**C-DAC Mumbai Date 29/09/2024**

**Subject: Algorithm and Data Structure**

**Assignment 3**

**Solve the assignment with following thing to be added in each question.**

-Program

-Flow chart

-Explanation

-Output

-Time and Space complexity

Submission Date: 01/10/2024

1. **Implement a Stack using an array.**

-Program

import java.util.Scanner;

public class StackUsingArray {

private int maxSize;

private int top;

private int[] stackArray;

public StackUsingArray(int size) {

maxSize = size;

stackArray = new int[maxSize];

top = -1;

}

public void push(int value) {

if (top >= maxSize - 1) {

System.out.println("Stack is full. Cannot push " + value + " onto the stack.");

} else {

stackArray[++top] = value;

}

}

public int pop() {

if (top < 0) {

System.out.println("Stack is empty. Cannot pop from the stack.");

return -1;

} else {

return stackArray[top--];

}

}

public void displayStack() {

if (top >= 0) {

System.out.print("Stack = [");

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

System.out.print(stackArray[i]);

if (i < top) {

System.out.print(", ");

}

}

System.out.println("]");

} else {

System.out.println("Stack is empty.");

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the size of the stack: ");

int size = scanner.nextInt();

StackUsingArray stack = new StackUsingArray(size);

while (true) {

System.out.println("1. Push an element onto the stack");

System.out.println("2. Pop an element from the stack");

System.out.println("3. Display the stack");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter the element to push: ");

int element = scanner.nextInt();

stack.push(element);

break;

case 2:

int poppedElement = stack.pop();

System.out.println("Popped element = " + poppedElement);

break;

case 3:

stack.displayStack();

break;

case 4:

System.exit(0);

break;

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

}

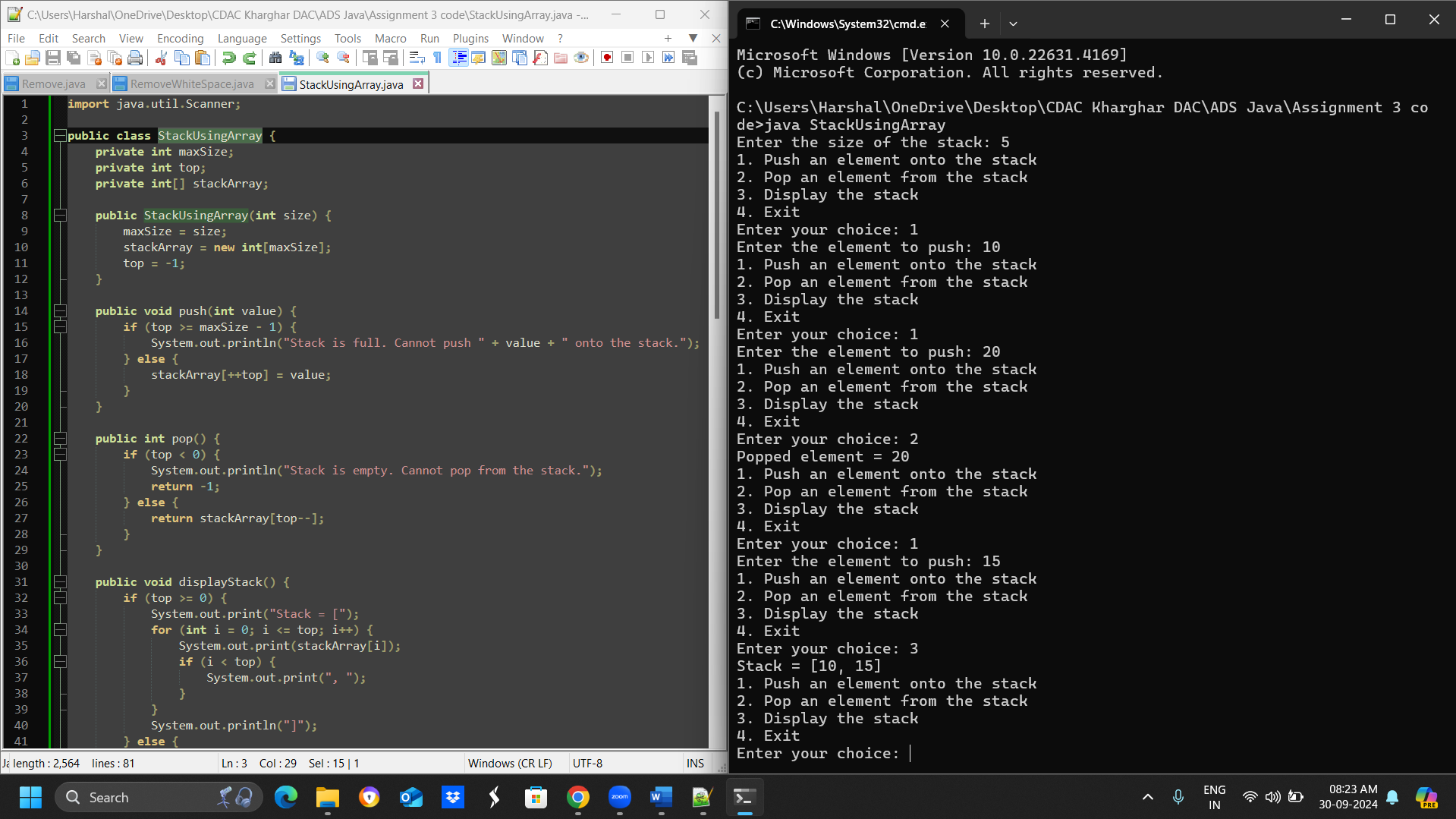
-Flow chart

1. **Start Program**: The program starts, and the user is prompted to enter the size of the stack.
2. **Enter Size of Stack**: The user enters the size of the stack, which is used to initialize the stack.
3. **Initialize Stack**: The stack is initialized by creating an array of the specified size and setting the top to -1.
4. **Display Menu**: The program displays a menu with options to push, pop, display, or exit.
5. **Get User Choice**: The user selects an option from the menu.
6. **Choice 1: Push**: If the user chooses to push, they are prompted to enter an element, which is then pushed onto the stack.
7. **Choice 2: Pop**: If the user chooses to pop, the top element is popped from the stack and displayed.
8. **Choice 3: Display**: If the user chooses to display, the entire stack is displayed.
9. **Choice 4: Exit**: If the user chooses to exit, the program ends.
10. **Invalid Choice**: If the user enters an invalid choice, an error message is displayed, and the menu is repeated.
11. **Repeat Menu**: The menu is repeated, allowing the user to make another selection.

-Explanation

1. Run the program and enter the size of the stack when prompted.
2. Choose an option from the menu:
   * To push an element onto the stack, enter **1** and then enter the element to push.
   * To pop an element from the stack, enter **2**.
   * To display the stack, enter **3**.
   * To exit the program, enter **4**.
3. Repeat step 2 until you're done.

-Output



-Time and Space complexity

**Time Complexity: O(n)**

**Space Complexity: O(n)**

* **Test Case 1**:  
  Input: Push 5, 3, 7, Pop  
  Output: Stack = [5, 3], Popped element = 7
* **Test Case 2**:  
  Input: Push 10, Push 20, Pop, Push 15  
  Output: Stack = [10, 15], Popped element = 20

1. **Check for balanced parentheses using a stack.**

-Program

import java.util.Stack;

public class BalancedParentheses {

public static String checkBalanced(String input) {

Stack<Character> stack = new Stack<>();

for (char c : input.toCharArray()) {

if (c == '(' || c == '{' || c == '[') {

stack.push(c);

} else if (c == ')' || c == '}' || c == ']') {

if (stack.isEmpty()) {

return "Not Balanced";

}

char top = stack.pop();

if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {

return "Not Balanced";

}

}

}

return stack.isEmpty() ? "Balanced" : "Not Balanced";

}

public static void main(String[] args) {

System.out.println(checkBalanced("({[()]})")); // Output: Balanced

System.out.println(checkBalanced("([)]")); // Output: Not Balanced

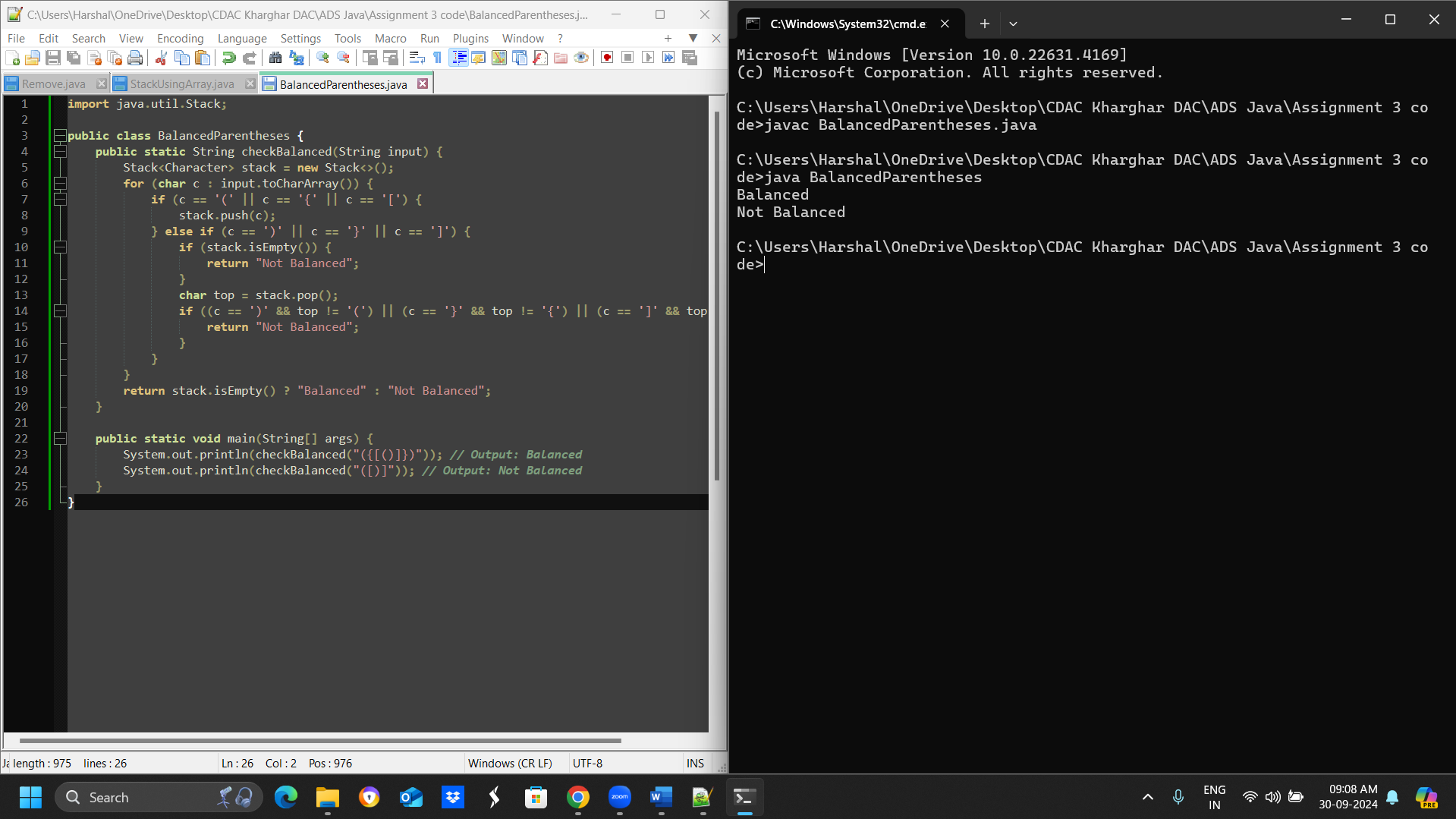
}

}

-Flow chart

-Explanation

-Output



-Time and Space complexity

The time complexity of the **checkBalanced** program is O(n)

The space complexity of the **checkBalanced** program is O(n)

* **Test Case 1**:  
  Input: "({[()]})"  
  Output: Balanced
* **Test Case 2**:  
  Input: "([)]"  
  Output: Not Balanced

1. **Reverse a string using a stack.**

-Program

import java.util.Stack;

public class StringReversal {

public static String reverseString(String s) {

Stack<Character> stack = new Stack<>();

for (char c : s.toCharArray()) {

stack.push(c);

}

StringBuilder reversedS = new StringBuilder();

while (!stack.isEmpty()) {

reversedS.append(stack.pop());

}

return reversedS.toString();

}

public static void main(String[] args) {

System.out.println(reverseString("hello")); // Output: "olleh"

System.out.println(reverseString("world")); // Output: "dlrow"

}

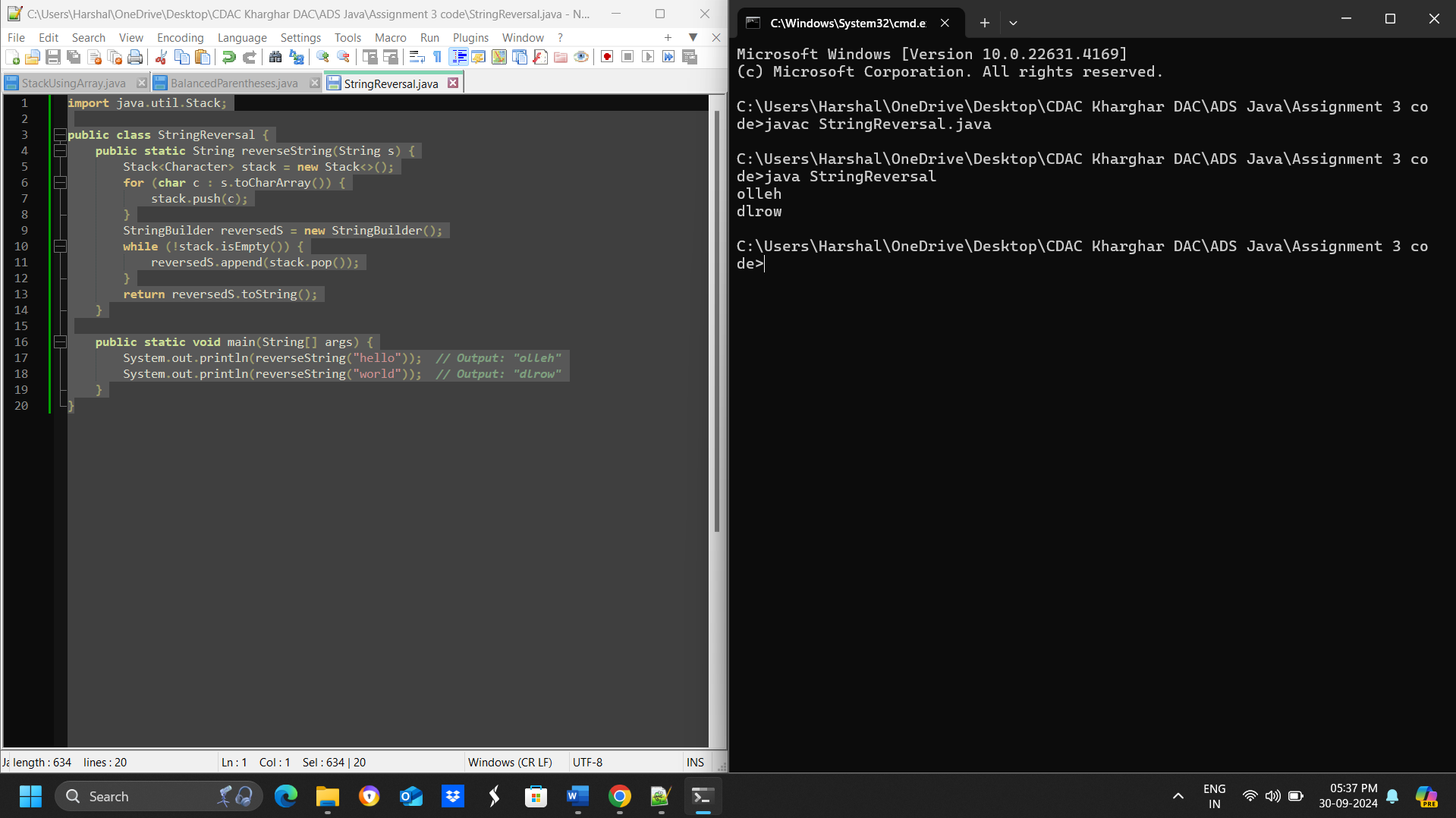
}

-Flow chart

**Explanation:**

* We create an empty stack **[]** to store the characters of the input string.
* We iterate through each character **char** in the input string **s** and push it onto the stack using **stack.append(char)**.
* We create an empty string **reversed\_s** to store the reversed string.
* We pop each character from the stack using **stack.pop()** and append it to the **reversed\_s** string.
* Finally, we return the reversed string **reversed\_s**.

-Output



-Time and Space complexity

**Time Complexity: O(n)**

**space Complexity: O(n)**

* **Test Case 1**:  
  Input: "hello"  
  Output: "olleh"
* **Test Case 2**:  
  Input: "world"  
  Output: "dlrow"

1. **Evaluate a postfix expression using a stack.**

-Program

import java.util.Stack;

public class PostfixEvaluator {

public static int evaluatePostfix(String expression) {

Stack<Integer> stack = new Stack<>();

String[] tokens = expression.split(" ");

for (String token : tokens) {

if (token.matches("\\d+")) {

stack.push(Integer.parseInt(token));

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

int result = calculate(operand1, operand2, token);

stack.push(result);

}

}

return stack.pop();

}

private static int calculate(int operand1, int operand2, String operator) {

switch (operator) {

case "+":

return operand1 + operand2;

case "-":

return operand1 - operand2;

case "\*":

return operand1 \* operand2;

case "/":

return operand1 / operand2;

default:

throw new UnsupportedOperationException("Unsupported operator");

}

}

public static void main(String[] args) {

System.out.println(evaluatePostfix("5 3 + 2 \*")); // Output: 16

System.out.println(evaluatePostfix("4 5 \* 6 /")); // Output: 3

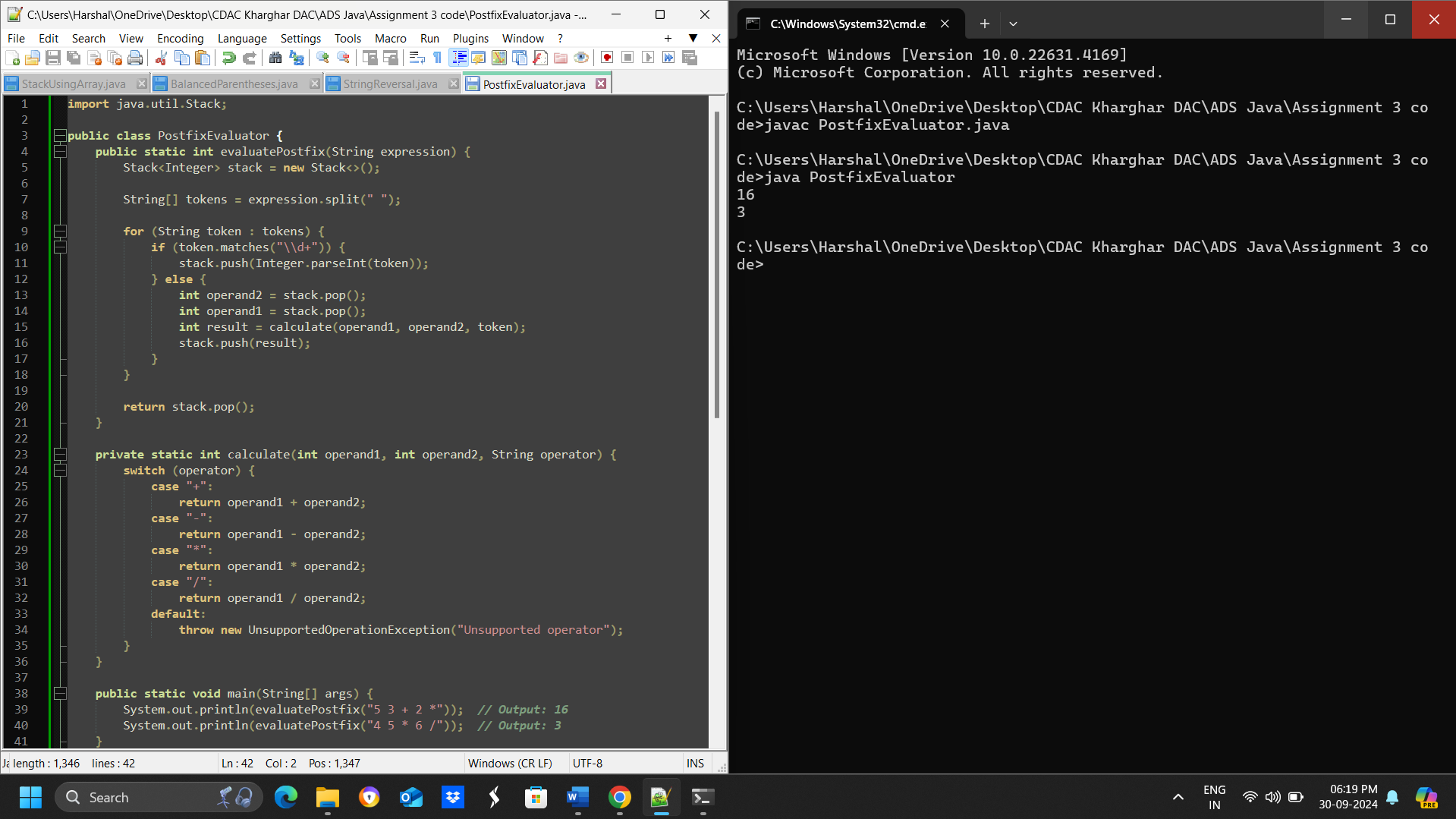
}

}

-Flow chart

-Explanation

-Output



-Time and Space complexity

* **Test Case 1**:  
  Input: "5 3 + 2 \*"  
  Output: 16
* **Test Case 2**:  
  Input: "4 5 \* 6 /"  
  Output: 3

1. **Convert an infix expression to postfix using a stack.**

**import java.util.Stack;**

**public class InfixToPostfix {**

**// Define operator precedence**

**private static final int PRECEDENCE\_ADD\_SUB = 1;**

**private static final int PRECEDENCE\_MUL\_DIV = 2;**

**// Function to get operator precedence**

**private static int getPrecedence(char operator) {**

**switch (operator) {**

**case '+':**

**case '-':**

**return PRECEDENCE\_ADD\_SUB;**

**case '\*':**

**case '/':**

**return PRECEDENCE\_MUL\_DIV;**

**default:**

**return -1;**

**}**

**}**

**// Function to convert infix to postfix**

**public static String infixToPostfix(String infix) {**

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

**StringBuilder postfix = new StringBuilder();**

**for (char c : infix.toCharArray()) {**

**if (Character.isLetterOrDigit(c)) {**

**postfix.append(c).append(" ");**

**} else if (c == '(') {**

**stack.push(c);**

**} else if (c == ')') {**

**while (stack.peek() != '(') {**

**postfix.append(stack.pop()).append(" ");**

**}**

**stack.pop(); // Remove '('**

**} else {**

**while (!stack.isEmpty() && getPrecedence(c) <= getPrecedence(stack.peek())) {**

**postfix.append(stack.pop()).append(" ");**

**}**

**stack.push(c);**

**}**

**}**

**// Pop remaining operators from the stack**

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

**postfix.append(stack.pop()).append(" ");**

**}**

**return postfix.toString().trim();**

**}**

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

**// Test Case 1**

**String infix1 = "A + B \* C";**

**String postfix1 = infixToPostfix(infix1);**

**System.out.println("Infix: " + infix1);**

**System.out.println("Postfix: " + postfix1);**

**// Test Case 2**

**String infix2 = "A \* B + C / D";**

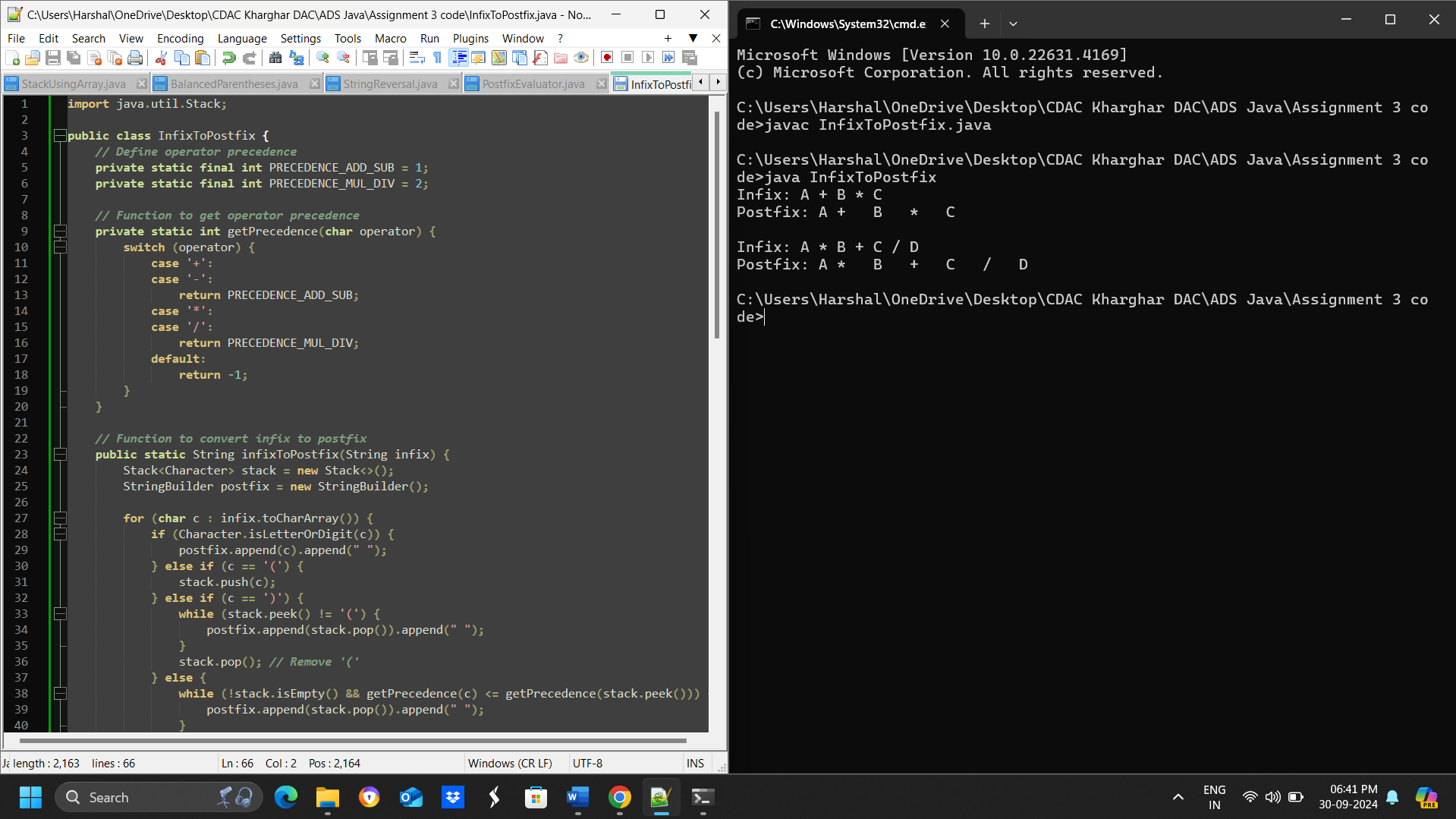
**String postfix2 = infixToPostfix(infix2);**

**System.out.println("\nInfix: " + infix2);**

**System.out.println("Postfix: " + postfix2);**

**}**

**}**

****

* **Test Case 1**:  
  Input: "A + B \* C"  
  Output: "A B C \* +"
* **Test Case 2**:  
  Input: "A \* B + C / D"  
  Output: "A B \* C D / +"

1. **Implement a Queue using an array.**

**public class Queue {**

**private int[] arr;**

**private int front;**

**private int rear;**

**private int capacity;**

**private int size;**

**public Queue(int capacity) {**

**this.capacity = capacity;**

**this.arr = new int[capacity];**

**this.front = 0;**

**this.rear = 0;**

**this.size = 0;**

**}**

**public void enqueue(int element) {**

**if (size == capacity) {**

**System.out.println("Queue is full. Cannot enqueue " + element);**

**return;**

**}**

**arr[rear] = element;**

**rear = (rear + 1) % capacity;**

**size++;**

**}**

**public int dequeue() {**

**if (size == 0) {**

**System.out.println("Queue is empty. Cannot dequeue");**

**return -1;**

**}**

**int element = arr[front];**

**front = (front + 1) % capacity;**

**size--;**

**return element;**

**}**

**public void printQueue() {**

**System.out.print("Queue = [");**

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

**System.out.print(arr[(front + i) % capacity]);**

**if (i < size - 1) {**

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

**}**

**}**

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

**}**

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

**Queue queue = new Queue(5);**

**// Test Case 1**

**queue.enqueue(5);**

**queue.enqueue(10);**

**int dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

**queue.printQueue();**

**// Test Case 2**

**queue = new Queue(5);**

**queue.enqueue(1);**

**queue.enqueue(2);**

**queue.enqueue(3);**

**dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

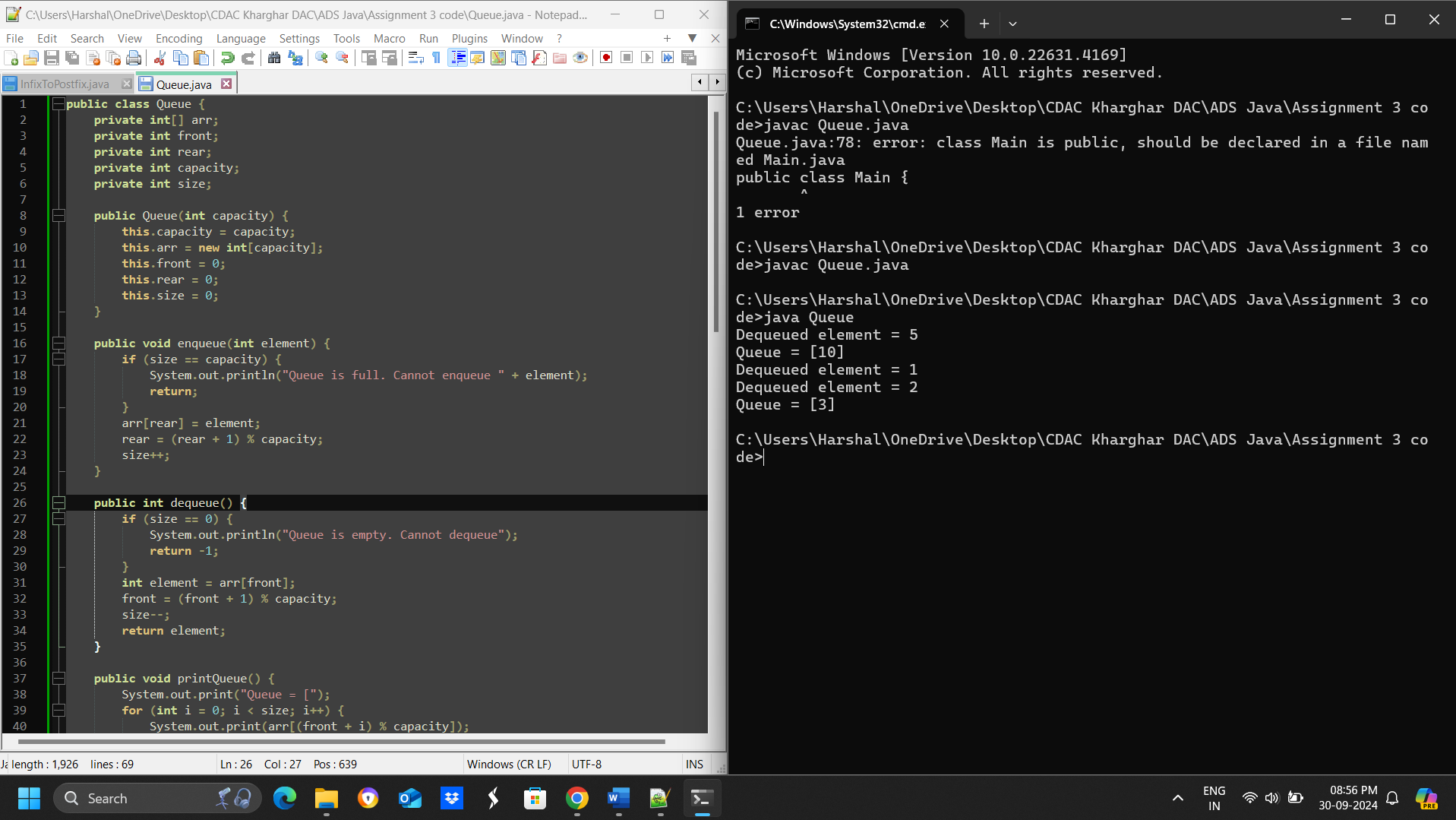
**dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

**queue.printQueue();**

**}**

**}**

****

* **Test Case 1**:  
  Input: Enqueue 5, Enqueue 10, Dequeue  
  Output: Queue = [10], Dequeued element = 5
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, Dequeue, Dequeue  
  Output: Queue = [3], Dequeued elements = 1, 2

1. **Implement a Circular Queue using an array.**

**import java.util.Scanner;**

**public class CircularQueue {**

**int[] arr;**

**int front, rear, size, capacity;**

**CircularQueue(int capacity) {**

**this.capacity = capacity;**

**this.arr = new int[capacity];**

**this.front = 0;**

**this.rear = 0;**

**this.size = 0;**

**}**

**void enqueue(int element) {**

**if (size == capacity) {**

**System.out.println("Queue is full. Cannot enqueue " + element);**

**return;**

**}**

**arr[rear] = element;**

**rear = (rear + 1) % capacity;**

**size++;**

**}**

**int dequeue() {**

**if (size == 0) {**

**System.out.println("Queue is empty. Cannot dequeue");**

**return -1;**

**}**

**int element = arr[front];**

**front = (front + 1) % capacity;**

**size--;**

**return element;**

**}**

**void printQueue() {**

**int i = front;**

**while (i != rear) {**

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

**i = (i + 1) % capacity;**

**}**

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

**}**

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

**CircularQueue queue = new CircularQueue(5);**

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

**while (true) {**

**System.out.print("Enter command (enqueue/dequeue/print): ");**

**String command = scanner.next();**

**if (command.equals("enqueue")) {**

**System.out.print("Enter element to enqueue: ");**

**int element = scanner.nextInt();**

**queue.enqueue(element);**

**} else if (command.equals("dequeue")) {**

**queue.dequeue();**

**} else if (command.equals("print")) {**

**queue.printQueue();**

**} else {**

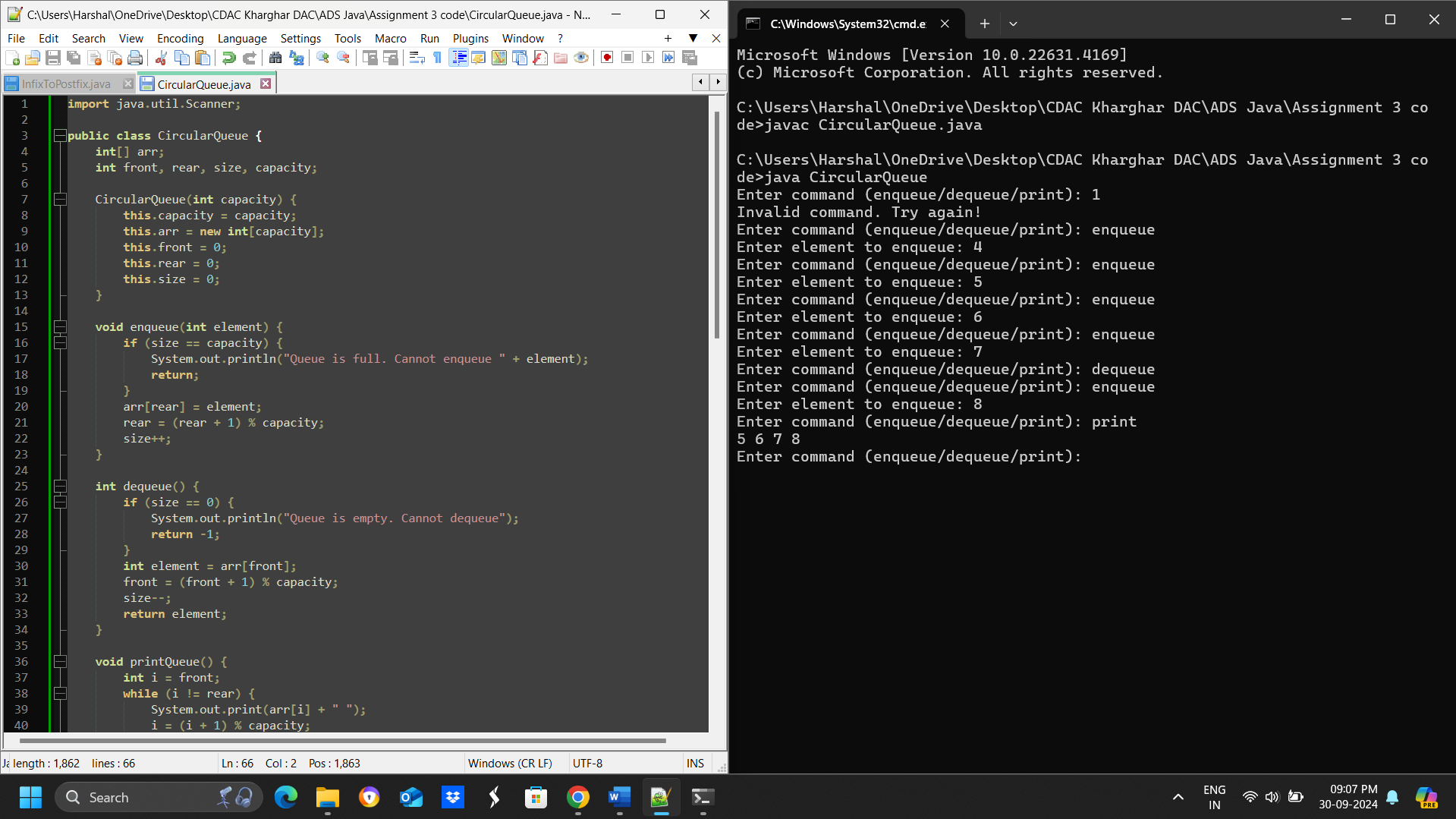
**System.out.println("Invalid command. Try again!");**

**}**

**}**

**}**

**}**

****

* **Test Case 1**:  
  Input: Enqueue 4, 5, 6, 7, Dequeue, Enqueue 8  
  Output: Queue = [8, 5, 6, 7]
* **Test Case 2**:  
  Input: Enqueue 1, 2, 3, 4, Dequeue, Dequeue, Enqueue 5  
  Output: Queue = [5, 3, 4]

1. **Implement a Queue using two Stacks.**

**import java.util.Stack;**

**class Queue {**

**private Stack<Integer> stack1;**

**private Stack<Integer> stack2;**

**public Queue() {**

**stack1 = new Stack<>();**

**stack2 = new Stack<>();**

**}**

**public void enqueue(int element) {**

**stack1.push(element);**

**}**

**public int dequeue() {**

**if (stack2.isEmpty()) {**

**while (!stack1.isEmpty()) {**

**stack2.push(stack1.pop());**

**}**

**}**

**return stack2.pop();**

**}**

**public void printQueue() {**

**System.out.print("Queue = [");**

**for (int i = stack2.size() - 1; i >= 0; i--) {**

**System.out.print(stack2.elementAt(i));**

**if (i > 0) {**

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

**}**

**}**

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

**}**

**}**

**public class Main {**

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

**Queue queue = new Queue();**

**// Test Case 1**

**queue.enqueue(3);**

**queue.enqueue(7);**

**int dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

**queue.printQueue();**

**// Test Case 2**

**queue = new Queue();**

**queue.enqueue(10);**

**queue.enqueue(20);**

**dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

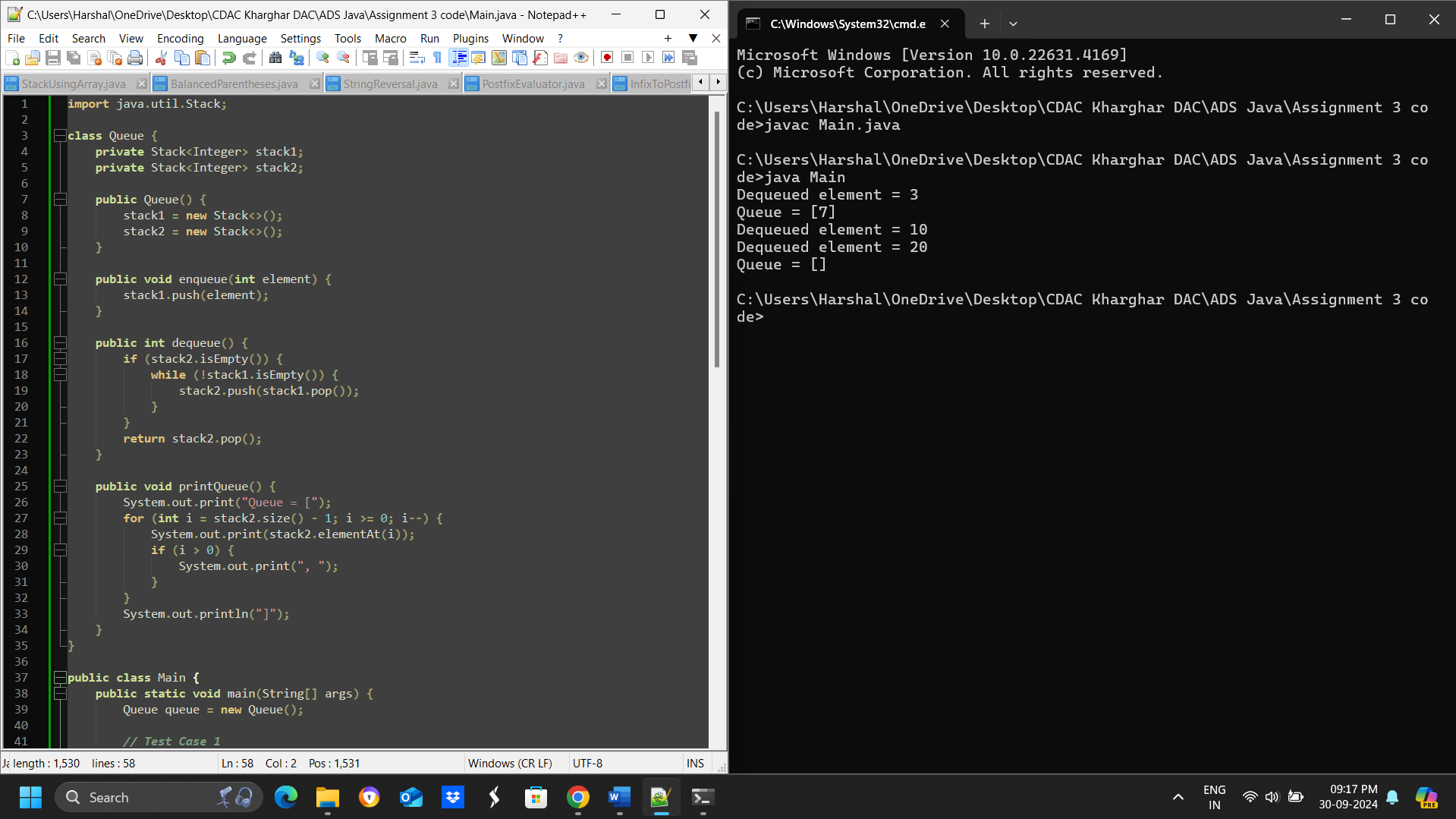
**dequeuedElement = queue.dequeue();**

**System.out.println("Dequeued element = " + dequeuedElement);**

**queue.printQueue();**

**}**

**}**

****

* **Test Case 1**:  
  Input: Enqueue 3, Enqueue 7, Dequeue  
  Output: Queue = [7], Dequeued element = 3
* **Test Case 2**:  
  Input: Enqueue 10, 20, Dequeue, Dequeue  
  Output: Queue = [], Dequeued elements = 10, 20

1. **Implement a Min-Heap.**

**public class MinHeap {**

**private int[] heap;**

**private int size;**

**public MinHeap(int capacity) {**

**heap = new int[capacity + 1];**

**size = 0;**

**}**

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

**if (size == heap.length - 1) {**

**resize();**

**}**

**heap[++size] = value;**

**percolateUp(size);**

**}**

**public int extractMin() {**

**if (size == 0) {**

**throw new RuntimeException("Heap is empty");**

**}**

**int min = heap[1];**

**heap[1] = heap[size--];**

**percolateDown(1);**

**return min;**

**}**

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

**while (index > 1 && heap[index / 2] > heap[index]) {**

**swap(index, index / 2);**

**index = index / 2;**

**}**

**}**

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

**while (index \* 2 <= size) {**

**int child = index \* 2;**

**if (child + 1 <= size && heap[child + 1] < heap[child]) {**

**child++;**

**}**

**if (heap[index] <= heap[child]) {**

**break;**

**}**

**swap(index, child);**

**index = child;**

**}**

**}**

**private void swap(int i, int j) {**

**int temp = heap[i];**

**heap[i] = heap[j];**

**heap[j] = temp;**

**}**

**private void resize() {**

**int[] newHeap = new int[heap.length \* 2];**

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

**newHeap[i] = heap[i];**

**}**

**heap = newHeap;**

**}**

**public void printHeap() {**

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

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

**}**

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

**}**

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

**MinHeap heap = new MinHeap(5);**

**// Test Case 1**

**heap.insert(10);**

**heap.insert(15);**

**heap.insert(20);**

**heap.insert(17);**

**System.out.println("Extracted Min: " + heap.extractMin());**

**heap.printHeap();**

**// Test Case 2**

**heap = new MinHeap(5);**

**heap.insert(30);**

**heap.insert(40);**

**heap.insert(20);**

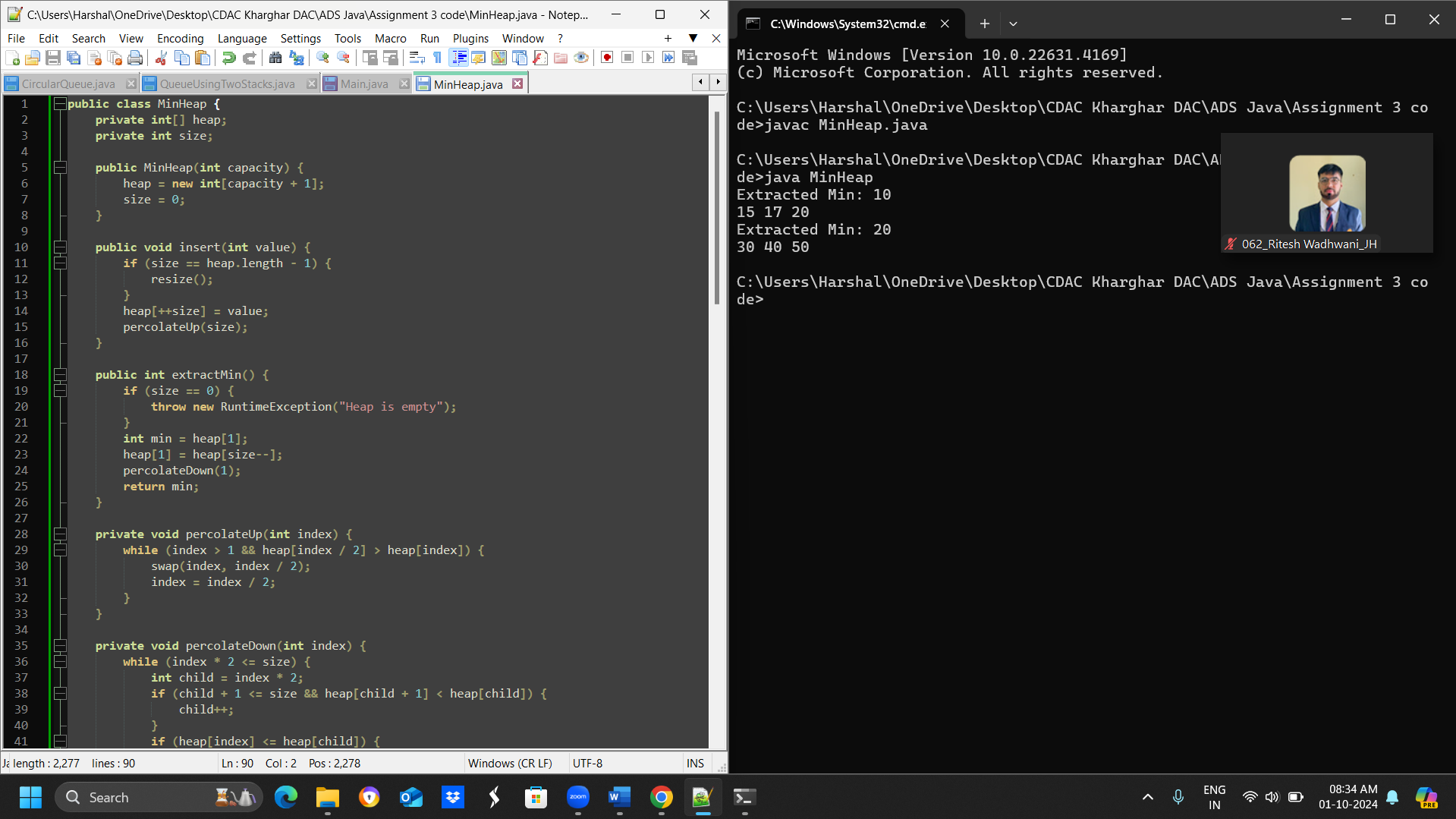
**heap.insert(50);**

**System.out.println("Extracted Min: " + heap.extractMin());**

**heap.printHeap();**

**}**

**}**

****

* **Test Case 1**:  
  Input: Insert 10, 15, 20, 17, Extract Min  
  Output: Min-Heap = [15, 17, 20], Extracted Min = 10
* **Test Case 2**:  
  Input: Insert 30, 40, 20, 50, Extract Min  
  Output: Min-Heap = [30, 40, 50], Extracted Min = 20

1. **Implement a Max-Heap.**

**public class MaxHeap {**

**private int[] heap;**

**private int size;**

**public MaxHeap(int capacity) {**

**heap = new int[capacity + 1];**

**size = 0;**

**}**

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

**if (size == heap.length - 1) {**

**resize();**

**}**

**heap[++size] = value;**

**int index = size;**

**while (index > 1 && heap[index / 2] < heap[index]) {**

**swap(index, index / 2);**

**index = index / 2;**

**}**

**}**

**public int extractMax() {**

**if (size == 0) {**

**throw new RuntimeException("Heap is empty");**

**}**

**int max = heap[1];**

**heap[1] = heap[size--];**

**heapifyDown();**

**return max;**

**}**

**private void heapifyDown() {**

**int index = 1;**

**while (hasLeftChild(index)) {**

**int largerChildIndex = getLargerChildIndex(index);**

**if (heap[index] >= heap[largerChildIndex]) {**

**break;**

**} else {**

**swap(index, largerChildIndex);**

**index = largerChildIndex;**

**}**

**}**

**}**

**private boolean hasLeftChild(int index) {**

**return index \* 2 <= size;**

**}**

**private int getLargerChildIndex(int index) {**

**if (hasRightChild(index) && heap[index \* 2] < heap[index \* 2 + 1]) {**

**return index \* 2 + 1;**

**} else {**

**return index \* 2;**

**}**

**}**

**private boolean hasRightChild(int index) {**

**return index \* 2 + 1 <= size;**

**}**

**private void swap(int i, int j) {**

**int temp = heap[i];**

**heap[i] = heap[j];**

**heap[j] = temp;**

**}**

**private void resize() {**

**int[] newHeap = new int[heap.length \* 2];**

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

**newHeap[i] = heap[i];**

**}**

**heap = newHeap;**

**}**

**public void printHeap() {**

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

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

**}**

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

**}**

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

**MaxHeap maxHeap = new MaxHeap(5);**

**// Test Case 1**

**maxHeap.insert(12);**

**maxHeap.insert(7);**

**maxHeap.insert(15);**

**maxHeap.insert(5);**

**System.out.println("Max-Heap = " + maxHeap.heap[1] + " " + maxHeap.heap[2] + " " + maxHeap.heap[3] + " " + maxHeap.heap[4]);**

**System.out.println("Extracted Max = " + maxHeap.extractMax());**

**// Test Case 2**

**maxHeap = new MaxHeap(5);**

**maxHeap.insert(8);**

**maxHeap.insert(20);**

**maxHeap.insert(10);**

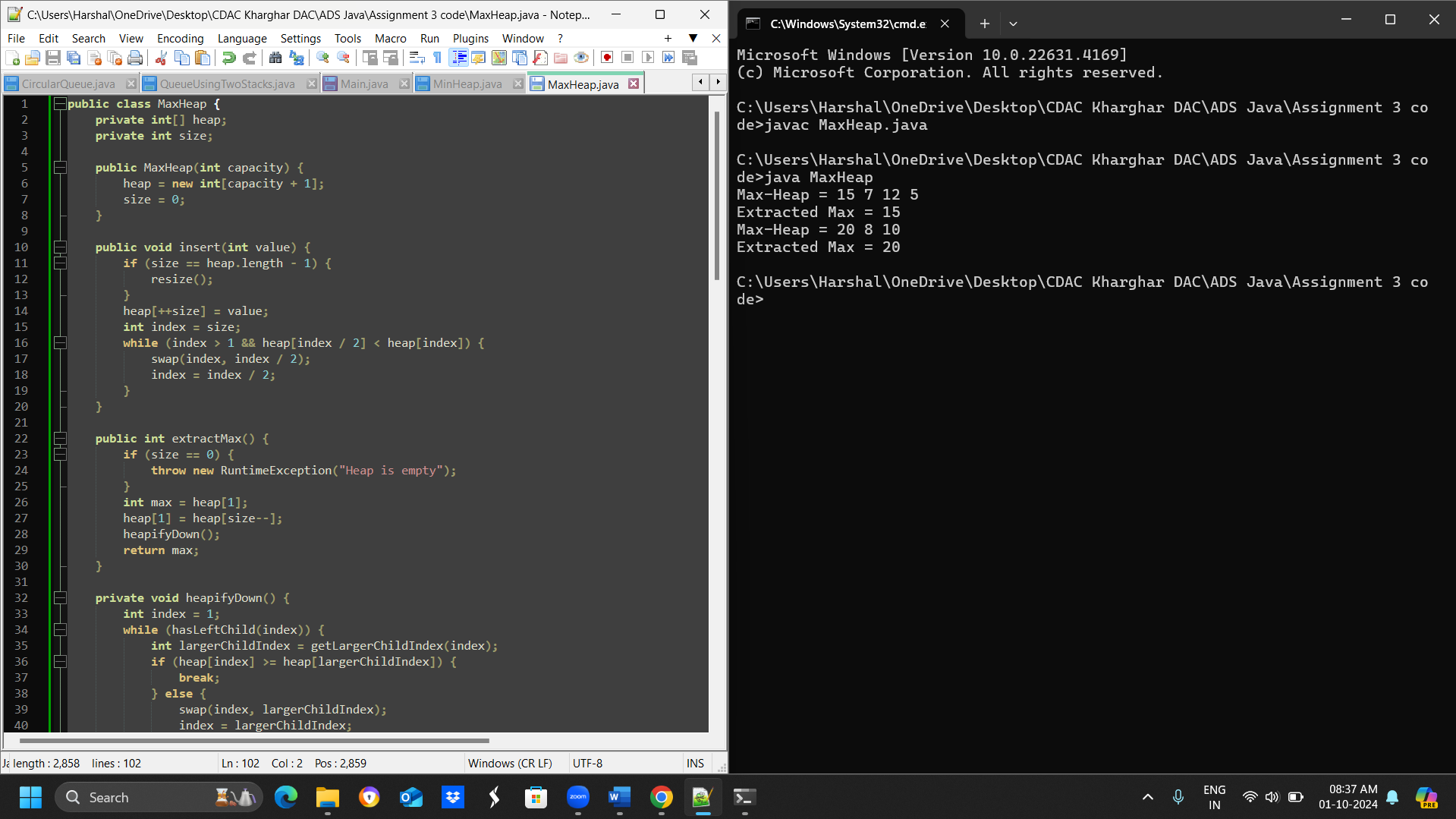
**maxHeap.insert(3);**

**System.out.println("Max-Heap = " + maxHeap.heap[1] + " " + maxHeap.heap[2] + " " + maxHeap.heap[3]);**

**System.out.println("Extracted Max = " + maxHeap.extractMax());**

**}**

**}**

****

* **Test Case 1**:  
  Input: Insert 12, 7, 15, 5, Extract Max  
  Output: Max-Heap = [12, 7, 5], Extracted Max = 15
* **Test Case 2**:  
  Input: Insert 8, 20, 10, 3, Extract Max  
  Output: Max-Heap = [10, 8, 3], Extracted Max = 20

1. **Sort an array using a heap (Heap Sort).**

**public class HeapSort {**

**public static void heapify(int[] arr, int n, int i) {**

**int largest = i;**

**int left = 2 \* i + 1;**

**int right = 2 \* i + 2;**

**if (left < n && arr[left] > arr[largest]) {**

**largest = left;**

**}**

**if (right < n && arr[right] > arr[largest]) {**

**largest = right;**

**}**

**if (largest != i) {**

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

**arr[i] = arr[largest];**

**arr[largest] = temp;**

**heapify(arr, n, largest);**

**}**

**}**

**public static void sort(int[] arr) {**

**int n = arr.length;**

**// Build max heap**

**for (int i = n / 2 - 1; i >= 0; i--) {**

**heapify(arr, n, i);**

**}**

**// Extract elements one by one**

**for (int i = n - 1; i >= 0; i--) {**

**int temp = arr[0];**

**arr[0] = arr[i];**

**arr[i] = temp;**

**heapify(arr, i, 0);**

**}**

**}**

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

**int[] arr1 = {5, 1, 12, 3, 9};**

**int[] arr2 = {20, 15, 8, 10};**

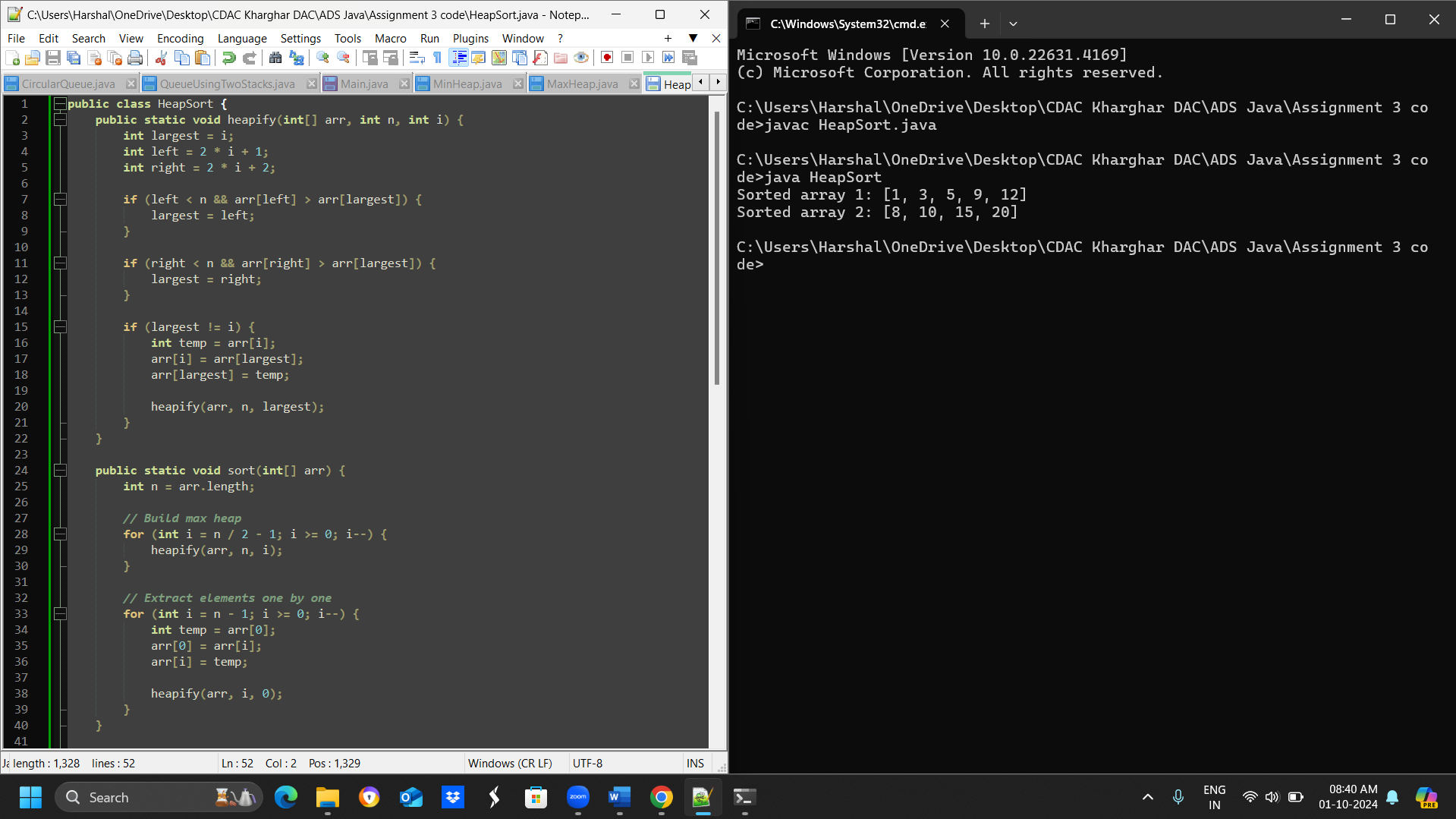
**sort(arr1);**

**sort(arr2);**

**System.out.println("Sorted array 1: " + java.util.Arrays.toString(arr1));**

**System.out.println("Sorted array 2: " + java.util.Arrays.toString(arr2));**

**}**

**}**

* **Test Case 1**:  
  Input: [5, 1, 12, 3, 9]  
  Output: [1, 3, 5, 9, 12]
* **Test Case 2**:  
  Input: [20, 15, 8, 10]  
  Output: [8, 10, 15, 20]

1. **Find the kth largest element in a stream of numbers using a heap.**

**import java.util.\*;**

**public class KthLargestElement {**

**public static int findKthLargest(int[] stream, int k) {**

**// Create a min-heap with capacity k**

**PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);**

**// Iterate through the stream of numbers**

**for (int num : stream) {**

**// Add the number to the min-heap**

**minHeap.add(num);**

**// If the heap size exceeds k, remove the smallest element**

**if (minHeap.size() > k) {**

**minHeap.poll();**

**}**

**}**

**// The kth largest element is the top of the min-heap**

**return minHeap.peek();**

**}**

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

**int[] stream1 = {3, 10, 5, 20, 15};**

**int k1 = 3;**

**System.out.println("Test Case 1: " + findKthLargest(stream1, k1)); // Output: 10**

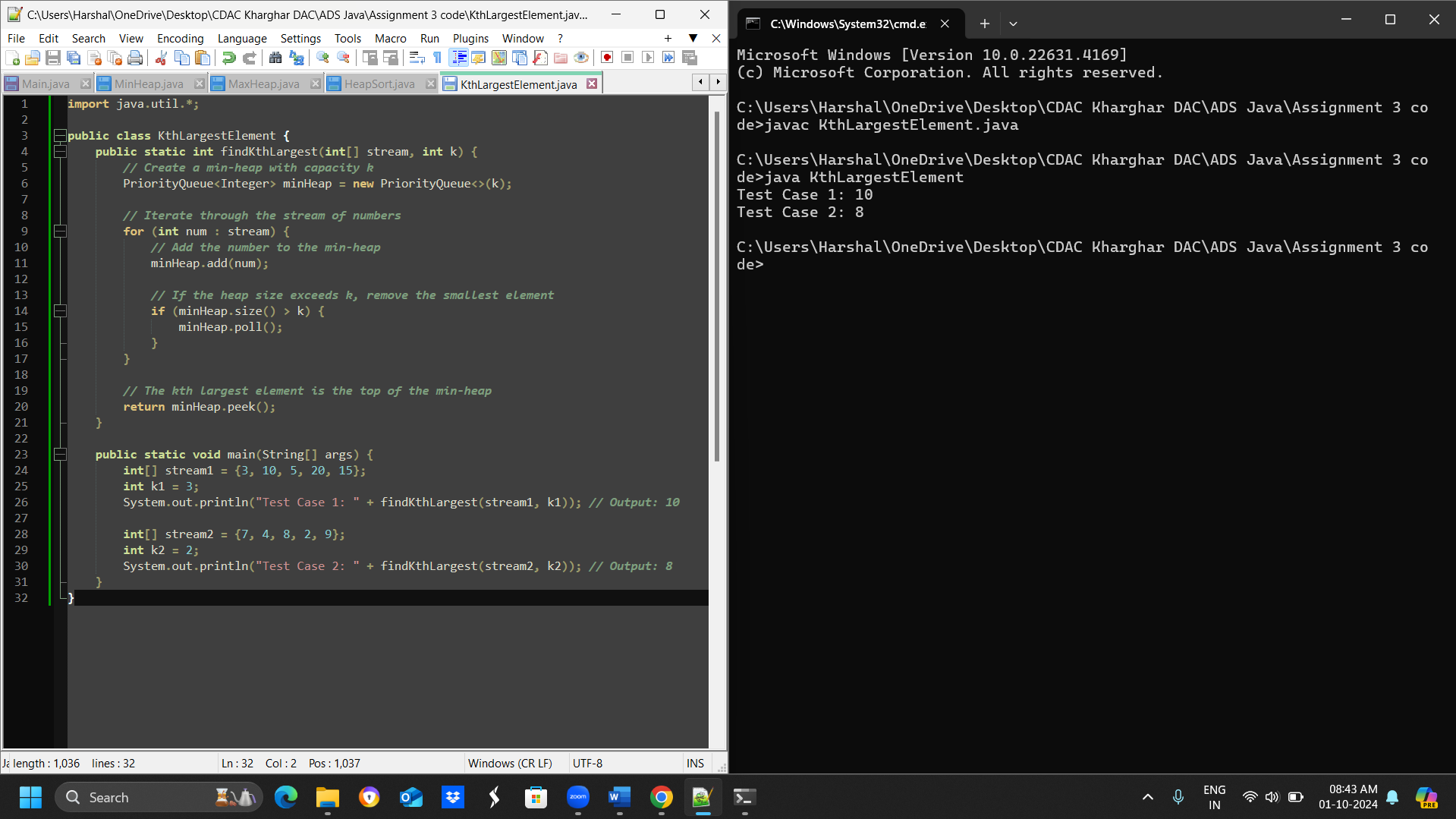
**int[] stream2 = {7, 4, 8, 2, 9};**

**int k2 = 2;**

**System.out.println("Test Case 2: " + findKthLargest(stream2, k2)); // Output: 8**

**}**

**}**

****

* **Test Case 1**:  
  Input: Stream = [3, 10, 5, 20, 15], k = 3  
  Output: 10
* **Test Case 2**:  
  Input: Stream = [7, 4, 8, 2, 9], k = 2  
  Output: 8

1. **Implement a Priority Queue using a heap.**

**import java.util.\*;**

**public class PriorityQueue {**

**private PriorityQueue() {} // prevent instantiation**

**public static class Heap {**

**private List<Element> elements;**

**public Heap() {**

**elements = new ArrayList<>();**

**}**

**public void enqueue(int value, int priority) {**

**Element element = new Element(value, priority);**

**elements.add(element);**

**heapifyUp(elements.size() - 1);**

**}**

**public int dequeue() {**

**if (elements.isEmpty()) {**

**throw new NoSuchElementException();**

**}**

**Element maxElement = elements.get(0);**

**elements.set(0, elements.get(elements.size() - 1));**

**elements.remove(elements.size() - 1);**

**heapifyDown(0);**

**return maxElement.value;**

**}**

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

**while (index > 0) {**

**int parentIndex = (index - 1) / 2;**

**if (elements.get(parentIndex).priority >= elements.get(index).priority) {**

**break;**

**}**

**swap(parentIndex, index);**

**index = parentIndex;**

**}**

**}**

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

**while (true) {**

**int leftChildIndex = 2 \* index + 1;**

**int rightChildIndex = 2 \* index + 2;**

**int largestIndex = index;**

**if (leftChildIndex < elements.size() && elements.get(leftChildIndex).priority > elements.get(largestIndex).priority) {**

**largestIndex = leftChildIndex;**

**}**

**if (rightChildIndex < elements.size() && elements.get(rightChildIndex).priority > elements.get(largestIndex).priority) {**

**largestIndex = rightChildIndex;**

**}**

**if (largestIndex == index) {**

**break;**

**}**

**swap(largestIndex, index);**

**index = largestIndex;**

**}**

**}**

**private void swap(int i, int j) {**

**Element temp = elements.get(i);**

**elements.set(i, elements.get(j));**

**elements.set(j, temp);**

**}**

**private static class Element {**

**int value;**

**int priority;**

**public Element(int value, int priority) {**

**this.value = value;**

**this.priority = priority;**

**}**

**}**

**}**

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

**Heap heap = new Heap();**

**// Test Case 1**

**heap.enqueue(3, 1);**

**heap.enqueue(10, 3);**

**heap.enqueue(5, 2);**

**System.out.println("Dequeued element = " + heap.dequeue()); // 10**

**System.out.println("Priority Queue = " + heap.elements); // [5, 3]**

**// Test Case 2**

**heap = new Heap();**

**heap.enqueue(7, 4);**

**heap.enqueue(8, 2);**

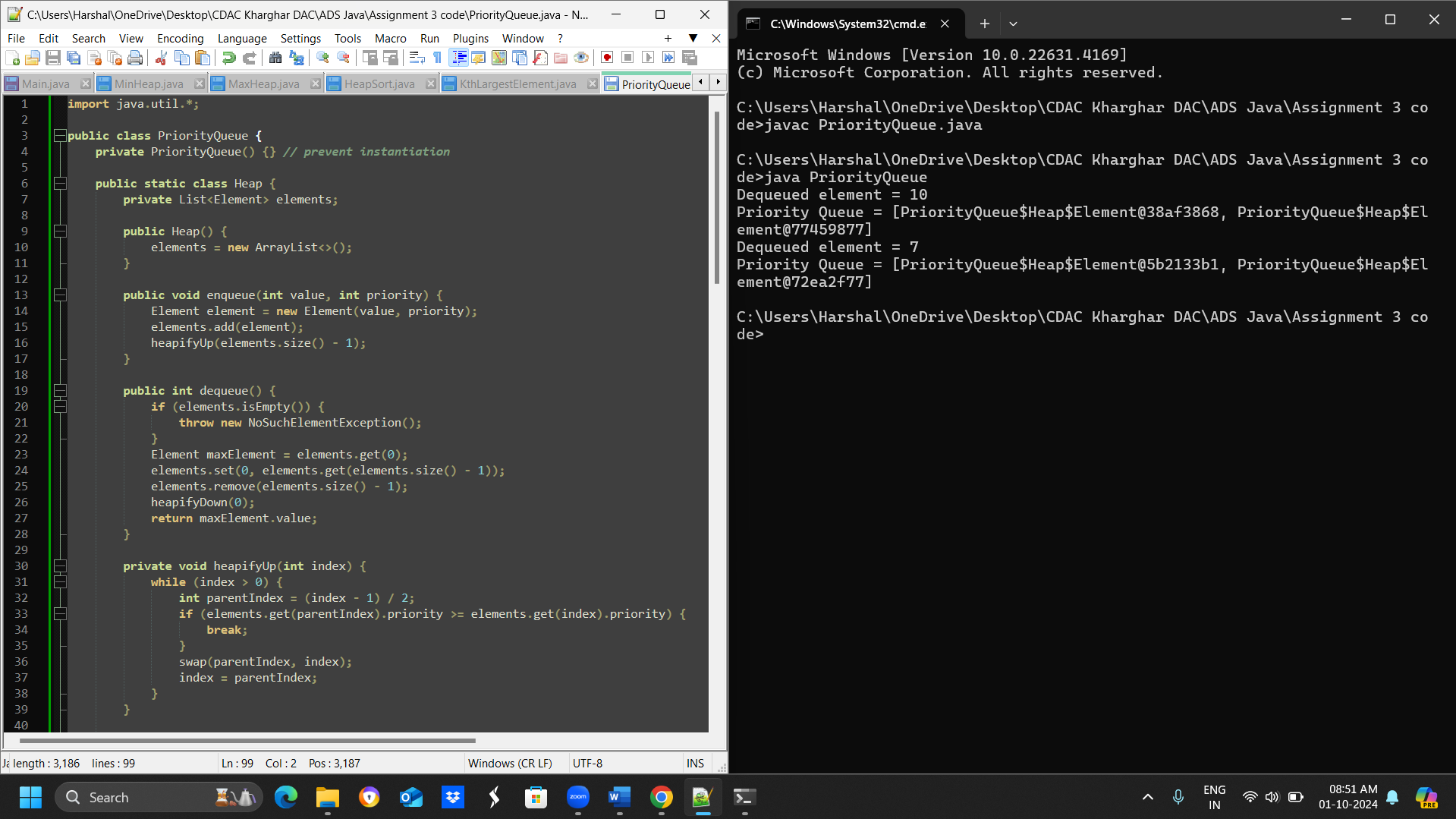
**heap.enqueue(6, 3);**

**System.out.println("Dequeued element = " + heap.dequeue()); // 7**

**System.out.println("Priority Queue = " + heap.elements); // [6, 8]**

**}**

**}**

****

* **Test Case 1**:  
  Input: Enqueue with priorities: 3 (priority 1), 10 (priority 3), 5 (priority 2), Dequeue  
  Output: Dequeued element = 10 (highest priority), Priority Queue = [5, 3]
* **Test Case 2**:  
  Input: Enqueue with priorities: 7 (priority 4), 8 (priority 2), 6 (priority 3), Dequeue  
  Output: Dequeued element = 7, Priority Queue = [6, 8]

1. **Design an algorithm to implement a stack with a getMin() function to return the minimum element in constant time.**

**import java.util.Stack;**

**public class MinStack {**

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

**Stack<Integer> minStack = new Stack<>();**

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

**stack.push(x);**

**if (minStack.isEmpty() || x <= minStack.peek()) {**

**minStack.push(x);**

**}**

**}**

**public void pop() {**

**if (stack.peek().equals(minStack.peek())) {**

**minStack.pop();**

**}**

**stack.pop();**

**}**

**public int getMin() {**

**return minStack.peek();**

**}**

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

**MinStack minStack = new MinStack();**

**minStack.push(5);**

**minStack.push(3);**

**minStack.push(7);**

**System.out.println("Min = " + minStack.getMin());**

**minStack = new MinStack();**

**minStack.push(10);**

**minStack.push(8);**

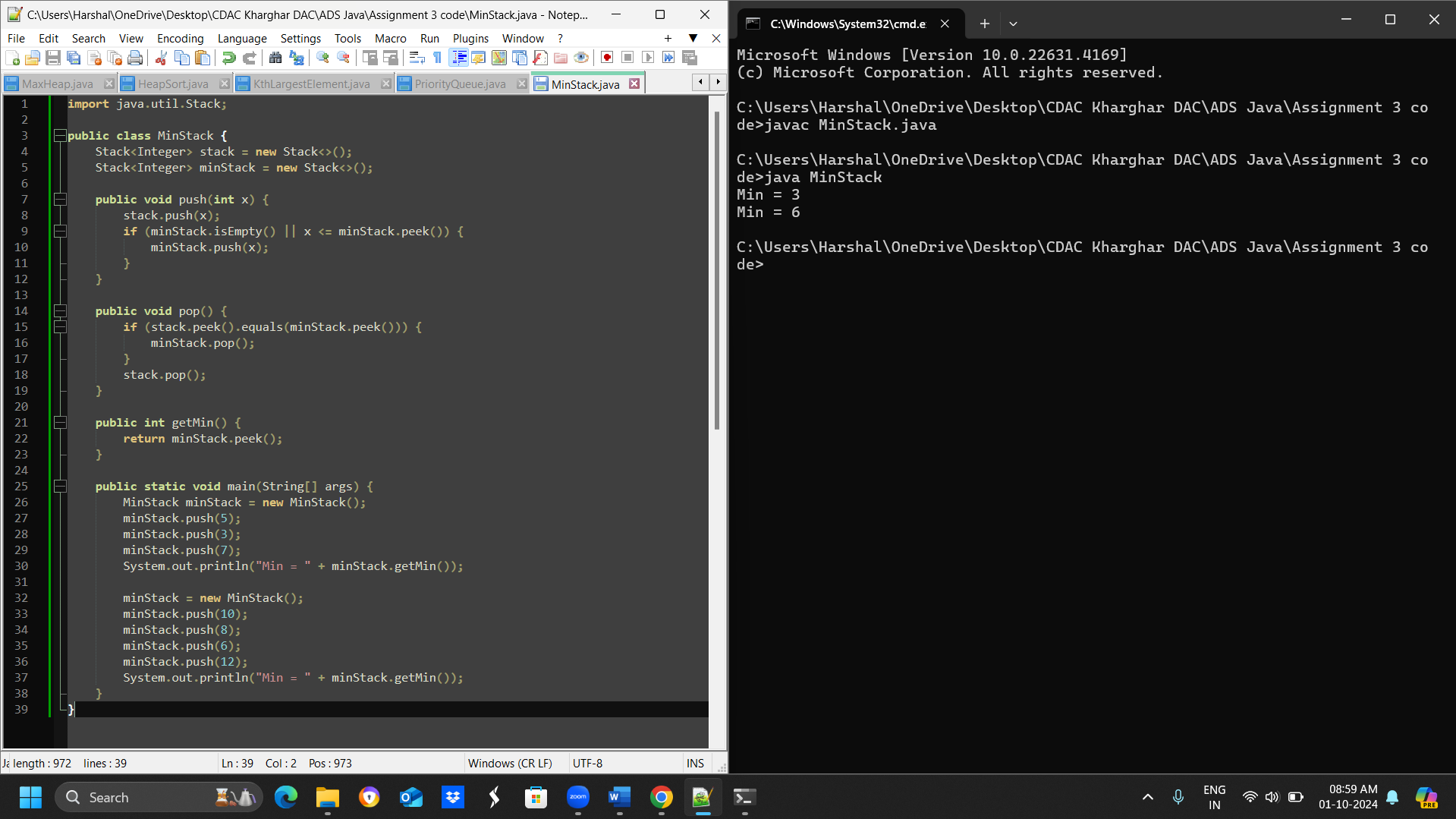
**minStack.push(6);**

**minStack.push(12);**

**System.out.println("Min = " + minStack.getMin());**

**}**

**}**

****

* **Test Case 1**:  
  Input: Push 5, Push 3, Push 7, Get Min  
  Output: Min = 3
* **Test Case 2**:  
  Input: Push 10, Push 8, Push 6, Push 12, Get Min  
  Output: Min = 6

1. **Design a Circular Queue with a fixed size, supporting enqueue, dequeue, and isFull/isEmpty operations.**

**public class FixedSizeQueue {**

**private int[] queue;**

**private int front;**

**private int rear;**

**private int size;**

**private int count;**

**public FixedSizeQueue(int size) {**

**this.queue = new int[size];**

**this.front = 0;**

**this.rear = 0;**

**this.size = size;**

**this.count = 0;**

**}**

**public boolean isEmpty() {**

**return count == 0;**

**}**

**public boolean isFull() {**

**return count == size;**

**}**

**public void enqueue(int element) {**

**if (isFull()) {**

**System.out.println("Queue is full. Cannot enqueue element " + element);**

**return;**

**}**

**queue[rear] = element;**

**rear = (rear + 1) % size;**

**count++;**

**}**

**public int dequeue() {**

**if (isEmpty()) {**

**System.out.println("Queue is empty. Cannot dequeue.");**

**return -1;**

**}**

**int element = queue[front];**

**front = (front + 1) % size;**

**count--;**

**return element;**

**}**

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

**// Test Case 1**

**FixedSizeQueue queue1 = new FixedSizeQueue(4);**

**queue1.enqueue(1);**

**queue1.enqueue(2);**

**queue1.enqueue(3);**

**queue1.enqueue(4);**

**System.out.println("Is queue full? " + queue1.isFull()); // Output: True**

**// Test Case 2**

**FixedSizeQueue queue2 = new FixedSizeQueue(3);**

**queue2.enqueue(5);**

**queue2.enqueue(6);**

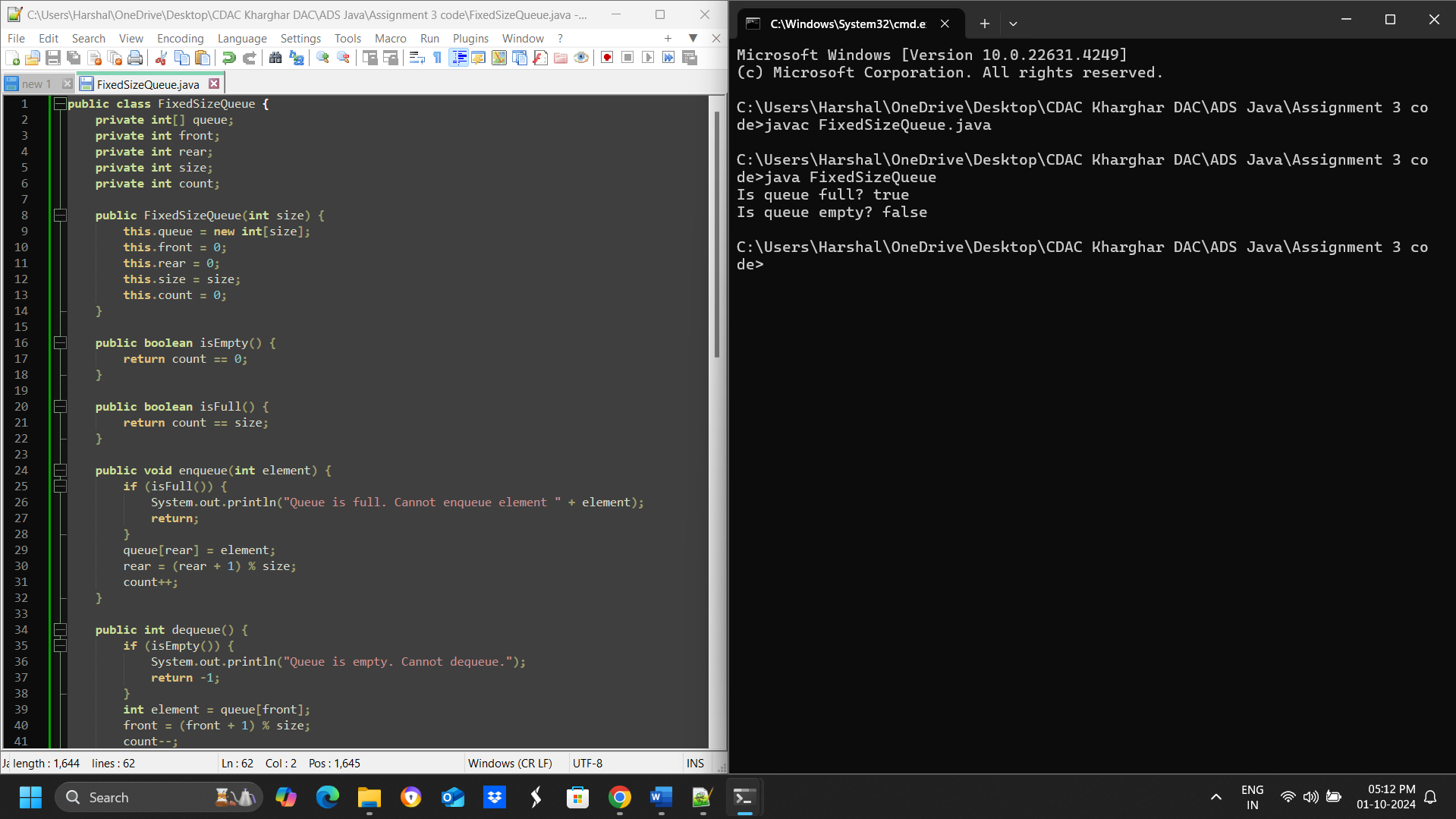
**queue2.dequeue();**

**queue2.enqueue(7);**

**System.out.println("Is queue empty? " + queue2.isEmpty()); // Output: False**

**}**

**}**

****

* **Test Case 1**:  
  Input: Size = 4, Enqueue 1, 2, 3, 4, isFull()  
  Output: True
* **Test Case 2**:  
  Input: Size = 3, Enqueue 5, 6, Dequeue, Enqueue 7, isEmpty()  
  Output: False