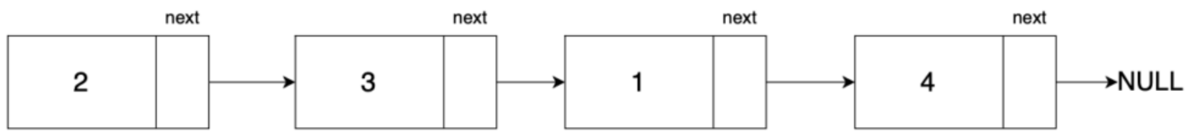
LinkedList Series

Contents

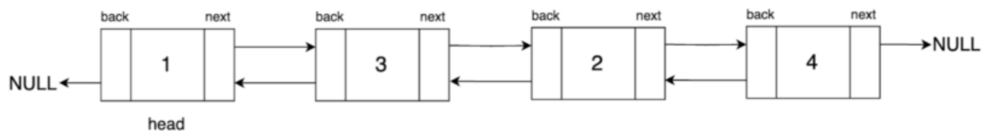
singly-linked list



A singly linked list

```
/**
 * Definition for singly-linked list.
 */

public class ListNode {
    int val;
    ListNode next;

    // Default constructor
    public ListNode() {}

    // Constructor with value
    public ListNode(int val) {
        this.val = val;
    }

    // Constructor with value and next node
    public ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }
}
```

doubly-linked list



A doubly linked list

```
/**
 * Definition for doubly-linked list.
 */
public class DoublyListNode {
    int val;
    DoublyListNode prev; // Reference to the previous node
    DoublyListNode next; // Reference to the next node

    // Default constructor
    public DoublyListNode() {}

    // Constructor with value
    public DoublyListNode(int val) {
        this.val = val;
    }

    // Constructor with value, previous node, and next node
    public DoublyListNode(int val, DoublyListNode prev,
DoublyListNode next) {
        this.val = val;
        this.prev = prev;
        this.next = next;
    }
}
```

Explanation:

- `val`: holds the data.
- `prev`: reference to the previous node.
- `next`: reference to the next node.
- Three constructors:
  - No-arg constructor (for flexibility)
  - Single-value constructor
  - Full constructor with `prev` and `next`

Circular Singly Linked List

```
/**
 * Definition for circular singly-linked list.
 */
public class CircularListNode {
    int val;
    CircularListNode next;

    // Default constructor
    public CircularListNode() {}

    // Constructor with value
    public CircularListNode(int val) {
        this.val = val;
    }

    // Constructor with value and next node
    public CircularListNode(int val, CircularListNode next) {
        this.val = val;
        this.next = next;
    }
}
```

Circular Behavior

In a circular singly linked list:

- The `next` pointer of the **last node** points back to the **head**, forming a loop.
- This structure enables continuous traversal without reaching `null`.

Optional: Method to Create Circular List from Array

```
public class CircularLinkedListUtil {
    public static CircularListNode createCircularList(int[]
values) {
        if (values.length == 0) return null;

        CircularListNode head = new CircularListNode(values[0]);
        CircularListNode current = head;

        for (int i = 1; i < values.length; i++) {
            current.next = new CircularListNode(values[i]);
            current = current.next;
        }

        current.next = head; // Complete the circle
        return head;
    }
}
```

Circular doubly linked list

Circular Doubly Linked List Node Class

```
/**
 * Definition for circular doubly-linked list node.
 */
public class CircularDoublyListNode {
    int val;
    CircularDoublyListNode prev;
    CircularDoublyListNode next;

    // Default constructor
    public CircularDoublyListNode() {}

    // Constructor with value
    public CircularDoublyListNode(int val) {
        this.val = val;
    }

    // Constructor with value, previous node, and next node
    public CircularDoublyListNode(int val,
CircularDoublyListNode prev, CircularDoublyListNode next) {
        this.val = val;
        this.prev = prev;
        this.next = next;
    }
}
```

Circular Behavior:

In a **circular doubly linked list**:

- The `next` pointer of the **last node** points to the **head**.
- The `prev` pointer of the **head** points to the **last node**.
- Allows **bi-directional traversal** in a circular loop.

Optional: Create Circular Doubly Linked List from Array

```
public class CircularDoublyLinkedListUtil {
    public static CircularDoublyListNode
createCircularDoublyList(int[] values) {
        if (values.length == 0) return null;
        CircularDoublyListNode head = new
CircularDoublyListNode(values[0]);
        CircularDoublyListNode prev = head;

        for (int i = 1; i < values.length; i++) {
            CircularDoublyListNode node = new
CircularDoublyListNode(values[i]);
            node.prev = prev;
            prev.next = node;
            prev = node;
        }

        // Closing the circle
        prev.next = head;
        head.prev = prev;

        return head;
    }
}
```

Features You Can Add:

- `insertAtHead`, `insertAtTail`
- `deleteNode`
- `displayForward` and `displayBackward`
- `search` or `length`

Goal: To reverse a singly-linked list.

Code Breakdown:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode node = null; // This will become the new head
of the reversed list

        while (head != null) {
            ListNode temp = head.next; // Store next node
            head.next = node;          // Reverse the link
            node = head;               // Move 'node' to current
'head'
            head = temp; // Move 'head' to the next node in the
original list
        }

        return node; // New head of the reversed list
    }
}
```

Key Variables:

- `head`: pointer to current node in the original list.
- `node`: builds the reversed list one node at a time.
- `temp`: temporarily stores the next node during iteration.

Example:

Input: `1 -> 2 -> 3 -> 4 -> 5 -> null`

Let's go through the loop:

1st Iteration:

- `head = 1, node = null`
- `temp = 2`
- `1.next = null` → reversed part is now `1 -> null`
- `node = 1`
- `head = 2`

2nd Iteration:

- `head = 2`, `node = 1 -> null`
- `temp = 3`
- `2.next = 1` → reversed part is now `2 -> 1 -> null`
- `node = 2`
- `head = 3`

3rd Iteration:

- `head = 3`, `node = 2 -> 1 -> null`
- `temp = 4`
- `3.next = 2` → `3 -> 2 -> 1 -> null`
- `node = 3`
- `head = 4`

4th Iteration:

- `head = 4`, `node = 3 -> 2 -> 1 -> null`
- `temp = 5`
- `4.next = 3` → `4 -> 3 -> 2 -> 1 -> null`
- `node = 4`
- `head = 5`

5th Iteration:

- `head = 5`, `node = 4 -> 3 -> 2 -> 1 -> null`
- `temp = null`
- `5.next = 4` → `5 -> 4 -> 3 -> 2 -> 1 -> null`
- `node = 5`
- `head = null` → loop ends

Final Output:

```
5 -> 4 -> 3 -> 2 -> 1 -> null
```

This is the reversed linked list returned by the function.

Summary:

The code **iteratively reverses a singly-linked list** using three pointers:

- One to store the next node (`temp`)
- One to reverse the pointer (`head.next = node`)
- One to build the new list (`node`)

Another Solution Using Given Constructors:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode node = null; // This will become the new head
of the reversed list

        while (head != null) {
            ListNode temp = head.next; // Store next node
            node =  new ListNode(head.val, node);

            //head.next = node;    // Reverse the link
            //node = head;         // Move 'node' to current
'head'
            head = temp; // Move 'head' to the next node in the
original list
        }

        return node; // New head of the reversed list
    }
}
```

Recursive version of reversing a singly linked list

Recursive Code to Reverse a Linked List:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        // Base case: empty list or single node
        if (head == null || head.next == null) {
            return head;
        }

        // Reverse the rest of the list recursively
        ListNode newHead = reverseList(head.next);

        // Reverse the current node's pointer
        head.next.next = head; // point the next node's next
back to current
        head.next = null;      // current node becomes the tail

        return newHead;        // return the new head node from
the bottom of recursion
    }
}
```

Explanation:

The recursion keeps going deeper until it reaches the end of the list, and then starts **rewiring the pointers** on the way back up the stack.

Example Input: **`1 -> 2 -> 3 -> 4 -> 5 -> null`**

Let’s trace what happens:

Step-by-Step Recursive Calls:

1. `reverseList(1)`
2. `reverseList(2)`
3. `reverseList(3)`
4. `reverseList(4)`
5. `reverseList(5)` → hits base case and returns `5`

Now the stack unwinds and re-links pointers:

- `4.next.next = 4` → `5 -> 4`, then `4.next = null`
- `3.next.next = 3` → `4 -> 3`, then `3.next = null`
- `2.next.next = 2` → `3 -> 2`, then `2.next = null`
- `1.next.next = 1` → `2 -> 1`, then `1.next = null`

Final Output:

```
5 -> 4 -> 3 -> 2 -> 1 -> null
```

Visual Summary:

```
Before:
1 → 2 → 3 → 4 → 5 → null

After:
5 → 4 → 3 → 2 → 1 → null
```

Reverse a Doubly Linked List

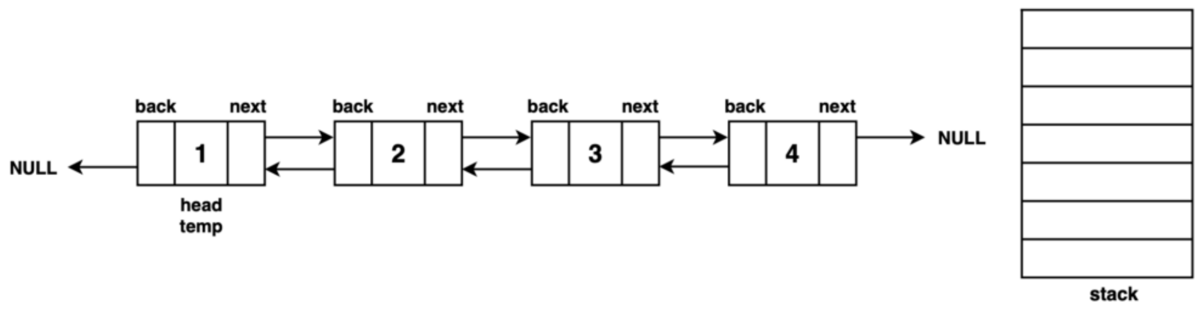
Problem Statement: Given a doubly linked list of size ‘N’ consisting of positive integers, your task is to reverse it and return the head of the modified doubly linked list.

Brute-Force approach

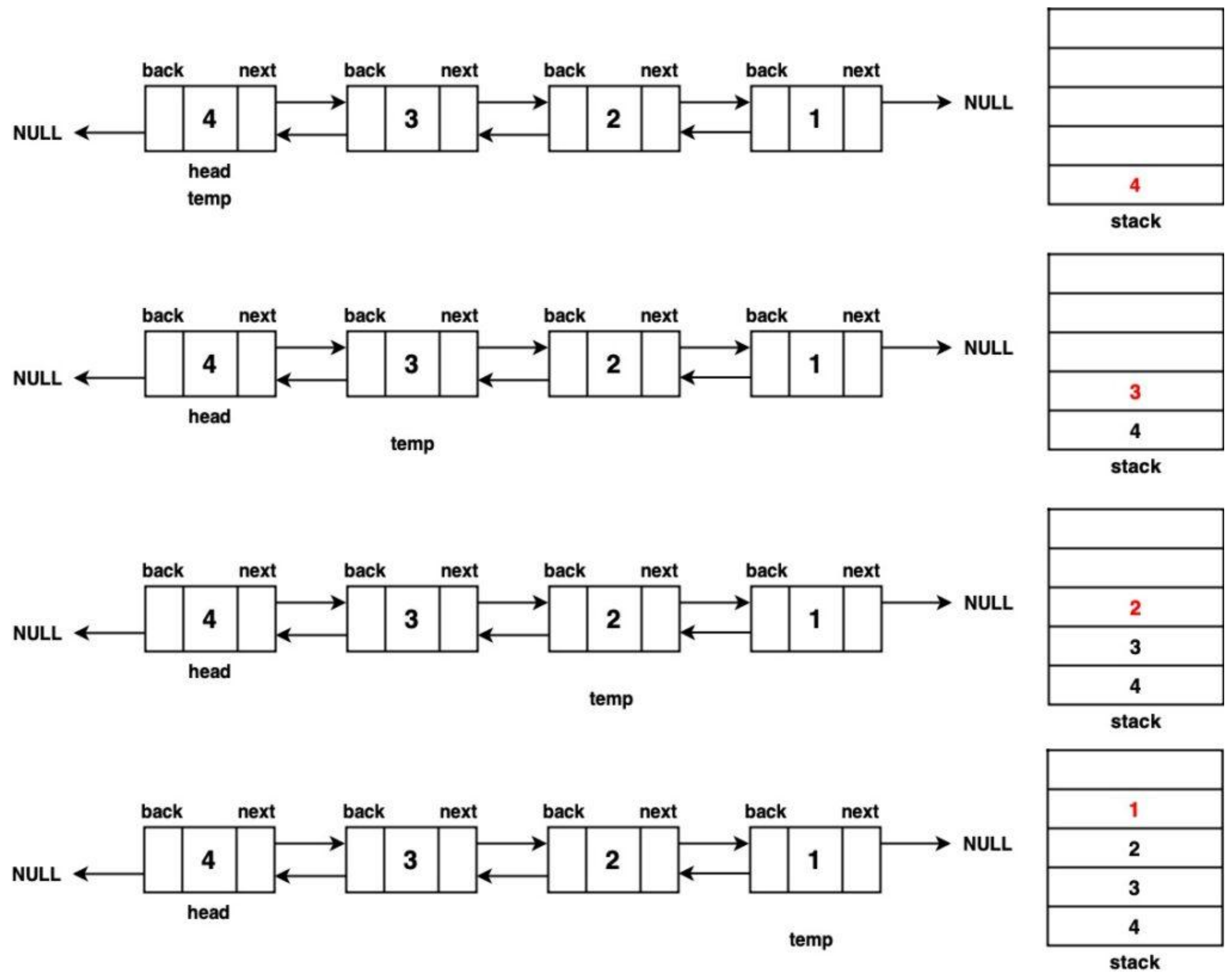
A brute-force approach involves replacing data in a doubly linked list. First, we traverse the list and store node data in a stack. Then, in a second pass, we assign elements from the stack to nodes, ensuring a reverse order replacement since stacks follow the Last-In-First-Out (LIFO) principle.

Algorithm:

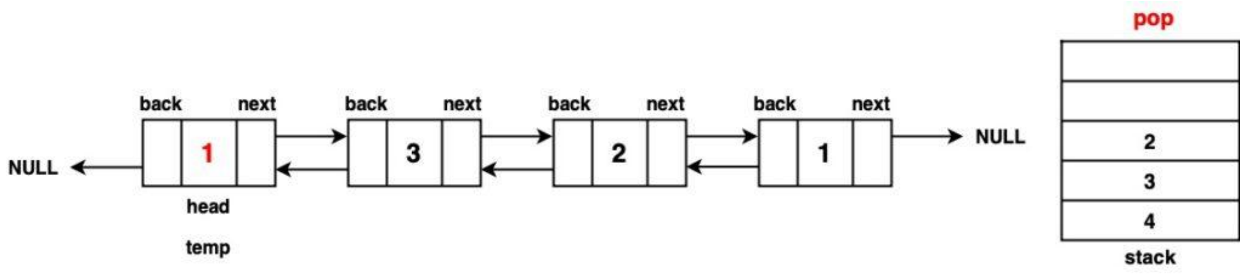
Step 1: Initialization a temp pointer to the head of the doubly linked list and a stack data structure to store the values from the list.

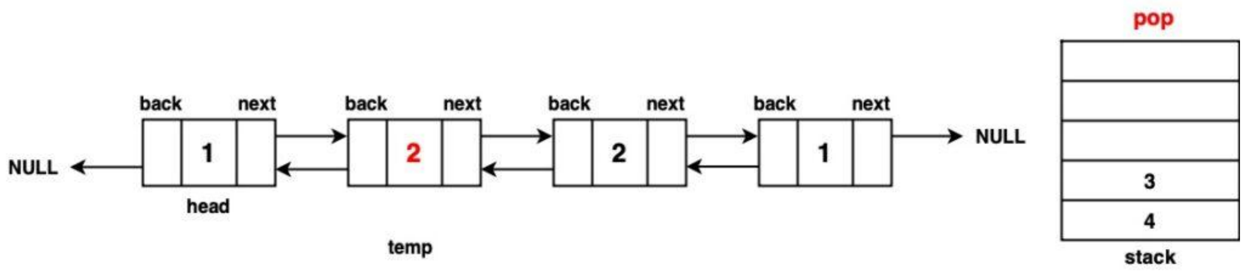


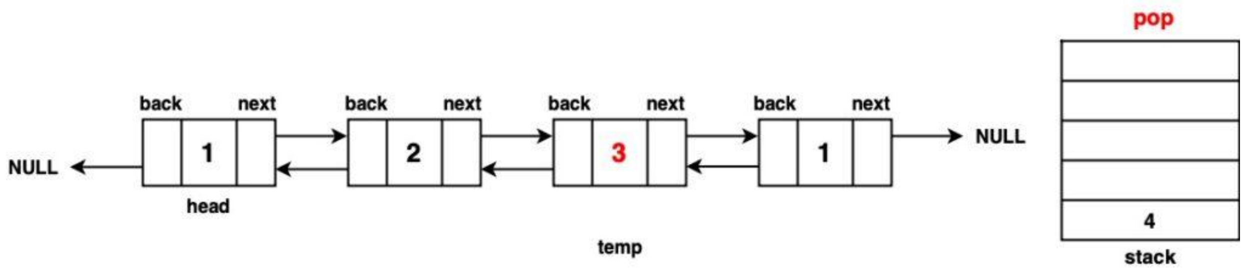
Step 2: Traverse the doubly linked list with the temp pointer and while traversing push the value at the current node temp onto the stack. Move the temp to the next node continuing until temp reaches null indicating the end of the list.

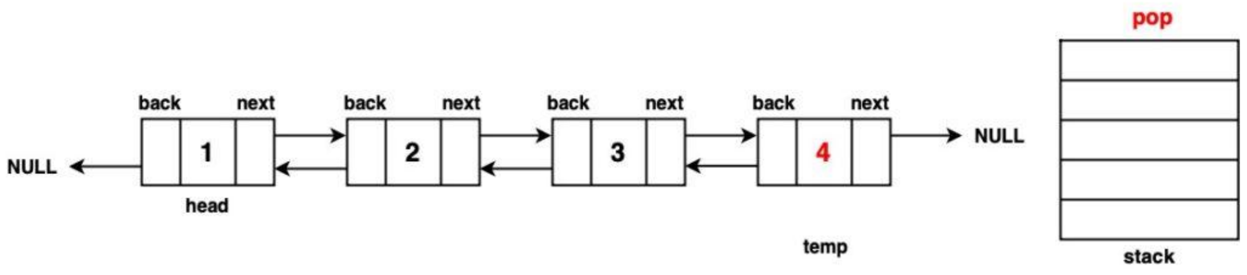


Step 3: Reset the temp pointer back to the head of the list and in this second iteration pop the element from the stack, replace the data at the current node with the popped value from the top of the stack and move temp to the next node. Repeat this step until temp reaches null or the stack becomes empty.









Code:

TRY IT YOURSELF.

Link: https://www.geeksforgeeks.org/problems/reverse-a-doubly-linked-list/1

```
/*
class DLLNode {
    int data;
    DLLNode next;
    DLLNode prev;

    DLLNode(int val) {
        data = val;
        next = null;
        prev = null;
    }
}
*/
class Solution {

    public DLLNode reverseDLL(DLLNode head) {
        // Your code here
        if(head == null || head.next == null){
            return head;
        }

        DLLNode back = null;
        DLLNode current = head;
        while(current != null){

            back = current.prev;

            current.prev = current.next;
            current.next = back;

            current = current.prev;

        }
        return back.prev;
    }
}
```

Link: https://takeuforward.org/plus/dsa/problems/reverse-a-doubly-linked-list

```
class Solution {
    public ListNode reverseDLL(ListNode head) {
        // Your code goes here
        if(head == null || head.next == null){
            return head;
        }

        ListNode back = null;
        ListNode current = head;

        while( current != null){

            back = current.prev;

            current.prev = current.next;
            current.next = back;

            current = current.prev;
        }
        return back.prev;

    }
}
```

Floyd’s cycle detection

Detection of loop (for general lists) or Floyd’s cycle detection implementation

Floyd’s Cycle Detection Algorithm (also known as the **Tortoise and Hare algorithm**)

It is used to detect a **loop in a singly linked list**.

```
/**
 * Definition for singly-linked list.
 */
public class ListNode {
    int val;
    ListNode next;

    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next
= next; }
}

class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) return false;

        ListNode slow = head;       // Moves one step at a time
        ListNode fast = head.next;  // Moves two steps at a time

        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false; // Reached the end — no cycle
            }
            slow = slow.next;
            fast = fast.next.next;
        }

        return true; // Cycle detected
    }
}
```

How it Works (Concept)

- Two pointers:
  - **Slow** pointer moves one step at a time.
  - **Fast** pointer moves two steps at a time.

- If there is **no loop**, the fast pointer will eventually reach the end (`null`).
- If there **is a loop**, the fast pointer will eventually “lap” the slow pointer and they’ll meet.

Example:

Linked list:
`1 → 2 → 3 → 4 → 5 → 3` (loop starts at node 3)

- Slow moves: `1 → 2 → 3 → 4`
- Fast moves: `1 → 3 → 5 → 4`
- Eventually: `slow == fast` at node `4` → **loop detected**

Bonus – Find the Start of the Loop:

```
public ListNode detectCycle(ListNode head) {
    ListNode slow = head, fast = head;

    // First detect if a cycle exists
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            // Cycle found, now find entry point
            ListNode entry = head;
            while (entry != slow) {
                entry = entry.next;
                slow = slow.next;
            }
            return entry; // Start of the cycle
        }
    }

    return null; // No cycle
}
```

LinkedList in Java

Source: https://www.geeksforgeeks.org/java/linked-list-in-java/

```
LinkedList<String> l = new LinkedList<String>();
```

Methods for Java LinkedList

Method	Description
add(int index, E element)	This method Inserts the specified element at the specified position in this list.
add(E e)	This method Appends the specified element to the end of this list.
addAll(int index, Collection<E> c)	This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
addAll(Collection<E> c)	This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
addFirst(E e)	This method Inserts the specified element at the beginning of this list.
addLast(E e)	This method Appends the specified element to the end of this list.
clear()	This method removes all of the elements from this list.

Method	Description
clone()	This method returns a shallow copy of this LinkedList.
contains(Object o)	This method returns true if this list contains the specified element.
descendingIterator()	This method returns an iterator over the elements in this deque in reverse sequential order.
element()	This method retrieves but does not remove, the head (first element) of this list.
get(int index)	This method returns the element at the specified position in this list.
getFirst()	This method returns the first element in this list.
getLast()	This method returns the last element in this list.
indexOf(Object o)	This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
lastIndexOf(Object o)	This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

Method	**Description**
listIterator(int index)	This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
offer(E e)	This method Adds the specified element as the tail (last element) of this list.
offerFirst(E e)	This method Inserts the specified element at the front of this list.
offerLast(E e)	This method Inserts the specified element at the end of this list.
peek()	This method retrieves but does not remove, the head (first element) of this list.
peekFirst()	This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
peekLast()	This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
poll()	This method retrieves and removes the head (first element) of this list.
pollFirst()	This method retrieves and removes the first element of this list, or returns null if this list is empty.

Method	Description
pollLast()	This method retrieves and removes the last element of this list, or returns null if this list is empty.
pop()	This method Pops an element from the stack represented by this list.
push(E e)	This method pushes an element onto the stack represented by this list.
remove()	This method retrieves and removes the head (first element) of this list.
remove(int index)	This method removes the element at the specified position in this list.
remove(Object o)	This method removes the first occurrence of the specified element from this list if it is present.
removeFirst()	This method removes and returns the first element from this list.
removeFirstOccurrence(Object o)	This method removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
removeLast()	This method removes and returns the last element from this list.
removeLastOccurrence(	This method removes the last occurrence of the specified

Method	Description
Object o)	element in this list (when traversing the list from head to tail).
set(int index, E element)	This method replaces the element at the specified position in this list with the specified element.
size()	This method returns the number of elements in this list.
spliterator()	This method creates a late-binding and fail-fast Spliterator over the elements in this list.
toArray()	This method returns an array containing all of the elements in this list in proper sequence (from first to last element).
toArray(T[] a)	This method returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
toString()	This method returns a string containing all of the elements in this list in proper sequence (from first to the last element), each element is separated by commas and the String is enclosed in square brackets.