Programs

**package** com.test.program.level1;

**public** **class** BinaryNumberOrNotExample {

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

**int** n = 11001011;

System.***out***.println(*isBinaryNumber*(n) ? n + " is binary number." : n + " is not binary number.");

n = 11001110;

System.***out***.println(*isBinaryNumber*(n) ? n + " is binary number." : n + " is not binary number.");

n = 10000112;

System.***out***.println(*isBinaryNumber*(n) ? n + " is binary number." : n + " is not binary number.");

n = 11005110;

System.***out***.println(*isBinaryNumber*(n) ? n + " is binary number." : n + " is not binary number.");

}

/\*\*

\* returns true if number is binary.

\*/

**public** **static** **boolean** isBinaryNumber(**int** n) {

**while** (n != 0) {

**if** (n % 10 > 1) {

**return** **false**; // number containing any digit greater than 1 means its not binary.

}

n = n / 10;

}

**return** **true**;

}

}

**package** com.test.program.level1;

**public** **class** FactorialExample {

**public** **static** **void** main(String... args) {

**int** num = 5;

System.***out***.println("Factorial of " + num + " is: " + *findFactorail*(num));

}

/\*

\* return factorial of num.

\*/

**public** **static** **int** findFactorail(**int** num) {

/\*

\* int factorial=1; while(num>0){ factorial=factorial\*num; num--; } return

\* factorial;

\*/

**int** fact = 1;

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

fact = fact \* i;

}

**return** fact;

}

}

**package** com.test.program.level1;

**public** **class** FibonacciSeriesExampleInJava {

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

**int** n=10; //number of elements in series.

*generateFibonacciSeries*(n);

}

**public** **static** **void** generateFibonacciSeries(**int** n){

/\* int first=0; //first number of series

int second=1; //second number of series

int temp;

System.out.print("FibonacciSeries: "+ first+" "+second+" ");

for(int i=0;i<n;i++){

temp=first+second;

first=second;

second=temp;

System.out.print(temp+" ");

}\*/

**int** s =0;

**int** a=0;

**int** b=1;

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

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

s=a+b;

a=b;

b=s;

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

}

}

}

**package** com.test.program.level1;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FindOnlyDuplicateNumberInListExample {

**public** **static** **void** main(String...a){

List<Integer> list = **new** ArrayList<Integer>();

**int** highestNumberInList=10; //you may change it in future.

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

list.add(i);

}

list.add(5); //add duplicate number in list(please ensure you don't add more than one duplicate number in list)

System.***out***.println("list is: "+list);

System.***out***.println("Only duplicate number in list is: "+*findOnlyDuplicateNumberInList*(list,highestNumberInList));

}

/\*\*

\* returns only duplicate number in list.

\*/

**public** **static** **int** findOnlyDuplicateNumberInList(List<Integer> list, **int** highestNumberInList){

**int** sumOfNumbersList = 0;

**for**(**int** n:list){

sumOfNumbersList =sumOfNumbersList+n;

}

**int** onlyDuplicateNumberInList = sumOfNumbersList - (((highestNumberInList)\*(highestNumberInList+1))/2);

**return** onlyDuplicateNumberInList;

}

}

**package** com.test.program.level1;

**public** **class** FindTwoMaximumNumbersInArrayExample {

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

**int** ar[] = { 66, 7, 9, 70, 89, 11, 2, 93 };

**int** maximum1 = 0;

**int** Maximum2 = 0;

**for** (**int** i = 0; i < ar.length; i++) { // only one iteration

**if** (maximum1 < ar[i]) {

Maximum2 = maximum1;

maximum1 = ar[i];

} **else** **if** (Maximum2 < ar[i]) {

Maximum2 = ar[i];

}

}

System.***out***.println("Maximum1 : " + maximum1);

System.***out***.println("Maximum2 : " + Maximum2);

}

}

**package** com.test.program.level1;

**public** **class** PrimeNumberExample {

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

**int** n = 11;

System.***out***.println(*isPrimeNumber*(n) ? n + " is prime number." : n + " is not prime number.");

n = 12;

System.***out***.println(*isPrimeNumber*(n) ? n + " is prime number." : n + " is not prime number.");

n = 13;

System.***out***.println(*isPrimeNumber*(n) ? n + " is prime number." : n + " is not prime number.");

n = 14;

System.***out***.println(*isPrimeNumber*(n) ? n + " is prime number." : n + " is not prime number.");

}

/\*\*

\* returns true if number is prime.

\*/

**public** **static** **boolean** isPrimeNumber(**int** n) {

**for** (**int** i = 2; i <= Math.*sqrt*(n); i++) {

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

**return** **false**;

}

}

**return** **true**; // means number wasn't divisible by any of the number, it's a prime number.

}

}

**package** com.test.program.level1;

**public** **class** RemoveDuplicatesFromSortedArrayExample {

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

**int**[] duplicateSortedAr = { 1, 3, 7, 8, 8, 9, 14, 16, 16, 17, 17 };

**int**[] nonDuplicateSortedAr = *removeDuplicates*(duplicateSortedAr);

System.***out***.print("Displaying contents of sorted array(with duplicate elements) : ");

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

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

}

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

System.***out***.print("Displaying contents of sorted array(with non-duplicate elements) : ");

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

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

}

}

/\*\*

\* This method returns sorted array with non duplicate elements

\*/

**public** **static** **int**[] removeDuplicates(**int**[] arr) {

**int** n = arr.length;

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

**return** arr;

}

**int** temp[] = **new** **int**[n];

// Start traversing elements

**int** j = 0;

**for** (**int** i = 0; i < n - 1; i++)

// If current element is not equal

// to next element then store that

// current element

**if** (arr[i] != arr[i + 1]) {

temp[j++] = arr[i];

}

// Store the last element as whether

// it is unique or repeated, it hasn't

// stored previously

temp[j++] = arr[n - 1];

**return** temp;

}

}

**package** com.test.program.level1;

**public** **class** ReverseNumberExample {

**public** **static** **void** main(String... args) {

**int** number = 12345; // number to be reversed

System.***out***.println("Original number: " + number);

System.***out***.println("Reversed number: " + *reverseNumber*(number));

}

**public** **static** **int** reverseNumber(**int** number) {

**int** reverse = 0;

**int** remainder;

**while** (number > 0) {

remainder = number % 10;

number = number / 10;

reverse = reverse \* 10 + remainder;

}

**return** reverse;

}

}

**package** com.test.program.level1;

**public** **class** ReverseString {

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

String str="Rakesh";

String rev=*reverse*(str);

System.***out***.println("original string : "+ str);

System.***out***.println("Reverse string: "+ rev);

}

**public** **static** String reverse(String originalStr) {

**char** [] arr=originalStr.toCharArray();

**for**(**int** i=0, j= arr.length-1; i<(arr.length/2); i++, j--) {

**char** temp =arr[i];

arr[i]=arr[j];

arr[j] = temp;

}

**return** **new** String(arr);

}

}

**package** com.test.program.level1;

**public** **class** ReverseStringUsingStringBufferExample {

**public** **static** **void** main(String... args) {

String originalString = "abcde"; // String to be reversed

StringBuffer sb = **new** StringBuffer(originalString);

System.***out***.println("Original String: " + originalString);

System.***out***.println("Reversed String: " + sb.reverse());

}

}

**package** com.test.program.level1;

**public** **class** SumOfDigitsInNumberExample {

**public** **static** **void** main(String...args){

**int** number=1234;

System.***out***.println("number : "+number);

System.***out***.println("sum of digits : "+*sumOfDigits*(number));

}

**public** **static** **int** sumOfDigits(**int** number){

/\* int sum=0;

int remainder;

while(number>0){

remainder=number%10;

number=number/10;

sum+=remainder;

}

return sum;\*/

**int** s=0;

**while**(number>0) {

**int** r=number%10;

s=s+r;

number = number/10;

}

**return** s;

}

}

**Level2 Program**

**package** com.test.program.level2;

**public** **class** FactorialRecursionExample {

**public** **static** **void** main(String... args) {

**int** num = 4;

System.***out***.println("Factorial of " + num + " is: " + *findFactorail*(num));

}

/\*

\* return factorial of num.

\*/

**public** **static** **int** findFactorail(**int** num) {

**if** (num == 0)

**return** 1;

**return** num \* *findFactorail*(num - 1);

}

}

**package** com.test.program.level2;

**public** **class** FibonacciRecursionExample {

**public** **static** **void** main(String... args) {

**int** n = 10; // generate series upto n.

System.***out***.print("FibonacciSeries : 0 1 ");

**for** (**int** i = 2; i <= n; i++) {

System.***out***.print(*fibonacciRecursion*(i) + " ");

}

}

/\*

\* return next number of FibonacciSeries using recursion.

\*/

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

**if** (n == 1 || n == 2) {

**return** 1;

}

**return** *fibonacciRecursion*(n - 1) + *fibonacciRecursion*(n - 2); // using tail recursion

}

}

**package** com.test.program.level2;

**import** java.util.LinkedHashMap;

**import** java.util.Map;

**public** **class** FirstNonRepeatedCharacterInStringExample {

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

String inputString="this is it";

System.***out***.println("The first non repeated character in inputString("+inputString+") is : " + *firstNonRepeatedCharacter*(inputString));

}

/\*\*

\* Method returns first non-repeating character in inputString.

\* Returns null if there is no non-repeating character in inputString

\*/

**public** **static** Character firstNonRepeatedCharacter(String inputString){

Map<Character,Integer> map= **new** LinkedHashMap<Character ,Integer>(); //LinkedHashMap used so that we could maintain insertion order.

**char** ar[]=inputString.toCharArray();

**char** ch ;

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

ch=ar[i];

**if**(map.containsKey(ch)) //if map already contains this character as key, get value corresponding to key and increment it.

map.put(ch, map.get(ch)+1 );

**else** //put character in map with value as 1 (showing first occurrence of key in string)

map.put(ch, 1) ;

}

/\*

\* Till this point of program, we have stored all unique characters in map as key & corresponding value representing count of character.

\*/

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

ch= ar[i];

**if**( map.get(ch) == 1 ) //we have found our first non-repeating character in string.

**return** ch;

}

**return** **null** ;

}

}

**package** com.test.program.level2;

**public** **class** PalindromeRecursionExample {

**public** **static** **void** main(String... args) {

String inputString = "aabaa";

System.***out***.println(*isPalindromeUsingRecursion*(inputString) ? inputString + " is a palindrome."

: inputString + "is not a palindrome.");

}

/\*\*

\* This methods finds out whether inputString is palindrome or not recursively.

\* Returns true if inputString is palindrome.

\*/

**public** **static** **boolean** isPalindromeUsingRecursion(String inputString) {

**if** (inputString.length() == 0 || inputString.length() == 1) {

**return** **true**;

}

**if** (inputString.charAt(0) == inputString.charAt(inputString.length() - 1)) {

**return** *isPalindromeUsingRecursion*(inputString.substring(1, inputString.length() - 1));

}

**return** **false**;

}

}

**package** com.test.program.level2;

**public** **class** StringRecursiveReversal {

String reverse = "";

**public** String reverseString(String str) {

**if** (str.length() == 1) {

**return** str;

} **else** {

reverse += str.charAt(str.length() - 1) + reverseString(str.substring(0, str.length() - 1));

**return** reverse;

}

}

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

StringRecursiveReversal srr = **new** StringRecursiveReversal();

System.***out***.println("Result: " + srr.reverseString("Rakesh"));

}

}

Level3 Program

**package** com.test.program.level3;

**public** **class** AllPossibleCombinatons {

**static** **void** printCombinations(**char**[] sequence, **int** N) {

**char**[] temp = **new** **char**[N];

**for** (**int** r = 0; r < sequence.length; r++)

*combinations*(sequence, temp, 0, N - 1, 0, r);

}

**private** **static** **void** combinations(**char**[] ar, **char**[] temp, **int** start, **int** end, **int** index, **int** r) {

**if** (index == r) {

**for** (**int** j = 0; j < r; j++) {

System.***out***.print(temp[j] + " ");

}

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

}

**for** (**int** i = start; i <= end && ((end - i + 1) >= (r - index)); i++) {

temp[index] = ar[i];

*combinations*(ar, temp, i + 1, end, index + 1, r);

}

}

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

String str = "ABCDE";

**char**[] sequence = str.toCharArray();// { 'a', 'b', 'c', 'd', 'e' };

System.***out***.print("The combinations are: ");

// printCombinations(sequence, sequence.length);

*printCombinations*(sequence, 3);

}

}

**package** com.test.program.level3;

// Java program to print all permutations of a

// given string.

**public** **class** Permutation {

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

String str = "ABCD";

**int** n = str.length();

Permutation permutation = **new** Permutation();

permutation.permute(str, 0, n - 1);

//permutation.permuteString(str, 0, n - 1);

}

/\*\*

\* permutation function

\*

\* **@param** str

\* string to calculate permutation for

\* **@param** low

\* starting index

\* **@param** high

\* end index

\*/

**private** **void** permute(String str, **int** low, **int** high) {

//print the string when low has reached to end of string

**if** (low == high)

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

**else** {

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

str = swap(str, low, i);

permute(str, low + 1, high);

str = swap(str, low, i);

}

}

}

**private** **void** permuteString(String s, **int** beg, **int** end) {

**if**(beg == end) {

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

}**else** {

**for** (**int** i = beg; i <= end; i++) {

s = swapString(s, beg, i);

permuteString(s, beg+1, end);

s = swapString(s, beg, i);

}

}

}

**public** String swapString(String s, **int** i, **int** j) {

**char** arr[] = s.toCharArray();

**char** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

**return** String.*valueOf*(arr);

}

/\*\*

\* Swap Characters at position

\*

\* **@param** a

\* string value

\* **@param** i

\* position 1

\* **@param** j

\* position 2

\* **@return** swapped string

\*/

**public** String swap(String a, **int** i, **int** j) {

**char** temp;

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

temp = charArray[i];

charArray[i] = charArray[j];

charArray[j] = temp;

**return** String.*valueOf*(charArray);

}

}

**package** com.test.program.level3;

**public** **class** PrintAllCombinationWithSpaceInBetween {

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

String str = "ABC";

**char**[] ar = str.toCharArray();

// Keep tempAr size double-1 than original

// So that it could hold String like "A B C"

**char**[] tempAr = **new** **char**[(2 \* ar.length) - 1];

tempAr[0] = ar[0];

*getCombinations*(ar, tempAr, 1, 1, ar.length);

}

**public** **static** **void** getCombinations(**char**[] ar, **char**[] tempAr, **int** i, **int** j, **int** len) {

**if** (i == len) {

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

tempAr[j] = ' ';

j++;

}

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

**return**;

}

tempAr[j] = ar[i];

*getCombinations*(ar, tempAr, i + 1, j + 1, len);

tempAr[j] = ' ';

tempAr[j + 1] = ar[i];

*getCombinations*(ar, tempAr, i + 1, j + 2, len);

**return**;

}

}

**Binary Tree**

**package** com.test.tree;

**class** Node {

**int** data;

Node left, right, parent;

Node(**int** item) {

data = item;

left = right = **null**;

}

}

**package** com.test.tree;

**public** **class** BinaryTree {

// Root of Binary Tree

Node root;

**public** BinaryTree() {

}

/\*

\* Given a binary tree, print its nodes according to the "bottom-up" postorder

\* traversal.

\*/

**public** **void** printPostorder(Node node) {

**if** (node == **null**)

**return**;

// first recur on left subtree

printPostorder(node.left);

// then recur on right subtree

printPostorder(node.right);

// now deal with the node

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

}

/\* Given a binary tree, print its nodes in inorder \*/

**public** **void** printInorder(Node node) {

**if** (node == **null**)

**return**;

/\* first recur on left child \*/

printInorder(node.left);

/\* then print the data of node \*/

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

/\* now recur on right child \*/

printInorder(node.right);

}

/\* Given a binary tree, print its nodes in preorder \*/

**public** **void** printPreorder(Node node) {

**if** (node == **null**)

**return**;

/\* first print data of node \*/

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

/\* then recur on left sutree \*/

printPreorder(node.left);

/\* now recur on right subtree \*/

printPreorder(node.right);

}

// Wrappers over above recursive functions

**public** **void** printPostorder() {

printPostorder(root);

}

**void** printInorder() {

printInorder(root);

}

**void** printPreorder() {

printPreorder(root);

}

// Driver method

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

BinaryTree tree = **new** BinaryTree();

/\* Constructed binary tree is

1

/ \

2 3

/ \

4 5

\*/

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

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

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

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

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

System.***out***.println("Preorder traversal of binary tree is ");

tree.printPreorder();

System.***out***.println("\nInorder traversal of binary tree is ");

tree.printInorder();

System.***out***.println("\nPostorder traversal of binary tree is ");

tree.printPostorder();

}

}

**package** com.test.tree;

**import** java.util.Stack;

**public** **class** BinaryTreeInOrderWithoutRecursion {

Node root;

//inorder using stack

**void** inorder() {

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

**return**;

}

//keep the nodes in the path that are waiting to be visited

Stack<Node> stack = **new** Stack<Node>();

Node node = root;

//first node to be visited will be the left one

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

stack.push(node);

node = node.left;

}

// traverse the tree

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

// visit the top node

node = stack.pop();

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

**if** (node.right != **null**) {

node = node.right;

// the next node to be visited is the leftmost

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

stack.push(node);

node = node.left;

}

}

}

}

**public** **void** preorder(Node node) {

// Base Case

**if** (node == **null**) {

**return**;

}

// Create an empty stack and push root to it

Stack<Node> nodeStack = **new** Stack<Node>();

nodeStack.push(root);

/\* Pop all items one by one. Do following for every popped item

a) print it

b) push its right child

c) push its left child

Note that right child is pushed first so that left is processed first \*/

**while** (nodeStack.empty() == **false**) {

// Pop the top item from stack and print it

Node mynode = nodeStack.peek();

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

nodeStack.pop();

// Push right and left children of the popped node to stack

**if** (mynode.right != **null**) {

nodeStack.push(mynode.right);

}

**if** (mynode.left != **null**) {

nodeStack.push(mynode.left);

}

}

}

**public** Node insert(Node node, **int** key) {

/\* If the tree is empty, return a new node \*/

**if** (node == **null**)

**return** **new** Node(key);

/\* Otherwise, recur down the tree \*/

**if** (key < node.data) {

node.left = insert(node.left, key);

node.left.parent = node;

} **else** **if** (key > node.data) {

node.right = insert(node.right, key);

node.right.parent = node;

}

/\* return the (unchanged) node pointer \*/

**return** node;

}

// Function to print inorder traversal using parent

// pointer

**void** inorder(Node root) {

**boolean** leftdone = **false**;

// Start traversal from root

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

// If left child is not traversed, find the

// leftmost child

**if** (!leftdone) {

**while** (root.left != **null**) {

root = root.left;

}

}

// Print root's data

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

// Mark left as done

leftdone = **true**;

// If right child exists

**if** (root.right != **null**) {

leftdone = **false**;

root = root.right;

}

// If right child doesn't exist, move to parent

**else** **if** (root.parent != **null**) {

// If this node is right child of its parent,

// visit parent's parent first

**while** (root.parent != **null** && root == root.parent.right)

root = root.parent;

**if** (root.parent == **null**)

**break**;

root = root.parent;

} **else**

**break**;

}

}

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

BinaryTreeInOrderWithoutRecursion tree = **new** BinaryTreeInOrderWithoutRecursion();

tree.root = tree.insert(tree.root, 24);

tree.root = tree.insert(tree.root, 27);

tree.root = tree.insert(tree.root, 29);

tree.root = tree.insert(tree.root, 34);

tree.root = tree.insert(tree.root, 14);

tree.root = tree.insert(tree.root, 4);

tree.root = tree.insert(tree.root, 10);

tree.root = tree.insert(tree.root, 22);

tree.root = tree.insert(tree.root, 13);

tree.root = tree.insert(tree.root, 3);

tree.root = tree.insert(tree.root, 2);

tree.root = tree.insert(tree.root, 6);

System.***out***.println("Inorder traversal is ");

tree.inorder(tree.root);

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

tree.inorder();

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

tree.preorder(tree.root);

}

}

**package** com.test.tree;

**public** **class** BinaryTreePrintLevelOrder {

// Root of the Binary Tree

Node root;

**public** BinaryTreePrintLevelOrder() {

root = **null**;

}

/\* function to print level order traversal of tree \*/

**void** printLevelOrder() {

**int** h = height(root);

**int** i;

**for** (i = 1; i <= h; i++)

printGivenLevel(root, i);

}

/\*

\* Compute the "height" of a tree -- the number of nodes along the longest path

\* from the root node down to the farthest leaf node.

\*/

**int** height(Node root) {

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

**return** 0;

**else** {

/\* compute height of each subtree \*/

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

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

/\* use the larger one \*/

**if** (lheight > rheight)

**return** (lheight + 1);

**else**

**return** (rheight + 1);

}

}

/\* Print nodes at the given level \*/

**void** printGivenLevel(Node root, **int** level) {

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

**return**;

**if** (level == 1)

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

**else** **if** (level > 1) {

printGivenLevel(root.left, level - 1);

printGivenLevel(root.right, level - 1);

}

}

/\* Driver program to test above functions \*/

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

BinaryTreePrintLevelOrder tree = **new** BinaryTreePrintLevelOrder();

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

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

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

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

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

System.***out***.println("Level order traversal of binary tree is ");

tree.printLevelOrder();

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

tree.printGivenLevel(tree.root, 3);

}

}

**package** com.test.tree;

**public** **class** KthLevelAllNodeBinaryTree {

Node root;

Integer sum=0;

**public** **void** printKDistant(Node node, **int** k) {

**if** (node == **null**)

**return**;

**if** (k == 0) {

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

sum += node.data;

**return**;

} **else** {

printKDistant(node.left, k - 1);

printKDistant(node.right, k - 1);

}

}

/\* Driver program to test above functions \*/

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

KthLevelSumBinaryTree tree = **new** KthLevelSumBinaryTree();

/\* Constructed binary tree is

1

/ \

2 3

/ \ /

4 5 8

\*/

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

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

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

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

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

tree.root.right.left = **new** Node(8);

tree.printKDistant(tree.root, 2);

}

}

**package** com.test.tree;

**public** **class** KthLevelSumBinaryTree {

Node root;

Integer sum=0;

**public** **void** printKDistant(Node node, **int** k) {

**if** (node == **null**)

**return**;

**if** (k == 0) {

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

sum += node.data;

**return**;

} **else** {

printKDistant(node.left, k - 1);

printKDistant(node.right, k - 1);

}

}

**public** Integer printSumKDistant(Node node, **int** k) {

**if** (node == **null**)

**return** 0;

**if** (k == 0) {

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

**return** sum+= node.data;

} **else** {

printSumKDistant(node.left, k - 1);

printSumKDistant(node.right, k - 1);

**return** sum;

}

}

/\* Driver program to test above functions \*/

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

KthLevelSumBinaryTree tree = **new** KthLevelSumBinaryTree();

/\* Constructed binary tree is

1

/ \

2 3

/ \ /

4 5 8

\*/

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

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

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

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

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

tree.root.right.left = **new** Node(8);

//tree.printKDistant(tree.root, 2);

Integer s=tree.printSumKDistant(tree.root, 2);

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

}

}