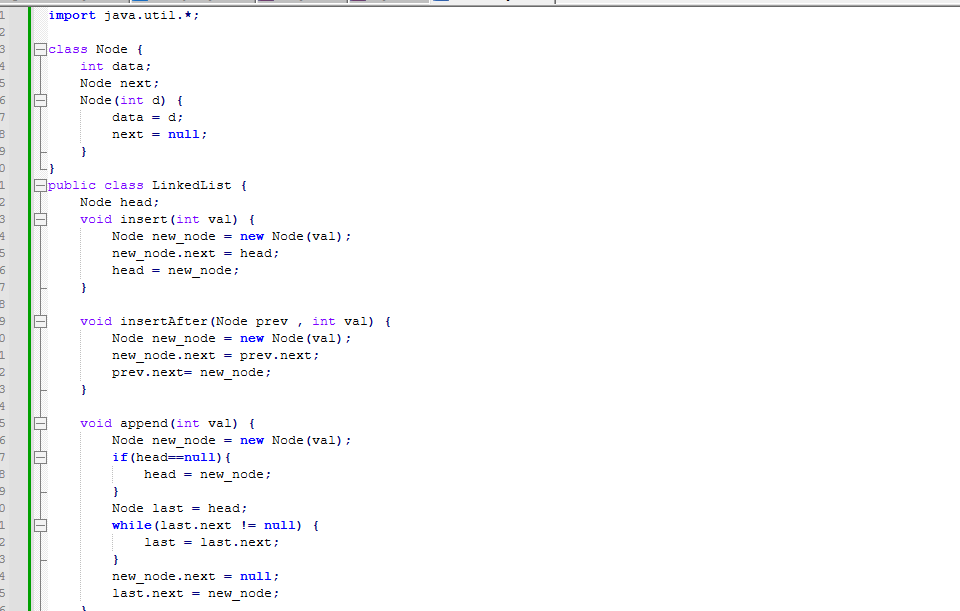
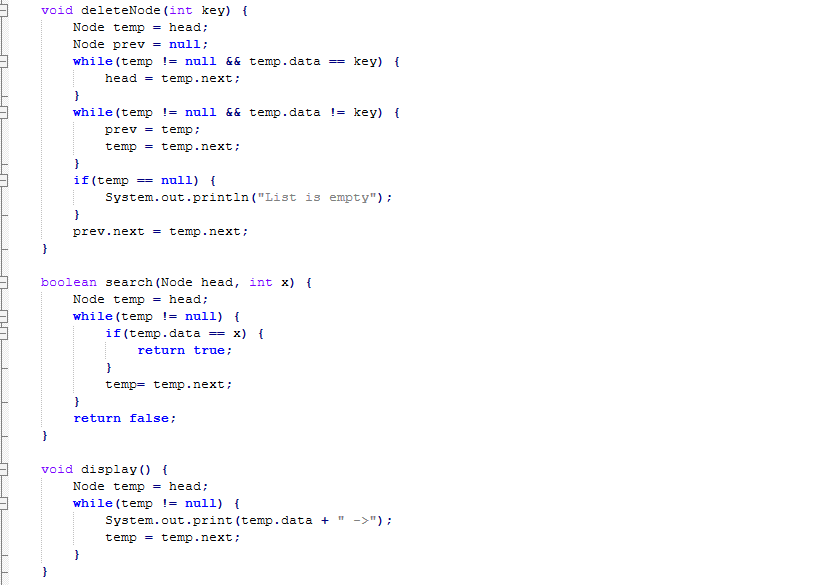
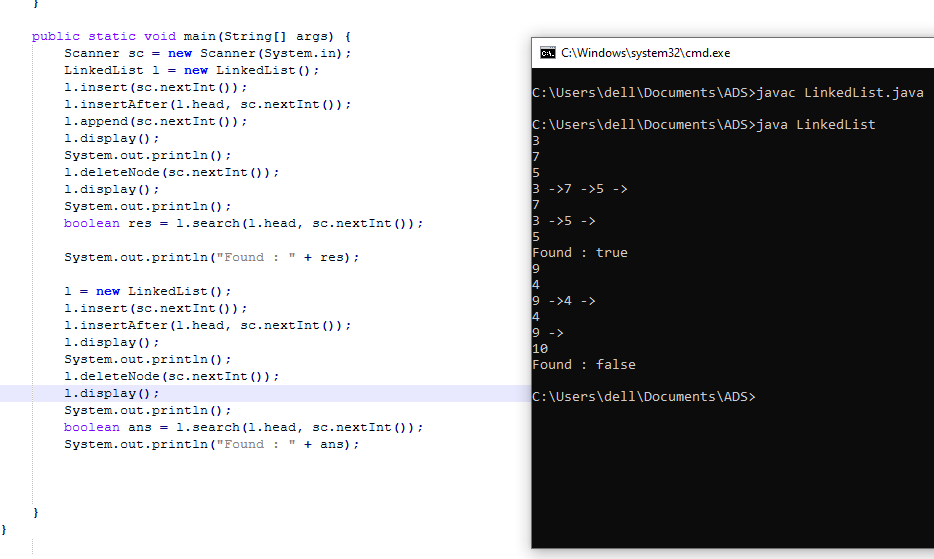
**Assignment -4**

1. Implement a singly linked list with basic operations: insert, delete, search.

* Test Case 1:  
  Input: Insert 3 → Insert 7 → Insert 5 → Delete 7 → Search 5  
  Output: List = [3, 5], Found = True
* Test Case 2:  
  Input: Insert 9 → Insert 4 → Delete 4 → Search 10  
  Output: List = [9], Found = False







**Flowchart-**

**Start Program**

**|**

**Create LinkedList**

**|**

**Insert node at head (insert)**

**|**

**| Insert node after head (insertAfter)|**

**|**

**Append node to end (append)**

**|**

**Display linked list**

**|**

**Delete node (deleteNode)**

**|**

**Display updated linked list**

**|**

**Search for a value (search)**

**|**

**Found: true / false**

**|**

**Reset LinkedList (new)**

**|**

**Repeat Insert, InsertAfter, Append**

**|**

**End**

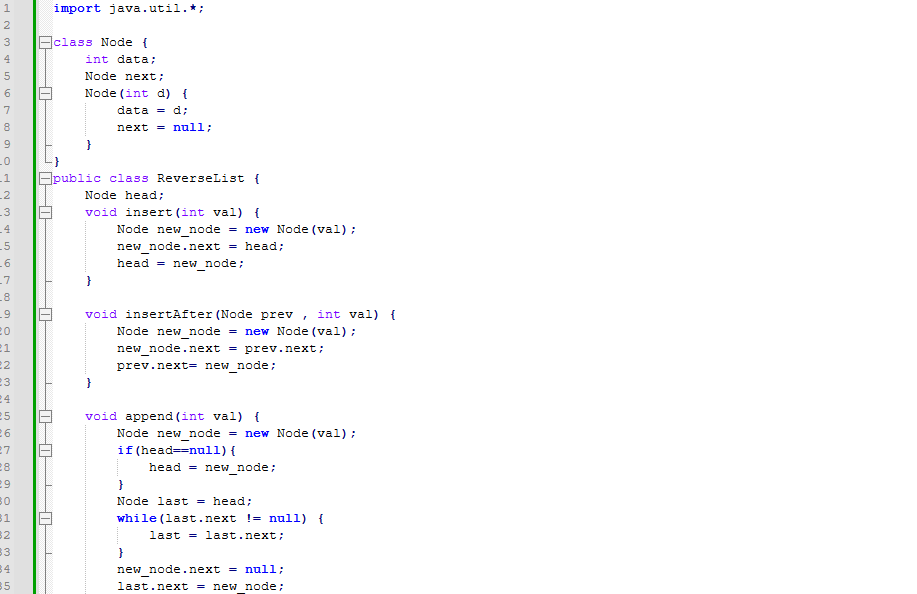
**Explanation -** The LinkedList program starts by creating an empty linked list. The user enters numbers. First, it adds a number to the beginning of the list, then adds another number right after the first one, and finally adds a number to the end of the list. After that, it shows the list with all the numbers. The user can then enter a number to delete it from the list, and the program will display the updated list again. It also lets the user check if a specific number is in the list and tells them whether it found the number or not.

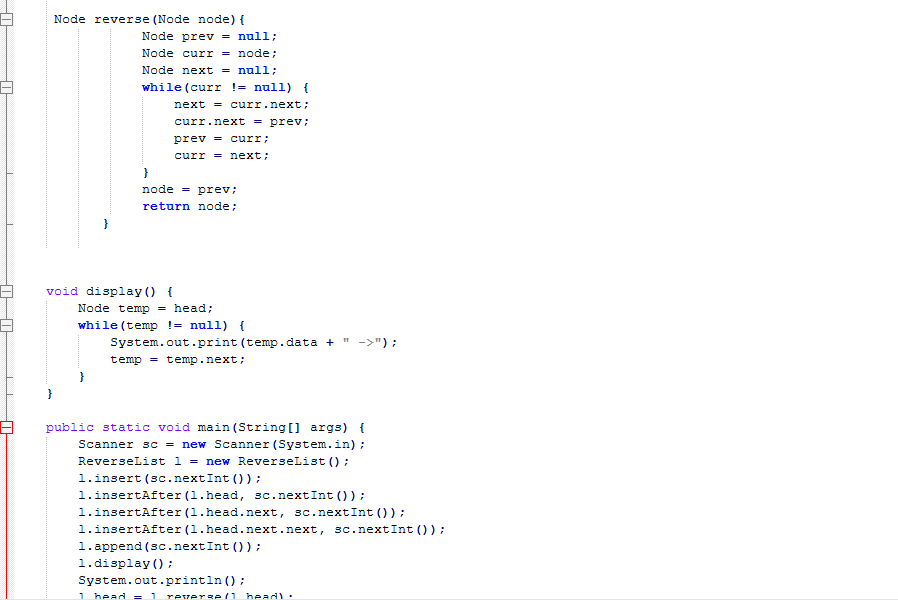
**Time Complexity -** O(n)

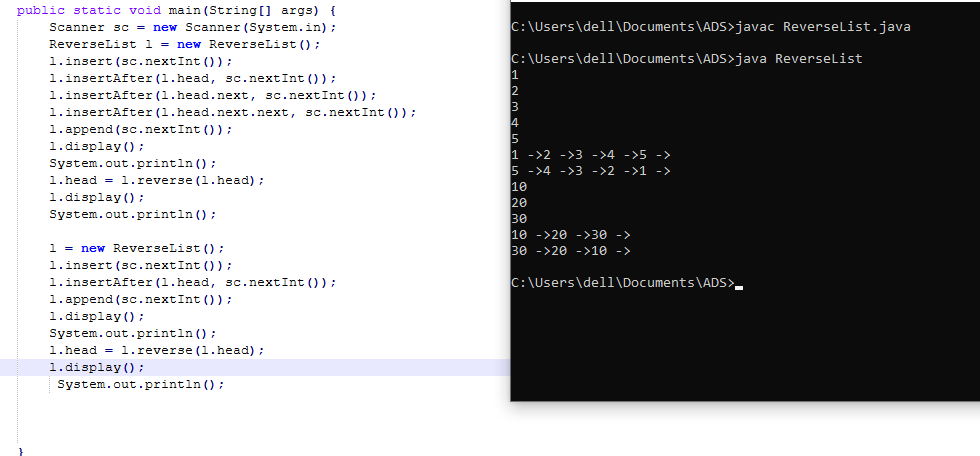
**Space Complexity** – O(n)

**2. Reverse a singly linked list.**

* **Test Case 1**:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: List = [5, 4, 3, 2, 1]
* **Test Case 2**:  
  Input: List = [10, 20, 30]  
  Output: List = [30, 20, 10]







**Flowchart-**

**Start**

**|**

**Created ReverseList**

**|**

**Insert Value**

**|**

**Insert After Specific Node (insertAfter method)**

**|**

**Insert More Values (insertAfter method)**

**|**

**Append Value at End (append method)**

**|**

**Display List (display method)**

**|**

**Reverse List (reverse method)**

**|**

**Display Reversed List (display method)**

**|**

**Repeat the Process for New List**

**|**

**End**

**Explanation -** The ReverseList class contains a head node and several methods The insert method inserts a new node at the head of the list. A new node is created, linked to the current head, and then the head is updated to point to this new node. The insertAfter method inserts a node after a given prev node, linking the new node's next to the node after prev and updating prev to point to the new node. The append method adds a node at the end of the list. It first checks if the list is empty; if so, the new node becomes the head. Otherwise, it traverses the list to find the last node and appends the new node after it.

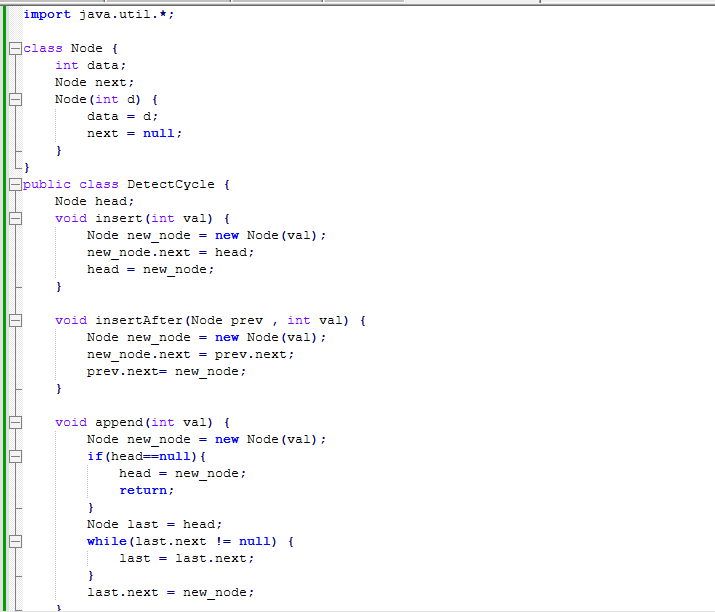
The reverse method reverses the linked list by iterating through the nodes and changing their next pointers to point to the previous node. The display() method traverses the list and prints the values of the nodes . In the main() method, the I’ve taken the input from user with help of Scanner class and invoke the methods.

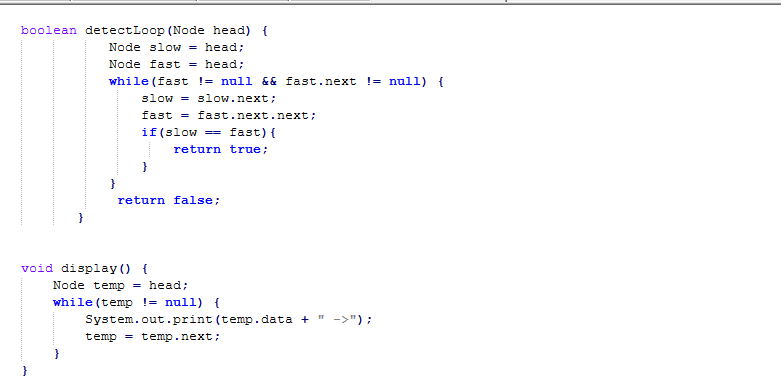
**Time Complexity -** O(n)

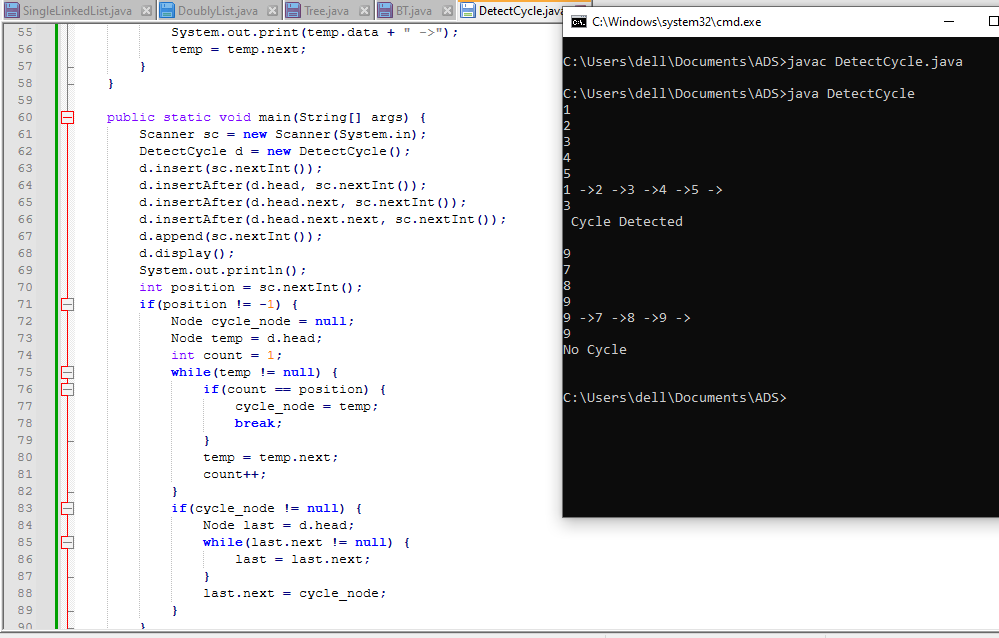
**Space Complexity** – O(n)

**3. Detect a cycle in a linked list.**

* **Test Case 1**:  
  Input: List = [1 → 2 → 3 → 4 → 5 → 3 (cycle)]  
  Output: Cycle Detected
* **Test Case 2**:  
  Input: List = [6 → 7 → 8 → 9]  
  Output: No Cycle







**Flowchart –**

**Start**

**|**

**Initialize Scanner**

**|**

**Create DetectCycle |**

**Input value and insert**

**|**

**Input values and insertAfter**

**|**

**Input value to append**

**|**

**Display Linked List**

**|**

**Input position to create cycle**

**|**

**If position != -1**

**|**

**Initialize cycle\_node**

**|**

**Traverse to find cycle\_node**

**|**

**Create cycle if found**

**|**

**Detect Loop**

**|**

**If cycle detected**

**Print "Cycle Detected"**

**Else**

**Print "No Cycle"**

**|**

**End**

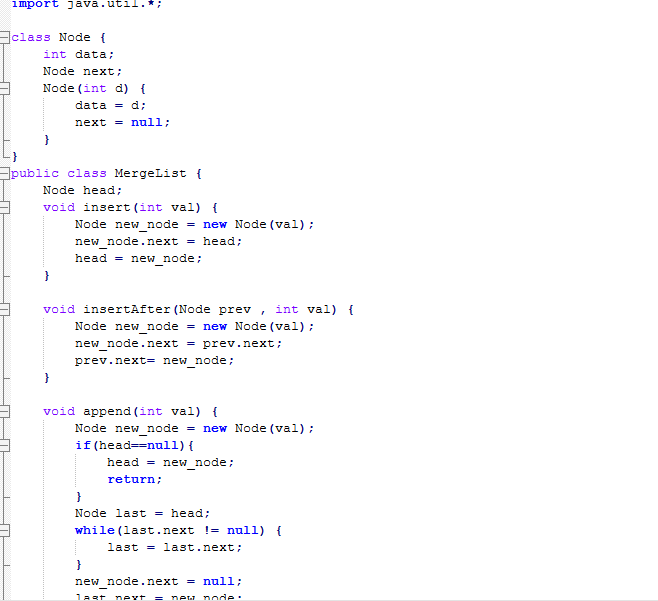
**Explanation -** The DetectCycle class contains methods for inserting nodes at the beginning, appending nodes to the end, inserting after a specified node, and detecting cycles. In the main method, the program initializes a Scanner for user input, allowing the user to enter values to construct the linked list. If a valid position is provided, the program links the last node of the list to the node at the specified position, forming a cycle. The cycle detection method is then called to check for cycles, and the program outputs whether a cycle was detected or not.

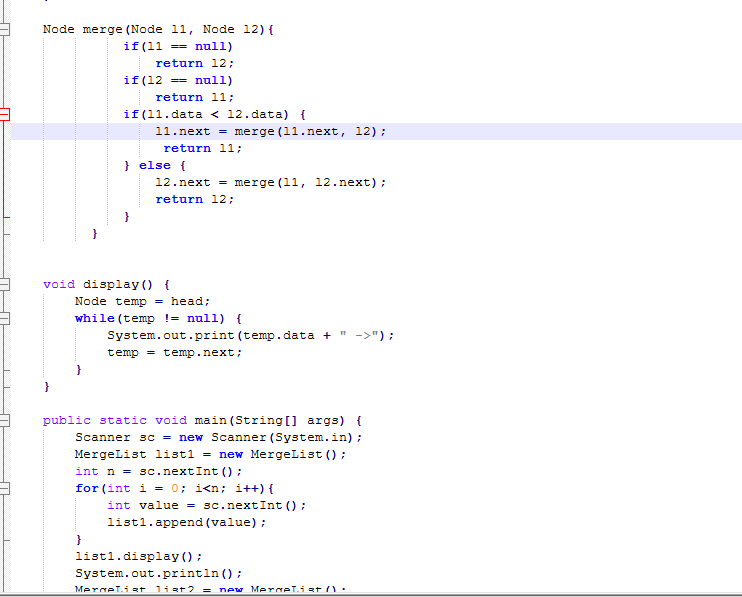
**Time Complexity -** O(n)

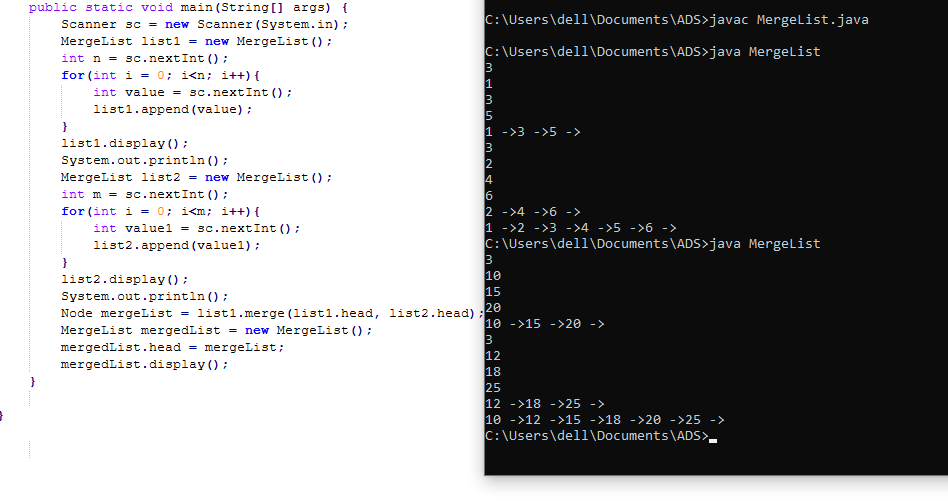
**Space Complexity** – O(n)

**4. Merge two sorted linked lists.**

* **Test Case 1**:  
  Input: List1 = [1, 3, 5], List2 = [2, 4, 6]  
  Output: Merged List = [1, 2, 3, 4, 5, 6]
* **Test Case 2**:  
  Input: List1 = [10, 15, 20], List2 = [12, 18, 25]  
  Output: Merged List = [10, 12, 15, 18, 20, 25]







**Flowchart-**

**Start**

**|**

**Input n**

**|**

**Loop (0 to n-1)**

**|**

**Input value**

**|**

**Call append(value)**

**|**

**Display list1**

**|**

**Input m**

**|**

**Loop (0 to m-1)**

**|**

**Input value1**

**|**

**Call append(value1)**

**|**

**Display list2**

**|**

**Call merge(list1.head, list2.head)**

**|**

**Create mergedList**

**|**

**Assign mergeList to mergedList.head**

**|**

**Display mergedList**

**|**

**End**

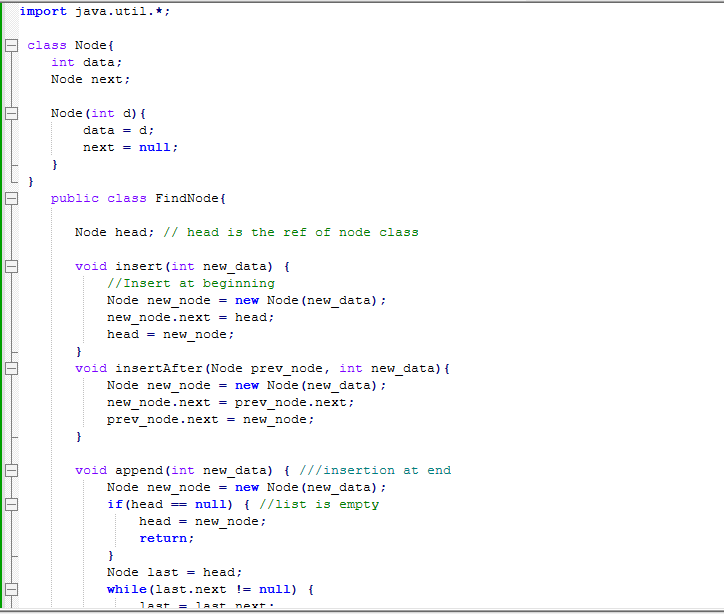
**Explanation –** In the class, MergeList, there are methods to insert nodes into the list, either at the beginning or the end, and a method merge() that combines two sorted linked lists into one. The program allows the user to input the number of elements for two separate linked lists, and the values for each list. After merging it results in a new sorted list, which is then displayed.

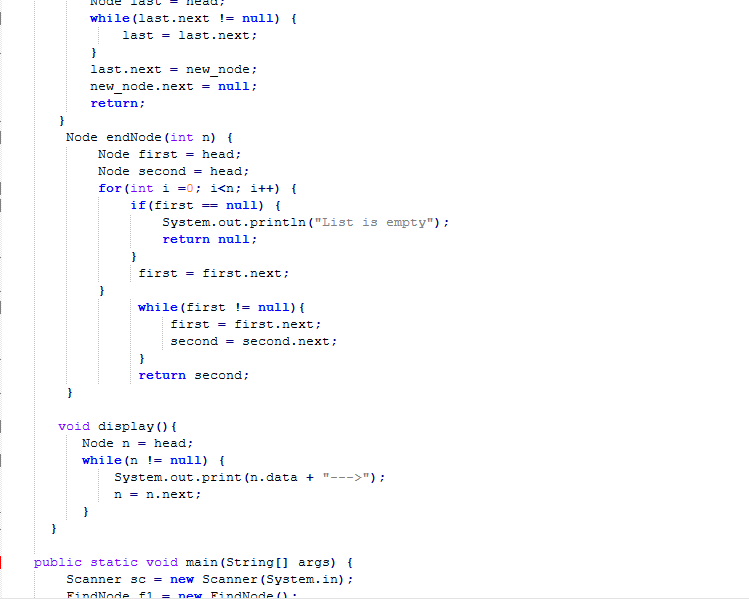
**Time Complexity –** O(n2+m2)

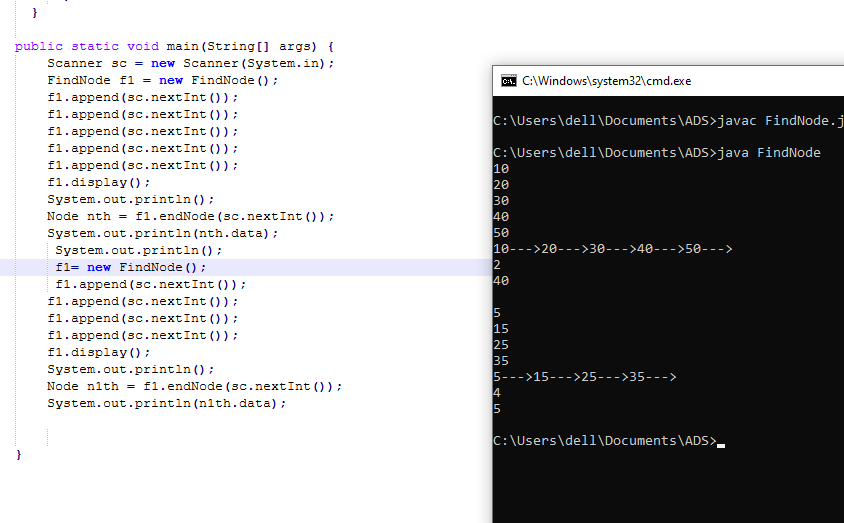
**Space Complexity** – O(n + m)

**5. Find the nth node from the end of a linked list.**

* **Test Case 1**:  
  Input: List = [10, 20, 30, 40, 50], n = 2  
  Output: 40
* **Test Case 2**:  
  Input: List = [5, 15, 25, 35], n = 4  
  Output: 5







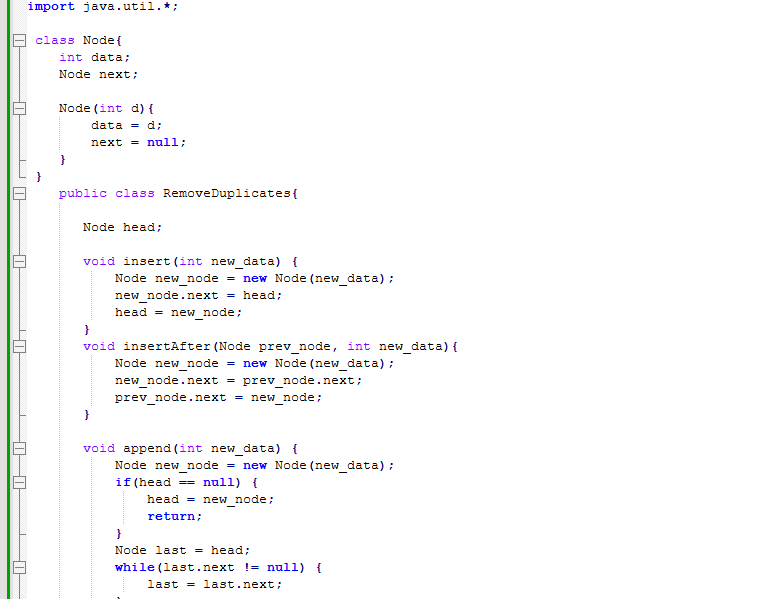
**Explanation –** In the FindNode there are several methods. The insert method adds a new node at the beginning of the list, while the insertAfter method adds a new node after a given node. The append method adds a new node at the end of the list by traversing to the last node and linking the new node. The endNode method finds the nth node from the end of the list using two pointers. The first pointer moves n steps ahead, and then both the first and second pointers move together until the first pointer reaches the end, at which point the second pointer is at the desired node. The display method shows node’s data.In the main method, the program then finds and prints the nth node from the end based on user input.

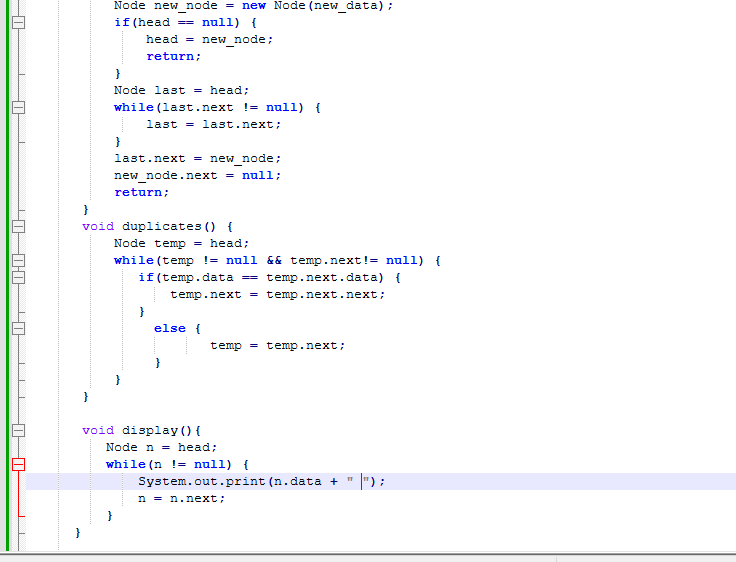
**Time Complexity –** O(n)

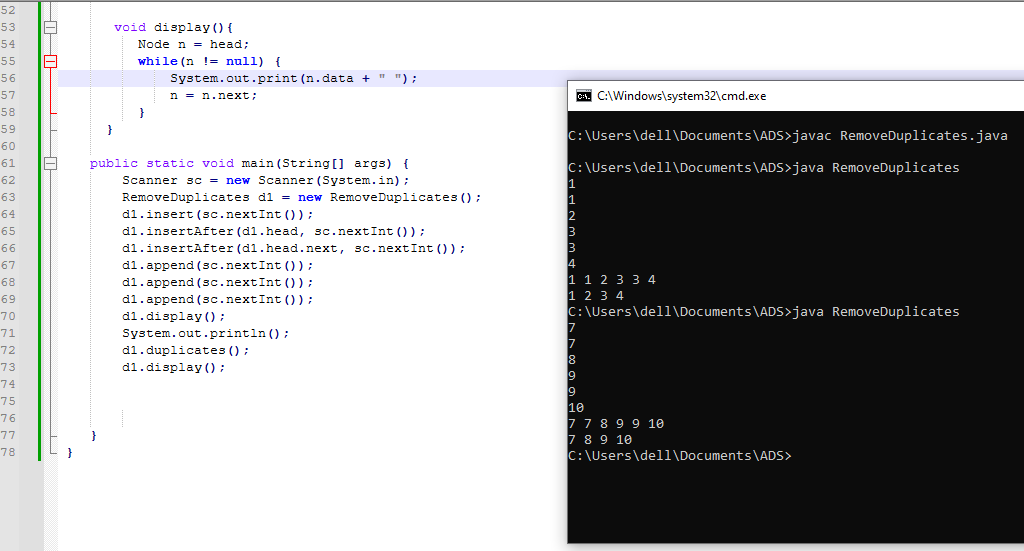
**Space Complexity** – O(n)

**6. Remove duplicates from a sorted linked list.**

* **Test Case 1**:  
  Input: List = [1, 1, 2, 3, 3, 4]  
  Output: List = [1, 2, 3, 4]
* **Test Case 2**:  
  Input: List = [7, 7, 8, 9, 9, 10]  
  Output: List = [7, 8, 9, 10]







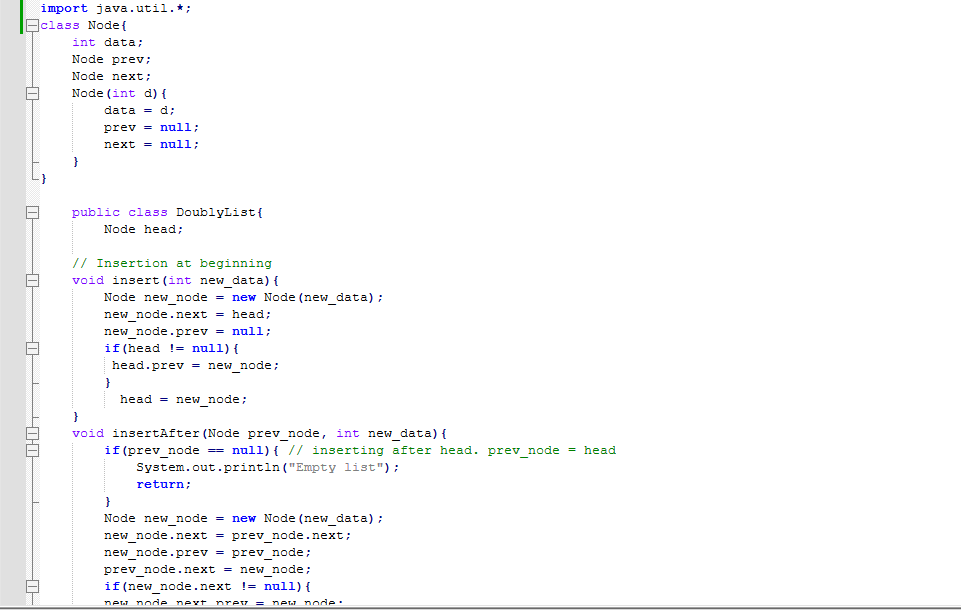
**Explanation -** The RemoveDuplicates class includes methods: like insert to add a new node at the beginning of the list, insertAfter to insert a new node after a specified node, and append to add a new node at the end of the list. The duplicates() method removes duplicate elements by traversing the list and comparing each node's data with the next node's data, skipping nodes with duplicate values. The display() method prints the current state of the linked list. In the main method, the program takes user input to create a linked list, displays the original list, removes duplicates, and displays the updated list.

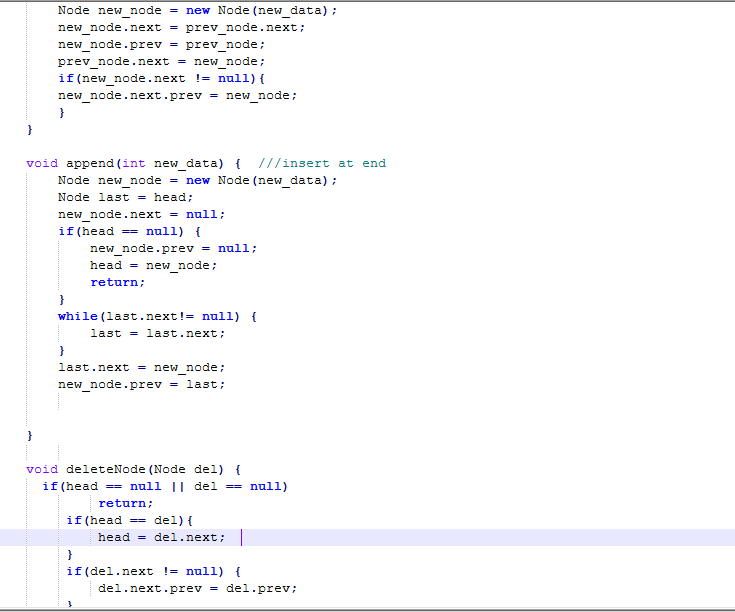
**Time Complexity –** O(n)

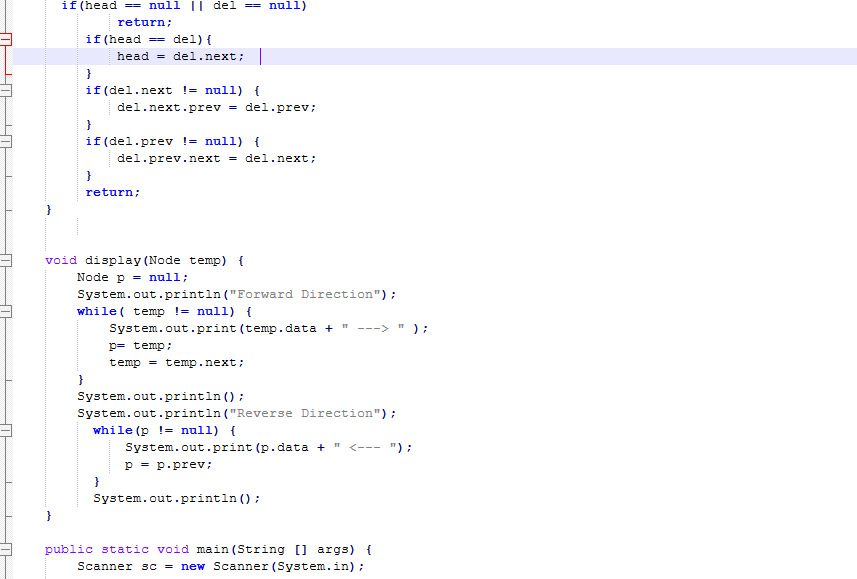
**Space Complexity** – O(n)

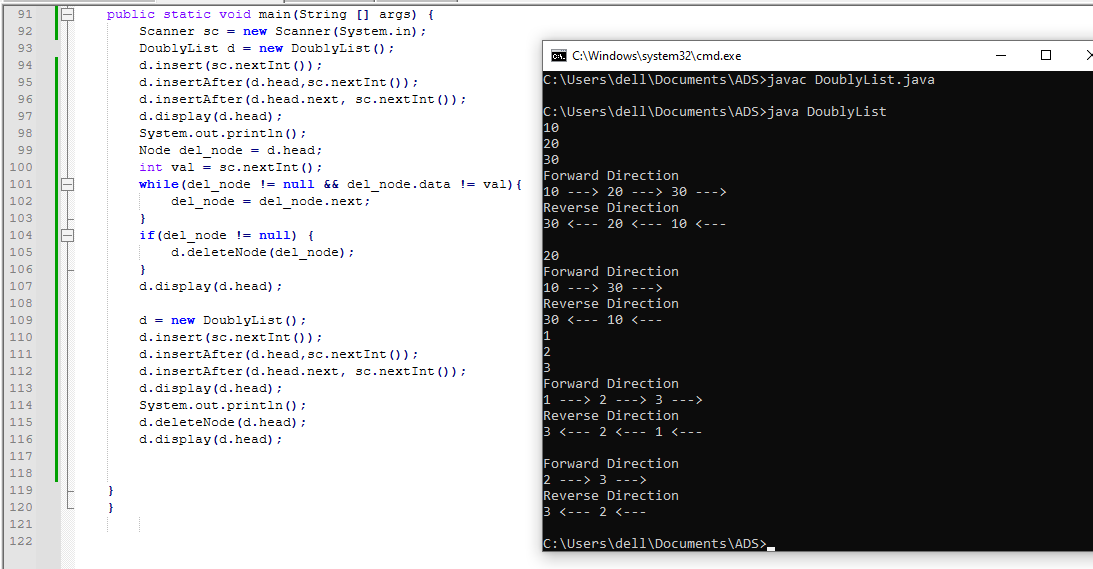
**7. Implement a doubly linked list with insert, delete, and traverse operations.**

* **Test Case 1**:  
  Input: Insert 10 → Insert 20 → Insert 30 → Delete 20  
  Output: List = [10, 30]
* **Test Case 2**:  
  Input: Insert 1 → Insert 2 → Insert 3 → Delete 1  
  Output: List = [2, 3]









**Flowchart-**

**Start**

**|**

**Input Values for First List**

**|**

**Insert Node 1**

**|**

**Insert Node 2**

**|**

**Insert Node 3**

**|**

**Display First List**

**|**

**Input Value to Delete**

**|**

**Initialize del\_node = d.head**

**|**

**Find Node to Delete**

**|**

**While del\_node != null and del\_node.data != input value**

**|**

**If del\_node != null**

**|**

**Call d.deleteNode(del\_node)**

**|**

**Display Updated First List (d.display(d.head))**

**|**

**Reset List**

**|**

**End**

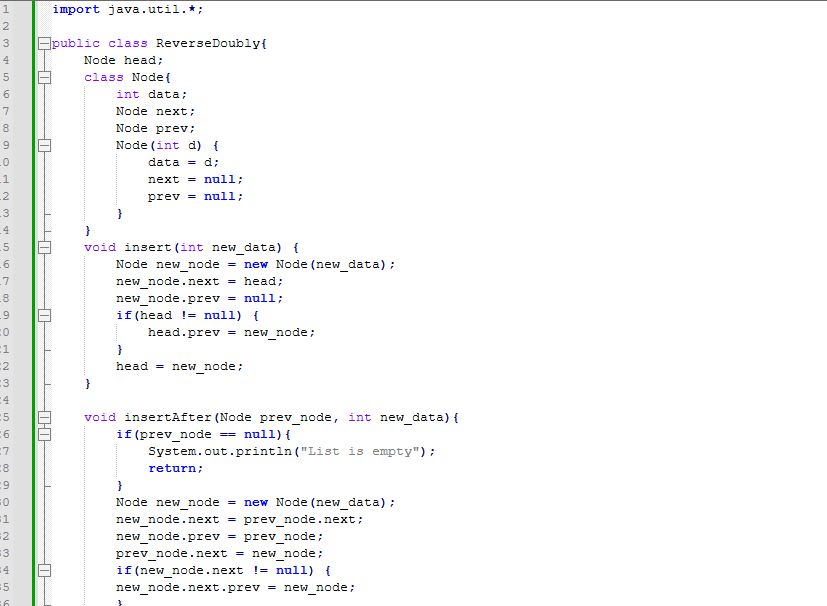
**Explanation –** In the DoublyList class there are methods for inserting nodes at the beginning inserting nodes after a specified node and appending nodes at the end . The deleteNode method removes a specified node from the list. The display method traverses the list, printing the elements in both forward and reverse order. In the main method, the program initializes a DoublyList, allows the user to input values to create the list and invoke the methods.

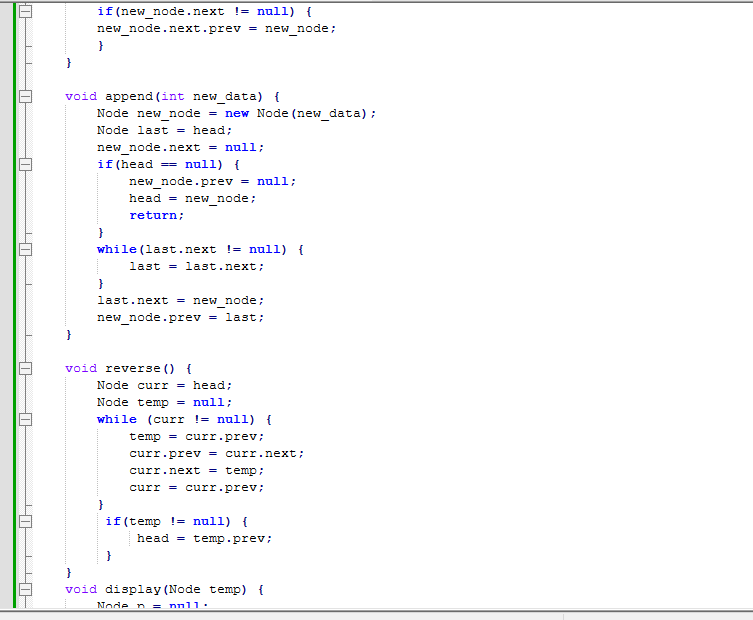
**Time Complexity –** O(n)

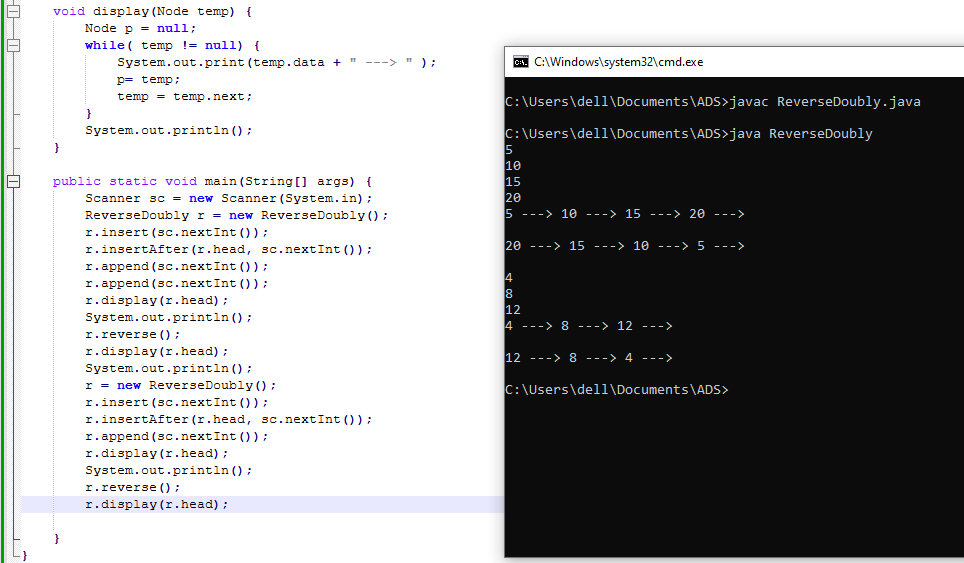
**Space Complexity** – O(n )

**8. Reverse a doubly linked list.**

* **Test Case 1**:  
  Input: List = [5, 10, 15, 20]  
  Output: List = [20, 15, 10, 5]
* **Test Case 2**:  
  Input: List = [4, 8, 12]  
  Output: List = [12, 8, 4]







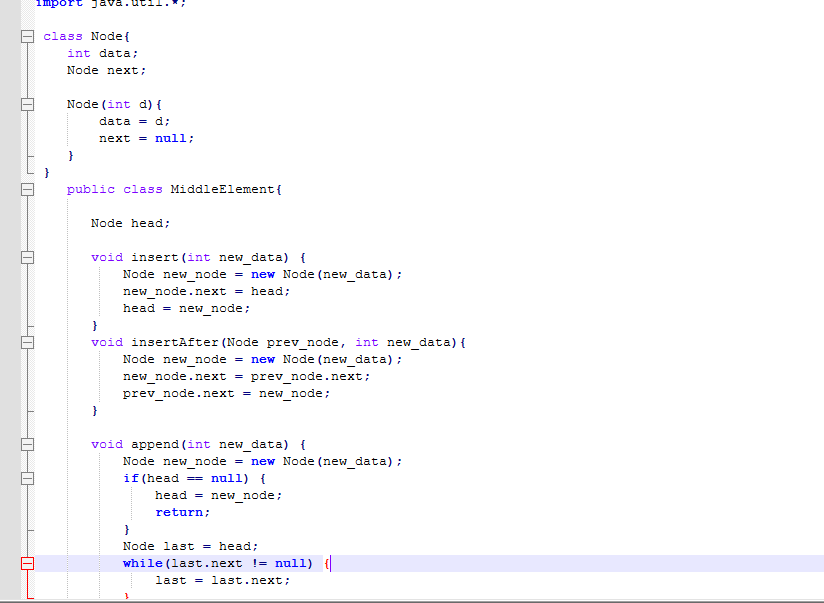
**Explanation –** The ReverseDoubly class includes methods like insert method which adds a new node at the beginning of the list. It adjusts the new node’s next pointer to point to the current head and, if the list is not empty, updates the previous head’s prev pointer to point back to the new node. It then assigns the new node as the new head of the list. The insertAfter method inserts a new node after a specified node . If prev\_node is null it prints a message and exits. Otherwise, it adjusts the next and prev pointers to correctly link the new node in the list. The append method adds a new node at the end of the list by traversing to the last node . The reverse method is designed to reverse the direction of the doubly linked list. It iterates through each node, swapping the next and prev pointers for each node to reverse the list. After the loop, the head is updated to point to the new head, which was previously the last node of the original list. The display method prints the data of each node in the list. In the main() method the user input values and the methods were invoked.

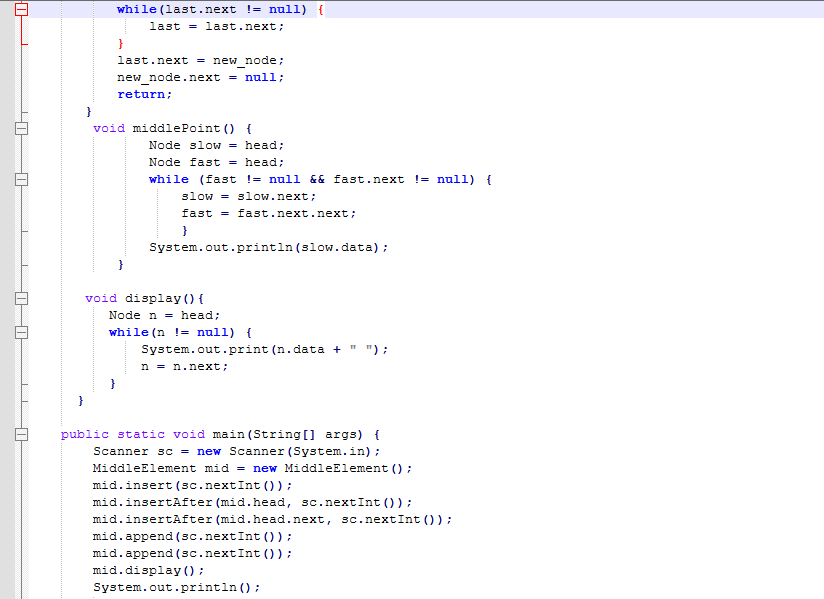
**Time Complexity –** O(n)

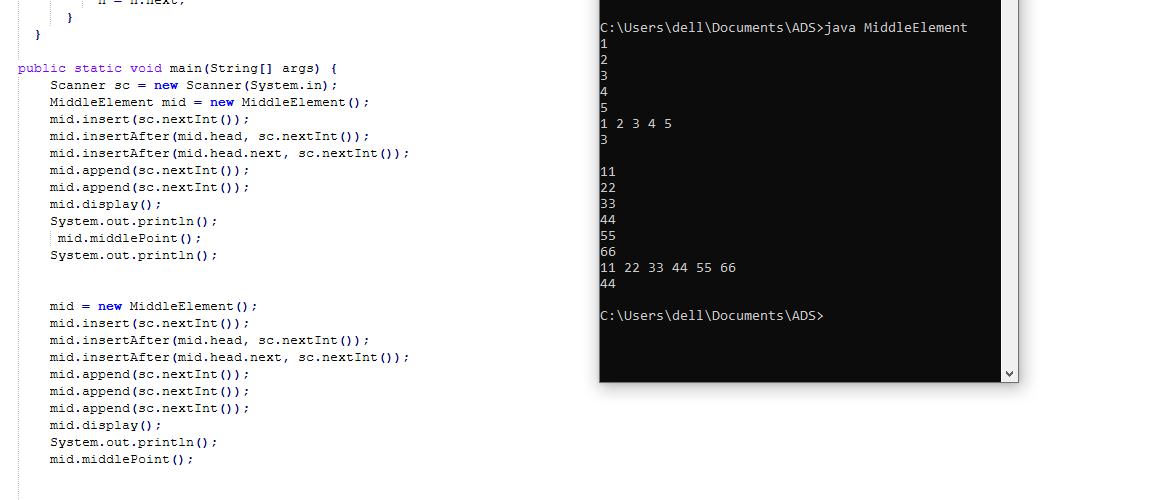
**Space Complexity** – O(1)

**15. Find the middle element of a linked list in one pass.**

* **Test Case 1**:  
  Input: List = [1, 2, 3, 4, 5]  
  Output: Middle = 3
* **Test Case 2**:  
  Input: List = [11, 22, 33, 44, 55, 66]  
  Output: Middle = 44







**Explanation –** In the main class, MiddleElement, there are methods like insert which adds a node at the head of the list, insertAfter which inserts a node after a given node, and append which adds a node at the end. To find the middle element of the list, the middlePoint method uses two pointers, a slow pointer that moves one node at a time and a fast pointer that moves two nodes at a time. When the fast pointer reaches the end, the slow pointer will be at the middle node. The display method is used to print all the elements in the list. In the main method, user inputs are taken and methods are invoked.

**Time Complexity –** O(n)

**Space Complexity** – O(n )