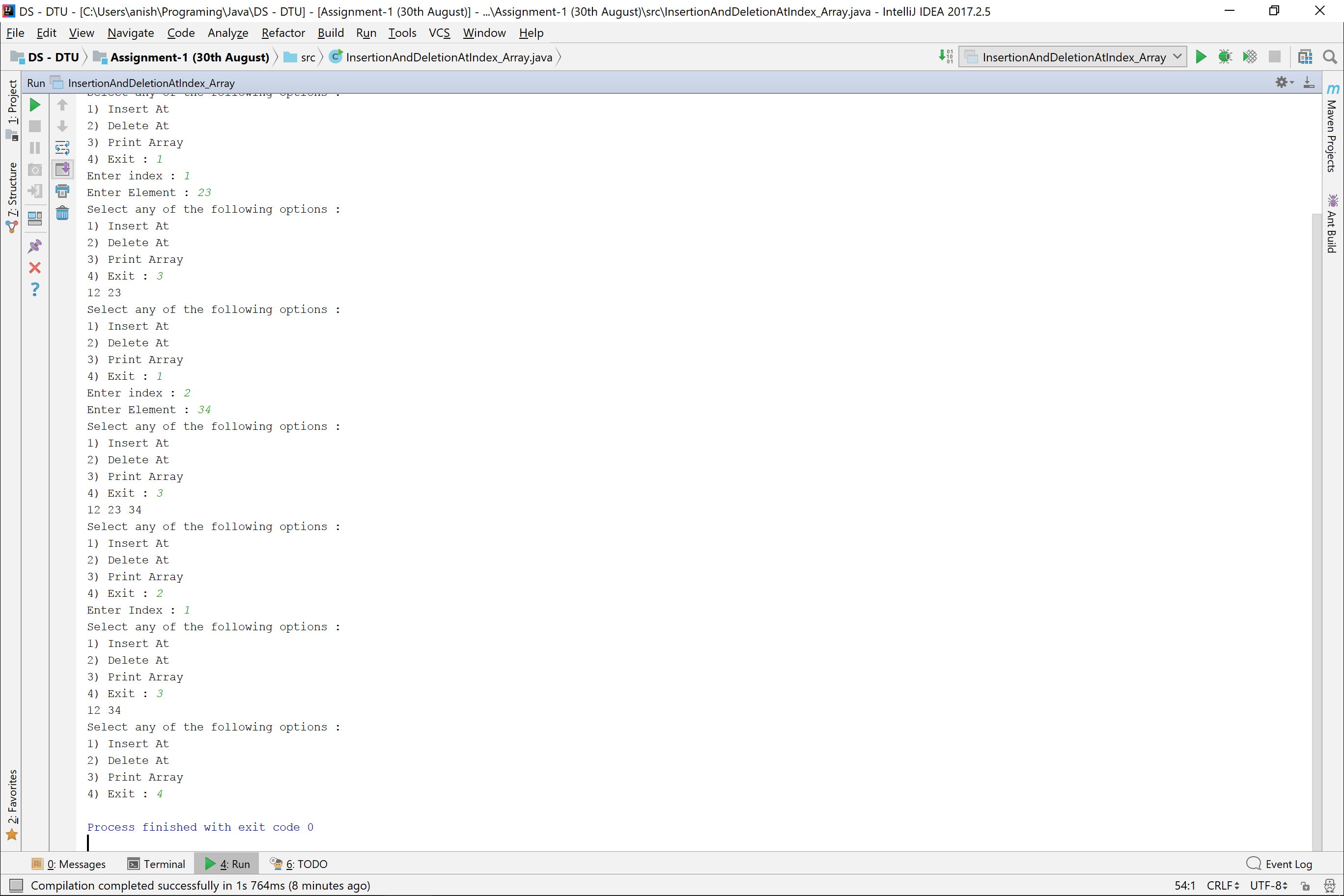
Array Program (Insetion and Deletion)

**import** java.util.Scanner;  
  
**public class** InsertionAndDeletionAtIndex\_Array {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) **throws** ArrayFullException {  
 **int** arr[] = **new int**[20];  
 **int** choice, size=0;  
 **final int** MAX\_SIZE = 20;  
  
 **do** {  
 System.***out***.println(**"Select any of the following options :"**);  
 System.***out***.println(**"1) Insert At"**);  
 System.***out***.println(**"2) Delete At "**);  
 System.***out***.println(**"3) Print Array"**);  
 System.***out***.print(**"4) Exit : "**);  
 choice = *in*.nextInt();  
  
 **switch** (choice){  
 **case** 1 :  
 **if**(size == MAX\_SIZE){ *//Checking To See If Array Is Full* **throw new** ArrayFullException(size);  
 }  
  
 System.***out***.print(**"Enter index : "**);  
 **int** index=*in*.nextInt();  
 **if**(index > size){ *//Checking to see if Index is Valid* **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
  
 System.***out***.print(**"Enter Element : "**);  
 **int** element = *in*.nextInt();  
  
 *//Shifting elements forward* **for**(**int** tempIndex=size ; tempIndex>index ; tempIndex--){  
 arr[tempIndex] = arr[tempIndex-1];  
 }  
 arr[index] = element;  
 size ++;  
 **break**;  
  
 **case** 2 :  
 System.***out***.print(**"Enter Index : "**);  
 index = *in*.nextInt();  
 **if**(index < 0){  
 **throw new** ArrayIndexOutOfBoundsException(index);  
 }  
  
 *//Shifting Elements Back and decreasing size by one* **for**(**int** tempIndex = index ; tempIndex<size ; tempIndex++){  
 arr[tempIndex] = arr[tempIndex + 1];  
 }  
 arr[size--] = 0;  
 **break**;  
  
 **case** 3 :  
 *print*(arr, size);  
 **break**;  
  
 **default**:  
 **if**(choice != 4) System.***out***.println(**"Incorrect Choice"**);  
 }  
  
 }**while** (choice < 4);  
 }  
  
 **private static void** print(**int** arr[], **int** size){  
 **for**(**int** index=0 ; index<size ; index++){  
 System.***out***.print(arr[index] + **" "**);  
 }  
 System.***out***.println();  
 }  
}  
  
**class** ArrayFullException **extends** Exception{  
 ArrayFullException(){}  
 ArrayFullException(**int** size){  
 System.***out***.println(**"Array Full Exception : "** + size);  
 }  
}

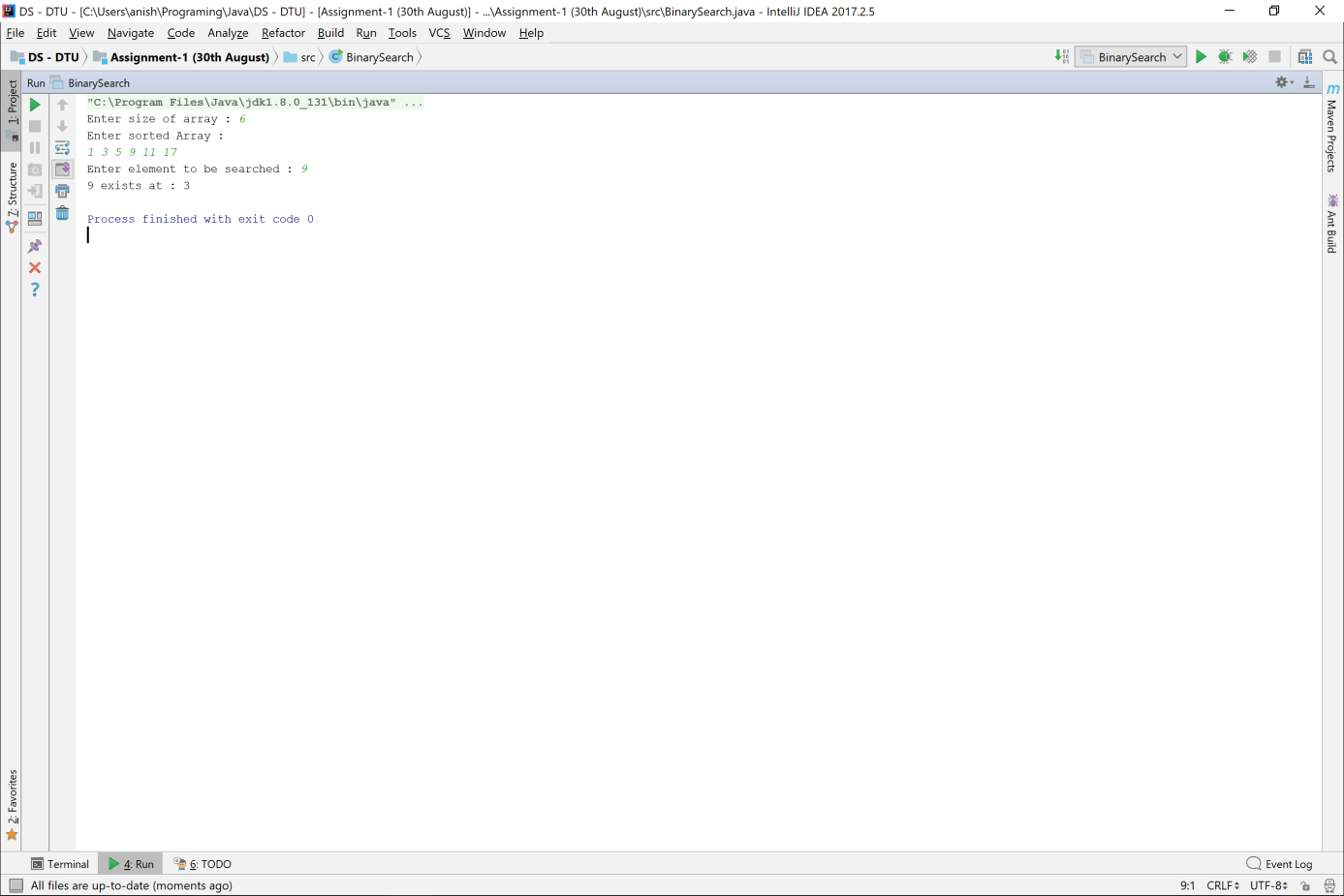


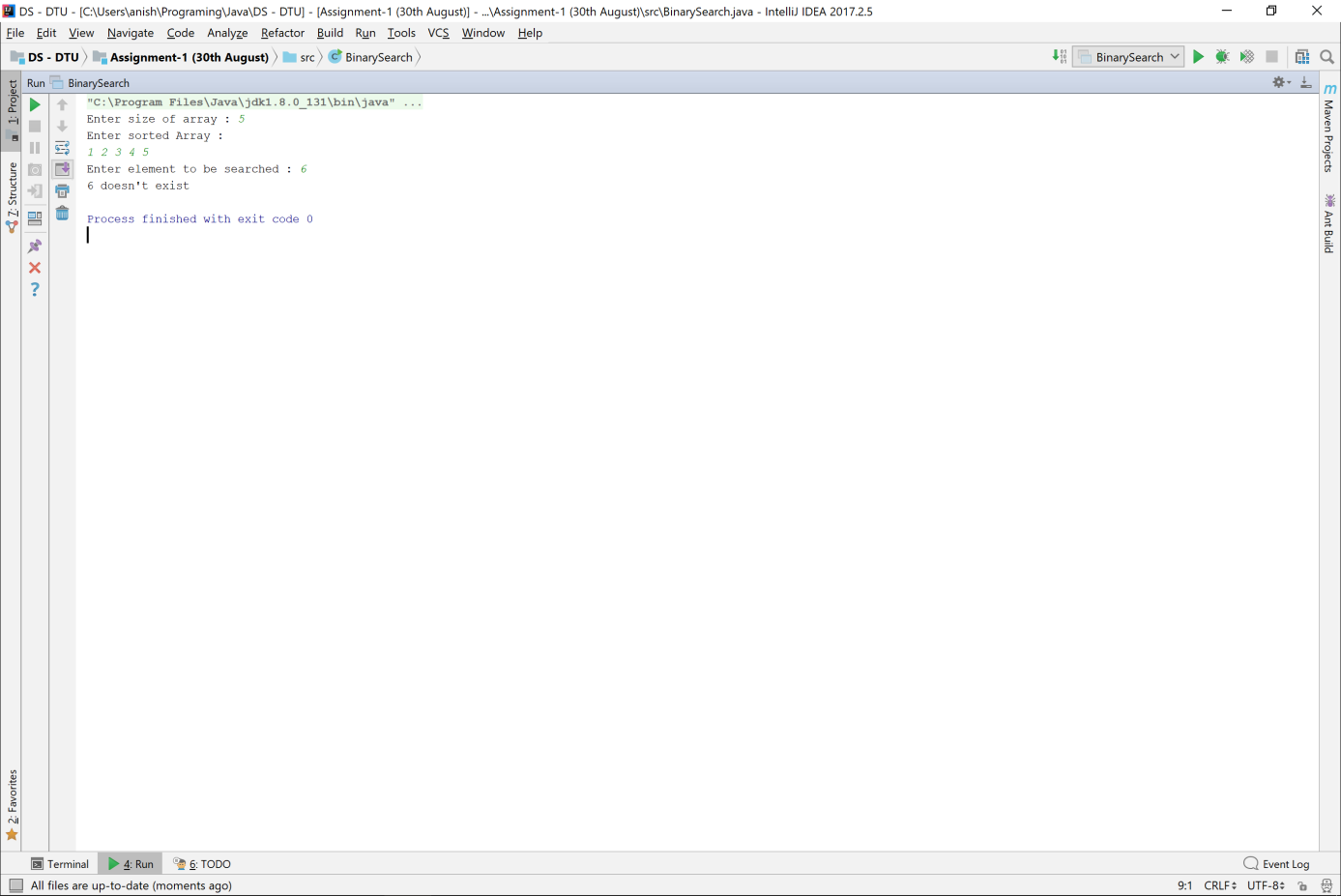
Linear Search

**public class** ArraySearch {  
 **public static void** main(String[] args) {  
 Scanner s = **new** Scanner(System.***in***);  
 **int** size;  
  
 System.***out***.print(**"Enter Size : "**);  
 size = s.nextInt();  
  
 System.***out***.println(**"Enter Array Elements : "**);  
 **int** arr[] = **new int**[size];  
 *input*(arr);  
  
 **int** element;  
 System.***out***.print(**"Enter element for searching : "**);  
 element = s.nextInt();  
  
 **int** ans = *linearSearch*(arr, element);  
 System.***out***.println(ans);  
 }  
  
 **private static int** linearSearch(**int** arr[], **int** element){  
 **for**(**int** index =0 ; index<arr.**length** ; index++){  
 **if**(arr[index] == element)  
 **return** index;  
 }  
 **return** -1;  
 }  
  
 **private static void** input(**int** arr[]){  
 Scanner s = **new** Scanner(System.***in***);  
 **for**(**int** index = 0 ; index<arr.**length** ; index++){  
 arr[index] = s.nextInt();  
 }  
 }  
}

Binary Search (Itterative)

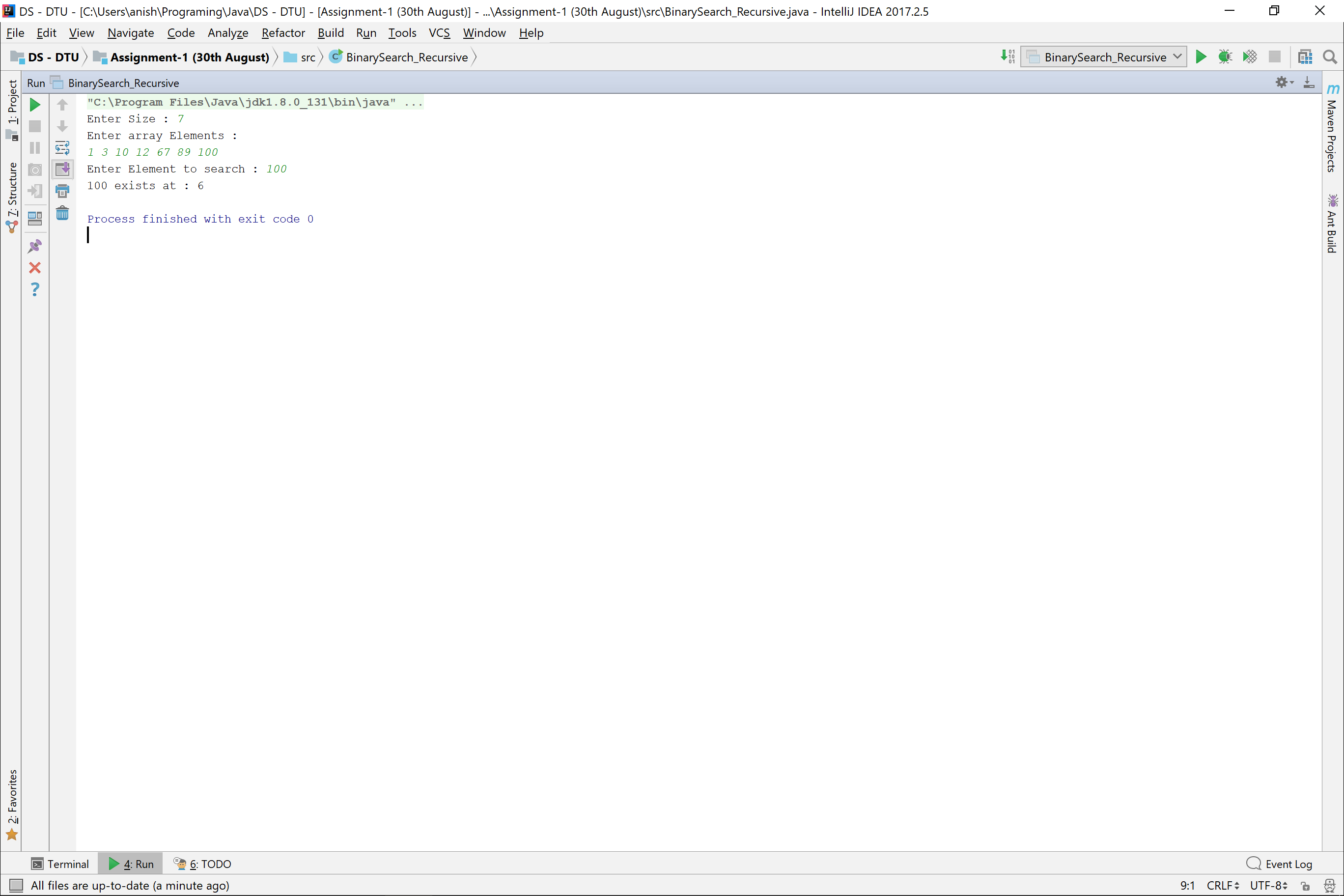
**public class** BinarySearch {  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
  
 System.***out***.print(**"Enter size of array : "**);  
 **int** size = in.nextInt();  
 **int** array[] = **new int**[size];  
  
 System.***out***.println(**"Enter sorted Array : "**);  
 **for**(**int** index =0 ; index<size ; index++){  
 array[index] = in.nextInt();  
 }  
  
 System.***out***.print(**"Enter element to be searched : "**);  
 **int** element = in.nextInt();  
  
 **int** index = *binarySearch*(array, element);  
 **if**(index != -1) System.***out***.println(element + **" exists at : "** + index);  
 **else** System.***out***.println(element + **" doesn't exist"**);  
 }  
  
 **private static int** binarySearch(**int** array[], **int** element){  
 **if**(array.**length** == 0)  
 **return** -1;  
  
 **if**(array.**length** == 1)  
 **return** array[0] == element ? 0 : -1;  
  
 **int** tail=0, head=array.**length**;  
 **while** (tail<head-1){  
 **if**(array[(tail+head)/2] == element)  
 **return** (tail+head)/2;  
 **else if**(array[(tail+head)/2] < element)  
 tail = (tail+head)/2;  
 **else** head = (tail+head)/2;  
 }  
 **return** -1;  
 }  
}

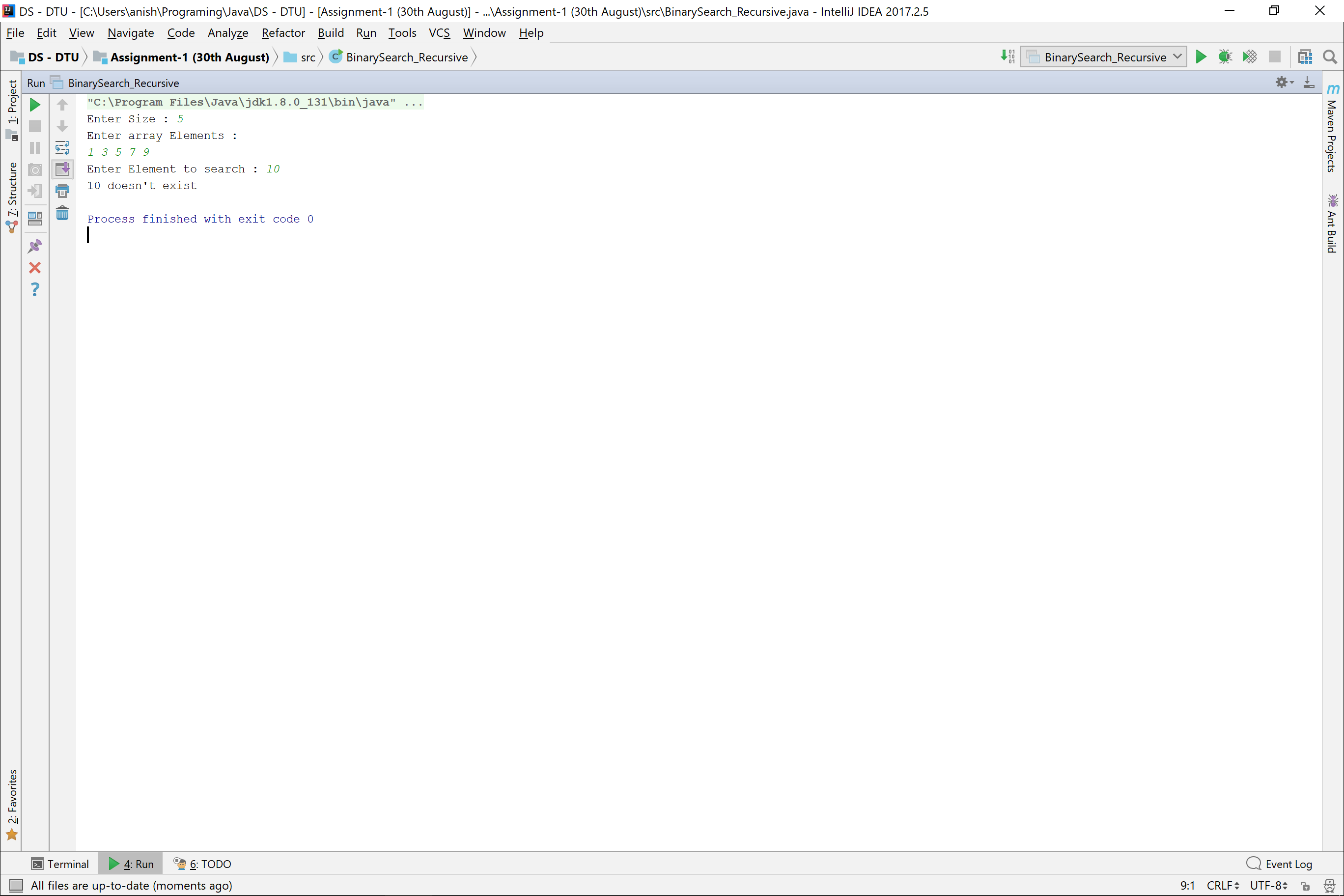




Binary Search (Reccursive)

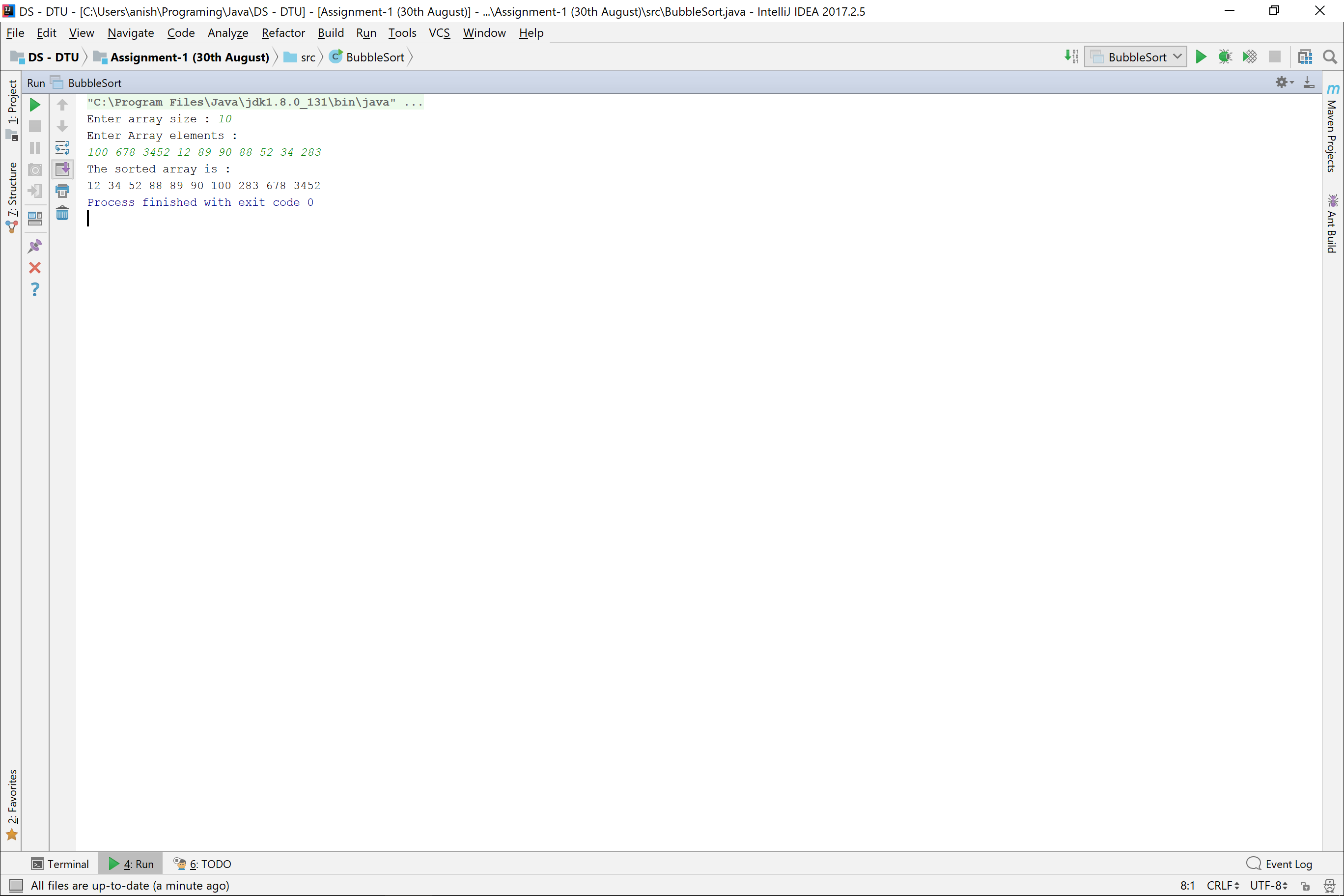
**public class** BinarySearch\_Recursive {  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
 System.***out***.print(**"Enter Size : "**);  
 **int** size = in.nextInt();  
 **int** array[] = **new int**[size];  
  
 System.***out***.println(**"Enter array Elements : "**);  
 **for**(**int** index = 0 ; index<size ; index++){  
 array[index] = in.nextInt();  
 }  
  
 System.***out***.print(**"Enter Element to search : "**);  
 **int** element = in.nextInt();  
  
 **int** index = *binarySearch*(array, element, 0, array.**length**);  
 **if**(index != -1) System.***out***.println(element + **" exists at : "** + index);  
 **else** System.***out***.println(element + **" doesn't exist"**);  
 }  
  
 **private static int** binarySearch(**int** array[], **int** element, **int** startIndex, **int** endIndex){  
 **if**(array.**length** == 0) **return** -1;  
 **if**(array.**length** == 1) **return** array[0] == element ? 0 : -1;  
 **if**(startIndex >= endIndex-1) **return** -1;  
  
 **if**(array[(startIndex+endIndex)/2] == element)  
 **return** (startIndex+endIndex)/2;  
 **else if**(array[(startIndex + endIndex)/2] < element)  
 **return** *binarySearch*(array, element, (startIndex+endIndex)/2, endIndex);  
 **else  
 return** *binarySearch*(array, element, startIndex, (startIndex+endIndex)/2);  
 }  
}





Bubble Sort

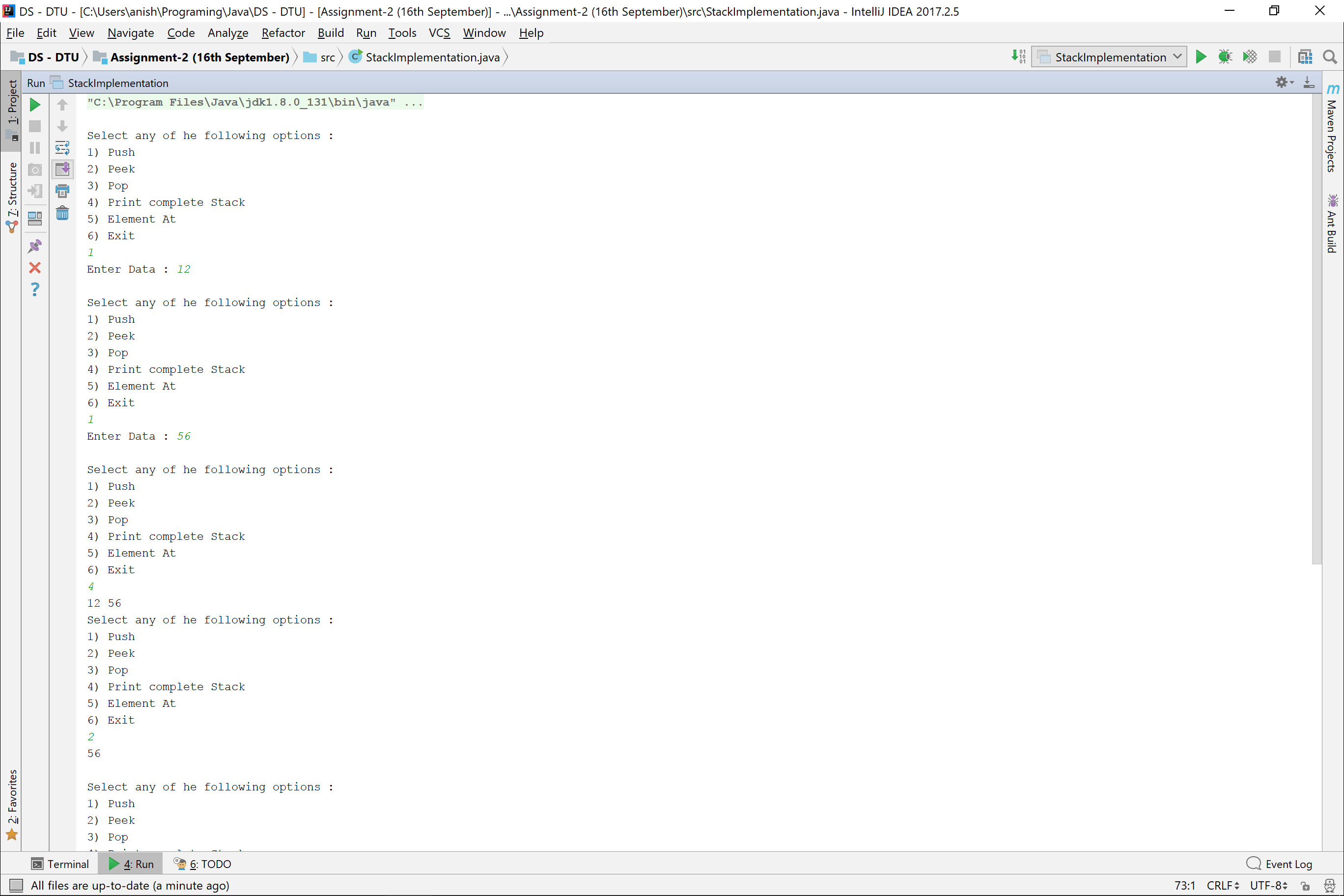
**public class** BubbleSort {  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
 System.***out***.print(**"Enter array size : "**);  
 **int** size = in.nextInt();  
  
 **int** array[] = **new int**[size];  
 System.***out***.println(**"Enter Array elements : "**);  
 **for**(**int** index = 0 ; index<size ; index++){  
 array[index] = in.nextInt();  
 }  
  
 *bubbleSort*(array);  
 System.***out***.println(**"The sorted array is : "**);  
 *print*(array);  
 }  
  
 **private static void** bubbleSort(**int** array[]){  
 **for**(**int** i=0 ; i<array.**length**-1 ; i++){  
 **for**(**int** index=0 ; index< array.**length**-1-i ; index++){  
 **if**(array[index] > array[index+1])  
 *swap*(index, index+1, array);  
 }  
 }  
 }  
  
 **private static void** swap(**int** index1, **int** index2, **int** array[]){  
 **int** temp = array[index1];  
 array[index1] = array[index2];  
 array[index2] = temp;  
 }  
  
 **private static void** print(**int** array[]){  
 **for**(**int** index = 0 ; index<array.**length** ; index++){  
 System.***out***.print(array[index] + **" "**);  
 }  
 }  
}

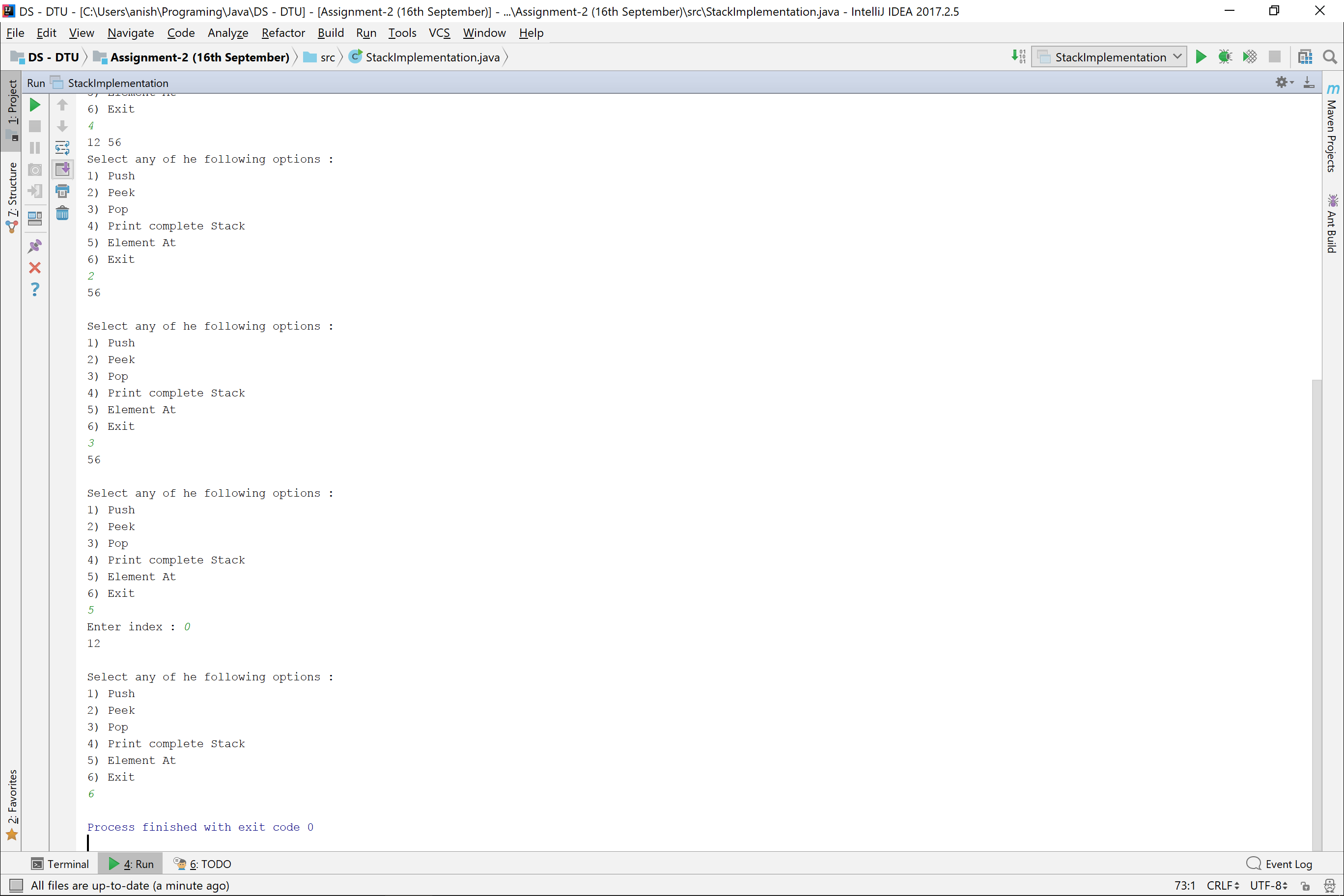


Sorting From Command Prompt

Stack Implementation

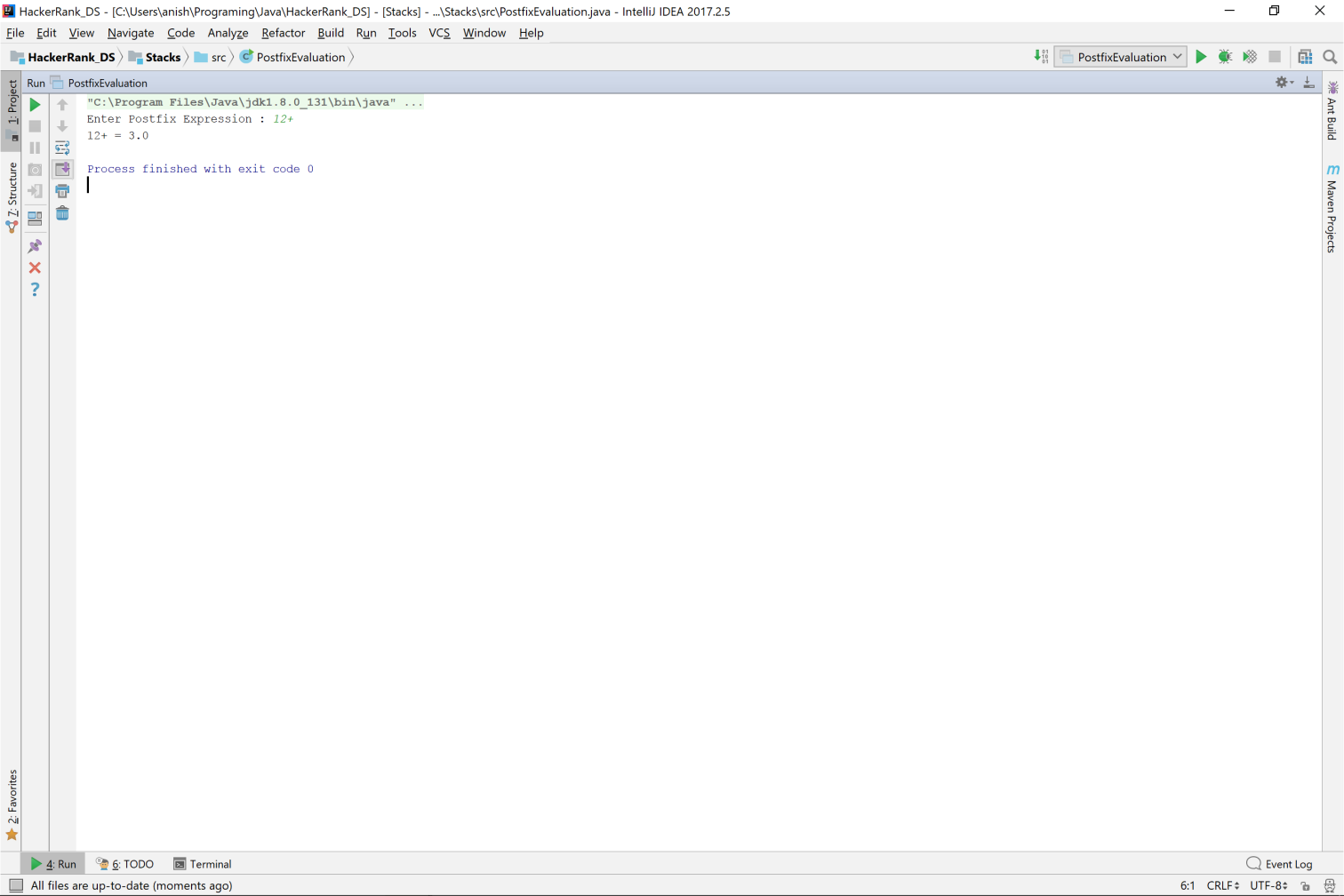
**import** java.util.Scanner;  
  
**class** StackIndexOutOfBoundsException **extends** Exception {  
 StackIndexOutOfBoundsException(){}  
 StackIndexOutOfBoundsException(**int** index){  
 System.***out***.println(**"StackIndexOutOfBoundsException : "** + index);  
 }  
}  
  
**class** StackUnderFlowException **extends** Exception{  
 StackUnderFlowException(){  
 System.***out***.println(**"StackUnderFlowException"**);  
 }  
}  
  
**class** StackOverflowException **extends** Exception{  
 StackOverflowException(){}  
 StackOverflowException(**int** size){  
 System.***out***.println(**"StackOverflowError : "** + size);  
 }  
}  
  
**class** Stack{  
 **private int array**[];  
 **private int size**;  
 **private int MAX\_SIZE**;  
 **private final double LOAD\_FACTOR** = 0.8;  
 **private final int MAX\_STACKFRAME\_SIZE** = 200;  
  
 *//Constructors* Stack(){  
 **array** = **new int**[10];  
 **MAX\_SIZE** = 10;  
 }  
 Stack(**int** stackSize){  
 **array** = **new int**[stackSize];  
 **MAX\_SIZE** = stackSize;  
 }  
 Stack(**int** arr[]) **throws** StackOverflowException{  
 **array** = **new int**[arr.**length** + 10];  
 **MAX\_SIZE** = **array**.**length**;  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 push(arr[index]);  
 }  
 }  
  
  
 *//Methods* **public void** push(**int** data) **throws** StackOverflowException{  
 **array**[**size**] = data;  
 **size**++;  
  
 **if**((**double**)**size**/**MAX\_SIZE** > **LOAD\_FACTOR**)  
 grow();  
 **if**(**size** > **MAX\_STACKFRAME\_SIZE**)  
 **throw new** StackOverflowException(**size**);  
 }  
  
 **private void** grow(){  
 *//Increases Stack Size - Doubles the array  
 //Keeps load factor constant* **int** temp[] = **new int**[2\***array**.**length**];  
 **for**(**int** index=0 ; index<**array**.**length** ; index++){  
 temp[index] = **array**[index];  
 }  
 **MAX\_SIZE** = 2 \* **MAX\_SIZE**;  
 **array** = temp;  
 }  
  
 **public int** pop() **throws** StackUnderFlowException{  
 **if**(**size** == 0)  
 **throw new** StackUnderFlowException();  
 **size**--;  
 **return array**[**size**];  
 }  
  
 **public int** peek() **throws** StackUnderFlowException{  
 **if**(**size** == 0)  
 **throw new** StackUnderFlowException();  
 **return array**[**size**-1];  
 }  
  
 **public int** elementAt(**int** index) **throws** StackIndexOutOfBoundsException{  
 **if**(index >= **size**)  
 **throw new** StackIndexOutOfBoundsException(index);  
  
 **return array**[index];  
 }  
  
 **public void** stackFrame() **throws** StackUnderFlowException{  
 **for**(**int** index=0 ; index<**size** ; index++){  
 System.***out***.print(**array**[index] + **" "**);  
 }  
 **if**(**size** == 0)  
 **throw new** StackUnderFlowException();  
 }  
}  
  
**class** StackImplementation{  
 **public static void** main(String[] args) **throws** StackOverflowException, StackUnderFlowException, StackIndexOutOfBoundsException {  
 Scanner in = **new** Scanner(System.***in***);  
 Stack stack = **new** Stack();  
 **int** n;  
  
 **do** {  
 System.***out***.println(**"\nSelect any of he following options : "**);  
 System.***out***.println(**"1) Push"**);  
 System.***out***.println(**"2) Peek"**);  
 System.***out***.println(**"3) Pop"**);  
 System.***out***.println(**"4) Print complete Stack"**);  
 System.***out***.println(**"5) Element At "**);  
 System.***out***.println(**"6) Exit"**);  
 n = in.nextInt();  
  
 **if**(n == 1){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = in.nextInt();  
 stack.push(data);  
 }**else if**(n == 2){  
 System.***out***.println(stack.peek());  
 } **else if**(n ==3){  
 System.***out***.println(stack.pop());  
 } **else if**(n == 4){  
 stack.stackFrame();  
 } **else if**(n == 5){  
 System.***out***.print(**"Enter index : "**);  
 **int** index = in.nextInt();  
 **int** data = stack.elementAt(index);  
 System.***out***.println(data);  
 }  
 }**while** (n != 6);  
 }  
}

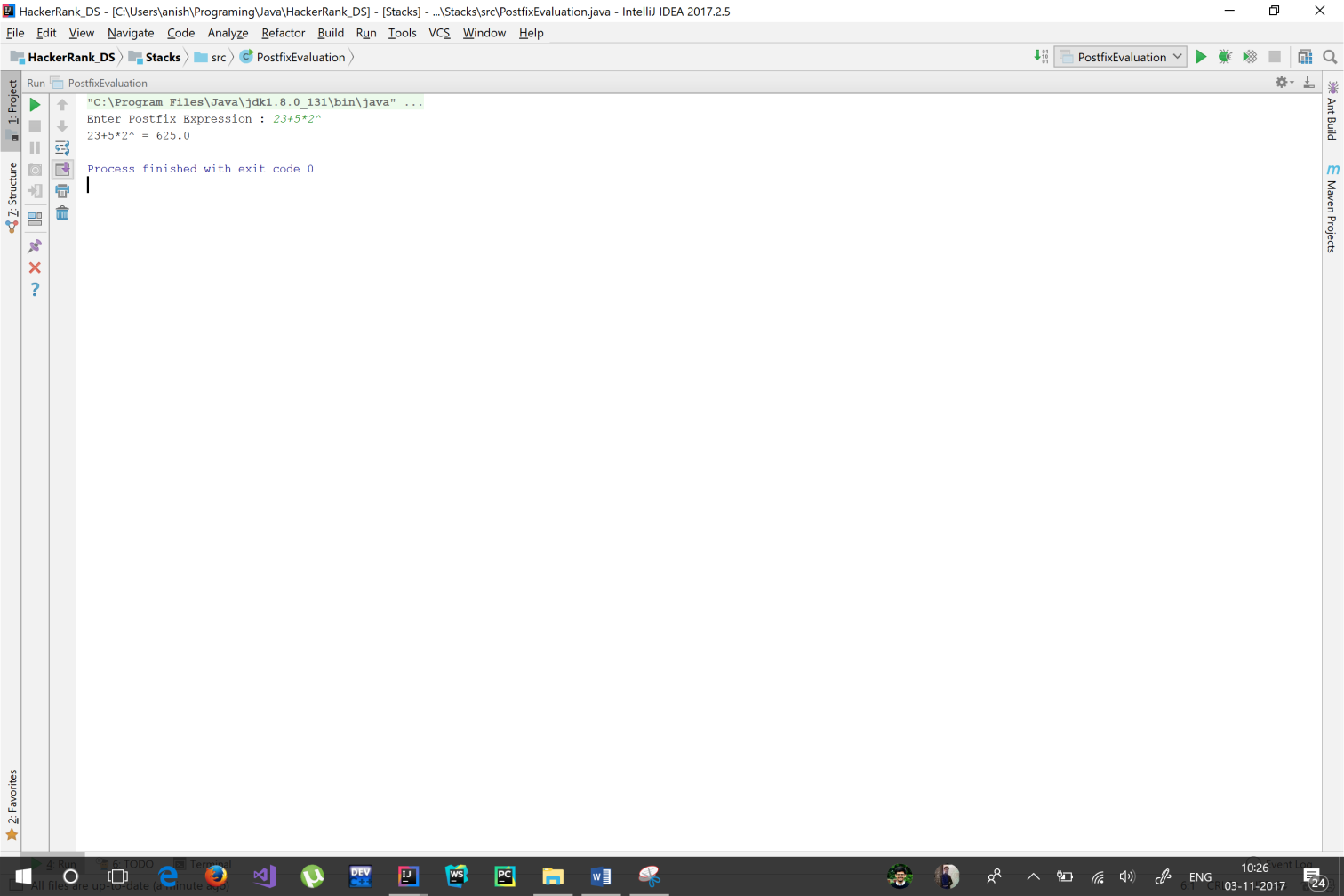




PostFix Evaluvation

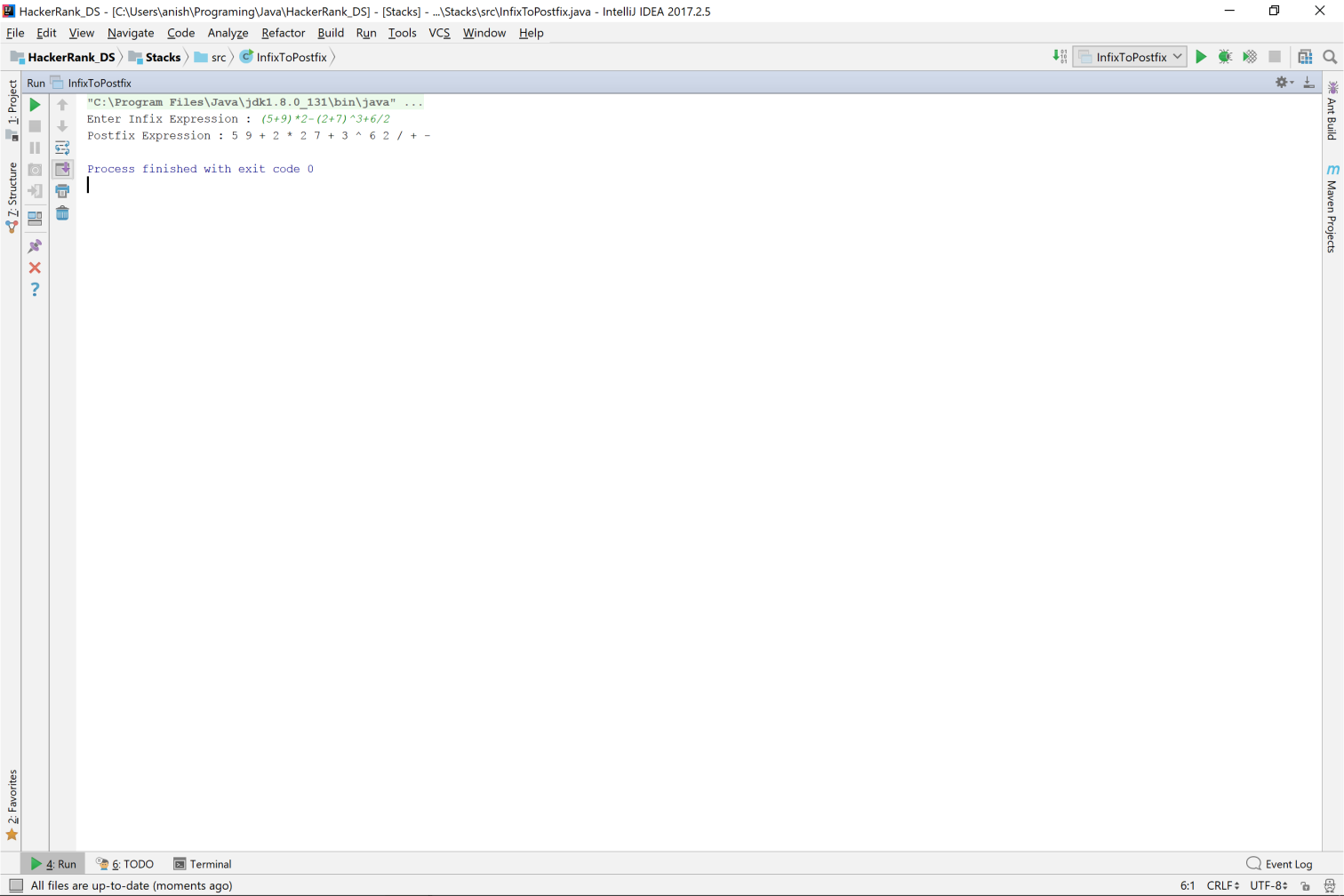
**public class** PostfixEvaluation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Enter Postfix Expression : "**);  
 String postfixExpression = *in*.next();  
 **double** ans = *evaluate*(postfixExpression);  
 System.***out***.println(postfixExpression + **" = "** + ans);  
 }  
  
 **private static double** evaluate(String postfix){  
 **double** ans = 0;  
 Stack<Double> operand = **new** Stack<>();  
  
 **for**(**int** index=0 ; index<postfix.length() ; index++){  
 **if**(*isOperand*(postfix.charAt(index)))  
 operand.push((**double**)*integer*(postfix.charAt(index)));  
  
 *//Is operator* **else**{  
 **double** b = operand.pop();  
 **double** a = operand.pop();  
 **double** operation = *performOperation*(a, b, postfix.charAt(index));  
 operand.push(operation);  
 }  
 }  
  
 **return** operand.peek();  
 }  
  
 **private static boolean** isOperand(Character operand){  
 **return** !(operand == **'+'** || operand == **'-'** ||  
 operand == **'\*'** || operand == **'/'** ||  
 operand == **'('** || operand == **')'** ||  
 operand == **'^'**);  
 }  
  
 **private static int** integer(Character number){  
 **return** (**int**)number - 48;  
 }  
  
 **private static double** performOperation(**double** a ,**double** b, Character operation){  
 **if**(operation == **'+'**)  
 **return** a + b;  
 **else if**(operation == **'-'**)  
 **return** a - b;  
 **else if**(operation == **'\*'**)  
 **return** a \* b;  
 **else if**(operation == **'/'**)  
 **return** a / b;  
 **return** Math.*pow*(a, b);  
 }  
}

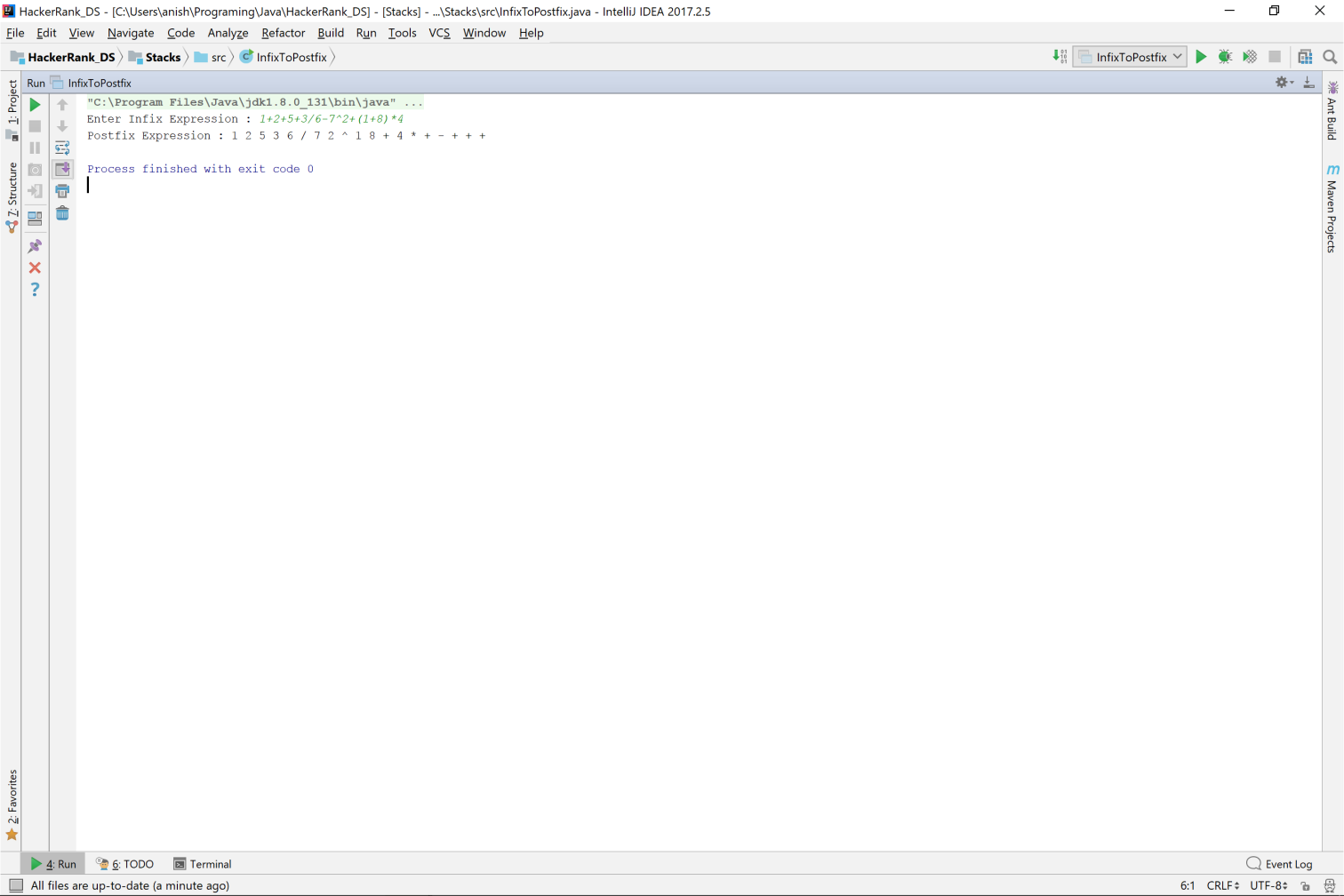




Infix To PostFix

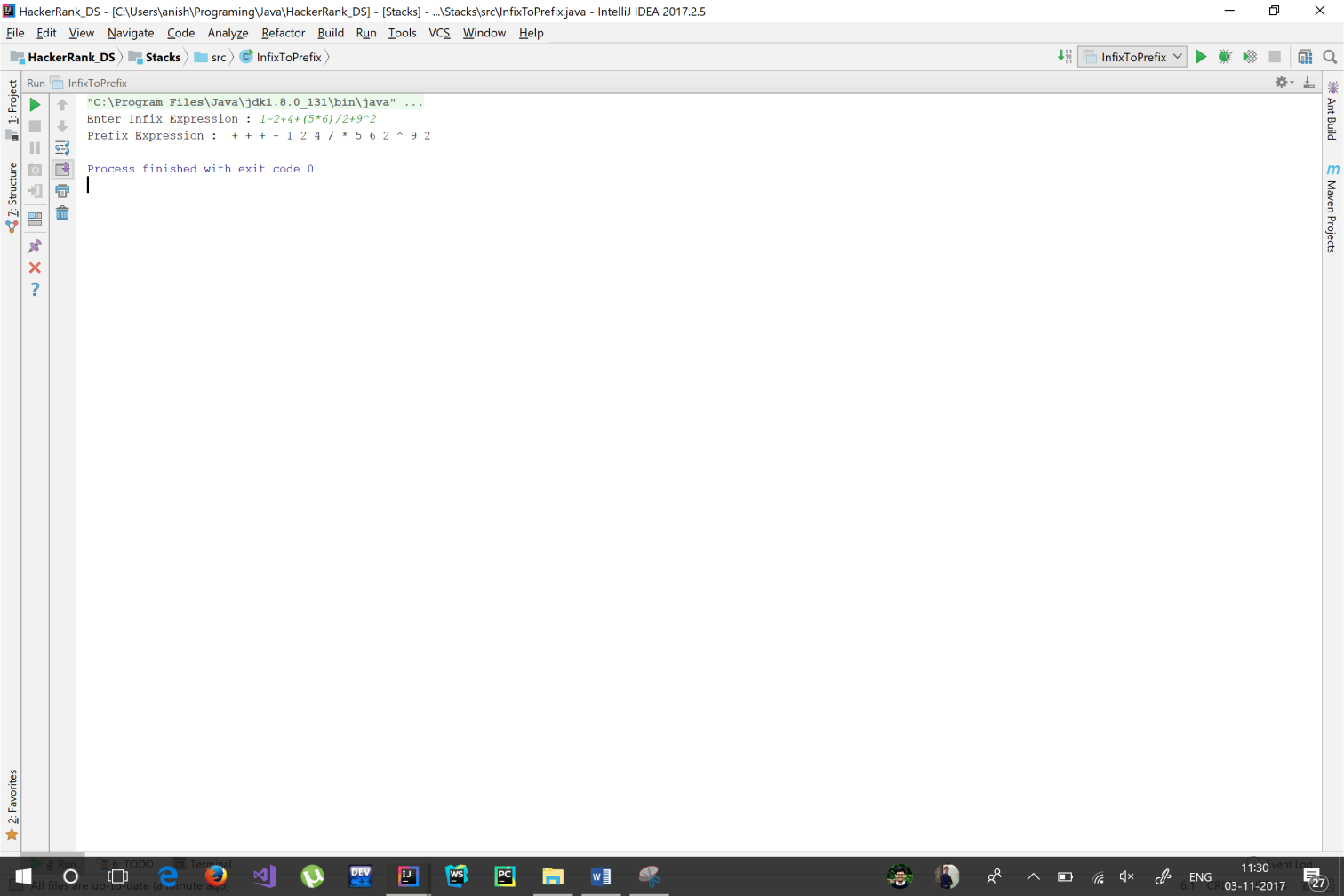
**public class** InfixToPostfix {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
 **private static** HashMap<Character, Integer> *priorityMap* = **new** HashMap<>();  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Enter Infix Expression : "**);  
 String infixExpression = *in*.next();  
 String postfixExpression = *postfix*(infixExpression);  
 System.***out***.println(**"Postfix Expression : "** + postfixExpression);  
 }  
  
 **private static** String postfix(String infix){  
 *initializePriorityMap*();  
 Stack<Character> operator = **new** Stack<>();  
 String postfixExpression = **""**;  
  
 **for**(**int** index=0 ; index<infix.length() ; index++){  
 *//System.out.println(postfixExpression);* **if**(*isOperand*(infix.charAt(index)))  
 postfixExpression += infix.charAt(index) + **" "**;  
  
 *//Is Operator or Bracket* **else** {  
 **if**(*isOpenBracket*(infix.charAt(index)))  
 operator.push(infix.charAt(index));  
 **else if**(*isClosedBracket*(infix.charAt(index))){  
 *//Run while encounters open bracket and add to postfix expression* **while** (**true**){  
 Character operand = operator.pop();  
 **if**(operand == **'('**)  
 **break**;  
 **else** postfixExpression += operand + **" "**;  
 }  
 } *//Encountered Operator* **else** {  
 *//Run while operator is higher precedence than stack.peek()* **while**(operator.size() !=0 && *priority*(infix.charAt(index),operator.peek()) && *isNotBracket*(operator.peek())){  
 Character operand = operator.pop();  
 postfixExpression += operand + **" "**;  
 } operator.push(infix.charAt(index));  
 }  
 }  
  
 *//Add Operator Stack to Postfix Expression* }  
 postfixExpression = *addToExpression*(operator, postfixExpression);  
  
 **return** postfixExpression;  
 }  
  
 **private static boolean** isNotBracket(Character bracket){  
 **return** !(bracket == **'('** || bracket == **')'**);  
 }  
  
 **private static boolean** isOpenBracket(Character bracket){  
 **return** bracket == **'('**;  
 }  
  
 **private static boolean** isClosedBracket(Character bracket){  
 **return** bracket == **')'**;  
 }  
  
 **private static boolean** isOperand(Character operand){  
 **return** !(operand == **'+'** || operand == **'-'** ||  
 operand == **'\*'** || operand == **'/'** ||  
 operand == **'('** || operand == **')'** ||  
 operand == **'^'**);  
 }  
  
 **private static void** initializePriorityMap(){  
 *priorityMap*.put(**'('**, 3); *priorityMap*.put(**')'**, 3);  
 *priorityMap*.put(**'^'**, 2);  
 *priorityMap*.put(**'\*'**, 1); *priorityMap*.put(**'/'**, 1);  
 *priorityMap*.put(**'+'**, 0); *priorityMap*.put(**'-'**, 0);  
 }  
  
 **private static boolean** priority(Character operator, Character stackOperator){  
 **return** *priorityMap*.get(operator) < *priorityMap*.get(stackOperator);  
 }  
  
 **private static** String addToExpression(Stack<Character> operator, String postfix){  
 **while** (!operator.isEmpty()){  
 postfix += operator.pop() + **" "**;  
 }  
 **return** postfix;  
 }  
}

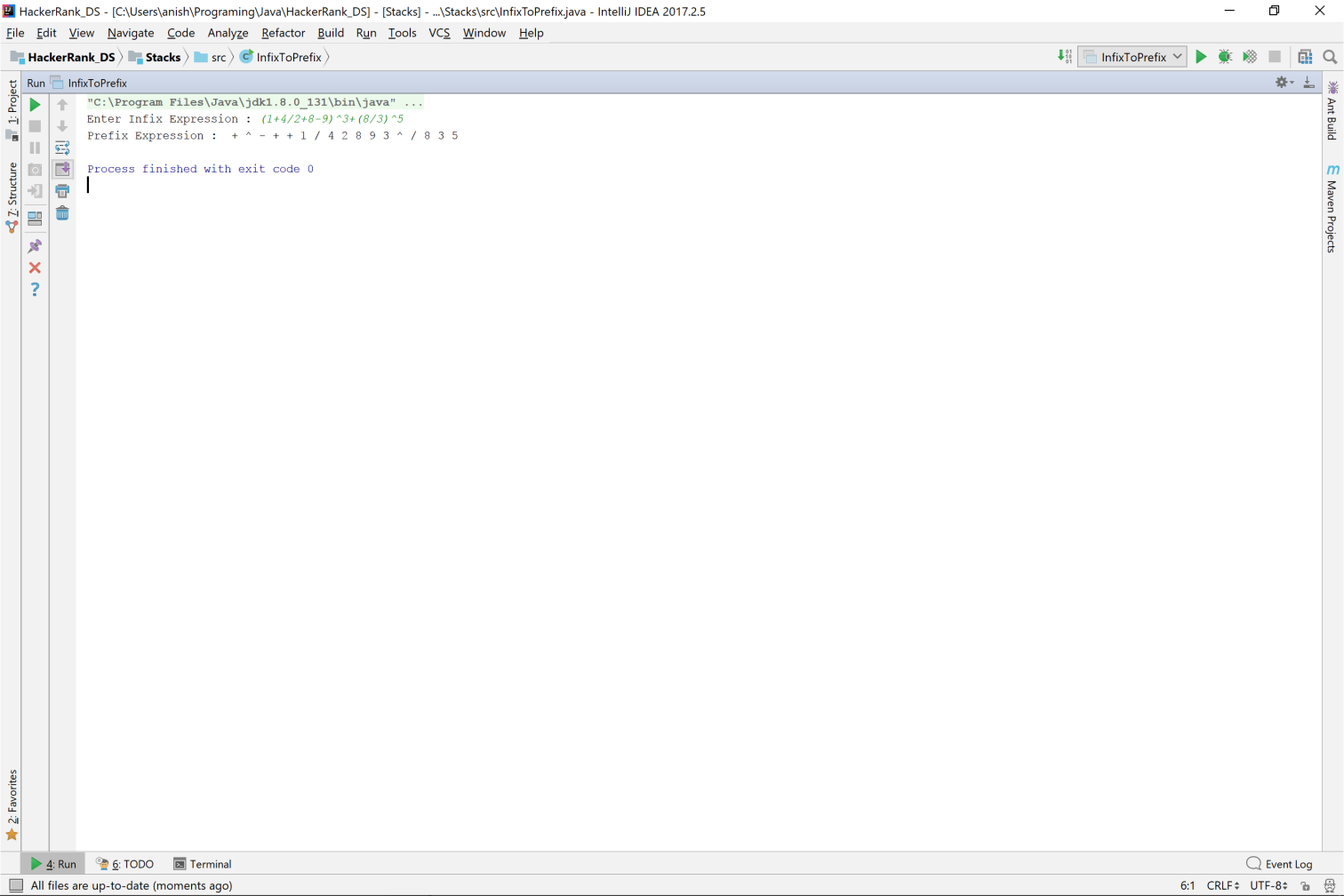




Infix To Prefix

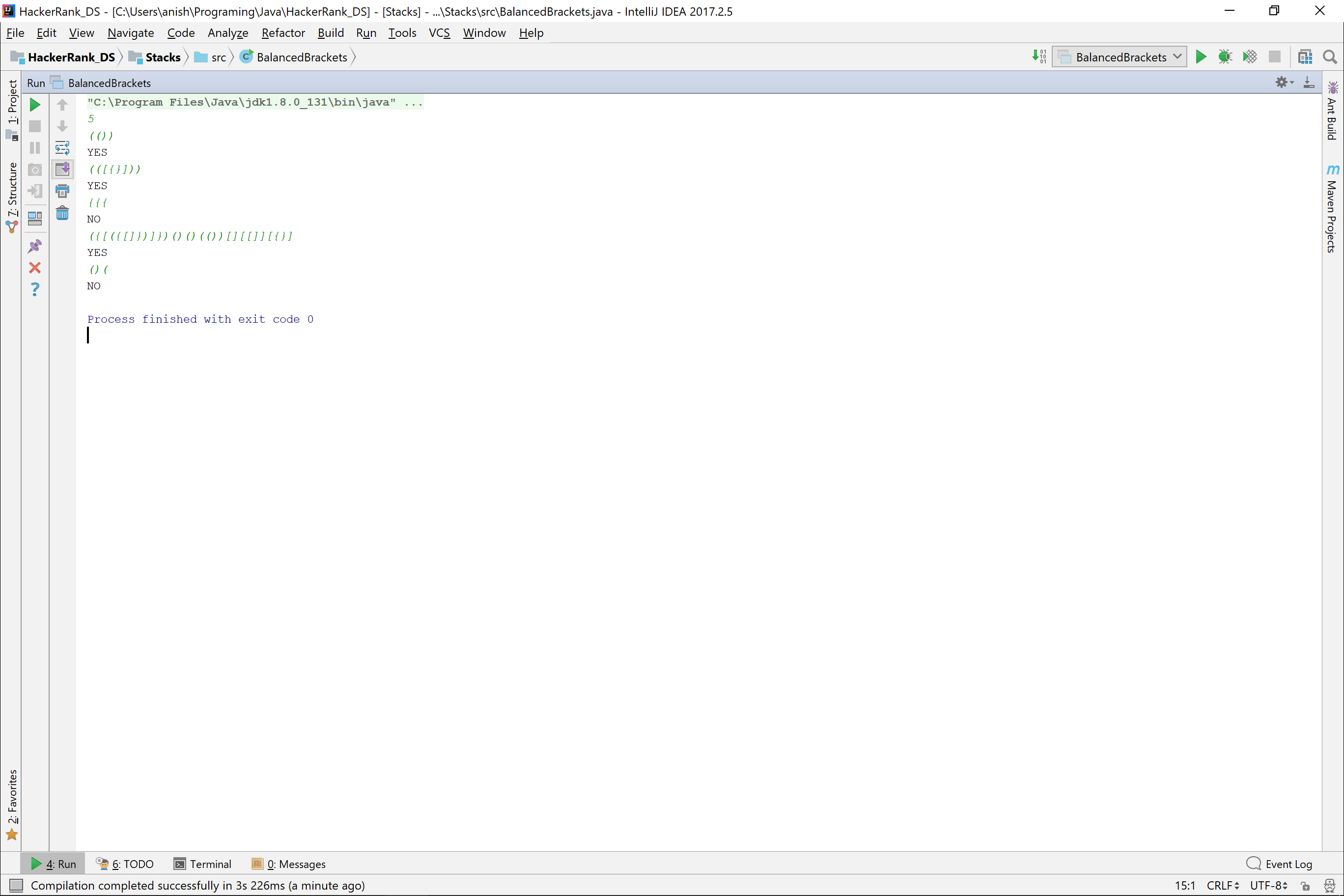
**public class** InfixToPrefix {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
 **private static** HashMap<Character, Integer> *priorityMap* = **new** HashMap<>();  
  
 **public static void** main(String[] args) {  
 *//Inputting Infix Expression* System.***out***.print(**"Enter Infix Expression : "**);  
 String infixExpression = *in*.next();  
 infixExpression = *reverse*(infixExpression); *//Reversing Infix Expression  
  
 //Calculating Postfix Expression* String postfixExpression = *postfix*(infixExpression);  
 String prefixExpression = *reverse*(postfixExpression); *//Reversing Postfix Expression* System.***out***.println(**"Prefix Expression : "** + prefixExpression);  
 }  
  
 **private static** String reverse(String expression){  
 String ans = **""**;  
 **for**(**int** index=0 ; index<expression.length() ; index++){  
 **if**(*isOperand*(expression.charAt(index)))  
 ans = expression.charAt(index) + ans;  
  
 *//Is Operator or bracket* **else**{  
 **if**(*isBracket*(expression.charAt(index)))  
 ans = *reverse*(expression.charAt(index)) + ans;  
 *//Is Operand* **else**{  
 ans = expression.charAt(index) + ans;  
 }  
 }  
 }  
 **return** ans;  
 }  
  
 **private static** String reverse(Character bracket){  
 **if**(*isOpenBracket*(bracket))  
 **return ")"**;  
 **return "("**;  
 }  
  
 **private static** String postfix(String infix){  
 *initializePriorityMap*();  
 Stack<Character> operator = **new** Stack<>();  
 String postfixExpression = **""**;  
  
 **for**(**int** index=0 ; index<infix.length() ; index++){  
 *//System.out.println(postfixExpression);* **if**(*isOperand*(infix.charAt(index)))  
 postfixExpression += infix.charAt(index) + **" "**;  
  
 *//Is Operator or Bracket* **else** {  
 **if**(*isOpenBracket*(infix.charAt(index)))  
 operator.push(infix.charAt(index));  
 **else if**(*isClosedBracket*(infix.charAt(index))){  
 *//Run while encounters open bracket and add to postfix expression* **while** (**true**){  
 Character operand = operator.pop();  
 **if**(operand == **'('**)  
 **break**;  
 **else** postfixExpression += operand + **" "**;  
 }  
 } *//Encountered Operator* **else** {  
 *//Run while operator is higher precedence than stack.peek()* **while**(operator.size() !=0 && *priority*(infix.charAt(index),operator.peek()) && *isNotBracket*(operator.peek())){  
 Character operand = operator.pop();  
 postfixExpression += operand + **" "**;  
 } operator.push(infix.charAt(index));  
 }  
 }  
  
 *//Add Operator Stack to Postfix Expression* }  
 postfixExpression = *addToExpression*(operator, postfixExpression);  
  
 **return** postfixExpression;  
 }  
  
 **private static boolean** isNotBracket(Character bracket){  
 **return** !(bracket == **'('** || bracket == **')'**);  
 }  
  
 **private static boolean** isOpenBracket(Character bracket){  
 **return** bracket == **'('**;  
 }  
  
 **private static boolean** isBracket(Character bracket){  
 **return** *isClosedBracket*(bracket) || *isOpenBracket*(bracket);  
 }  
  
 **private static boolean** isClosedBracket(Character bracket){  
 **return** bracket == **')'**;  
 }  
  
 **private static boolean** isOperand(Character operand){  
 **return** !(operand == **'+'** || operand == **'-'** ||  
 operand == **'\*'** || operand == **'/'** ||  
 operand == **'('** || operand == **')'** ||  
 operand == **'^'**);  
 }  
  
 **private static void** initializePriorityMap(){  
 *priorityMap*.put(**'('**, 3); *priorityMap*.put(**')'**, 3);  
 *priorityMap*.put(**'^'**, 2);  
 *priorityMap*.put(**'\*'**, 1); *priorityMap*.put(**'/'**, 1);  
 *priorityMap*.put(**'+'**, 0); *priorityMap*.put(**'-'**, 0);  
 }  
  
 **private static boolean** priority(Character operator, Character stackOperator){  
 **return** *priorityMap*.get(operator) < *priorityMap*.get(stackOperator);  
 }  
  
 **private static** String addToExpression(Stack<Character> operator, String postfix){  
 **while** (!operator.isEmpty()){  
 postfix += operator.pop() + **" "**;  
 }  
 **return** postfix;  
 }  
}





Check Balanced Parentheses

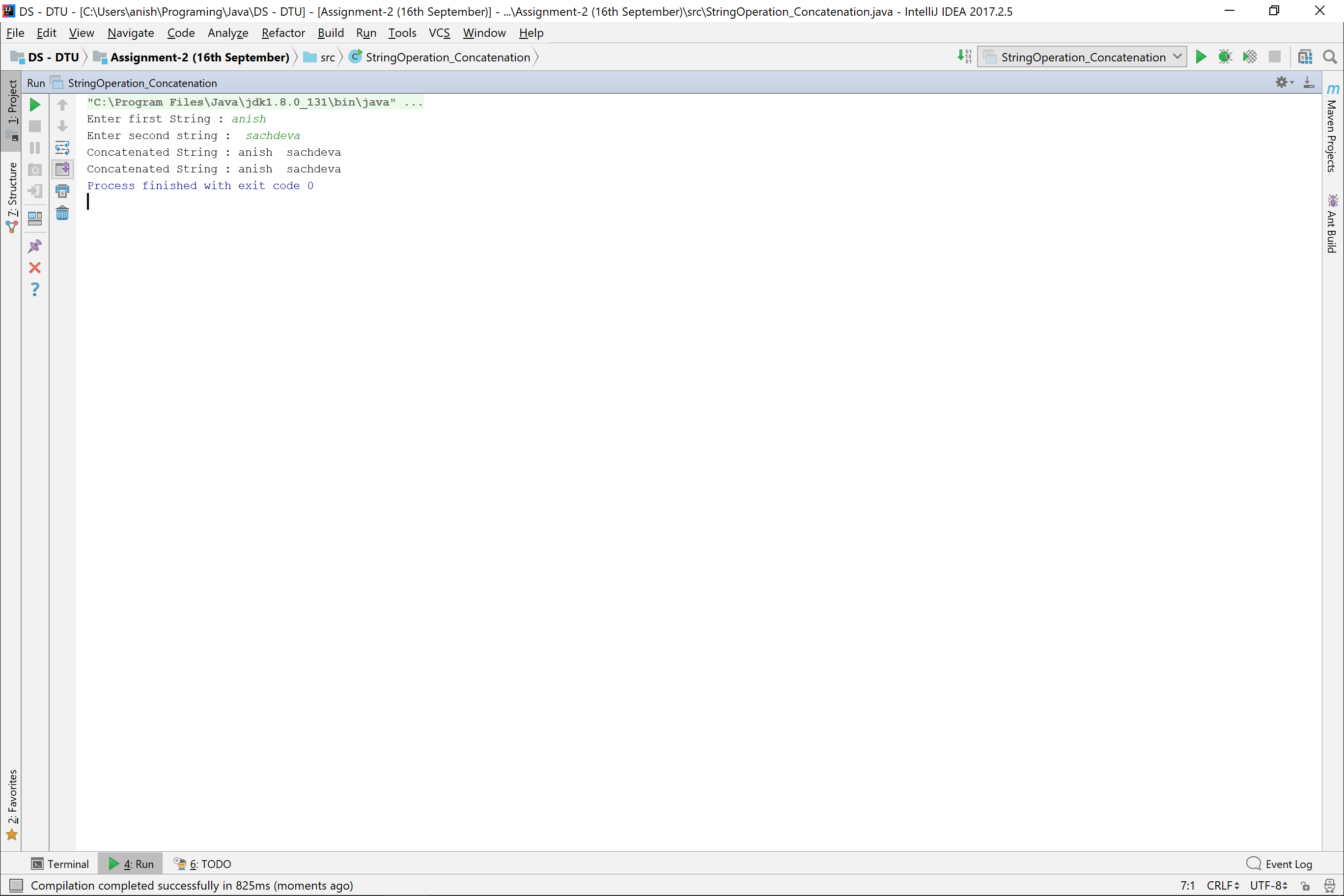
**private static** Scanner *in* = **new** Scanner(System.***in***);  
  
**public static void** main(String[] args) {  
 **int** queries = *in*.nextInt();  
 **while** (queries-- > 0){  
 String bracketString = *in*.next();  
 String ans = *isBalancedBracketString*(bracketString);  
 System.***out***.println(ans);  
 }  
}  
  
**private static** String isBalancedBracketString(String bracketString){  
 **boolean** ans = **true**;  
 Stack<Character> bracketStack = **new** Stack<>();  
  
 **for**(**int** index = 0 ; index<bracketString.length() ; index++){  
 **if**(*isOpenBracket*(bracketString.charAt(index))){  
 bracketStack.push(bracketString.charAt(index));  
 } **else if**(bracketStack.size() > 0){  
 **if**(*isBracketPair*(bracketStack.peek(), bracketString.charAt(index) )){  
 bracketStack.pop();  
 } **else** {  
 ans = **false**;  
 **break**;  
 }  
 } **else** {  
 ans = **false**;  
 **break**;  
 }  
 }  
  
 **if**(ans) **return "YES"**;  
 **return "NO"**;  
}  
  
**private static boolean** isOpenBracket(Character ch){  
 **return** ch == **'('** || ch == **'['** || ch == **'{'**;  
}  
  
**private static boolean** isBracketPair(**char** ch1, **char** ch2){  
 **return** (ch1 == **'('** && ch2 == **')'**) ||  
 (ch1 == **'['** && ch2 == **']'**) ||  
 (ch1 == **'{'** && ch2 == **'}'**);  
}



String Operations

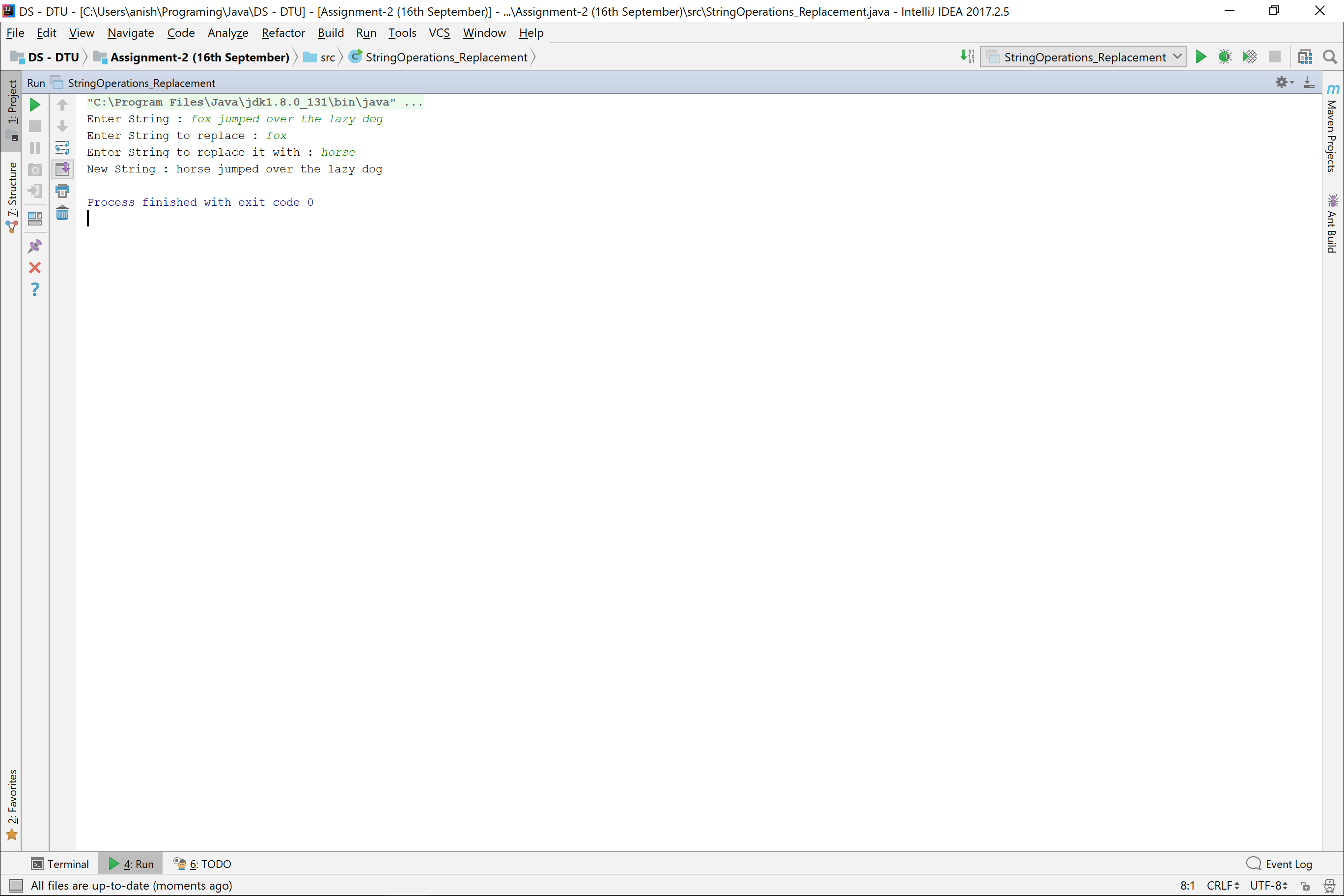
# Concatenation

**public class** StringOperation\_Concatenation {  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
 String first, second;  
  
 System.***out***.print(**"Enter first String : "**);  
 first = in.nextLine();  
 System.***out***.print(**"Enter second string : "**);  
 second = in.nextLine();  
  
 *//Language Defined Operator* String concatenatedString = first + second;  
 System.***out***.println(**"Concatenated String : "** + concatenatedString);  
  
 *//User-Defined Function* **char**[] firstString = *toChar*(first);  
 **char**[] secondString = *toChar*(second);  
 **char**[] concatenated = *concat*(firstString, secondString);  
 System.***out***.print(**"Concatenated String : "** );  
 *print*(concatenated);  
 }  
  
 **private static** String concat(String first, String second){  
 String ans =**""**;  
 **for**(**int** index=0 ; index<first.length() ; index++){  
 ans = ans + first.charAt(index);  
 }  
 **for**(**int** index =0 ; index<second.length() ; index++){  
 ans = ans + second.charAt(index);  
 }  
 **return** ans;  
 }  
  
 **private static char**[] concat(**char**[] first, **char**[] second){  
 **char**[] ans = **new char**[first.**length** + second.**length**];  
 **for**(**int** index=0 ; index<first.**length** ; index++){  
 ans[index] = first[index];  
 }  
 **for**(**int** index=0 ; index<second.**length** ; index++){  
 ans[index + first.**length**] = second[index];  
 }  
 **return** ans;  
 }  
  
 **private static char**[] toChar(String string){  
 **char**[] value = **new char**[string.length()];  
 **for**(**int** index=0 ; index<value.**length** ; index++){  
 value[index] = string.charAt(index);  
 }  
 **return** value;  
 }  
  
 **private static void** print(**char**[] string){  
 **for**(**int** index=0 ; index<string.**length** ; index++){  
 System.***out***.print(string[index]);  
 }  
 }  
}



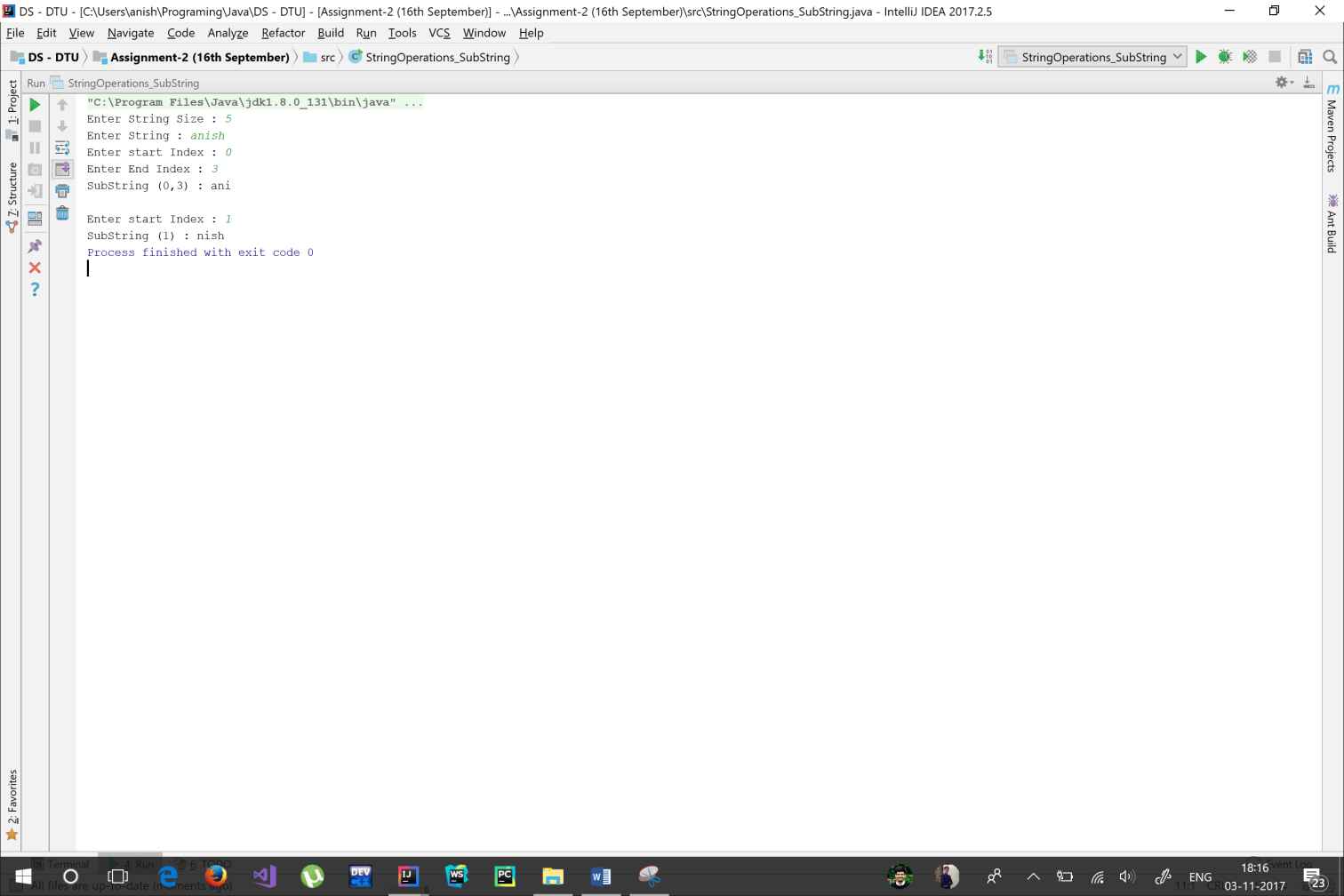
# Replace with String

**public class** StringOperations\_Replacement {  
  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
 String mainString, replaceString, newString;  
  
 System.***out***.print(**"Enter String : "**);  
 mainString = in.nextLine();  
 System.***out***.print(**"Enter String to replace : "**);  
 replaceString = in.next();  
 System.***out***.print(**"Enter String to replace it with : "**);  
 newString = in.next();  
  
 mainString = *replace*(mainString, replaceString, newString);  
 System.***out***.println(**"New String : "** + mainString);  
 }  
  
 **private static** String replace(String main, String replace, String newString){  
 String ans = **""**;  
 ArrayList<Integer> indexList = *instancesOf*(main, replace);  
 **for**(**int** i=0 ; i<indexList.size() ; i++){  
 ans = ans + newString + main.substring(indexList.get(i) + replace.length(),  
 i == indexList.size()-1 ? main.length() : indexList.get(i+1));  
 }  
 **return** ans;  
 }  
  
 **private static** ArrayList<Integer> instancesOf(String main, String replace){  
 ArrayList<Integer> indexList = **new** ArrayList<>();  
 **for**(**int** index =0 ; index<main.length() ; index++){  
 **if**(main.charAt(index) == replace.charAt(0)){  
 **int** factor = (index + replace.length() > main.length() ? (index + replace.length() - main.length()) : 0);  
 **if**(main.substring(index, index + replace.length() - factor).equals(replace))  
 indexList.add(index);  
 }  
 }  
 **return** indexList;  
 }  
}



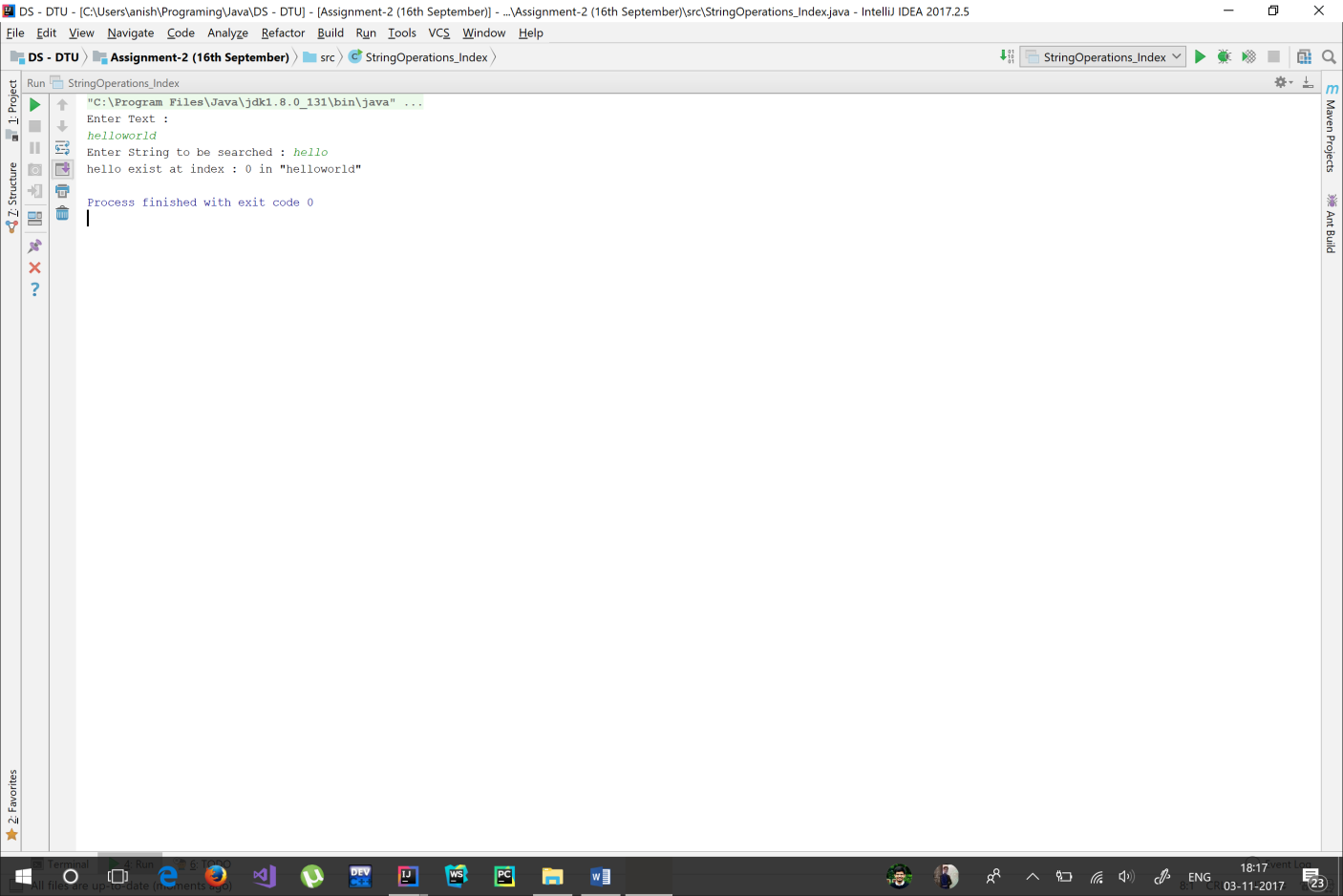
# String SubString

**public class** StringOperations\_SubString {  
 **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
  
 System.***out***.print(**"Enter String Size : "**);  
 **int** size = in.nextInt();  
 **char** string[] = **new char**[size];  
  
 System.***out***.print(**"Enter String : "**);  
 String temp = in.next();  
 **for**(**int** index=0 ; index<string.**length** ; index++){  
 string[index] = temp.charAt(index);  
 }  
  
 *//Substring Function, First Kind* System.***out***.print(**"Enter start Index : "**);  
 **int** startIndex = in.nextInt();  
 System.***out***.print(**"Enter End Index : "**);  
 **int** endIndex = in.nextInt();  
  
 **char** ans[] = *subString*(string, startIndex, endIndex);  
 System.***out***.print(**"SubString ("** + startIndex + **","** + endIndex + **") : "**);  
 *print*(ans);  
  
  
 *//Substring Function, Second Kind* System.***out***.print(**"\n\nEnter start Index : "**);  
 startIndex = in.nextInt();  
  
 ans = *subString*(string, startIndex);  
 System.***out***.print(**"SubString ("** + startIndex + **") : "**) ;  
 *print*(ans);  
 }  
  
 **private static char**[] subString(**char**[] string, **int** startIndex, **int** endIndex){  
 **char** ans[] = **new char**[endIndex - startIndex];  
 **for**(**int** index=startIndex ; index<endIndex ; index++){  
 ans[index-startIndex] = string[index];  
 }  
 **return** ans;  
 }  
  
 **private static char**[] subString(**char**[] string, **int** startIndex){  
 **return** *subString*(string, startIndex, string.**length**);  
 }  
  
 **private static void** print(**char**[] string){  
 **for**(**int** index=0 ; index<string.**length** ; index++){  
 System.***out***.print(string[index]);  
 }  
 }  
}



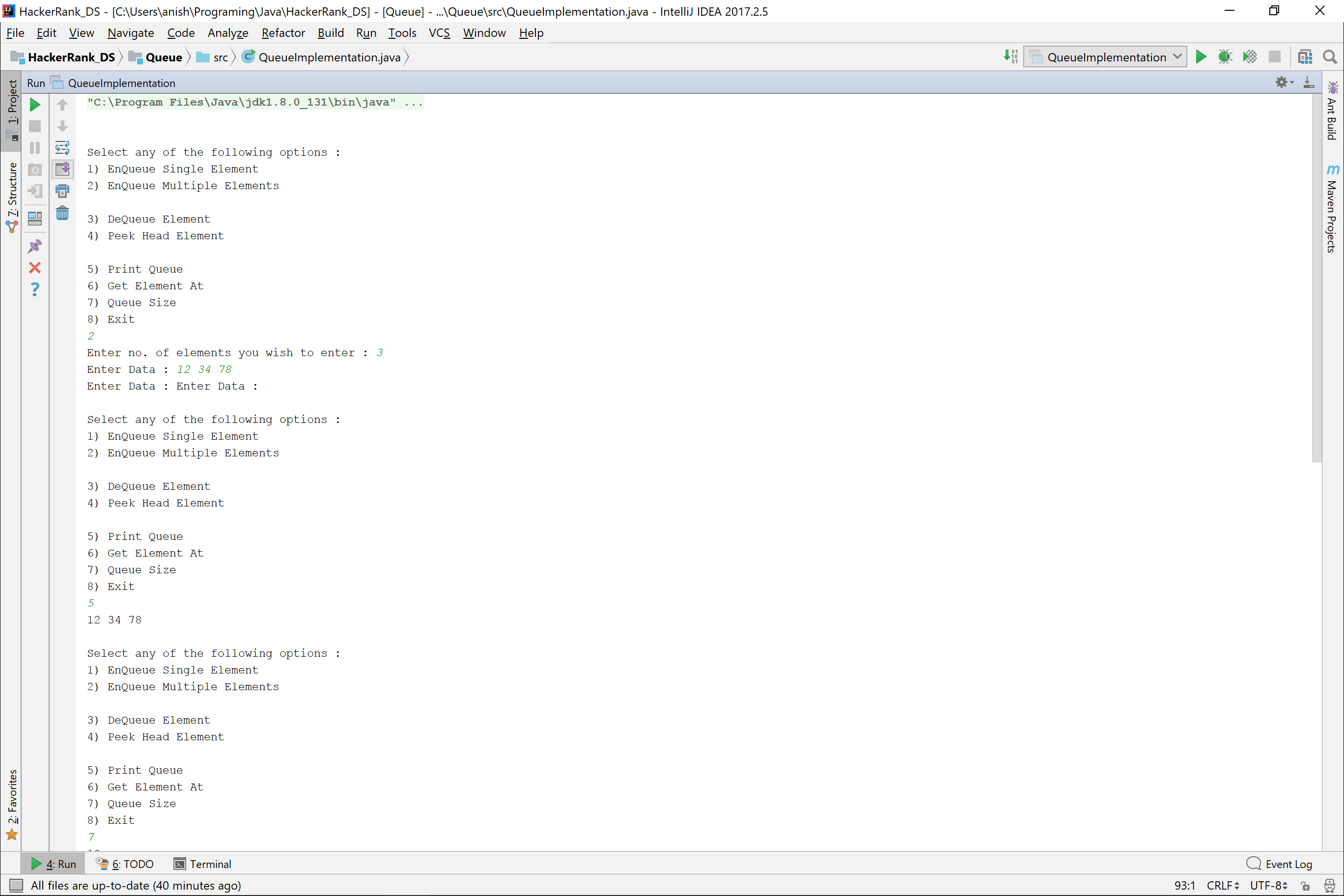
# String Index

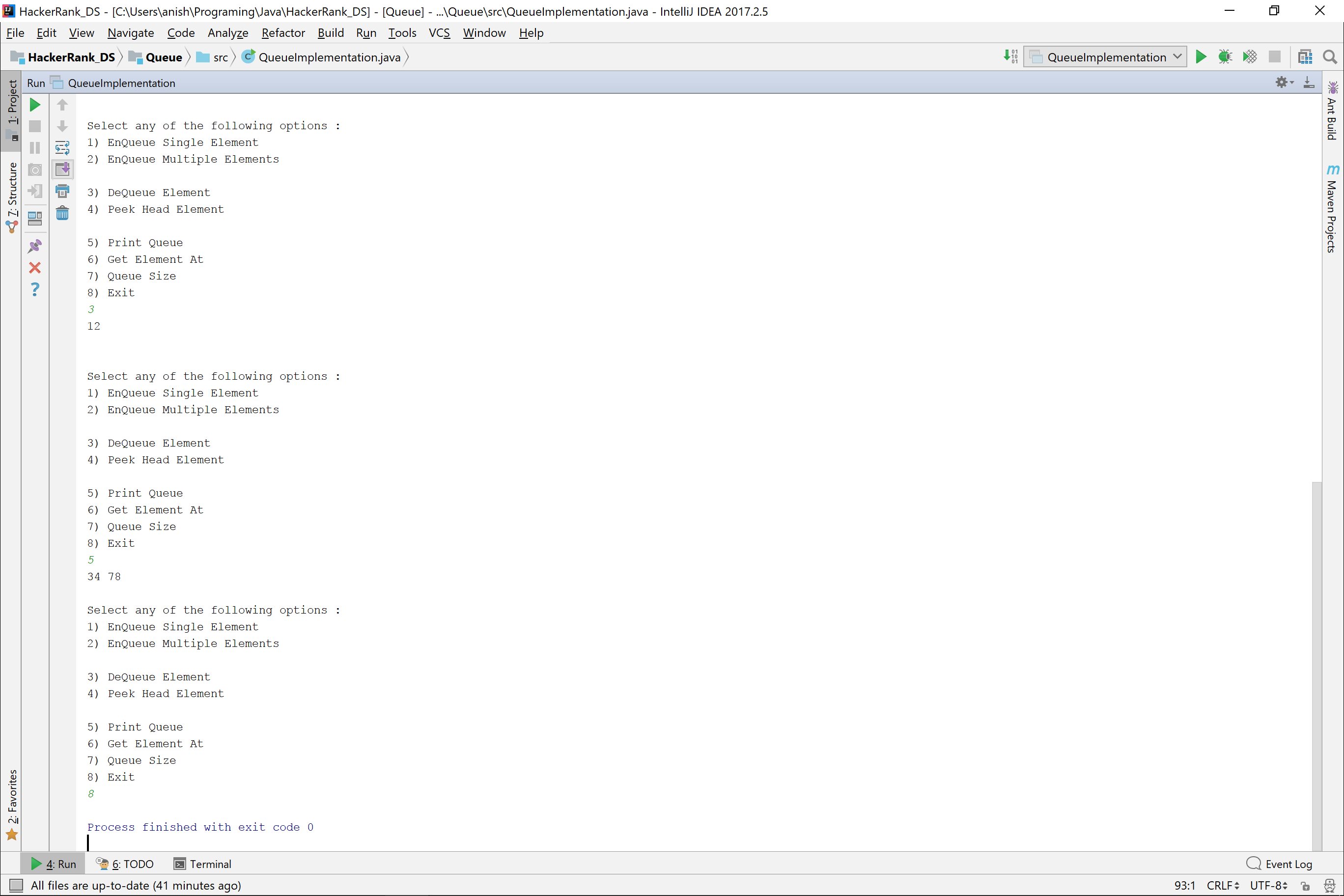
**public class** StringOperations\_Index {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.println(**"Enter Text : "**);  
 String text = *in*.nextLine();  
 System.***out***.print(**"Enter String to be searched : "**);  
 String string = *in*.next();  
  
 **int** index = *stringSearch*(text, string);  
 **if**(index != -1) System.***out***.println(string + **" exist at index : "** + index + **" in \""** + text + **"\""**);  
 **else** System.***out***.println(string + **" doesn't exist in "** + text);  
 }  
  
 **private static int** stringSearch(String text, String string){  
 **for**(**int** index1=0 ; index1<text.length()-string.length() ; index1++){  
 **if**(text.charAt(index1) == string.charAt(0)){  
 **if**(string.substring(1).equals(text.substring(index1+1, index1 + string.length())))  
 **return** index1;  
 }  
 }  
 **return** -1;  
 }  
}



Queue Implementation

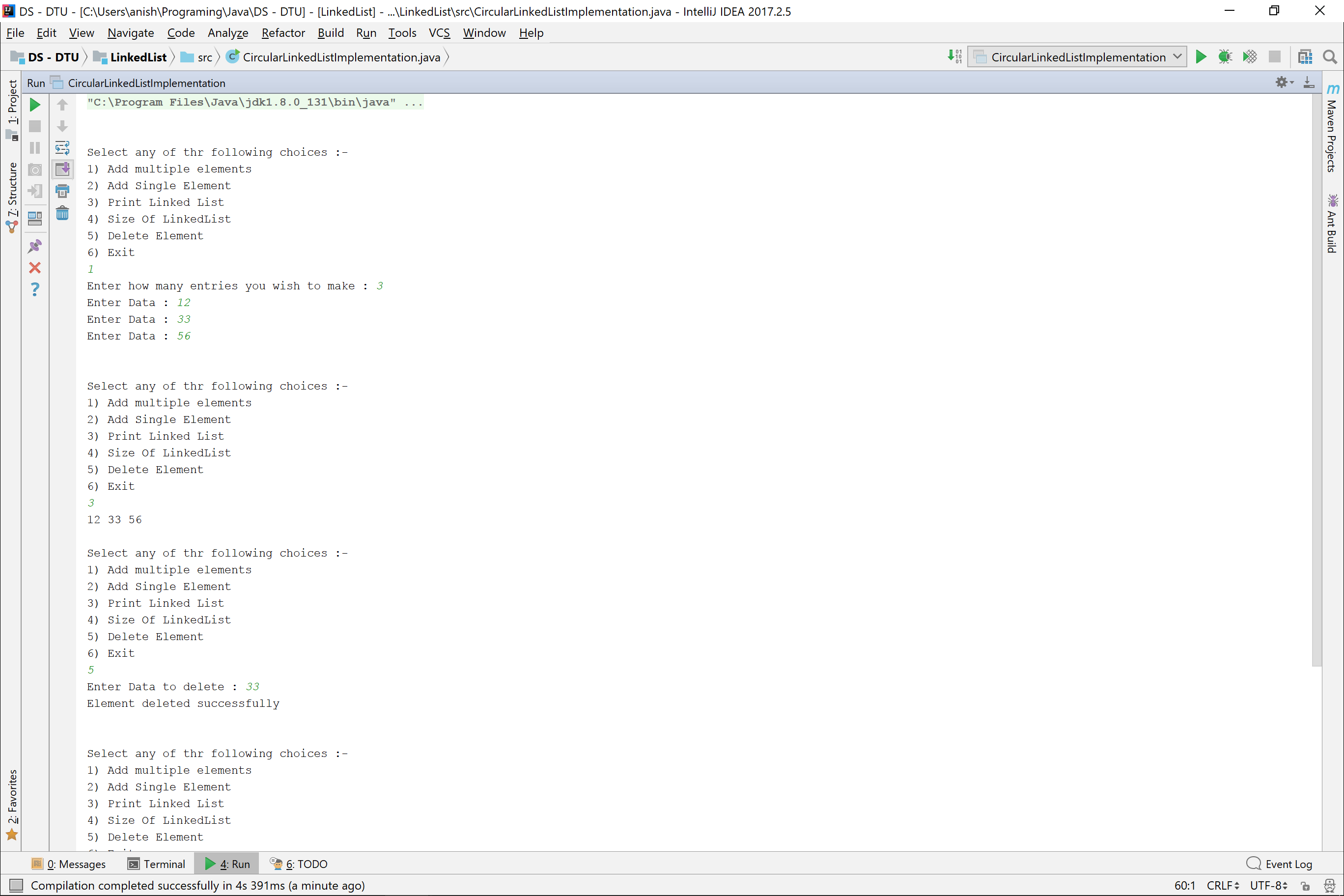
**import** java.util.Scanner;  
  
**class** QueueOverFlowException **extends** Exception{  
 QueueOverFlowException(){}  
 QueueOverFlowException(**int** index){  
 System.***out***.println(**"QueueOverFlowException : "** + index);  
 }  
}  
  
**class** QueueUnderFlowException **extends** Exception{  
 QueueUnderFlowException(){}  
 QueueUnderFlowException(**int** index){  
 System.***out***.println(**"QueueUnderFlowException : "** + index);  
 }  
}  
  
**class** QueueIndexOutOfBoundsException **extends** Exception{  
 QueueIndexOutOfBoundsException(){}  
 QueueIndexOutOfBoundsException(**int** index){  
 System.***out***.println(**"QueueIndexOutOfBoundsException : "** + index);  
 }  
}  
  
**class** Queue{  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
 **private int size**=0;  
 **private int arr**[];  
 **private int head** =0, **tail**=-1, **length**;  
 **private final double LOAD\_FACTOR** = 0.8;  
 **private final int MAX\_SIZE** = 500;  
  
 Queue(){  
 **arr** = **new int**[10];  
 **length** = **arr**.**length**;  
 }  
 Queue(**int** size){  
 **arr** = **new int**[size];  
 **length** = size;  
 }  
  
 *//Insertion Methods* **public void** insertSingle() **throws** QueueOverFlowException{  
 **if**(**size** + 1 >= **MAX\_SIZE**)  
 **throw new** QueueOverFlowException(**size**+1);  
 insert();  
 }  
  
 **public void** insertMultiple() **throws** QueueOverFlowException{  
 System.***out***.print(**"Enter no. of elements you wish to enter : "**);  
 **int** numberOfElements = **in**.nextInt();  
 **if**(numberOfElements + **size** >= **MAX\_SIZE**)  
 **throw new** QueueOverFlowException(numberOfElements + **size**);  
  
 **while** (numberOfElements-- > 0){  
 insert();  
 }  
 }  
  
 **private void** insert(){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = **in**.nextInt();  
 insertData(data);  
 }  
  
 **private void** insertData(**int** data){  
 **arr**[(**tail**+1) % **length**] = data;  
 **tail**++;  
 **size**++;  
 **if**((**double**)**size** / **arr**.**length** >= **LOAD\_FACTOR**)  
 grow();  
 }  
  
 **private void** grow(){  
 **int** temp[] = **new int**[2 \* **arr**.**length**];  
 addTo(temp);  
 **arr** = temp;  
 **tail** = **size** - 1;  
 **length** = 2\***length**;  
 **head** = 0;  
 }  
  
 **private void** addTo(**int** temp[]){  
 **for**(**int** index=**head**, tempIndex=0 ; tempIndex < **size** ; index++, tempIndex++) {  
 temp[tempIndex] = **arr**[index % **length**];  
 }  
 }  
  
 *//Removal and Peeking methods* **public int** deQueue() **throws** QueueUnderFlowException{  
 **if**(**size** == 0)  
 **throw new** QueueUnderFlowException();  
 **int** data = **arr**[**head** % **length**];  
 **head**++;  
 **size**--;  
 **return** data;  
 }  
  
 **public int** peek(){  
 **return arr**[**head** % **length**];  
 }  
  
 *//Displaying Methods* **public void** printQueue(){  
 **for**(**int** index=0, queueIndex=**head** ; index<**size** ; index++, queueIndex++){  
 System.***out***.print(**arr**[queueIndex%**length**] + **" "**);  
 }  
 }  
  
 **public int** get(**int** index) **throws** QueueIndexOutOfBoundsException{  
 **if**((**head** + index) % **length** >= **size**)  
 **throw new** QueueIndexOutOfBoundsException((**head** + index) % **length**);  
 **return arr**[ (**head** + index) % **length**];  
 }  
  
 **public int** queueSize(){  
 System.***out***.println(**length**);  
 **return size**;  
 }  
}  
  
**public class** QueueImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) **throws** QueueOverFlowException, QueueUnderFlowException, QueueIndexOutOfBoundsException{  
 Queue queue = **new** Queue();  
 **int** option;  
  
 **do** {  
 System.***out***.println(**"\n\nSelect any of the following options : "**);  
 System.***out***.println(**"1) EnQueue Single Element"**);  
 System.***out***.println(**"2) EnQueue Multiple Elements\n"**);  
  
 System.***out***.println(**"3) DeQueue Element "**);  
 System.***out***.println(**"4) Peek Head Element\n"**);  
  
 System.***out***.println(**"5) Print Queue"**);  
 System.***out***.println(**"6) Get Element At"**);  
 System.***out***.println(**"7) Queue Size"**);  
 System.***out***.println(**"8) Exit"**);  
 option = *in*.nextInt();  
  
 **switch** (option){  
 **case** 1 :  
 queue.insertSingle();  
 **break**;  
 **case** 2 :  
 queue.insertMultiple();  
 **break**;  
 **case** 3 :  
 *print*(queue.deQueue());  
 **break**;  
 **case** 4 :  
 *print*(queue.peek());  
 **break**;  
 **case** 5 :  
 queue.printQueue();  
 **break**;  
 **case** 6 :  
 System.***out***.print(**"Enter Index : "**);  
 **int** index = *in*.nextInt();  
 *print*(**"The element at "** + index + **" is : "**, queue.get(index));  
 **break**;  
 **case** 7 :  
 *print*(**"Queue Size, No. of elements = "**, queue.queueSize());  
 }  
 }**while** (option != 8);  
 }  
  
 **private static void** print(**int** data){  
 *print*(**""**, data);  
 }  
  
 **private static void** print(String string, **int** data){  
 System.***out***.println(string + data);  
 }  
}

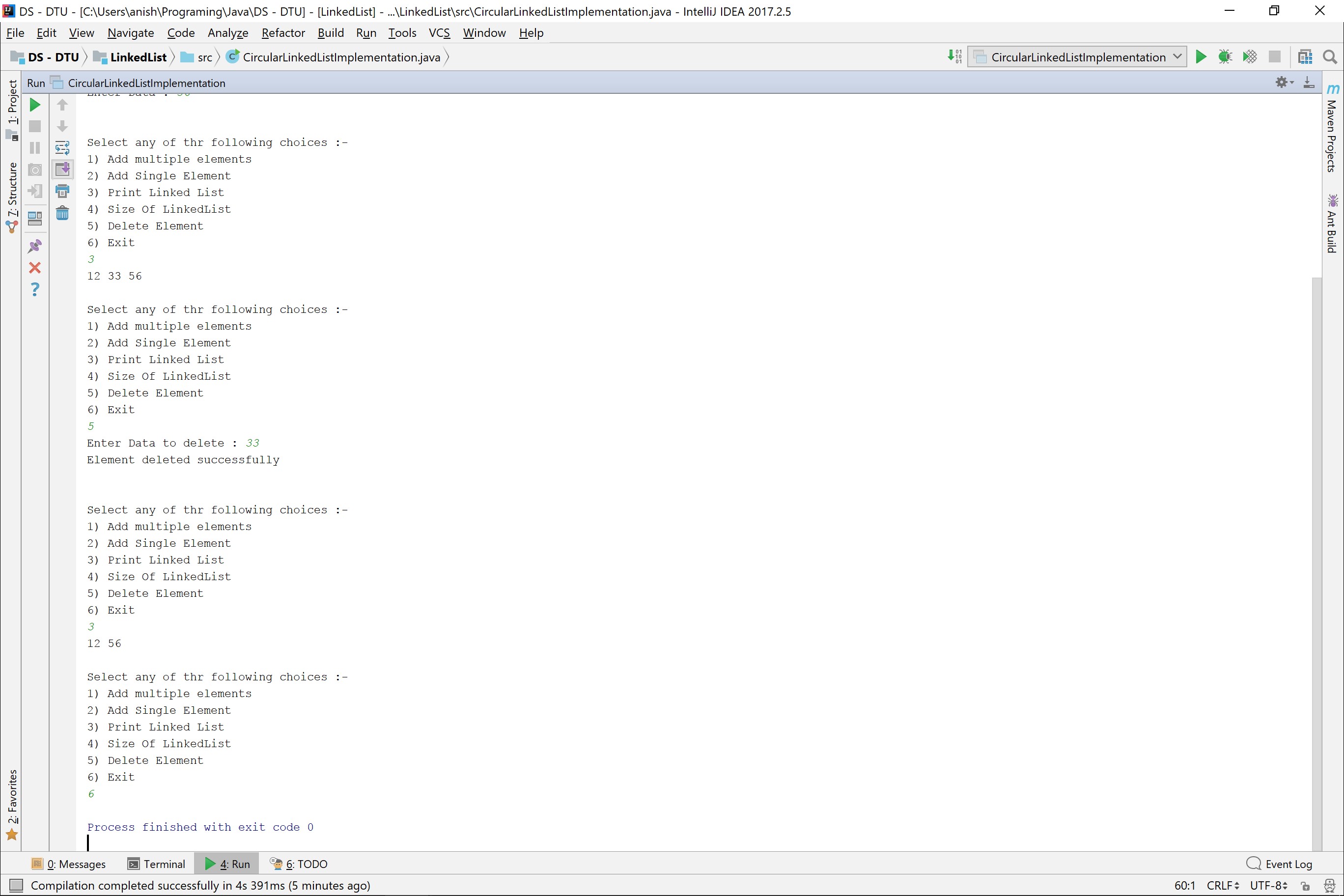




Circular Queue Implementation

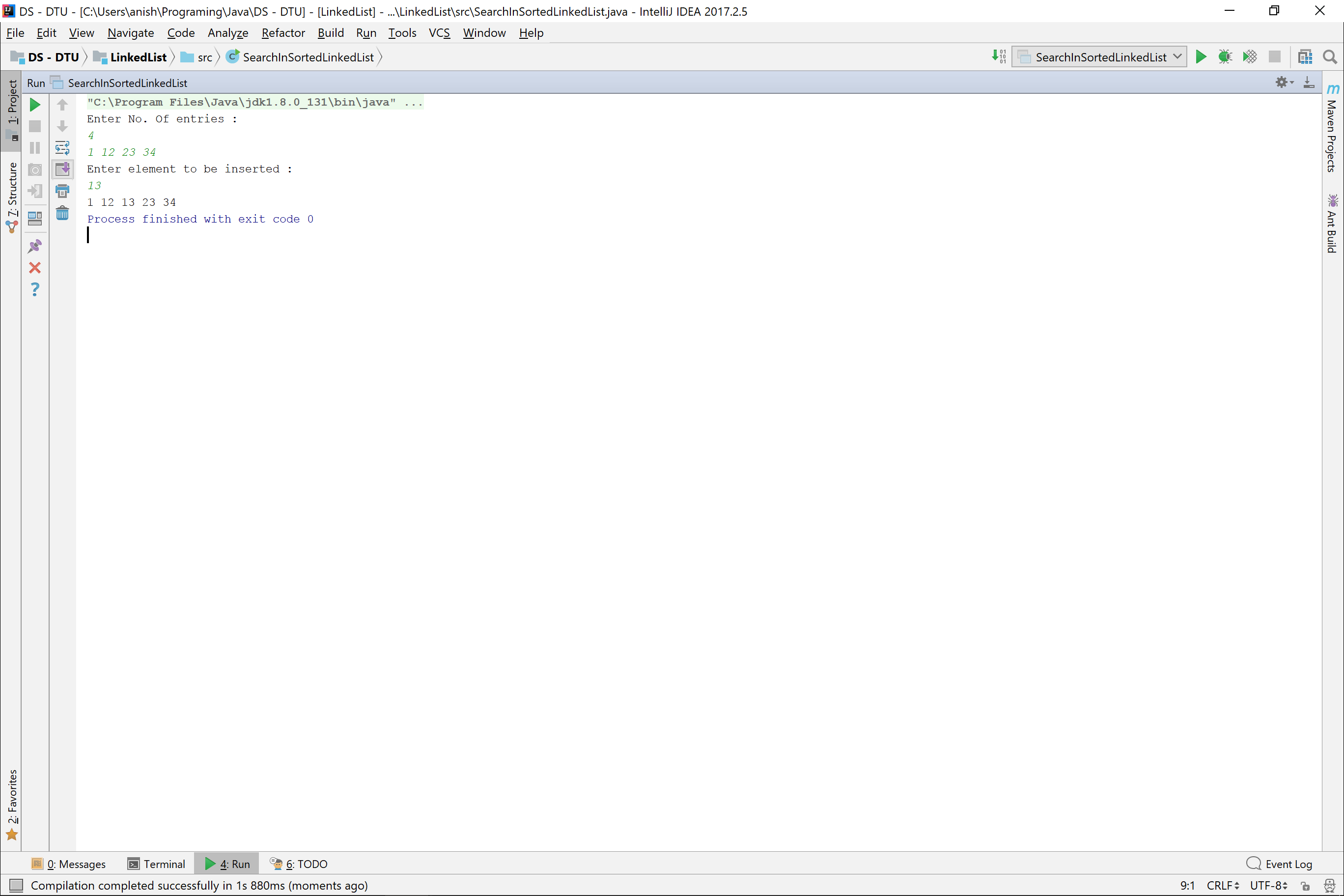
**import** java.util.Scanner;  
  
**class** QueueOverFlowException **extends** Exception{  
 QueueOverFlowException(){}  
 QueueOverFlowException(**int** index){  
 System.***out***.println(**"QueueOverFlowException : "** + index);  
 }  
}  
  
**class** QueueUnderFlowException **extends** Exception{  
 QueueUnderFlowException(){}  
 QueueUnderFlowException(**int** index){  
 System.***out***.println(**"QueueUnderFlowException : "** + index);  
 }  
}  
  
**class** QueueIndexOutOfBoundsException **extends** Exception{  
 QueueIndexOutOfBoundsException(){}  
 QueueIndexOutOfBoundsException(**int** index){  
 System.***out***.println(**"QueueIndexOutOfBoundsException : "** + index);  
 }  
}  
  
**class** Queue{  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
 **private int size**=0;  
 **private int arr**[];  
 **private int head** =0, **tail**=-1, **length**;  
 **private final double LOAD\_FACTOR** = 0.8;  
 **private final int MAX\_SIZE** = 500;  
  
 Queue(){  
 **arr** = **new int**[10];  
 **length** = **arr**.**length**;  
 }  
 Queue(**int** size){  
 **arr** = **new int**[size];  
 **length** = size;  
 }  
  
 *//Insertion Methods* **public void** insertSingle() **throws** QueueOverFlowException{  
 **if**(**size** + 1 >= **MAX\_SIZE**)  
 **throw new** QueueOverFlowException(**size**+1);  
 insert();  
 }  
  
 **public void** insertMultiple() **throws** QueueOverFlowException{  
 System.***out***.print(**"Enter no. of elements you wish to enter : "**);  
 **int** numberOfElements = **in**.nextInt();  
 **if**(numberOfElements + **size** >= **MAX\_SIZE**)  
 **throw new** QueueOverFlowException(numberOfElements + **size**);  
  
 **while** (numberOfElements-- > 0){  
 insert();  
 }  
 }  
  
 **private void** insert(){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = **in**.nextInt();  
 insertData(data);  
 }  
  
 **private void** insertData(**int** data){  
 **arr**[(**tail**+1) % **length**] = data;  
 **tail**++;  
 **size**++;  
 **if**((**double**)**size** / **arr**.**length** >= **LOAD\_FACTOR**)  
 grow();  
 }  
  
 **private void** grow(){  
 **int** temp[] = **new int**[2 \* **arr**.**length**];  
 addTo(temp);  
 **arr** = temp;  
 **tail** = **size** - 1;  
 **length** = 2\***length**;  
 **head** = 0;  
 }  
  
 **private void** addTo(**int** temp[]){  
 **for**(**int** index=**head**, tempIndex=0 ; tempIndex < **size** ; index++, tempIndex++) {  
 temp[tempIndex] = **arr**[index % **length**];  
 }  
 }  
  
 *//Removal and Peeking methods* **public int** deQueue() **throws** QueueUnderFlowException{  
 **if**(**size** == 0)  
 **throw new** QueueUnderFlowException();  
 **int** data = **arr**[**head** % **length**];  
 **head**++;  
 **size**--;  
 **return** data;  
 }  
  
 **public int** peek(){  
 **return arr**[**head** % **length**];  
 }  
  
 *//Displaying Methods* **public void** printQueue(){  
 **for**(**int** index=0, queueIndex=**head** ; index<**size** ; index++, queueIndex++){  
 System.***out***.print(**arr**[queueIndex%**length**] + **" "**);  
 }  
 }  
  
 **public int** get(**int** index) **throws** QueueIndexOutOfBoundsException{  
 **if**((**head** + index) % **length** >= **size**)  
 **throw new** QueueIndexOutOfBoundsException((**head** + index) % **length**);  
 **return arr**[ (**head** + index) % **length**];  
 }  
  
 **public int** queueSize(){  
 System.***out***.println(**length**);  
 **return size**;  
 }  
}  
  
**public class** QueueImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) **throws** QueueOverFlowException, QueueUnderFlowException, QueueIndexOutOfBoundsException{  
 Queue queue = **new** Queue();  
 **int** option;  
  
 **do** {  
 System.***out***.println(**"\n\nSelect any of the following options : "**);  
 System.***out***.println(**"1) EnQueue Single Element"**);  
 System.***out***.println(**"2) EnQueue Multiple Elements\n"**);  
  
 System.***out***.println(**"3) DeQueue Element "**);  
 System.***out***.println(**"4) Peek Head Element\n"**);  
  
 System.***out***.println(**"5) Print Queue"**);  
 System.***out***.println(**"6) Get Element At"**);  
 System.***out***.println(**"7) Queue Size"**);  
 System.***out***.println(**"8) Exit"**);  
 option = *in*.nextInt();  
  
 **switch** (option){  
 **case** 1 :  
 queue.insertSingle();  
 **break**;  
 **case** 2 :  
 queue.insertMultiple();  
 **break**;  
 **case** 3 :  
 *print*(queue.deQueue());  
 **break**;  
 **case** 4 :  
 *print*(queue.peek());  
 **break**;  
 **case** 5 :  
 queue.printQueue();  
 **break**;  
 **case** 6 :  
 System.***out***.print(**"Enter Index : "**);  
 **int** index = *in*.nextInt();  
 *print*(**"The element at "** + index + **" is : "**, queue.get(index));  
 **break**;  
 **case** 7 :  
 *print*(**"Queue Size, No. of elements = "**, queue.queueSize());  
 }  
 }**while** (option != 8);  
 }  
  
 **private static void** print(**int** data){  
 *print*(**""**, data);  
 }  
  
 **private static void** print(String string, **int** data){  
 System.***out***.println(string + data);  
 }  
}





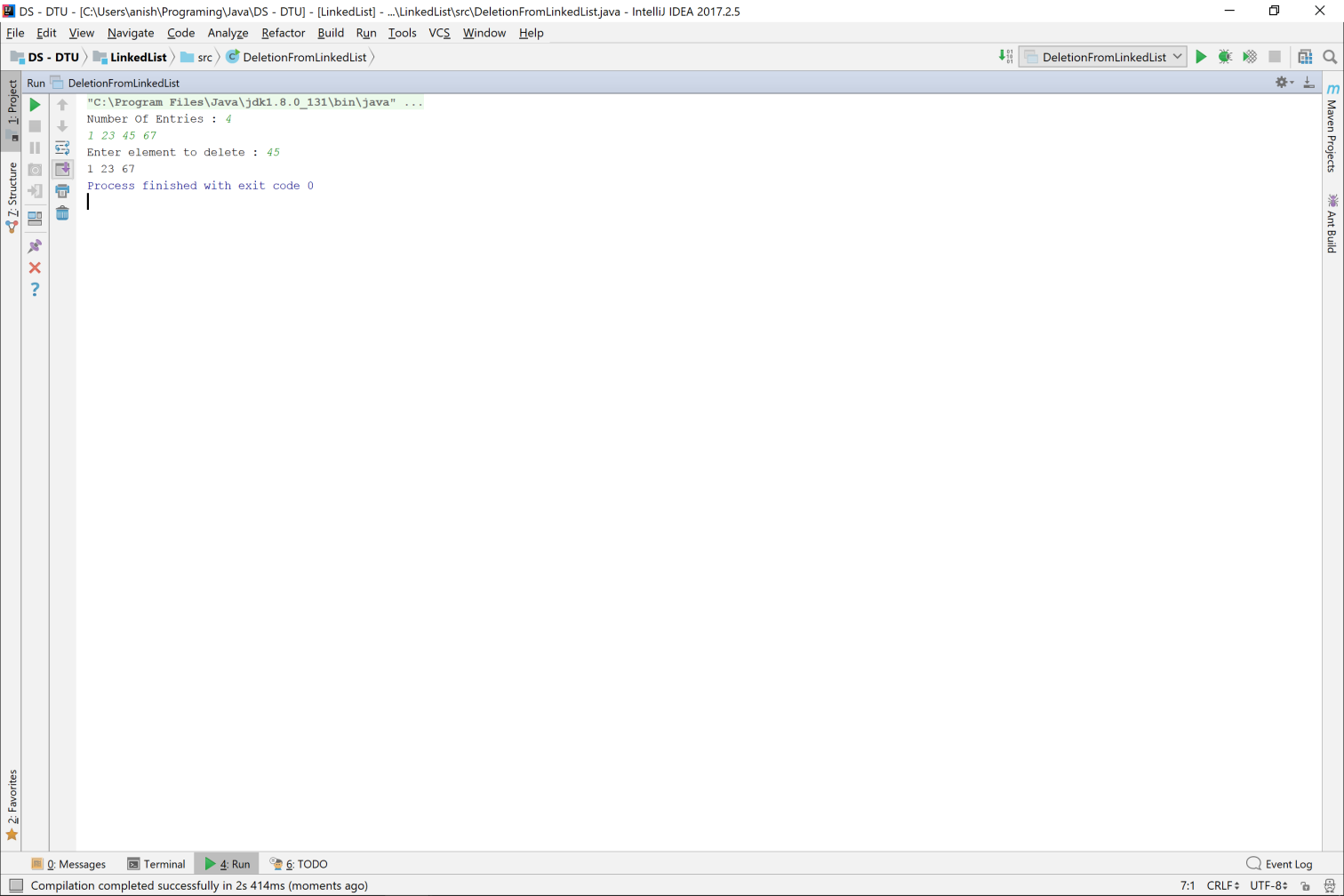
Insertion in sorted linked list

**public class** SearchInSortedLinkedList {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 Node head = **new** Node(0);  
 Node temp = head;  
  
 System.***out***.println(**"Enter No. Of entries : "**);  
 **int** numberOfEntries = *in*.nextInt();  
 **while** (numberOfEntries-- > 0){  
 **int** element = *in*.nextInt();  
 temp.**next** = **new** Node(element);  
 temp = temp.**next**;  
 }  
  
 System.***out***.println(**"Enter element to be inserted : "**);  
 **int** element = *in*.nextInt();  
 temp = head;  
 **while** (temp.**next** != **null** && element > temp.**next**.**data**){  
 temp = temp.**next**;  
 }  
 Node ans = **new** Node(element);  
 ans.**next** = temp.**next**;  
 temp.**next** = ans;  
  
 *print*(head);  
 }  
  
 **private static void** print(Node head){  
 Node temp = head.**next**;  
 **while** (temp != **null**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**next**;  
 }  
 }  
}



Deletion From Linked List

**public class** DeletionFromLinkedList {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 Node head = **new** Node(0), temp = head;  
 System.***out***.print(**"Number Of Entries : "**);  
 **int** numberOfEntries = *in*.nextInt();  
  
 **while** (numberOfEntries-- > 0){  
 **int** element = *in*.nextInt();  
 temp.**next** = **new** Node(element);  
 temp = temp.**next**;  
 }  
  
 System.***out***.print(**"Enter element to delete : "**);  
 **int** element = *in*.nextInt();  
  
 temp = head;  
 **while** (temp.**next** != **null**){  
 **if**(temp.**next**.**data** == element){  
 temp.**next** = temp.**next**.**next**;  
 }temp = temp.**next**;  
 }  
  
 *print*(head);  
 }  
  
 **private static void** print(Node head){  
 Node temp = head.**next**;  
 **while** (temp != **null**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**next**;  
 }  
 }  
}



Doubly Linked List Implementation

**import** java.util.Scanner;  
  
**public class** DoublyLinkedList {  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
 **private class** Node{  
 **int data**;  
 Node **prev**, **next**;  
  
 Node(){  
 **data** = 0;  
 **prev** = **next** = **null**;  
 }  
 Node(**int** data){  
 **this**.**data** = data;  
 **prev** = **next** = **null**;  
 }  
 }  
 **private int size** = 0;  
 **private** Node **head** = **new** Node(0);  
 **private** Node **slider** = **head**;  
  
 DoublyLinkedList(){}  
  
 **public void** insert(**int** data){  
 **slider**.**next** = **new** Node(data);  
 **slider**.**next**.**prev** = **slider**;  
 **slider** = **slider**.**next**;  
 **size**++;  
 }  
  
 **public int** remove() **throws** EmptyLinkedList{  
 **if**(**size** <= 0){  
 **throw new** EmptyLinkedList();  
 }  
  
 **int** data = **slider**.**data**;  
 **slider** = **slider**.**prev**;  
 **slider**.**next** = **null**;  
 **size**--;  
 **return** data;  
 }  
 **public boolean** remove(**int** data) **throws** EmptyLinkedList{  
 **if**(**size** <= 0){  
 **throw new** EmptyLinkedList();  
 }  
  
 Node temp = **head**.**next**;  
 **while** (temp.**next** != **null**){  
 **if**(temp.**next**.**data** == data){  
 temp.**next** = temp.**next**.**next**;  
 **if**(temp.**next** != **null**) temp.**next**.**prev** = temp;  
 **size**--;  
 **return true**;  
 }temp = temp.**next**;  
 }  
  
 **return false**;  
 }  
  
 **public void** print(){  
 Node temp = **head**.**next**;  
 **while** (temp != **null**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**next**;  
 }  
 System.***out***.println();  
 }  
 **public void** printReverse(){  
 Node temp = **slider**;  
 **while** (temp != **head**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**prev**;  
 }  
 System.***out***.println();  
 }  
  
 **public int** size(){  
 **return size**;  
 }  
  
 **public boolean** isEmpty(){  
 **return size** == 0;  
 }  
}  
  
**class** EmptyLinkedList **extends** Exception{  
 EmptyLinkedList(){  
 System.***out***.println(**"Linked List Empty Exception"**);  
 }  
}

**public class** DoublyLinkedListImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 *//Driver Function* **public static void** main(String[] args) **throws** EmptyLinkedList {  
 DoublyLinkedList linkedList = **new** DoublyLinkedList();  
  
 linkedList.insert(12);  
 linkedList.insert(34);  
 linkedList.insert(100);  
 linkedList.print();  
 linkedList.printReverse();  
  
 System.***out***.println(linkedList.remove());  
 System.***out***.println(linkedList.remove());  
  
 linkedList.insert(67);  
 linkedList.insert(90);  
 linkedList.insert(11);  
 linkedList.print();  
  
 System.***out***.println(linkedList.size());  
  
 **if**(linkedList.remove(11)) System.***out***.println(**"11 removed successfully"**);  
 **else** System.***out***.println(**"11 not found"**);  
 }  
}

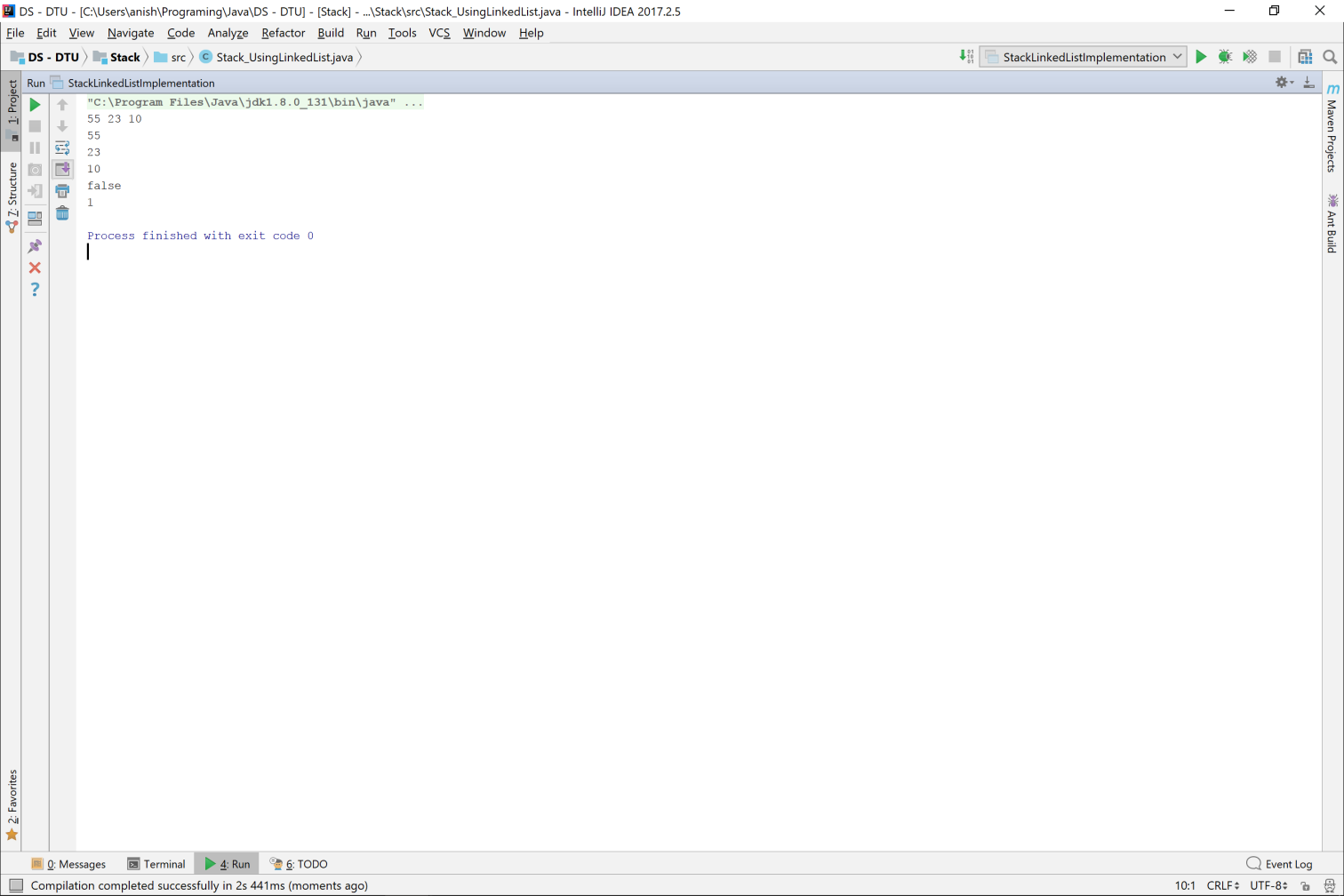


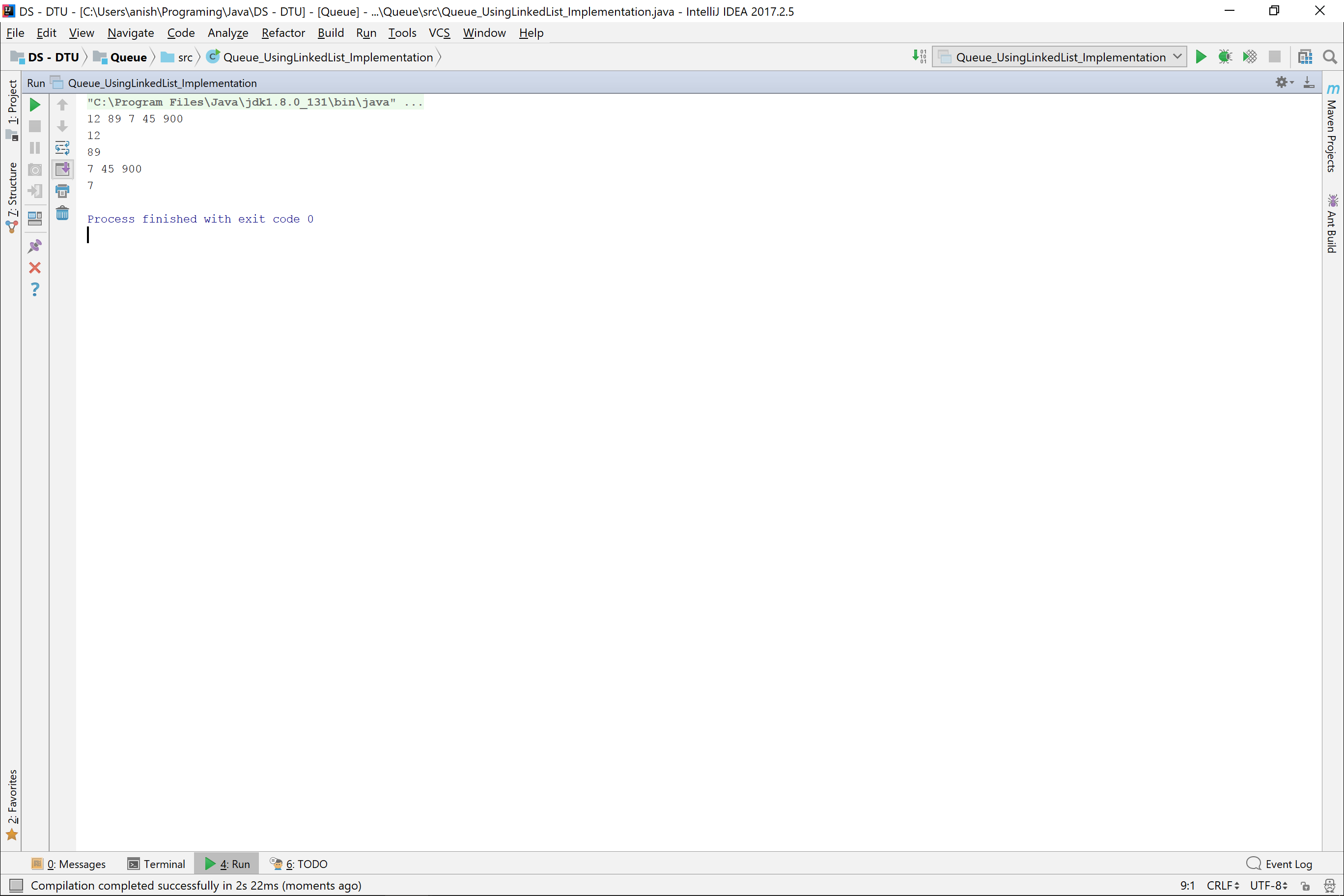


Stack using Linked List

**public class** Stack\_UsingLinkedList {  
 **private class** Node{  
 **int data**;  
 Node **next**;  
  
 Node(){  
 **next** = **null**;  
 **data** = 0;  
 }  
 Node(**int** data){  
 **this**.**data** = data;  
 **next** = **null**;  
 }  
 }  
 **private int size**=0;  
 **private** Node **head** = **new** Node(0);  
  
 **public void** push(**int** data){  
 Node ans = **new** Node(data);  
 ans.**next** = **head**.**next**;  
 **head**.**next** = ans;  
 **size**++;  
 }  
  
 **public int** peek() **throws** StackUnderFlowException{  
 **if**(**size** == 0){  
 **throw new** StackUnderFlowException();  
 }  
 **return head**.**next**.**data**;  
 }  
  
 **public int** pop() **throws** StackUnderFlowException{  
 **if**(**size** <= 0){  
 **throw new** StackUnderFlowException();  
 }  
  
 **int** element = **head**.**next**.**data**;  
 **head**.**next** = **head**.**next**.**next**;  
 **size**--;  
 **return** element;  
 }  
  
 **public void** print(){  
 Node temp = **head**.**next**;  
 **while** (temp != **null**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**next**;  
 }  
 System.***out***.println();  
 }  
  
 **public boolean** isEmpty(){  
 **return size** == 0;  
 }  
  
 **public int** size(){  
 **return size**;  
 }  
}  
  
  
**class** StackUnderFlowException **extends** Exception{  
 StackUnderFlowException(){  
 System.***out***.println(**"StackUnderFlowException"**);  
 }  
}

**public class** StackLinkedListImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 *//Driver Function* **public static void** main(String[] args) **throws** StackUnderFlowException {  
 Stack\_UsingLinkedList stack = **new** Stack\_UsingLinkedList();  
  
 stack.push(10);  
 stack.push(23);  
 stack.push(55);  
 stack.print();  
  
 System.***out***.println(stack.pop());  
 System.***out***.println(stack.pop());  
 stack.print();  
  
 System.***out***.println(stack.isEmpty());  
 System.***out***.println(stack.size());  
 }  
}

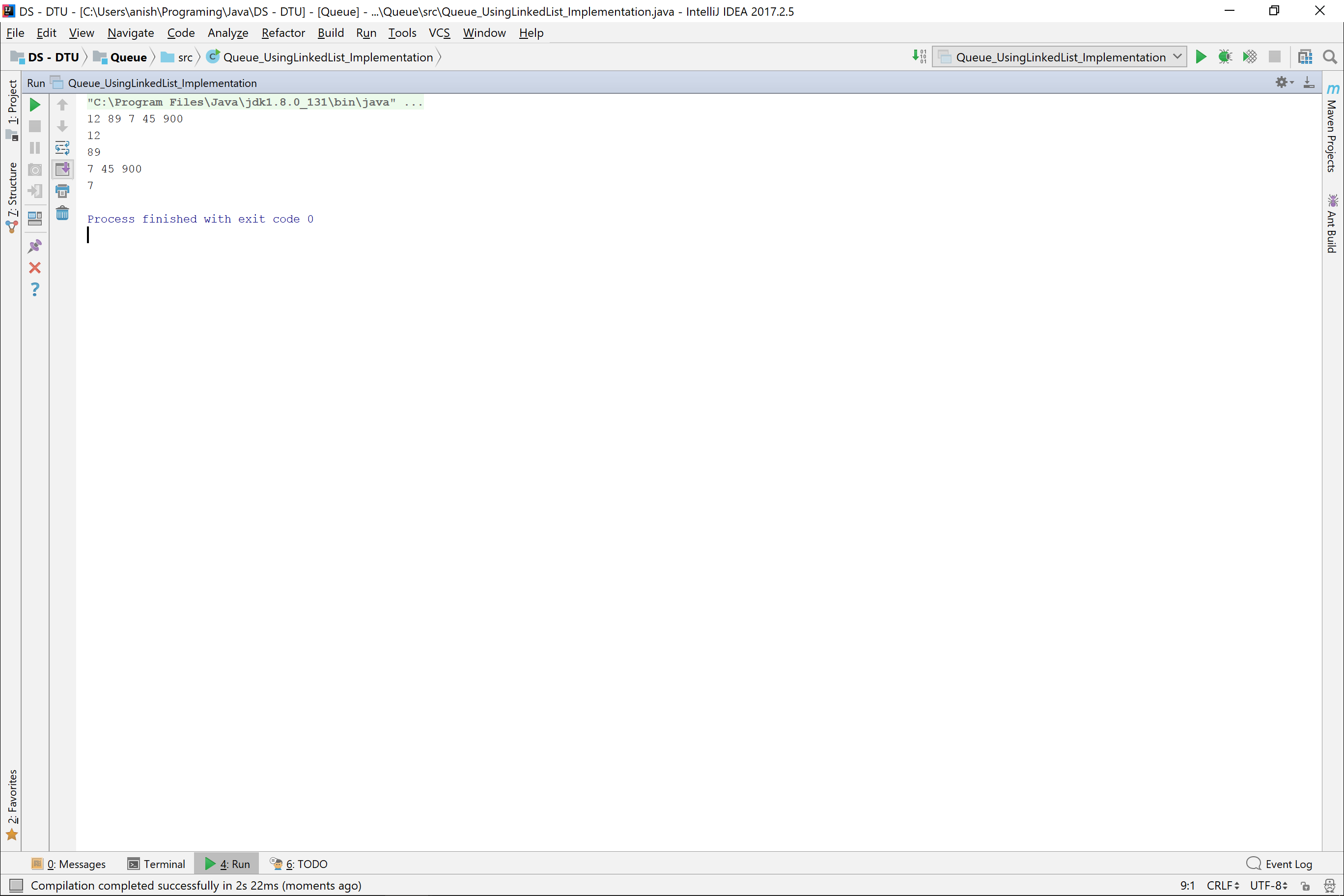




Queue using Linked List

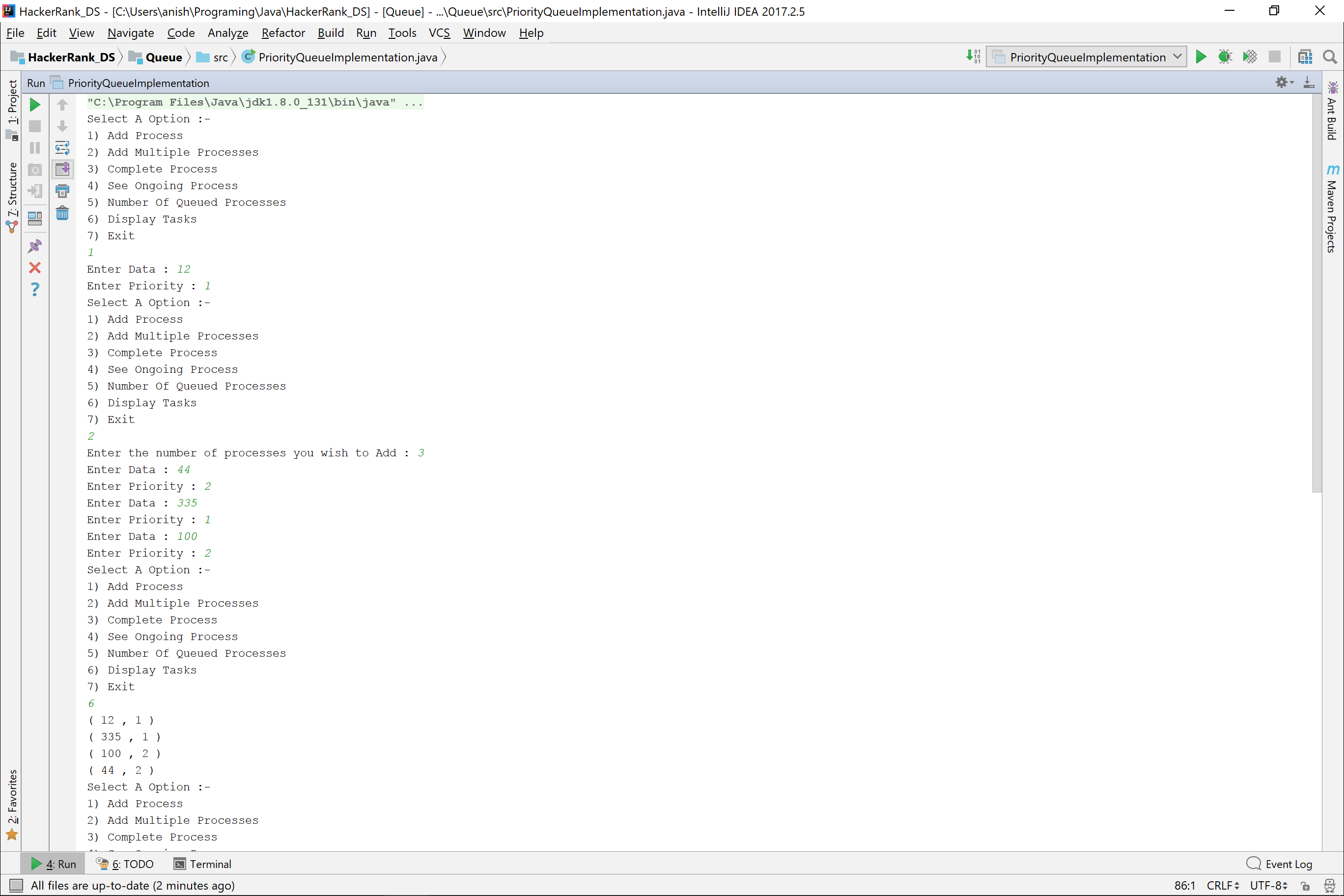
**public class** Queue\_UsingLinkedList {  
 **private class** Node{  
 **int data**;  
 Node **next**;  
  
 Node(){  
 **data** = 0;  
 **next** = **null**;  
 }  
 Node(**int** data){  
 **this**.**data** = data;  
 **next** = **null**;  
 }  
 }  
 **private int size** = 0;  
 **private** Node **head** = **new** Node(0), **slider** = **head**;  
  
 **public void** enQueue(**int** data){  
 **slider**.**next** = **new** Node(data);  
 **slider** = **slider**.**next**;  
 **size**++;  
 }  
  
 **public int** deQueue() **throws** QueueUnderFlowException{  
 **if**(**size** <= 0){  
 **throw new** QueueUnderFlowException();  
 }  
 **int** data = **head**.**next**.**data**;  
 **head**.**next** = **head**.**next**.**next**;  
 **size**--;  
 **return** data;  
 }  
  
 **public int** top() **throws** QueueUnderFlowException{  
 **if**(**size** <= 0){  
 **throw new** QueueUnderFlowException();  
 }  
 **return head**.**next**.**data**;  
 }  
  
 **public boolean** isEmpty(){  
 **return size** == 0;  
 }  
  
 **public int** size(){  
 **return size**;  
 }  
  
 @Override  
 **public** String toString() {  
 String ans = **""**;  
 Node temp = **head**.**next**;  
 **while** (temp != **null**){  
 ans = ans + temp.**data** + **" "**;  
 temp = temp.**next**;  
 }  
  
 **return** ans;  
 }  
}  
  
**class** QueueUnderFlowException **extends** Exception{  
 QueueUnderFlowException(){  
 System.***out***.println(**"QueueUnderFlowException"**);  
 }  
}

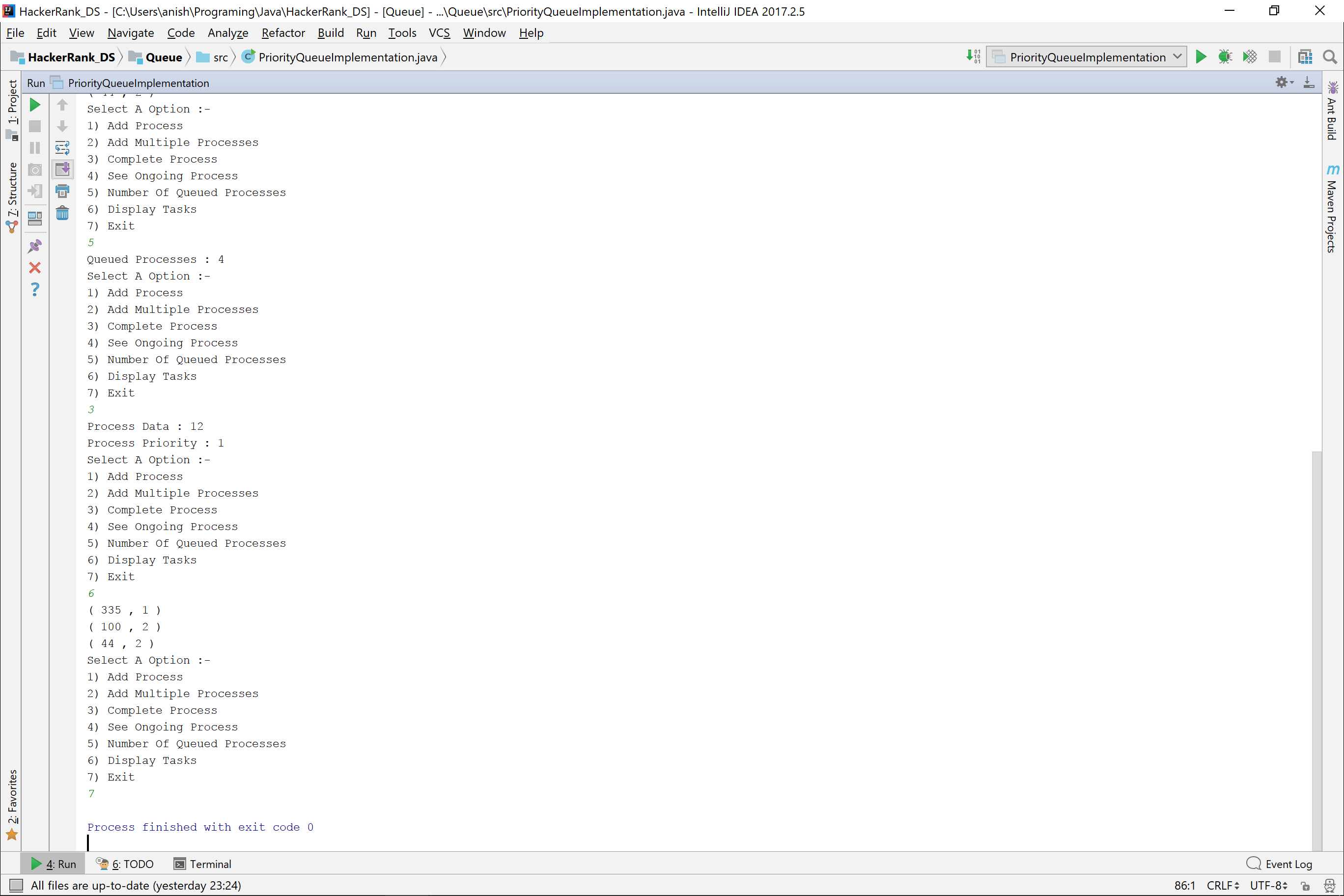
**public class** Queue\_UsingLinkedList\_Implementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 *//Driver Functions* **public static void** main(String[] args) **throws** QueueUnderFlowException {  
 Queue\_UsingLinkedList queue = **new** Queue\_UsingLinkedList();  
  
 queue.enQueue(12);  
 queue.enQueue(89);  
 queue.enQueue(7);  
 queue.enQueue(45);  
 queue.enQueue(900);  
 System.***out***.println(queue);  
  
 System.***out***.println(queue.deQueue());  
 System.***out***.println(queue.deQueue());  
 System.***out***.println(queue);  
  
 System.***out***.println(queue.top());  
 }  
}



Priority Queue using Linked List

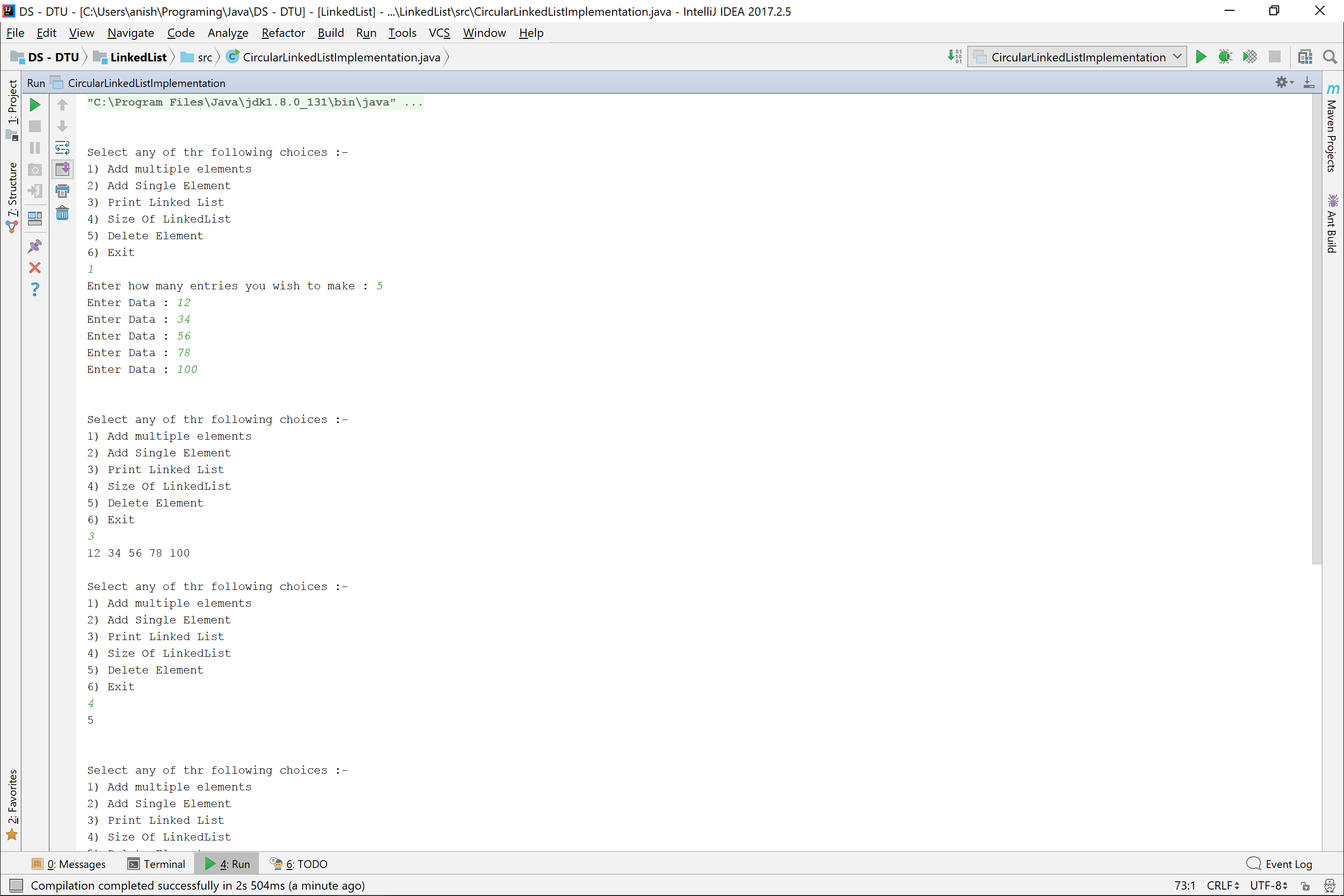
**import** java.util.Scanner;  
  
*//Exceptions***class** PriorityQueueUnderFlowException **extends** Exception {  
 PriorityQueueUnderFlowException(){  
 System.***out***.println(**"PriorityQueueUnderFlowException"**);  
 }  
}  
  
  
**class** Process{  
 **private int data**;  
 **private int priority**;  
 **public** Process **next**;  
  
 Process(){}  
 Process(**int** priority){  
 **this**.**priority** = priority;  
 }  
 Process(**int** data, **int** priority){  
 **this**.**data** = data;  
 **this**.**priority** = priority;  
 }  
  
 **public int** getPriority(){  
 **return priority**;  
 }  
 **public int** getData(){  
 **return data**;  
 }  
}  
  
  
**class** LinkedList{  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
 **private** Process **head**;  
 **private int size**=0;  
  
 LinkedList(){  
 **head** = **new** Process();  
 }  
  
 **public void** addElement(){  
 addProcess();  
 }  
  
 **public void** addElements(){  
 System.***out***.print(**"How many data points do you wish to Add : "**);  
 **int** numberPoints = **in**.nextInt();  
 **while** (numberPoints-- > 0){  
  
 }  
 }  
  
 **private void** addProcess(){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = **in**.nextInt();  
 System.***out***.print(**"Enter Priority : "**);  
 **int** priority = **in**.nextInt();  
 Process process = **new** Process(data, priority);  
 **if**(**size** == 0)  
 **head** = process;  
 **else head**.**next** = process;  
  
 **size**++;  
 }  
  
 **public void** addProcess(Process process){  
 **if**(process.getPriority() < **head**.getPriority()){  
 process.**next** = **head**;  
 **head** = process;  
 }  
 Process temp = **head**;  
 **while** (temp.**next** != **null** && process.getPriority() > temp.**next**.getPriority()){  
 temp = temp.**next**;  
 }  
 process.**next** = temp.**next**;  
 temp.**next** = process;  
  
 **size**++;  
 }  
  
 **public void** display() **throws** PriorityQueueUnderFlowException, InterruptedException {  
 **if**(**size** == 0)  
 **throw new** PriorityQueueUnderFlowException();  
  
 Process process = **head**;  
 **while** (process != **null**){  
 print(process);  
 Thread.*sleep*(1000);  
 process = process.**next**;  
 }  
 }  
  
 **private void** print(Process process){  
 System.***out***.println(**"( "** + process.getData() + **" , "** + process.getPriority() + **" )"**);  
 }  
  
 **public** Process deQueue(){  
 Process process = **new** Process(**head**.getData(), **head**.getPriority());  
 **head** = **head**.**next**;  
 **size**--;  
 **return** process;  
 }  
  
 **public** Process peek(){  
 **return new** Process(**head**.getData(), **head**.getPriority());  
 }  
  
 **public int** getSize(){  
 **return size**;  
 }  
}  
  
**class** PriorityQueue{  
 **private** LinkedList **priorityQueue**;  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
  
 PriorityQueue(){  
 **priorityQueue** = **new** LinkedList();  
 }  
  
 **public void** addElement(){  
 **if**(**priorityQueue**.getSize() == 0){  
 **priorityQueue**.addElement();  
 } **else** {  
 Process process = getProcess();  
 **priorityQueue**.addProcess(process);  
 }  
 }  
  
 **public void** addElements(){  
 System.***out***.print(**"Enter the number of processes you wish to Add : "**);  
 **int** numberProcesses = **in**.nextInt();  
 **while** (numberProcesses -- > 0){  
 addElement();  
 }  
 }  
  
 **private** Process getProcess(){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = **in**.nextInt();  
 System.***out***.print(**"Enter Priority : "**);  
 **int** priority = **in**.nextInt();  
 **return new** Process(data, priority);  
 }  
  
 **public** Process doTask() **throws** PriorityQueueUnderFlowException{  
 **if**(**priorityQueue**.getSize() == 0)  
 **throw new** PriorityQueueUnderFlowException();  
 **return priorityQueue**.deQueue();  
 }  
  
 **public** Process currentTask() **throws** PriorityQueueUnderFlowException{  
 **if**(**priorityQueue**.getSize() == 0)  
 **throw new** PriorityQueueUnderFlowException();  
 **return priorityQueue**.peek();  
 }  
  
 **public int** numberOfQueuedTasks(){  
 **return priorityQueue**.getSize();  
 }  
  
 **public void** displayTasks() **throws** PriorityQueueUnderFlowException, InterruptedException{  
 **priorityQueue**.display();  
 }  
}  
  
**public class** PriorityQueueImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) **throws** PriorityQueueUnderFlowException, InterruptedException {  
 PriorityQueue priorityQueue = **new** PriorityQueue();  
 **int** choice;  
  
 **do** {  
 System.***out***.println(**"Select A Option :- "**);  
 System.***out***.println(**"1) Add Process"**);  
 System.***out***.println(**"2) Add Multiple Processes"**);  
 System.***out***.println(**"3) Complete Process"**);  
 System.***out***.println(**"4) See Ongoing Process"**);  
 System.***out***.println(**"5) Number Of Queued Processes "**);  
 System.***out***.println(**"6) Display Tasks"**);  
 System.***out***.println(**"7) Exit"**);  
 choice = *in*.nextInt();  
  
 **switch** (choice){  
 **case** 1 :  
 priorityQueue.addElement();  
 **break**;  
 **case** 2 :  
 priorityQueue.addElements();  
 **break**;  
 **case** 3 :  
 Process task = priorityQueue.doTask();  
 *print*(task);  
 **break**;  
 **case** 4 :  
 task = priorityQueue.currentTask();  
 *print*(task);  
 **break**;  
 **case** 5 :  
 *print*(priorityQueue.numberOfQueuedTasks());  
 **break**;  
 **case** 6 :  
 priorityQueue.displayTasks();  
 **break**;  
 }  
  
 } **while** (choice != 7);  
 }  
  
 **private static void** print(Process process){  
 System.***out***.println(**"Process Data : "** + process.getData());  
 System.***out***.println(**"Process Priority : "** + process.getPriority());  
 }  
  
 **private static void** print(**int** size){  
 System.***out***.println(**"Queued Processes : "** + size);  
 }  
}





Circular Linked List Implementation

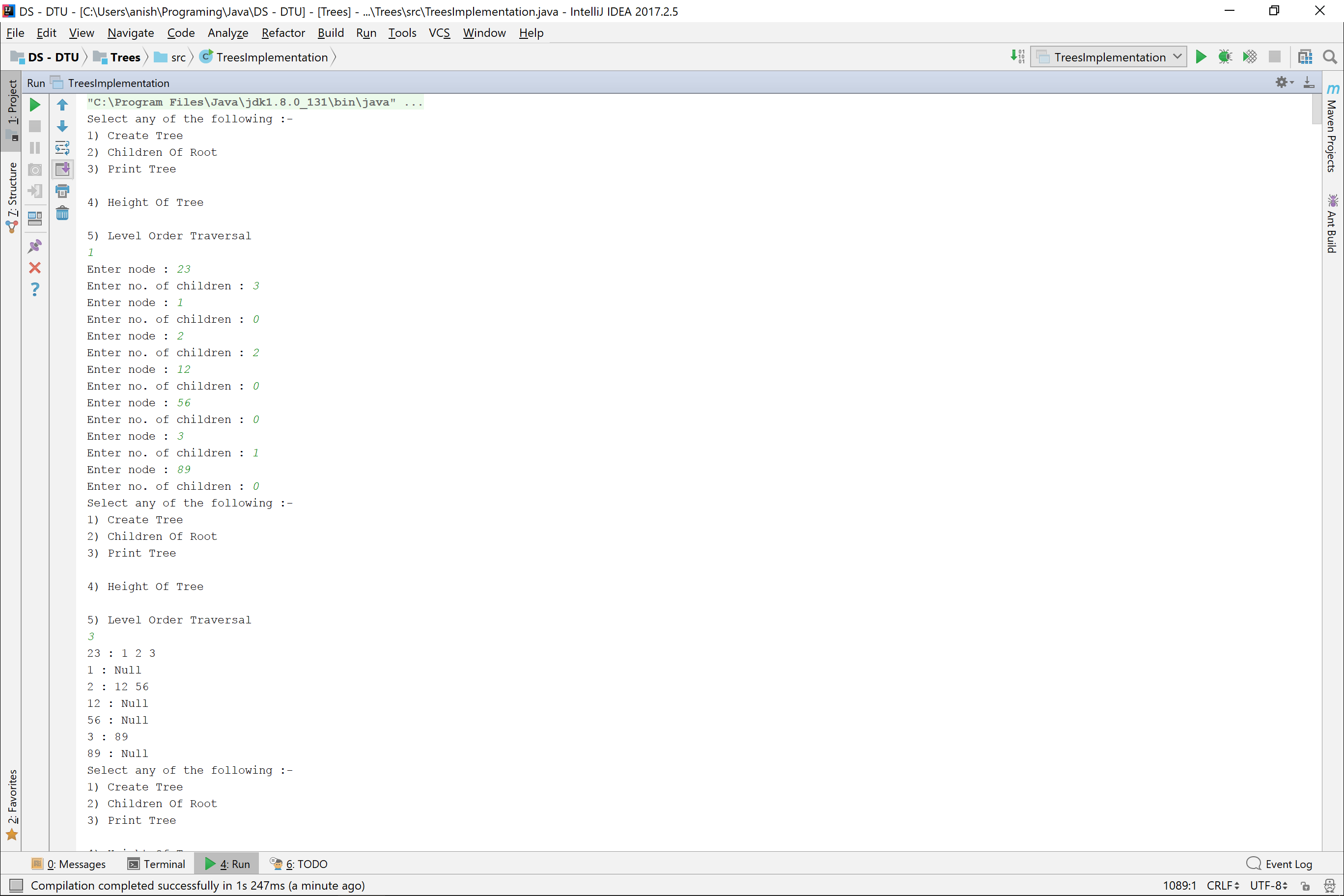
**import** java.util.Scanner;  
  
**class** Node{  
 **int data**;  
 Node **next**;  
  
 Node(**int** data){  
 **this**.**data** = data;  
 }  
}  
  
**class** CircularLinkedList{  
 **private** Scanner **in** = **new** Scanner(System.***in***);  
 **private** Node **head**, **current**;  
 **private int size**;  
  
 CircularLinkedList(){  
 **head** = **new** Node(0);  
 **size**=0;  
 }  
  
 **public void** insertMultipleData(){  
 System.***out***.print(**"Enter how many entries you wish to make : "**);  
 **int** n = **in**.nextInt();  
 **while** (n-- > 0){  
 insertData();  
 }  
 }  
 **public void** insertData(){  
 System.***out***.print(**"Enter Data : "**);  
 **int** data = **in**.nextInt();  
 **if**(**size** == 0){  
 **head**.**next** = **new** Node(data);  
 **current** = **head**.**next**;  
 **current**.**next** = **head**;  
 } **else** {  
 **current**.**next** = **new** Node(data);  
 **current** = **current**.**next**;  
 **current**.**next** = **head**;  
 }  
 **size**++;  
 }  
  
 **public void** print(){  
 Node temp = **head**.**next**;  
 **while** (temp != **head**){  
 System.***out***.print(temp.**data** + **" "**);  
 temp = temp.**next**;  
 }  
 }  
  
 **public int** getSize(){  
 **return size**;  
 }  
}  
  
**public class** CircularLinkedListImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 **int** choice;  
 CircularLinkedList linkedList = **new** CircularLinkedList();  
 **do** {  
 System.***out***.println(**"\n\nSelect any of thr following choices :-"**);  
 System.***out***.println(**"1) Add multiple elements"**);  
 System.***out***.println(**"2) Add Single Element"**);  
 System.***out***.println(**"3) Print Linked List"**);  
 System.***out***.println(**"4) Size Of LinkedList"**);  
 System.***out***.println(**"5) Exit"**);  
 choice = *in*.nextInt();  
  
 **switch** (choice){  
 **case** 1 :  
 linkedList.insertMultipleData();  
 **break**;  
 **case** 2 :  
 linkedList.insertData();  
 **break**;  
 **case** 3 :  
 linkedList.print();  
 **break**;  
 **case** 4 :  
 System.***out***.println(linkedList.getSize());  
 **break**;  
 }  
 }**while** (choice != 5);  
 }  
}

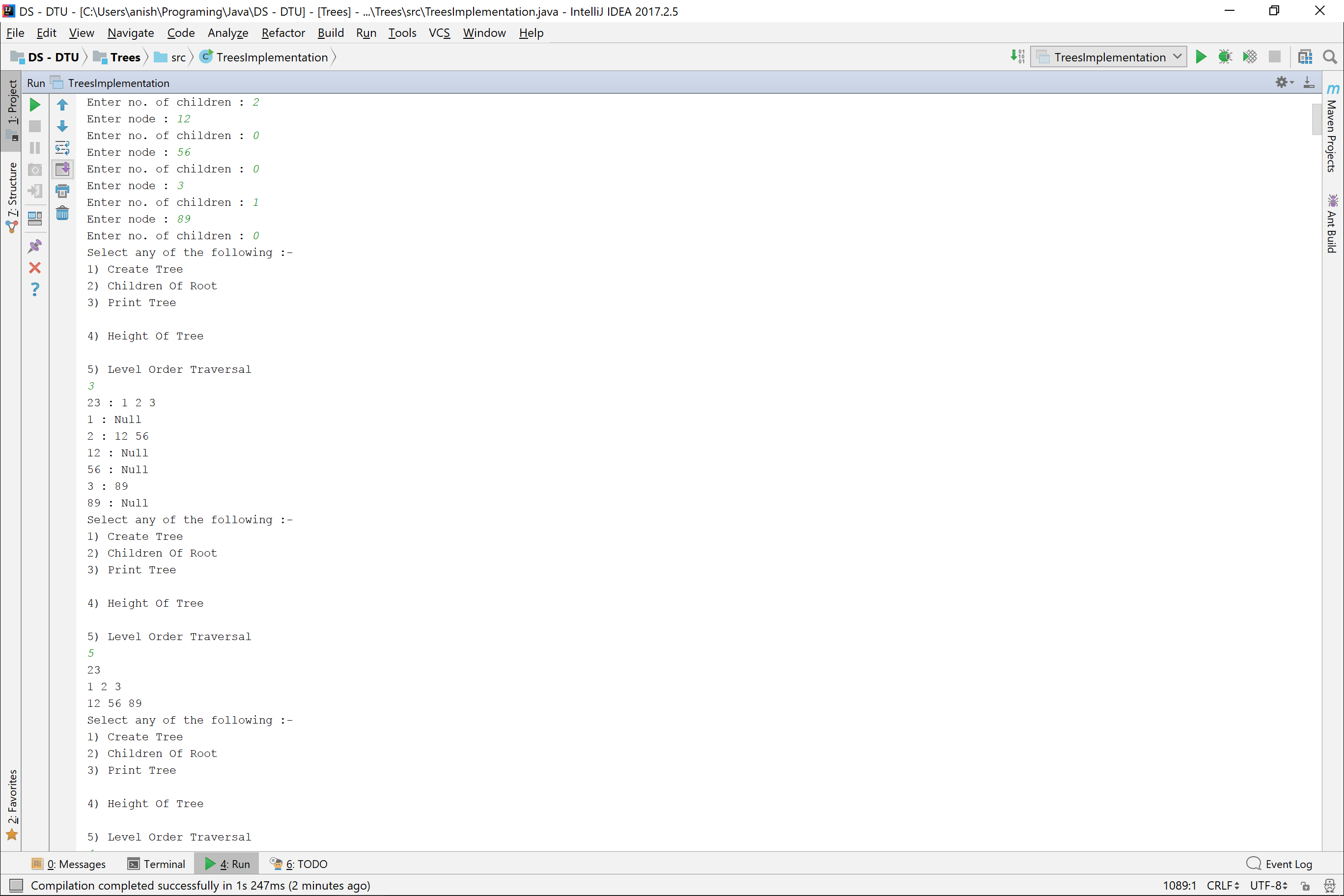


Tree

**public class** Tree {  
 *//Tree Node Properties* **private** ArrayList<Tree> **children**;  
 **private int data**;  
  
 *//Tree Properties* **private int height**; **private boolean isHeightUpdated** = **false**;  
 **private int diameter**; **private boolean isDiameterUpdated** = **false**;  
 **private boolean isBST**; **private boolean isBSTUpdated** = **false**;  
  
 *//Constructors* Tree(){  
 **children** = **new** ArrayList<>();  
 **data** = 0;  
 }  
 Tree(**int** data){  
 **children** = **new** ArrayList<>();  
 **this**.**data** = data;  
 }  
  
 *//Data and Children Assignment and Getting* **public boolean** assign(**int** data){  
 **this**.**data** = data;  
 **return true**;  
 }  
 **public int** getData(){ **return data**; }  
 **public** Tree getChild(**int** index){  
 **return children**.get(index);  
 }  
 **public void** addChild(Tree child){  
 **children**.add(child);  
 **isHeightUpdated** = **false**;  
 }  
 **public int** childrenNumber(){  
 **return children**.size();  
 }  
  
  
 *//Printing Methods* **public void** print(){  
 *printTree*(**this**);  
 }  
 **private static void** printTree (Tree root) {  
 System.***out***.print(root.**data** + **" : "**);  
 **for**(**int** i=0 ; i<root.**children**.size() ; System.***out***.print(root.**children**.get(i).**data** + **" "**), i++);  
 System.***out***.print(root.**children**.size() == 0 ? **"Null"** : **""**);  
  
 System.***out***.println(**""**);  
 **for**(**int** i=0 ; i<root.**children**.size() ; *printTree*(root.**children**.get(i)), i++);  
 }  
  
  
 *//Tree Properties Methods* **public int** height(){ *//Tree Height* **if**(**isHeightUpdated**) **return height**;  
 **this**.**height** = height(**this**);  
 **isHeightUpdated** = **true**;  
 **return height**;  
 }  
 **private int** height(Tree root){  
 **int** height = 0;  
 **for**(**int** index=0 ; index<root.**children**.size() ; index++){  
 height = Math.*max*(height(**children**.get(index)), height);  
 }  
 **return** 1 + height;  
 }  
  
  
 **private class** TreeProperties{  
 **public int diameter** = 0;  
 **public int height** = 0;  
 TreeProperties(){}  
 TreeProperties(**int** diameter, **int** height){  
 **this**.**diameter** = diameter;  
 **this**.**height** = height;  
 }  
 }  
 **public int** diameter(){ *//Tree Diameter* **if**(**isDiameterUpdated**) **return diameter**;  
 **diameter** = diameter(**this**).**diameter**;  
 **isDiameterUpdated** = **true**;  
 **return diameter**;  
 }  
 **private** TreeProperties diameter(Tree root){  
 **if**(root == **null**){  
 **return new** TreeProperties(0, 0);  
 }  
  
 **int** height=0, diameter=0, index=0;  
 **for**(Tree child : root.**children**){  
 TreeProperties properties = diameter(child);  
 diameter = Math.*max*(diameter, properties.**diameter**);  
 **if**(index++ > 0){  
 diameter = (height + properties.**height** + 1 > diameter ? height + properties.**height** + 1 : diameter);  
 height = 1 + Math.*max*(height, properties.**height**);  
 } **else** {  
 diameter = Math.*max*(diameter, height + 1);  
 }  
 }  
  
 **return new** TreeProperties(diameter, height);  
 }  
  
  
 **public** ArrayList<ArrayList<Tree>> levelOrderTraversal(){ *//Level Order Traversal* **return** levelOrderTraversal(**this**);  
 }  
 **private** ArrayList<ArrayList<Tree>> levelOrderTraversal(Tree root){  
 LinkedList<Tree> levelOrder = levelOrder(root);  
 ArrayList<ArrayList<Tree>> ans = **new** ArrayList<>();  
 ArrayList<Tree> level = **new** ArrayList<>();  
  
 **while** (!levelOrder.isEmpty()){  
 Tree tree = levelOrder.pop();  
 **if**(tree != **null**){  
 level.add(tree);  
 }  
 **else** {  
 ans.add(level);  
 level = **new** ArrayList<>();  
 }  
 }  
  
 **return** ans;  
 }  
 **private** LinkedList<Tree> levelOrder(Tree root){  
 LinkedList<Tree> linkedList = **new** LinkedList<>();  
 LinkedList<Tree> ans = **new** LinkedList<>();  
 linkedList.add(root);  
 linkedList.add(**null**);  
  
 **while** (!linkedList.isEmpty()){  
 Tree tree = linkedList.pop();  
 ans.add(tree);  
  
 **if**(tree != **null**)  
 linkedList.addAll(tree.**children**);  
 **else if**(linkedList.size() > 0) linkedList.add(**null**);  
 }  
 **return** ans;  
 }  
  
  
 **public static** Tree createTreeInorderAndPreOrder(**int** inOrder[], **int** preOrder[]){  
 **return** *createTreeInorderAndPreOrder*(inOrder, preOrder, 0, 0, inOrder.**length**-1);  
 }  
 **private static** Tree createTreeInorderAndPreOrder(**int** inOrder[], **int** preOrder[], **int** index, **int** startIndex, **int** endIndex){  
 **if**(index == preOrder.**length**){  
 **return null**;  
 }  
  
 **int** searchIndex = *linearSearch*(inOrder, preOrder[index], startIndex, endIndex);  
 Tree node = **new** Tree(inOrder[searchIndex]);  
 node.addChild(*createTreeInorderAndPreOrder*(inOrder, preOrder, index+1, startIndex, searchIndex-1));  
 node.addChild(*createTreeInorderAndPreOrder*(inOrder, preOrder, index+2, searchIndex+1, endIndex));  
  
 **return** node;  
 }  
 **private static int** linearSearch(**int** array[], **int** element, **int** startIndex, **int** endIndex){  
 **while** (startIndex <= endIndex){  
 **if**(array[startIndex++] == element)  
 **return** startIndex-1;  
 }  
 **return** -1;  
 }  
}

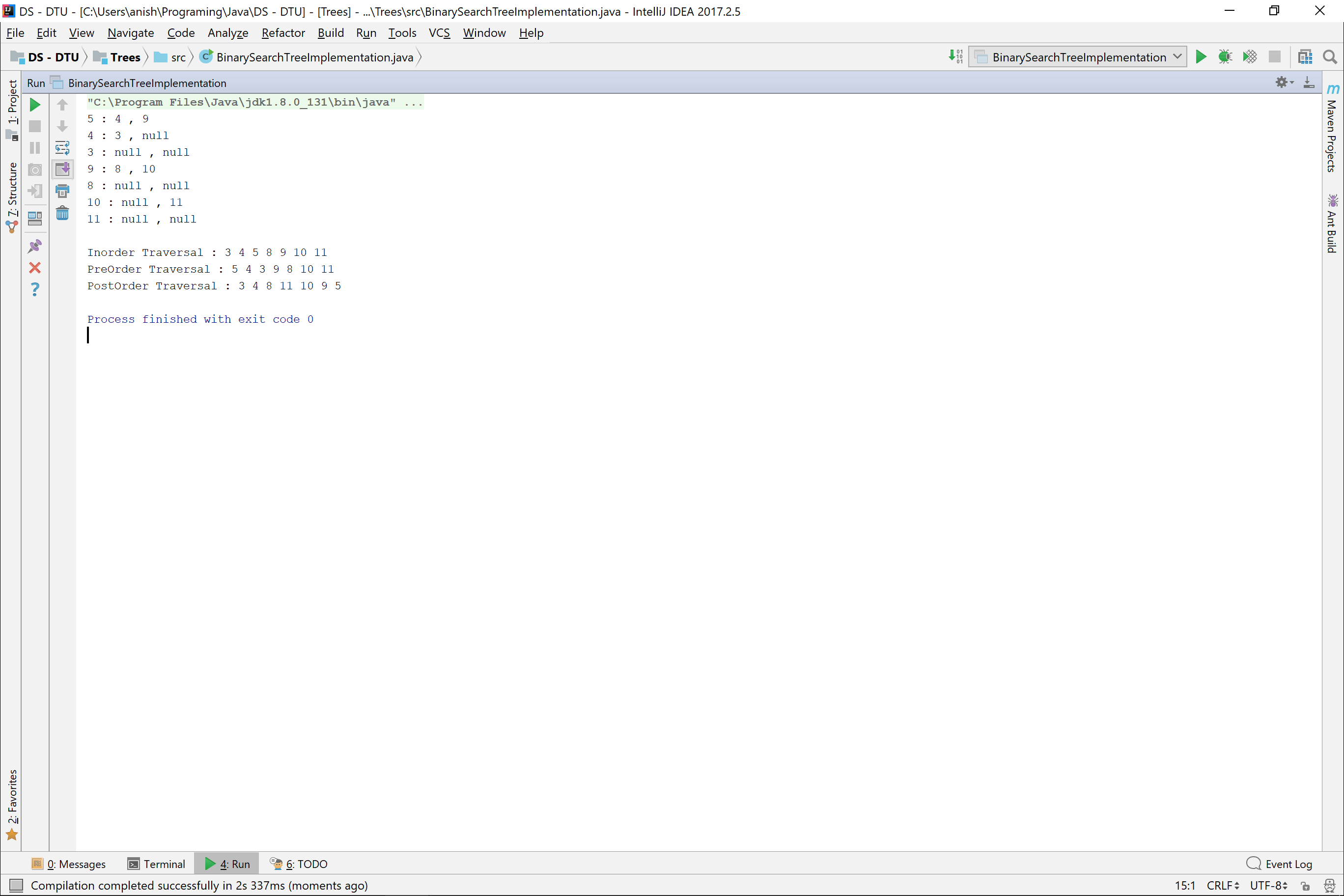
**public class** TreesImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 **int** choice;  
 Tree root = **new** Tree();  
  
 **do** {  
 System.***out***.println(**"Select any of the following :-"**);  
 System.***out***.println(**"1) Create Tree"**);  
 System.***out***.println(**"2) Children Of Root"**);  
 System.***out***.println(**"3) Print Tree"**);  
 System.***out***.println(**"\n4) Height Of Tree"**);  
 System.***out***.println(**"\n5) Level Order Traversal"**);  
 choice = *in*.nextInt();  
  
 **switch** (choice){  
 **case** 1 :  
 root = *createTree*();  
 **break**;  
 **case** 2 :  
 **int** children = root.childrenNumber();  
 System.***out***.println(children);  
 **break**;  
 **case** 3 :  
 root.print();  
 **break**;  
 **case** 4 :  
 System.***out***.println(root.height());  
 **break**;  
 **case** 5 :  
 ArrayList<ArrayList<Tree>> ans = root.levelOrderTraversal();  
 *print*(ans);  
 **break**;  
 }  
 } **while** (choice <= 5 );  
 }  
  
 **public static** Tree createTree () {  
 System.***out***.print(**"Enter node : "**);  
 **int** node = *in*.nextInt();  
  
 Tree root = **new** Tree(node);  
  
 System.***out***.print(**"Enter no. of children : "**);  
 **int** child = *in*.nextInt();  
  
 **for**(**int** i=0 ; i<child ; i++){  
 root.addChild(*createTree*());  
 }  
  
 **return** root;  
 }  
  
 **private static void** print(ArrayList<ArrayList<Tree>> list2D){  
 **for**(ArrayList<Tree> level : list2D){  
 *printList*(level);  
 }  
 }  
 **private static void** printList(ArrayList<Tree> level){  
 **for**(Tree tree : level){  
 System.***out***.print(tree.getData() + **" "**);  
 }  
 System.***out***.println();  
 }  
}





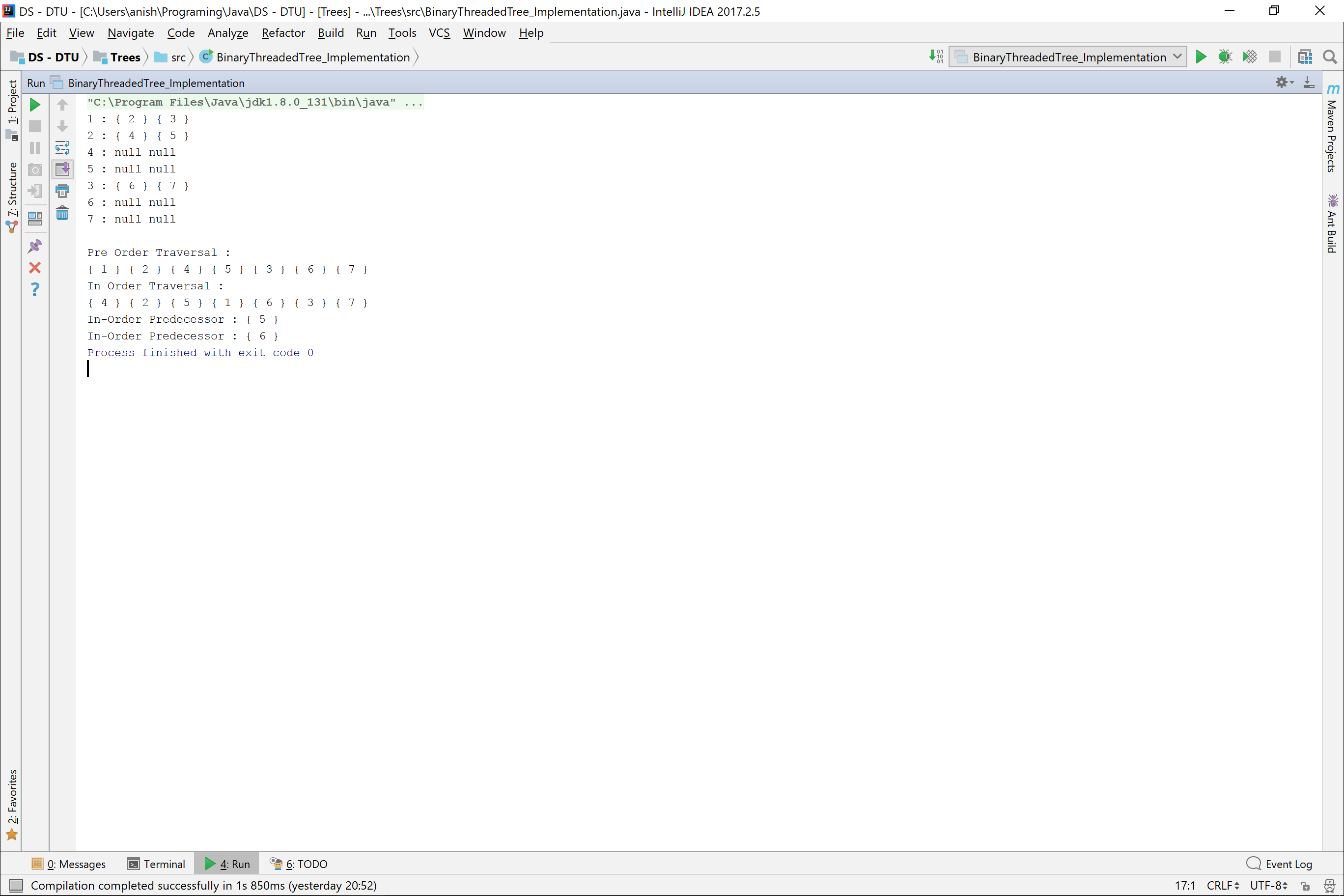
Binary Search Tree

*//Binary Search Tree Class***class** BinarySearchTree{  
 **private** Integer **data**;  
 **private** BinarySearchTree **left**, **right**;  
 **private int size**=0;  
 **private int height**=0;  
  
 *//Constructors* **public** BinarySearchTree(){  
 **left** = **right** = **null**;  
 **data** = **null**;  
 }  
 **private** BinarySearchTree(**int** data){  
 **this**.**data** = data;  
 **left** = **right** = **null**;  
 **size** = 1;  
 **height** = 1;  
 }  
  
 *//Adding Data and children Methods* **private void** assign(**int** data){  
 **if**(**this**.**data** == **null**) **size**++;  
 **this**.**data** = data;  
 }  
  
 **public void** add(**int** data){  
 add(**this**, data);  
 }  
 **private void** add(BinarySearchTree root, **int** data){  
 **if**(root.**data** == **null**){  
 root.assign(data);  
 **return**;  
 }  
  
 root.**size**++;  
  
 **if**(data > root.**data**){  
 **if**(root.**right** == **null**){  
 root.**right** = **new** BinarySearchTree(data);  
 root.**height** = 1 + Math.*max*(root.**left** == **null** ? 0 : root.**left**.**height**, root.**right**.**height**);  
 **return**;  
 }  
 add(root.**right**, data);  
 **return**;  
 }  
  
 **if**(root.**left** == **null**) {  
 root.**left** = **new** BinarySearchTree(data);  
 root.**height** = 1 + Math.*max*(root.**left**.**height** , root.**right** == **null** ? 0 : root.**right**.**height**);  
 **return**;  
 }  
 add(root.**left**, data);  
 root.**height** = 1 + Math.*max*(root.**left** == **null** ? 0 : root.**left**.**height**, root.**right** == **null** ? 0 : root.**right**.**height**);  
 }  
  
 *//Accessing Data and Children* **public** Integer data() {  
 **return data**;  
 }  
  
 **public** BinarySearchTree left() {  
 **return left**;  
 }  
  
 **public** BinarySearchTree right() {  
 **return right**;  
 }  
  
 *//Binary Search Tree Properties* **public int** size(){  
 **return size**;  
 }  
 **public int** height(){  
 **return height**;  
 }  
  
 *//Printing Methods* **public void** preOrderTraversal(){  
 preOrderTraversal(**this**);  
 System.***out***.println();  
 }  
 **private void** preOrderTraversal(BinarySearchTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 System.***out***.print(root.**data** + **" "**);  
 preOrderTraversal(root.**left**);  
 preOrderTraversal(root.**right**);  
 }  
  
 **public void** inOrderTraversal(){  
 inOrderTraversal(**this**);  
 System.***out***.println();  
 }  
 **private void** inOrderTraversal(BinarySearchTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 inOrderTraversal(root.**left**);  
 System.***out***.print(root.**data** + **" "**);  
 inOrderTraversal(root.**right**);  
 }  
  
 **public void** postOrderTraversal(){  
 postOrderTraversal(**this**);  
 System.***out***.println();  
 }  
 **private void** postOrderTraversal(BinarySearchTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 postOrderTraversal(root.**left**);  
 postOrderTraversal(root.**right**);  
 System.***out***.print(root.**data** + **" "**);  
 }  
  
 **public void** print(){  
 print(**this**);  
 System.***out***.println();  
 }  
 **private void** print(BinarySearchTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 System.***out***.println(root.**data** + **" : "** + (root.**left** == **null** ? **"null"** : root.**left**.**data**) + **" , "** + (root.**right** == **null** ? **"null"** : root.**right**.**data**) );  
 print(root.**left**);  
 print(root.**right**);  
 }  
  
 **public** ArrayList<ArrayList<BinarySearchTree>> levelOrder(){  
 **return** levelOrder(**this**);  
 }  
 **private** ArrayList<ArrayList<BinarySearchTree>> levelOrder(BinarySearchTree root){  
 LinkedList<BinarySearchTree> levelList = levelOrderTraversal(root);  
 ArrayList<ArrayList<BinarySearchTree>> levels = **new** ArrayList<>();  
 ArrayList<BinarySearchTree> level = **new** ArrayList<>();  
  
 **while** (!levelList.isEmpty()){  
 BinarySearchTree node = levelList.pop();  
  
 **if**(node == **null**){  
 levels.add(level);  
 level = **new** ArrayList<>();  
 }  
 **else** {  
 level.add(node);  
 }  
 }  
  
 **return** levels;  
 }  
 **private** LinkedList<BinarySearchTree> levelOrderTraversal(BinarySearchTree root){  
 LinkedList<BinarySearchTree> levelOrder = **new** LinkedList<>();  
 LinkedList<BinarySearchTree> ans = **new** LinkedList<>();  
 levelOrder.add(root);  
 levelOrder.add(**null**);  
  
 **while** (!levelOrder.isEmpty()){  
 BinarySearchTree node = levelOrder.pop();  
 ans.add(node);  
  
 **if**(node == **null**){  
 **if**(levelOrder.size() != 0) levelOrder.add(**null**);  
 } **else** {  
 **if**(node.**left** != **null**) levelOrder.add(node.**left**);  
 **if**(node.**right** != **null**) levelOrder.add(node.**right**);  
 }  
 }  
 **return** ans;  
 }  
}  
  
  
*//Driver Class***public class** BinarySearchTreeImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 BinarySearchTree tree = **new** BinarySearchTree();  
 tree.add(5);  
 tree.add(4);  
 tree.add(9);  
 tree.add(3);  
 tree.add(10);  
 tree.add(11);  
 tree.add(8);  
 tree.print();  
  
 System.***out***.print(**"Inorder Traversal : "**); tree.inOrderTraversal();  
 System.***out***.print(**"PreOrder Traversal : "**); tree.preOrderTraversal();  
 System.***out***.print(**"PostOrder Traversal : "**); tree.postOrderTraversal();  
 }  
}



Threaded Binary Tree

**import** java.util.ArrayList;  
**import** java.util.Scanner;  
*/\*  
 A Binary Threaded Tree extends the functionality of a Binary Tree by  
 utilizing the null left and right pointers in a Binary Tree Node  
 \*/***public class** BinaryThreadedTree {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 *//Internal Data* **private** Integer **data**;  
 **private** BinaryThreadedTree **left**, **right**;  
 **private int size**=0;  
 *//Boolean variables to see if the left branches are threads or not* **private boolean hasLeftThread**, **hasRightThread**;  
  
 *//Constructors* BinaryThreadedTree(){  
 **hasLeftThread** = **false**;  
 **hasRightThread** = **false**;  
 }  
  
 BinaryThreadedTree(**int** data){  
 **this**.**data** = data;  
 **hasRightThread** = **false**;  
 **hasLeftThread** = **false**;  
 }  
  
 *//Methods  
 //To create tree* **public void** create(){  
 create(**this**);  
 addThreads();  
 }  
 **private** BinaryThreadedTree create(BinaryThreadedTree root){  
 **int** data = *in*.nextInt();  
 **if**(data == -1){  
 **return null**;  
 }  
 root.**data** = data;  
 **size**++;  
  
 root.**left** = create(**new** BinaryThreadedTree());  
 root.**right** = create(**new** BinaryThreadedTree());  
  
 **return** root;  
 }  
 **private void** addThreads(){  
 ArrayList<BinaryThreadedTree> inOrder = inOrderSequence(**this**);  
 **for**(**int** index=0 ; index<inOrder.size() ; index++){  
 BinaryThreadedTree tree = inOrder.get(index);  
 *//Adding Left Thread* **if**(tree.**left** == **null**){  
 **if**(index > 0){  
 tree.**left** = inOrder.get(index-1);  
 tree.**hasLeftThread** = **true**;  
 }  
 }  
  
 *//Adding Right Thread* **if**(tree.**right** == **null**){  
 **if**(index < inOrder.size()-1){  
 tree.**right** = inOrder.get(index+1);  
 tree.**hasRightThread** = **true**;  
 }  
 }  
 }  
 }  
  
 *//Tree Creation from PreOrder and Inorder Traversal* **private static int** *index*=0;  
 **public static** BinaryThreadedTree create(**int** inOrder[], **int** preOrder[]){  
 *index* =0;  
 BinaryThreadedTree ans = *createTree*(inOrder, preOrder);  
 ans.addThreads();  
 *index*=0;  
 **return** ans;  
 }  
 **private static** BinaryThreadedTree createTree(**int** inOrder[], **int** preOrder[]){  
 **if**(inOrder.**length** == 1){  
 **return new** BinaryThreadedTree(inOrder[0]);  
 }  
  
 BinaryThreadedTree root = **new** BinaryThreadedTree(preOrder[*index*]);  
 **int** nodeIndex = *linearSearch*(preOrder[*index*], inOrder);  
 **int** inOrder1[] = **new int**[nodeIndex];  
 **int** inOrder2[] = **new int**[inOrder.**length** - inOrder1.**length** - 1];  
 *copy*(inOrder, inOrder1, 0, nodeIndex-1);  
 *copy*(inOrder, inOrder2, nodeIndex+1);  
  
 *index*++;  
 root.**left** = *createTree*(inOrder1, preOrder);  
 *index*++;  
 root.**right** = *createTree*(inOrder2, preOrder);  
  
 **return** root;  
 }  
 **private static int** linearSearch(**int** element, **int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 **if**(arr[index] == element)  
 **return** index;  
 }  
 **return** -1;  
 }  
 **private static void** copy(**int** original[], **int** newArray[], **int** startIndex, **int** endIndex){  
 **int** k=0;  
 **while** (startIndex <= endIndex){  
 newArray[k++] = original[startIndex++];  
 }  
 }  
 **private static void** copy(**int** original[], **int** newArray[], **int** startIndex){  
 *copy*(original, newArray, startIndex, original.**length**-1);  
 }  
  
 *//To Find InOrder Successor* **public** BinaryThreadedTree inOrderSuccessor(){  
 **return** inOrderSuccessor(**this**);  
 }  
 **public** BinaryThreadedTree inOrderSuccessor(BinaryThreadedTree root){  
 **if**(root.**hasRightThread**)  
 **return** root.**right**;  
 **if**(root.**right** == **null**)  
 **return null**;  
  
 root = root.**right**;  
 **while** (!root.**hasLeftThread**){  
 root = root.**left**;  
 }  
  
 **return** root;  
 }  
  
 *//To Find InOrder Predecessor* **public** BinaryThreadedTree inOrderPredecessor(){  
 **return** inOrderPredecessor(**this**);  
 }  
 **public** BinaryThreadedTree inOrderPredecessor(BinaryThreadedTree root){  
 **if**(root == **null** || root.**left** == **null**){  
 **return null**;  
 }  
 **if**(root.**hasLeftThread**){  
 **return** root.**left**;  
 }  
  
 root = root.**left**;  
 **while** (!root.**hasRightThread**){  
 root = root.**right**;  
 }  
  
 **return** root;  
 }  
  
 *//Traversals* **public** ArrayList<BinaryThreadedTree> inOrderSequence(BinaryThreadedTree root){  
 **if**(root == **null**){  
 **return new** ArrayList<>();  
 }  
  
 ArrayList<BinaryThreadedTree> leftList = inOrderSequence(root.**left**);  
 leftList.add(root);  
 leftList.addAll(inOrderSequence(root.**right**));  
 **return** leftList;  
 }  
  
 **public void** inOrderTraversal(){  
 inOrderTraversal(**this**);  
 }  
 **private void** inOrderTraversal(BinaryThreadedTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 inOrderTraversal(root.**hasLeftThread** ? **null** : root.**left**);  
 System.***out***.print(root + **" "**);  
 inOrderTraversal(root.**hasRightThread** ? **null** : root.**right**);  
 }  
  
 **public void** preOrderTraversal(){  
 preOrderTraversal(**this**);  
 }  
 **private void** preOrderTraversal(BinaryThreadedTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 System.***out***.print(root + **" "**);  
 preOrderTraversal(root.**hasLeftThread** ? **null** : root.**left**);  
 preOrderTraversal(root.**hasRightThread** ? **null** : root.**right**);  
 }  
  
 **public void** postOrderTraversal(){  
 postOrderTraversal(**this**);  
 }  
 **private void** postOrderTraversal(BinaryThreadedTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 postOrderTraversal(root.**hasLeftThread** ? **null** : root.**left**);  
 postOrderTraversal(root.**hasRightThread** ? **null** : root.**right**);  
 System.***out***.print(root + **" "**);  
 }  
  
 *//Printing Methods* @Override  
 **public** String toString() {  
 **if**(**this** == **null**)  
 **return "null"**;  
 **return** (**"{ "** + **data** + **" }"**);  
 }  
  
 **public void** print(){  
 print(**this**);  
 }  
 **private void** print(BinaryThreadedTree root){  
 **if**(root == **null**){  
 **return**;  
 }  
  
 System.***out***.print(root.**data** + **" : "**);  
 System.***out***.print(root.**hasLeftThread** ? **"null "** : root.**left** + **" "**);  
 System.***out***.println(root.**hasRightThread** ? **"null "** : root.**right** + **" "**);  
 print(root.**hasLeftThread** ? **null** : root.**left**);  
 print(root.**hasRightThread** ? **null** : root.**right**);  
 }  
  
 *//Getter Methods* **public int** size(){  
 **return size**;  
 }  
}

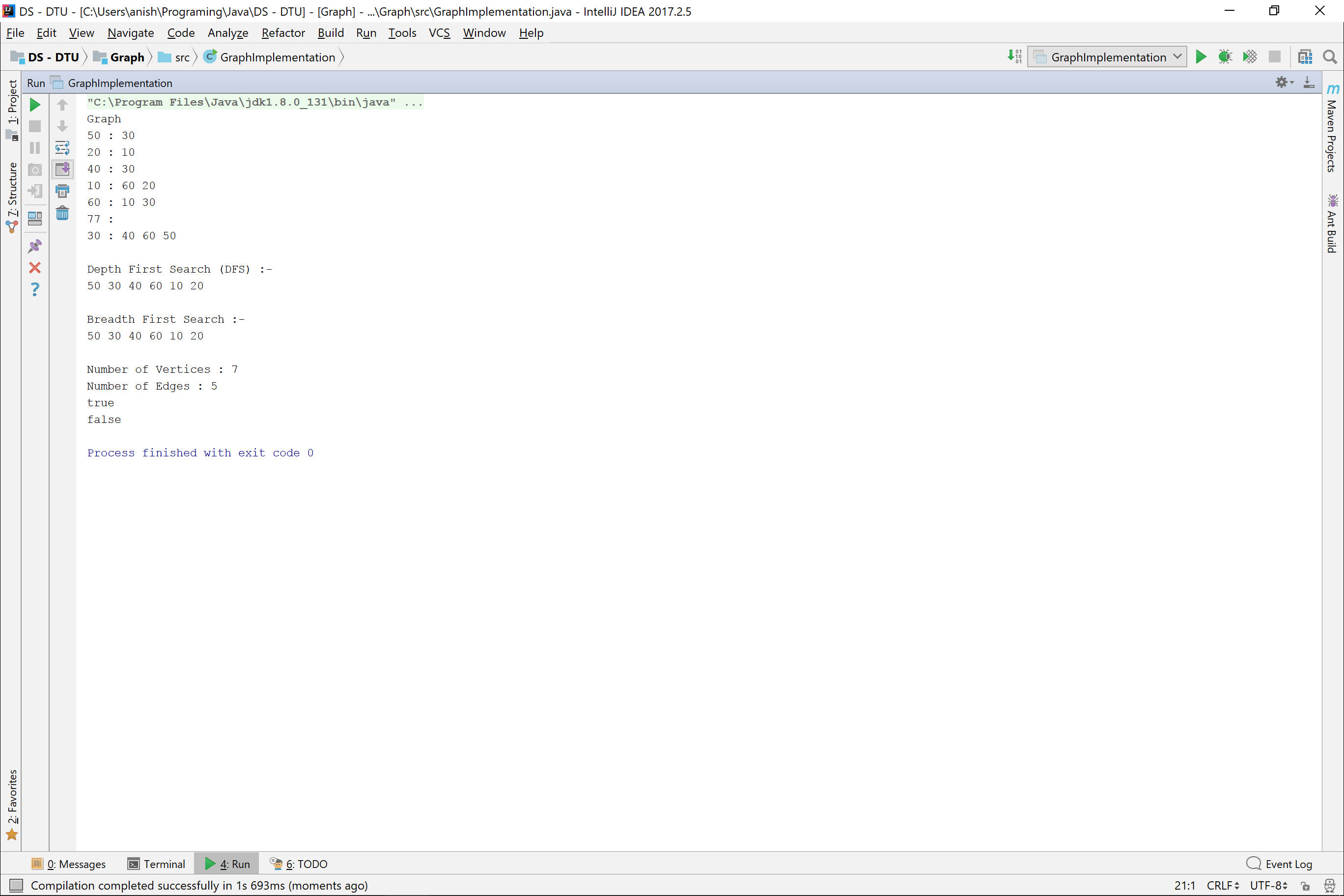


Graph

**import** java.util.\*;  
  
**public class** Graph {  
 **private** HashMap<Integer, Vertex> **vertices**;  
 **private int edges**=0;  
  
 *//Constructors* Graph(){  
 **vertices** = **new** HashMap<>();  
 }  
  
 *//Methods* **public boolean** containsVertex(**int** vertexData){  
 **return vertices**.containsKey(vertexData);  
 }  
  
 **public void** addVertex(**int** vertexData) **throws** AlreadyContainsVertexException{  
 **if**(containsVertex(vertexData))  
 **throw new** AlreadyContainsVertexException();  
  
 Vertex vertex = **new** Vertex(vertexData);  
 **vertices**.put(vertexData, vertex);  
 }  
  
 **public void** createEdge(**int** v1, **int** v2, Integer weight) **throws** VertexDoesNotExistException{  
 **if**(!**vertices**.containsKey(v1) || !**vertices**.containsKey(v2))  
 **throw new** VertexDoesNotExistException(**vertices**.containsKey(v1) ? v2 : v1);  
  
 Edge edge = **new** Edge(**vertices**.get(v1), **vertices**.get(v2), weight);  
 **vertices**.get(v1).addEdge(edge);  
 **vertices**.get(v2).addEdge(edge);  
 **edges**++;  
 }  
 **public void** createEdge(**int** v1, **int** v2) **throws** VertexDoesNotExistException{  
 createEdge(v1, v2, **null**);  
 }  
  
 **public boolean** isConnected(**int** v1, **int** v2) **throws** VertexDoesNotExistException{  
 **if**(!**vertices**.containsKey(v1) || !**vertices**.containsKey(v2))  
 **throw new** VertexDoesNotExistException();  
  
 **return** existsPathBetween(v1, v2);  
 }  
 **private boolean** existsPathBetween(**int** v1, **int** v2){  
 HashSet<Vertex> processedVertices = **new** HashSet<>();  
 **return** searchInForest(**vertices**.get(v1), **vertices**.get(v2), processedVertices);  
 }  
 **private boolean** searchInForest(Vertex main, Vertex search, HashSet<Vertex> processedVertices){  
 **if**(main == search)  
 **return true**;  
  
 **for**(Edge edge : main.EdgeSet()){  
 Vertex otherMain = edge.vertexFrom() == main ? edge.vertexTo() : edge.vertexFrom();  
 **if**(!processedVertices.contains(otherMain)){  
 processedVertices.add(otherMain);  
 **if**(searchInForest(otherMain, search, processedVertices))  
 **return true**;  
 }  
 }  
 **return false**;  
 }  
  
 *//Graph Properties* **public int** edges(){  
 **return edges**;  
 }  
 **public int** vertices(){  
 **return vertices**.size();  
 }  
  
 *//Traversals  
 //Depth First Search* **public void** depthFirstSearch(){  
 **if**(vertices() == 0) **return**;  
  
 HashSet<Vertex> processedVertices = **new** HashSet<>();  
 Vertex start = **new** Vertex();  
 **for**(Map.Entry<Integer, Vertex> entry : **vertices**.entrySet()){  
 start = entry.getValue();  
 **break**;  
 }  
 depthFirstSearch(start, processedVertices);  
 System.***out***.println();  
 }  
 **private void** depthFirstSearch(Vertex vertex, HashSet<Vertex> processedVertices){  
 **if**(vertex == **null**){  
 **return**;  
 }  
 **if**(processedVertices.contains(vertex)){  
 **return**;  
 }  
  
 processedVertices.add(vertex);  
 System.***out***.print(vertex.data() + **" "**);  
  
 **for**(Edge edge : vertex.EdgeSet()){  
 depthFirstSearch(edge.vertexFrom() == vertex ? edge.vertexTo() : edge.vertexFrom() ,  
 processedVertices);  
 }  
 }  
  
 *//Breadth First Search* **public void** breadthFirstSearch(){  
 **if**(vertices() == 0) **return**;  
  
 HashSet<Vertex> processedVertices = **new** HashSet<>();  
 Vertex start = **new** Vertex();  
 **for**(Map.Entry<Integer, Vertex> entry : **vertices**.entrySet()){  
 start = entry.getValue();  
 **break**;  
 }  
  
 breadthFirstSearch(start, processedVertices);  
 System.***out***.println();  
 }  
 **private void** breadthFirstSearch(Vertex vertex, HashSet<Vertex> processedVertices){  
 LinkedList<Vertex> verticesList = **new** LinkedList<>();  
 verticesList.add(vertex);  
  
 **while** (!verticesList.isEmpty()){  
 Vertex node = verticesList.pop();  
 System.***out***.print(node.data() + **" "**);  
 processedVertices.add(node);  
  
 addAdjacent(node, verticesList, processedVertices);  
 }  
 }  
 **private void** addAdjacent(Vertex vertex, LinkedList<Vertex> linkedList, HashSet<Vertex> processedVertices){  
 **for**(Edge edge : vertex.EdgeSet()){  
 Vertex otherVertex = edge.vertexFrom() == vertex ? edge.vertexTo() : edge.vertexFrom();  
 **if**(!processedVertices.contains(otherVertex))  
 linkedList.add(otherVertex);  
 }  
 }  
  
 *//Printing* **public void** print(){  
 **for**(Map.Entry<Integer, Vertex> entry : **vertices**.entrySet()){  
 System.***out***.print(entry.getKey() + **" : "**);  
 print(entry.getValue().EdgeSet(), entry.getValue());  
 }  
 }  
 **private void** print(HashSet<Edge> edgeSet, Vertex from){  
 **for**(Edge edge : edgeSet){  
 System.***out***.print( (edge.vertexFrom() == from ? edge.vertexTo().data() : edge.vertexFrom().data()) + **" "** );  
 }  
 System.***out***.println();  
 }  
}  
  
  
**class** AlreadyContainsVertexException **extends** Exception{  
 AlreadyContainsVertexException(){  
 System.***out***.println(**"AlreadyContainsVertexException"**);  
 }  
}  
  
**class** VertexDoesNotExistException **extends** Exception{  
 VertexDoesNotExistException(){}  
 VertexDoesNotExistException(**int** vertexData){  
 System.***out***.println(**"VertexDoesNotExistException : "** + vertexData);  
 }  
}

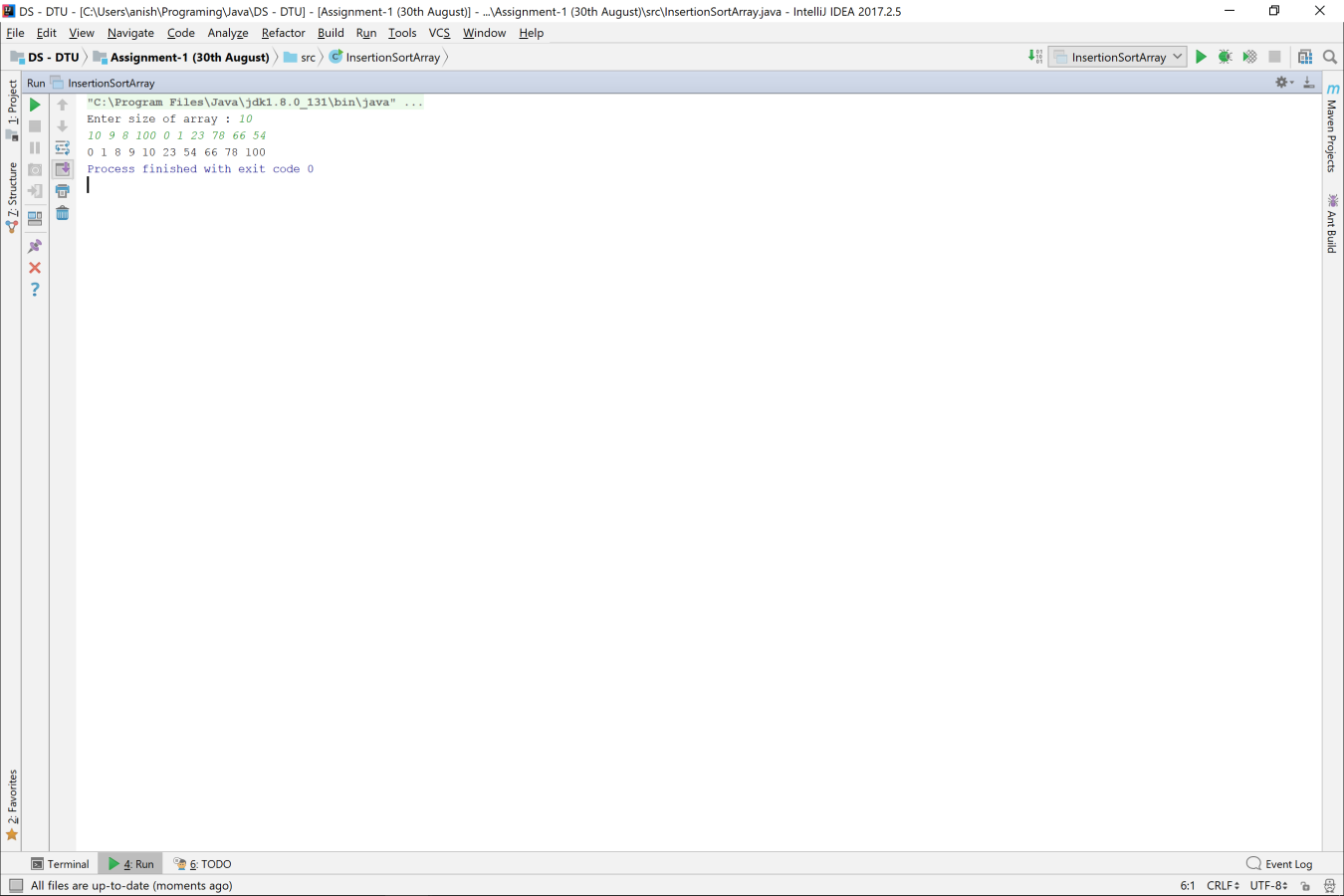
**import** java.util.HashSet;  
  
**public class** Vertex {  
 **private** Integer **data**;  
 **private** HashSet<Edge> **edgeSet**;  
  
 *//Constructors* Vertex(){  
 **edgeSet** = **new** HashSet<>();  
 }  
 Vertex(**int** data){  
 **this**.**data** = data;  
 **edgeSet** = **new** HashSet<>();  
 }  
  
 *//Methods* **public void** addEdge(Edge edge){  
 **edgeSet**.add(edge);  
 }  
 **public int** degree(){  
 **return edgeSet**.size();  
 }  
  
 *//Getter Methods* **public int** data(){  
 **return data**;  
 }  
 **public** HashSet<Edge> EdgeSet() {  
 **return edgeSet**;  
 }  
}  
  
  
**class** Edge{  
 **private** Vertex **vertex1**, **vertex2**;  
 **private** Integer **weight**;  
  
 *//Constructors* Edge(){}  
 Edge(**int** weight){  
 **this**.**weight** = weight;  
 }  
 Edge(Vertex vertex1, Vertex vertex2){  
 **this**.**vertex1** = vertex1;  
 **this**.**vertex2** = vertex2;  
 }  
 Edge(Vertex vertex1, Vertex vertex2, Integer weight){  
 **this**.**vertex1** = vertex1;  
 **this**.**vertex2** = vertex2;  
 **this**.**weight** = weight;  
 }  
  
 *//Getter Methods* **public int** weight(){  
 **return weight**;  
 }  
 **public** Vertex vertexFrom() {  
 **return vertex1**;  
 }  
  
 **public** Vertex vertexTo() {  
 **return vertex2**;  
 }  
}

**public class** GraphImplementation {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) **throws** AlreadyContainsVertexException, VertexDoesNotExistException {  
 Graph graph = **new** Graph();  
  
 graph.addVertex(10); graph.addVertex(20);  
 graph.addVertex(77);  
 graph.createEdge(10, 20, 100);  
  
 graph.addVertex(30); graph.addVertex(40);  
 graph.addVertex(50); graph.addVertex(60);  
 graph.createEdge(30, 40, 12);  
 graph.createEdge(30, 50, 33);  
 graph.createEdge(30, 60, 100);  
 graph.createEdge(10, 60, 47);  
 System.***out***.println(**"Graph "**);  
 graph.print();  
  
 System.***out***.println(**"\nDepth First Search (DFS) :- "**);  
 graph.depthFirstSearch();  
  
 System.***out***.println(**"\nBreadth First Search :- "**);  
 graph.breadthFirstSearch();  
  
 System.***out***.println(**"\nNumber of Vertices : "** + graph.vertices());  
 System.***out***.println(**"Number of Edges : "** + graph.edges());  
  
 System.***out***.println(graph.isConnected(10, 30));  
 System.***out***.println(graph.isConnected(10, 77));  
 }  
}



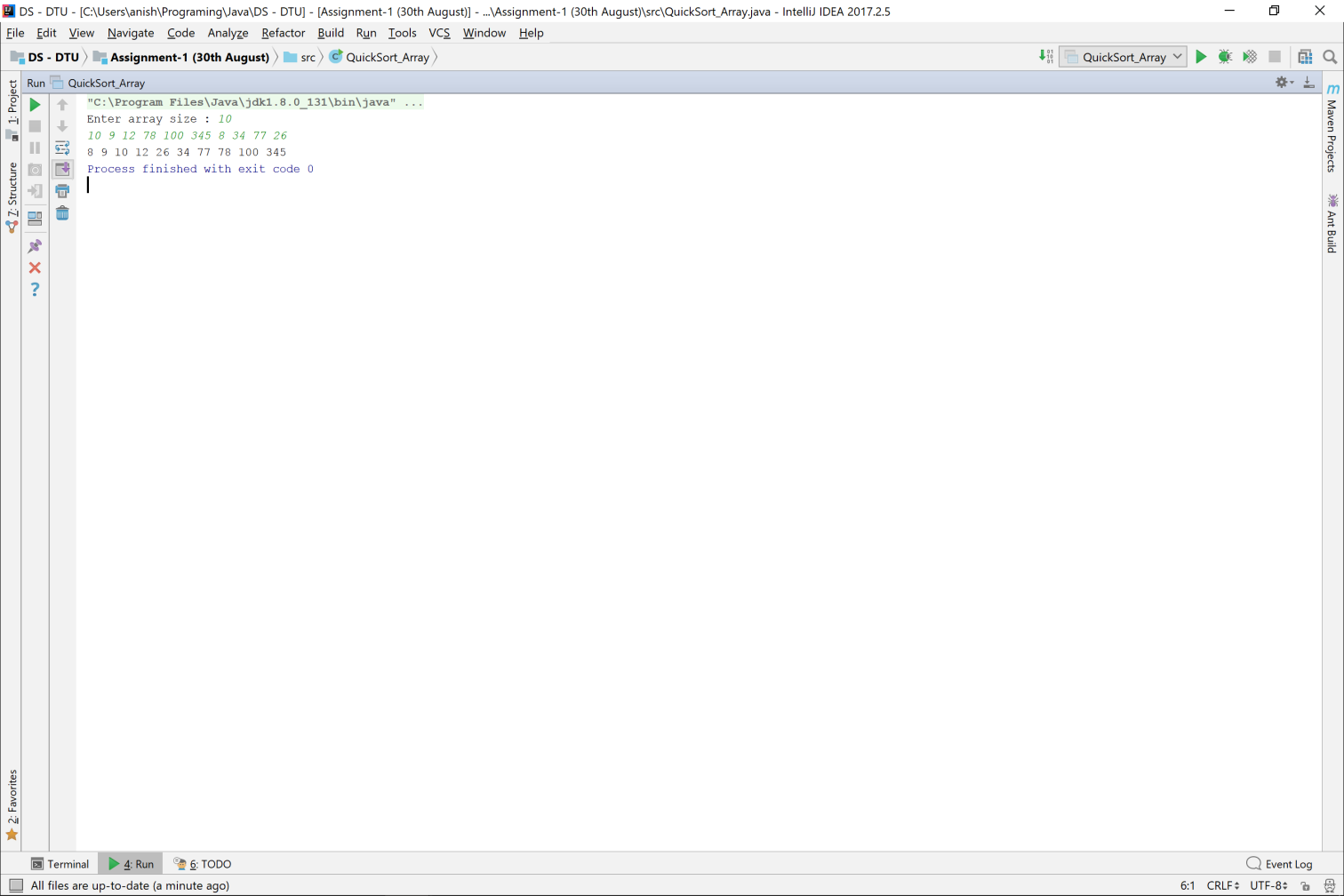
Insertion Sort

**public class** InsertionSortArray {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Enter size of array : "**);  
 **int** size = *in*.nextInt();  
 **int** arr[] = **new int**[size];  
 *input*(arr);  
  
 *insertionSort*(arr);  
 *print*(arr);  
 }  
  
 **private static void** input(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 arr[index] = *in*.nextInt();  
 }  
 }  
 **private static void** print(**int** arr[]){  
 **for**(**int** val : arr){  
 System.***out***.print(val + **" "**);  
 }  
 }  
  
 **private static void** insertionSort(**int** arr[]){  
 **for**(**int** index=1 ; index<arr.**length** ; index++){  
 *insertAt*(arr, index);  
 }  
 }  
 **private static void** insertAt(**int** arr[], **int** index){  
 **for**(**int** startIndex=0 ; startIndex <index ; startIndex++){  
 **if**(arr[index] < arr[startIndex]){  
 **int** val = arr[index];  
 *cycle*(arr, startIndex, index);  
 arr[startIndex] = val;  
 **break**;  
 }  
 }  
 }  
 **private static void** cycle(**int** arr[], **int** startIndex, **int** endIndex){  
 **for**(**int** index=endIndex ; index > startIndex ; index--){  
 arr[index] = arr[index-1];  
 }  
 }  
}



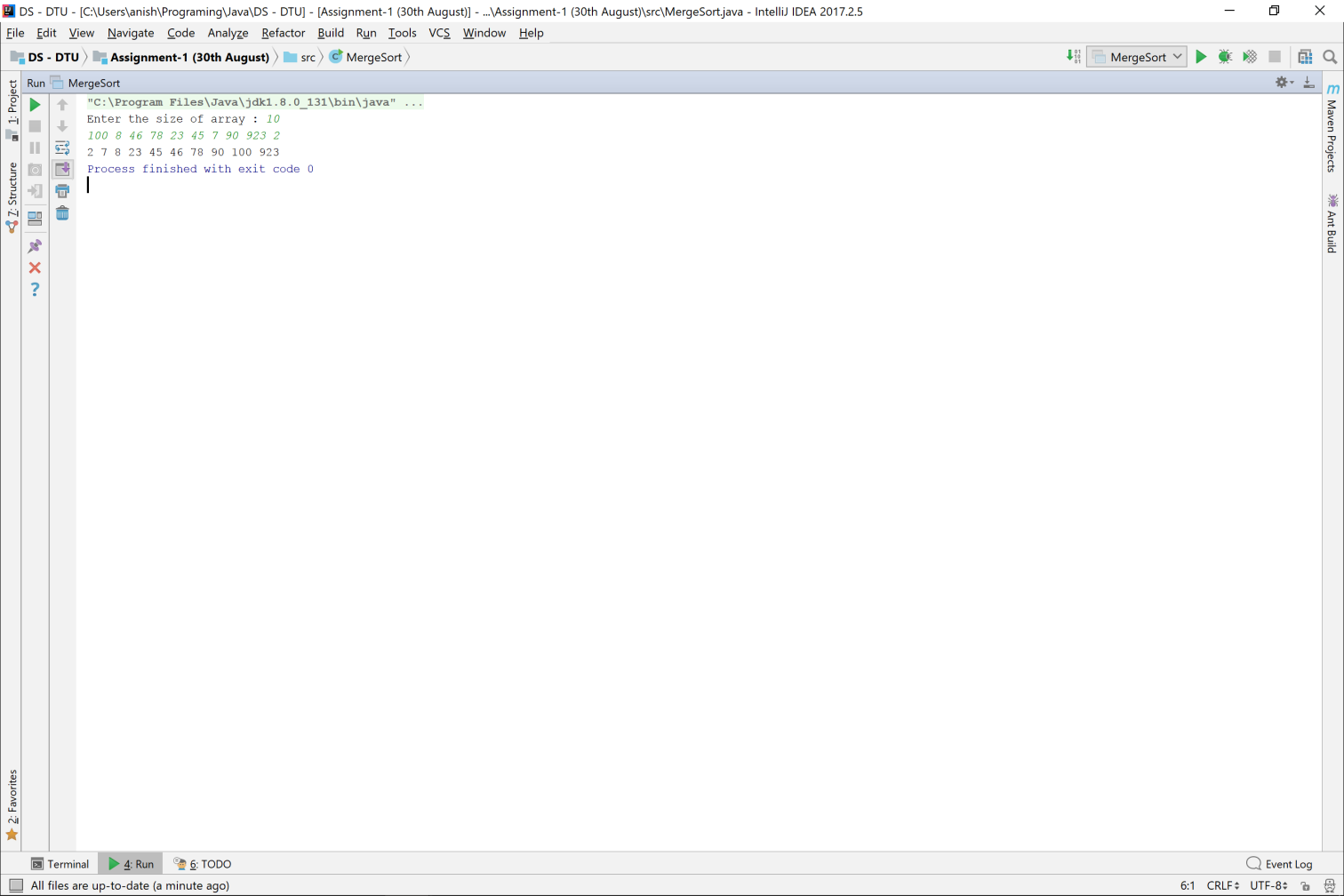
Quick Sort

**public class** QuickSort\_Array {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Enter array size : "**);  
 **int** size = *in*.nextInt();  
 **int** arr[] = **new int**[size];  
 *input*(arr);  
  
 *quickSort*(arr);  
 *print*(arr);  
 }  
  
 **private static void** input(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 arr[index] = *in*.nextInt();  
 }  
 }  
 **private static void** print(**int** arr[]){  
 **for**(**int** val : arr){  
 System.***out***.print(val + **" "**);  
 }  
 }  
  
 **private static void** quickSort(**int** arr[]){  
 *quickSort*(arr, 0, arr.**length**-1);  
 }  
 **private static void** quickSort(**int** arr[], **int** startIndex, **int** endIndex){  
 **if**(startIndex < endIndex){  
 **int** pivot = *partition*(arr, startIndex, endIndex);  
 *quickSort*(arr, startIndex, pivot-1);  
 *quickSort*(arr, pivot+1, endIndex);  
 }  
 }  
  
 **private static int** partition(**int** arr[], **int** start, **int** end){  
 **for**(**int** index=end ; index > start ; ){  
 **if**(arr[index] < arr[start]){  
 *swap*(arr, start+1, index);  
 *swap*(arr, start, start+1);  
 start++;  
 } **else** index--;  
 }  
 **return** start;  
 }  
  
 **private static void** swap(**int** arr[], **int** index1, **int** index2){  
 **int** val = arr[index1];  
 arr[index1] = arr[index2];  
 arr[index2] = val;  
 }  
}



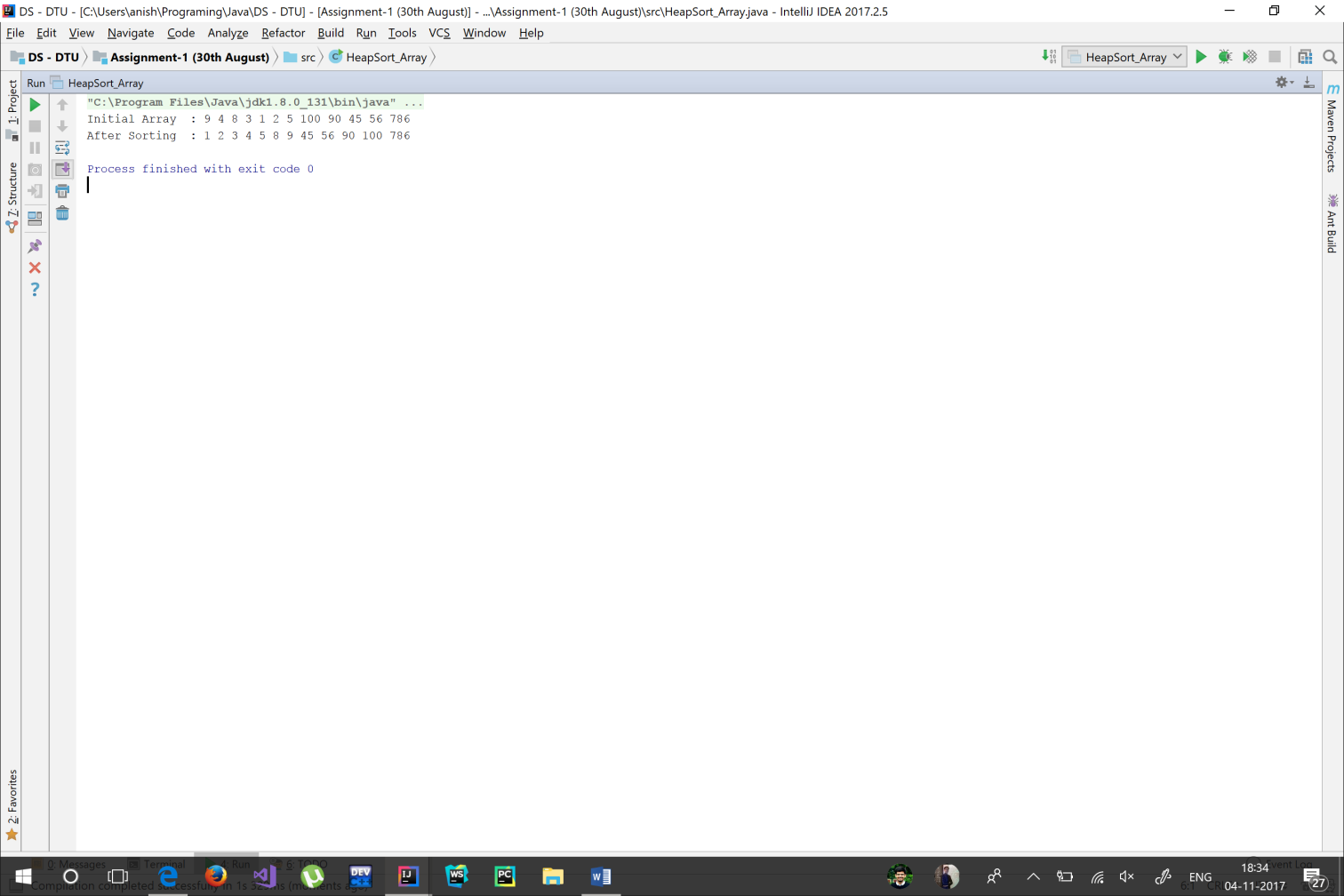
Merge Sort

**public class** MergeSort {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Enter the size of array : "**);  
 **int** size = *in*.nextInt();  
 **int** arr[] = **new int**[size];  
 *input*(arr);  
  
 arr = *mergeSort*(arr);  
 *print*(arr);  
 }  
  
 **private static void** input(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 arr[index] = *in*.nextInt();  
 }  
 }  
 **private static void** print(**int** arr[]){  
 **for**(**int** val : arr){  
 System.***out***.print(val + **" "**);  
 }  
 }  
  
 **private static int**[] mergeSort(**int** arr[]){  
 **if**(arr.**length** <= 1){  
 **return** arr;  
 }  
  
 **int** smallArray1[] = **new int**[arr.**length**/2];  
 **int** smallArray2[] = **new int**[arr.**length** - smallArray1.**length**];  
 **int** index=0;  
 **while** (index < smallArray1.**length**){  
 smallArray1[index] = arr[index++];  
 }  
 **while** (index < arr.**length**){  
 smallArray2[index - smallArray1.**length**] = arr[index++];  
 }  
  
 smallArray1 = *mergeSort*(smallArray1);  
 smallArray2 = *mergeSort*(smallArray2);  
  
 **return** *merge*(smallArray1, smallArray2);  
 }  
  
 **private static int**[] merge(**int** arr1[], **int** arr2[]){  
 **int** ans[] = **new int**[arr1.**length** + arr2.**length**], i=0, j=0, k=0;  
  
 **while** ( i<arr1.**length** && j<arr2.**length** ){  
 **if**(arr1[i] <= arr2[j]){  
 ans[k++] = arr1[i++];  
 }  
 **else** ans[k++] = arr2[j++];  
 }  
  
 **while** (i < arr1.**length**){  
 ans[k++] = arr1[i++];  
 }  
 **while** (j < arr2.**length**){  
 ans[k++] = arr2[j++];  
 }  
  
 **return** ans;  
 }  
}



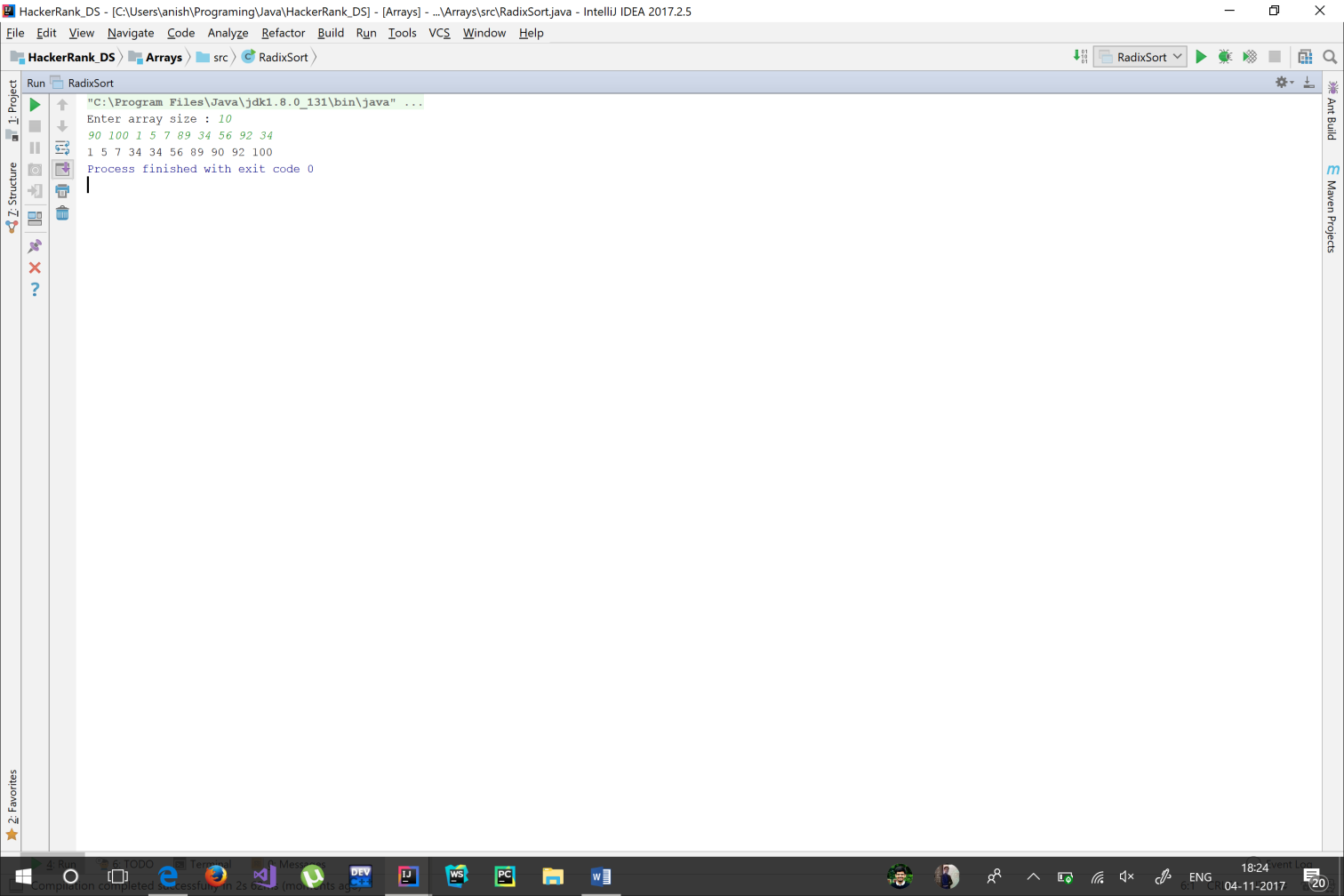
Heap Sort

**public class** HeapSort\_Array {  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 *//Heap Sort* **public static void** main(String[] args) {  
 *//Initializing array* **int** arr[] = {9, 4, 8, 3, 1, 2, 5, 100, 90, 45 , 56 , 786};  
 System.***out***.print(**"Initial Array : "**);  
 *printArray*(arr);  
 arr = *heapsort*(arr);  
 System.***out***.print(**"After Sorting : "**) ;  
 *printArray*(arr);  
 }  
 **public static void** printArray(**int**[] arr) {  
 **for**(**int** i = 0; i < arr.**length**; i++) {  
 System.***out***.print(arr[i] + **" "**);  
 }  
  
 System.***out***.println();  
 }  
 *//Sorting in non decreasing order* **public static int**[] heapsort(**int** arr[]) {  
 **int** N = arr.**length**;  
 *//creating a heap* MaxHeap heap = *createHeap*(arr, N);  
  
 *//Repeating the below steps till the size of the heap is 1.* **while**(heap.**len** > 1) {  
 *//Replacing largest element with the last item of the heap  
 swap*(heap, 0, heap.**len** - 1);  
 heap.**len**--;*//Reducing the heap size by 1  
 heapify*(heap, 0);  
 }  
  
 **return** heap.**arr**;  
 }  
 **public static** MaxHeap createHeap(**int** arr[], **int** N) {  
 MaxHeap maxheap = **new** MaxHeap(N, arr);  
 **int** i = (maxheap.**len** - 2) / 2;  
  
 **while**(i >= 0) {  
 maxheap = *heapify*(maxheap, i);  
 i--;  
 }  
  
 **return** maxheap;  
 }  
 **public static** MaxHeap heapify(MaxHeap maxheap, **int** N) {  
 **int** largest = N;  
 **int** left = 2 \* N + 1; *//index of left child* **int** right = 2 \* N + 2; *//index of right child* **if**(left < maxheap.**len** && maxheap.**arr**[left] > maxheap.**arr**[largest]) {  
 largest = left;  
 }  
  
 **if**(right < maxheap.**len** && maxheap.**arr**[right] > maxheap.**arr**[largest]) {  
 largest = right;  
 }  
  
 **if**(largest != N) {  
 *swap*(maxheap, largest, N);  
 *heapify*(maxheap, largest);  
 }  
  
 **return** maxheap;  
 }  
 **public static void** swap(MaxHeap maxheap, **int** i, **int** j) {  
 **int** temp;  
 temp = maxheap.**arr**[i];  
 maxheap.**arr**[i] = maxheap.**arr**[j];  
 maxheap.**arr**[j] = temp;  
 }  
 **static class** MaxHeap {  
 **int len**;  
 **int arr**[];  
 MaxHeap(**int** l, **int** a[]) {  
 **len** = l;  
 **arr** = a;  
 }  
 }  
  
}



Radix Sort

**public class** RadixSort {  
  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 **int** size = *in*.nextInt();  
 **int** arr[] = **new int**[size];  
 *input*(arr);  
 *radixSort*(arr);  
 *print*(arr);  
 }  
  
 **private static void** input(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 arr[index] = *in*.nextInt();  
 }  
 }  
  
 **private static void** radixSort(**int** arr[]){  
 **int** maxElement = *max*(arr);  
 **int** length = *intLen*(maxElement);  
  
 **for**(**int** i=0 ; i<length ; i++){  
  
 *//Create 10 Buckets and Initialize* Bucket[] buckets = **new** Bucket[10];  
 **for**(**int** index=0 ; index<buckets.**length** ; index++){  
 buckets[index] = **new** Bucket();  
 }  
  
 *//Add to Buckets  
 addToBuckets*(arr, buckets, i);  
  
 *//ReInitialize - Array  
 initialize*(arr, buckets);  
 System.***out***.println();  
 *print*(arr);  
 System.***out***.println();  
 }  
 }  
  
 **private static void** addToBuckets(**int** arr[], Bucket[] buckets, **int** position){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 **int** element = *digitAt*(arr[index], position);  
 System.***out***.println(element);  
 buckets[element].add(arr[index]);  
 }  
 }  
  
 **private static int** digitAt(**int** number, **int** position){  
 **return** (**int**)(number/Math.*pow*(10, position) + 10) % 10;  
 }  
  
 **private static void** initialize(**int** arr[], Bucket[] buckets){  
 **int** arrayIndex = 0;  
 **for**(**int** bucketIndex=0 ; bucketIndex<10 ; bucketIndex++){  
 arrayIndex = *initialize*(arr, buckets[bucketIndex], arrayIndex);  
 }  
 }  
  
 **private static int** initialize(**int** arr[], Bucket bucket, **int** index){  
 **for**(**int** arrayIndex=index ; arrayIndex<bucket.size() + index ; arrayIndex++){  
 arr[arrayIndex] = bucket.get(arrayIndex - index);  
 }  
 **return** index + bucket.size();  
 }  
  
 **private static int** max(**int** arr[]){  
 **int** max=arr[0];  
 **for**(**int** index=1 ; index<arr.**length** ; index++){  
 **if**(arr[index] > max)  
 max = arr[index];  
 }  
 **return** max;  
 }  
  
 **private static int** intLen(**int** number){  
 **int** length;  
 **for**(length=0 ; number != 0 ; number /= 10, length++);  
 **return** length;  
 }  
  
 **private static void** print(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 System.***out***.print(arr[index] + **" "**);  
 }  
 }  
}



Bucket Sort

**import** java.util.ArrayList;  
**import** java.util.Scanner;  
  
**public class** BucketSort\_Algo {  
  
 **private static** Scanner *in* = **new** Scanner(System.***in***);  
  
 **public static void** main(String[] args) {  
 System.***out***.print(**"Size of Array : "**);  
 **int** size = *in*.nextInt();  
 **int** arr[] = **new int**[size];  
 Bucket buckets[] = **new** Bucket[size/5 + 1];  
  
 **for**(**int** index=0 ; index<size ; index++){  
 arr[index] = *in*.nextInt();  
 }  
 arr = *bucketSort*(arr, buckets);  
 *print*(arr);  
 }  
  
 **private static int**[] bucketSort(**int** arr[], Bucket buckets[]){  
 **for**(**int** index=0 ; index<buckets.**length** ; index++){  
 buckets[index] = **new** Bucket();  
 }  
 *addToBucket*(arr, buckets);  
 *sortBuckets*(buckets);  
 **return** *sortArray*(arr.**length**, buckets);  
 }  
  
 **private static void** sortBuckets(Bucket[] buckets){  
 **for**(**int** index=0 ; index<buckets.**length** ; index++){  
 buckets[index].sort();  
 }  
 }  
  
 **private static int**[] sortArray(**int** size, Bucket[] buckets){  
 **int** ans[] = **new int**[size];  
 **for**(**int** index=0, arrayIndex=0 ; index<buckets.**length** ; index++){  
 arrayIndex = *addTo*(ans, buckets[index], arrayIndex);  
 }  
 **return** ans;  
 }  
  
 **private static int** addTo(**int** arr[], Bucket bucket, **int** index){  
 **return** bucket.addTo(arr, index);  
 }  
  
 **private static void** addToBucket(**int** arr[], Bucket[] buckets){  
 **int** minVal = *min*(arr);  
 **int** maxVal = *max*(arr);  
 **final int** bucketSize = buckets.**length**;  
 **final double** range = maxVal - minVal + 1;  
  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 **int** bucketIndex = (**int**)(bucketSize \* (**double**)(arr[index] - minVal) / (range));  
 buckets[bucketIndex].add(arr[index]);  
 }  
 }  
  
 **private static int** max(**int** arr[]){  
 **int** maxVal, index;  
 **for**(index=0, maxVal=arr[0] ; index<arr.**length** ; index++){  
 **if**(arr[index] > maxVal)  
 maxVal = arr[index];  
 }  
 **return** maxVal;  
 }  
  
 **private static int** min(**int** arr[]){  
 **int** minVal, index;  
 **for**(index=0, minVal = arr[0] ; index<arr.**length** ; index++){  
 **if**(arr[index] < minVal)  
 minVal = arr[index];  
 }  
 **return** minVal;  
 }  
  
 **private static void** print(**int** arr[]){  
 **for**(**int** index=0 ; index<arr.**length** ; index++){  
 System.***out***.print(arr[index] + **" "**);  
 }  
 }  
}  
  
**class** Bucket{  
 **private** ArrayList<Integer> **list**;  
  
 Bucket(){  
 **list** = **new** ArrayList<>();  
 }  
 Bucket(**int** size){  
 **list** = **new** ArrayList<>(size);  
 }  
  
 **public void** add(Integer data){  
 **list**.add(data);  
 }  
  
 **public int** size(){  
 **return list**.size();  
 }  
  
 **public int** get(**int** index){  
 **return list**.get(index);  
 }  
  
 **public void** sort(){  
 **list** = mergeSort(**list**);  
 }  
  
 **private** ArrayList<Integer> mergeSort(ArrayList<Integer> list){  
 **if**(list.size() <= 1)  
 **return** list;  
  
 ArrayList<Integer> smallList1 = **new** ArrayList<>();  
 ArrayList<Integer> smallList2 = **new** ArrayList<>();  
 **int** index=0;  
  
 **for**(; index<list.size()/2 ; index++){  
 smallList1.add(list.get(index));  
 }  
 **for**( ; index<list.size() ; index++){  
 smallList2.add(list.get(index));  
 }  
  
 ArrayList<Integer> ans1 = mergeSort(smallList1);  
 ArrayList<Integer> ans2 = mergeSort(smallList2);  
  
 **return** merge(ans1, ans2);  
 }  
  
 **private** ArrayList<Integer> merge(ArrayList<Integer> list1, ArrayList<Integer> list2){  
 **int** i, j, t;  
 ArrayList<Integer> ans = **new** ArrayList<>();  
  
 **for**(i=0, j=0 ; i<=list1.size() && j<=list2.size() ; ){  
 **if**(i == list1.size()){  
 **for**(t=0; t<list2.size() - j ; ans.add(list2.get(t+j)), t++);  
 **break**;  
 } **else if**(j == list2.size()){  
 **for**(t=0 ; t< list1.size() - i ; ans.add(list1.get(t+i)), t++);  
 **break**;  
 }  
  
 **if**((**int**)list1.get(i) <= (**int**)list2.get(j)){  
 ans.add(list1.get(i));  
 i++;  
 } **else** {  
 ans.add(list2.get(j));  
 j++;  
 }  
 }  
 **return** ans;  
 }  
  
 **public int** addTo(**int** arr[], **int** index){  
 **for**(**int** count=0 ; count<**list**.size() ; index++, count++){  
 arr[index] = (**int**)**list**.get(count);  
 }  
 **return** index;  
 }  
}

