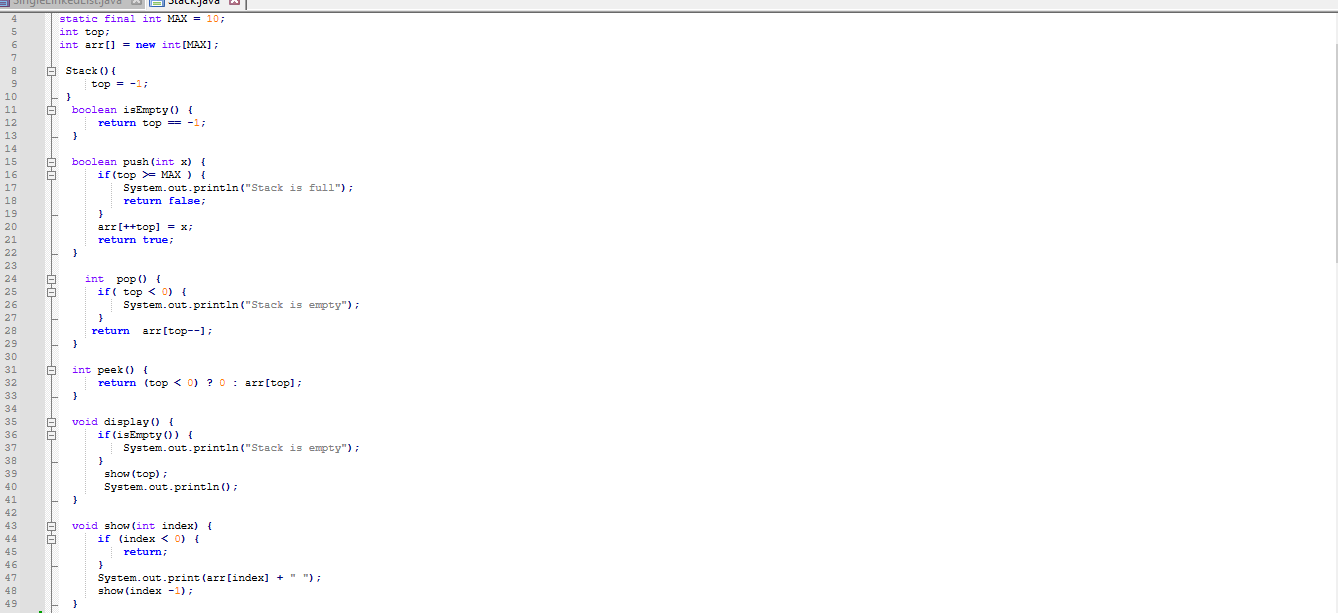
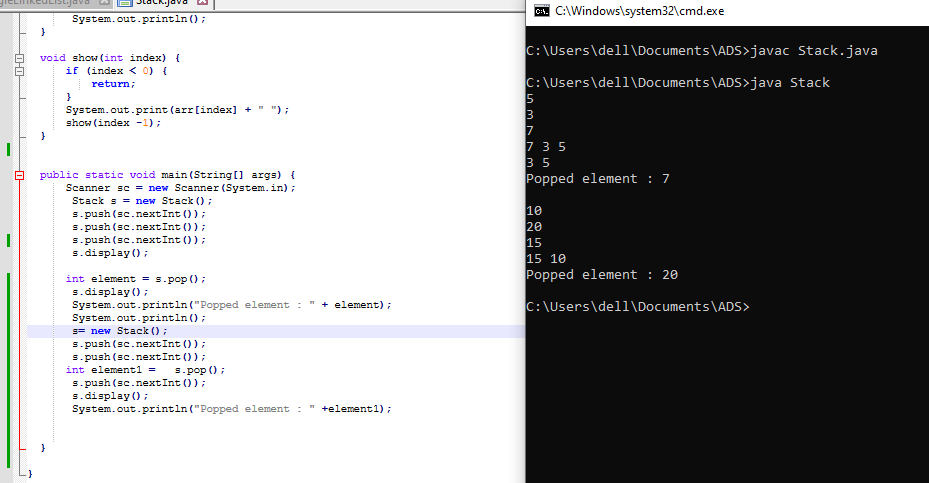
**Assignment – 3**

**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



****

**Flowchart-**

**Start**

|

**Initialize Stack**

**|**

**Check if Stack is Full?**

**|**

**|**

**Yes No**

**| |**

**Display "Stack Push Element**

**is Full"**

**|**

**Check if Stack is Empty?**

**|**

**Yes No**

**| |**

**Display "Stack Pop Element**

**is Empty”**

**| |**

**Display Elements Display Elements**

**|**

**End**

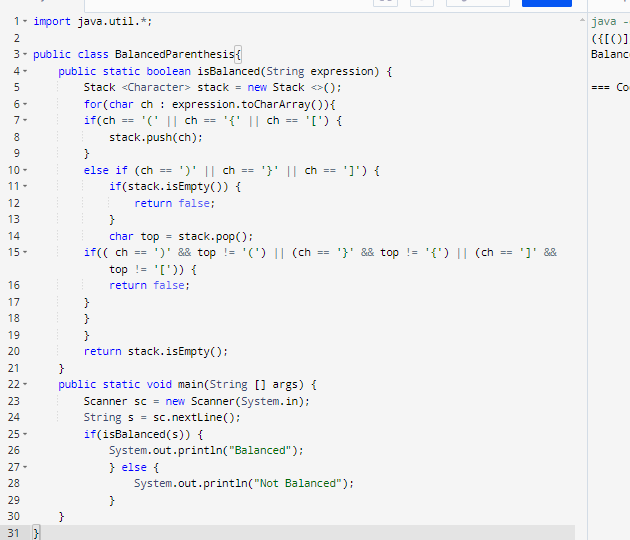
**Explanation –** Firstly I’ve created the Stack class. The stack has a maximum size defined by the constant MAX, set to 10. The class includes methods for stack operations like isEmpty() which checks if the stack is empty, push(int x) which adds an element to the top of the stack, and pop() removes and returns the top element. If the stack is full during a push operation, a message is displayed that indicates that the stack is full. If a pop operation is tried on an empty stack, a message indicates that the stack is empty. The display() method prints the current elements of the stack. In the main method I’ve taken the input from user using Scanner class and invoked the stack operations.

**Time Complexity:** Push : O(1), Pop: O(1), Peek: O(1), Display: O(n)

**Space Complexity:** O(n)

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

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





**Flowchart-**

**Start**

**|**

**Initialize an empty Stack**

**|**

**For each character in the expression**

**|**

**Is it an opening parenthesis?**

**|**

**Yes -> Push it onto the Stack**

**|**

**Is it a closing parenthesis?**

**Yes -> Is the Stack empty?**

**|**

**Yes -> Not Balanced (Stop)**

**|**

**No -> Pop from the Stack**

**|**

**Is it a matching pair?**

**|**

**No -> Not Balanced (Stop)**

**|**

**After checking all characters**

**|**

**Is the Stack empty?**

**|**

**Yes -> Balanced**

**|**

**No -> Not Balanced**

**|**

**End**

**Explanation -** The isBalanced method takes a string expression and iterates through each character. If the character is an opening parenthesis it is pushed onto the stack. If the character is a closing parenthesis the program first checks if the stack is empty. If the stack is empty the method returns false. If the stack isn't empty, the program pops the top element and checks if it matches the current closing parenthesis. If there's a mismatch, the method returns false. Once all characters have been processed, the method checks if the stack is empty. If it is, it means all parentheses were matched correctly, and the method returns true. If not, it returns false. In the main method, a Scanner is used to take input from the user, which is passed to the isBalanced method. The result is printed, indicating whether the input expression is balanced or not.

**Time Complexity:**  O(n)

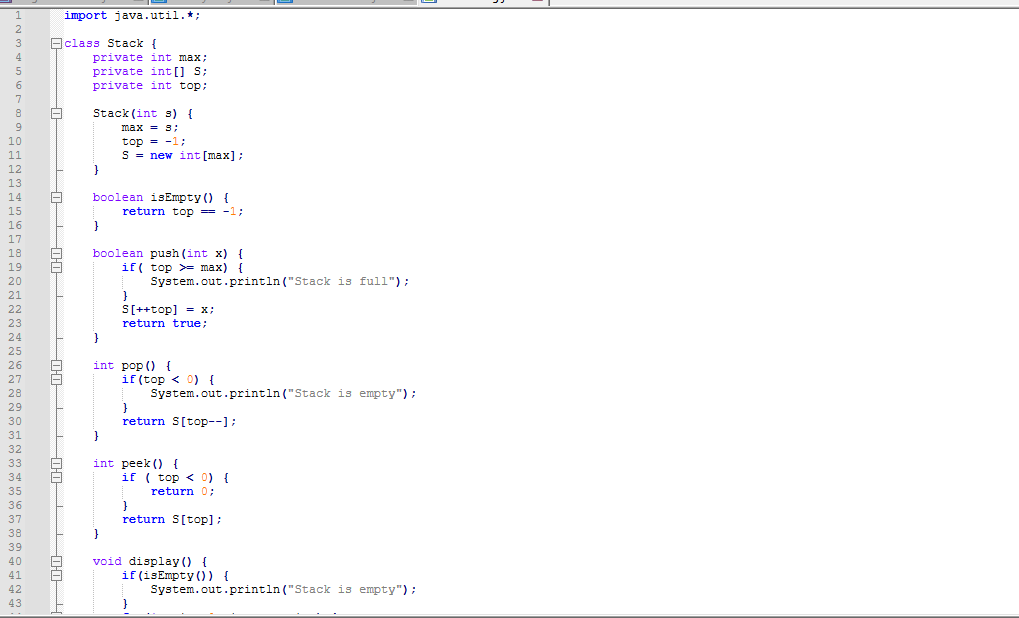
**Space Complexity:** O(n)

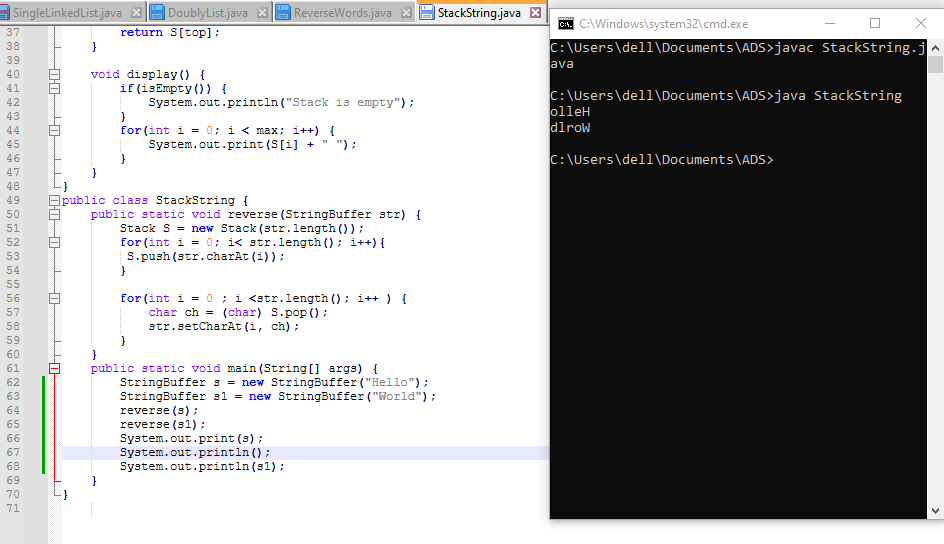
**3. Reverse a string using a stack.**

* **Test Case 1**:  
  Input: "hello"  
  Output: "olleh"

**Test Case 2**:  
 Input: "world"

Output: "dlrow"

* 



**Start**

|

**Initialize Stack S**

**with max capacity**

|

**For each character in the string**

**Push character onto Stack S**

|

**For each position in the string**

**Pop character from Stack S**

**Replace character at current position**

|

**Display Reversed**

**String**

|

**End**

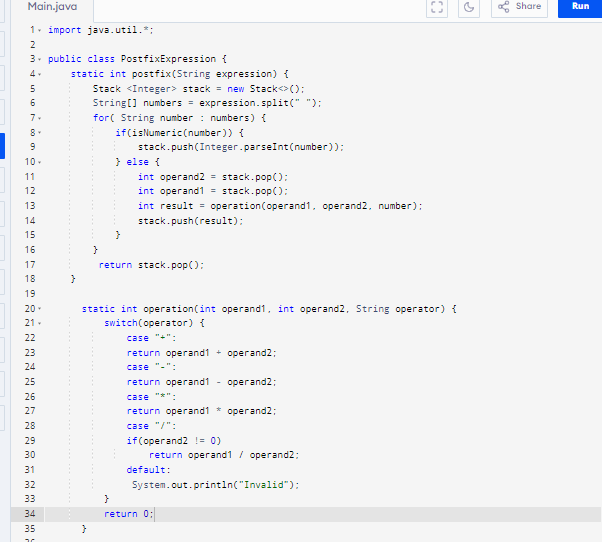
**Explanation-** A Stack object is created with a size equal to the length of the string. The program iterates through each character of the input string and pushes them into the stack one by one. In the next phase, the program iterates through the string positions again, this time popping characters from the stack and replacing the original string's characters in reverse order. Finally, the reversed string is displayed. In the main method I’ve created the object of StringBuffer class and invoked the reversed string.

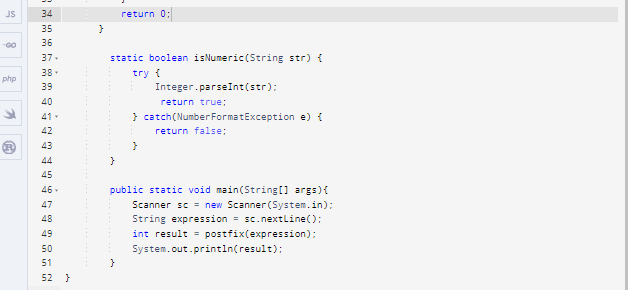
**Time Complexity:** O(n)

**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





**Flowchart –**

**Start**

**|**

**Read postfix expression**

**|**

**Initialize empty stack**

**|**

**Split expression into**

**elements by spaces**

**|**

**For each element in the**

**|expression, do the steps**

**|**

**Is the element numeric?**

**| |**

**Yes No**

**/ |**

**Push onto stack Pop two elements from stack**

**| |**

**Perform operation (operand1**

**operand2) based on operator**

**|**

**Push result back onto stack**

**|**

**Continue to next element**

**|**

**End loop when all elements are**

**processed**

**|**

**Pop and display the final result**

**|**

**End**

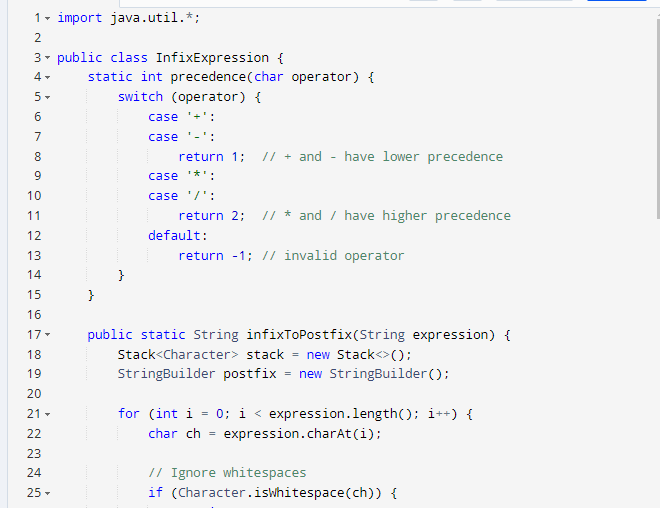
**Explanation –** In the postfix method, an empty integer stack is initialized, and the expression is split into individual elements based on spaces. For each element, if it's numeric (checked using the isNumeric method), it gets pushed onto the stack. If it's an operator two operands are popped from the stack, and the operation method performs the corresponding arithmetic operation on them. The result is pushed back onto the stack. After processing all elements, the stack's final value, representing the evaluated expression, is returned .

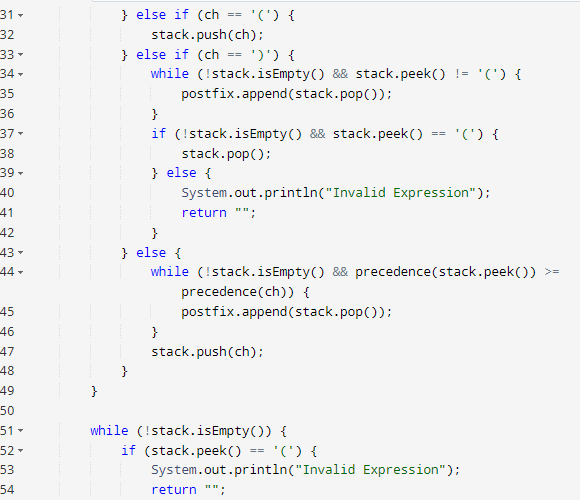
**Time Complexity:** O(n)

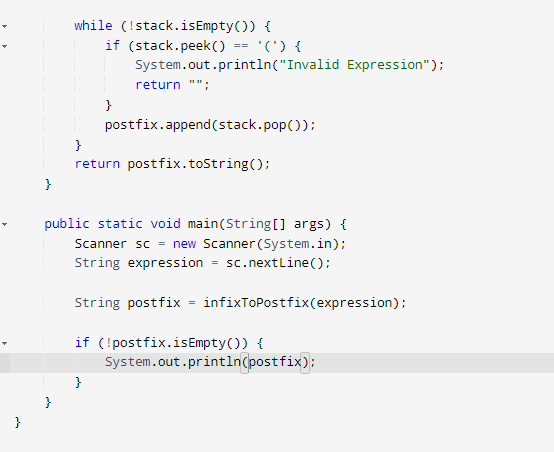
**Space Complexity:** O(n)

**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 / +"









**Flowchart –**

**Start**

**|**

**Input the infix expression**

**|**

**Initialize an empty stack and an empty postfix string**

**|**

**For each character in the infix expression:**

**|**

**Is the character an operand ?**

**| |**

**Append to postfix string**

**| |**

**Continue to the next character**

**|**

**Is the character '('?**

**| |**

**Push '(' onto the stack**

**| |**

**Continue to the next character**

**|**

**Is the character ')'?**

**| |**

**Pop from stack to postfix until '(' is found**

**| |**

**Remove '(' from stack**

**| |**

**Continue to the next character**

**|**

**Is the character an operator?**

**|**

**While stack is not empty and precedence of the top of the stack**

**is greater than or equal to the current operator's precedence:**

**|**

**Pop from stack to postfix**

**|**

**Push the current operator onto the stack**

**|**

**Continue to the next character**

**|**

**End of infix expression:**

**|**

**Pop all remaining operators from the stack to postfix**

**|**

**Output the postfix expression**

**|**

**End**

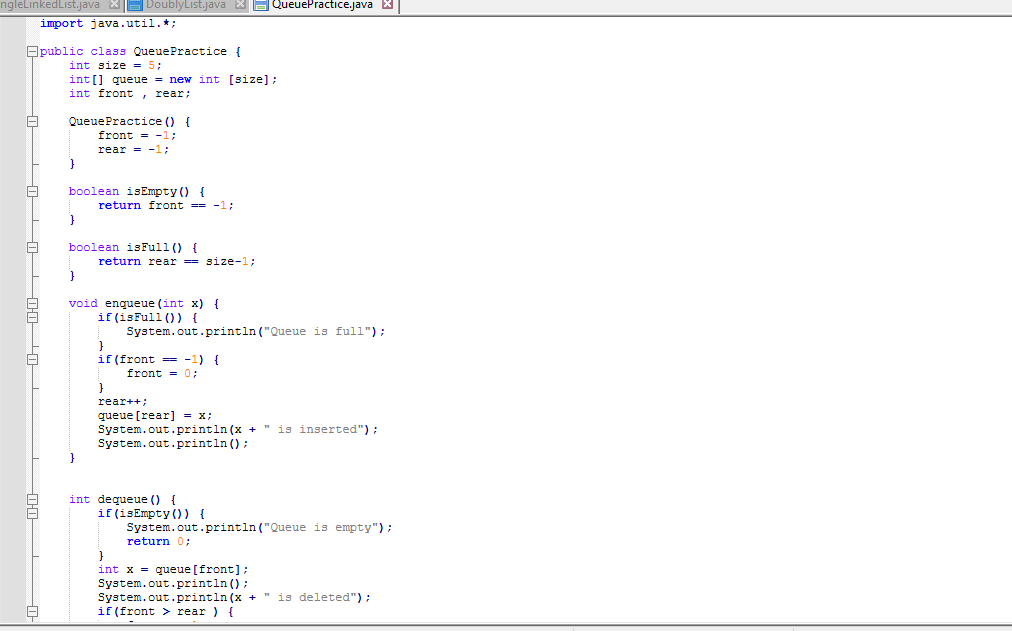
**Explanation -** First, an empty stack and an empty postfix string are initialized. If the character is an operand it is directly added to the postfix string. If it is an opening parenthesis it is pushed into the stack. When there is a closing parenthesis operators are popped from the stack and appended to the postfix string. For operators the algorithm checks the precedence of the operator at the top of the stack. As long as the stack's top operator has higher or equal precedence than the current one, operators are popped from the stack to the postfix string.. The current operator is then pushed onto the stack.Once all characters of the infix expression are processed, any remaining operators on the stack are popped and appended to the postfix string. This results in the final postfix expression.

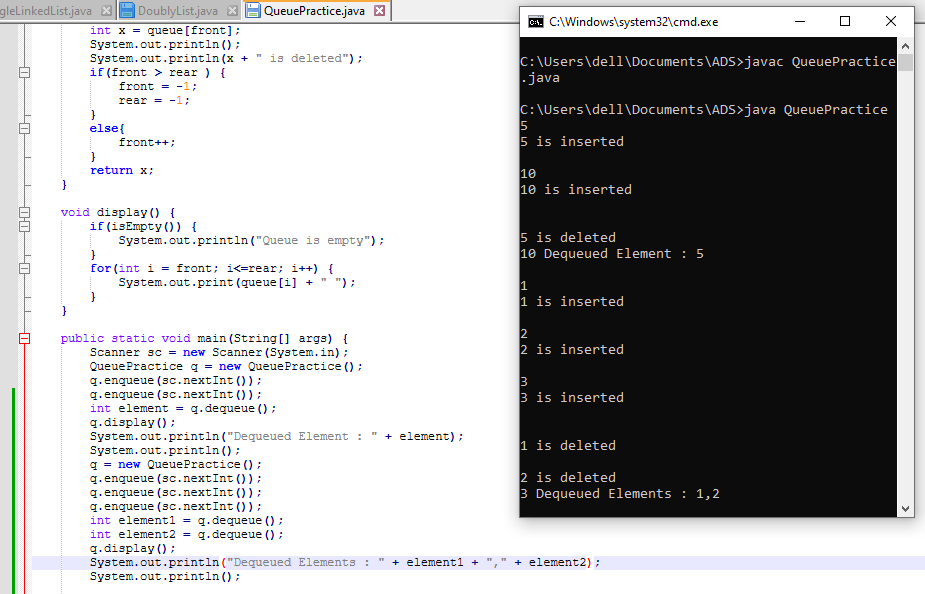
**Time Complexity:** O(n)

**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





**Flowchart -**

**START**

|

**Initialize size = 5, queue[size], front = -1, rear = -1**

|

**Main Menu**

**Enqueue**

**Dequeue**

**Display**

**Exit** |

**If option = 1 (Enqueue)**

|

**Check if (rear == size - 1)**

|

/ \

**Yes No**

| |

**Display Check if (front == -1)**

**"Queue is full"**

**Yes No**

| |

**Return Set Increament rear**

**front = 0 queue[rear] = x**

**Display "x is inserted"**

|

**Return to Main Menu**

**If option = 2 (Dequeue**)

|

**Check if (front == -1)**

|

/ \

**Yes No**

| |

**Display Set x = queue[front]**

**"Queue is empty" Display "x is deleted"**

**Check if (front > rear)**

/ \

**Yes No**

| |

**Return Set front=-1 Increment front**

**& rear=-1**

|

**Return to Main Menu**

**If option = 3 (Display)**

|

**Check if (front == -1)**

|

/ \

**Yes No**

| |

**Display Iterate from i = front to rear**

**"Queue Print queue[i]**

**is empty"**

|

**Return to Main Menu**

**If option = 4 (Exit)**

|

**Display "Exiting the program"**

|

**END**

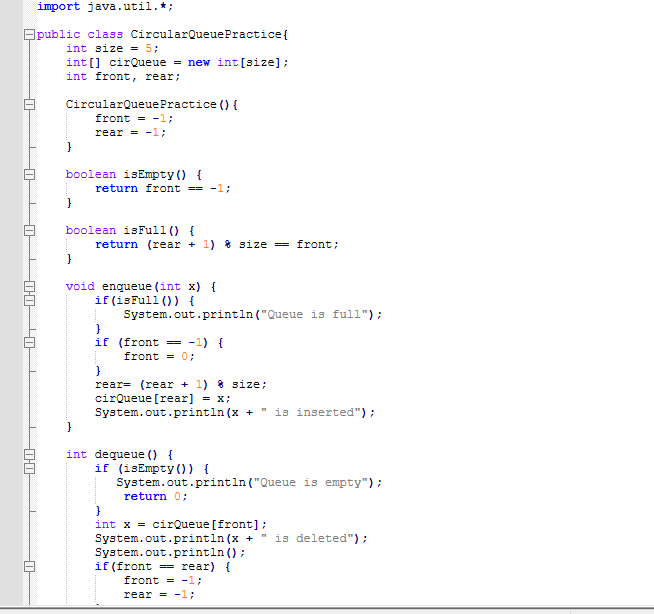
**Explanation –** In the Queue program I’ve initialized an array with a fixed size of 5. The front and rear variables track the front and rear positions of the queue, initialized to -1. In the enqueue method, when an element is added, the program first checks if the queue is full . If not, it sets front to 0 if it's the first element and then increments rear to add the element at the next position. The element is stored in the queue array, and a message confirms the insertion. The dequeue method checks if the queue is empty. If the queue is not empty, it retrieves and removes the element at the front position, displays the removed element, and increments front. If front exceeds rear after the removal, both front and rear are reset to -1 to indicate the queue is now empty again. The display method iterates from front to rear to show all the elements currently in the queue. The main method handles user input.

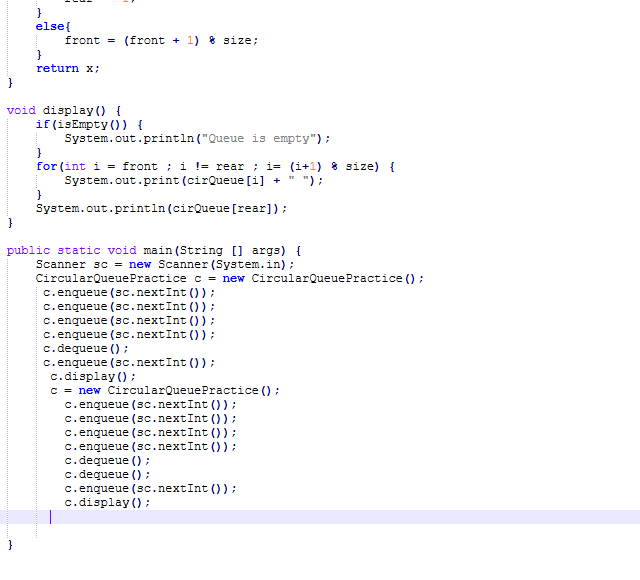
**Time Complexity:** Enqueue : 0(1) , Dequeue: O(1), display: O(n)

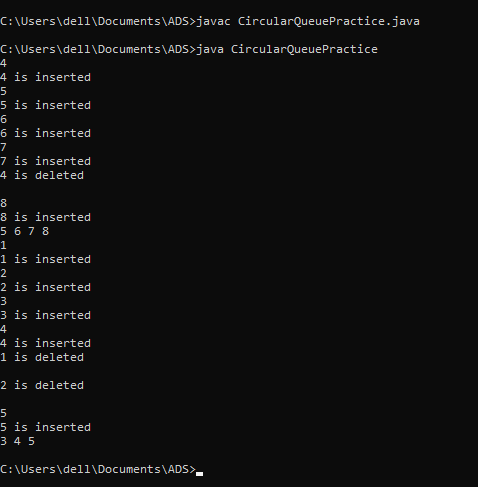
**Space Complexity:** O(n)

**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]

****





**Explanation -** In the Circular Queue program I’ve initialized an array with a fixed size of 5. The front and rear variables track the front and rear positions of the queue, initialized to -1. The isEmpty method checks whether the queue is empty by returning true. The isFull method checks if the queue is full The enqueue method inserts an element x into the queue. If the queue is full, it displays "Queue is full." If front is -1, it sets front to 0 to indicate the start of the queue. The dequeue method removes an element from the queue. If the queue is empty it displays "Queue is empty." The display method prints the elements from front to rear in a circular manner, iterating from front to rear.

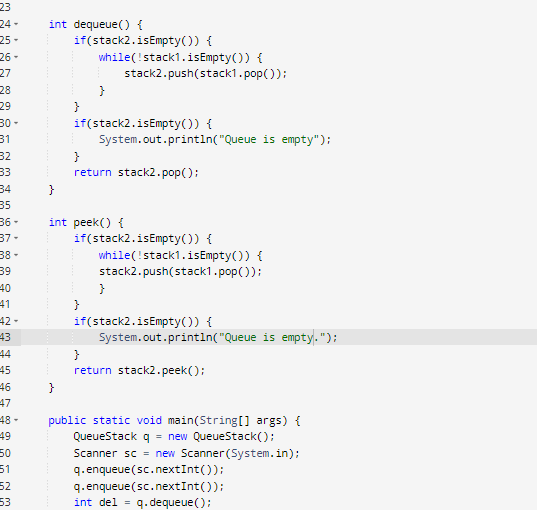
**Time Complexity:** Enqueue : 0(1) , Dequeue: O(1), display: O(n)

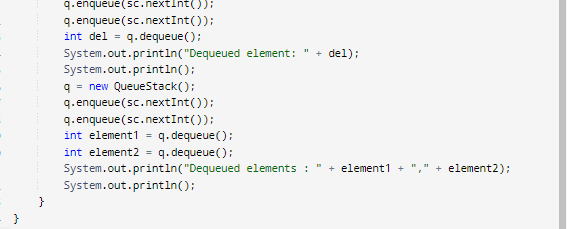
**Space Complexity:** O(n)

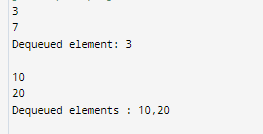
**8. Implement a Queue using two Stacks.**

* **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









**Flowchart –**

**Start**

**|**

**Initialize stack1**

**and stack2 as empty**

**|**

**Enqueue(value)**

**|**

**Push value**

**into stack1**

**|**

**Dequeue()**

**|**

**Is stack2 empty?**

**/ \**

**Yes No**

**/ \**

**Pop elements Pop from stack2**

**from stack1**

**and push to**

**stack2 until**

**stack1 is empty**

**|**

**Return popped**

**element from**

**stack2**

**|**

**Pop from**

**stack2**

**|**

**Return dequeued element**

**|**

**End**

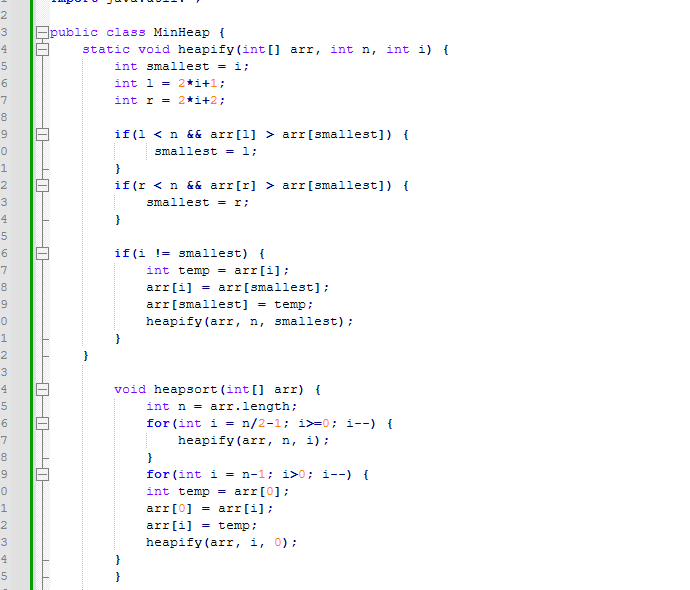
**Explanation –** Firstly, I’ve created two empty stacks, stack1 and stack2. When enqueuing an element, it is pushed into stack1. For dequeuing, the process first checks if stack2 is empty. If it is, all elements from stack1 are popped and pushed onto stack2. This transfer reverses the order of elements, and the element that was enqueued first to be on top of stack2. If stack2 is not empty, the top element of stack2 is popped and returned as the dequeued element. This method ensures that the queue maintains the correct order for adding and removing elements.

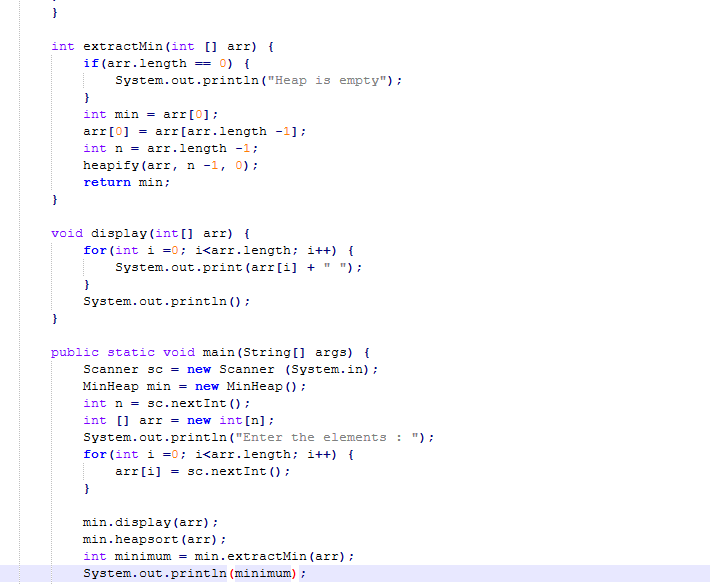
**Time Complexity:** Enqueue : 0(1) , Dequeue: O(1)

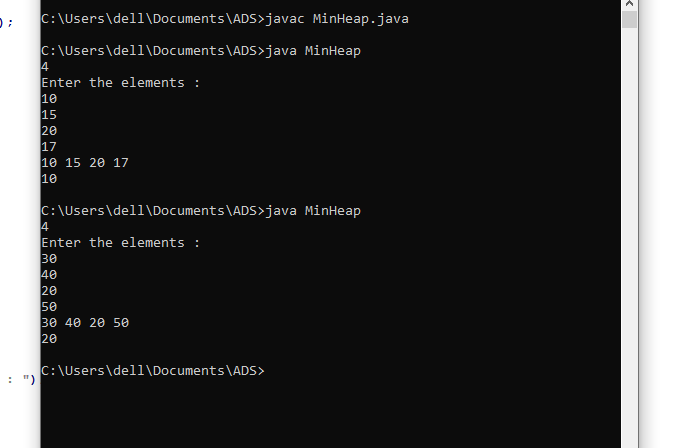
**Space Complexity:** O(n)

**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







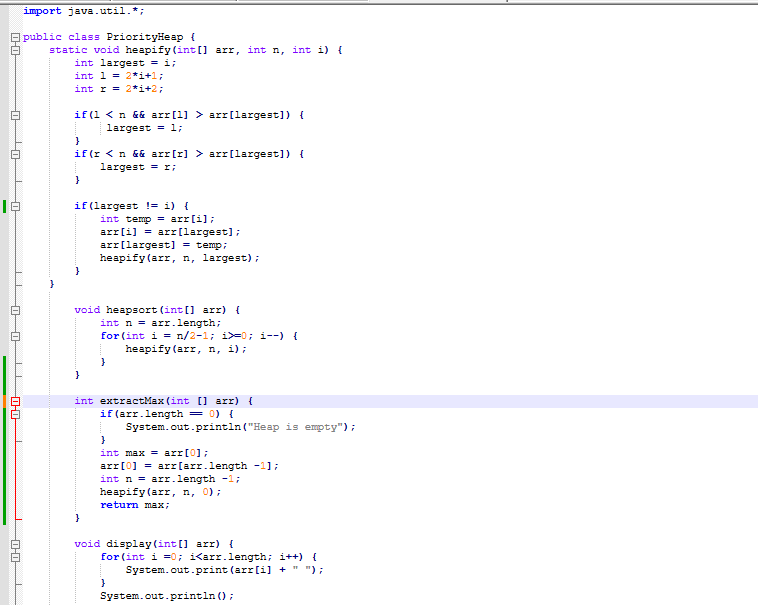
**Explanation -** In the MinHeap, I’ve first created the heapify method which is responsible for ensuring that the subtree rooted at any index satisfies the min-heap property. It compares the parent node with its left and right children and swaps them if necessary. The heapsort method converts the array into a min-heap by calling heapify on all non-leaf nodes, starting from the last non-leaf node up to the root. Once the array is transformed into a min-heap, it swaps the root with the last element of the current heap. Due to this the largest elements in their correct sorted positions are placed at the end of the array. After each swap, heapify is called to restore the min-heap property for the reduced heap. The display method prints the array before and after sorting, while the main method handles user input and calls the sorting process.

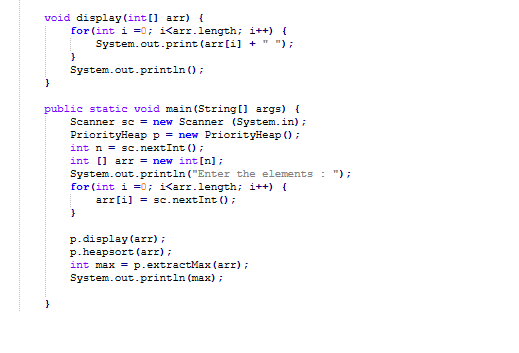
**Time Complexity:** O(log n)

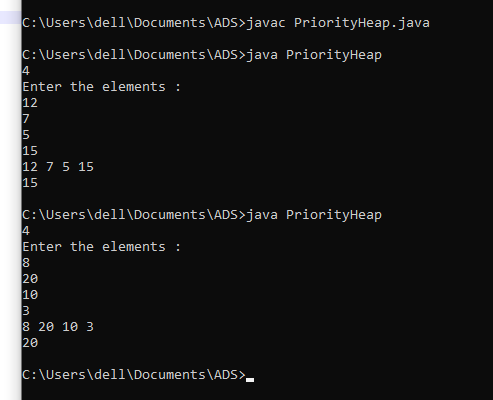
**Space Complexity:** O(1)

**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







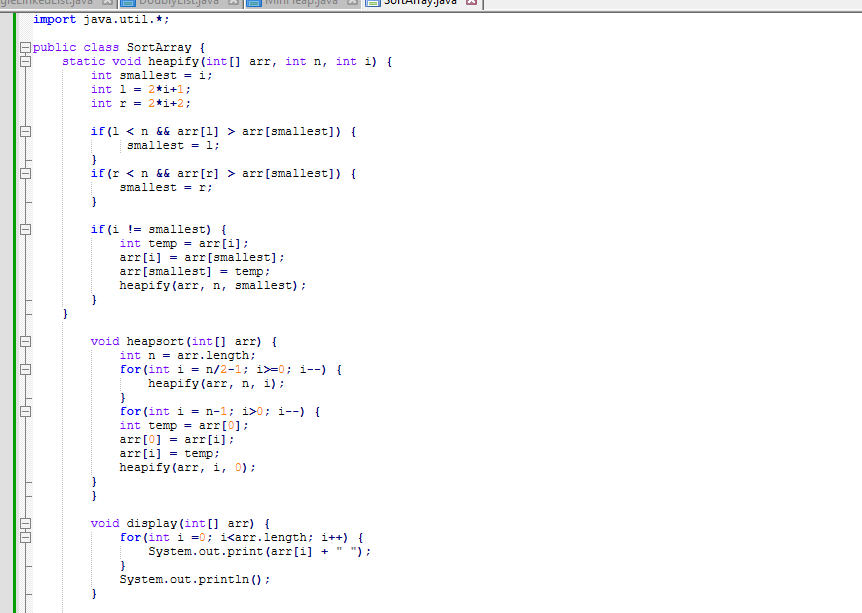
**Explanation –** In the PriorityHeap, I’ve first created the heapify method which is responsible for ensuring that the subtree rooted at any index satisfies the max-heap property. It compares the parent node with its left and right children and swaps them if necessary. The heapsort method converts the array into a max-heap by calling heapify on all non-leaf nodes, starting from the last non-leaf node up to the root. Once the array is transformed into a max-heap, it swaps the root with the last element of the current heap . Due to this the largest elements in their correct sorted positions are placed at the beginning of the array. After each swap, heapify is called to restore the max-heap property for the reduced heap. The display method prints the array before and after sorting, while the main method handles user input and calls the sorting process.

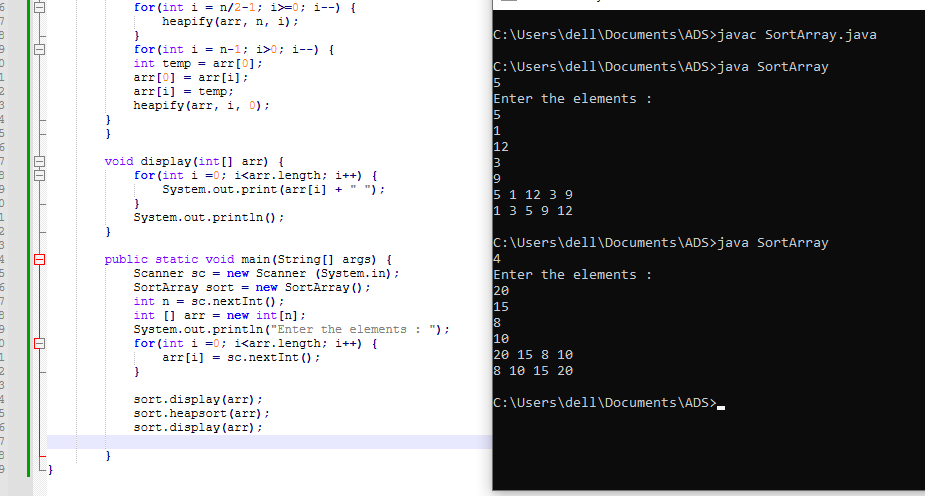
**Time Complexity:** O(n log n)

**Space Complexity:** O(log 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]





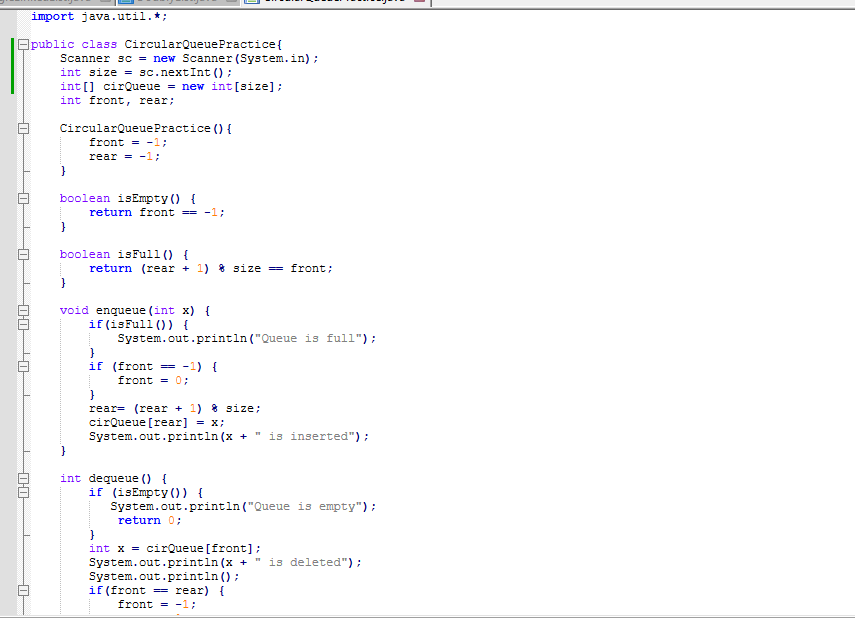
**Explanation -** In the SortArray, I’ve first created the heapify method which is responsible for ensuring that the subtree rooted at any index satisfies the min-heap property. It compares the parent node with its left and right children and swaps them if necessary. The heapsort method converts the array into a min-heap by calling heapify on all non-leaf nodes, starting from the last non-leaf node up to the root. Once the array is transformed into a min-heap, it swaps the root with the last element of the current heap. Due to this the largest elements in their correct sorted positions are placed at the end of the array.

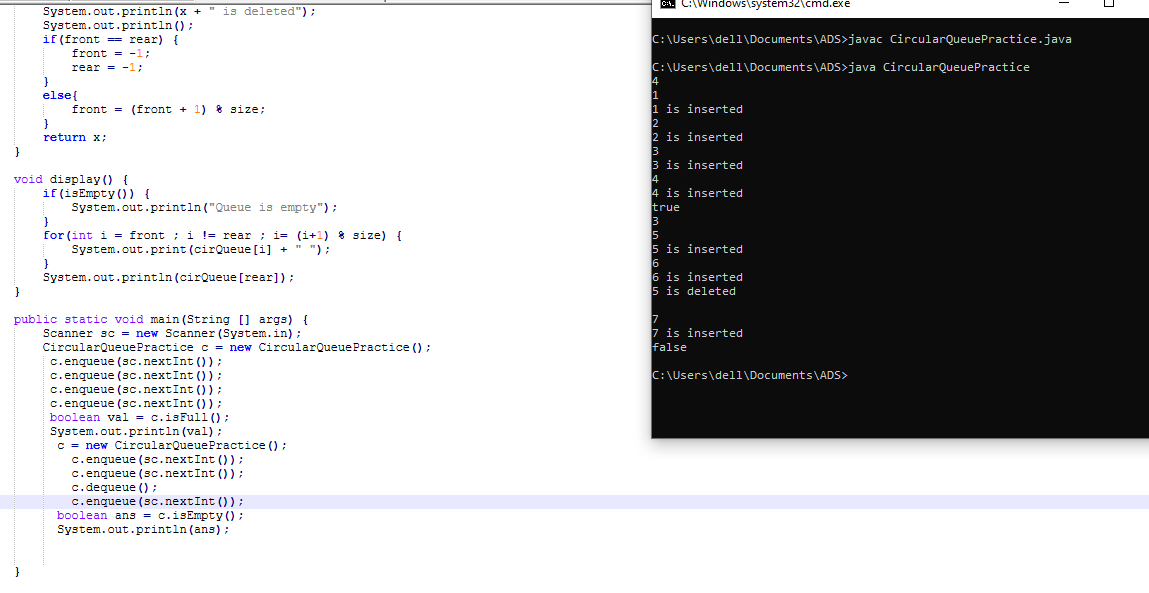
**Time Complexity:** O(n log n)

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

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





**Explanation -** In the Circular Queue program I’ve initialized an array with a fixed size of 5. The front and rear variables track the front and rear positions of the queue, initialized to -1. The isEmpty method checks whether the queue is empty by returning true. The isFull method checks if the queue is full The enqueue method inserts an element x into the queue. If the queue is full, it displays "Queue is full." If front is -1, it sets front to 0 to indicate the start of the queue. The dequeue method removes an element from the queue. If the queue is empty it displays "Queue is empty."

**Time Complexity:** Enqueue : 0(1) , Dequeue: O(1)

**Space Complexity:** O(n)