**IF, ELSE, CONTINUE, FOR LOOP, WHILE LOOP**

IF:

If statement checks condition and allows to compile set of blocks

ELSE:

Else is executed when if condition is failed.

*if (condition)*

*{*

*// Executes this block if*

*// condition is true*

*}*

*else*

*{*

*// Executes this block if*

*// condition is false*

*}*

CONTINUE:

Continue statement breaks one iteration(in the loop).

*continue;*

FOR LOOP & WHILE LOOP:

When you know exactly how many times you want to loop through a block of code, use the  for loop instead of a while loop:

*for (initialization expr; test expr; update exp)*

*{*

*// body of the loop*

*// statements we want to execute*

*}*

**while loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

*while (test\_expression)*

*{*

*// statements*

*update\_expression;*

*}*

CODE IS BELOW

*public class Solution*

*{*

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

*{*

*int a=10;*

*int b=139;*

*if(a==b)// checks whether the given conditon is true or not*

*System.out.println("Both are equal");*

*else// if if condition is failed then else lock statements are executed*

*System.out.println("Both are not equal");*

*for(int i=0;i<10;i++)// for loop*

*{*

*if(i==5)*

*continue;//continue statement breaks one iteration (in the loop)*

*else*

*System.out.println(i);*

*}*

*int i=0;*

*while(i<10)*

*{*

*if(i==5)*

*break;// break statement is used to terminate the running loop*

*System.out.println(i);*

*}*

*int c=3;*

*switch (c)//switch statement is with arguments checks the respective case*

*{*

*case 1:// checks for case if it works excutes the below statement*

*System.out.println("case is 1");*

*break;*

*case 2:*

*System.out.println("case is 2");*

*break;*

*case 3:*

*System.out.println("case is 3");*

*break;*

*default://if no case matches then default case is executed*

*System.out.println("This is default case");*

*}*

*}*

*}*

**Closures**

A method in method is known as closures. These are similar to lamda functions in java.

*(argument\_list) -> {func\_body}*

*interface Example*

*{*

*public String salHello();*

*}*

*class Solution*

*{*

*// Driver code*

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

*{*

*// Lambda Expression*

*Example obj = () ->*

*{*

*return "Hello, World!";*

*};*

*// Calling the above interface*

*System.out.println(obj.salHello());*

*}*

*}*

**Recurssion**

The process in which a function calls itself directly or indirectly is called recursion and this function is called as recursive function.

*import java.util.Scanner*;

*public class Recurrsion*

*{*

*static void recur(int n)*

*{*

*if(n==1)// base condition*

*{*

*System.out.println(n);*

*return ;*

*}*

*recur(n-1);//recurrsive function call*

*System.out.println(n);*

*}*

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

*{*

*Scanner obj=new Scanner(System.in);*

*int n=obj.nextInt();*

*//printing first n numbers through recurssion*

*recur(n);*

*}*

*}*

BinarySearch Tree

Binary search tree is a binary tree in which the value of all the left child nodes is less than or equal to the value of its parent node, and the value of all the right child nodes is greater than or equal to the value of its parent node.

*import java.util.\*;*

*public class BinarySearchTree*

*{*

*public static class Node*

*{*

*int data;*

*Node left;*

*Node right;*

*public Node(int data)*

*{*

*this.data = data;*

*this.left = null;*

*this.right = null;*

*}*

*}*

*public Node root;*

*public BinarySearchTree()*

*{*

*root = null;*

*}*

*public void insert(int data) //insert() function will add new node to the binary search tree*

*{*

*Node newNode = new Node(data);//Create a new node*

*if(root == null)//Check whether tree is empty*

*{*

*root = newNode;*

*return;*

*}*

*else*

*{*

*Node node = root, head = null; //node node point to root of the tree*

*while(true)*

*{*

*head = node;//head keep track of the head node of node node.*

*if(data < node.data) //If data is less than node's data, node will be inserted to the left of tree*

*{*

*node = node.left;*

*if(node == null)*

*{*

*head.left = newNode;*

*return;*

*}*

*}*

*//If data is greater than node's data, node will be inserted to the right of tree*

*else*

*{*

*node = node.right;*

*if(node == null)*

*{*

*head.right = newNode;*

*return;*

*}*

*}*

*}*

*}*

*}*

*//minNode() will find out the minimum node*

*public Node minNode(Node root)*

*{*

*if (root.left != null)*

*return minNode(root.left);*

*else*

*return root;*

*}*

*public Node deleteNode(Node node, int value)//deleteNode() function will delete the given node from the binary search tree*

*{*

*if(node == null)*

*return null;*

*else*

*{*

*if(value < node.data)//value is less than node's data then, search the value in left subtree*

*node.left = deleteNode(node.left, value);*

*else if(value > node.data)//value is greater than node's data then, search the value in right subtree*

*node.right = deleteNode(node.right, value);*

*else //If value is equal to node's data that is, we have found the node to be deleted*

*{*

*if(node.left == null && node.right == null)//If node to be deleted has no child then, set the node to null*

*node = null;*

*else if(node.left == null)//If node to be deleted has only one right child*

*node = node.right;*

*else if(node.right == null) //If node to be deleted has only one left child*

*node = node.left;*

*else //If node to be deleted has two children node*

*{*

*Node temp = minNode(node.right);//then find the minimum node from right subtree*

*node.data = temp.data;//Exchange the data between node and temp*

*node.right = deleteNode(node.right, temp.data);//Delete the node duplicate node from right subtree*

*}*

*}*

*return node;*

*}*

*}*

*public void inorder(Node node) //inorder() will perform inorder traversal on binary search tree*

*{*

*if(root == null)//Check whether tree is empty*

*return;*

*else*

*{*

*if(node.left!= null)*

*inorder(node.left);*

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

*if(node.right!= null)*

*inorder(node.right);*

*}*

*}*

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

*{*

*Scanner obj=new Scanner(System.in);*

*int n=obj.nextInt();// takes number of nodes as input*

*BinarySearchTree bt = new BinarySearchTree();*

*for(int i=0;i<n;i++)//Add nodes to the binary tree*

*bt.insert(obj.nextInt());*

*System.out.println("Binary search tree after insertion:");*

*//Displays the binary tree*

*bt.inorder(bt.root);*

*int x=obj.nextInt();// takes input of a node to delete*

*Node deletedNode = null;*

*//Deletes node x which has no child*

*deletedNode = bt.deleteNode(bt.root, x);*

*System.out.println("\nBinary search tree after deleting node");*

*bt.inorder(bt.root);*

*}*

*}*

**7.2 Implementation of getElementByID**

*public static Element getElementById(Document doc, String id)*

*{*

*NodeList nodeList = doc.getElementsByTagName("\*");*

*for (int i = 0; i &lt; nodeList.getLength(); i++)*

*{*

*Node node = nodeList.item(i);*

*if (node.getNodeType() == Node.ELEMENT\_NODE)*

*{*

*Element element = (Element) node;*

*if (element.hasAttribute("id"))*

*{*

*if (element.getAttribute("id").equals(id))*

*return element;*

*}*

*}*

*}*

*return null;*

}

**8.1 Check wheather the number(x) is prime or not and difference between next prime number and x.**

*function isPrime(x)*

*{*

*if (x <= 1)*

*return false;*

*for (var i=2;i\*i<= x; i++)*

*if (x % i == 0)*

*return false;*

*return true;*

*}*

*function nextPrime(x)*

*{*

*if (x<= 2)*

*return 2;*

*if (x % 2 == 0)*

*x++;*

*while (!isPrime(x))*

*{*

*x += 2;*

*}*

*return x;*

*}*

*var x=prompt();*

*if(isPrime(x))*

*console.log(x+" is Prime");*

*else*

*console.log(x+" is not Prime");*

*var y=nextPrime(x);*

*console.log("The difference between "+x+" and next prime number "+y+" is "+(y-x));*