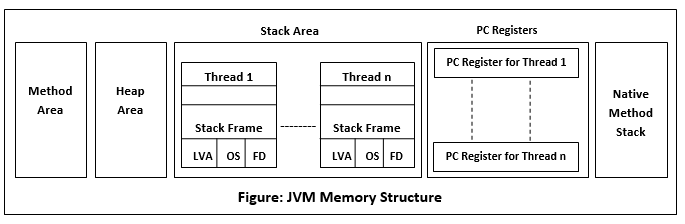
**Data Structures – Foundation**

**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*;

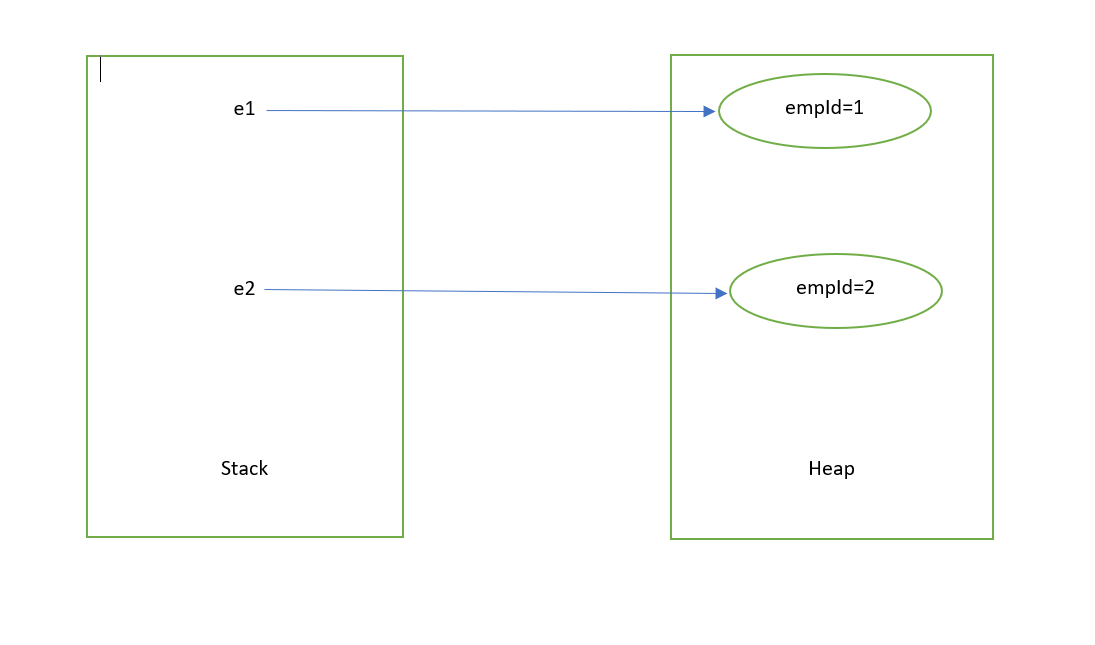
**public** **static** **void** main(String[] args) {

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();

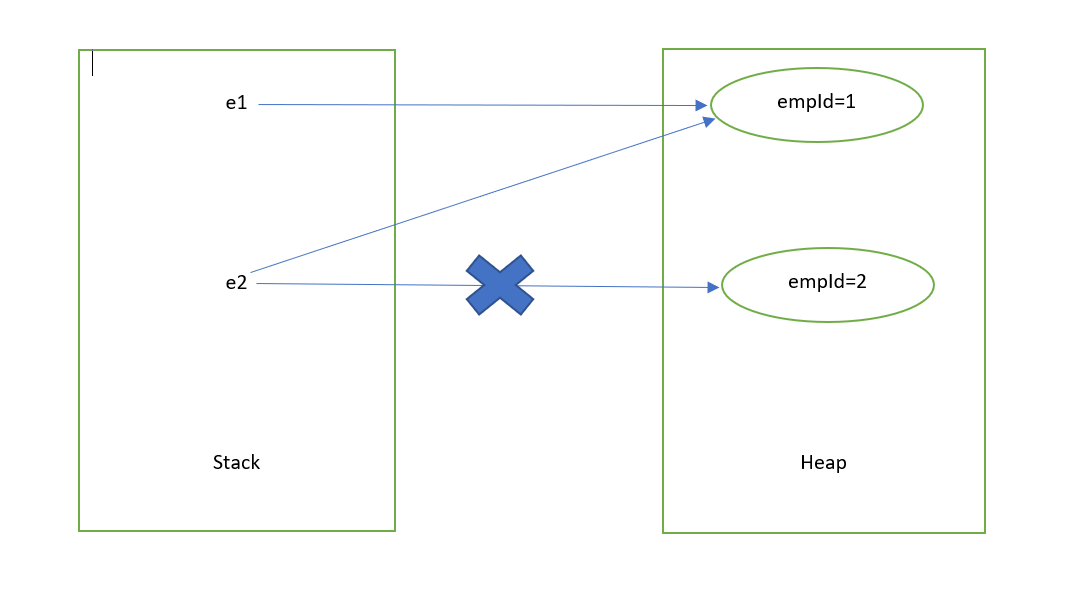
Employee e2 = **new** Employee();

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;

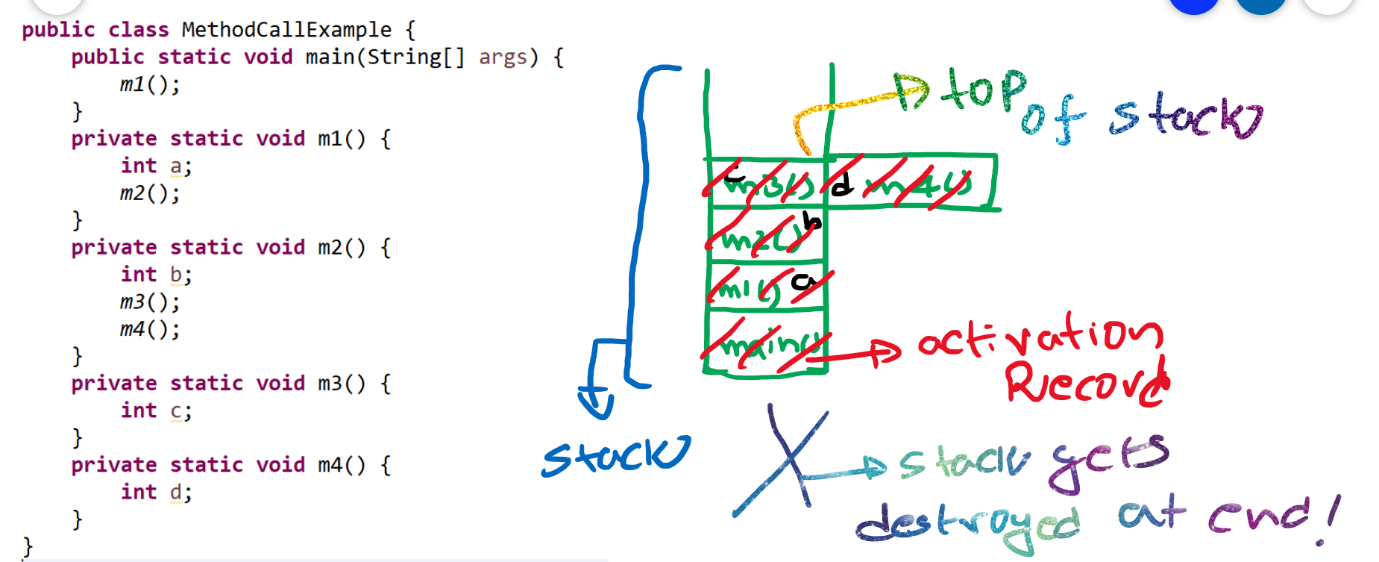
}

**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.

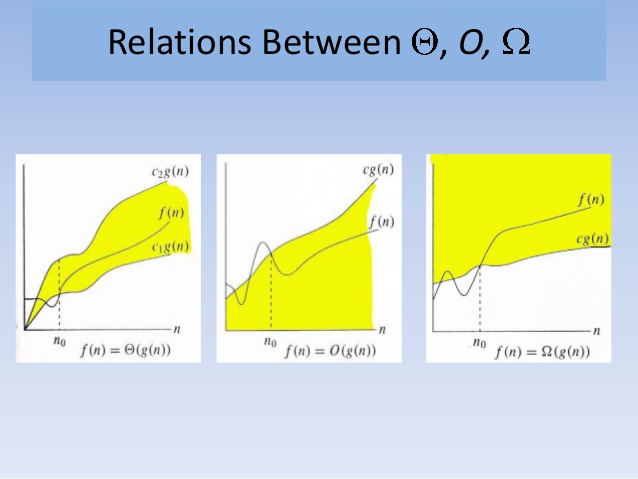
But, in case 1, if it’s a small text file, it is sent over internet in no time. But, if the file is larger, say 1 TB, it takes days to transfer that file electronically. So the growth rate depends on the input size of your file.

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)

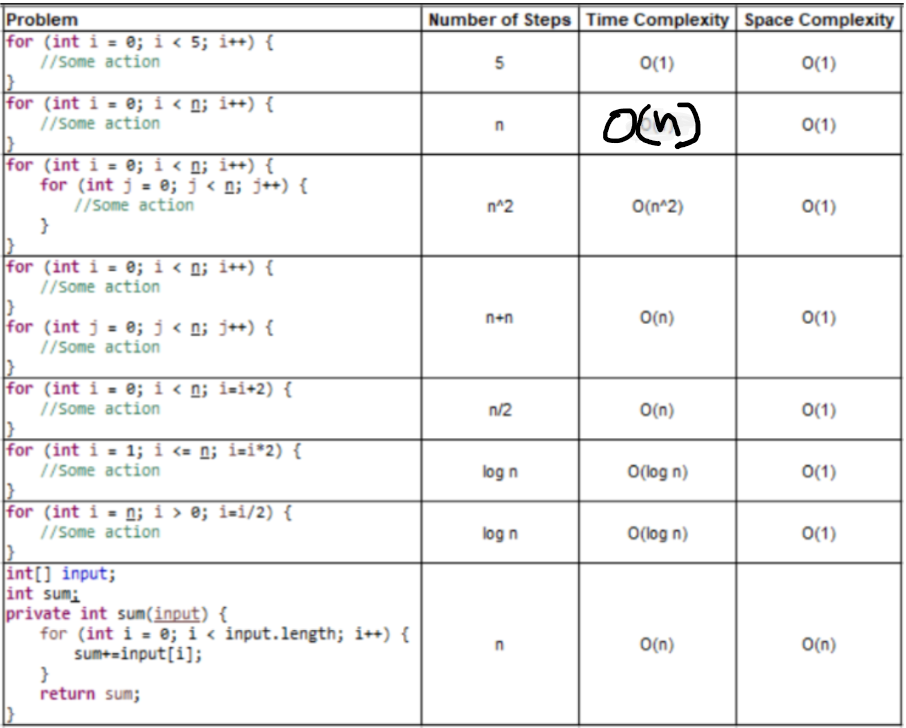
Say, you went to buy a pair of Shoes and Socks. Your friend came and asked you what you are buying. You simply say that you came to buy pair of shoes. You simply ignore socks, even if you had already bought them. The reason why you ignore is that the value of socks is negligible in comparison to that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

Similarly,

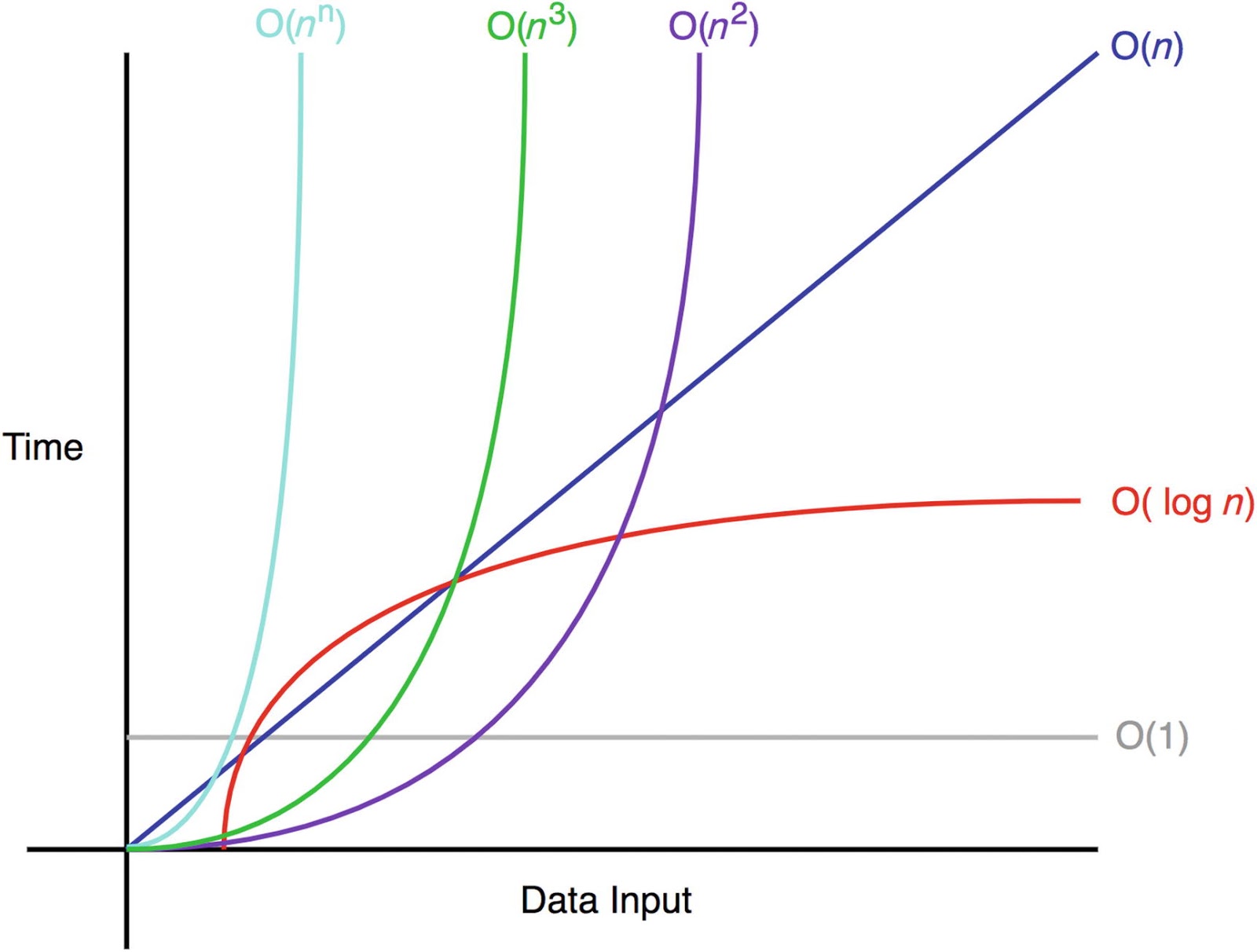
in **f(n)=3n+2**, we ignore 3 and 2 as they are constants the time complexity becomes **O(n)**

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

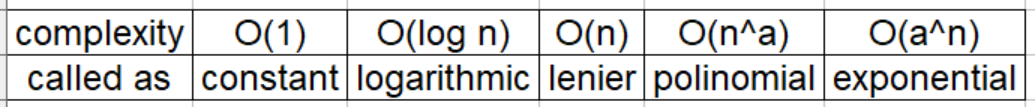
We only concentrate on Big-O notations as we analyse worst cases. There is no point of analysing best case because, most algorithms will be O(1) in the best case.



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:

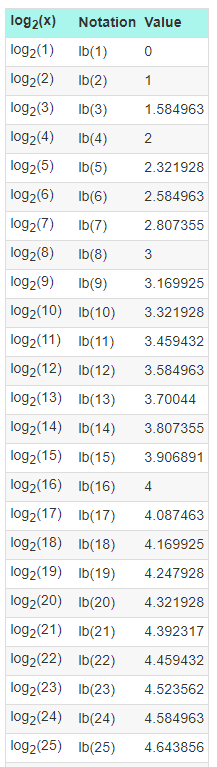
Say input is 8. We can divide 8 with 2, for 3 times to make its value to be 1.

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;

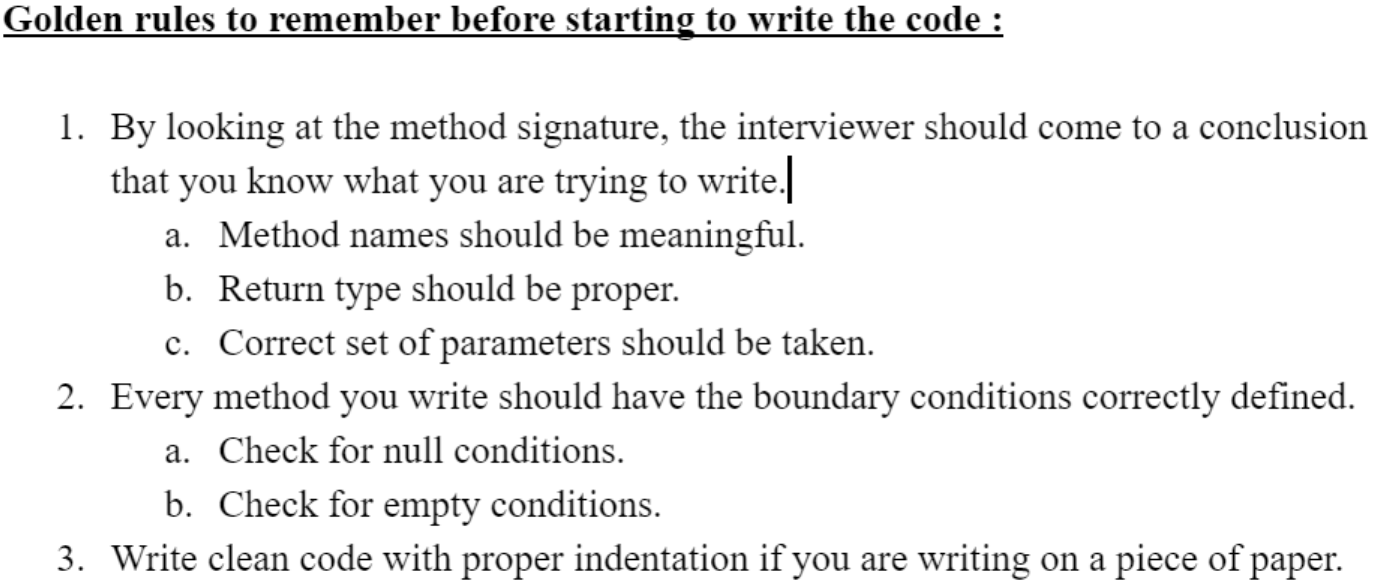
}

}

**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, "");

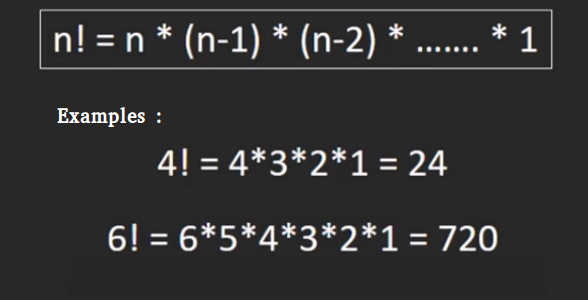
Employee changedEmployee = *changeName*(e1, "");

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;

}

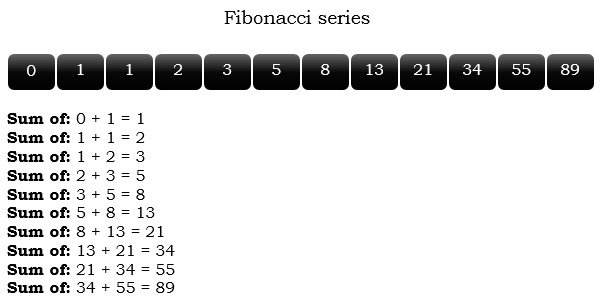
**return** result;

}

}

**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);

}

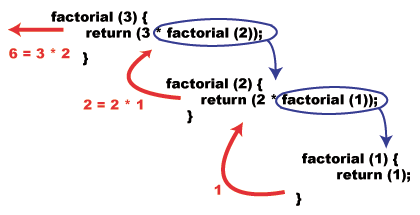
}

**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);

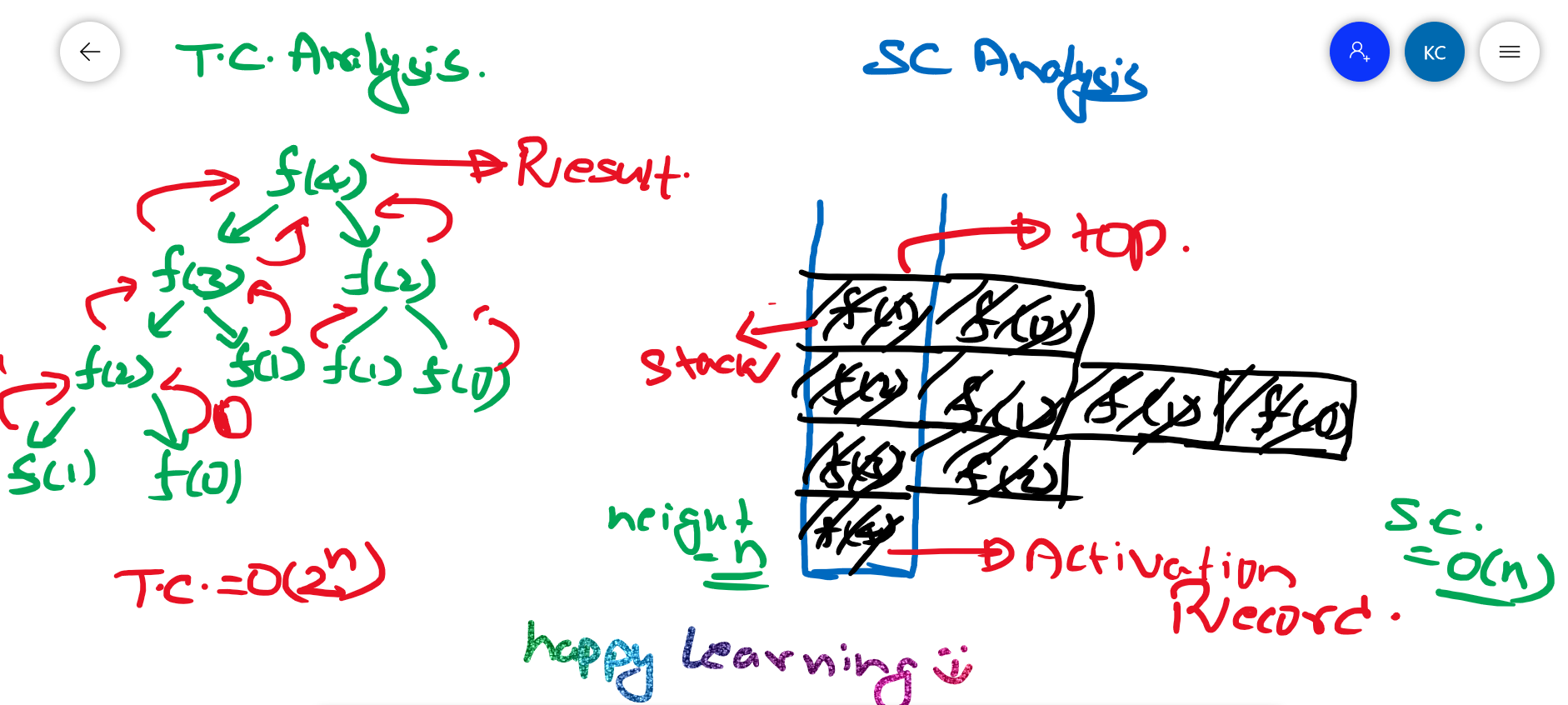
}

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

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

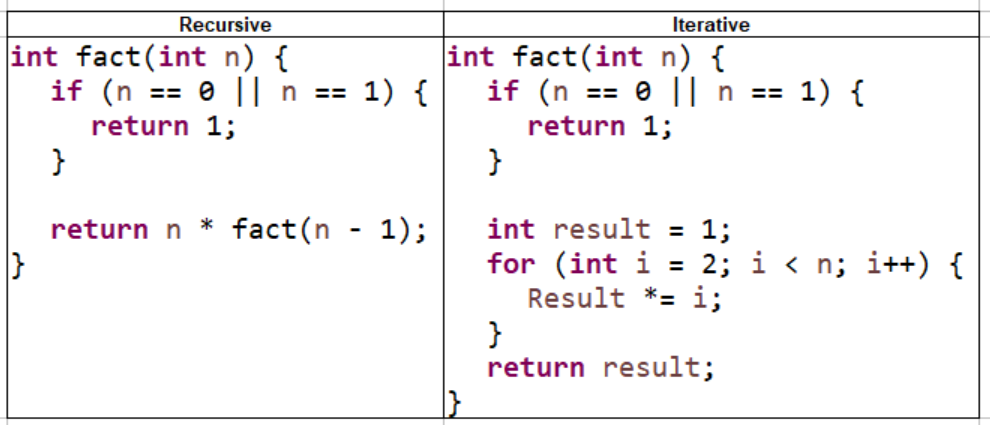
}

}

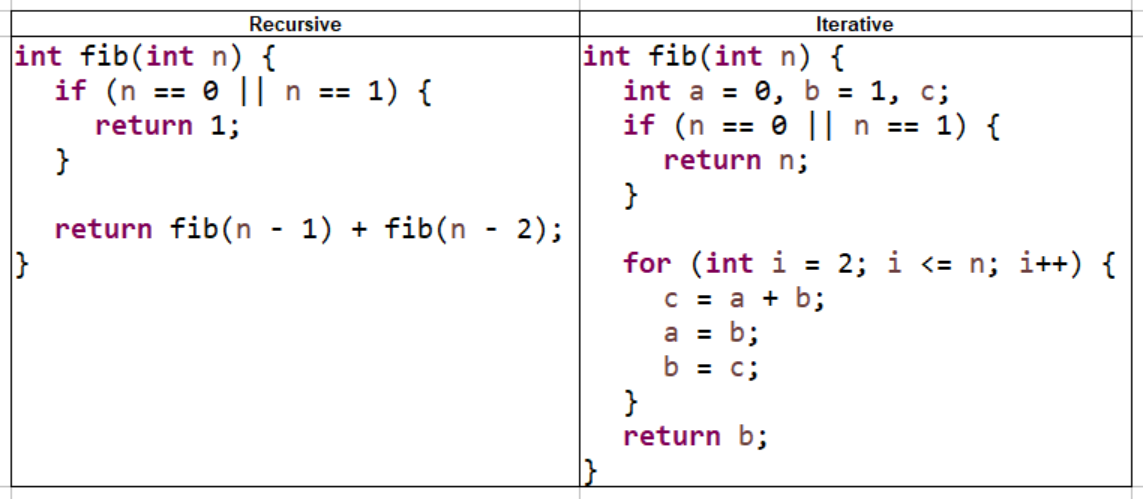


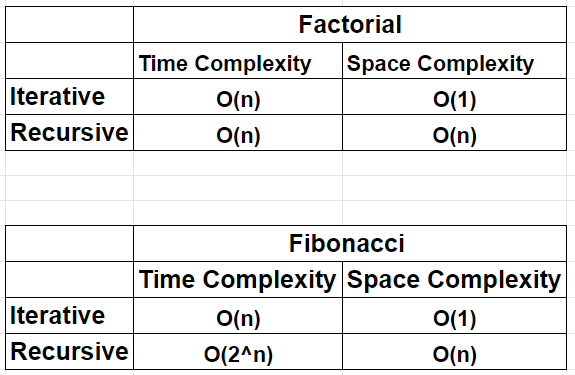
**Comparison: -**

**Program for Factorial:-**

****

**Program for Fibonacci series: -**

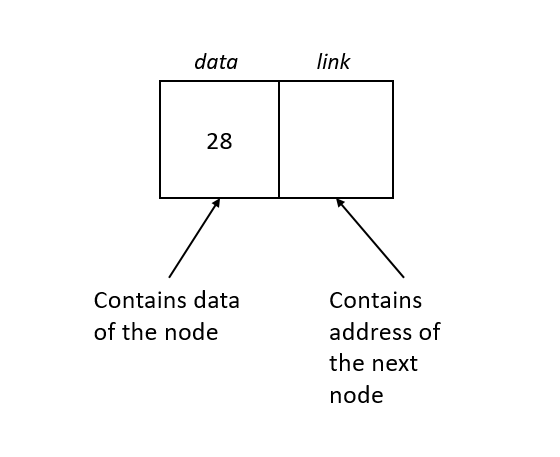
****



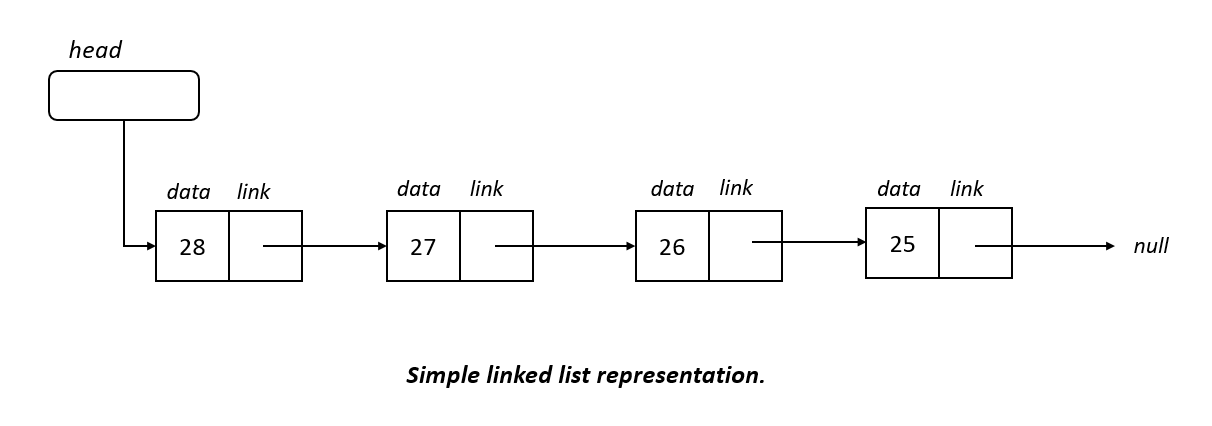
**Linked List**

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.

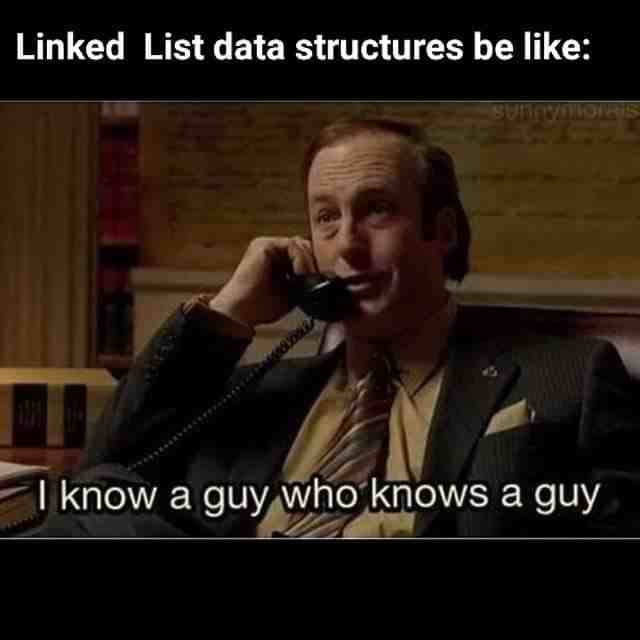
**Below is the representation of a single Node: -**



**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);

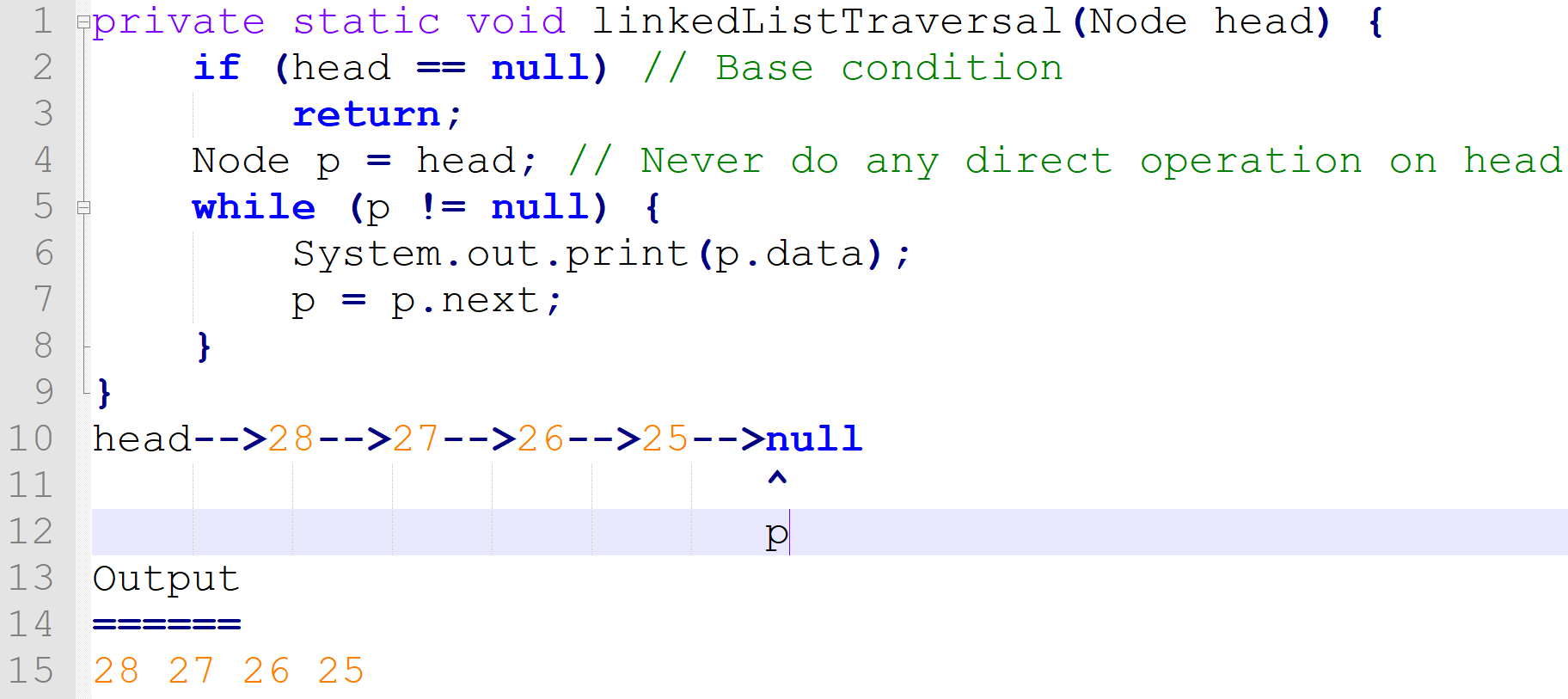
p = p.next;

}

}

}

**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: -**

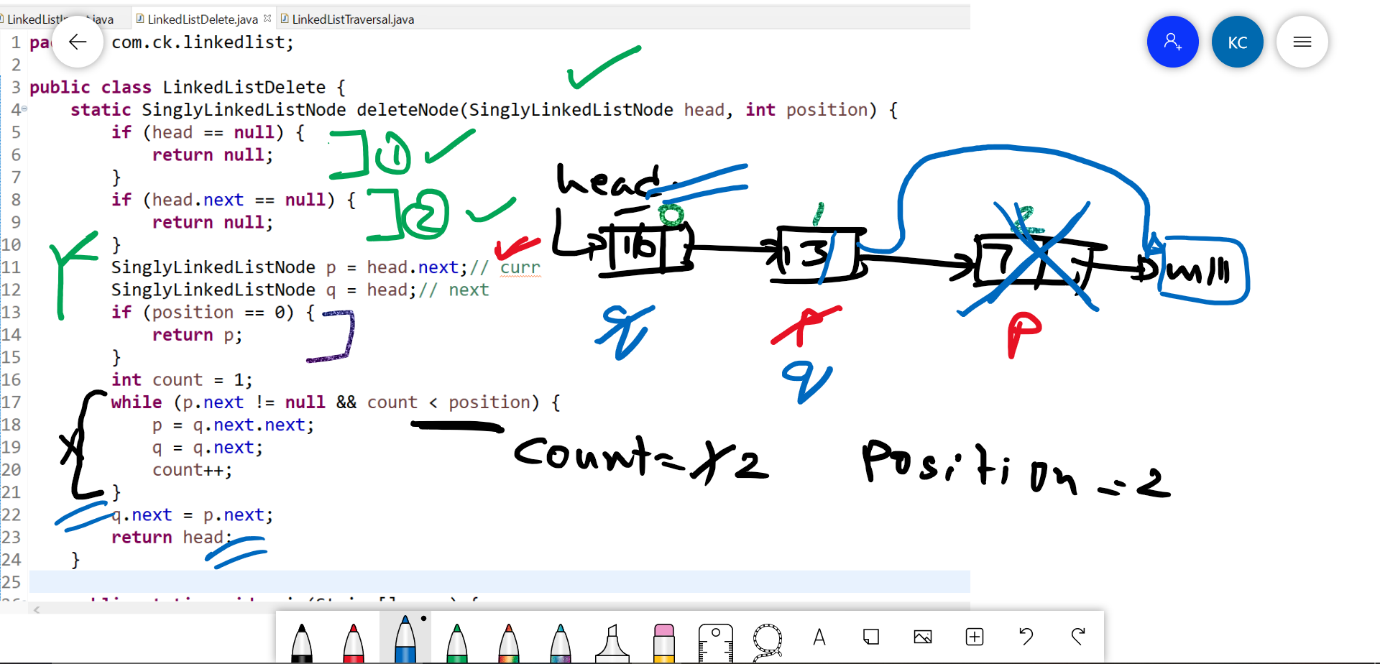
<https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list/problem>

<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;

            }

        }

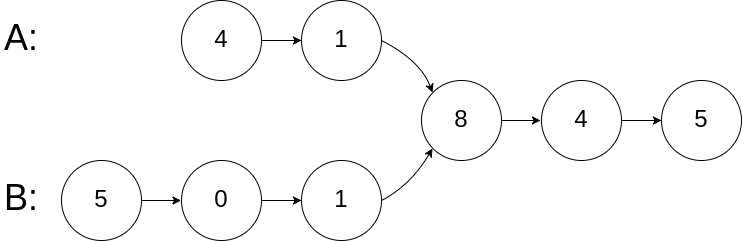
        return false;

    }

**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;

}

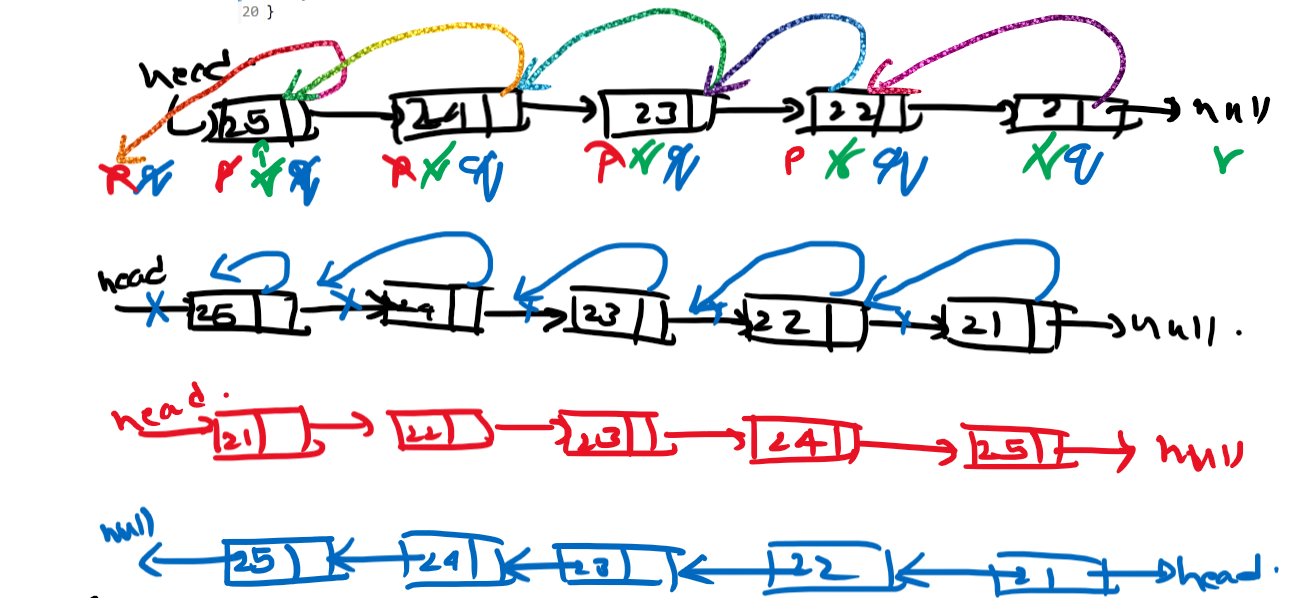
}

**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;

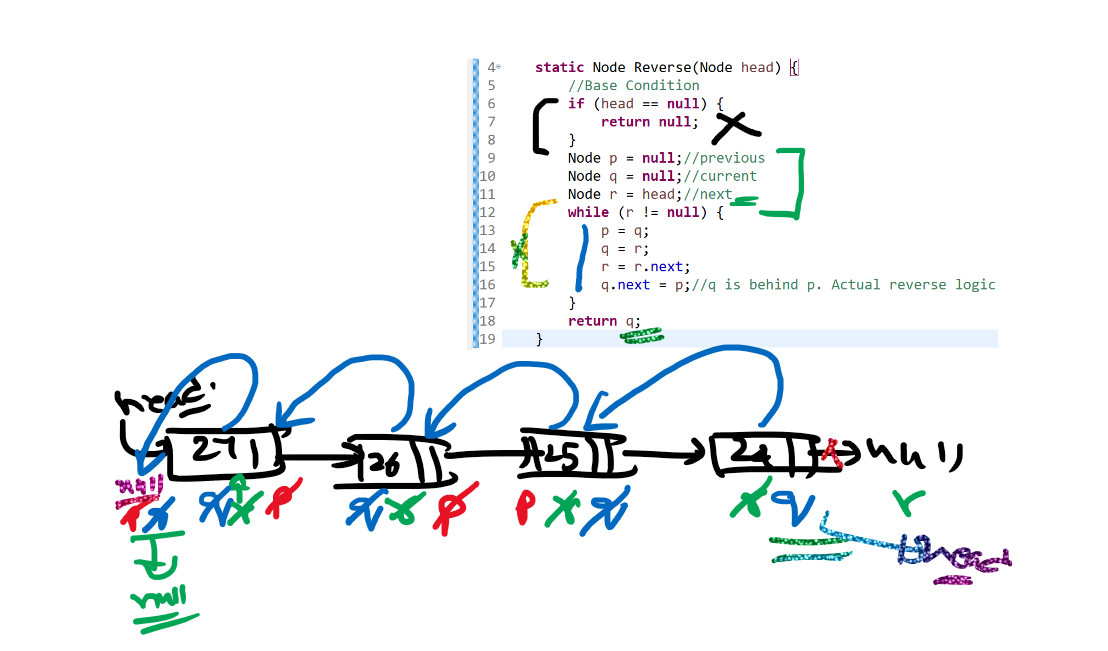
LinkedListTraversal.*linkedListTraversal*(head);

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;

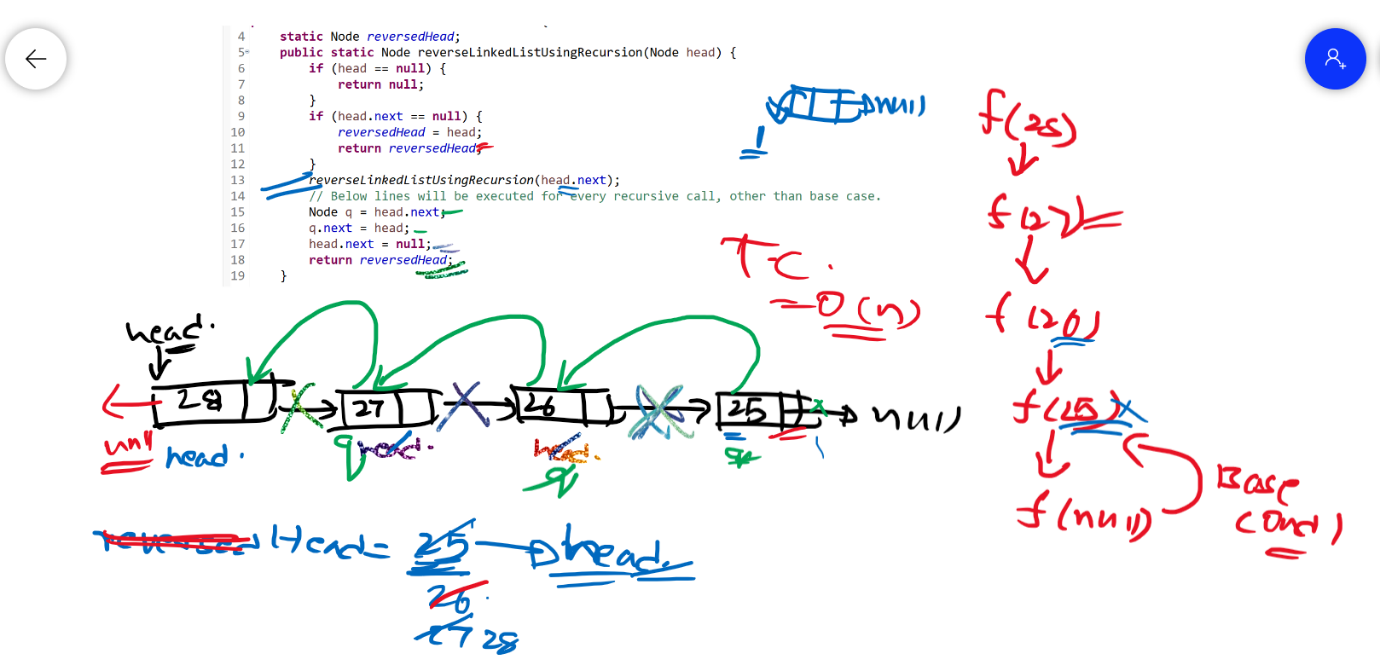
node26.next = node25;

Node reversedLinkedList = *reverseLinkedListUsingRecursion*(head);

LinkedListTraversal.*linkedListTraversal*(reversedLinkedList);

}

}

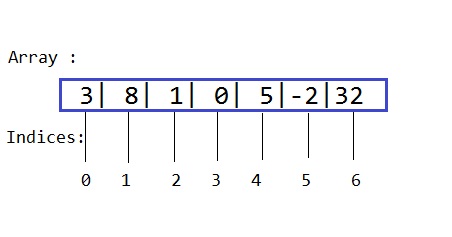


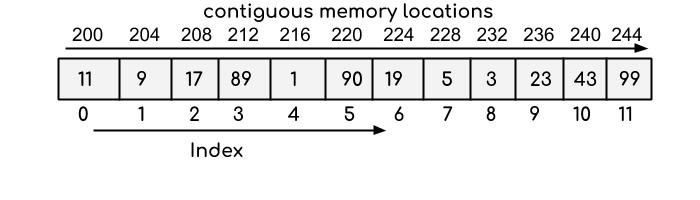
**Hacker Rank Link: -**

[**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]);

}

}

}

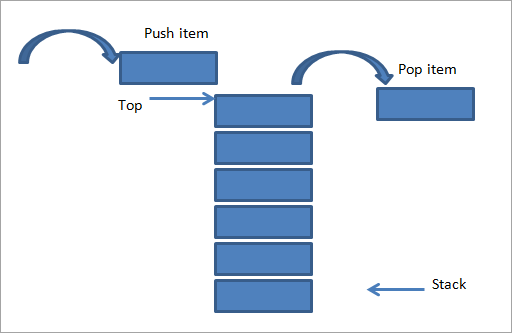
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time Complexity** | | | **Time Complexity** | | |
|  | **Array** | **LinkedList** |  | **Array** | **LinkedList** |
| **Insert@End** | **O(1)** | **O(n)** | **Delete@End** | **O(1)** | **O(n)** |
| **Insert@Begin** | **O(n)** | **O(1)** |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

**Stacks and Queues:-**

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

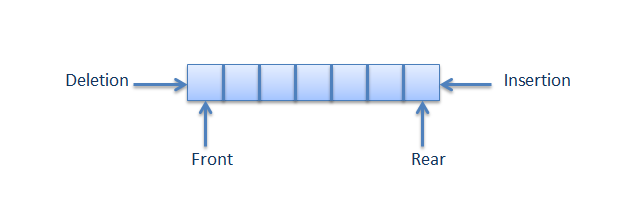
**Stack :-**

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.

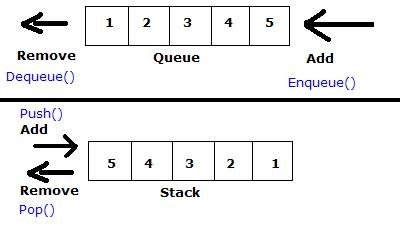


**Queue: -**

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:-**

****

**Conditions: -**

|  |  |  |
| --- | --- | --- |
|  | **Using Array** | |
|  | **Stack with Array** | **Queue with Array** |
| **initial** | **top = -1** | **front = -1, rear = -1** |
| **empty** | **top == -1** | **front == rear** |
| **full** | **top == lenght -1** | **rear = length -1** |
| **elements count** | **top + 1** | **rear - front** |

|  |  |  |
| --- | --- | --- |
|  | **Using LinkedList** | |
|  | **Stack With LL** | **Queue with LL** |
| **inital** | **top = null** | **front = null, rear = null** |
| **empty** | **top == null** | **front == null** |
| **full** | **n/a** | **n/a** |
| **elements count** | **maintain count** | **maintain count** |

**public** **class** MyStackArr {

**static** **int** *top* = -1;// Initial Condition

**static** **int**[] *arr* = **new** **int**[5];// Max size of 500

**public** **static** **void** push(**int** x) {

**if** (!*isFull*())

*arr*[++*top*] = x;

**else**

System.***out***.println("Stack OverFlow!");

}

**public** **static** **int** pop() {

**if** (!*isEmpty*())

**return** *arr*[*top*--];

**else**

System.***out***.println("Stack UnderFlow!");

**return** -1;// Assume all numbers in stack are positive.

}

**private** **static** **boolean** isEmpty() {

**if** (*top* == -1)

**return** **true**;

**else**

**return** **false**;

}

**private** **static** **boolean** isFull() {

**if** (*top* == *arr*.length - 1)

**return** **true**;

**else**

**return** **false**;

}

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

*push*(10);

*push*(20);

*push*(30);

*push*(40);

*push*(50);

*printStack*();

System.***out***.println("Testing pop()");

System.***out***.println("Popped out value from stack : "+*pop*());

System.***out***.println("Popped out value from stack : "+*pop*());

System.***out***.println("Popped out value from stack : "+*pop*());

}

**private** **static** **void** printStack() {

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

**if**(*arr*[i]!=0) {

System.***out***.print(*arr*[i]+" ");

}

}

}

}

**public** **class** MyQueueArr {

**int** rear, front = -1;// Initial condition

**int**[] arr = **new** **int**[100];

**public** **void** enQueue(**int** x) {

**if** (!isFull()) {

arr[++rear] = x;

}

}

**public** **int** deQueue() {

**if** (!isEmpty()) {

**return** arr[++front];

}

**return** -1;// Assume numbers in stack are positive.

}

**private** **boolean** isEmpty() {

**if** (front == rear)

**return** **true**;

**else**

**return** **false**;

}

**private** **boolean** isFull() {

**if** (rear == arr.length - 1)

**return** **true**;

**else**

**return** **false**;

}

}

**public** **class** MyQueueArr {

**int** rear, front = -1;// Initial condition

**int**[] arr = **new** **int**[100];

**public** **void** enQueue(**int** x) {

**if** (!isFull()) {

arr[++rear] = x;

}

}

**public** **int** deQueue() {

**if** (!isEmpty()) {

**return** arr[++front];

}

**return** -1;// Assume numbers in stack are positive.

}

**private** **boolean** isEmpty() {

**if** (front == rear)

**return** **true**;

**else**

**return** **false**;

}

**private** **boolean** isFull() {

**if** (rear == arr.length - 1)

**return** **true**;

**else**

**return** **false**;

}

}

**public** **class** MyQueueLL {

Node front, rear = **null**;

**public** **void** push(**int** x) {

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

rear.next = p;

}

**public** **int** pop() {

**if** (!isEmpty()) {

**int** x = front.data;

front = front.next;

**return** x;

}

**return** -1;// Assume numbers in stack are positive.

}

**private** **boolean** isEmpty() {

**if** (front == **null**)

**return** **true**;

**else**

**return** **false**;

}

}

<https://leetcode.com/problems/implement-queue-using-stacks/>

1- Implement queue using two stacks

**public** **class** MyQueueUsingTwoStacks {

**static** MyStackArr *insertStack*;

**static** MyStackArr *deleteStack* ;

**static** **void** enQueue(**int** x) {

*insertStack*.*push*(x);

}

**static** **int** deQueue() {

**if**(!*deleteStack*.*isEmpty*()) {

**return** *deleteStack*.*pop*();

}

**while**(!*insertStack*.*isEmpty*()) {

*deleteStack*.*push*(*insertStack*.*pop*());

}

**return** *deleteStack*.*pop*();

}

}

2- Implement stack using two queues

**public** **class** MyStackUsingTwoQueues {

**static** MyQueueArr *myQueueInsert*;

**static** MyQueueArr *myQueueDelete*;

**void** push(**int** x) {

*myQueueInsert*.enQueue(x);

}

**void** pop() {

**while** (*myQueueInsert*.getLenght() > 1) {

*myQueueDelete*.enQueue(*myQueueInsert*.deQueue());

}

MyQueueArr temp = *myQueueInsert*;

*myQueueInsert* = *myQueueDelete*;

*myQueueDelete* = temp;

*myQueueDelete*.deQueue();

}

}

3- Implement stack using one queue

**public** **class** MyStackUsingOneQueue {

**static** MyQueueArr *originalQueue*;

**void** push(**int** x) {

*originalQueue*.enQueue(x);

}

**int** pop() {

**int** size = *originalQueue*.getLenght();

**while**(size>1) {

*originalQueue*.enQueue(*originalQueue*.deQueue());

size--;

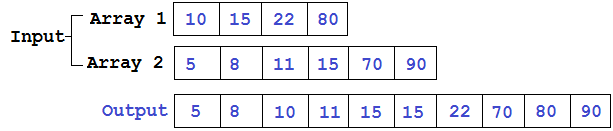
}

**return** *originalQueue*.deQueue();

}

}

4- Merge two sorted arrays



**public** **class** MergingTwoSortedArraysVerbose {

**static** **int**[] merge(**int**[] arr1, **int**[] arr2) {

**int**[] result = **new** **int**[arr1.length + arr2.length];

// Base case 1

**if** (arr1.length == 0 && arr2.length == 0) {

**return** result;

}

// Base case 2

**if** (arr1.length == 0) {

**return** arr2;

}

// Base case 3

**if** (arr2.length == 0) {

**return** arr1;

}

**int** i, j, k;

i = j = k = 0;

**while** (i < arr1.length && j < arr2.length) {

**if** (arr1[i] < arr2[j]) {

result[k] = arr1[i];

i++;

k++;

} **else** **if** (arr1[i] > arr2[j]) {

result[k] = arr2[j];

j++;

k++;

}

}

**while** (i < arr1.length) {

result[k++] = arr1[i++];

}

**while** (j < arr2.length) {

result[k++] = arr2[j++];

}

**return** result;

}

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

**int**[] arr1 = { 1, 3, 5, 7, 9, 20, 30, 40 };

**int**[] arr2 = { 2, 4, 6, 8 };

**int**[] result = *merge*(arr1, arr2);

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

System.***out***.print(result[i] + " ");

}

}

}

**Brute force solution: -**

**TC->O(n^2), SC->O(1)**

**public** **class** IsUnique {

**public** **static** **boolean** isUnique(String input) {

**char**[] charArray = input.toCharArray();

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

**for** (**int** j = i + 1; j < charArray.length; j++) {

**if** (charArray[j] == charArray[i]) {

**return** **false**;

}

}

}

**return** **true**;

}

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

String input = "Helo Hai";

System.***out***.println(input+" "+*isUnique*(input));

}

}

**Efficient Solution: -**

**TC->O(n), SC->O(1)**

**public** **class** IsUniqueEfficient {

**public** **static** **boolean** isUnique(String input) {

**boolean**[] chars = **new** **boolean**[256];

**char**[] charArray = input.toCharArray();

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

**int** value = charArray[i];

**if** (chars[value]) {

**return** **false**;

}

chars[value] = **true**;

}

**return** **true**;

}

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

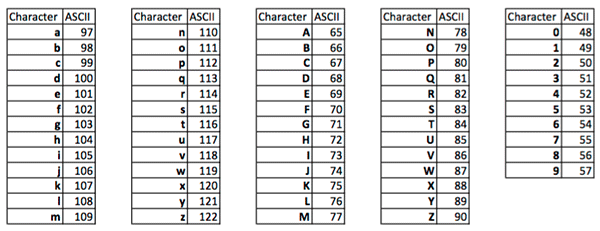
String input = "Helo hai";

System.***out***.println(input+" "+*isUnique*(input));

}

}

**Panagram Anaslysis: -**



**Algorithm**

1. Handle base cases.
   1. null || size == 0 🡪 Invalid input.
   2. size<26 🡪 Not Panagram.
2. Iterate over input.
   1. mark true for every char in input.
3. Iterate over mark array.
   1. if any index is not marked(false) 🡪 not Panagram.
4. String is a Panagram.

**public** **class** CheckAsciiValue {

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

*printAsciiValue*('a');

}

**static** **void** printAsciiValue(**char** ch) {

System.***out***.println((**int**) ch);

}

}

**public** **class** PanagramCheck {

**public** **static** **void** checkPanagram(String input) {

// Base case 1

**if** (input == **null** || input.length() == 0) {

System.***out***.println("Invalid Input");

}

// Base case 2

**if** (input.length() < 26) {

System.***out***.println("Not Panagram");

**return**;

}

**boolean**[] markArray = **new** **boolean**[26];

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

**if** (input.charAt(i) >= 'A' && input.charAt(i) <= 'Z') {// caps

markArray[input.charAt(i) - 'A'] = **true**;

} **else** **if** (input.charAt(i) >= 'a' && input.charAt(i) <= 'z') {// small case

markArray[input.charAt(i) - 'a'] = **true**;

} **else** {// other char

**continue**;

}

}

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

**if**(!markArray[i]) {

System.***out***.println("Not panagram");

**return**;

}

}

System.***out***.println("Panagram");

}

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

*checkPanagram*("The quick brown fox jumps over the lazy dog");

}

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ascii Code** | **Ascii Value** | **Ascii Code** | **Ascii Value** | **Index** | **MarkArray** |
| 65 | **A** | 97 | **a** | 0 |  |
| 66 | **B** | 98 | **b** | 1 |  |
| 67 | **C** | 99 | **c** | 2 |  |
| 68 | **D** | 100 | **d** | 3 |  |
| 69 | **E** | 101 | **e** | 4 |  |
| 70 | **F** | 102 | **f** | 5 |  |
| 71 | **G** | 103 | **g** | 6 |  |
| 72 | **H** | 104 | **h** | 7 |  |
| 73 | **I** | 105 | **i** | 8 |  |
| 74 | **J** | 106 | **j** | 9 |  |
| 75 | **K** | 107 | **k** | 10 |  |
| 76 | **L** | 108 | **l** | 11 |  |
| 77 | **M** | 109 | **m** | 12 |  |
| 78 | **N** | 110 | **n** | 13 |  |
| 79 | **O** | 111 | **o** | 14 |  |
| 80 | **P** | 112 | **p** | 15 |  |
| 81 | **Q** | 113 | **q** | 16 |  |
| 82 | **R** | 114 | **r** | 17 |  |
| 83 | **S** | 115 | **s** | 18 |  |
| 84 | **T** | 116 | **t** | 19 |  |
| 85 | **U** | 117 | **u** | 20 |  |
| 86 | **V** | 118 | **v** | 21 |  |
| 87 | **W** | 119 | **w** | 22 |  |
| 88 | **X** | 120 | **x** | 23 |  |
| 89 | **Y** | 121 | **y** | 24 |  |
| 90 | **Z** | 122 | **z** | 25 |  |

**Trees**

This is how a normal tree looks like

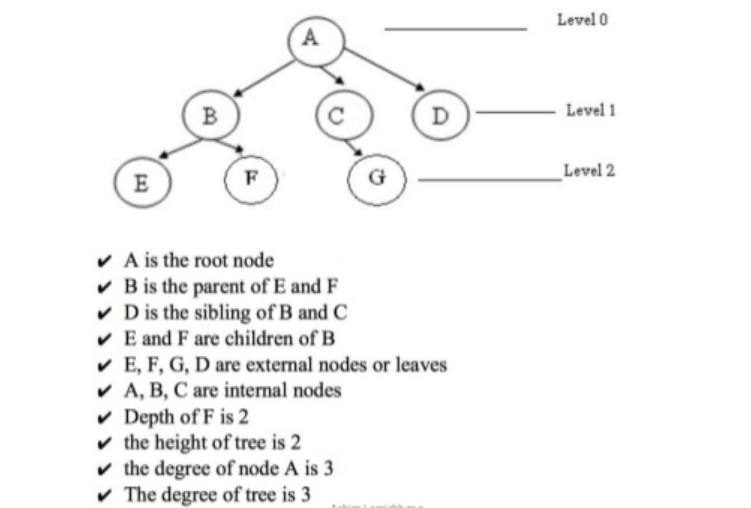


But for programmers, it's all different. It's inverted.



**Terminology: -**

* **Node:** Each element in the tree.
* **Edge:** Line connecting nodes.
* **Parent Node:** Immediate predecessor of a Node.
* **Child Node:** Immediate successor of a Node.
* **Root Node:** Node without parent.
* **Grand Parent Node:** Parent of Parent.
* **Level:** Distance of Node from root.
* **Siblings:** Children of the same parent Node.
* **Degree:** Maximum number of children a node can have.
* **Leaf Node**: Node without children.
* **Height/Depth**: Level of tree counted from root to the lowermost leaf node.
* **External Node**: All Leaf Nodes.
* **Internal Nodes**: All Non-Leaf Nodes.



**Binary Trees: -**

Tree with degree two is called a **Binary Tree.**

**Class Representation of a tree: -**

|  |
| --- |
| **public** **class** **BTNode** {  **int** data;  BTNode[] children = **new** BTNode[degree]; } |

**For a BinaryTree, the degree is 2, so, it can be represented as below: -**

|  |
| --- |
| **public** **class** **BTNode** {  **int** data;  BTNode[] children = **new** BTNode[2]; } |

**BinaryTree can be represented as below to understand easier: -**

|  |
| --- |
| **public** **class** **BTNode**{  **int** data;  Node left;  Node right; } |

**public** **class** PreOrder {

**public** **static** **void** preOrder(Node root) {

// Base condition

**if** (root == **null**) {

**return**;

}

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

*preOrder*(root.left);

*preOrder*(root.right);

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** InOrder {

**public** **static** **void** inOrder(Node root) {

// Base condition

**if** (root == **null**) {

**return**;

}

*inOrder*(root.left);

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

*inOrder*(root.right);

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** PostOrder {

**public** **static** **void** postOrder(Node root) {

// Base condition

**if** (root == **null**) {

**return**;

}

*postOrder*(root.left);

*postOrder*(root.right);

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

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** HeightBTree {

**static** **int** height(Node root) {

// Base Condition

**if** (root == **null**) {

**return** -1;

}

**int** leftHeight = *height*(root.left);

**int** rightHeight = *height*(root.right);

**if** (leftHeight > rightHeight) {

**return** leftHeight + 1;

} **else** {

**return** rightHeight + 1;

}

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** NumberOfNodesInBinaryTree {

**static** **int** getNodesCount(Node root) {

**if** (root == **null**) {

**return** 0;

}

**int** leftCount = *getNodesCount*(root.left);

**int** rightCount = *getNodesCount*(root.right);

**return** leftCount + rightCount + 1;

}

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

Node root = **new** Node(1);

root.left = **new** Node(2);

root.right = **new** Node(3);

root.left.left = **new** Node(4);

root.left.right = **new** Node(5);

System.***out***.println(*getNodesCount*(root));

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** MaximumElementInBinaryTree {

**static** **int** getMaxElement(Node root) {

// Base Condition

**if** (root == **null**) {

**return** Integer.***MIN\_VALUE***;

}

**int** leftMax = *getMaxElement*(root.left);

**int** rightMax = *getMaxElement*(root.right);

**return** Math.*max*(root.data, Math.*max*(leftMax, rightMax));

}

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

Node root = **new** Node(-1);

root.left = **new** Node(-2);

root.right = **new** Node(-3);

root.right.left = **new** Node(-4);

root.right.right = **new** Node(-5);

System.***out***.println(*getMaxElement*(root));

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** NumberOfLeafNodes {

**static** **int** getNumberOfLeafNodes(Node root) {

**if** (root == **null**) {

**return** 0;

}

**if** (root.left == **null** && root.right == **null**) {

**return** 1;

}

**int** leftLeaves = *getNumberOfLeafNodes*(root.left);

**int** rightLeaves = *getNumberOfLeafNodes*(root.right);

**return** leftLeaves + rightLeaves;

}

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

Node root = **new** Node(1);

root.left = **new** Node(2);

root.right = **new** Node(3);

root.right.left = **new** Node(4);

root.right.right = **new** Node(5);

System.***out***.println(*getNumberOfLeafNodes*(root));

}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**public** **class** SearchForElementInBinaryTree {

**static** Node search(Node root, **int** x) {

**if** (root == **null**) {

**return** **null**;

}

**if** (root.data == x) {

**return** root;

}

Node leftSearch = *search*(root.left, x);

**if** (leftSearch != **null**) {

**return** leftSearch;

}

Node rightSearch = *search*(root.right, x);

**if** (rightSearch != **null**) {

**return** rightSearch;

}

**return** **null**;

}

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

Node root = **new** Node(1);

root.left = **new** Node(2);

root.right = **new** Node(3);

root.right.left = **new** Node(4);

root.right.right = **new** Node(5);

System.***out***.println(*search*(root, 2).data);

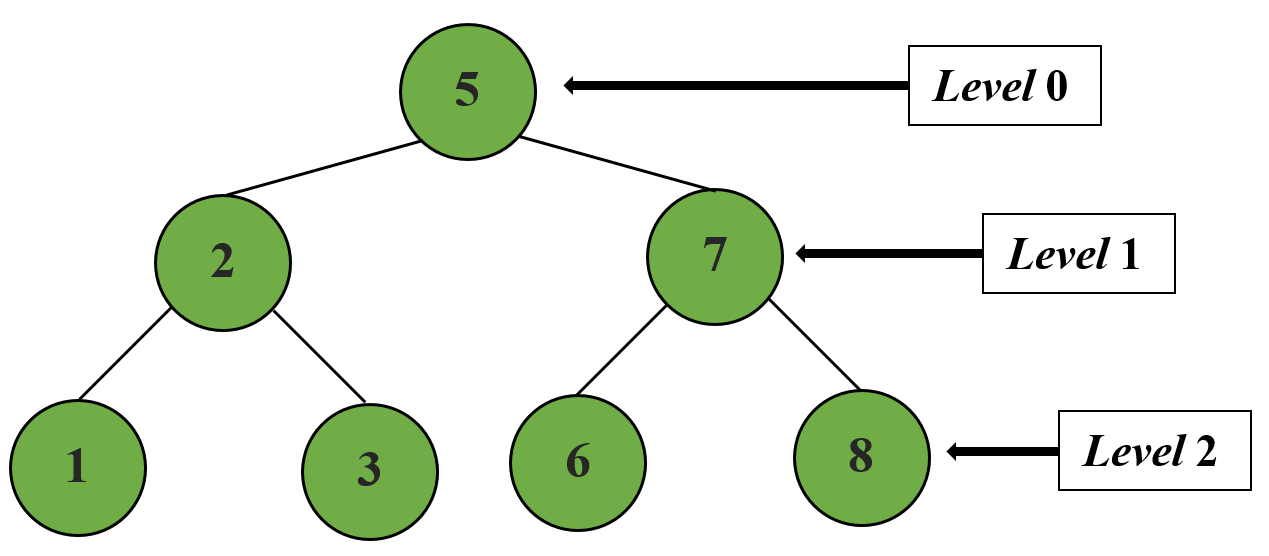
}

}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**Level Order Traversal: -**



**public** **class** LevelOrderTraversal {

**public** **static** **void** printLevel(Node root, **int** level) {

**if** (root == **null**) {

**return**;

}

**if** (level == 1) {

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

}

*printLevel*(root.left, level - 1);

*printLevel*(root.right, level - 1);

}

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

Node root = **new** Node(1);

root.left = **new** Node(2);

root.right = **new** Node(3);

root.right.left = **new** Node(4);

root.right.right = **new** Node(5);

**int** height = HeightBTree.*height*(root);

**for** (**int** i = 1; i <= height; i++) {

*printLevel*(root, i);

}

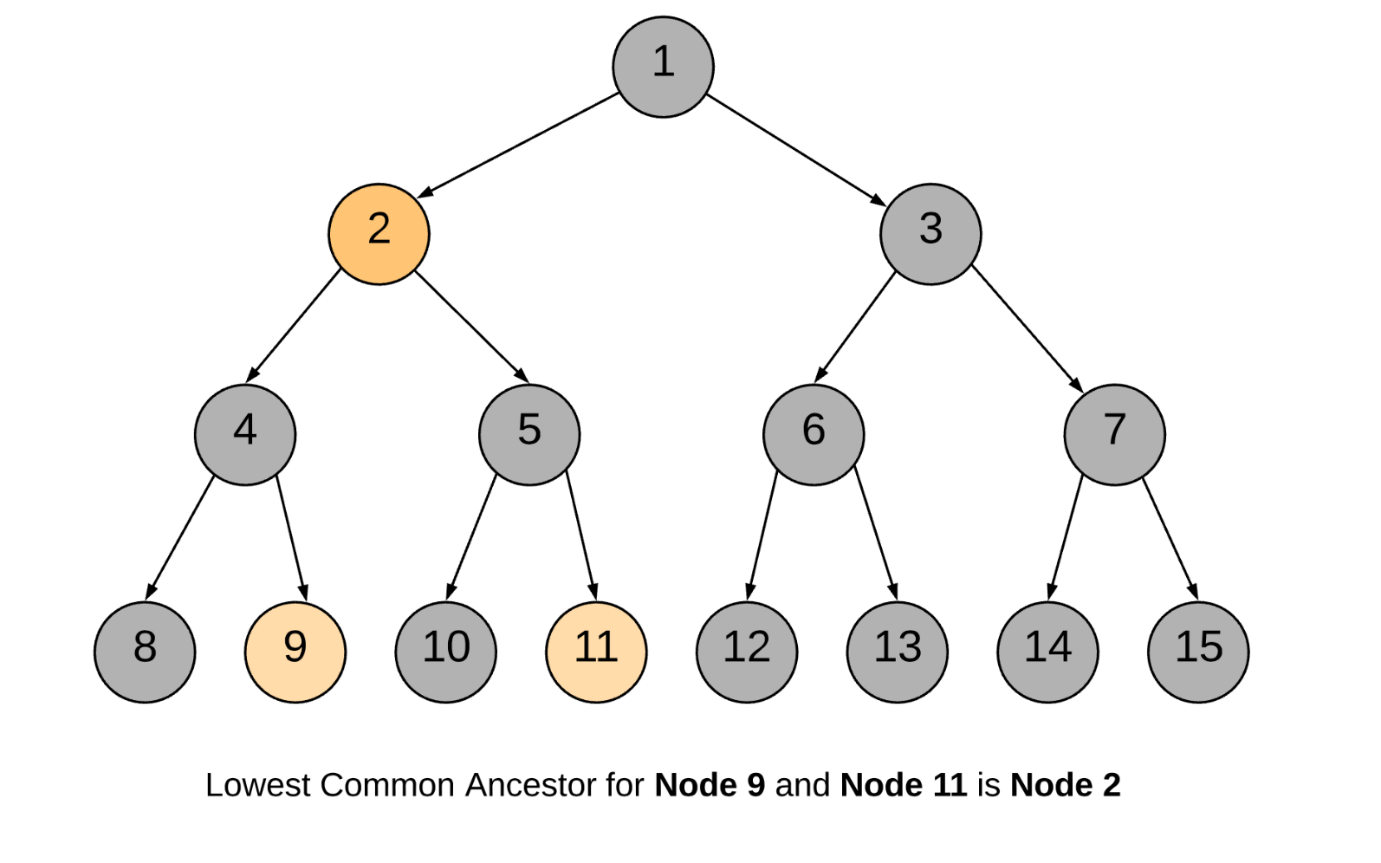
}

}

**TC 🡪 O(n \* h)**

**SC 🡪 O(h)**

**Lowest/Closest Common Ancestor**



**public** **class** LowestCommonAncestor {

**static** Node lca(Node root, **int** node1, **int** node2) {

**if** (root == **null**) {

**return** **null**;

}

**if** (root.data == node1 || root.data == node2) {

**return** root;

}

Node Llca = *lca*(root.left, node1, node2);

Node Rlca = *lca*(root.right, node1, node2);

**if** (Llca == **null** && Rlca == **null**) {

**return** **null**;

}

**if** (Llca == **null**) {

**return** Rlca;

}

**if** (Rlca == **null**) {

**return** Llca;

}

**return** root;

}

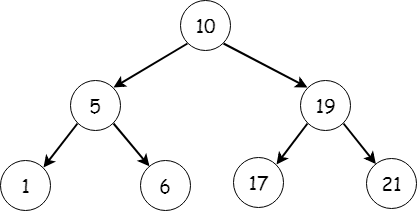
}

**TC 🡪 O(n)**

**SC 🡪 O(h)**

**Binary Search Tree: -**

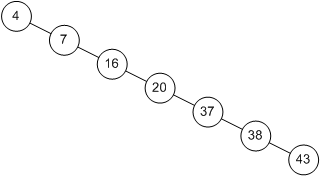
**BST🡪BT+Constraint (Constraint🡪left.data<root.data<right.data)**



**Why BST?**

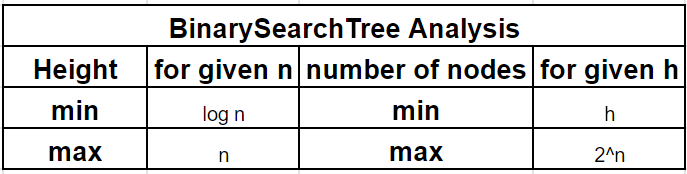
Searching in BT is done in O(n). But, searching is a basic operation and this can be done better. So, the above constraint is introduced so that either left or right subtree is ignored for every iteration. So, the searching can be done in o(h).

**But problem still exists in worst case: -**



So, in worst case, h is still equal to n.

So, the tree needs to be Balanced or Full Binary Tree to get O(log n) time complexity for searching. This can be achieved via rotations:



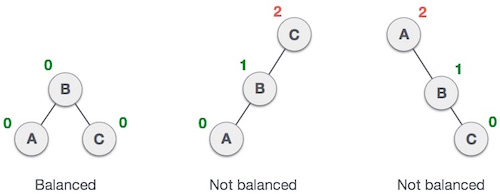
It is observed that BST's worst-case performance is closest to linear search algorithms, that is Ο(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –

**Steps: -**

1. ***Insert***
2. ***Can we do better? (calculate Balance Factor)***
3. ***Rorate.***



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor*** = height(left-subtree) − height(right- subtree)subtree)

***BalanceFactor*** = height(left-subtree) − height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

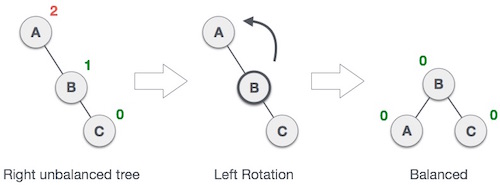
To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

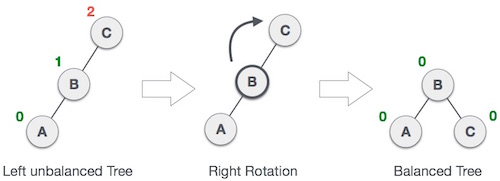
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



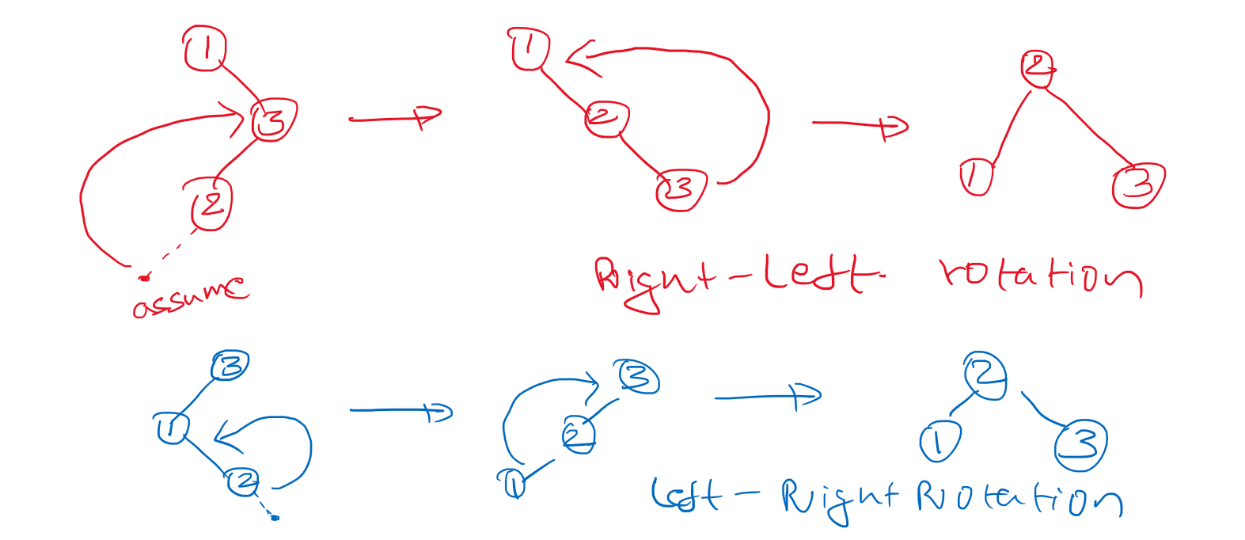
In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



**public** **class** SearchForElementInBinarySearchTree {

**static** Node search(Node root, **int** x) {

**if** (root == **null**) {

**return** **null**;

}

**if** (root.data == x) {

**return** root;

} **else** **if** (root.data < x) {

**return** *search*(root.right, x);

} **else** {

**return** *search*(root.left, x);

}

}

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

Node root = **new** Node(2);

root.left = **new** Node(1);

root.right = **new** Node(4);

root.right.left = **new** Node(3);

root.right.right = **new** Node(6);

System.***out***.println(*search*(root, 3).data);

}

}

**public** **class** InsertBST {

**public** **static** Node insert(Node root, **int** data) {

**if** (root == **null**) {

root = **new** Node(data);

**return** root;

}

**if** (data < root.data) {

root.left = *insert*(root.left, data);

} **else** {

root.right = *insert*(root.right, data);

}

**return** root;

}

}

**public** **class** IsBST {

**public** **boolean** isBST(Node root) {

**if** (root == **null**) {

**return** **true**;

}

**if** (root.left != **null** && root.left.data > root.data) {

**return** **false**;

}

**if** (root.right != **null** && root.right.data > root.data) {

**return** **false**;

}

**return** isBST(root.left) && isBST(root.right);

}

}

**On the below line: -**

**if** (root.left != **null** && root.left.data > root.data)

**it should be: -**

**if** (root.left != **null** && max(root.left.data) > root.data)

So, now the TC will become **O(n^2)**

So, the comparison should not be with children, but it should be with parent.

**public** **class** IsBSTBetter {

**public** **boolean** isBST(Node root) {

**return** isBSTHealper(root, 0, Integer.***MAX\_VALUE***);

}

**public** **boolean** isBSTHealper(Node root, **int** min, **int** max) {

**if** (root == **null**) {

**return** **true**;

}

**if** (root.data <= min || root.data > max) {

**return** **false**;

}

**return** isBSTHealper(root.left, min, root.data) &&

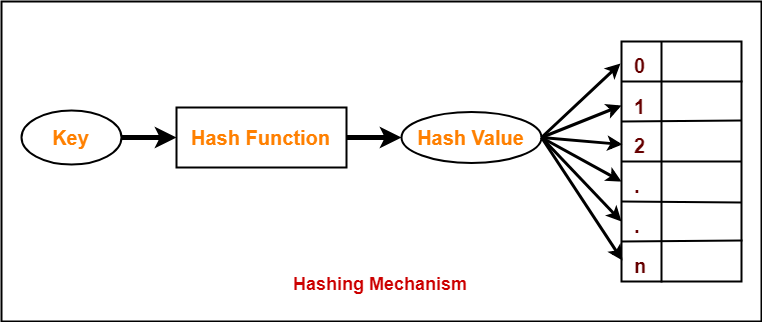
isBSTHealper(root.right, root.data, max);

}

}

**Hashing: -**

Hashing is the process of converting a given key into another value. A **hash function** is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a **hash value** or simply, a **hash**.



Hashing can be used to build, search, or delete from a table.

The basic idea behind hashing is to take a field in a record, known as the **key**, and convert it through some fixed process to a numeric value, known as the **hash key**, which represents the position to either store or find an item in the table. The numeric value will be in the range of 0 to n-1, where n is the maximum number of slots (or **buckets**) in the table.

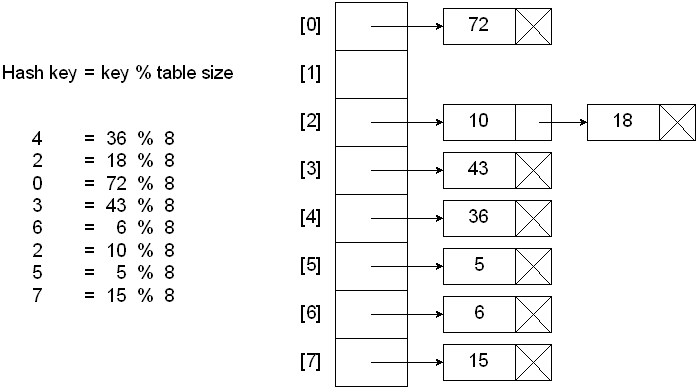
The fixed process to convert a key to a hash key is known as a **hash function**. This function will be used whenever access to the table is needed.

One common method of determining a hash key is the **division method** of hashing. The formula that will be used is:

hash key = key % number of slots in the table



The problem with above is that there may me collisions. In that case, we can do something like below: -



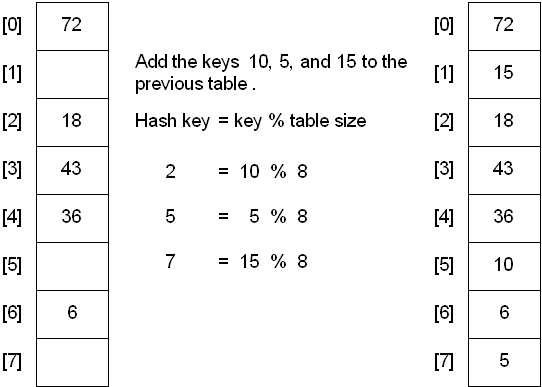
The above is called as Open Addressing. The problem with above that say our numbers are pointing to the same bucket, then the linked list size increases and that the hash table has many empty buckets in it. We can overcome this by using closed addressing.

## Hashing with Linear Probe

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.

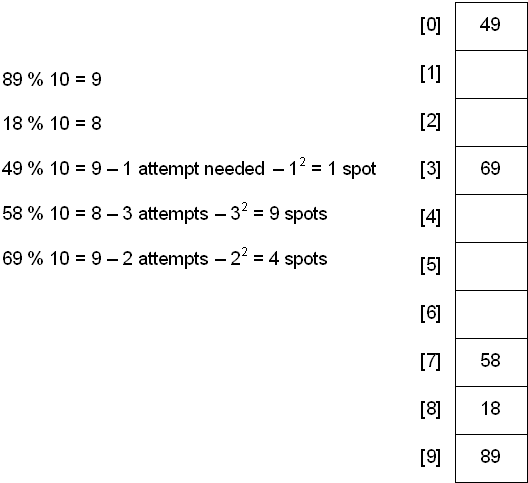
If an empty slot is not found before reaching the point of collision, the table is full.



A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as **primary clustering**.

## Hashing with Quadratic Probe

To resolve the primary clustering problem, **quadratic probing** can be used. With quadratic probing, rather than always moving one spot, move i2 spots from the point of collision, where i is the number of attempts to resolve the collision.



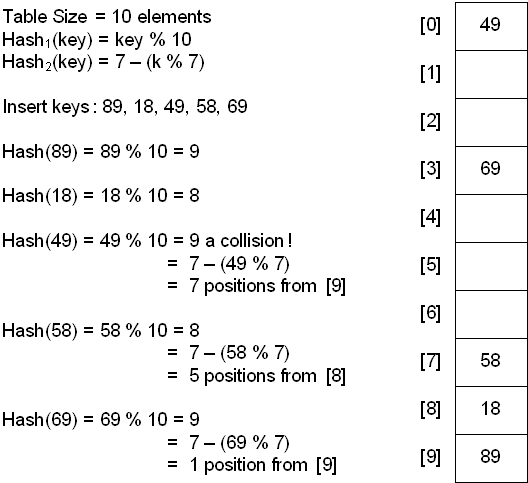
**Hashing with Double Hashing**

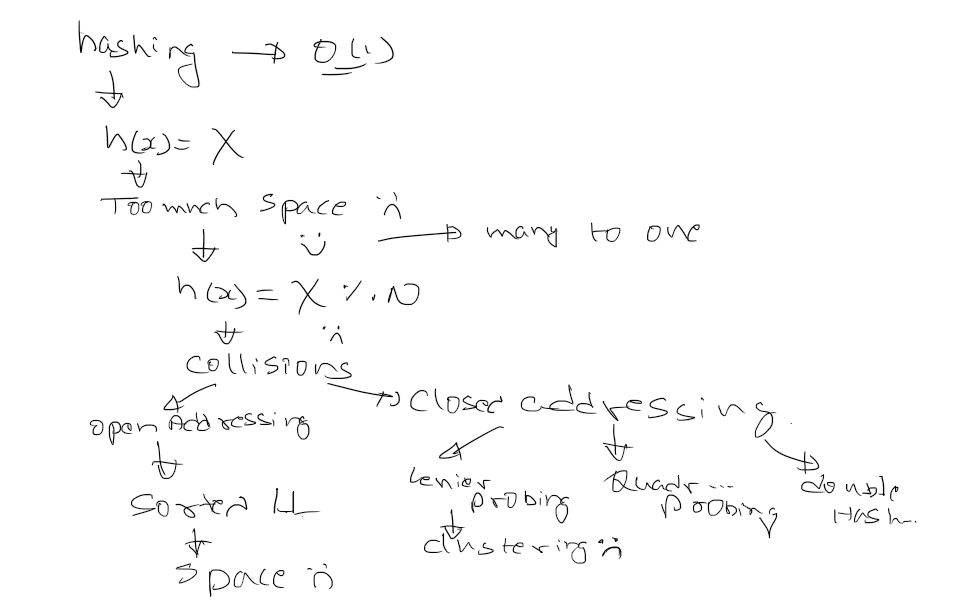
Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

* it must never evaluate to 0
* must make sure that all cells can be probed

A popular second hash function is: Hash2(key) = R - ( key % R ) where R is a prime number that is smaller than the size of the table.



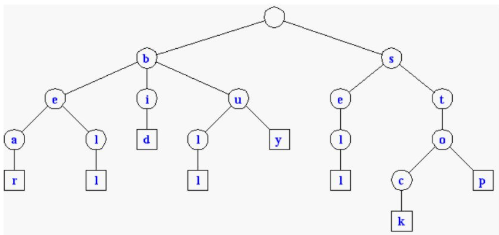


**What is Trie?**

Trie is an efficient information retrieval data structure. A Trie is a tree in which each node has many children. The value at each node consists of 2 things.

1. A character
2. A Boolean to say whether this character represents the end of a word.

Tries are also known as Prefix Trees.



**Representation: -**

**public** **class** TrieNode {

**private** **char** c;

**private** HashMap<Character, TrieNode> children = **new** HashMap<>();

**private** **boolean** isLeaf;

**public** TrieNode() {

}

**public** TrieNode(**char** c) {

**this**.c = c;

}

**public** HashMap<Character, TrieNode> getChildren() {

**return** children;

}

**public** **void** setChildren(HashMap<Character, TrieNode> children) {

**this**.children = children;

}

**public** **boolean** isLeaf() {

**return** isLeaf;

}

**public** **void** setLeaf(**boolean** isLeaf) {

**this**.isLeaf = isLeaf;

}

}

**Basic Operations: -**

**public** **class** Trie {

**private** TrieNode root;

**public** Trie() {

root = **new** TrieNode();

}

**public** **void** insert(String word) {

HashMap<Character, TrieNode> children = root.getChildren();

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

**char** c = word.charAt(i);

TrieNode node;

**if** (children.containsKey(c)) {

node = children.get(c);

} **else** {

node = **new** TrieNode(c);

children.put(c, node);

}

children = node.getChildren();

**if** (i == word.length() - 1) {

node.setLeaf(**true**);

}

}

}

**public** **boolean** search(String word) {

HashMap<Character, TrieNode> children = root.getChildren();

TrieNode node = **null**;

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

**char** c = word.charAt(i);

**if** (children.containsKey(c)) {

node = children.get(c);

children = node.getChildren();

} **else** {

node = **null**;

**break**;

}

}

**if** (node != **null** && node.isLeaf()) {

**return** **true**;

} **else** {

**return** **false**;

}

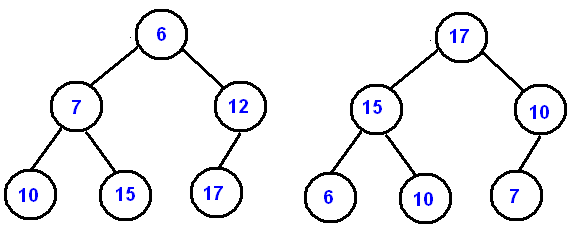
}

}

A **binary heap** is a complete binary tree which satisfies the heap ordering property.

There are two types of heap: -

1. **Min heap:** every parent is less than or equal to its children
2. **Max heap:** Every parent is greater than or equal to its children



A binary heap must be a complete tree, children are added at each level from left to right, and usually implemented as arrays. The maximum or minimum value will always be at the root of the tree, this is the advantage of using a heap.

For Heapify, the process of converting a binary tree into a heap, is often has to be done after an insertion or deletion.

**To implement the heaps as arrays: -**

1. We put the root at array[0]
2. Then we traverse each level from left to right, and so the left child of the root would go into array[1],its right child would be into array[2],etc.

For the node at array[i], we can get left child using this formula **(2i + 1)**, for the right child we can use this one **(2i + 2),** and for parent item **floor((i — 1) / 2)**.This works just with complete binary trees.

**To insert into heap: -**

1. Always add new items to the end of the array
2. Then we have to fix the heap(heapify process)
3. We compare the new item against its parent
4. If the item is greater than its parent, we swap it with its parent
5. We then rinse and repeat

**Implementation: -**

**public** **class** Heap {

**private** **int**[] heap;

**private** **int** size;

**public** Heap(**int** capacity) {

heap = **new** **int**[capacity];

}

**public** **void** insert(**int** value) {

**if** (isFull()) {

**throw** **new** IndexOutOfBoundsException("Heap is full");

}

heap[size] = value;

fixHeapAbove(size);

size++;

}

**private** **void** fixHeapAbove(**int** index) {

**int** newValue = heap[index];

**while** (index > 0 && newValue > heap[getParent(index)]) {

heap[index] = heap[getParent(index)];

index = getParent(index);

}

heap[index] = newValue;

}

**public** **boolean** isFull() {

**return** heap.length == size;

}

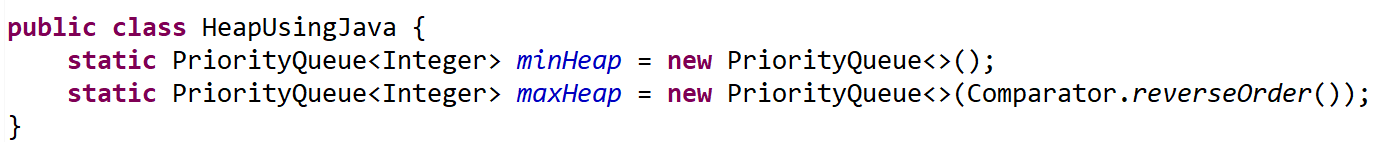
**public** **int** getParent(**int** index) {

**return** (index - 1) / 2;

}

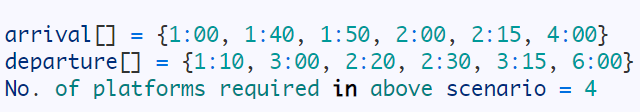
}

**Default implementations in Java: -**

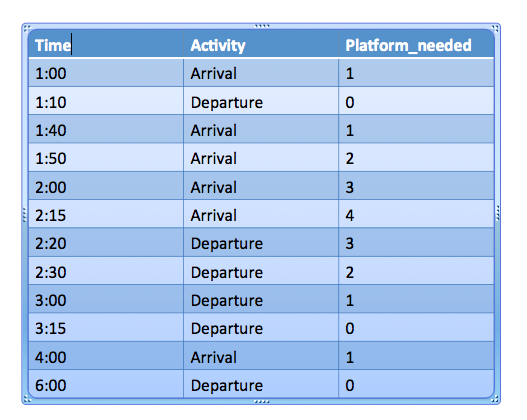


**Problem: -**

You are given arrival and departure time of trains reaching to a particular station. You need to find minimum number of platforms required to accommodate the trains at any point of time.



**Explanation: -**



1. Combine the departure and arrival list
2. Sort the list based on time in increasing order
3. for each element in new list
   1. If arrival time, arrival++
   2. If departure time, departure++
   3. Add (arrival - departure) into new list
4. Return maximum number in new list

**public** **class** TrainTime {

Integer time;

Boolean isArrival;

**public** TrainTime(Integer time, Boolean isArrival) {

**super**();

**this**.time = time;

**this**.isArrival = isArrival;

}

**public** Integer getTime() {

**return** time;

}

**public** **void** setTime(Integer time) {

**this**.time = time;

}

**public** **boolean** isArrival() {

**return** isArrival;

}

**public** **void** setArrival(**boolean** isArrival) {

**this**.isArrival = isArrival;

}

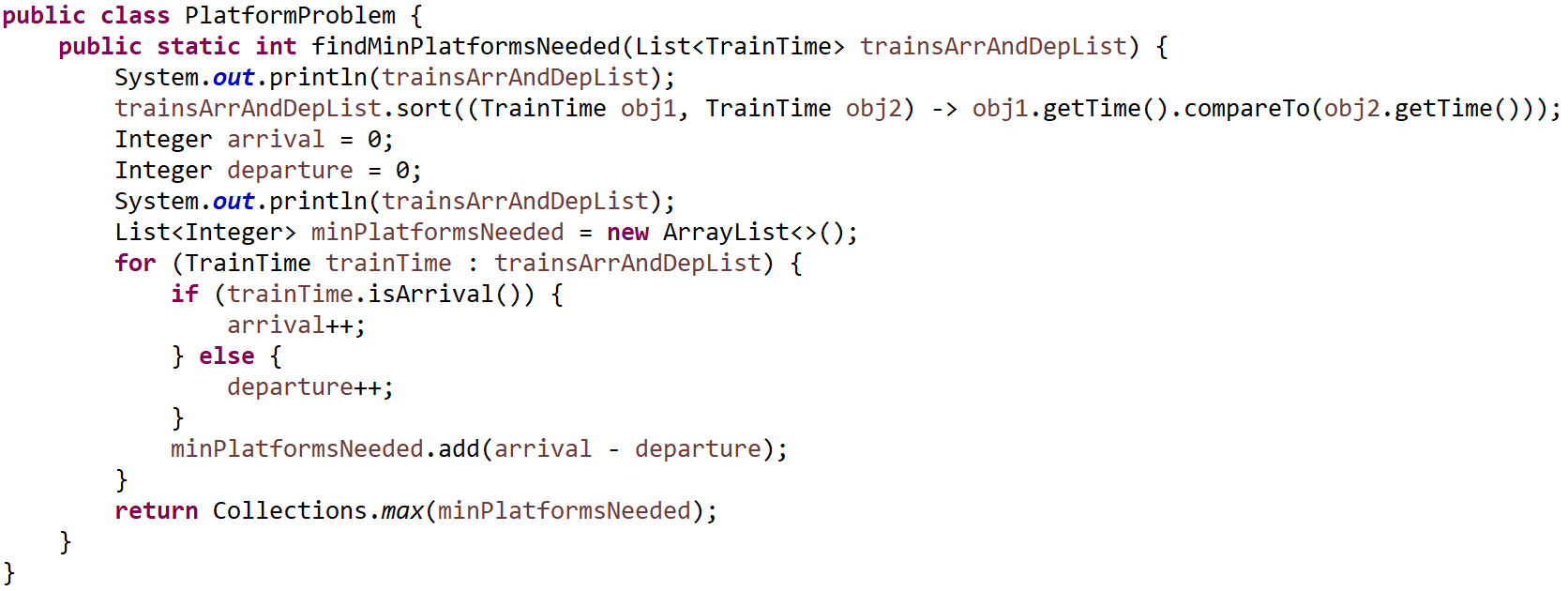
@Override

**public** String toString() {

**return** "\n" + "TrainTime [time=" + time + ", isArrival=" + isArrival + "]";

}

}



**public** **class** PlatformProblemTest {

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

Integer arr[] = { 100, 140, 150, 200, 215, 400 };

Integer dep[] = { 110, 300, 220, 230, 315, 600 };

List<TrainTime> trainTimeList = **new** ArrayList<>();

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

trainTimeList.add(**new** TrainTime(arr[i], **true**));

}

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

trainTimeList.add(**new** TrainTime(dep[i], **false**));

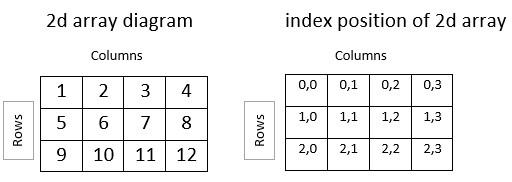
}

**int** minPlatformsNeeded = PlatformProblem.*findMinPlatformsNeeded*(trainTimeList);

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

}

}



**Search in a row wise and column wise sorted matrix**

Given an n x n matrix and a number x, find the position of x in the matrix if it is present in it.

**public** **class** SearchSortedMatrix {

**private** **static** **void** search(**int**[][] matrix, **int** n, **int** x) {

// Base Condition.

**if** (matrix.length == 0) {

System.***out***.print("Invalid Input!");

**return**;

}

**int** i = 0, j = n - 1;

**while** (i < n && j >= 0) {

**if** (matrix[i][j] == x) {

System.***out***.print("Found number at Column: " + i + " Row: " + j);

**return**;

}

**if** (matrix[i][j] > x) {

j--;

} **else** {

i++;

}

}

System.***out***.print("Number not found");

}

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

**int** mat[][] = { { 10, 20, 30, 40 },

{ 15, 25, 35, 45 },

{ 27, 29, 37, 48 },

{ 32, 33, 39, 50 } };

// mat = new int[0][0];

*search*(mat, 4, 29);

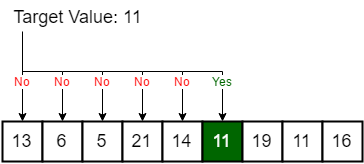
}

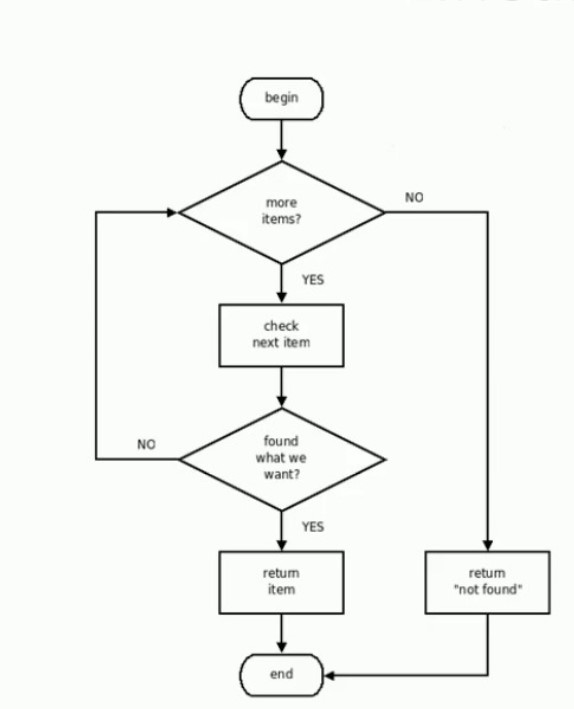
}

**Searching and Sorting Algorithms: -**

1. Searching
   1. Linear Search
   2. Binary Search
2. Sorting
   1. Insertion Sort
   2. Selection Sort
   3. Bubble Sort
   4. Merge Sort
   5. Quick Sort
   6. Counting Sort
   7. Radix Sort
3. Time and Space Complexity Analysis

**Linear Search**





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

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

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

**boolean** result = *linearSearch*(arr, 60);

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**;

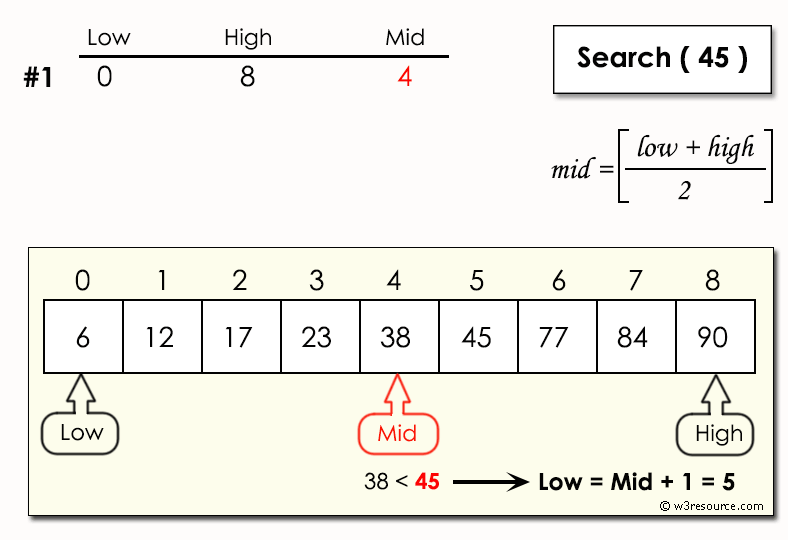
}

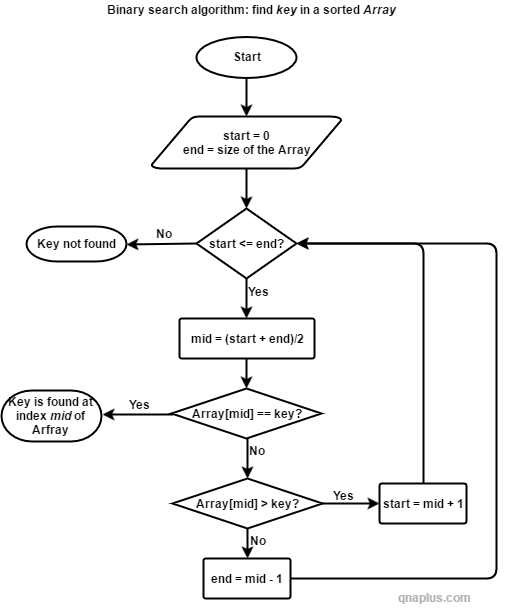
}

**TC: - O(n)**

**SC: - O(1)**

**Binary Search: -**





**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** start, **int** end) {

**while** (start <= end) {

**int** mid = (start + end) / 2;

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

**return** **true**;

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

start = mid + 1;

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

end = mid - 1;

}

}

**return** **false**;

}

}

**public** **class** SortUtil {

**public** **static** **int**[] swap(**int**[] arr, **int** index1, **int** index2) {

**int** temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

**return** arr;

}

}

**public** **class** SortTester {

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

**int**[] inputDataArr = { 6, 4, 2, 1, 3, 5 };

InsertionSortImpl.*insertionSort*(inputDataArr);

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

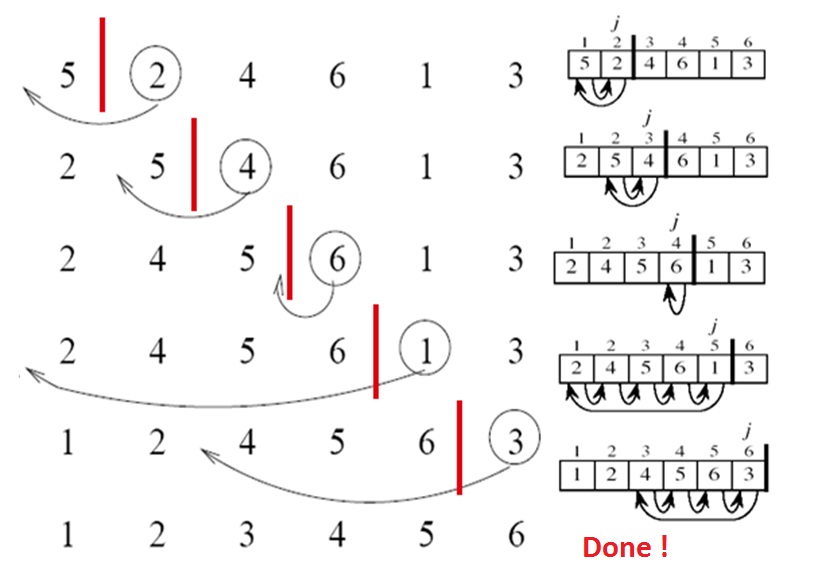
System.***out***.print(inputDataArr[i] + " ");

}

}

}

**Insertion Sort: -**



**public** **class** InsertionSortImpl {

**public** **static** **void** insertionSort(**int**[] inputDataArr) {

**int** length = inputDataArr.length;

**int** temp;

// i=>for cards iteration of cards till end

// j=>where to start?

**for** (**int** i = 1; i < length; i++) {

**for** (**int** j = i; j > 0; j--) {

**if** (inputDataArr[j] < inputDataArr[j - 1]) {

SortUtil.*swap*(inputDataArr, j, j-1);

}

}

}

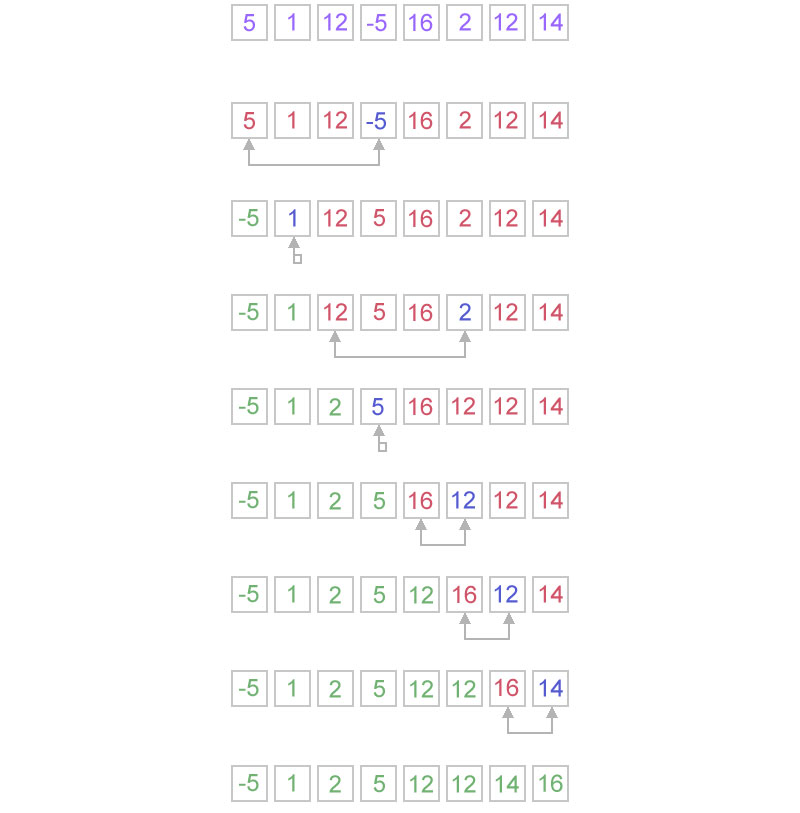
}

}

**TC: - O(n)**

**SC: - O(1)**

**Selection Sort: -**



**public** **class** SelectionSortImpl {

**public** **static** **void** selectionSort(**int**[] inputDataArr) {

**int** length = inputDataArr.length;

**int** smallestIndex;

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

smallestIndex = i;

**for** (**int** j = i + 1; j < length; j++) {

**if** (inputDataArr[smallestIndex] > inputDataArr[j]) {

smallestIndex = j;

}

}

SortUtil.*swap*(inputDataArr, i, smallestIndex);

}

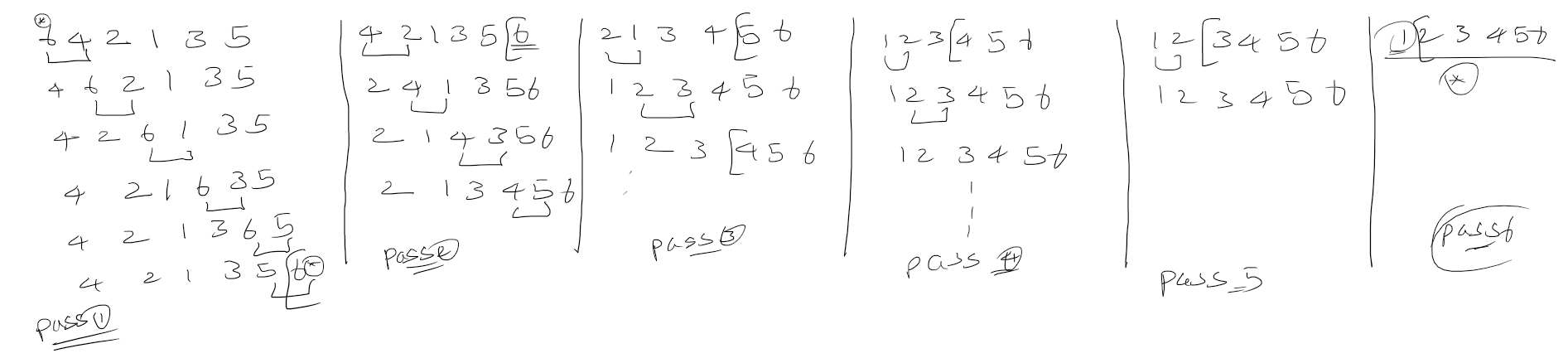
}

}

**TC: - O(n)**

**SC: - O(1)**

**Bubble Sort: -**

****

**public** **class** BubbleSortImpl {

**static** **int**[] bubbleSort(**int**[] inputDataArr) {

**for** (**int** i = 0; i < inputDataArr.length - 1; i++) {

**for** (**int** j = 0; j < inputDataArr.length - i - 1; j++) {

**if** (inputDataArr[j] > inputDataArr[j + 1]) {

SortUtil.*swap*(inputDataArr, j, j + 1);

}

}

}

**return** inputDataArr;

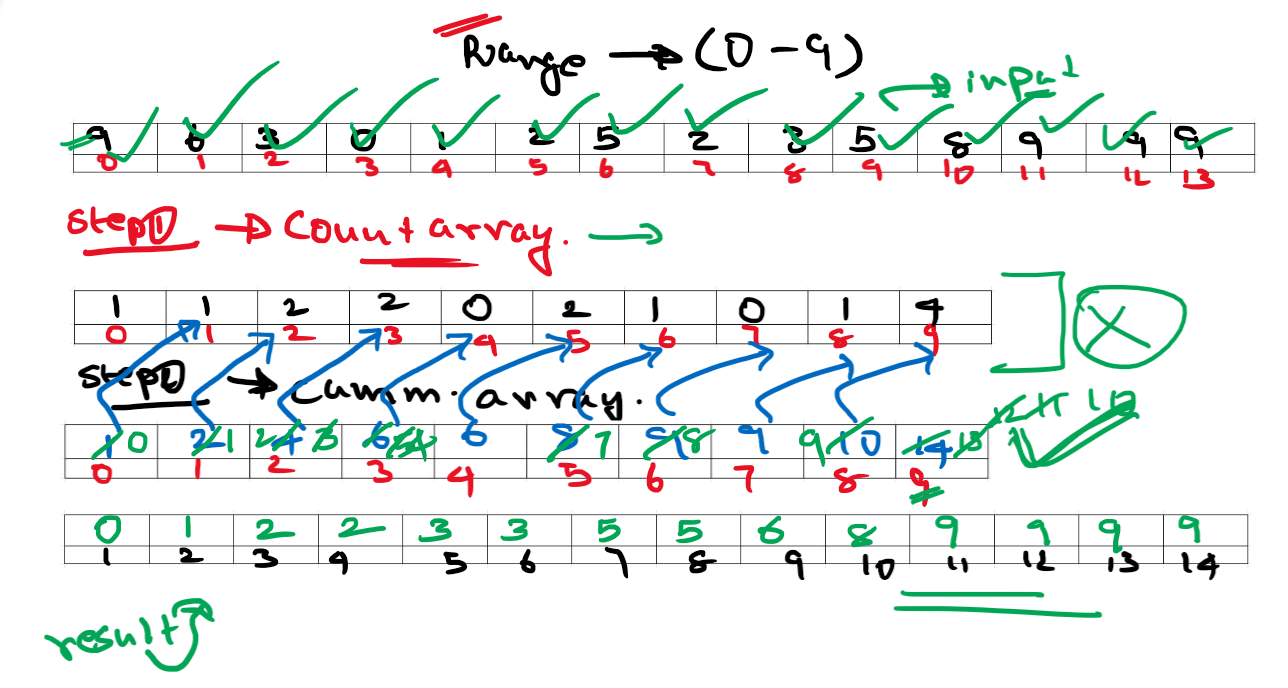
}

}

**TC: - O(n)**

**SC: - O(1)**

**Radix Sort: -**



**public** **class** CountingSort {

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

**int**[] arr = { 9, 6, 3, 0, 1, 2, 5, 2, 3, 5, 8, 9, 9, 9 };

System.***out***.println("Before Sorting!");

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

System.***out***.print(arr[i] + " ");

}

**int**[] countArray = **new** **int**[arr.length];

**int**[] cumulativeArray = **new** **int**[arr.length

**int**[] result = **new** **int**[arr.length];

// Filling countArray

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

countArray[arr[i]]++;

}

// Filling cumulativeArray

cumulativeArray[0] = countArray[0];

**for** (**int** i = 1; i < countArray.length; i++) {

cumulativeArray[i] = cumulativeArray[i - 1] + countArray[i];

}

// Finding where every element in input array should be placed

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

result[cumulativeArray[arr[i]] - 1] = arr[i];

cumulativeArray[arr[i]]--;

}

// Printing result array

System.***out***.println("\n"+"After Sorting!");

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

System.***out***.print(result[i] + " ");

}

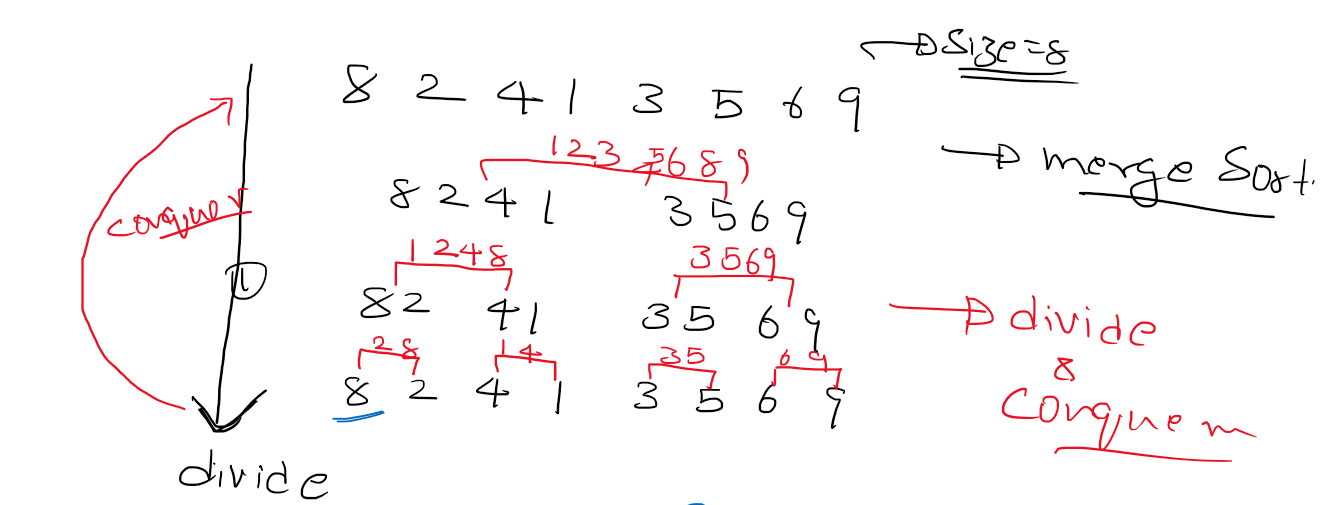
}

}

**TC: - O(n+r) 🡪 n: input size; r: range size**

**SC: - O(n+r) 🡪 n: input size; r: range size**

**Merge Sort: -**

****

**Need to understand the below two to understand Merge sort: -**

1. Recursion
2. Merge two sorted arrays

**public** **class** MergingTwoSortedArraysVerbose {

**static** **int**[] merge(**int**[] arr1, **int**[] arr2) {

**int**[] result = **new** **int**[arr1.length + arr2.length];

// Base case 1

**if** (arr1.length == 0 && arr2.length == 0) {

**return** result;

}

// Base case 2

**if** (arr1.length == 0) {

**return** arr2;

}

// Base case 3

**if** (arr2.length == 0) {

**return** arr1;

}

**int** i, j, k;

i = j = k = 0;

**while** (i < arr1.length && j < arr2.length) {

**if** (arr1[i] < arr2[j]) {

result[k] = arr1[i];

i++;

k++;

} **else** **if** (arr1[i] > arr2[j]) {

result[k] = arr2[j];

j++;

k++;

}

}

**while** (i < arr1.length) {

result[k] = arr1[i];

i++;

k++;

}

**while** (j < arr2.length) {

result[k] = arr2[j];

j++;

k++;

}

**return** result;

}

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

**int**[] arr1 = { 1, 3, 5, 7, 9, 20, 30, 40 };

**int**[] arr2 = { 2, 4, 6, 8 };

**int**[] result = *merge*(arr1, arr2);

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

System.***out***.print(result[i] + " ");

}

}

}

**Better way to code: -**

**public** **class** MergingTwoSortedArrays {

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

**int**[] arr1 = { 1, 3, 5, 7 };

**int**[] arr2 = { 2, 4, 6, 8 };

**int**[] result = **new** **int**[arr1.length + arr2.length];

**int** i, j, k;

i = j = k = 0;

**while** (i < arr1.length && j < arr2.length) {

result[k++] = arr1[i] < arr2[j] ? arr1[i++] : arr2[j++];

}

**while** (i < arr1.length) {

result[k++] = arr1[i++];

}

**while** (j < arr2.length) {

result[k++] = arr2[j++];

}

**for** (**int** k2 = 0; k2 < result.length; k2++) {

System.***out***.print(result[k2] + " ");

}

}

}

**public** **class** MergeSort {

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

**int**[] arr = { 8, 2, 4, 1, 3, 5, 6, 9 };

**int**[] helper = **new** **int**[arr.length];

*mergeSortHelper*(arr, helper, 0, arr.length - 1);

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

System.***out***.print(arr[i] + " ");

}

}

**static** **void** mergeSortHelper(**int**[] arr, **int**[] helper, **int** low, **int** high) {

// Base condition

**if** (low >= high) {

**return**;

}

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

*mergeSortHelper*(arr, helper, low, mid);

*mergeSortHelper*(arr, helper, mid + 1, high);

*merge*(arr, helper, low, mid, high);

}

**static** **void** merge(**int**[] arr, **int**[] helper, **int** low, **int** mid, **int** high) {

// Copy all elements into the helper array for this iteration

**for** (**int** k = low; k <= high; k++) {

helper[k] = arr[k];

}

// This is the only additional assignment you should do and nothing else.

**int** i = low;

**int** j = mid + 1;

**int** k = low;

**while** (i <= mid && j <= high) {

arr[k++] = helper[i] < helper[j] ? helper[i++] : helper[j++];

}

**while** (i <= mid) {

arr[k++] = helper[i++];

}

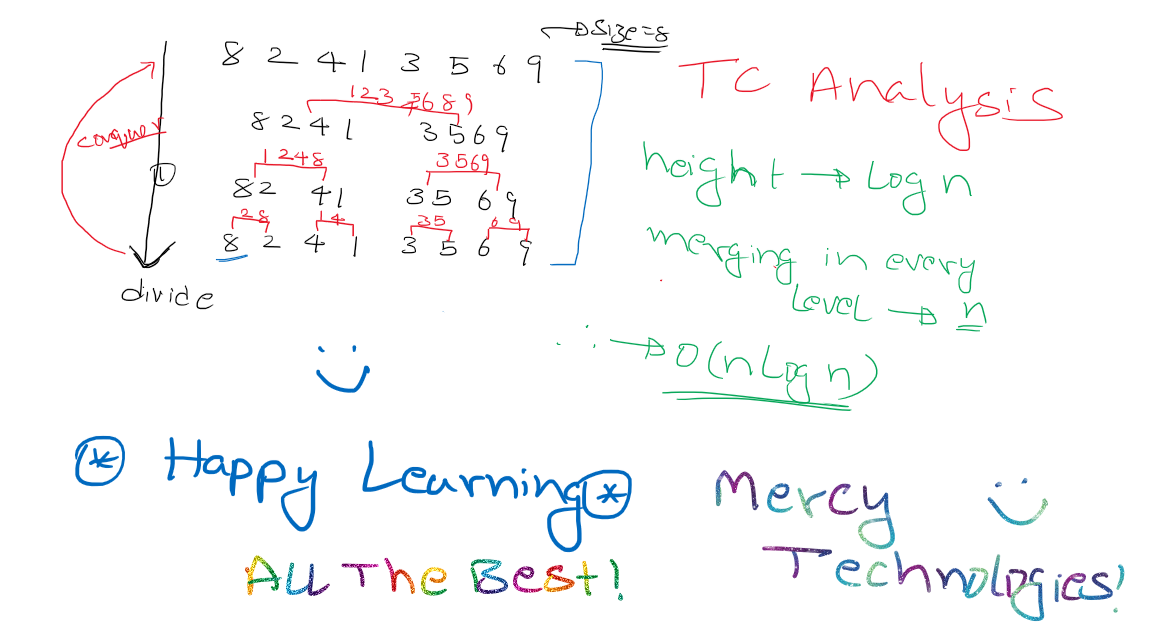
**while** (j <= high) {

arr[k++] = helper[j++];

}

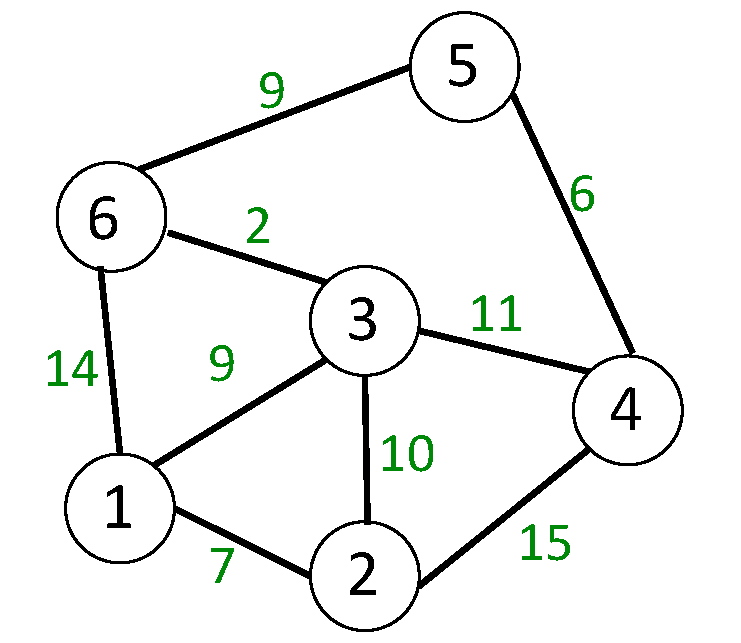
}

}

****

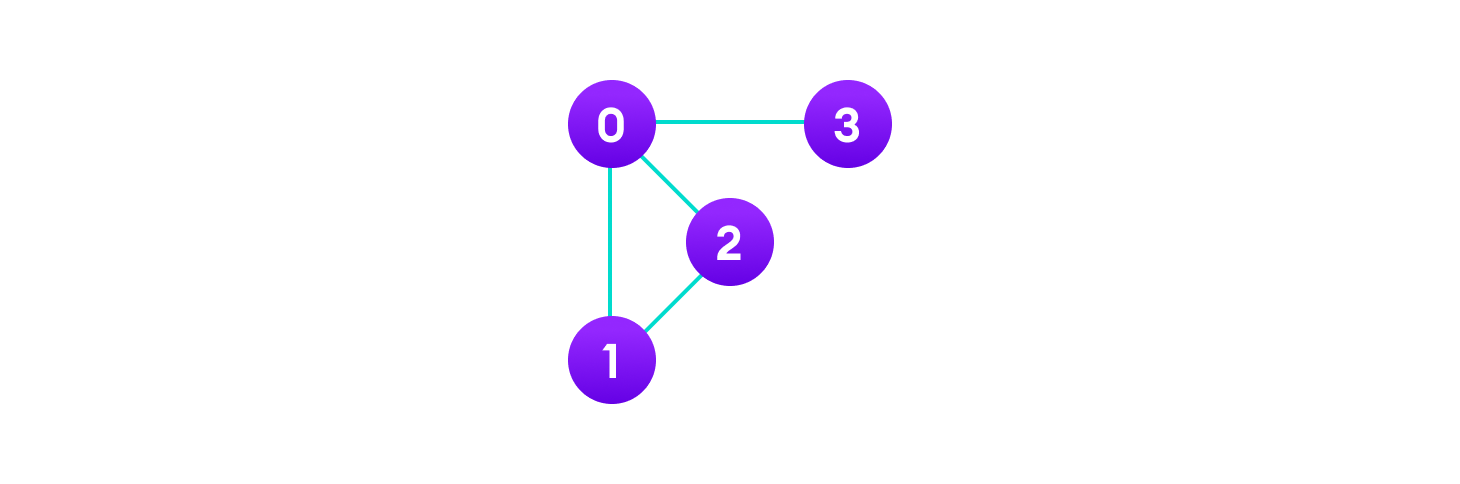
A **graph** is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.



More precisely, a graph is a data structure (V, E) that consists of

* A collection of vertices V
* A collection of edges E, represented as ordered pairs of vertices (u,v)

Vertices and edges in the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

**Graph Terminology**

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge

**Graph Representation**

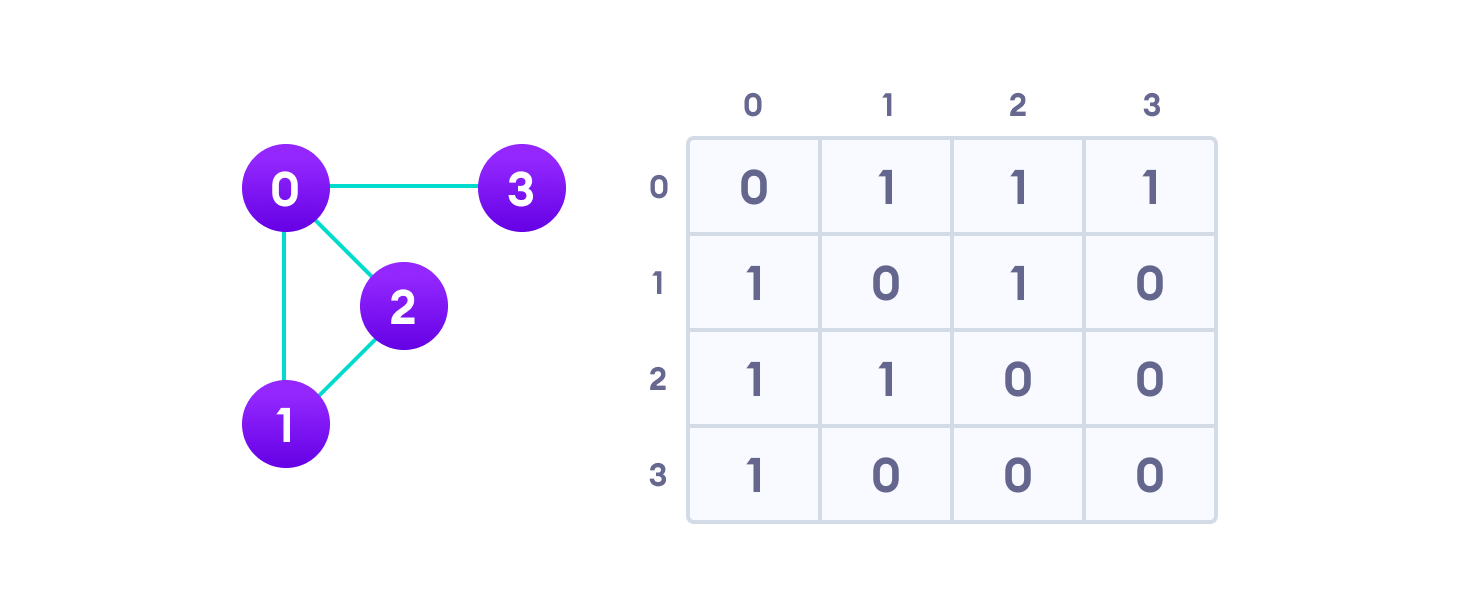
Graphs are commonly represented in two ways:

**1. Adjacency Matrix**

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

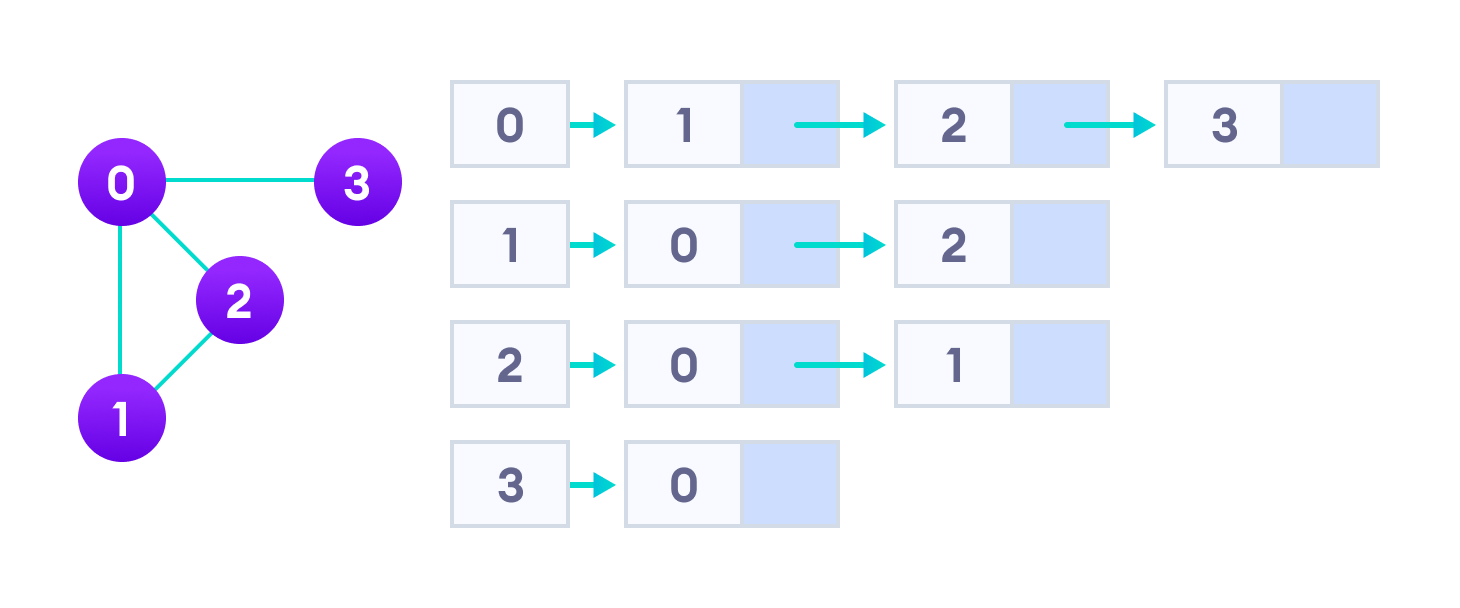
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

### 2. Adjacency List

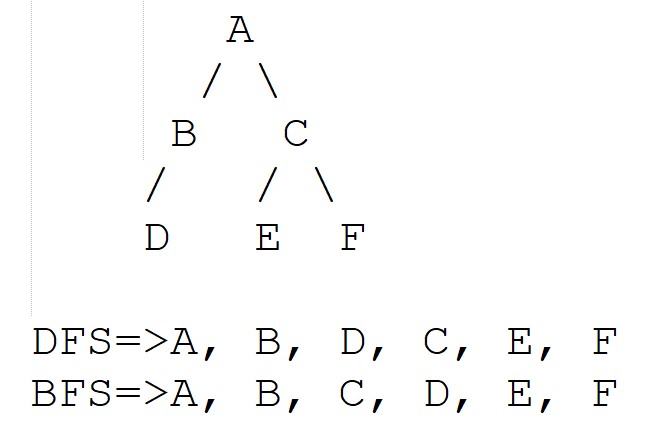
An adjacency list represents a graph as an array of linked lists.

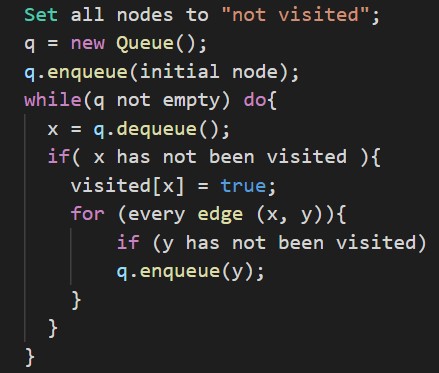
The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.





**public** **class** MyGraph {

**private** **int** numVertices;

**private** LinkedList<Integer> adjLists[];

**private** **boolean** visited[];

MyGraph(**int** vertices) {

numVertices = vertices;

adjLists = **new** LinkedList[vertices];

visited = **new** **boolean**[vertices];

**for** (**int** i = 0; i < vertices; i++)

adjLists[i] = **new** LinkedList<Integer>();

}

**void** addEdge(**int** source, **int** destination) {

adjLists[source].add(destination);

}

**public** **int** getNumVertices() {

**return** numVertices;

}

**public** **void** setNumVertices(**int** numVertices) {

**this**.numVertices = numVertices;

}

**public** LinkedList<Integer>[] getAdjLists() {

**return** adjLists;

}

**public** **void** setAdjLists(LinkedList<Integer>[] adjLists) {

**this**.adjLists = adjLists;

}

**public** **boolean**[] getVisited() {

**return** visited;

}

**public** **void** setVisited(**boolean**[] visited) {

**this**.visited = visited;

}

}

**public** **class** MyBFS {

**static** **void** doBFS(MyGraph g, **int** source) {

**boolean**[] visitedArray = g.getVisited();

// Create a new queue for BFS

LinkedList<Integer> queue = **new** LinkedList<Integer>();

// Mark the current node as visited and enqueue it

visitedArray[source] = **true**;

queue.add(source);

**while** (queue.size() > 0) {

// pop a vertex from queue and print it

source = queue.poll();

System.***out***.print(source + " ");

// Traverse all adj vertices,if not visited, mark visited and enqueue

Iterator<Integer> it = g.getAdjLists()[source].listIterator();

**while** (it.hasNext()) {

**int** curr = it.next();

**if** (!visitedArray[curr]) {

visitedArray[curr] = **true**;

queue.add(curr);

}

}

}

}

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

MyGraph g = **new** MyGraph(6);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

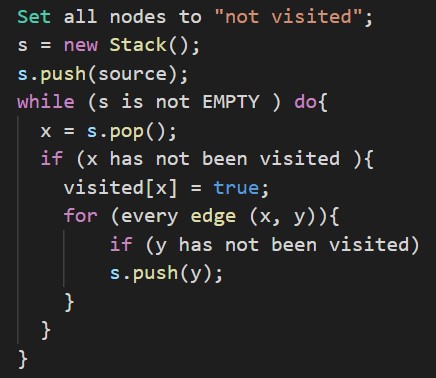
g.addEdge(2, 5);

System.***out***.println("Following is Breadth First Traversal");

*doBFS*(g, 0);

}

}



**public** **class** MyDFS {

**static** **void** doDFS(MyGraph g, **int** source) {

**boolean**[] visitedArray = g.getVisited();

visitedArray[source] = **true**;

System.***out***.print(source + " ");

Iterator itr = g.getAdjLists()[source].listIterator();

**while** (itr.hasNext()) {

**int** adj\_node = (**int**) itr.next();

**if** (!visitedArray[adj\_node])

*doDFS*(g, adj\_node);

}

}

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

MyGraph g = **new** MyGraph(6);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.addEdge(2, 5);

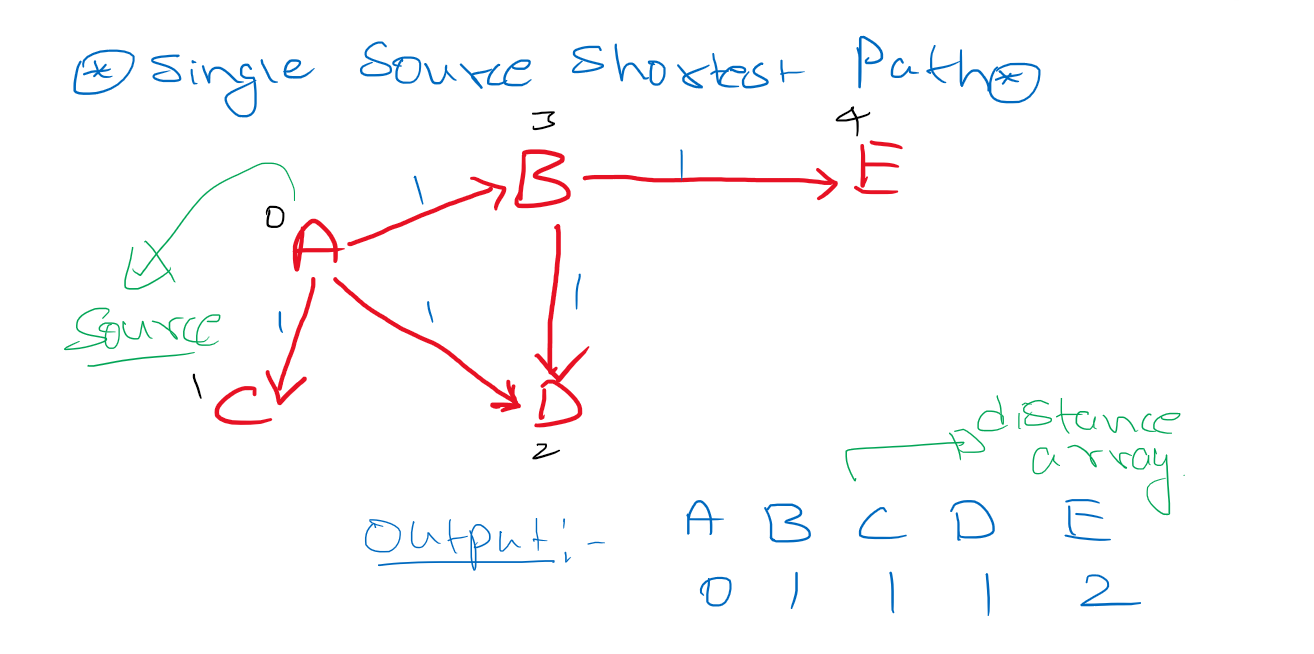
System.***out***.println("Following is Depth First Traversal");

*doDFS*(g, 0);

}

}

**Single Source Shortest Path problem for unweighted graphs: -**

****

**public** **class** MyGraphWithDistance **extends** MyGraph {

**private** **int** distance[];

MyGraphWithDistance(**int** vertices) {

**super**(vertices);

distance = **new** **int**[vertices];

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

distance[i] = -1;

}

}

**public** **int**[] getDistance() {

**return** distance;

}

**public** **void** setDistance(**int**[] distance) {

**this**.distance = distance;

}

}

**public** **class** MySSSP {

**static** **void** getSSSP(MyGraphWithDistance g, **int** source) {

**boolean**[] visitedArray = g.getVisited();

**int**[] distanceArray = g.getDistance();

LinkedList<Integer> queue = **new** LinkedList<Integer>();

visitedArray[source] = **true**;

// Mark distance of source to be 0

distanceArray[source] = 0;

queue.add(source);

**while** (queue.size() > 0) {

// pop a vertex from queue and print it

source = queue.poll();

System.***out***.print(source + " ");

// Traverse all adj vertices,if not visited, mark visited and enqueue

Iterator<Integer> it = g.getAdjLists()[source].listIterator();

**while** (it.hasNext()) {

**int** curr = it.next();

**if** (!visitedArray[curr]) {

visitedArray[curr] = **true**;

distanceArray[curr] = distanceArray[source] + 1;

queue.add(curr);

}

}

}

}

**private** **static** **void** printShortestPath(MyGraphWithDistance g) {

System.***out***.println("\nFollowing is Shortest path using BFS");

**int**[] distanceArray = g.getDistance();

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

System.***out***.print(distanceArray[i] + " ");

}

}

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

MyGraphWithDistance g = **new** MyGraphWithDistance(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(0, 3);

g.addEdge(3, 2);

g.addEdge(3, 4);

System.***out***.println("Following is Breadth First Traversal");

*getSSSP*(g, 0);

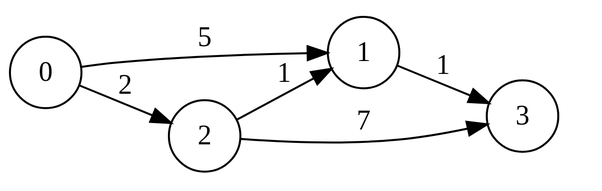
*printShortestPath*(g);

}

}

**Djkstra’s Algorithm: -**

Used to find the shortest path for weighted graphs.



**public** **class** MyWeightedGraph {

**private** **int** numVertices;

**private** LinkedList<GraphNode> adjLists[];

**private** Integer distance[];

MyWeightedGraph(**int** vertices) {

numVertices = vertices;

adjLists = **new** LinkedList[vertices];

distance = **new** Integer[vertices];

**for** (**int** i = 0; i < vertices; i++) {

adjLists[i] = **new** LinkedList<GraphNode>();

distance[i] = Integer.***MAX\_VALUE***;

}

}

**void** addEdge(**int** source, **int** destination, **int** weight) {

adjLists[source].add(**new** GraphNode(source, destination, weight));

}

**public** **int** getNumVertices() {

**return** numVertices;

}

**public** **void** setNumVertices(**int** numVertices) {

**this**.numVertices = numVertices;

}

**public** LinkedList<GraphNode>[] getAdjLists() {

**return** adjLists;

}

**public** **void** setAdjLists(LinkedList<GraphNode>[] adjLists) {

**this**.adjLists = adjLists;

}

**public** Integer[] getDistance() {

**return** distance;

}

**public** **void** setDistance(Integer[] distance) {

**this**.distance = distance;

}

@Override

**public** String toString() {

**return** "MyWeightedGraph [numVertices=" + numVertices + ", adjLists=" + Arrays.*toString*(adjLists) + ", distance="

+ Arrays.*toString*(distance) + "]";

}

}

**public** **class** MyDijkstraImpl {

**static** **void** getSSSPUsingDijkstraAlgo(MyWeightedGraph g, **int** source) {

Integer[] distanceArray = g.getDistance();

// Create a new PQ

PriorityQueue<Integer> pq = **new** PriorityQueue<>();

// Mark distance of source to be 0

distanceArray[source] = 0;

pq.add(source);

**while** (pq.size() > 0) {

// pop a vertex from queue and print it

source = pq.poll();

System.***out***.print(source + " ");

// Traverse all adj vertices,if not visited, mark visited and enqueue

Iterator<GraphNode> it = g.getAdjLists()[source].listIterator();

**while** (it.hasNext()) {

GraphNode curr = it.next();

**if** (distanceArray[curr.getDestination()] > distanceArray[source] + curr.getWeight()) {

distanceArray[curr.getDestination()] = distanceArray[source] + curr.getWeight();

pq.add(curr.getDestination());

}

}

}

}

**private** **static** **void** printShortestPath(MyWeightedGraph g) {

System.***out***.println("\nFollowing is Shortest path using Dijkstra's Algorithm");

Integer[] distanceArray = g.getDistance();

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

System.***out***.print(distanceArray[i] + " ");

}

}

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

MyWeightedGraph g = **new** MyWeightedGraph(4);

g.addEdge(0, 1, 5);

g.addEdge(0, 2, 2);

g.addEdge(1, 3, 1);

g.addEdge(2, 1, 1);

g.addEdge(2, 3, 7);

System.***out***.println("Following is the traversal using Dijkstra's Algorithm");

*getSSSPUsingDijkstraAlgo*(g, 0);

*printShortestPath*(g);

}

}

**public** **class** GraphNode {

**int** source;

**int** destination;

Integer weight;

**public** GraphNode(**int** source, **int** destination, Integer weight) {

**this**.source = source;

**this**.destination = destination;

**this**.weight = weight;

}

**public** **int** getSource() {

**return** source;

}

**public** **void** setSource(**int** source) {

**this**.source = source;

}

**public** **int** getDestination() {

**return** destination;

}

**public** **void** setDestination(**int** destination) {

**this**.destination = destination;

}

**public** Integer getWeight() {

**return** weight;

}

**public** **void** setWeight(Integer weight) {

**this**.weight = weight;

}

@Override

**public** String toString() {

**return** "Edge [source=" + source + ", destination=" + destination + ", weight=" + weight + "]";

}

}

**We have 10 identical bottles of identical pills (each bottle contains 100 of pills). Out of 10 bottles 9 have 1 gram of pills but 1 bottle has pills of weight of 1.1 gram. Given a measurement scale, how would you find the heavy bottle? You can use the scale only once.**

**Answer:**

First, arrange the bottles on shelf and now take, 1 pill from the first bottle, 2 pills from the second bottle, 3 pills from the third bottle, and so on. Ideally you would have **(10)\*(11)/2=55** pills weighing 55 grams, when you put the entire pile of pills on the weighing scale. The deviation from 55 g would tell you which bottle contains the heavy pills.

If it is .1 gram more, it is 1st bottle which has heavy pill, if it is .2 more, gram 2nd bottle has heavy pills, if it is .3 more, gram 3rd bottle has heavy pills.

**How to measure exactly 4 gallons of water from 3 gallon and 5-gallon jars, Given, you have unlimited water supply from a running tap.**

[](http://www.crazyforcode.com/wp-content/uploads/2013/06/water-measure.jpeg)

**Solution:**

Step 1. Fill 3-gallon jar with water. ( 5p – 0, 3p – 3)

Step 2. Pour all its water into 5-gallon jar. (5p – 3, 3p – 0)

Step 3. Fill 3-gallon jar again. ( 5p – 3, 3p – 3)

Step 4. Pour its water into 5-gallon jar until it is full. Now you will have exactly 1-gallon water remaining in 3-gallon jar. (5p – 5, 3p – 1)

Step 5. Empty 5-gallon jar, pour 1-gallon water from 3 gallon jar into it. Now 5-gallon jar has exactly 1 gallon of water. (5p – 1, 3p – 0)

Step 6. Fill 3-gallon jar again and pour all its water into 5-gallon jar**,** thus 5-gallon jar will have exactly 4 gallons of water. (5p – 4, 3p – 0)

**You’ve got someone working for you for seven days and a gold bar to pay him. The gold bar is segmented into seven connected pieces. You must give them a piece of gold at the end of every day. What and where are the fewest number of cuts to the bar of gold that will allow you to pay him 1/7th each day?**

[gold bar puzzle for interview](http://www.crazyforcode.com/wp-content/uploads/2013/07/gold-bar.png)

**Puzzle Solution**:  
Lets split the chain as,  
[](http://www.crazyforcode.com/wp-content/uploads/2013/07/gold-bar-cut.png)  
Day 1: Give A (+1)  
Day 2: Get back A, give B (-1, +2)  
Day 3: Give A (+1)  
Day 4:Get back A and B, give C (-2,-1,+4)  
Day 5:Give A (+1)  
Day 6:Get back A, give B  (-1,+2)  
Day 7:Give A (+1)

**Puzzle:** Four people need to cross a rickety bridge at night. Unfortunately, they have only one torch and the bridge is too dangerous to cross without one. The bridge is only strong enough to support two people at a time. Not all people take the same time to cross the bridge. **Times for each person: 1 min, 2 mins, 7 mins and 10 mins.** What is the shortest time needed for all four  [[](http://www.crazyforcode.com/wp-content/uploads/2013/12/wooden_bridge.jpg)](http://www.crazyforcode.com/wp-content/uploads/2013/12/wooden_bridge.jpg)

It is 17 mins.  
1 and 2 go first, then 1 comes back. Then 7 and 10 go and 2 comes back. Then 1 and 2 go again, it makes a total of 17 minutes.

**Recursion**

* A method/procedure where the solution to a problem depends on solutions to smaller instances of the same problem
* So we break the task into smaller subtasks
* The approach can be applied to many types of problems and recursion is one of the central ideas of computer science
* We have to define base cases in order to avoid infinite loops
* We can solve problems with recursion or with iteration

**Head VS tail recursion**

* If the recursive call occurs at the end of a method 🡪 it is called a tail recursion
* The tail recursion is similar to a loop
* The method executes all the statements before jumping into the next recursive call
* If the recursive call occurs at the beginning of a method, it is called a head recursion.
* The method saves the state before jumping into the next recursive call

**public** **class** RecursionExample {

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

**int** result = *recursionSum*(3);

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

}

**private** **static** **int** recursionSum(**int** n) {

// Base case

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

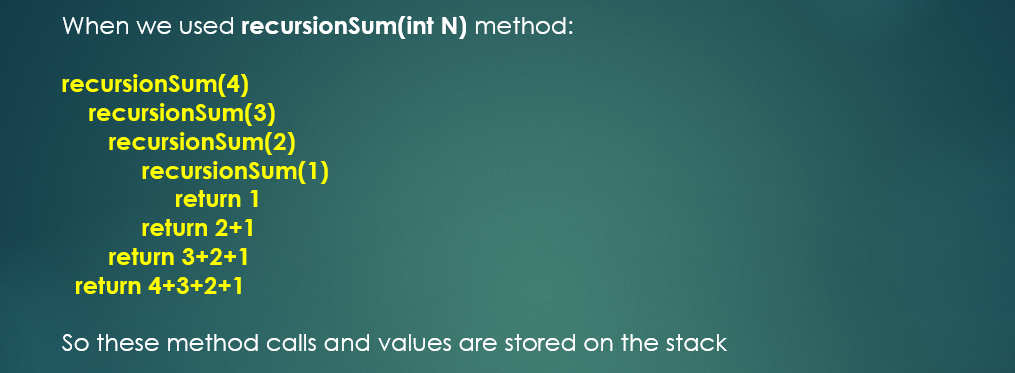
**return** 1;

}

**return** n + *recursionSum*(n - 1);

}

}



Comparing recursive implementation against iterative implementation 🡪 **recursion is at least twice slower because first we unfold recursive calls (pushing them on a stack) until we reach the base case and then we traverse the stack and retrieve all recursive calls**

**Recursion vs Iterative example:**

**public** **class** SumOfNumbers {

**static** **int** *result* = 0;

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

**for** (**int** i = 1; i <= n; i++) {

*result* += i;

}

**return** *result*;

}

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

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

**return** 1;

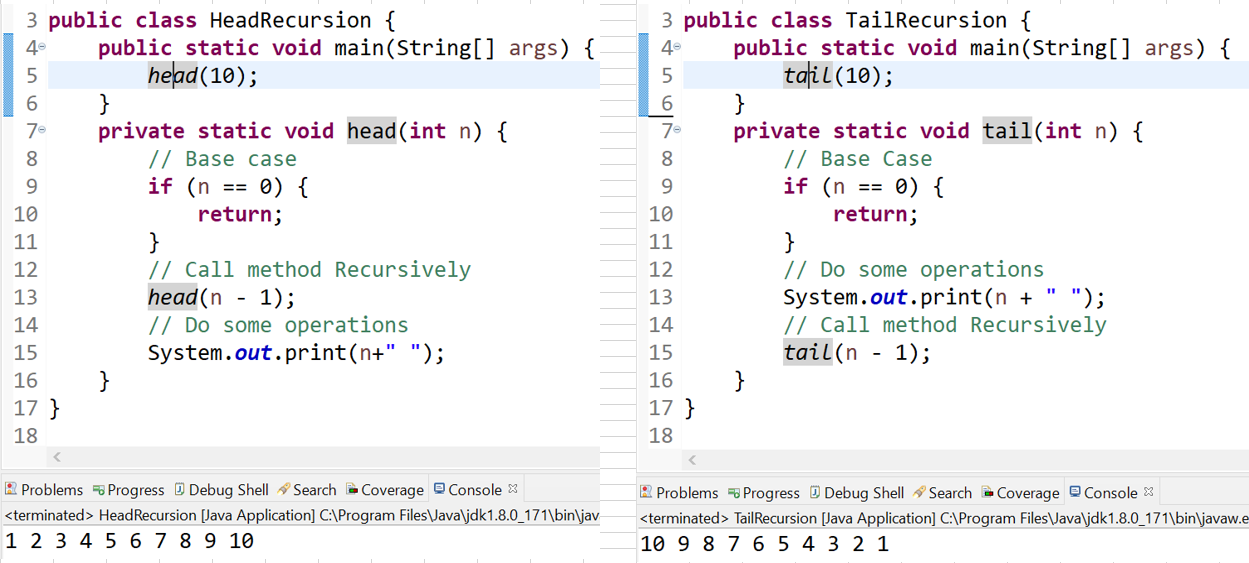
}

**return** n + *sumUsingRecursion*(n - 1);

}

}

In case of Head Recursion, we need to save state and need to backtrack, while in Tail Recursion, we need not save the state. This is just similar to a loop in java.



In head recursion, there is a heavy dependency on the previous function call result. To avoid this, we use a accumulator in tail recursion, so that the dependency on the previous function call result is avoided.

**public** **class** FactorialHeadRecursion {

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

// Base Case

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

**return** 1;

}

// Do some operations

**int** res = *factorial*(n - 1);

**int** result = n \* res;

**return** result;

}

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

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

}

}

**public** **class** FactorialTailRecursion {

**public** **static** **int** factorial(**int** n, **int** accumulator) {

// Base Case

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

**return** accumulator;

}

**return** *factorial*(n - 1, accumulator \* n);

}

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

System.***out***.println(*factorial*(5, 1));

}

}

**public** **class** FibHeadRecursion {

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

System.***out***.println(*fibHead*(10));

}

**private** **static** **int** fibHead(**int** n) {

// Base Case

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

**return** n;

}

// Make recursive calls

**int** fib1 = *fibHead*(n - 1);

**int** fib2 = *fibHead*(n - 2);

// Make some operations

**int** result = fib1 + fib2;

**return** result;

}

}

**In tail recursion, we need to use two accumulators as we have two recursive calls. Initial values for the two accumulators should be 0 and 1 respectively.**

**public** **class** FibTailRecursion {

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

System.***out***.println(*fibTail*(10, 0, 1));

}

**private** **static** **int** fibTail(**int** n, **int** acc1, **int** acc2) {

// Base Case

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

**return** acc1;

}

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

**return** acc2;

}

**return** *fibTail*(n - 1, acc2, acc1 + acc2);

}

}

**Backtracking**

* IT IS A FORM OF RECURSION !!!
* General algorithm for finding all solutions to some computationall problems 🡪  ”constraint satisfaction problems”
* We iincrementally build candidates to the solutions
* If partial candidate A cannot be completed to a valid solution: we abandon A as a solution
* For example: eight-queens problem or sudoku
* Backtracking is often much faster than brute force enumeration of all complete candidates, because it can eliminate a large number of candidates with a single test
* Backtracking is an important tool for solving constraint satisfaction problems 🡪 combinatorial optimization problems!!!
* The partial candidates are represented as the nodes of a tree structure
* *”* ***potential search tree****”*
* Each partial candidate is the parent of the candidates that differ from it by a single extension step
* The leaves of the tree are the partial candidates that cannot be extended any further
* The backtracking algorithm traverses this search tree recursively, from the root down (like DFS)
* This is why backtracking is sometimes called depth-first search !!!
* For every node the algorithm checks whether the given node can be completed to a valid solution
* If it can not 🡪 the whole subtree is skipped !!!
* Recursively enumerates all subtree of the node

**Brute force vs Backtracking:**

