

Name: Vaishali Ramesh Kale

Email id: kalevaishalir16@gmail.com

DAY 6 Assignments of DS

Task 2: Linked List Middle Element Search You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.

Solution:::

```
package com.wipro.linear;
public class LinkedList {
    private Node head;
    private Node tail;
    private int length;
    class Node {
        int value;
        Node next;
        public Node(int value) {
            super();
            this.value = value;
        }
    }

    public LinkedList(int value) {
        super();
        Node newNode = new Node(value);
        // System.out.println("Node :" +newNode);
        head = newNode;
        tail = newNode;
        length = 1;
    }
}
```

```

public void getHead() {
    System.out.println("Head :" + head.value);
}

public void getTail() {
    System.out.println("Tail : " + tail.value);
}

public void getLength() {
    System.out.println("Length :" + length);
}

public void printList() {
    Node temp = head;
    System.out.println("\n");
    getHead();
    getTail();
    getLength();
    System.out.println("Items in list :");
    while (temp != null) {
        System.out.print("--->" + temp.value + " \t");
        temp = temp.next;
    }
}

//*****

public void append(int value) {
    Node newNode = new Node(value);
    if (length == 0) {
        head = newNode;
        tail = newNode;
    }
}

```

```

        } else {

            tail.next = newNode;

            tail = newNode;

        }

        length++;

    }

    /*******

public Node findMiddleElement() {

    if (head == null)

        return null;

    Node slow = head; // Slow pointer

    Node fast = head; // Fast pointer

    while (fast != null && fast.next != null) {

        slow = slow.next; // Move slow pointer by one step

        fast = fast.next.next; // Move fast pointer by two steps

    }

    return slow; // Slow pointer will be at the middle when fast pointer reaches the end

}

    public static void main(String[] args) {

        LinkedList myll = new LinkedList(11);

        myll.printList();

        myll.append(3);

        myll.append(23);

        myll.append(7);

        myll.append(16);

```

```

        myll.append(20);

        myll.printList();

        System.out.println("\n middle element is : "+myll.findMiddleElement().value);

    }

}

```

OUTPUT::::

```

java -cp /tmp/2K93A1m10L7/LinkEdLiS...

Head :11
Tail : 11
Length :1
Items in list :
--->11

Head :11
Tail : 20
Length :6
Items in list :
--->11 --->3 --->23 --->7 --->16 --->20
middle element is : 7

=== Code Execution Successful ===

```

Task 3: Queue Sorting with Limited Space You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.

SOLUTION::::

```

package com.wipro.linear;
import java.util.Queue;

import java.util.LinkedList;

import java.util.Stack;

public class QueueSortingWithOneStack {

```

```

// Method to sort the queue using an additional stack
public static void sortQueue(Queue<Integer> queue) {
    // Check if the queue is null or empty, if so, no sorting needed
    if (queue == null || queue.isEmpty()) return;

    // Stack to use as additional space
    Stack<Integer> stack = new Stack<>();

    // Get the size of the queue
    int n = queue.size();

    // Repeat the process for each element in the queue
    for (int i = 0; i < n; i++) {
        // Find the index of the minimum element in the unsorted part of the queue
        int minIndex = findMinIndex(queue, n - i);

        // Move the found minimum element to the end of the queue
        moveToEnd(queue, stack, minIndex, n - i);
    }
}

// Method to find the index of the minimum element in the queue
private static int findMinIndex(Queue<Integer> queue, int sortedSize) {
    int minIndex = -1;
    int minValue = Integer.MAX_VALUE;
    int size = queue.size();

    // Iterate through the queue to find the minimum element in the unsorted part
    for (int i = 0; i < size; i++) {
        int current = queue.poll();

```

```

        if (i < sortedSize && current < minValue) {
            minValue = current;
            minIndex = i;
        }

        // Re-enqueue the element back to the queue
        queue.add(current);
    }

    // Return the index of the minimum element
    return minIndex;
}

// Method to move the minimum element to the end of the queue
private static void moveToEnd(Queue<Integer> queue, Stack<Integer> stack, int minIndex,
int sortedSize) {
    int size = queue.size();
    int minValue = Integer.MAX_VALUE;

    // Iterate through the queue and move elements to the stack except the minimum
    element
    for (int i = 0; i < size; i++) {
        int current = queue.poll();
        if (i == minIndex) {
            minValue = current;
        } else {
            stack.push(current);
        }
    }
}

// Move elements back from the stack to the queue

```

```

        while (!stack.isEmpty()) {
            queue.add(stack.pop());
        }

        // Add the minimum element at the end of the queue
        queue.add(minValue);
    }

    // Main method to test the queue sorting
    public static void main(String[] args) {
        // Initialize the queue with some elements
        Queue<Integer> queue = new LinkedList<>();
        queue.add(9);
        queue.add(3);
        queue.add(7);
        queue.add(2);
        queue.add(5);

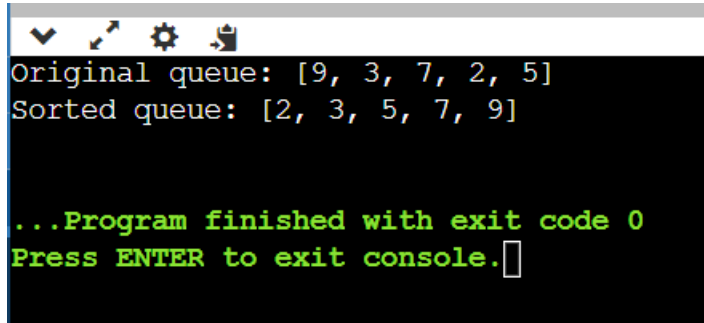
        // Print the original queue
        System.out.println("Original queue: " + queue);

        // Sort the queue
        sortQueue(queue);

        // Print the sorted queue
        System.out.println("Sorted queue: " + queue);
    }
}

```

OUTPUT::



```
Original queue: [9, 3, 7, 2, 5]
Sorted queue: [2, 3, 5, 7, 9]

...Program finished with exit code 0
Press ENTER to exit console.
```

Steps to sort the elements in the queue::

1. Class and Import Statements:
Imports the necessary classes for using Queue, LinkedList, and Stack.
2. SortQueue Method:
 - Checks if the queue is null or empty.
 - Initializes a stack to use as additional space.
 - Gets the size of the queue and iterates to sort it.
 - For each iteration, finds the minimum element's index and moves it to the end of the queue.
3. findMinIndex Method:
 - Finds the index of the minimum element within the unsorted portion of the queue.
 - Iterates through the queue, updating the minimum value and its index.
 - Restores the queue by re-enqueueing each element.
4. moveToEnd Method:
 - Moves the minimum element (found by findMinIndex) to the end of the queue.
 - Temporarily stores elements in the stack to facilitate the operation.
 - Moves elements back to the queue from the stack, ensuring the minimum element is added last.
5. main Method:
 - Creates a queue with example elements.
 - Prints the original queue.
 - Calls sortQueue to sort the queue.

- Prints the sorted queue.

Task 4: Stack Sorting In-Place You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

Solution::

```
import java.util.Stack;
```

```
public class StackSortingInPlace {
```

```
    // Method to sort the stack in place
```

```
    public static void sortStack(Stack<Integer> stack) {
```

```
        // Temporary stack to hold elements in sorted order
```

```
        Stack<Integer> tempStack = new Stack<>();
```

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

```
            // Pop an element from the original stack
```

```
            int current = stack.pop();
```

```
            // While temporary stack is not empty and the top of tempStack is greater than  
            current
```

```
            while (!tempStack.isEmpty() && tempStack.peek() < current) {
```

```
                // Pop from tempStack and push it back to the original stack
```

```
                stack.push(tempStack.pop());
```

```
            }
```

```
            // Push the current element onto the tempStack
```

```
            tempStack.push(current);
```

```

    }

    // Transfer the sorted elements back to the original stack
    while (!tempStack.isEmpty()) {
        stack.push(tempStack.pop());
    }
}

public static void main(String[] args) {
    // Test the sortStack method with an example stack
    Stack<Integer> stack = new Stack<>();
    stack.push(34);
    stack.push(3);
    stack.push(31);
    stack.push(98);
    stack.push(92);
    stack.push(23);

    System.out.println("Original stack: " + stack);

    // Sort the stack
    sortStack(stack);

    // Print the sorted stack
    System.out.println("Sorted stack: " + stack);
}
}

```

OUTPUT:::

```
Original stack: [34, 3, 31, 98, 92, 23]

Sorted stack: [3, 23, 31, 34, 92, 98]

...Program finished with exit code 0
Press ENTER to exit console.
```

Explanation::

1. sortStack Method:
 - The method takes a stack as input and sorts it using an additional temporary stack.
 - It iteratively pops elements from the original stack and places them in the correct order in the temporary stack.
2. Main Loop:
 - While the original stack is not empty, pop an element from it and store it in current.
 - Compare current with the elements in the temporary stack. If the top element of the temporary stack is greater than current, pop elements from the temporary stack back into the original stack until the correct position for current is found.
 - Push current onto the temporary stack.
3. Transfer Back:
 - Once all elements are sorted in the temporary stack (with the largest at the bottom and the smallest at the top), transfer them back to the original stack. This will ensure that the smallest elements are on the top of the original stack.
4. Main Method:
 - Creates a stack with example elements.
 - Prints the original stack.
 - Calls sortStack to sort the stack.
 - Prints the sorted stack.

Task 5: Removing Duplicates from a Sorted Linked List A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.

Solution:::

Here is a step-by-step description of the algorithm:

1. Initialize a Pointer: Start with a pointer (current) at the head of the linked list.
2. Traverse the List: Iterate through the list while the current node and its next node are not null.
3. Check for Duplicates: For each node, compare the value of the current node with the value of the next node.
4. If they are equal, skip the next node by updating the next pointer of the current node to point to the node after the next node.
5. If they are not equal, move the current pointer to the next node.
6. End of List: The algorithm stops when the current node or its next node is null.
7. This approach ensures that each node is visited only once, making the algorithm efficient with a time complexity of $O(n)$, where n is the number of nodes in the list.

CODE:::

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class RemoveDuplicatesFromSortedList {
    public static ListNode deleteDuplicates(ListNode head) {
        // Initialize the current node to the head of the list
        ListNode current = head;

        // Traverse the list until the end
        while (current != null && current.next != null) {
            // Compare the current node with the next node
            if (current.val == current.next.val) {
                // If they are equal, skip the next node
                current.next = current.next.next;
            } else {
                // If they are not equal, move to the next node
                current = current.next;
            }
        }
    }
}
```

```

    }

    // Return the head of the modified list
    return head;
}

// Helper method to print the list
public static void printList(ListNode head) {
    ListNode current = head;
    while (current != null) {
        System.out.print(current.val + " ");
        current = current.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // Creating a sorted linked list with duplicates: 1 -> 1 -> 2 -> 3 -> 3
    ListNode head = new ListNode(1);
    head.next = new ListNode(1);
    head.next.next = new ListNode(2);
    head.next.next.next = new ListNode(3);
    head.next.next.next.next = new ListNode(3);

    System.out.println("Original list:");
    printList(head);

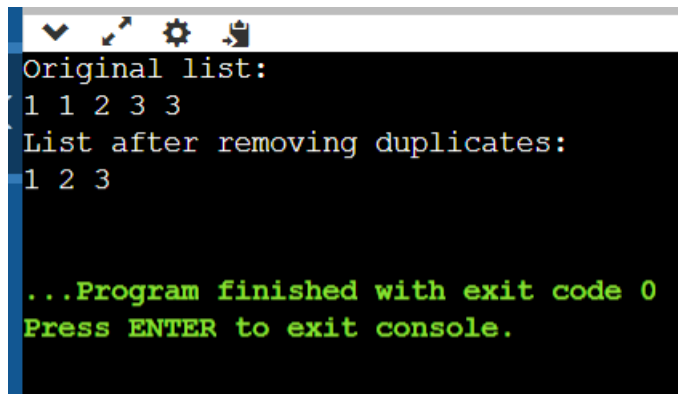
    // Removing duplicates
    head = deleteDuplicates(head);

    System.out.println("List after removing duplicates:");
    printList(head);
}

```

```
}  
}
```

OUTPUT:::



```
Original list:  
1 1 2 3 3  
List after removing duplicates:  
1 2 3  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Task 6: Searching for a Sequence in a Stack Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.

Solution:::

```
import java.util.Stack  
  
public class StackSequenceSearch {  
    public static boolean isSequencePresent(Stack<Integer> stack, int[] sequence) {  
        // Edge case: if sequence array is empty, return true  
        if (sequence.length == 0) {  
            return true;  
        }  
  
        // Initialize index to track the position in the sequence array  
        int index = sequence.length - 1;  
  
        // Iterate through the stack  
        while (!stack.isEmpty() && index >= 0) {  
            // If the current element of the stack matches the current element of the sequence
```

```

        if (stack.peek() == sequence[index]) {
            // Decrement the index to check the next element of the sequence
            index--;
        }
        // Pop the element from the stack regardless of whether it matches the sequence or
not
        System.out.println("Popping: " + stack.pop());
    }
    // If all elements of the sequence were found, return true
    return index == -1;
}

public static void main(String[] args) {
    // Test the isSequencePresent method with a sample stack and sequence
    Stack<Integer> stack = new Stack<>();
    stack.push(4);
    stack.push(3);
    stack.push(2);
    stack.push(1);
    stack.push(5);
    stack.push(6);

    int[] sequence = {3, 2, 1}; // Sequence to search for
    // Print the stack elements
    System.out.println("Stack elements:");
    for (Integer element : stack) {
        System.out.println(element);
    }
    // Check if the sequence is present in the stack
    boolean sequencePresent = isSequencePresent(stack, sequence);

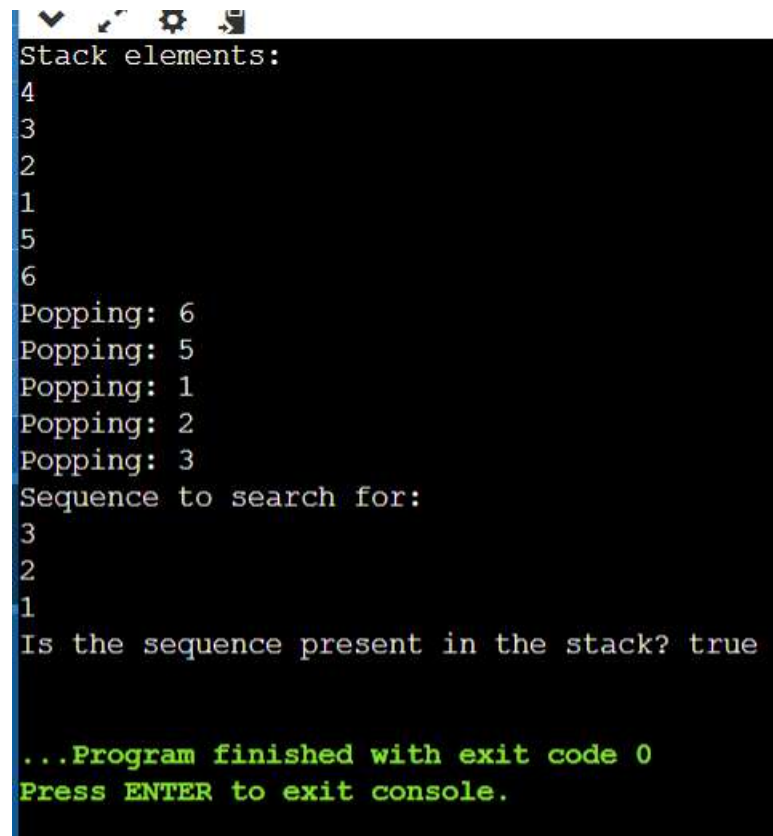
```

```

// Print the sequence to search for
System.out.println("Sequence to search for:");
for (int num : sequence) {
    System.out.println(num);
}
// Print the result
System.out.println("Is the sequence present in the stack? " + sequencePresent);
}
}

```

OUTPUT:::



```

Stack elements:
4
3
2
1
5
6
Popping: 6
Popping: 5
Popping: 1
Popping: 2
Popping: 3
Sequence to search for:
3
2
1
Is the sequence present in the stack? true

...Program finished with exit code 0
Press ENTER to exit console.

```

Task 7: Merging Two Sorted Linked Lists You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).

Solution:::

Steps::

1. Initialize Pointers: Create a dummy node to serve as the head of the merged list. Initialize pointers current and prev to the dummy node.
2. Merge Lists: Iterate through both lists simultaneously. Compare the values of the current nodes of the two lists.
3. Update Pointers:
 - If the value of the current node in the first list is smaller or equal to the value of the current node in the second list, connect the next pointer of the prev node to the current node of the first list. Move the current pointer of the first list to its next node.
 - Otherwise, connect the next pointer of the prev node to the current node of the second list. Move the current pointer of the second list to its next node.
4. Handle Remaining Nodes: After one list is fully traversed, if there are any remaining nodes in the other list, connect them directly to the next of the prev node.
5. Return Merged List: Return the next of the dummy node, which is the head of the merged list.

CODE:::

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class MergeTwoSortedLinkedLists {
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // Edge case: if any of the lists is null, return the other list
        if (l1 == null) return l2;
        if (l2 == null) return l1;

        // Create a dummy node to serve as the head of the merged list
        ListNode dummy = new ListNode(0);
        // Pointer to track the current node in the merged list
        ListNode current = dummy;

        // Merge the lists while both of them have nodes
```

```

while (l1 != null && l2 != null) {
    // Compare the values of the current nodes of the two lists
    if (l1.val <= l2.val) {
        // Connect the next pointer of the current node to the smaller node
        current.next = l1;

        // Move the current pointer of the first list to its next node
        l1 = l1.next;
    } else {
        // Connect the next pointer of the current node to the smaller node
        current.next = l2;

        // Move the current pointer of the second list to its next node
        l2 = l2.next;
    }

    // Move the current pointer of the merged list to its next node
    current = current.next;
}

// Connect any remaining nodes of the non-null list to the merged list
current.next = (l1 != null) ? l1 : l2;

// Return the head of the merged list (next of the dummy node)
return dummy.next;
}

// Helper method to print the linked list
public static void printList(ListNode head) {
    ListNode current = head;
    while (current != null) {
        System.out.print(current.val + " ");
    }
}

```

```

        current = current.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // Create two sorted linked lists
    ListNode l1 = new ListNode(1);
    l1.next = new ListNode(3);
    l1.next.next = new ListNode(5);

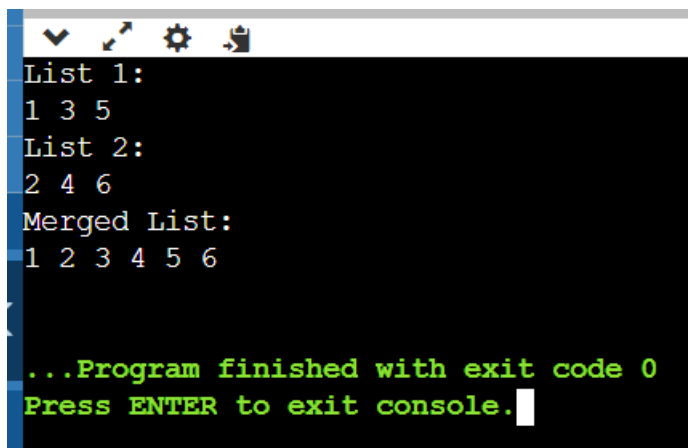
    ListNode l2 = new ListNode(2);
    l2.next = new ListNode(4);
    l2.next.next = new ListNode(6);

    System.out.println("List 1:");
    printList(l1);
    System.out.println("List 2:");
    printList(l2);

    // Merge the two lists
    ListNode mergedList = mergeTwoLists(l1, l2);
    // Print the merged list
    System.out.println("Merged List:");
    printList(mergedList);
}
}

```

OUTPUT:::



```
✓ ↗ ⚙ 📄
List 1:
1 3 5
List 2:
2 4 6
Merged List:
1 2 3 4 5 6

...Program finished with exit code 0
Press ENTER to exit console.
```

Task 8: Circular Queue Binary Search Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.

Solution::::

Steps:::

1. Find the Rotation Index:
 - Perform a modified binary search to find the rotation index, which is the index where the rotation of elements starts.
 - The rotation index is the index where the current element is greater than the next element. This indicates the rotation point.
2. Determine the Search Range:
 - Based on the rotation index, determine which portion of the circular queue to search.
 - If the target element is greater than or equal to the first element of the queue and less than or equal to the last element, search the entire queue.
 - Otherwise, search either the left or right portion of the queue.
3. Perform Binary Search:
 - Use a modified binary search algorithm to search for the target element within the determined search range.
 - Adjust the mid index calculation to wrap around the circular queue.
4. Handle Edge Cases:
5. Consider edge cases where the target element may be at the ends of the circular queue or not found at all.

CODE:::

```
public class CircularQueueBinarySearch {

    public static int searchCircularQueue(int[] queue, int target) {

        int rotationIndex = findRotationIndex(queue);

        System.out.println("Rotation Index: " + rotationIndex);

        int left = 0, right = queue.length - 1;

        // Determine the search range based on the rotation index
        if (target >= queue[0] && target <= queue[right]) {
            // Search the entire queue
            return binarySearch(queue, target, left, right);
        } else if (target >= queue[0]) {
            // Search the left portion of the queue
            right = rotationIndex;
        } else {
            // Search the right portion of the queue
            left = rotationIndex + 1;
            right = queue.length - 2; // Exclude the last element from the search range
        }

        // Perform binary search within the determined search range
        return binarySearch(queue, target, left, right);
    }

    public static int binarySearch(int[] queue, int target, int left, int right) {
        while (left <= right) {
            int mid = left + (right - left) / 2;
```

```

        System.out.println("Mid Index: " + mid);
        if (queue[mid] == target) {
            return mid;
        }

        // Check if the mid index is on the "lower" side or the "upper" side of the circular
        queue
        boolean midOnLowerSide = queue[mid] >= queue[0];

        // Check if the target is on the "lower" side or the "upper" side of the circular queue
        boolean targetOnLowerSide = target >= queue[0];

        System.out.println("Mid Value: " + queue[mid] + ", Target: " + target);

        // Adjust the binary search based on the relative positions of mid and target in the
        circular queue
        if ((midOnLowerSide && targetOnLowerSide && queue[mid] < target) ||
            (!midOnLowerSide && !targetOnLowerSide && queue[mid] < target) ||
            (midOnLowerSide && !targetOnLowerSide) ||
            (!midOnLowerSide && targetOnLowerSide)) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    // Element not found
    return -1;
}

public static int findRotationIndex(int[] queue) {

```

```

int left = 0, right = queue.length - 1;

while (left < right) {
    int mid = left + (right - left) / 2;
    if (queue[mid] > queue[right]) {
        left = mid + 1;
    } else {
        right = mid;
    }
}
return left;
}

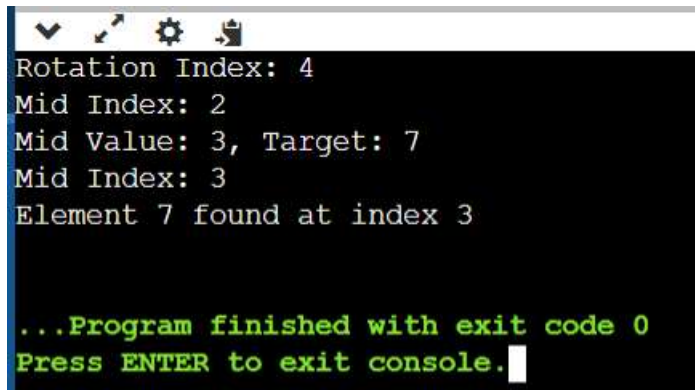
public static void main(String[] args) {
    int[] queue = {4, 5, 3, 7, 0, 1, 2};
    int target = 7;

    int index = searchCircularQueue(queue, target);

    if (index != -1) {
        System.out.println("Element " + target + " found at index " + index);
    } else {
        System.out.println("Element " + target + " not found in the circular queue");
    }
}
}

```

OUTPUT:::

A screenshot of a terminal window with a black background and white text. The window has a title bar with standard icons (minimize, maximize, close, and a gear icon). The text inside the terminal shows the steps of a binary search algorithm. It starts with 'Rotation Index: 4', followed by 'Mid Index: 2', 'Mid Value: 3, Target: 7', and 'Mid Index: 3'. The final line of the search is 'Element 7 found at index 3'. Below this, in green text, it says '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a white cursor at the end.

```
Rotation Index: 4
Mid Index: 2
Mid Value: 3, Target: 7
Mid Index: 3
Element 7 found at index 3

...Program finished with exit code 0
Press ENTER to exit console.
```