**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

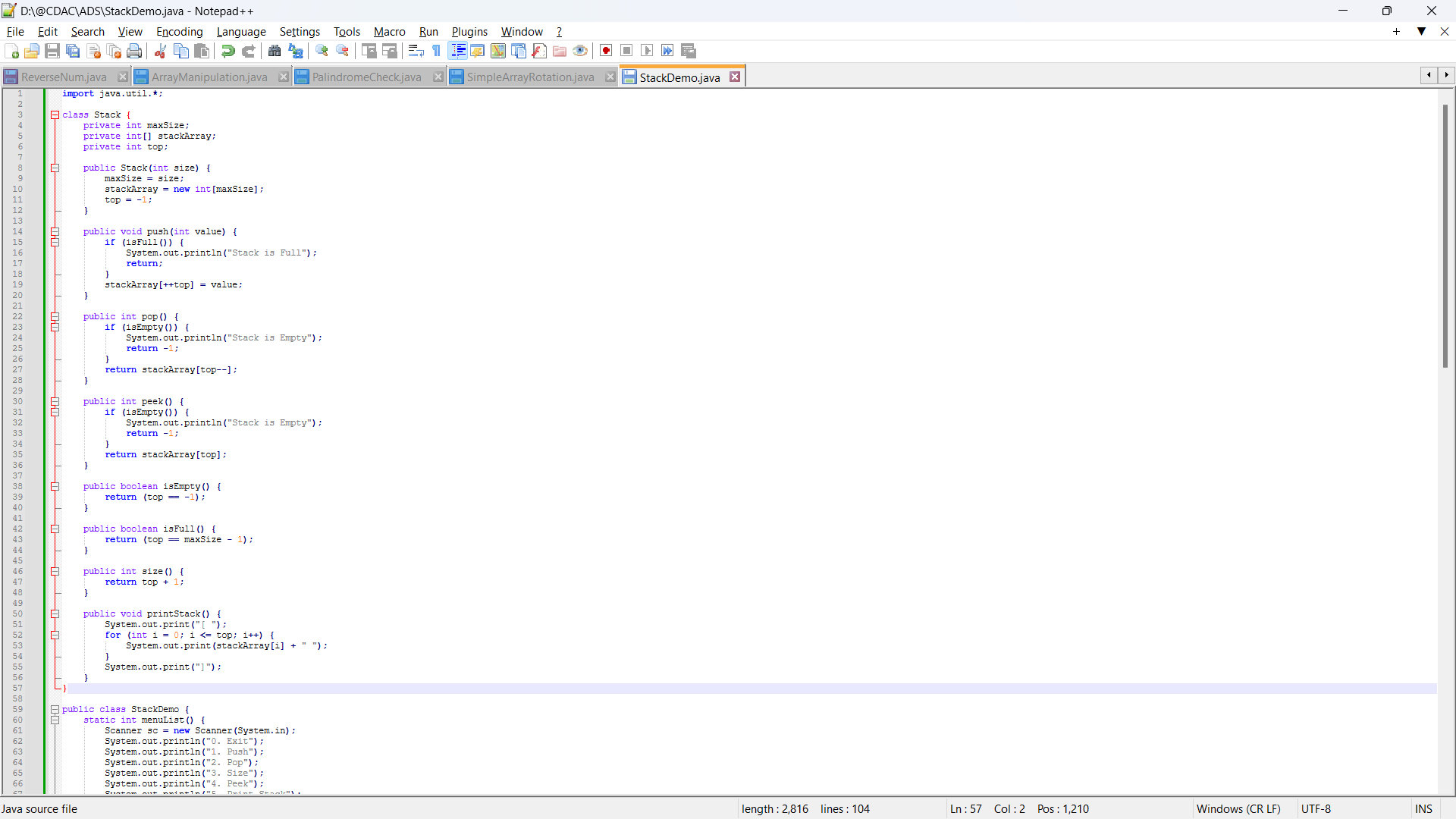
-Output

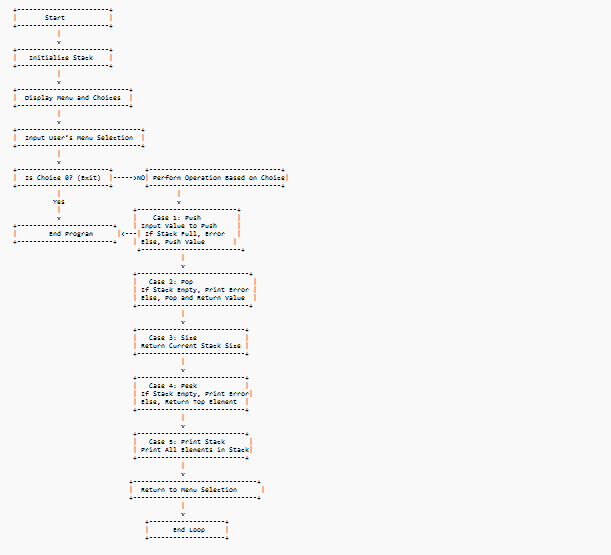
-Time and Space complexity

Submission Date: 01/10/2024

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

* **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





 **Start**: The program begins execution.

 **Initialize Stack**: A stack object is created with a predefined maximum size.

 **Display Menu**: The program displays a menu offering the following options:

* 0: Exit
* 1: Push
* 2: Pop
* 3: Size
* 4: Peek
* 5: Print Stack

 **Input User's Menu Selection**: The user inputs a choice from the menu.

 **Check if Choice is 0 (Exit)**:

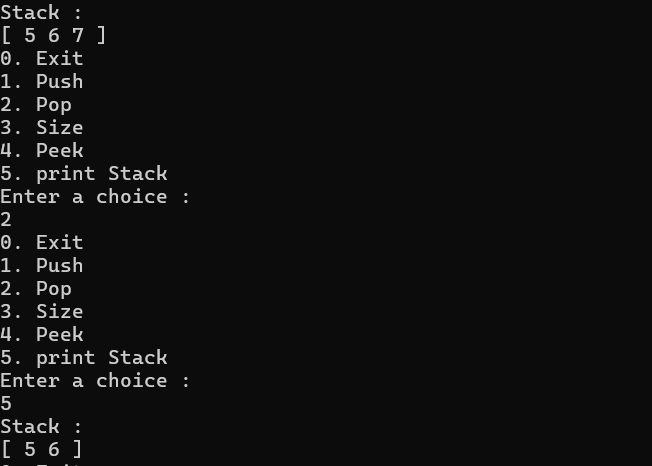
* If the choice is 0, the program ends.
* If the choice is not 0, the program moves on to perform the corresponding stack operation.

 **Perform Operation Based on Choice**:

* **Case 1: Push**:
  + Input the value to be pushed onto the stack.
  + Check if the stack is full:
    - If full, print an error message ("Stack is Full").
    - If not full, push the value and update the stack.
* **Case 2: Pop**:
  + Check if the stack is empty:
    - If empty, print an error message ("Stack is Empty").
    - If not empty, pop the top value and return it.
* **Case 3: Size**:
  + Return the current size of the stack (number of elements).
* **Case 4: Peek**:
  + Check if the stack is empty:
    - If empty, print an error message ("Stack is Empty").
    - If not empty, return the value at the top of the stack.
* **Case 5: Print Stack**:
  + Print all elements currently in the stack.

 **Return to Menu**: After performing the selected operation, the program returns to the menu for another input.

 **End Program**: If the user selects 0, the program ends



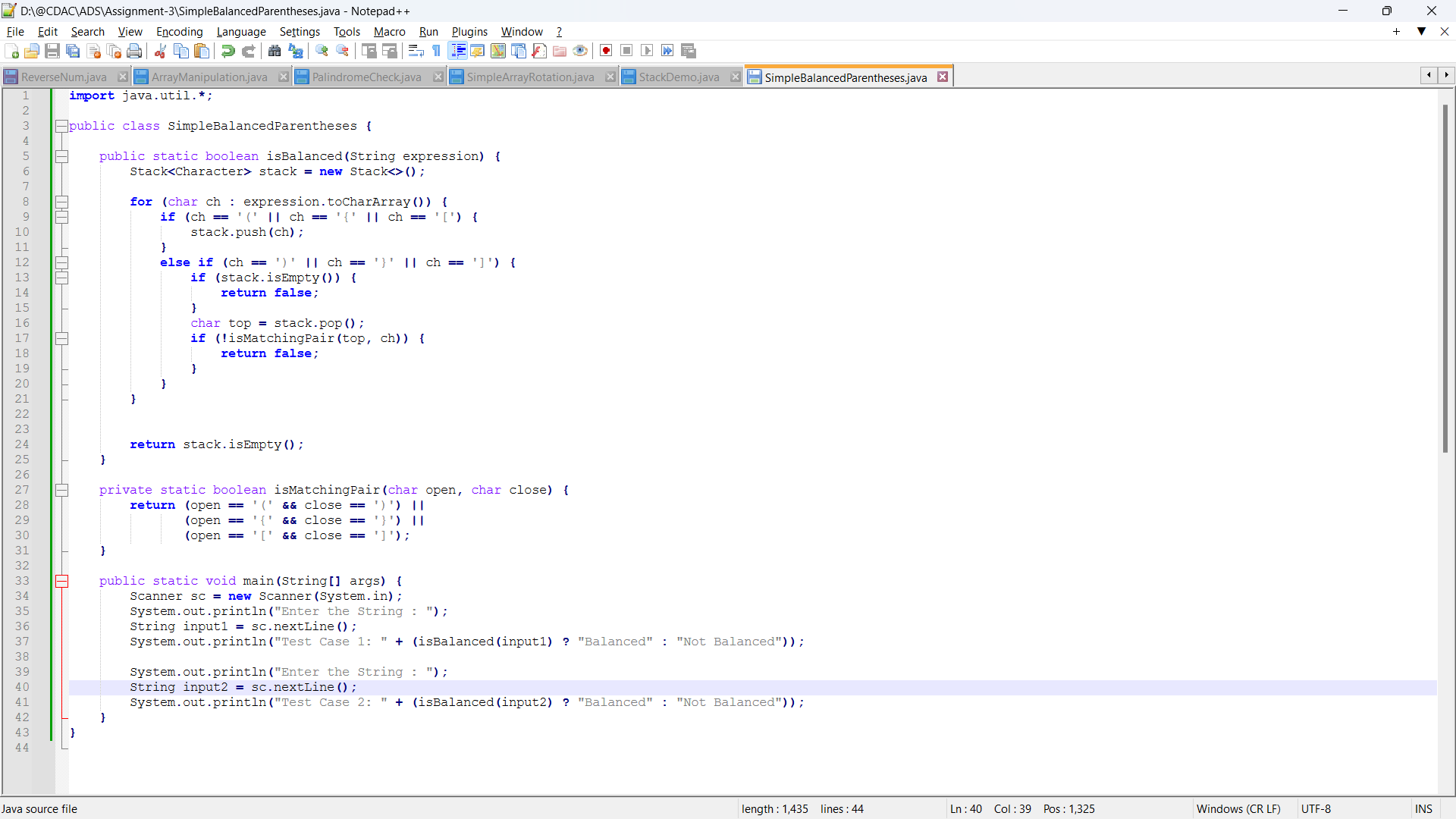
 **Time Complexity**:

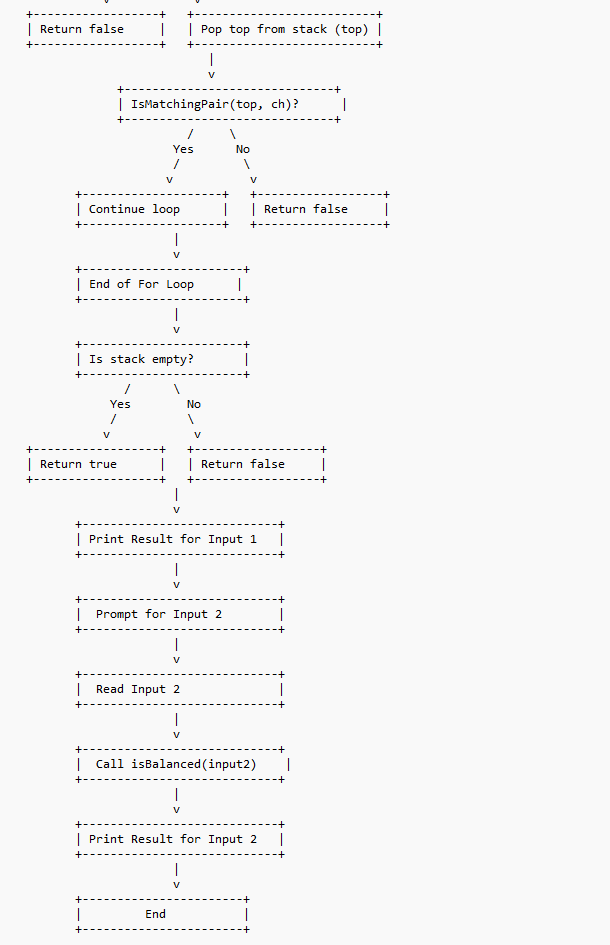
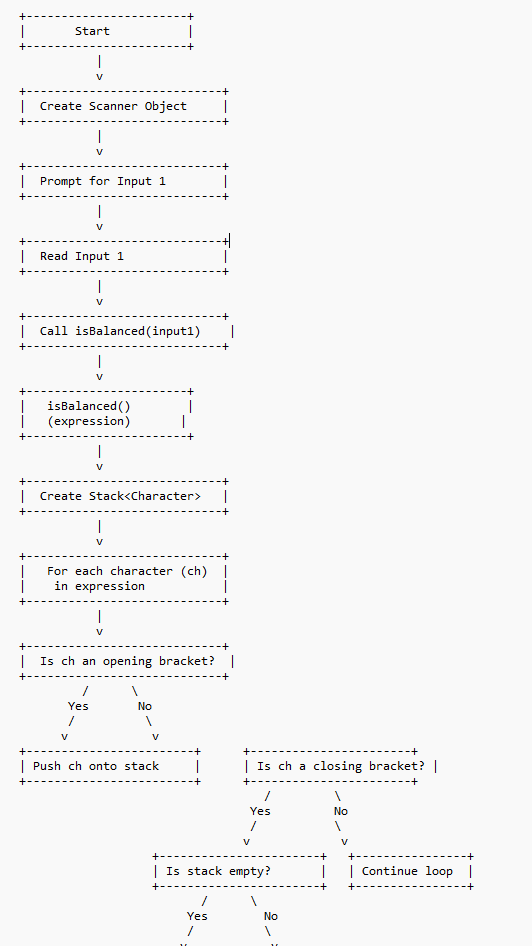
* push(): O(1)
* pop(): O(1)
* peek(): O(1)
* size(): O(1)
* printStack(): O(n)

 **Space Complexity**: O(n) (for storing up to n elements in the stack).

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

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





 **Start**: The program begins execution.

 **Create Scanner Object**: A scanner object is created to read input from the user.

 **Prompt for Input 1**: The user is prompted to enter the first string to check for balanced parentheses.

 **Read Input 1**: The input string is read.

 **Call isBalanced(input1)**: The isBalanced method is called with the first input string.

 **isBalanced() Method**:

* A stack of characters is created to store opening brackets.
* For each character in the expression:
  + If the character is an opening bracket, it is pushed onto the stack.
  + If it is a closing bracket, check if the stack is empty:
    - If the stack is empty, return false (not balanced).
    - If not empty, pop the top element from the stack and check if it matches the closing bracket:
      * If it matches, continue the loop.
      * If it doesn’t match, return false (not balanced).

 **End of For Loop**: After iterating through all characters:

* Check if the stack is empty:
  + If empty, return true (balanced).
  + If not empty, return false (not balanced).

 **Print Result for Input 1**: The result for the first input is printed.

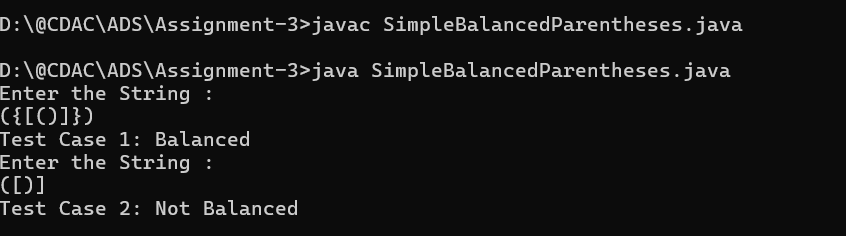
 **Prompt for Input 2**: The user is prompted to enter the second string.

 **Read Input 2**: The second input string is read.

 **Call isBalanced(input2)**: The isBalanced method is called with the second input string.

 **Print Result for Input 2**: The result for the second input is printed.

 **End**: The program ends.

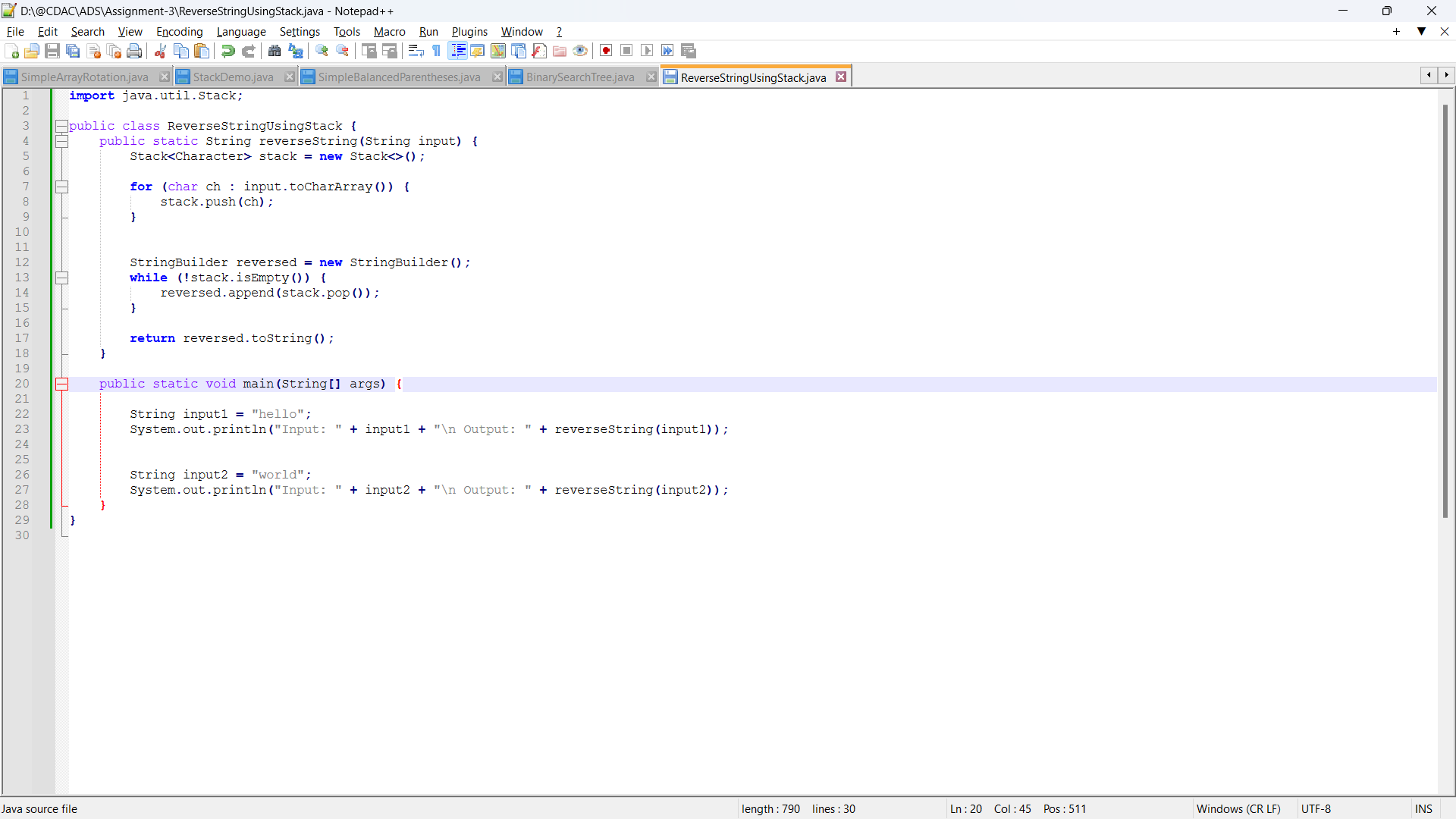


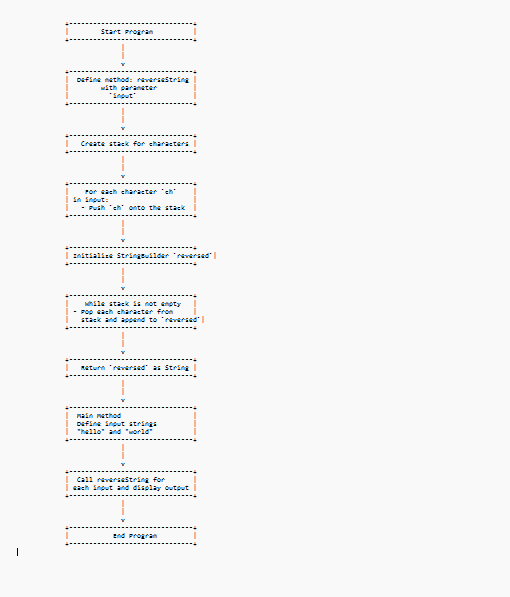
 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

**3. Reverse a string using a stack.**

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





**Expalnation :**

** Start Program:**

* **The program execution begins by defining the class ReverseStringUsingStack.**

** Define Method: reverseString:**

* **The method reverseString takes a single parameter, input, which is the string we want to reverse.**

** Create Stack for Characters:**

* **A stack is initialized to store each character of the input string temporarily.**

** For Each Character ch in Input:**

* **A loop iterates over each character in the input string.**
* **Each character (ch) is pushed onto the stack, one by one. Stacking these characters sets up the reverse order since stacks follow a Last-In, First-Out (LIFO) principle.**

** Initialize StringBuilder reversed:**

* **A StringBuilder named reversed is initialized to construct the reversed version of the input string.**

** While Stack is Not Empty:**

* **A loop begins, running as long as the stack is not empty.**
* **Inside this loop:**
  + **Each character is popped from the stack and appended to reversed. Since the stack’s LIFO order gives characters in reverse, reversed accumulates the input string in reverse.**

** Return reversed as String:**

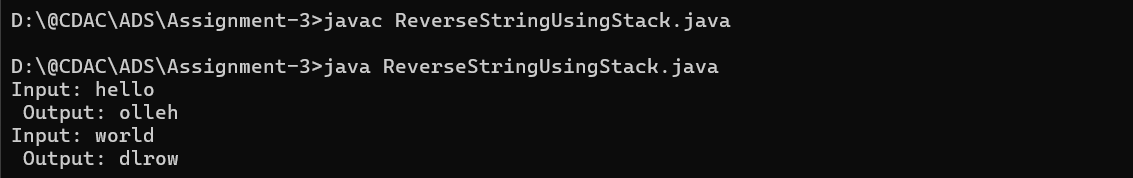
* **After the stack is empty and all characters have been added to reversed, it is converted to a string and returned as the final reversed string.**

** Main Method:**

* **The main method starts, defining two input strings: "hello" and "world".**
* **The reverseString method is called for each input, and the returned reversed strings are printed.**

** End Program:**

* **Once both input strings have been processed, the program ends.**

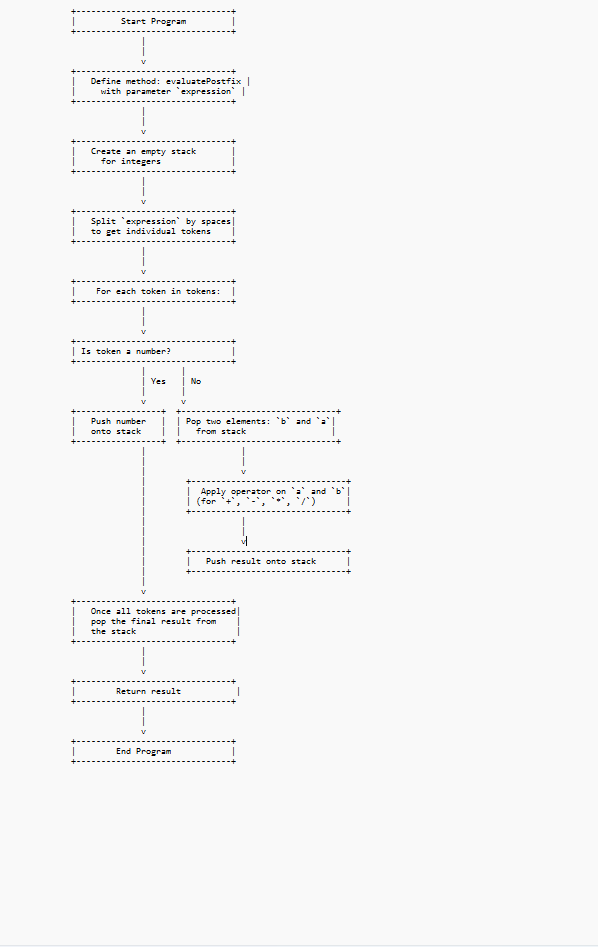
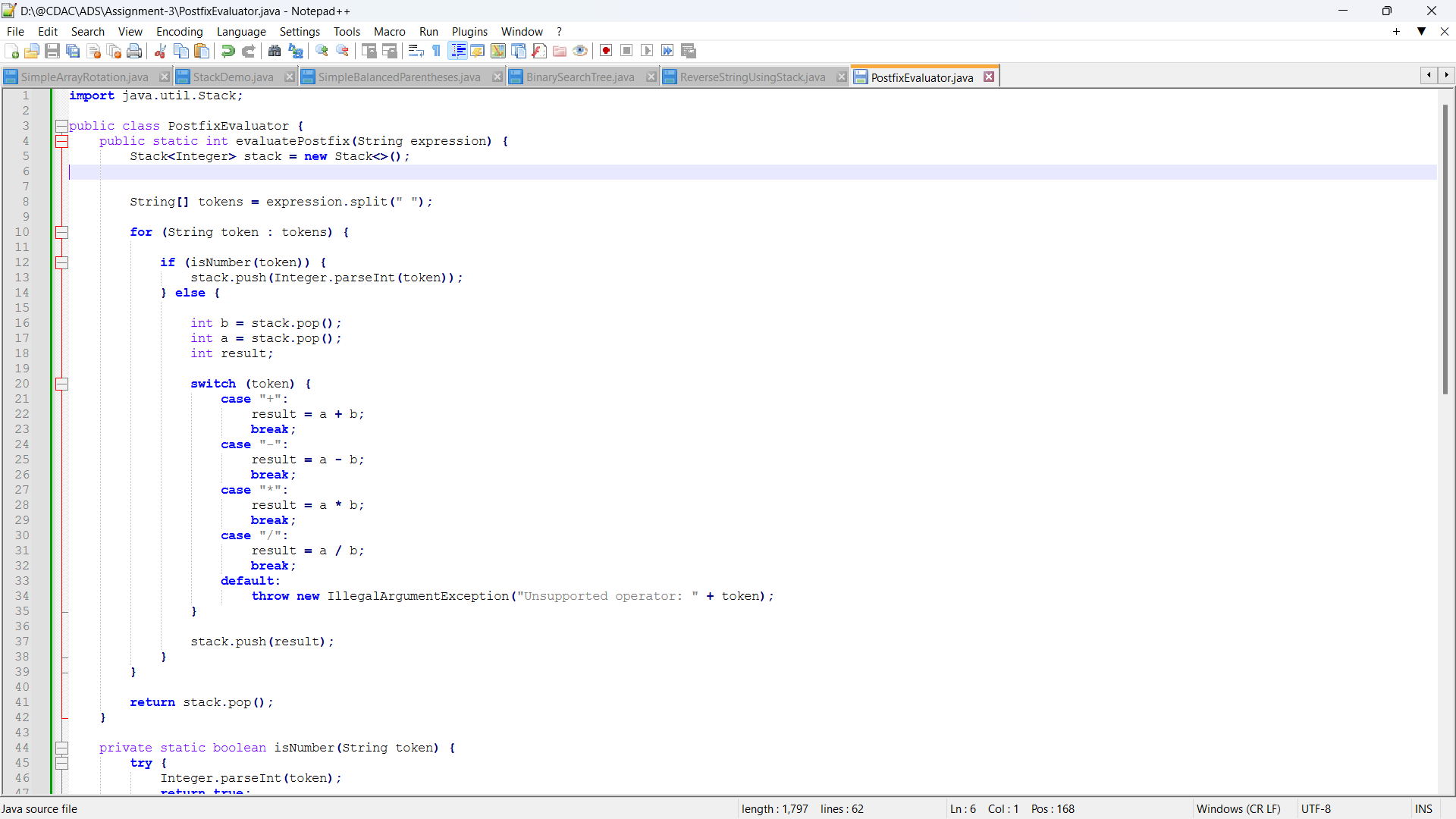
****

** Time Complexity: O(n)**

** Space Complexity: O(n)**

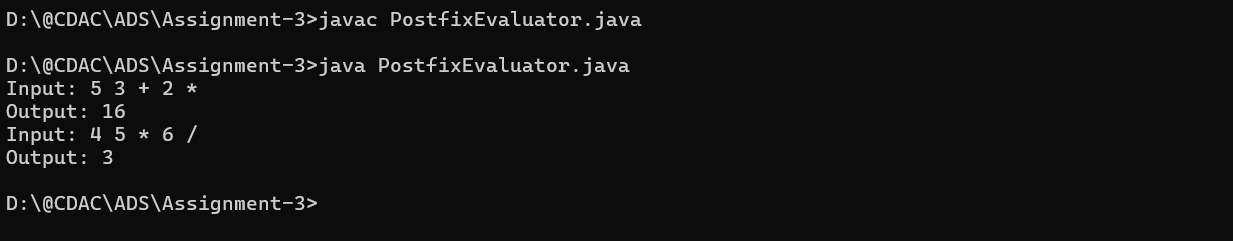
**4. Evaluate a postfix expression using a stack.**

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



**Explanation :**

1. **Start Program**:
   * The program begins execution by defining the evaluatePostfix method.
2. **Define Method & Initialize Stack**:
   * evaluatePostfix takes a String parameter expression and initializes an empty stack.
3. **Split Expression into Tokens**:
   * The expression is split into tokens based on spaces, producing an array of tokens to process individually.
4. **Loop Through Tokens**:
   * For each token in the tokens array:
     + If the token is a number, it is pushed onto the stack.
     + If the token is an operator, two numbers (a and b) are popped from the stack.
     + The operator is applied to a and b, and the result is pushed back onto the stack.
5. **End of Tokens**:
   * Once all tokens have been processed, the final result is popped from the stack.
6. **Return Result**:
   * The final result is returned as the output of the evaluatePostfix method.
7. **End Program**:
   * The program finishes, and the result of the postfix evaluation is displayed.

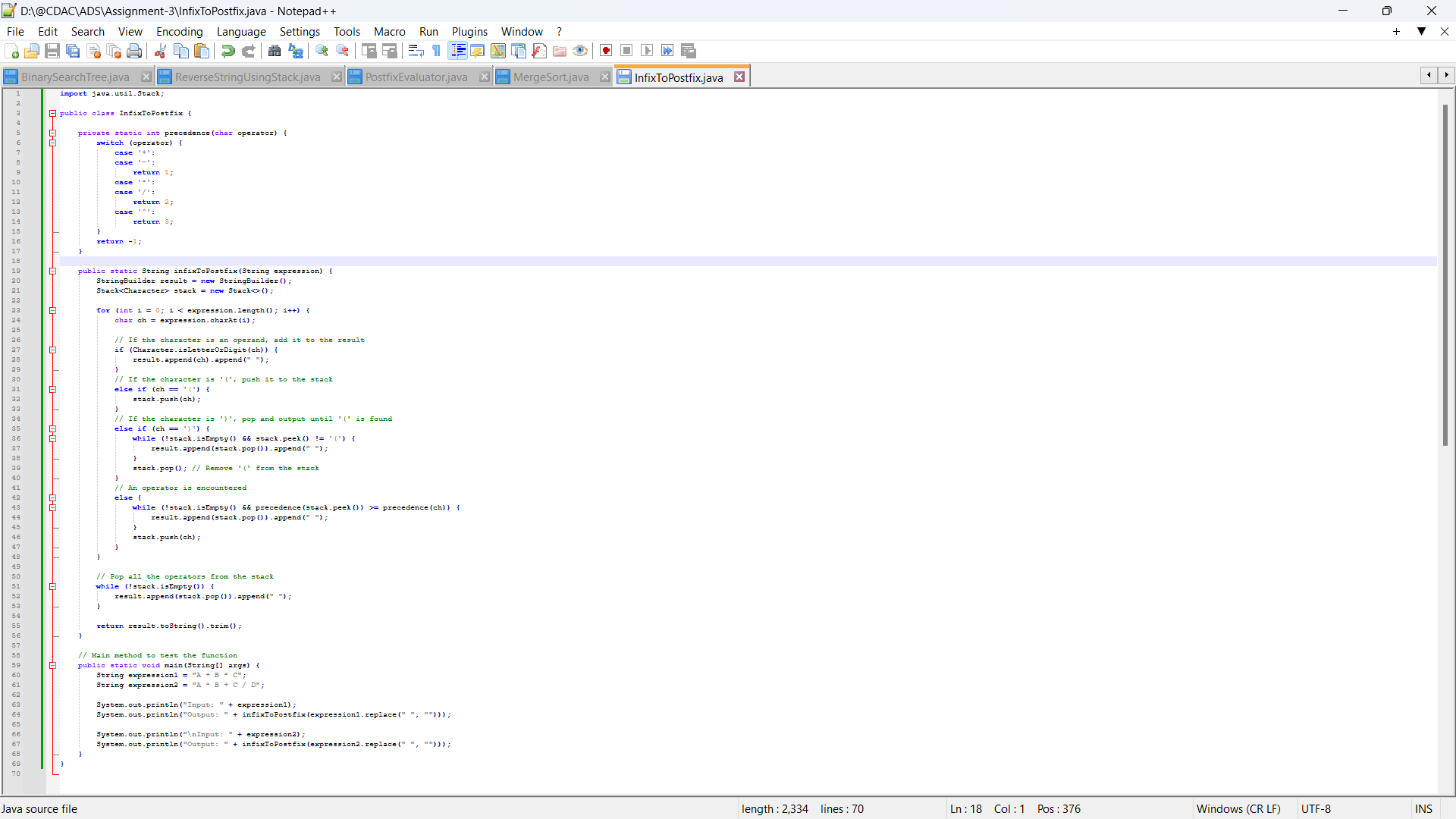


 **Time Complexity**: O(n)

 **Space Complexity**: O(n)

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

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



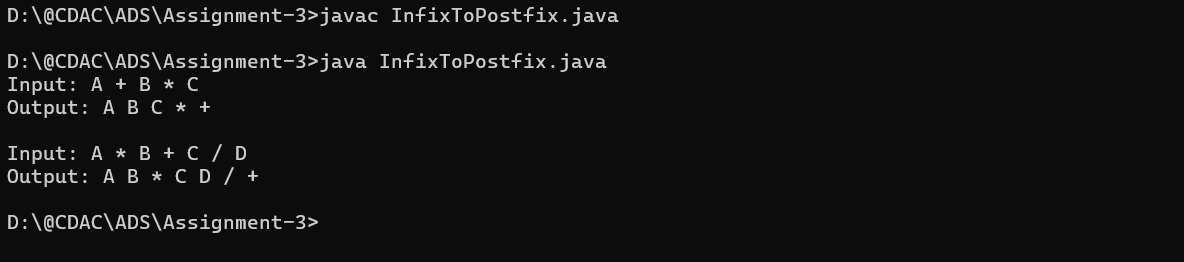
 **Initialize Result and Stack**: Start by initializing the result as an empty string and the stack as empty.

 **Loop through Each Character in the Expression**:

* **If Character is an Operand**: Directly add it to the result.
* **If Character is '('**: Push it to the stack.
* **If Character is ')'**: Pop all operators from the stack until '(' is encountered, then remove '('.
* **If Character is an Operator**: Pop operators from the stack to the result as long as the stack isn’t empty and the precedence of the operator at the top is greater than or equal to the current character's precedence. Then, push the current operator onto the stack.

 **After Loop Completes**: Pop any remaining operators in the stack and add them to the result.

 **Return the Final Postfix Expression**.

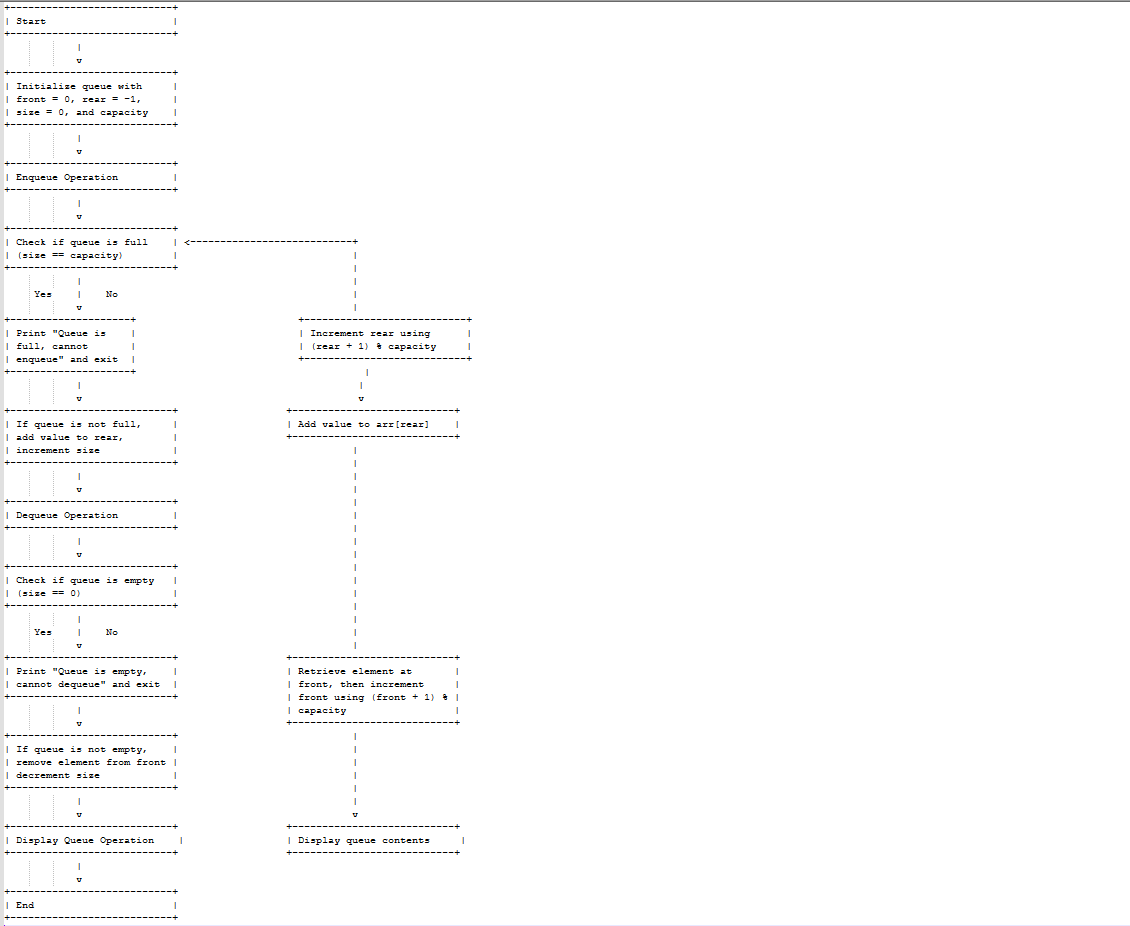
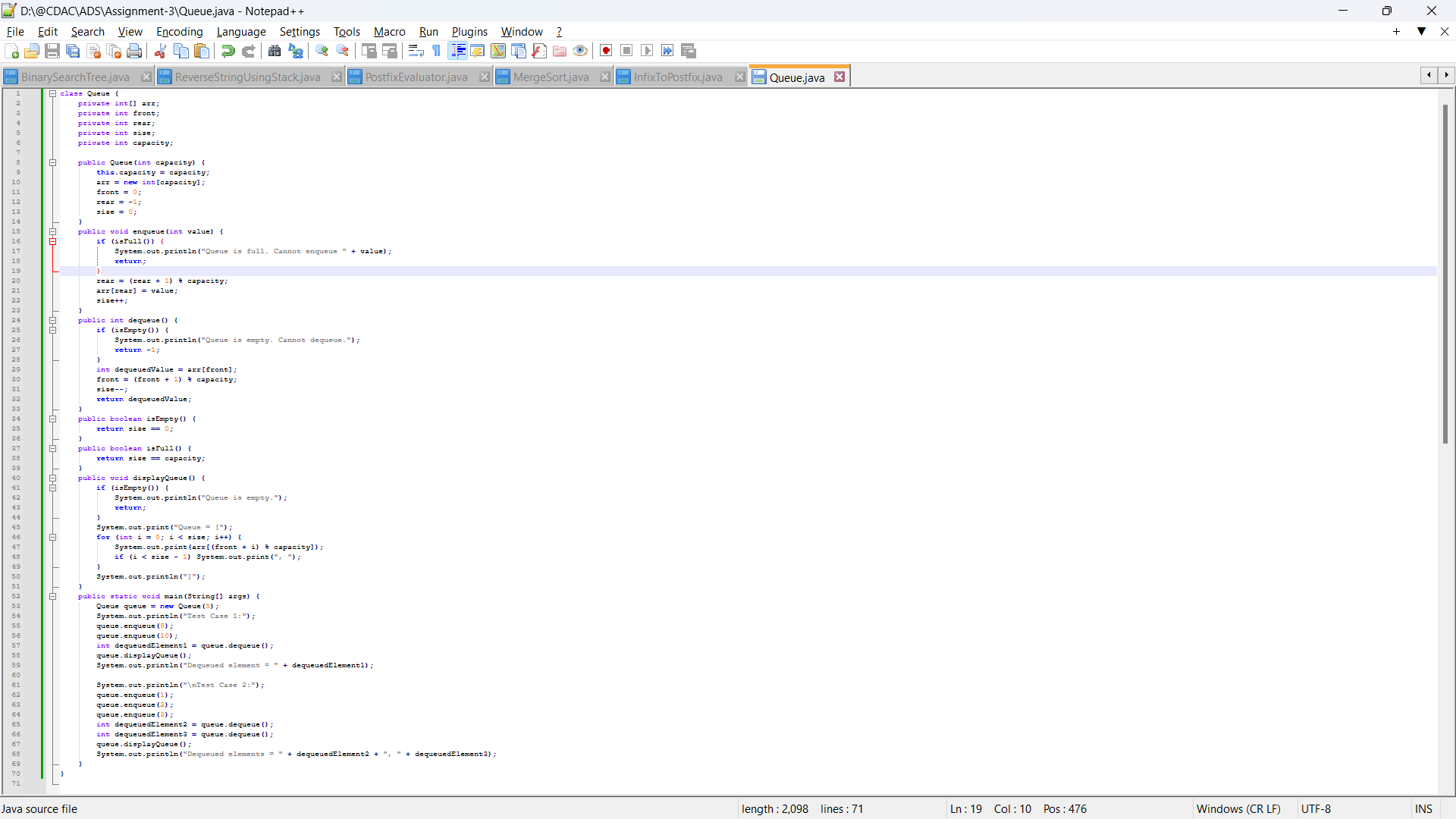


Time Complexity : **O(n).**

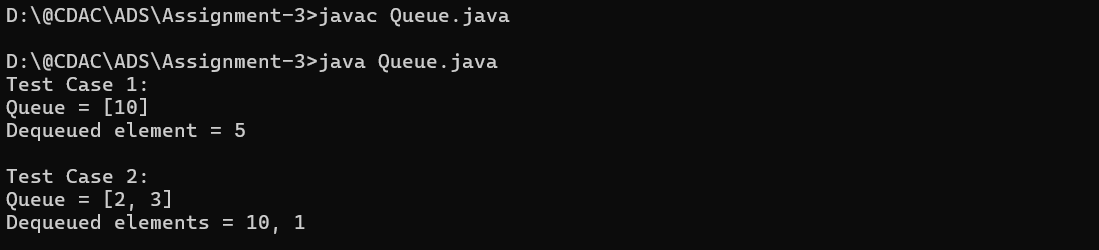
Space Complexity : **O(n).**

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

* **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. **Start and Initialize Queue**: Set the initial values of front, rear, size, and capacity.
2. **Enqueue Operation**:
   * **Check Full Condition**: If size == capacity, print a message that the queue is full.
   * **Insert Element**: If the queue is not full, update rear to point to the next position (using (rear + 1) % capacity) and add the element at this position. Increment the size.
3. **Dequeue Operation**:
   * **Check Empty Condition**: If size == 0, print a message that the queue is empty.
   * **Remove Element**: If the queue is not empty, retrieve and remove the front element, update front using (front + 1) % capacity, and decrement size.
4. **Display Queue Operation**: Traverse the queue from front to rear and display its contents.
5. **End**

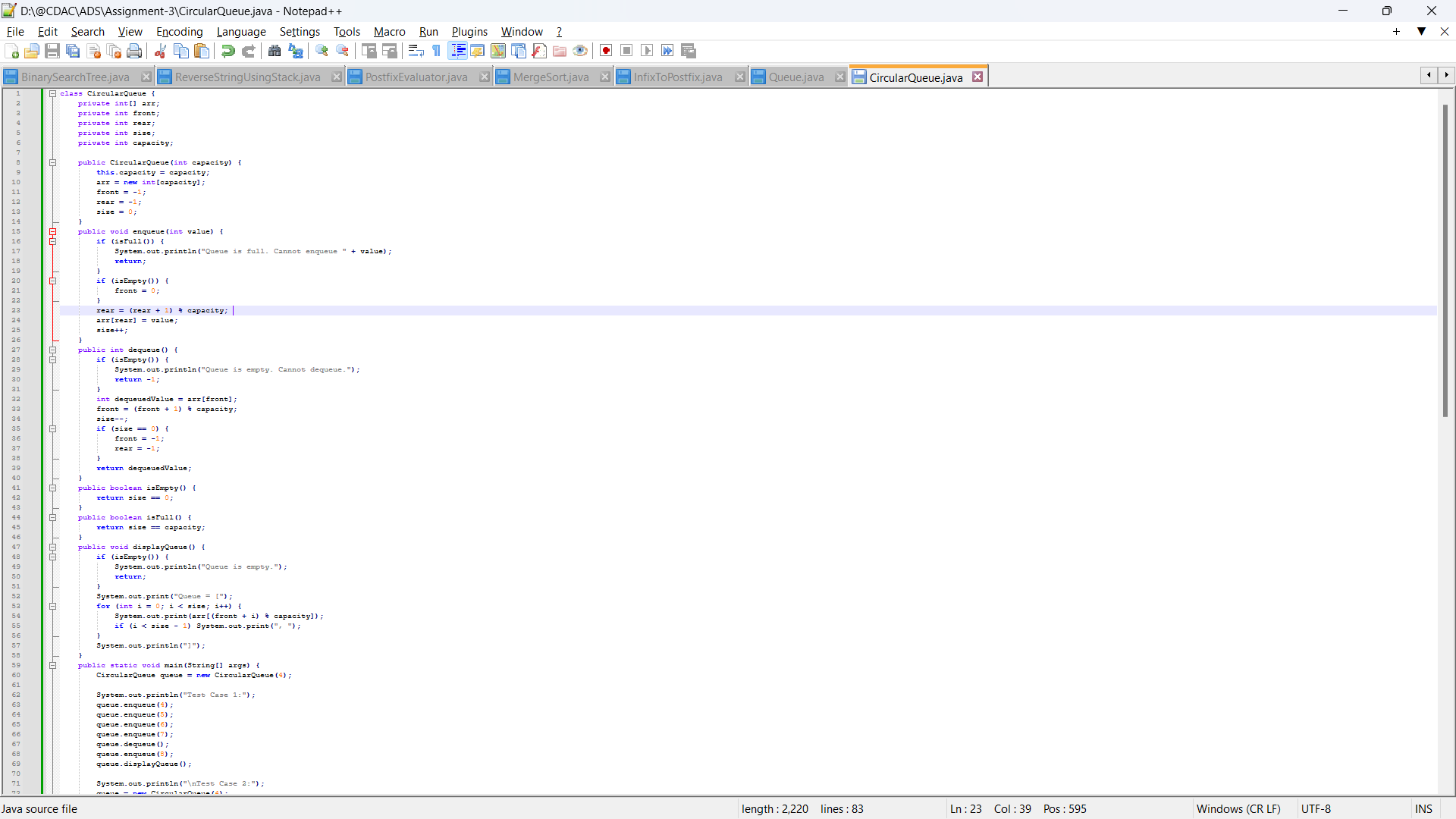


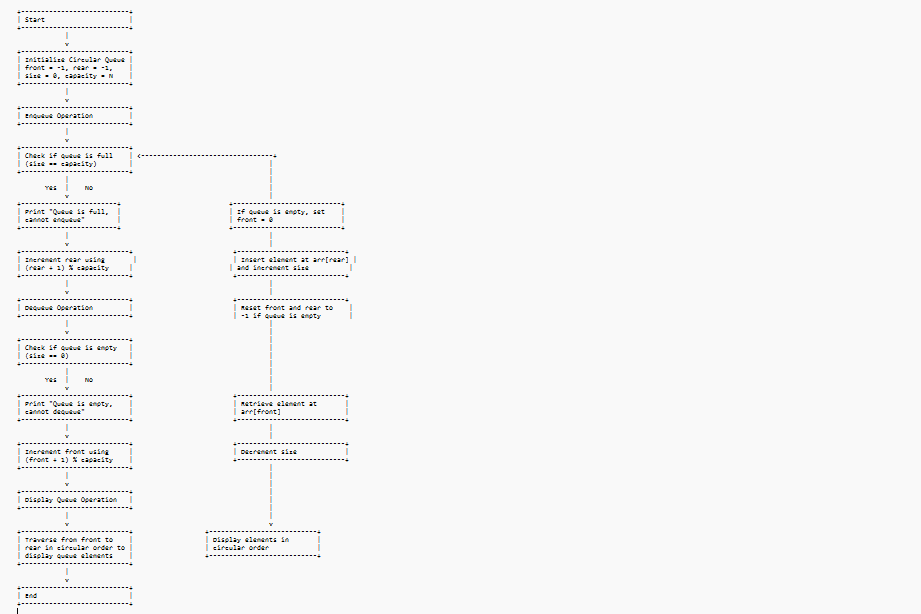
The time complexity **O(1)**

The space complexity **O(n)**

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

* **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]





 **Initialization**: Start by initializing the circular queue with front = -1, rear = -1, size = 0, and capacity = N.

 **Enqueue Operation**:

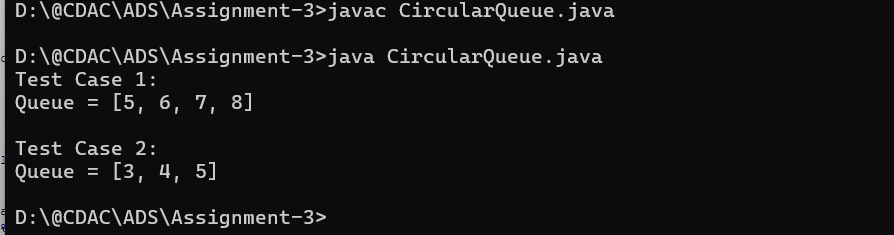
* **Check Full Condition**: If size == capacity, print "Queue is full, cannot enqueue" and exit the enqueue operation.
* **Insert Element**: If the queue is empty, set front = 0 (for the first element). Then, update rear using circular increment (rear + 1) % capacity, insert the element at arr[rear], and increase size.

 **Dequeue Operation**:

* **Check Empty Condition**: If size == 0, print "Queue is empty, cannot dequeue" and exit the dequeue operation.
* **Remove Element**: Retrieve the element at arr[front], increment front using (front + 1) % capacity, and decrease size. If the queue becomes empty (i.e., size == 0), reset front and rear to -1.

 **Display Queue Operation**: Traverse the queue from front to rear in circular order and print each element.

 **End**

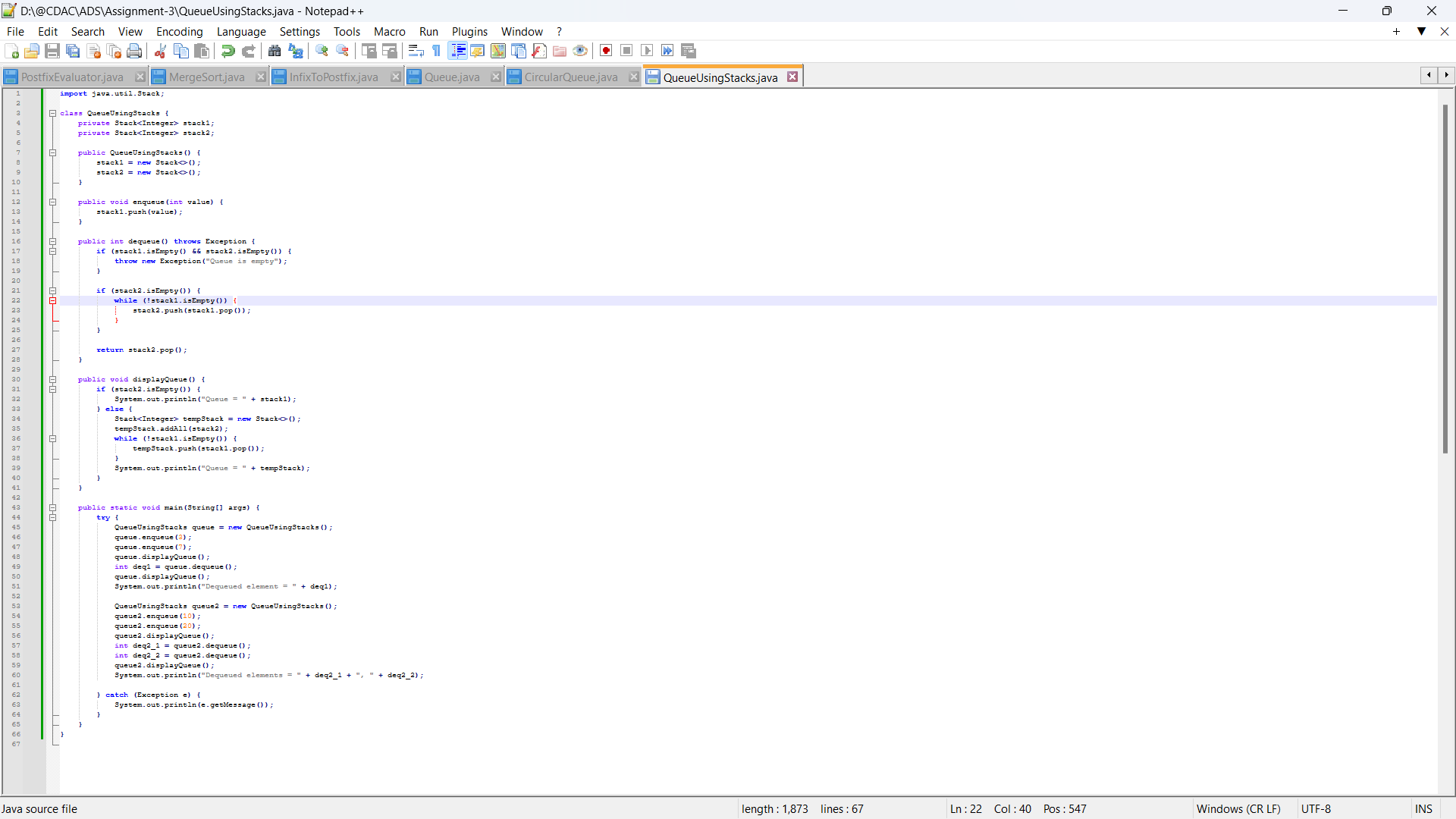


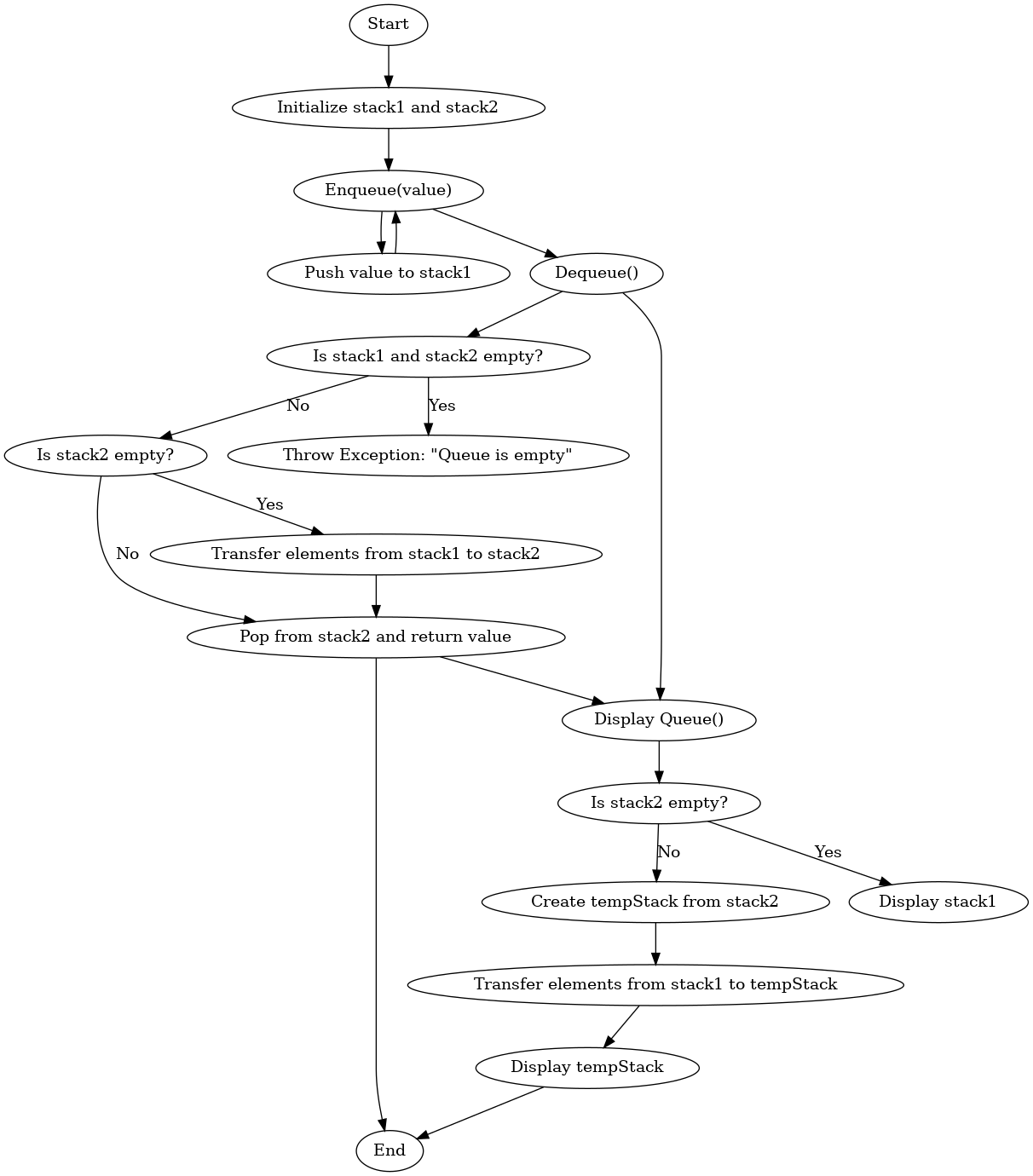
The time complexity O**(1)**

The space complexity **O(n)**

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

* **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. Start:**

* The program begins execution.

**2. Initialize stack1 and stack2:**

* Two stacks (stack1 and stack2) are initialized when the QueueUsingStacks object is created.

**3. Enqueue Operation:**

* **Input**: The enqueue(value) method is called.
* **Push value to stack1**: The input value is added (pushed) to stack1. This simulates adding an element to the queue.

**4. Dequeue Operation:**

* **Input**: The dequeue() method is called to remove and return the front element from the queue.
* **Check if both stacks are empty**:
  + If both stack1 and stack2 are empty, this means the queue is empty, so an exception is thrown ("Queue is empty").
* **Check if stack2 is empty**:
  + If stack2 is not empty, it means there are elements ready to be dequeued, so we skip the next step.
  + If stack2 is empty, we **transfer all elements from stack1 to stack2**. This reversal of stack1 into stack2 ensures the elements are in the correct order for dequeueing.
* **Pop from stack2**: The element at the top of stack2 (which is the front element of the queue) is popped and returned as the result of the dequeue operation.

**5. Display Queue:**

* **Input**: The displayQueue() method is called to print the current state of the queue.
* **Check if stack2 is empty**:
  + If stack2 is empty, the current queue is represented by stack1.
  + If stack2 is not empty:
    - A temporary stack (tempStack) is created by copying the elements of stack2.
    - **Transfer elements from stack1 to tempStack**: All remaining elements in stack1 are pushed to the temporary stack in reverse order.
    - The temporary stack is displayed to represent the current state of the queue.

**6. End:**

* After the required operations (enqueue, dequeue, or display) are complete, the program execution ends.

 **Time Complexity**:

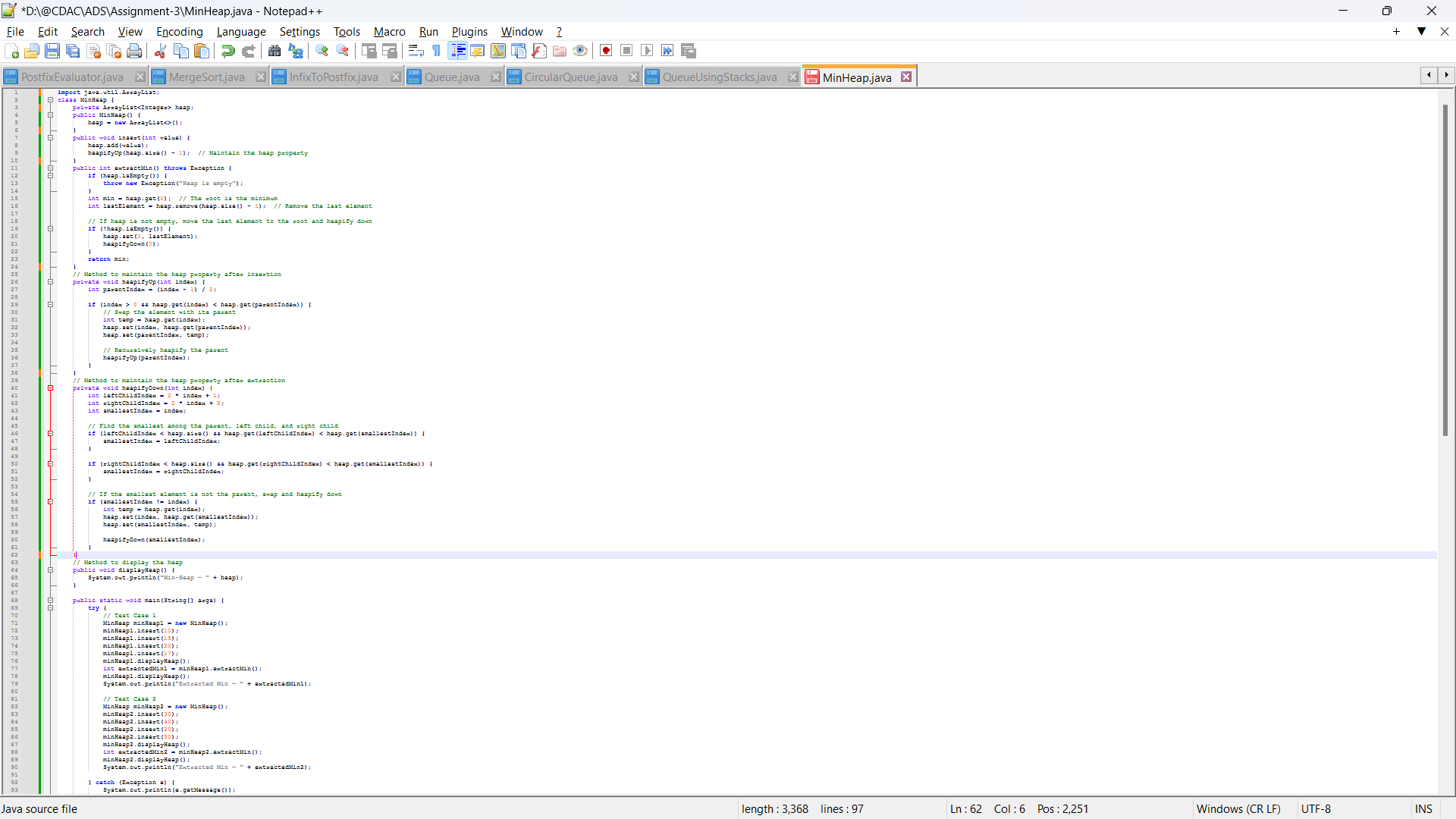
* Enqueue: O(1)
* Dequeue : O(1)

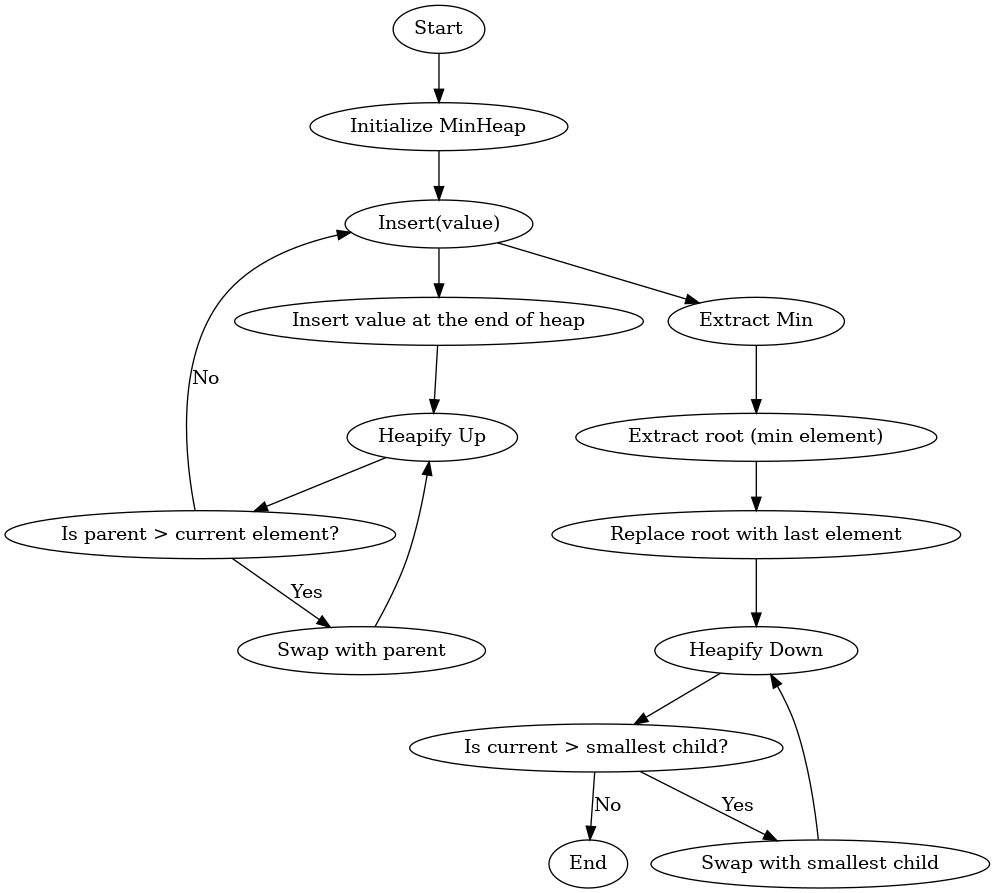
 **Space Complexity**: O(n)



**9. Implement a Min-Heap.**

* **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. MinHeap Class:**

The class encapsulates the heap operations and stores the heap elements in an ArrayList<Integer>, representing the binary heap structure.

**Fields:**

* heap: A dynamic list to hold the elements of the heap.

**Constructor:**

* MinHeap(): Initializes the heap as an empty ArrayList.

**2. Insert Operation (insert(int value)):**

This operation adds an element to the heap while maintaining the heap property.

1. **Insert the element**: The element is added to the end of the heap array.
2. **Heapify Up**: The heapifyUp() method is called to ensure the Min-Heap property is maintained. It compares the inserted element with its parent. If the parent is larger, it swaps them. This process continues until the correct position is found for the inserted element.

**Heapify Up (heapifyUp(int index)):**

This function ensures that the element at index is smaller than its parent and restores the heap property by repeatedly swapping the current element with its parent until the element is smaller than its parent or it reaches the root.

* **If parent > current element**: Swap the element with its parent.
* **Recursion**: Repeat the process for the parent.

**3. Extract Min Operation (extractMin()):**

This operation removes and returns the minimum element (the root of the heap).

1. **Check if the heap is empty**: If it is, throw an exception.
2. **Extract the minimum (root)**: The minimum element is always at index 0 (the root).
3. **Replace the root with the last element**: The last element in the heap is moved to the root.
4. **Heapify Down**: The heapifyDown() method is called to restore the heap property by moving the element downwards if it is larger than its children.

**Heapify Down (heapifyDown(int index)):**

This function ensures that the element at index is smaller than both its children. If it’s larger than either child, it swaps with the smaller child and continues the process until the heap property is restored.

* **Find the smallest child**: Compare the left and right children, and select the smaller one.
* **If the current element > smallest child**: Swap with the smallest child.
* **Recursion**: Repeat the process for the new position.

**4. Display Heap (displayHeap()):**

This method simply prints the current state of the heap array.

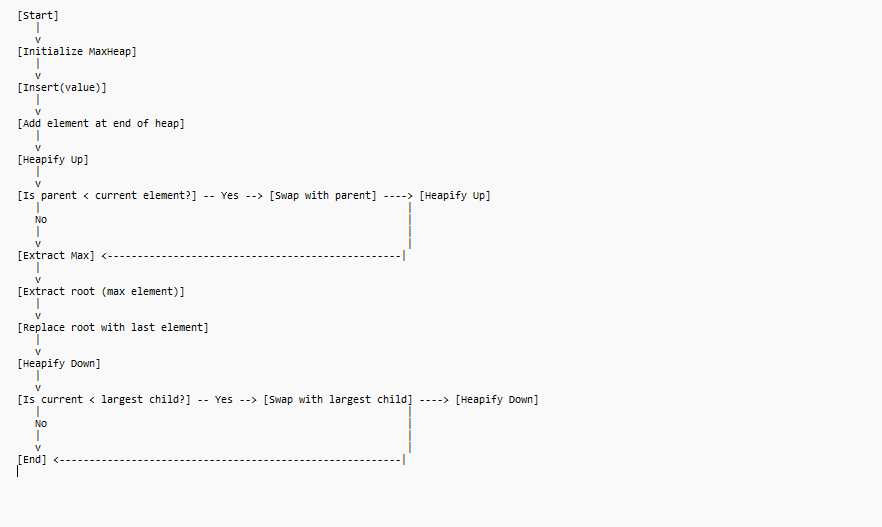
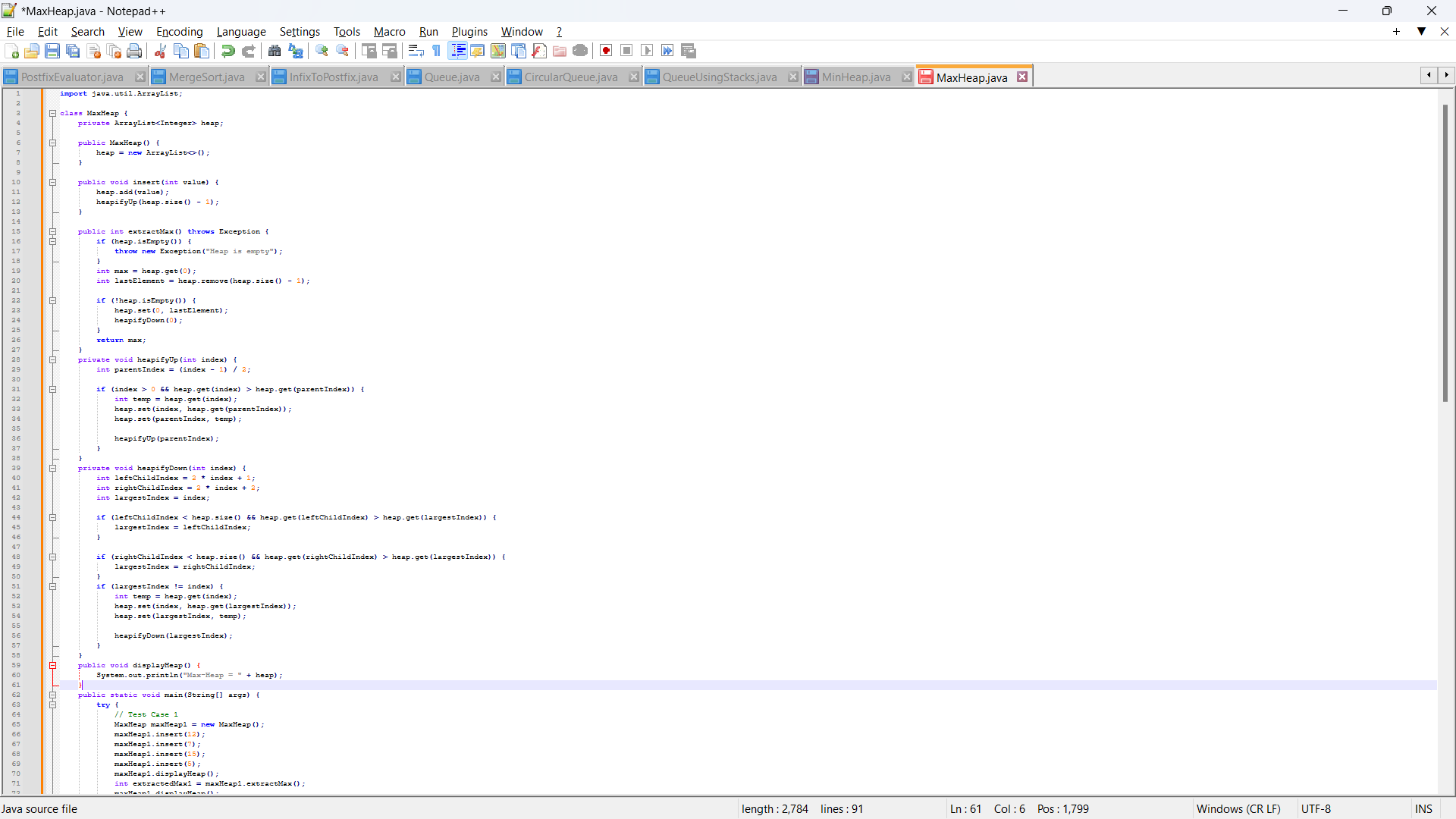
 **Time Complexity**:

* **Insert**: O(log⁡n)
* **Extract Min**: O(log⁡n)
* **Display**: O(n)

 **Space Complexity**: O(n)

**10. Implement a Max-Heap.**

* **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



 **Start**: The program begins execution.

 **Initialize MaxHeap**: Create an instance of the MaxHeap class.

 **Insert(value)**: Call the insert method with the specified value.

* **Add element at end of heap**: The new value is added to the end of the array that represents the heap.
* **Heapify Up**: This process maintains the max-heap property.
  + **Is parent < current element?**: Check if the parent is smaller than the current element.
    - **Yes**: If true, swap with the parent and continue heapifying up.
    - **No**: If false, finish the process.

 **Extract Max**: Call the extract max method.

* **Extract root (max element)**: The maximum element (root) is removed.
* **Replace root with last element**: The last element is moved to the root.
* **Heapify Down**: This process restores the max-heap property.
  + **Is current < largest child?**: Check if the current element is smaller than the largest child.
    - **Yes**: If true, swap with the largest child and continue heapifying down.
    - **No**: If false, finish the process.

 **End**: The program execution ends



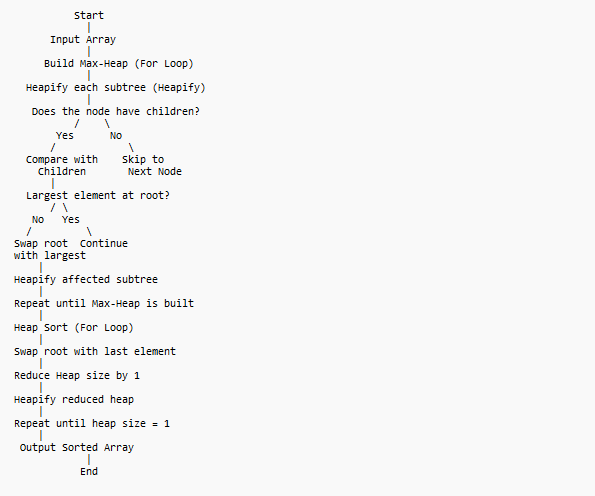
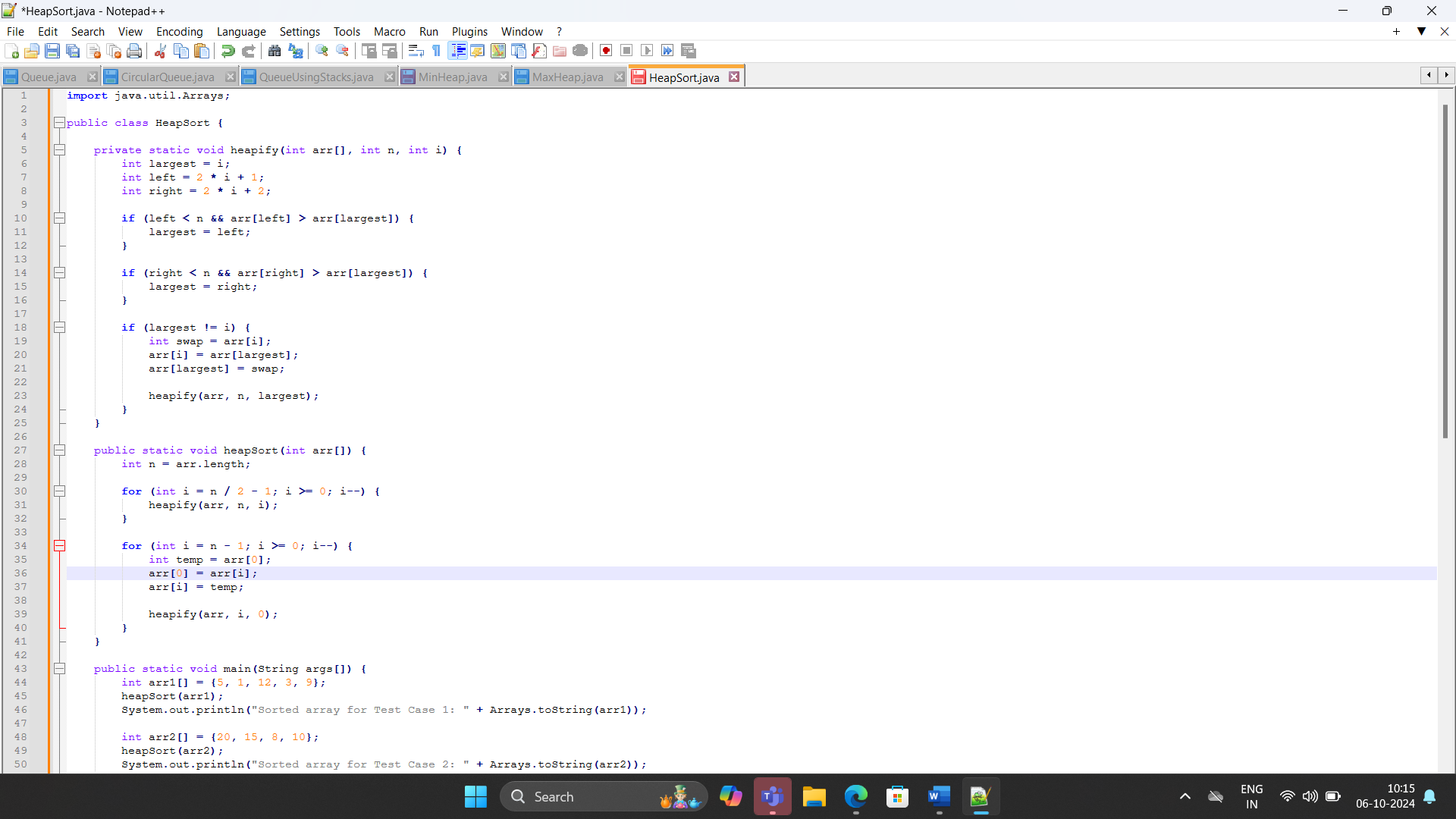
 **Time Complexity**:

* **Insert**: O(log⁡n)
* **Extract Max**: O(log⁡n)
* **Display**: O(n)

 **Space Complexity**: O(n)

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

* **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]



 **Start**: Begin the program.

 **Input Array**: Receive an input array.

 **Build Max-Heap**:

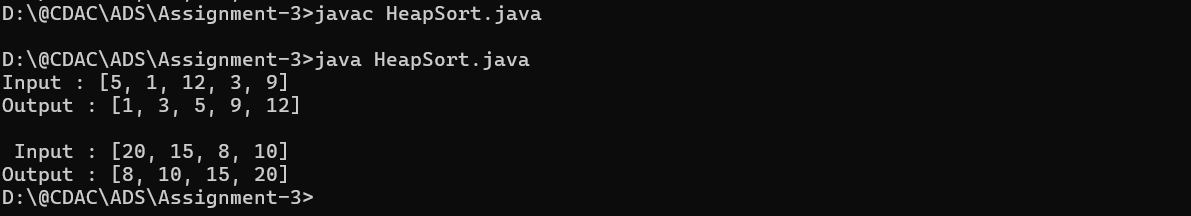
* Loop through the non-leaf nodes of the array (from the middle to the root) and call heapify() on each node.
* Each node is compared with its children to ensure the max-heap property is maintained.

 **Heap Sort (Sorting Process)**:

* Swap the root element (largest in the max-heap) with the last element.
* Reduce the heap size.
* Call heapify() on the reduced heap to restore the max-heap property.
* Repeat the process until the heap size is reduced to 1.

 **Output Sorted Array**: Print the sorted array.

 **End**: Finish the program.



 **Time Complexity:** O(n log n) for all cases (worst, average, best).

 **Space Complexity:** O(1) auxiliary space (in-place sorting).

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

* **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

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

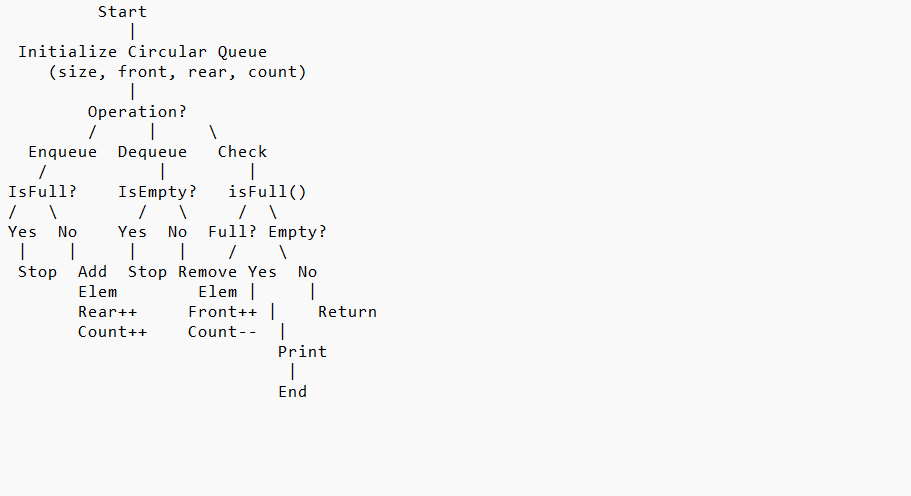
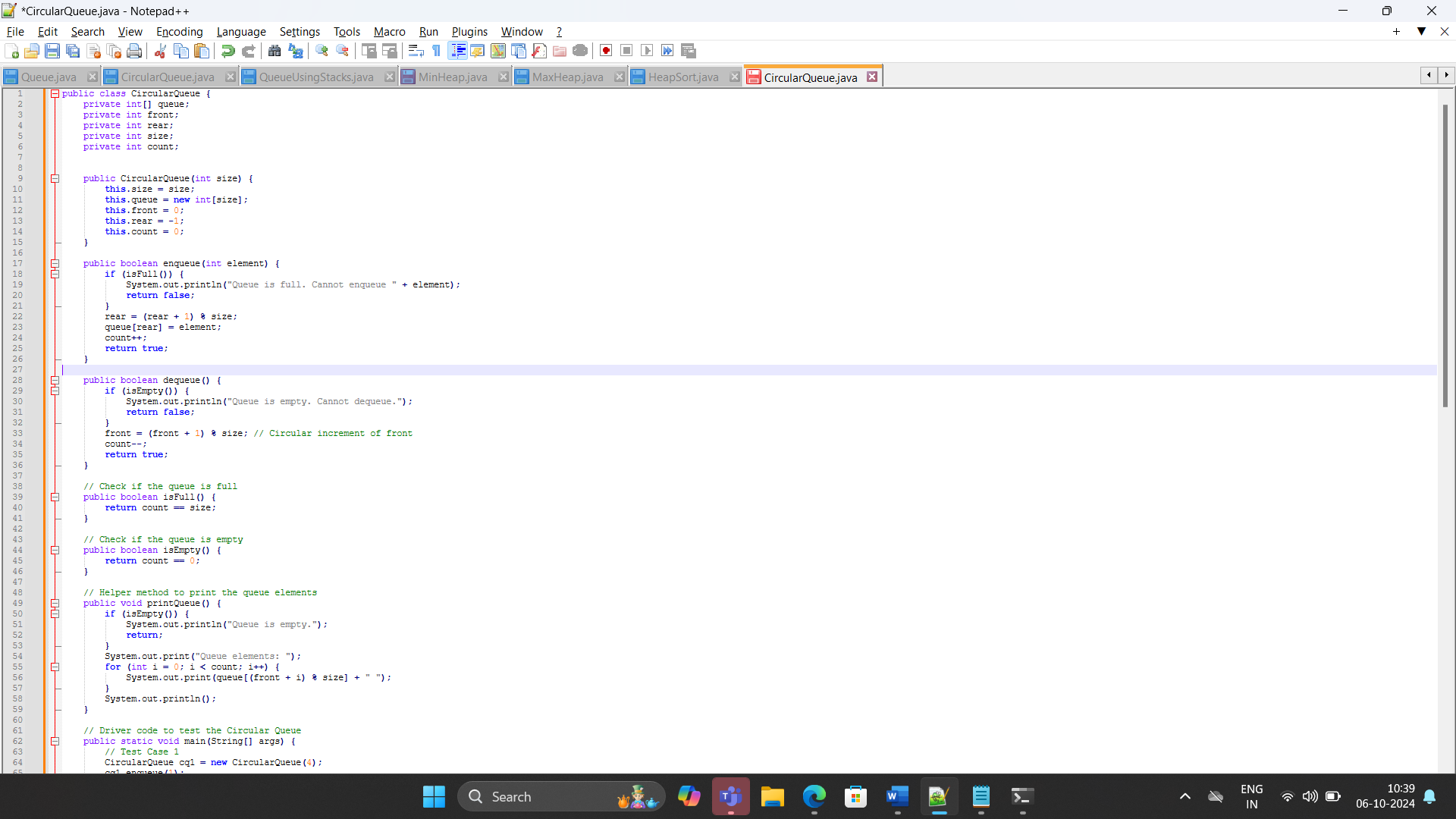
* **Test Case 1**:  
  Input: Enqueue with priorities: 3 (priority 1), 10 (priority 3), 5 (priority 2), Dequeue  
  Output: Dequeued element = 10 (highest priority), Priority Queue = [5, 3]
* **Test Case 2**:  
  Input: Enqueue with priorities: 7 (priority 4), 8 (priority 2), 6 (priority 3), Dequeue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

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

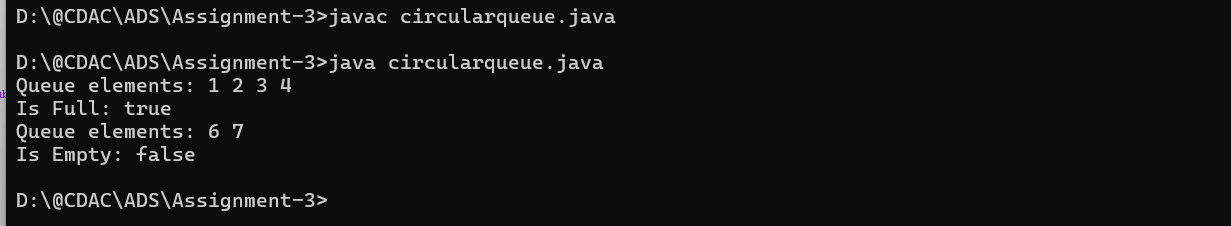
* **Test Case 1**:  
  Input: Push 5, Push 3, Push 7, Get Min  
  Output: Min = 3
* **Test Case 2**:  
  Input: Push 10, Push 8, Push 6, Push 12, Get Min  
  Output: Min = 6

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

* **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



1. **Start**: Initialize the program.
2. **Initialize Circular Queue**: Set up the queue with a fixed size, and initialize front, rear, and count.
3. **Enqueue Operation**:
   * Check if the queue is full:
     + **Yes**: Print "Queue is full" and stop the operation.
     + **No**: Insert the element at the rear index, increment rear circularly, and update the count.
4. **Dequeue Operation**:
   * Check if the queue is empty:
     + **Yes**: Print "Queue is empty" and stop the operation.
     + **No**: Remove the element at the front index, increment front circularly, and decrement the count.
5. **Check isFull**: Compare the number of elements (count) with the size.
6. **Check isEmpty**: If count is 0, return true for empty.
7. **Print Queue**: Display the elements from the front index, handling the circular wrap-around.



 **Enqueue, Dequeue, isFull, isEmpty**: O(1)

 **Print Queue**: O(n)

 **Space Complexity**: O(n)