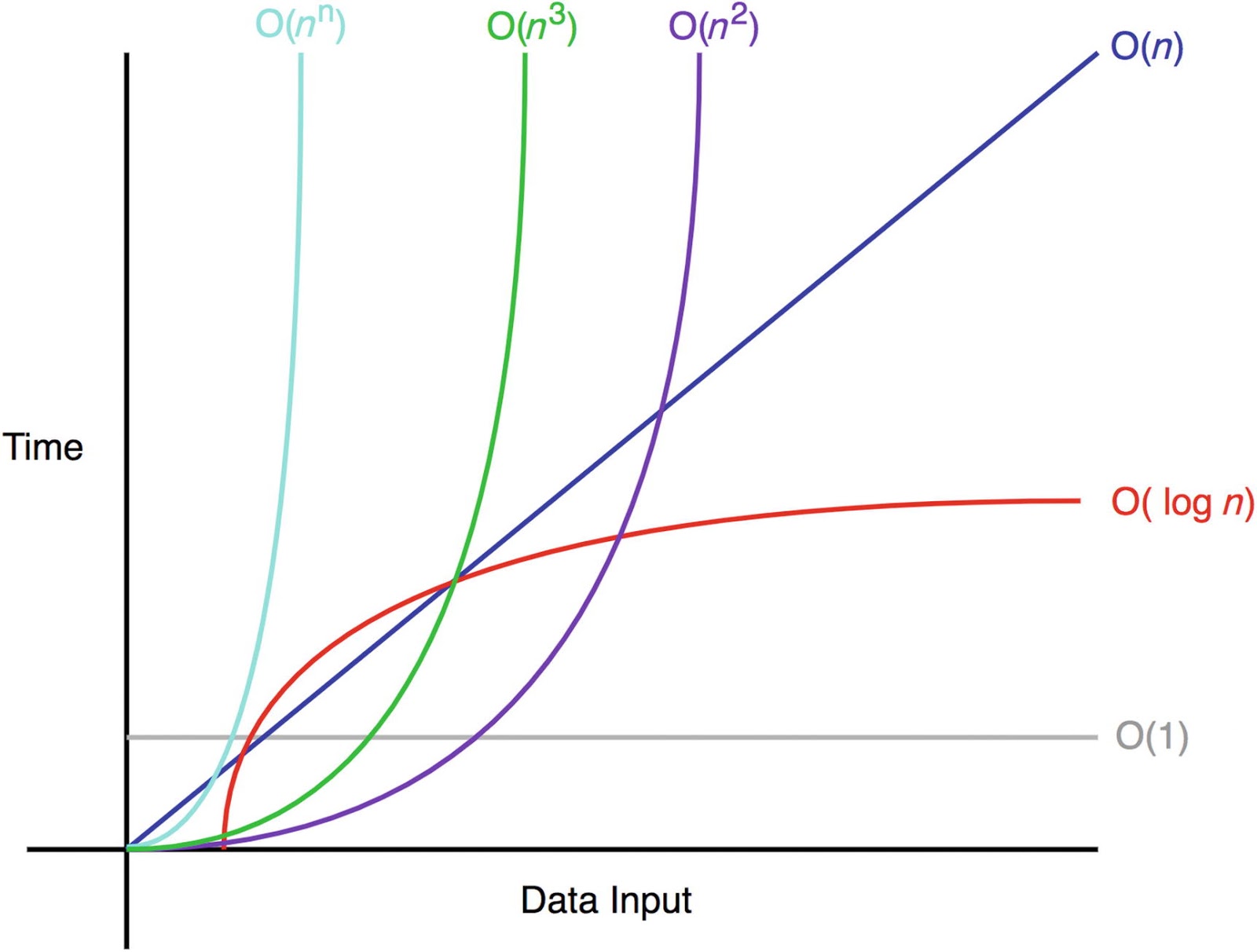
in **f(n)=3n+2**, we ignore 3 and 2 as they are constants the time complexity becomes **O(n)**

If, **f(n)=3n^2+2n+3**, we ignore all and the time complexity becomes **O(n^2)**

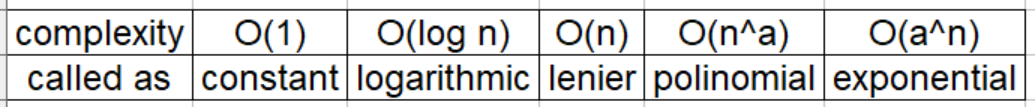
We only concentrate on Big-O notations as we analyse worst cases. There is no point of analysing best case because, most algorithms will be O(1) in the best case.



Graphical representation of various runtime complexities:



**Complexities are also known as :-**

c

**Few log formula that will become handy :**

* **loga 1 = 0**
* **loga a = 1**
* **loga(x\*y) = logax + logay**
* **loga(x/y)  = logax - logay**
* **loga xp = p logax**

Log in Mathematics is generally to base 10. But, in Computer Science, log is to base 2, by default.

We can remember the log in the below simple manner. Since the base is 2, we get the log value of a particular number by calculating the number of steps required bring the number to 1, by dividing it by 2.

Let’s take an example:

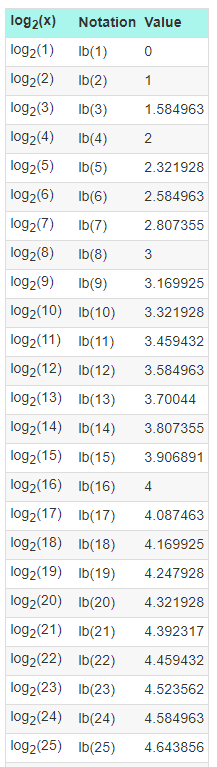
Say input is 8. We can divide 8 with 2, for 3 times to make its value to be 1.

1. 8/2=4
2. 4/2=2
3. 2/2=1

Now, say input is 16, we need 4 steps.

1. 16/2=8
2. 8/2=4
3. 4/2=2
4. 2/2=1

Log values for first 25 numbers:



**Linear Search vs Binary Search:**

**Linear search:** Searching for an element, one element at a time without skipping any item.

**Binary Search:** Cut down your search to half as soon as you find middle of a sorted list.

**public** **class** LinearSearch {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 10, 20, 30, 40, 50 };

**boolean** result = *linearSearch*(arr, 166);

System.***out***.println(result);

}

**static** **boolean** linearSearch(**int**[] arr, **int** x) {

**for** (**int** i = 0; i < arr.length; i++) {

**if** (arr[i] == x) {

**return** **true**;

}

}

**return** **false**;

}

}

**public** **class** BinarySearch {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 10, 20, 30, 40, 50 };

**boolean** result = *binarySearch*(arr, 50, 0, arr.length - 1);

System.***out***.println(result);

}

**private** **static** **boolean** binarySearch(**int**[] arr, **int** x, **int** low, **int** high) {

**while** (low <= high) {

**int** mid = (low + high) / 2;

**if** (arr[mid] == x) {

**return** **true**;

} **else** **if** (arr[mid] < x) {

low = mid + 1;

} **else** **if** (arr[mid] > x) {

high = mid - 1;

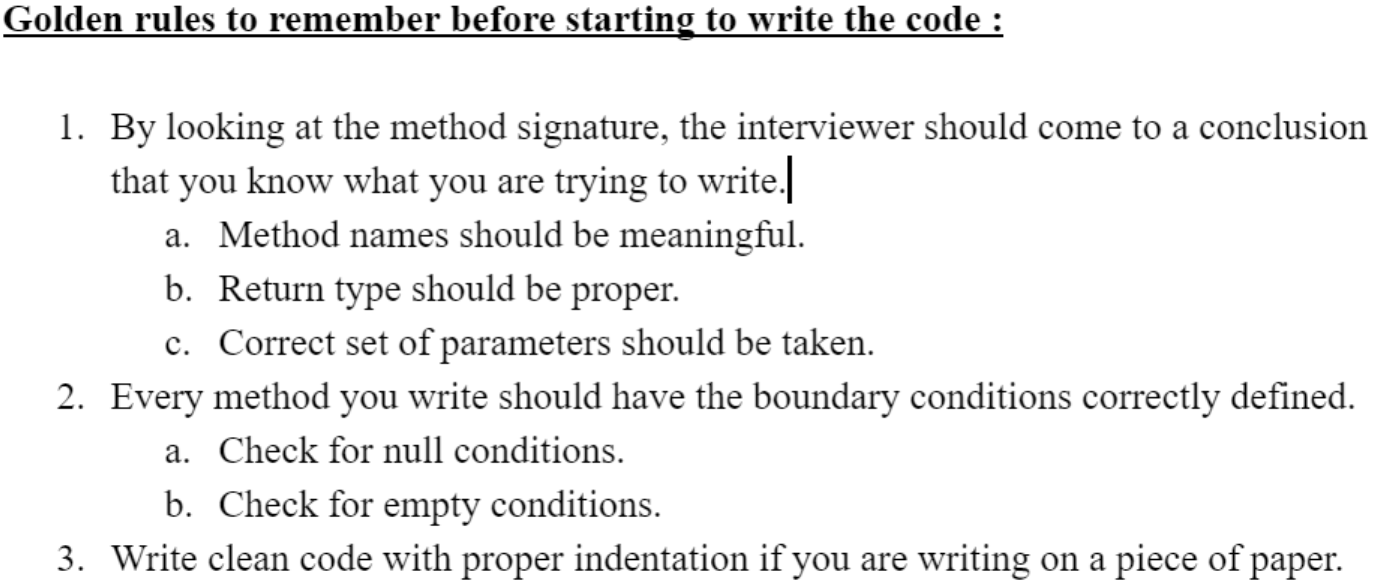
}

}

**return** **false**;

}

}



**Example code following above rules:**

**public** **class** Employee {

**int** empId;

String empName;

**public** Employee(**int** empId, String empName) {

**super**();

**this**.empId = empId;

**this**.empName = empName;

}

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getEmpName() {

**return** empName;

}

**public** **void** setEmpName(String empName) {

**this**.empName = empName;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + empId;

result = prime \* result + ((empName == **null**) ? 0 : empName.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**if** (empId != other.empId)

**return** **false**;

**if** (empName == **null**) {

**if** (other.empName != **null**)

**return** **false**;

} **else** **if** (!empName.equals(other.empName))

**return** **false**;

**return** **true**;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", empName=" + empName + "]";

}

}

**public** **class** EmployeeTest {

**public** **static** Employee changeName(Employee emp, String name) {

// Base condition 1

**if** (emp == **null**) {

**return** **null**;

}

// Base condition 2

**if** (name == **null**) {

**return** emp;

}

// Base condition 3

**if** (name.isEmpty()) {

emp.setEmpName("Empty");

**return** emp;

}

// Actual logic.

emp.setEmpName(name);

**return** emp;

}

**public** **static** **void** main(String[] args) {

Employee e1 = **new** Employee(1, "");

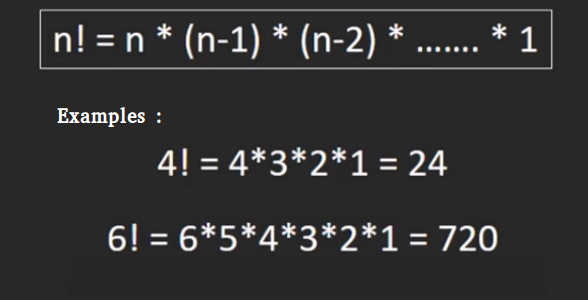
Employee changedEmployee = *changeName*(e1, "");

System.***out***.println(changedEmployee);

}

}

**Factorial Implementation: -**



**public** **class** FactorialIterative {

**public** **static** **void** main(String[] args) {

System.***out***.println(*fact*(6));

}

**static** **int** fact(**int** n) {

**if** (n == 0 || n == 1) {

**return** 1;

}

**int** result = 1;

**for** (**int** i = n; i >= 2; i--) {

result \*= i;

}

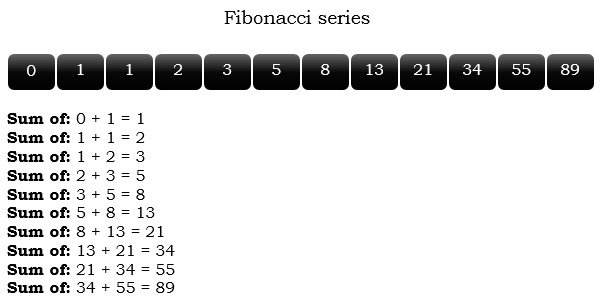
**return** result;

}

}

**Time Complexity: O(n) : Space Complexity : O(1)**

**Fibonacci Series implementation: -**

**public** **class** FibonacciSeriesIterative {

**public** **static** **int** fib(**int** n) {

**int** a = 0;

**int** b = 1;

**int** c = 1;

System.***out***.print(a + "," + b);

**for** (**int** i = 1; i <= n; i++) {// Iteration starts form 1 and not 0.

a = b;

b = c;

c = a + b;

System.***out***.print("," + c);

}

**return** c;

}

**public** **static** **void** main(String[] args) {

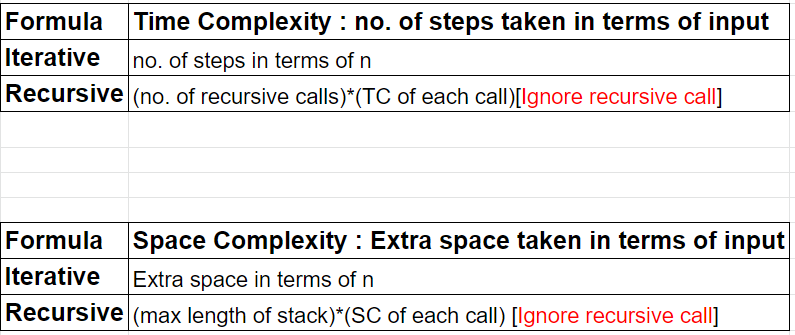
*fib*(7);

}

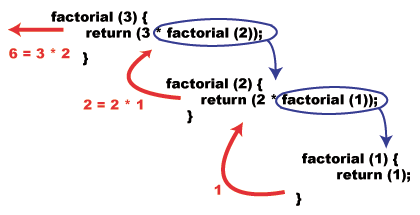
}

**Time Complexity: O(n) : Space Complexity : O(1)**

**Formulae: -**



**Factorial Using Recursion: -**



**public** **class** FactorialRecursive {

**public** **static** **void** main(String[] args) {

System.***out***.println(*fact*(5));

}

**static** **int** fact(**int** n) {

**if** (n == 0 || n == 1) {

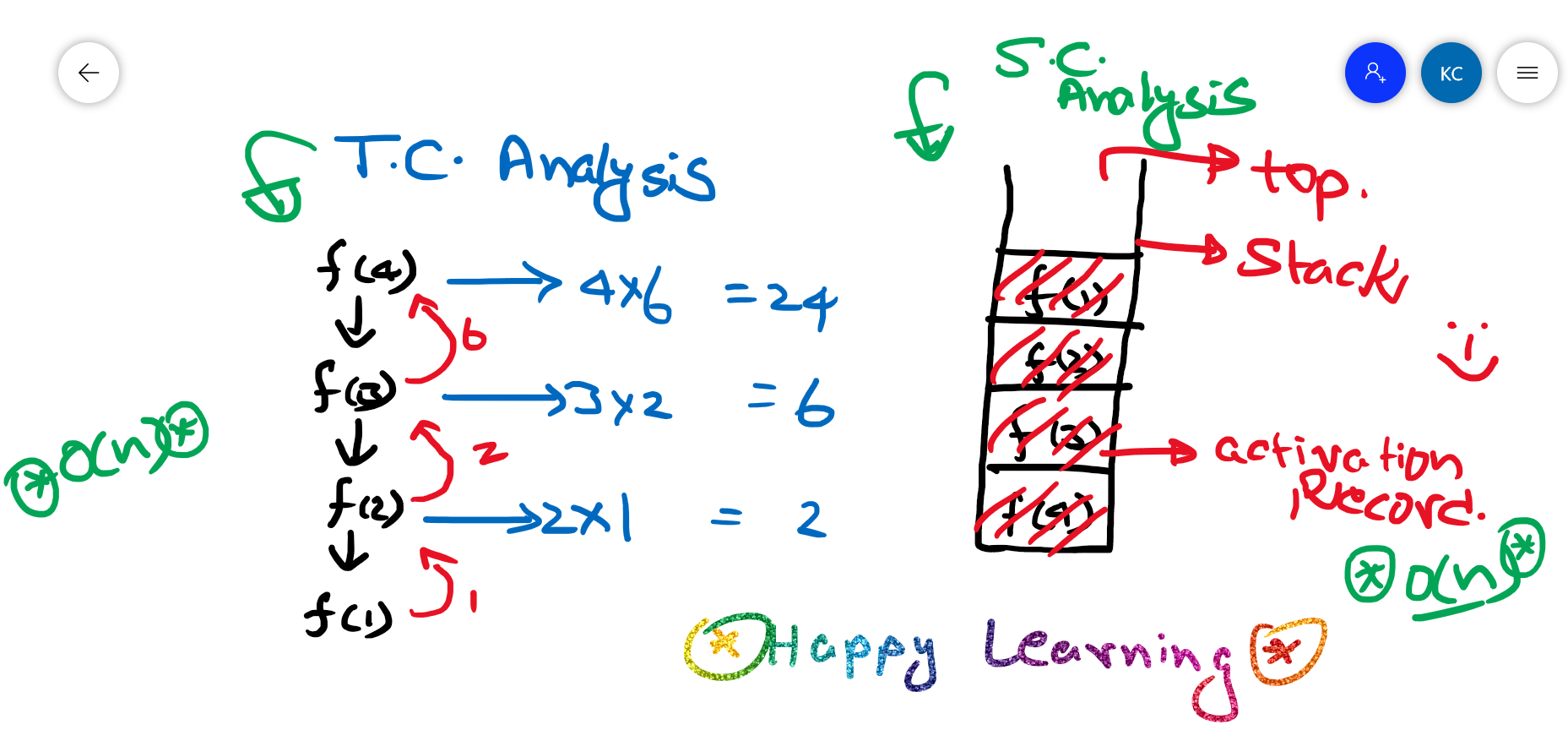
**return** 1;

}

**return** n \* *fact*(n - 1);

}

}



**public** **class** FibonacciSeriesRecursive {

**public** **static** **int** fibonacci(**int** n) {

**if** (n <= 0) {

**return** 0;

}

**if** (n == 1) {

**return** 1;

}

**return** *fibonacci*(n - 1) + *fibonacci*(n - 2);

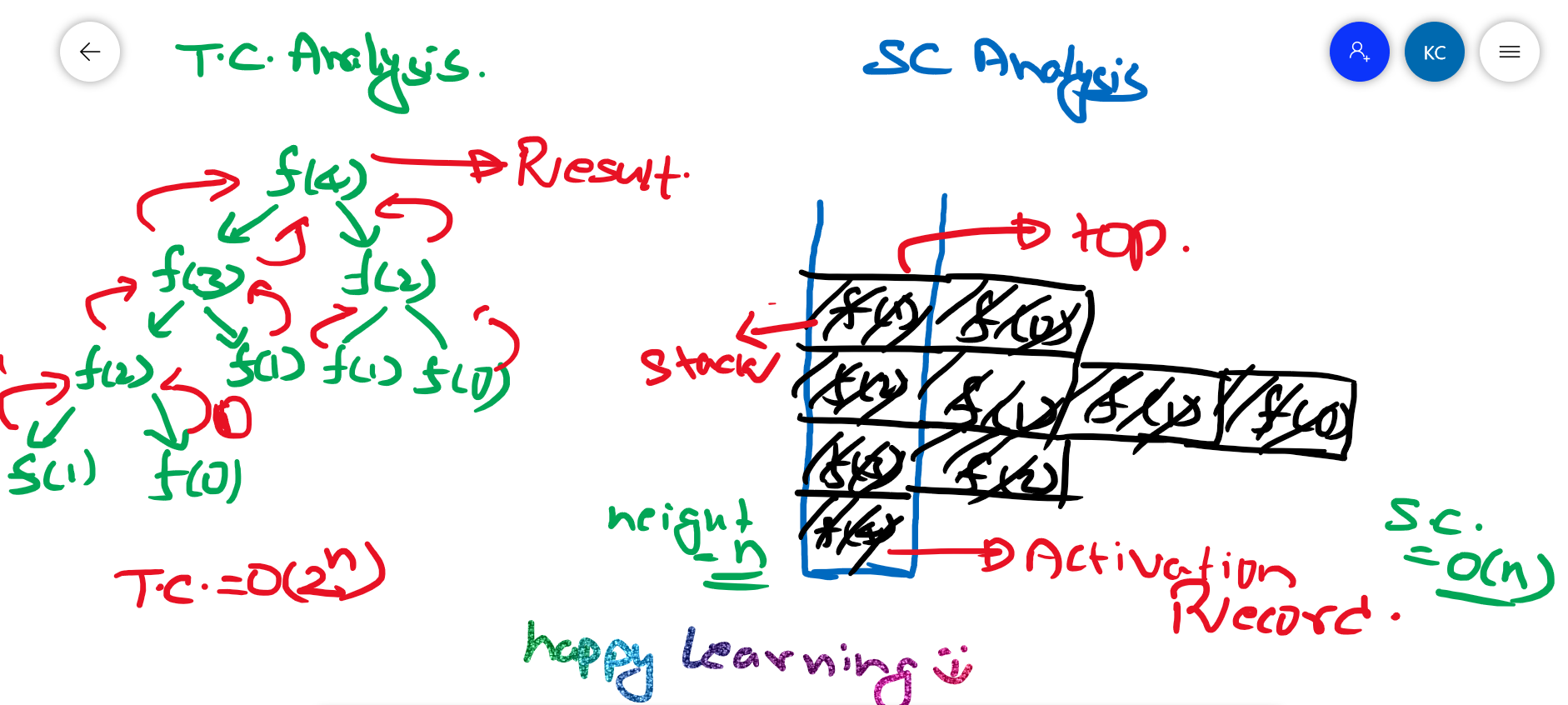
}

**public** **static** **void** main(String[] args) {

System.***out***.println(*fibonacci*(4));

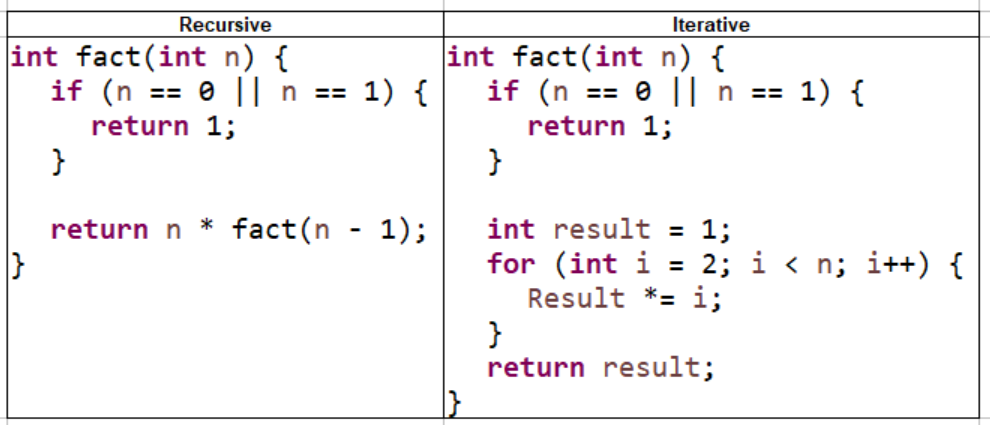
}

}

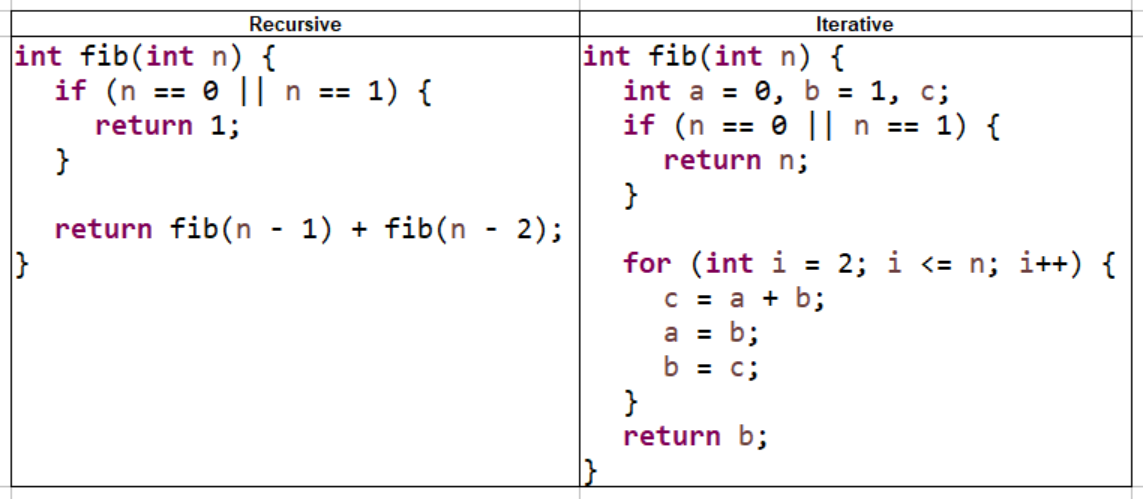


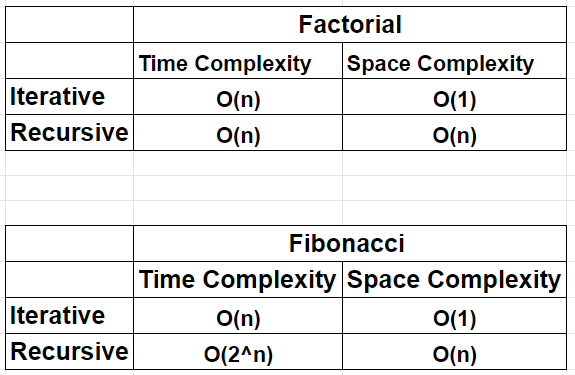
**Comparison: -**

**Program for Factorial:-**

****

**Program for Fibonacci series: -**

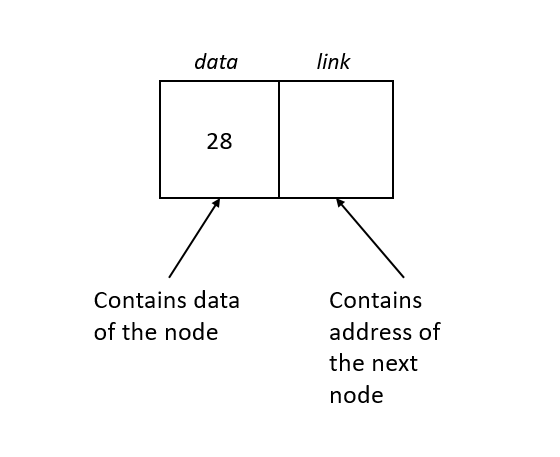
****



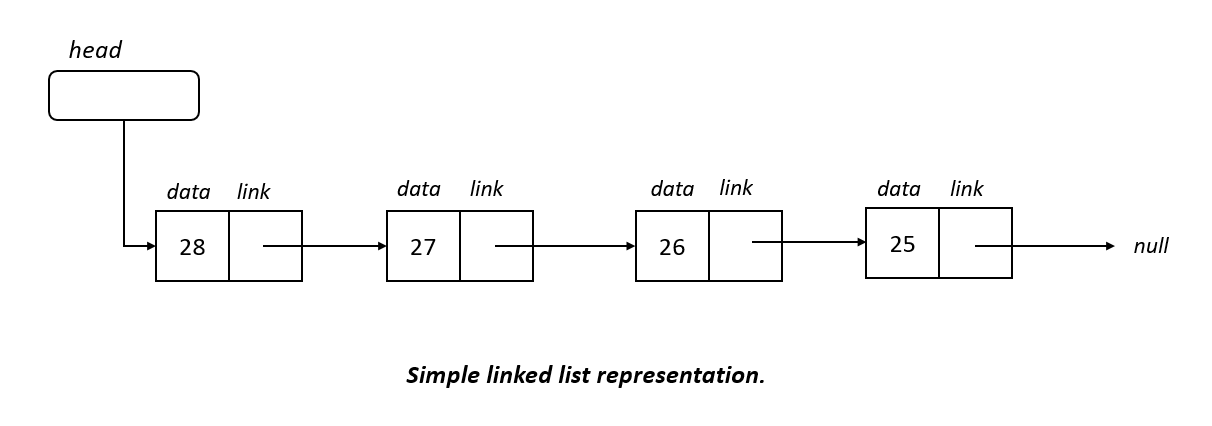
**Linked List**

1. It’s a Physical Data Structure.
2. It is represented in form of Node.
   1. Every Node has two parts
      1. Data
      2. Pointer to next Node
   2. Node is represented as an Object.
3. Address of the first node is stored in the head. So, if the pointer to head is lost, we lose access to the LinkedList. So, we should be very careful while coding.

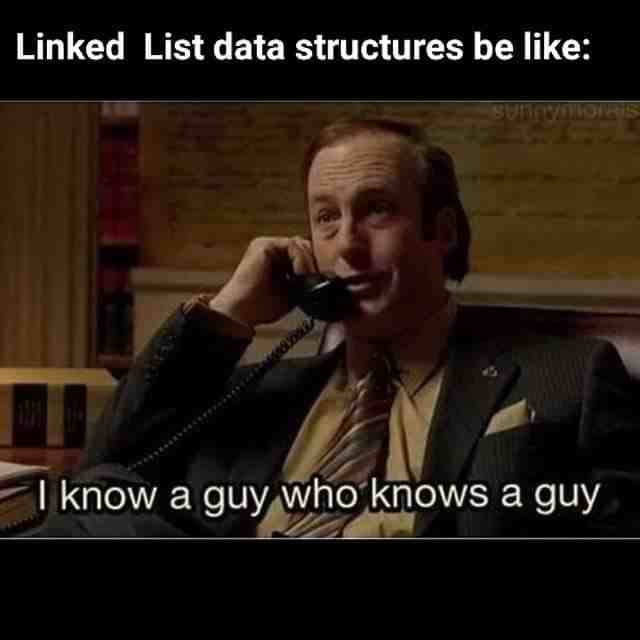
**Below is the representation of a single Node: -**



**Below is simple representation of a Linked List: -**

****

**Just a Meme: -** 😊



**public** **class** Node {

**public** **int** data;

**public** Node next;

Node(){

}

**public** Node(**int** data) {

**this**.data = data;

}

}

**public** **class** LinkedListTraversal {

**private** **void** LLTraversal(Node head) {

**if** (head == **null**) // Base condition

**return**;

Node p = head; // Never do any direct operation on head

**while** (p != **null**) {

System.***out***.println(p.data);

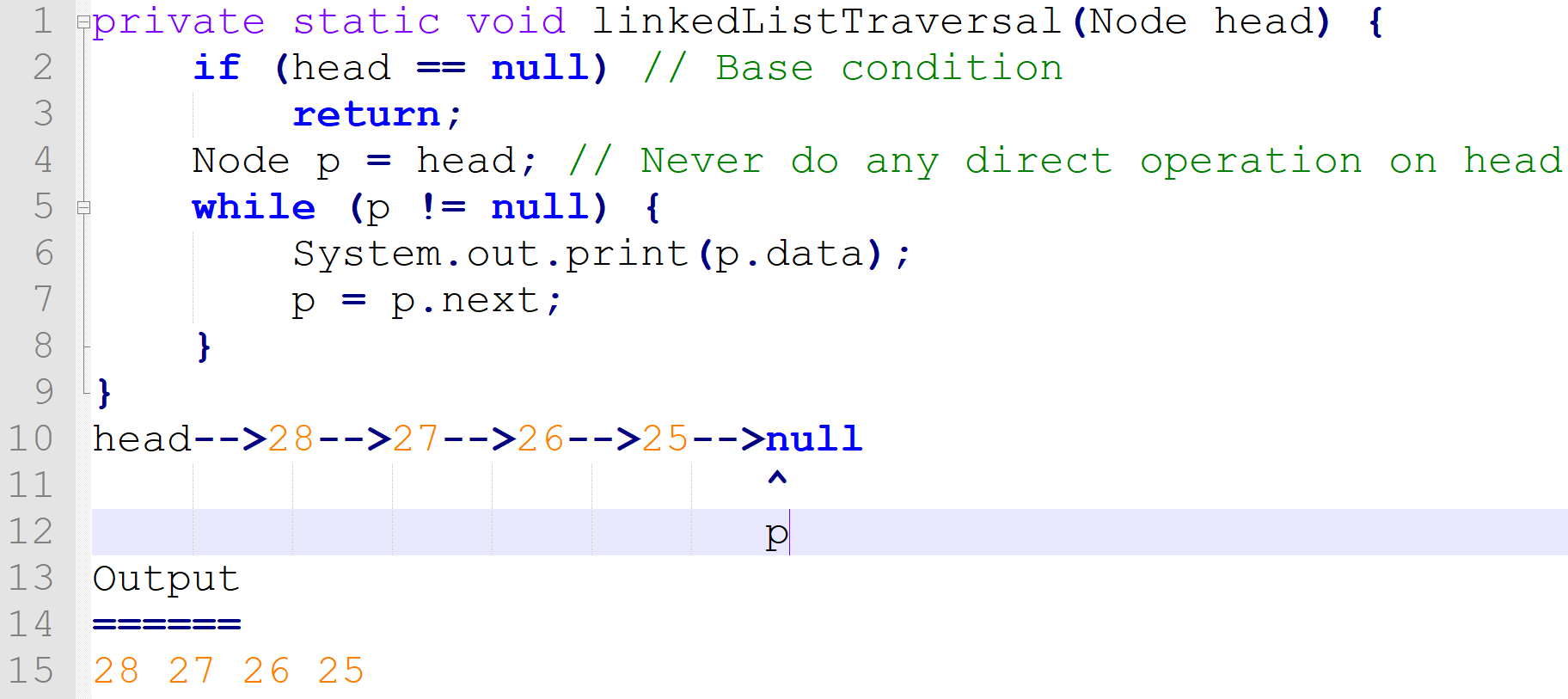
p = p.next;

}

}

}

**Class Explanation:**



**Solve below Hacker Rank problem: -**

<https://www.hackerrank.com/challenges/print-the-elements-of-a-linked-list/problem>

**Finding Length of a LinkedList: -**

**public** **class** LinkedListLength {

**public** **static** **int** getLinkedListLength(Node head) {

**if** (head == **null**) {

**return** 0;

}

Node p = head;

**int** count = 0;

**while** (p != **null**) {

count++;

p = p.next;

}

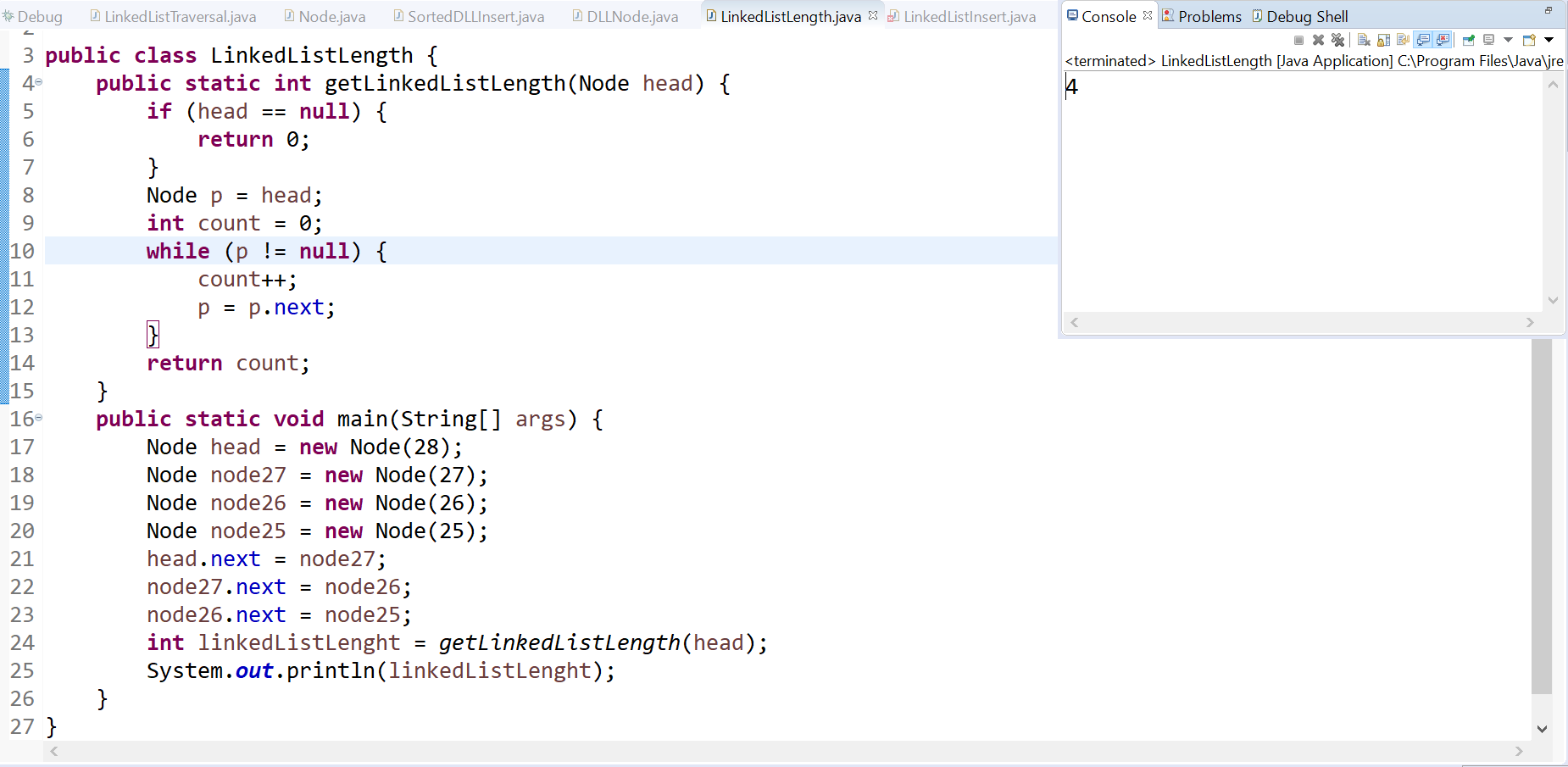
**return** count;

}

}

O(n)-->TC

O(1)-->SC



**Insert first in Linked List**

**public** Node insertFirst(Node head, **int** x) {

Node q = **new** Node(x);

// Base Condition

**if** (head == **null**) {

**return** q;

}

Node p = head;

q.next = p;

**return** q;

}

O(1)-->TC

O(1)-->SC

<https://www.hackerrank.com/challenges/insert-a-node-at-the-head-of-a-linked-list/problem>

**Insert First in LinkedList: -**

**public** **static** Node insertFirst(Node head, **int** x) {

Node p = **new** Node(x);

**if** (head == **null**) {

**return** p;

}

p.next = head;

**return** p;

}

**Insert Last in LinkedList: -**

**public** **static** Node insertLast(Node head, **int** x) {

Node p = **new** Node(x);

**if** (head == **null**) {

**return** p;

}

Node q = head;

**while** (q.next != **null**) {

q = q.next;

}

q.next = p;

**return** head;

}

**Insert Last in LinkedList: -**

**public** **static** Node insertAtIndex(Node head, **int** x, **int** index) {

**int** lenght = LinkedListLength.*getLinkedListLength*(head);

Node updatedLinkedList = **null**;

**if** (index > lenght) {

**return** **null**;// You can throw an exception here!

}

**if** (index == 0) {

updatedLinkedList = *insertFirst*(head, x);

**return** updatedLinkedList;

}

**if** (index == lenght) {

updatedLinkedList = *insertLast*(head, x);

**return** updatedLinkedList;

}

Node p = head;

Node q = **new** Node(x);

**int** count = 1;

**while** (p.next != **null**) {

Node r = p.next;

**if** (index == count) {

p.next = q;

q.next = r;

**break**;

}

p=p.next;

count++;

}

**return** head;

}

**Hacker Rank Solution: -**

static SinglyLinkedListNode insertNodeAtPosition(SinglyLinkedListNode head, int data, int position) {

    return insertAtIndex(head, data, position);

}

public static SinglyLinkedListNode insertFirst(SinglyLinkedListNode head, int x) {

    SinglyLinkedListNode p = new SinglyLinkedListNode(x);

    if (head == null) {

        return p;

    }

    p.next = head;

    return p;

}

public static SinglyLinkedListNode insertLast(SinglyLinkedListNode head, int x) {

    SinglyLinkedListNode p = new SinglyLinkedListNode(x);

    if (head == null) {

        return p;

    }

    SinglyLinkedListNode q = head;

    while (q.next != null) {

        q = q.next;

    }

    q.next = p;

    return head;

}

public static SinglyLinkedListNode insertAtIndex(SinglyLinkedListNode head, int x, int index) {

    int lenght = getLinkedListLength(head);

    SinglyLinkedListNode updatedLinkedList = null;

    if (index > lenght) {

        return null;// You can throw an exception here!

    }

    if (index == 0) {

        updatedLinkedList = insertFirst(head, x);

        return updatedLinkedList;

    }

    if (index == lenght) {

        updatedLinkedList = insertLast(head, x);

        return updatedLinkedList;

    }

    SinglyLinkedListNode p = head;

    SinglyLinkedListNode q = new SinglyLinkedListNode(x);

    int count = 1;

    while (p.next != null) {

        SinglyLinkedListNode r = p.next;

        if (index == count) {

            p.next = q;

            q.next = r;

            break;

        }

        p=p.next;

        count++;

    }

    return head;

}

//Not needed as per our Hacker Rank Problem.

public static int getLinkedListLength(SinglyLinkedListNode head) {

    if (head == null) {

        return 0;

    }

    SinglyLinkedListNode p = head;

    int count = 0;

    while (p != null) {

        count++;

        p = p.next;

    }

    return count;

}

**Solve below Hacker Rank problem: -**

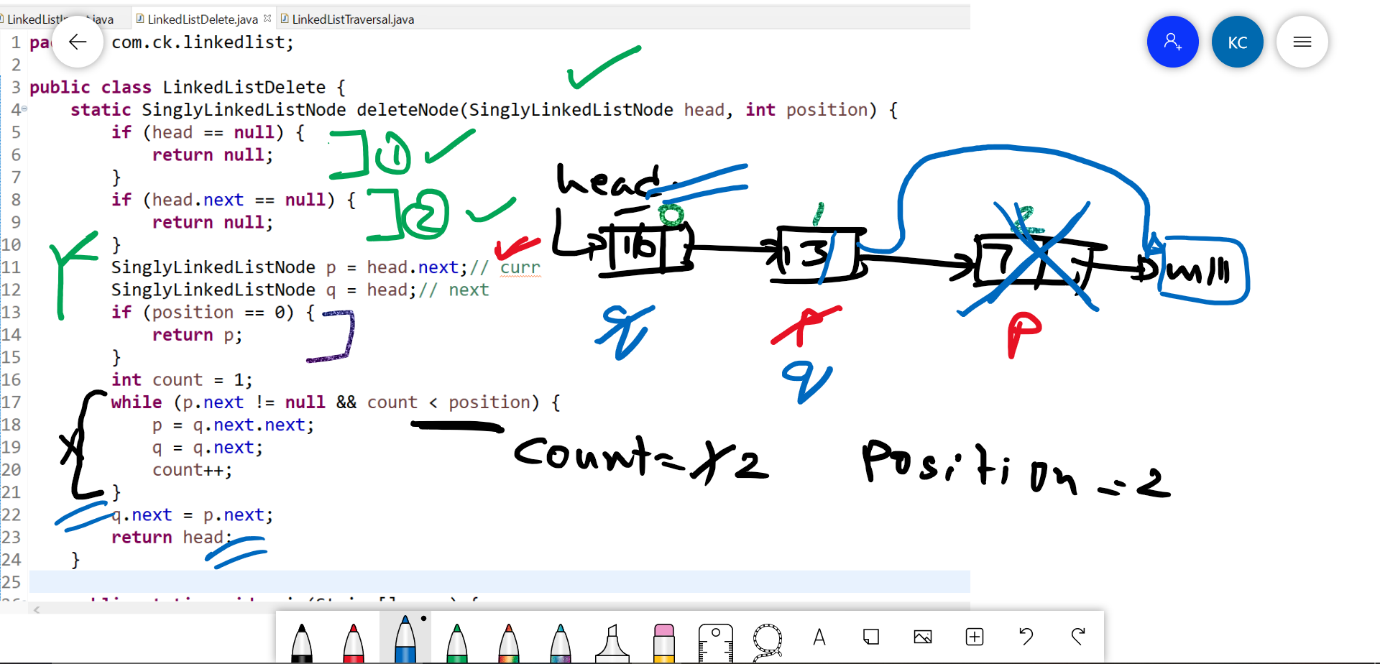
<https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list/problem>

<https://www.hackerrank.com/challenges/insert-a-node-at-the-tail-of-a-linked-list/problem>

**Deleting in Linked List**

There are three places where deletion can happen.

1. Delete First
2. Delete Last
3. Delete at position



**Code in eclipse: -**

**public** **class** LinkedListDelete {

**static** SinglyLinkedListNode deleteNode(SinglyLinkedListNode head, **int** position) {

**if** (head == **null**) {

**return** **null**;

}

**if** (head.next == **null**) {

**return** **null**;

}

SinglyLinkedListNode p = head.next;// curr

SinglyLinkedListNode q = head;// next

**if** (position == 0) {

**return** p;

}

**int** count = 1;

**while** (p.next != **null** && count < position) {

p = q.next.next;

q = q.next;

count++;

}

q.next = p.next;

**return** head;

}

**public** **static** **void** main(String[] args) {

SinglyLinkedListNode head = **new** SinglyLinkedListNode(16);

SinglyLinkedListNode node13 = **new** SinglyLinkedListNode(13);

SinglyLinkedListNode node7 = **new** SinglyLinkedListNode(7);

head.next = node13;

node13.next = node7;

*deleteNode*(head, 2);

LinkedListTraversal.*linkedListTraversalNew*(head);

}

}

**Hacker Rank code: -**

static SinglyLinkedListNode deleteNode(SinglyLinkedListNode head, int position) {

        if(head==null){

            return null;

        }

        if(head.next==null){

            return null;

        }

        SinglyLinkedListNode p = head.next;//curr

        SinglyLinkedListNode q = head;//next

        if(position==0){

            return p;

        }

        int count =1;

        while(p.next!=null && count<position){

            p=q.next.next;

            q=q.next;

            count++;

        }

        System.out.println("p="+p.data);

        q.next=p.next;

        return head;

    }

**Find middle element of a LinkedList :-**

**Approach 1 :**

**public** **class** FindMiddleOfLinkedList {

**public** **static** Node findMiddle(Node head) {

**if**(head == **null**) {

**return** **null**;

}

**if**(head.next==**null**) {

**return** head;

}

Node p = head;

**int** lenght = LinkedListLength.*getLinkedListLength*(head);

**int** middle = (lenght/2)-1;

**int** count = 0;

**while**(count<middle) {

p=p.next;

count++;

}

**return** p;

}

**public** **static** **void** main(String[] args) {

Node head = **new** Node(28);

Node node27 = **new** Node(27);

Node node26 = **new** Node(26);

Node node25 = **new** Node(25);

Node node24 = **new** Node(24);

head.next = node27;

node27.next = node26;

node26.next = node25;

node25.next = node24;

Node middleNode = *findMiddle*(head);

System.***out***.println(middleNode.data);

}

}

Time Complexity -> O(n)+O(n/2) =O(n)

Space Complexity -> O(1)

**Approach 2:-**

**private** **static** Node getMiddleElementOfLinkedList(Node head) {

**if** (head == **null**)// Base condition

**return** **null**;

Node p = head;// slow pointer

Node q = head;// fast pointer

**while** (q != **null** && q.next != **null**) {

p = p.next;

q = q.next.next;

}

**return** p;

}

Time Complexity -> O(n/2) =O(n)

Space Complexity -> O(1)

**Is Circular Linked List**

**public** **class** IsCircularLinkedList {

**public** **static** **boolean** isCircular(Node head) {

**if** (head == **null** || head.next == **null** || head.next.next == **null**) {

**return** **false**;

}

Node p = head;// slow pointer

Node q = head;// fast pointer

**while** (q != **null** && q.next != **null**) {

p = p.next;

q = q.next.next;

**if** (p == q) {

**return** **true**;

}

}

**return** **false**;

}

**public** **static** **void** main(String[] args) {

Node head = **new** Node(28);

Node node28 = head;

Node node27 = **new** Node(27);

Node node26 = **new** Node(26);

Node node25 = **new** Node(25);

Node node24 = **new** Node(24);

head.next = node27;

node27.next = node26;

node26.next = node25;

node25.next = node24;

node24.next = node28;

**boolean** isCircular = *isCircular*(head);

System.***out***.println(isCircular);

}

}

**Find if Loop exists: -**

static boolean hasCycle(SinglyLinkedListNode head) {

        if (head == null || head.next == null || head.next.next == null) {

            return false;

        }

        SinglyLinkedListNode p = head;// slow pointer

        SinglyLinkedListNode q = head;// fast pointer

        while (q != null && q.next != null) {

            p = p.next;

            q = q.next.next;

            if (p == q) {

                return true;

            }

        }

        return false;

    }

**Link:**

<https://www.hackerrank.com/challenges/detect-whether-a-linked-list-contains-a-cycle/problem>

**Intersection point in a Linked List :-**



**Algorithm: -**

1. Finalize the base conditions
2. Get lengths of two linked lists
3. Start with LinkedList with larger length and iterate until the length difference.
4. Start iterating both LinkedLists until the next pointer becomes null
   1. If both nodes are equal in iteration, return true
   2. Else, return false.

**Program: -**

**public** **class** IntersectionPoint {

**public** Node getIntersectionNode(Node headA, Node headB) {

//Base condition

**if**(headA==**null** || headB==**null**) {

**return** **null**;

}

// Step 1

**int** lenA = getLength(headA);

**int** lenB = getLength(headB);

// Step 2

Node p = headA;

Node q = headB;

**while** (lenA > lenB) {

p = p.next;

lenA--;

}

**while** (lenA < lenB) {

q = q.next;

lenB--;

}

// Step 3

**while** (p != q) {

p = p.next;

q = q.next;

}

**return** p;

}

**private** **int** getLength(Node node) {

**int** length = 0;

**while** (node != **null**) {

node = node.next;

length++;

}

**return** length;

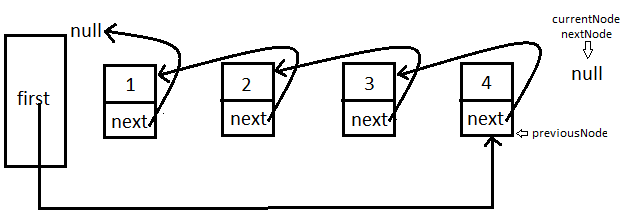
}

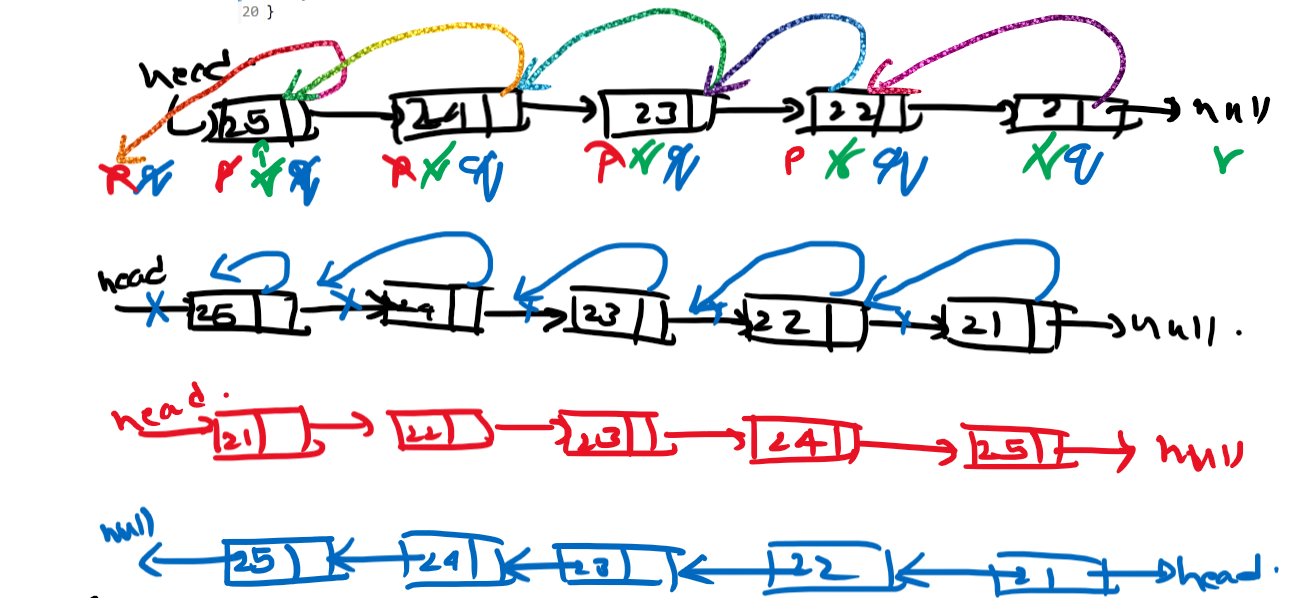
}

**LeetCode Link: -**

<https://leetcode.com/problems/intersection-of-two-linked-lists/>

**Find reverse of a LinkedList: -**





**public** **class** ReverseLinkedList {

**static** Node Reverse(Node head) {

//Base Condition

**if** (head == **null**) {

**return** **null**;

}

Node p = **null**;//previous

Node q = **null**;//current

Node r = head;//next

**while** (r != **null**) {

p = q;

q = r;

r = r.next;

q.next = p;//q is behind p. Actual reverse logic.

}

**return** q;

}

**public** **static** **void** main(String[] args) {

Node head = **new** Node(28);

Node node27 = **new** Node(27);

Node node26 = **new** Node(26);

Node node25 = **new** Node(25);

head.next = node27;

node27.next = node26;

node26.next = node25;

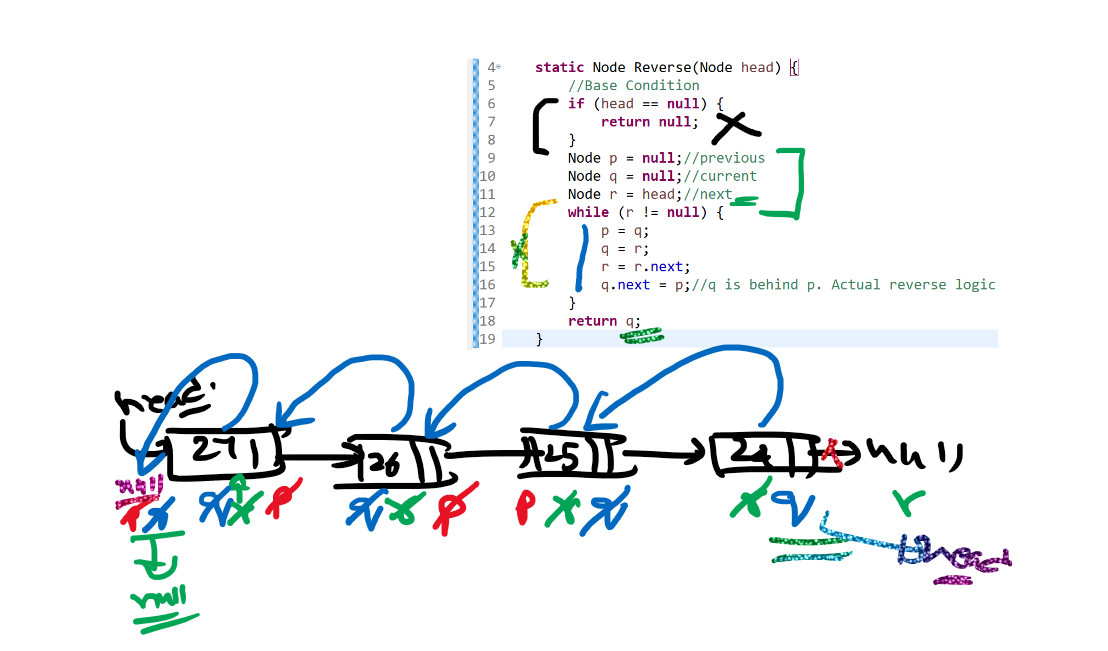
LinkedListTraversal.*linkedListTraversal*(head);

Node reverseLinkedList = *Reverse*(head);

LinkedListTraversal.*linkedListTraversal*(reverseLinkedList);

}

}



**Print Reverse order of a LinkedList using Recursion :-**

**public** **static** Node reverseUsingRecursion(Node head) {

**if**(head==**null**) {

**return** **null**;

}

Node p = head;

*reverseUsingRecursion*(p.next);

System.***out***.print(head.data+" ");

**return** head;

}

**public** **static** **void** main(String[] args) {

Node head = **new** Node(28);

Node node27 = **new** Node(27);

Node node26 = **new** Node(26);

Node node25 = **new** Node(25);

head.next = node27;

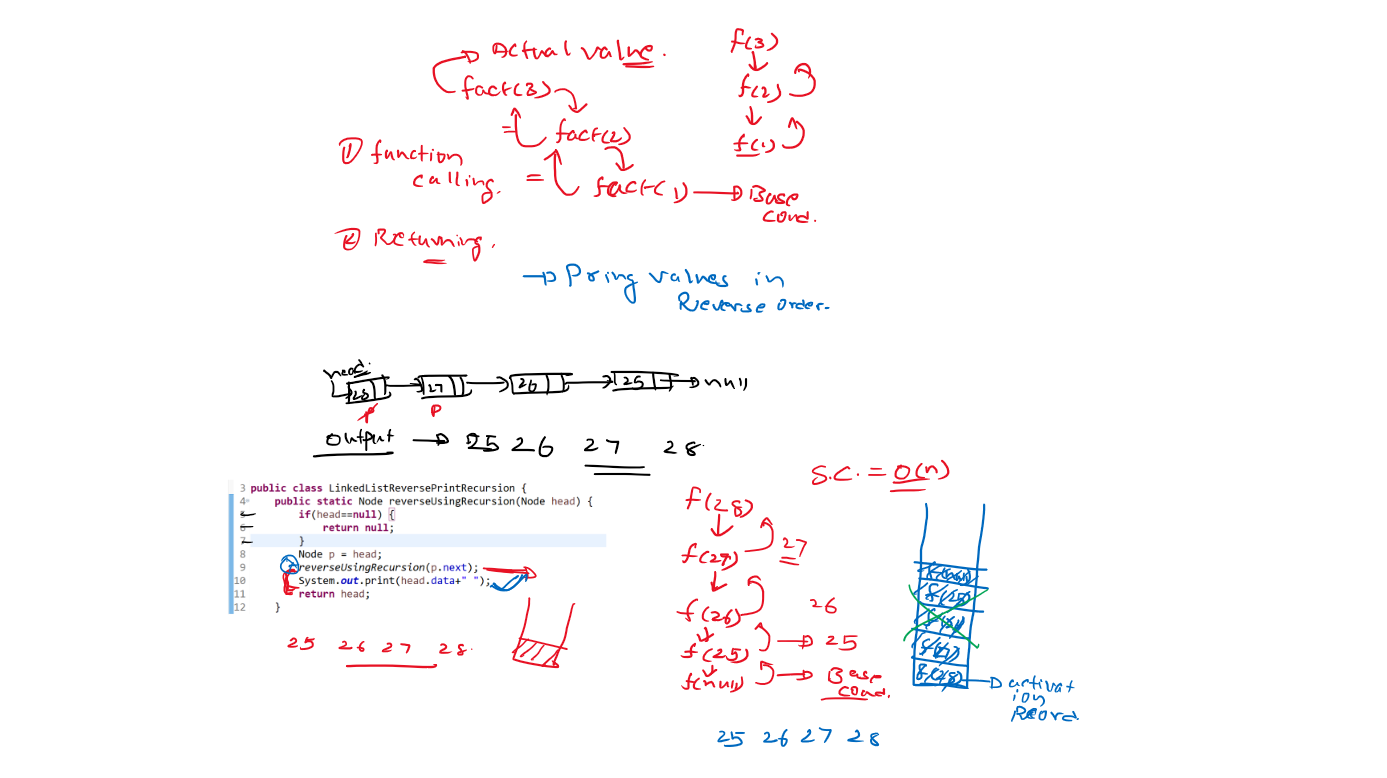
node27.next = node26;

node26.next = node25;

*reverseUsingRecursion*(head);

}

**Explanation: -**



**Using Recursion: -**

**public** **class** LinkedListReverseRecursion {

**static** Node *reversedHead*;

**public** **static** Node reverseLinkedListUsingRecursion(Node head) {

**if** (head == **null**) {

**return** **null**;

}

**if** (head.next == **null**) {

*reversedHead* = head;

**return** *reversedHead*;

}

*reverseLinkedListUsingRecursion*(head.next);

// executed for every recursive call, other than base case.

Node q = head.next;

q.next = head;

head.next = **null**;

**return** *reversedHead*;

}

**public** **static** **void** main(String[] args) {

Node head = **new** Node(28);

Node node27 = **new** Node(27);

Node node26 = **new** Node(26);

Node node25 = **new** Node(25);

head.next = node27;

node27.next = node26;

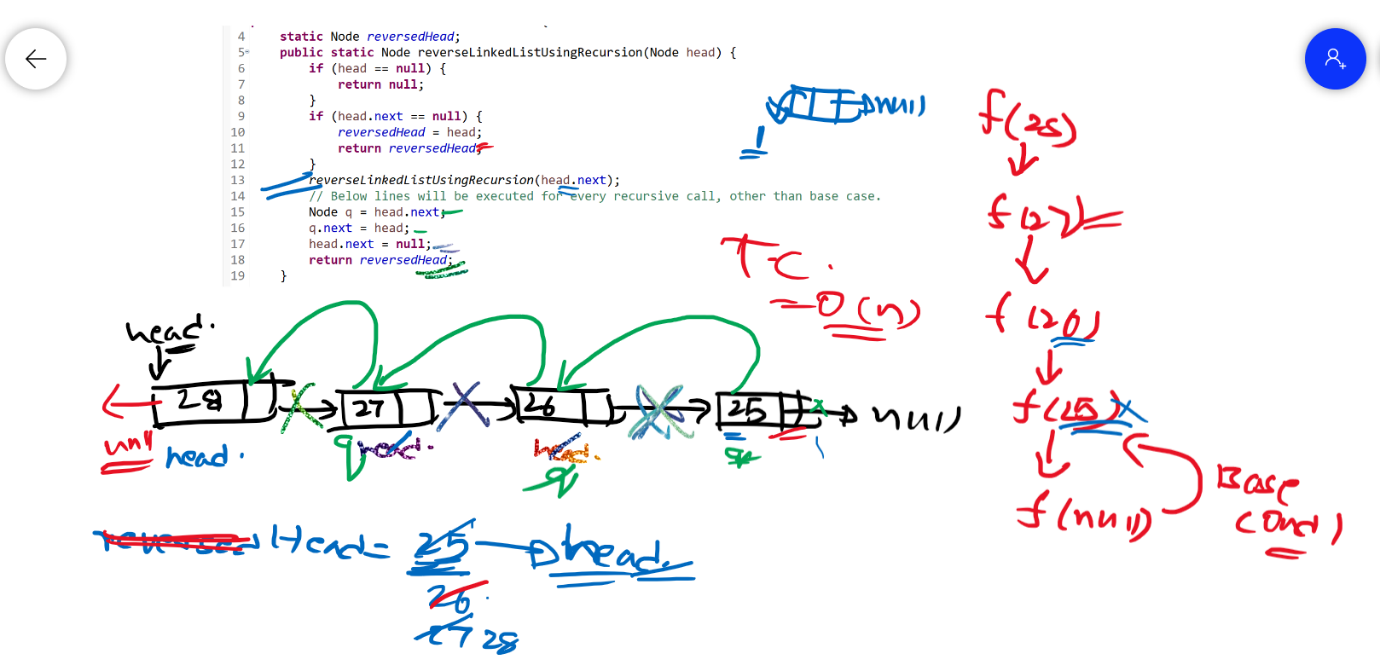
node26.next = node25;

Node reversedLinkedList = *reverseLinkedListUsingRecursion*(head);

LinkedListTraversal.*linkedListTraversal*(reversedLinkedList);

}

}

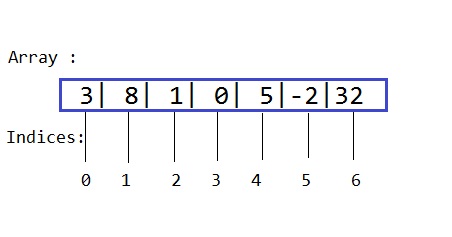


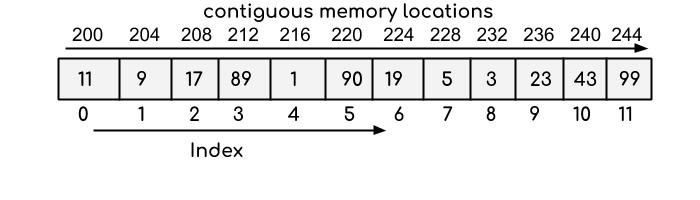
**Hacker Rank Link: -**

[**https://www.hackerrank.com/challenges/reverse-a-linked-list/problem**](https://www.hackerrank.com/challenges/reverse-a-linked-list/problem)

**Array: -**

* The length of an array is fixed.
* An array is a collection of homogeneous (same type) data items.
* The data items are stored in contiguous memory locations.





**public** **class** ArrayTraverse {

**public** **static** **void** main(String[] args) {

**int**[] strArray = { 11, 9, 17, 89, 1, 90, 19, 5, 3, 23, 43, 99 };

**for** (**int** i = 0; i < strArray.length; i++) {

System.***out***.println(strArray[i]);

}

}

}

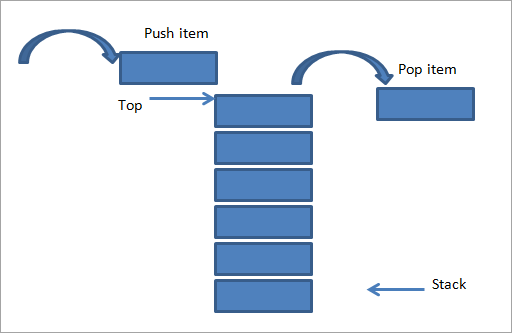
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time Complexity** | | | **Time Complexity** | | |
|  | **Array** | **LinkedList** |  | **Array** | **LinkedList** |
| **Insert@End** | **O(1)** | **O(n)** | **Delete@End** | **O(1)** | **O(n)** |
| **Insert@Begin** | **O(n)** | **O(1)** | **Delete@Begin** | **O(n)** | **O(1)** |

**Stacks and Queues:-**

Both Stacks and Queues are **Logical data structures.**

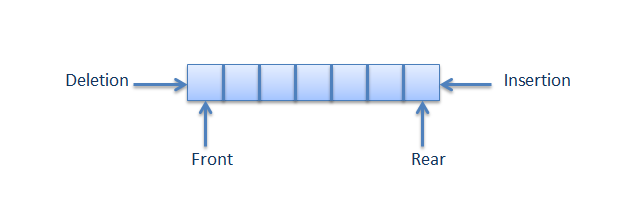
**Stack :-**

1. Insertion and Deletion happens at one end only, called as **top**.
2. This follows **LIFO** (Last in First Out)
3. It supports two main operations:
   1. **push()** => used for insertion on top.
   2. **pop()** => used for deletion from top.
4. This can be implemented using arrays and linked lists.

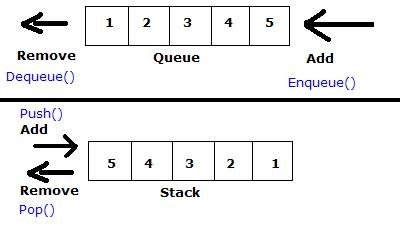


**Queue: -**

1. Insertion and Deletion happens at two different ends called as **front** and **rear.**.
2. This follows **FIFO**(First In First Out)
3. It supports two main operations :
   1. **enque()** => used for insertion on top.
   2. **deque()** => used for deletion from top.
4. This can be implemented using arrays and linked lists.



**Difference between Stack and Queue:-**

****