



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Nov 22, 2025 

Contributed by



DsaA Lab 2 - Linked Lists

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

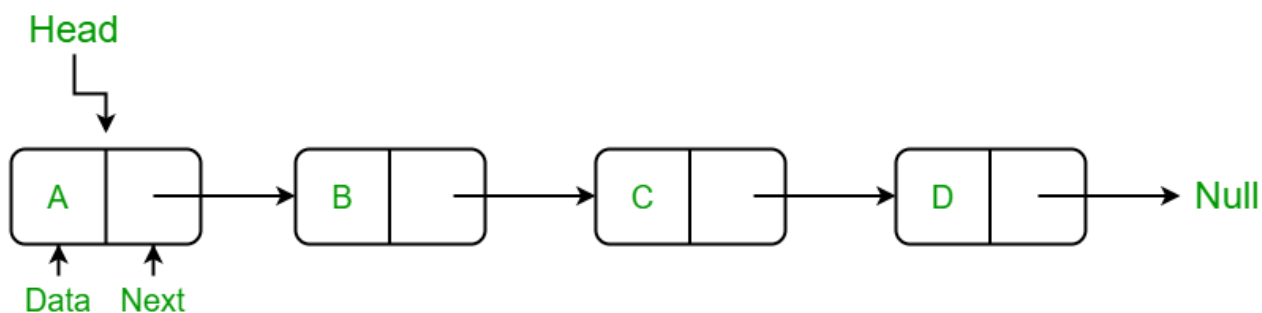
Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to learn about Linked Lists. What are they, how can we implement them, their key terminology and types, and how they can be utilized to solve interesting problems and to implement other complex data structures.

Definition

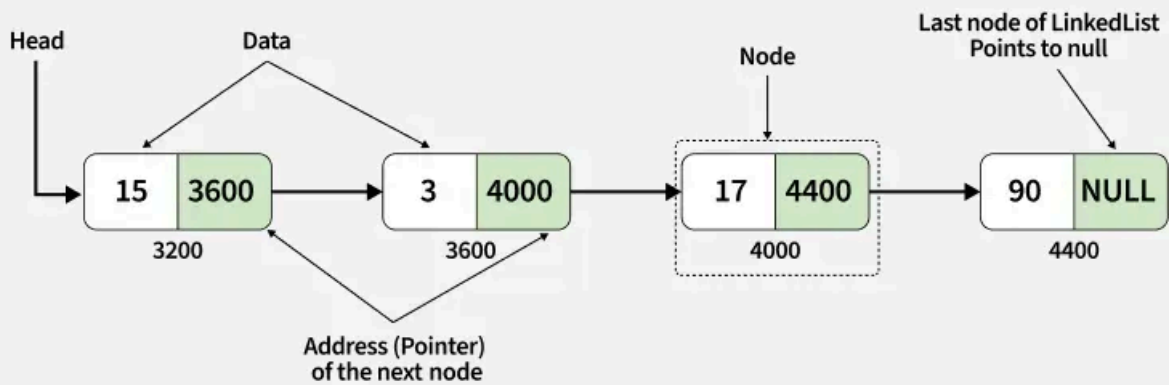
What is a Linked List?

A linked list is a linear data structure that stores elements in a sequence of chained nodes. Each node is an object that has 2 attributes: the data it holds, and a pointer to the next node.



Key Terminology

To deal with any data structure, you should understand its terminology first:



- **A Node:** A node is an object which holds 2 values:

1. **The data it holds** (String, int, Object, etc.)
2. **A pointer to the next (or previous) Node**

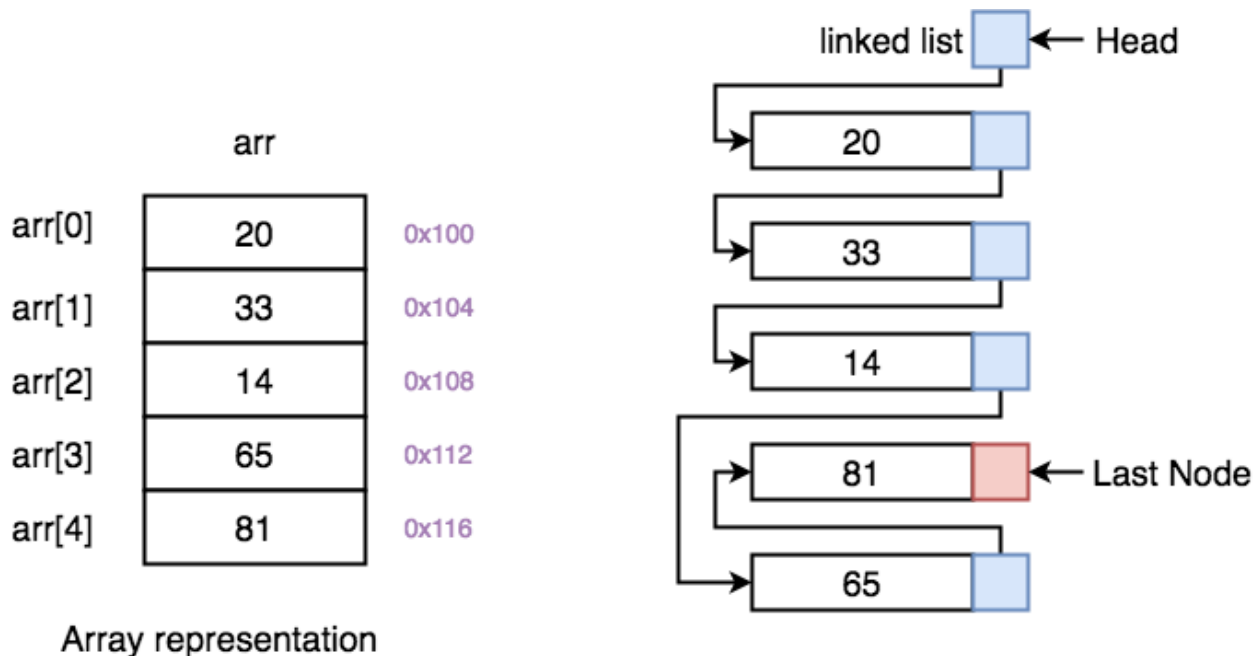
```

1  class Node<T> {
2      T data;
3      Node<T> next;
4
5      public Node(T data) {
6          this.data = data;
7          this.next = null;
8      }
9
10     public Node<T> getNext(){
11         return next;
12     }
13
14     public void setNext(Node<T> node){
15         this.next = node;
16     }
17
18 }

```

- **The Head:** the head of the linked list is basically a pointer to its first node, which serves as an entry point for traversing the linked list and performing operations on its elements
- **The Tail:** the tail of the linked list is a pointer to its last element. In some implementations, especially in singly linked lists, this pointer is not included, but in other types of linked lists where traversal often happens from either end of the list, its crucial to include it to improve efficiency.

How is it different from an array?



The key difference between arrays and linked lists is that arrays store elements in contiguous memory locations, allowing for constant-time access $O(1)$ to any element by index.

In contrast, the nodes of a linked list are not stored in a contiguous fashion, but rather in a series of chained objects, where each object has a pointer that points to the next node (where each node is stored on the heap is an advanced topic which is beyond the scope of this course). This makes it costly to access a specific node, which, in the worst case scenario (accessing the last element without having a `tail` pointer), could take up to $O(n)$, where n is the length of the linked list.

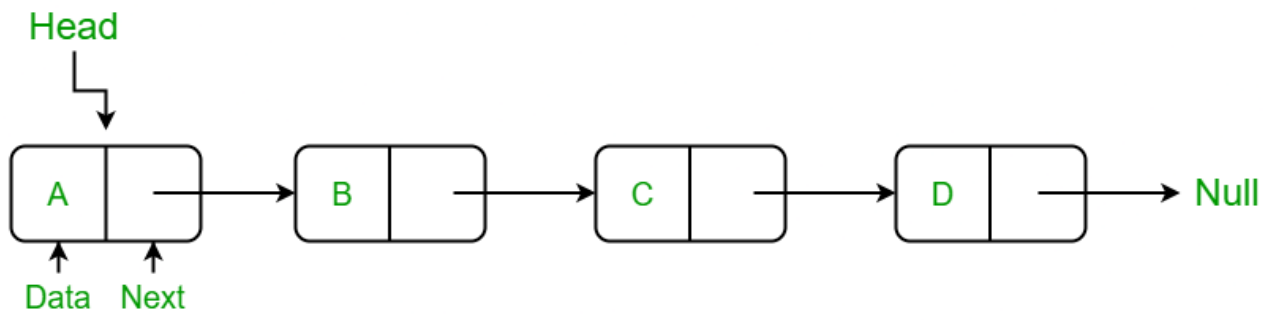
However, linked lists have significant advantages in specific operations. For example, removing the first element in the list could simply be done by advancing the head pointer to the second element, effectively removing the first element. The garbage collector will take care of freeing the deleted node's memory.

Types of Linked Lists

Singly Linked List

Its a linked list where each node has only **one pointer** to the next node, meaning it can only be accessed from one end and traversed in one direction.

By convention, the `next` pointer of the last node is set to `null`, indicating the end of the list.



Implementation in Java:


```
1 public class SinglyLinkedList<E> {
2
3     private static class Node<E> {
4         private E element;
5         private Node<E> next;
6
7         public Node(E e, Node<E> n) {
8             element = e;
9             next = n;
10        }
11
12        public E getElement() {
13            return element;
14        }
15
16        public Node<E> getNext() {
17            return next;
18        }
19
20        public void setNext(Node<E> n) {
21            next = n;
22        }
23    }
24
25    private Node<E> head = null;
26    private Node<E> tail = null;
27    private int size = 0;
28
29    public SinglyLinkedList() {}
30
31    public int size() {
32        return size;
33    }
34
35    public boolean isEmpty() {
36        return size == 0;
37    }
38
39    public E first() {
40        if (isEmpty()) return null;
41        return head.getElement();
42    }
43
44    public E last() {
45        if (isEmpty()) return null;
46        return tail.getElement();
47    }
48
49    public void addFirst(E e) {
50        head = new Node<>(e, head);
51        if (size == 0)
52            tail = head;
53        size++;
54    }
55
56    public void addLast(E e) {
57        Node<E> newest = new Node<>(e, null);
58        if (isEmpty())
```

```

59         head = newest;
60     else
61         tail.setNext(newest);
62     tail = newest;
63     size++;
64 }
65
66 public E removeFirst() {
67     if (isEmpty()) return null;
68     E answer = head.getElement();
69     head = head.getNext();
70     size--;
71     if (size == 0)
72         tail = null;
73     return answer;
74 }
75
76 @Override
77 public String toString() {
78     StringBuilder sb = new StringBuilder("");
79     Node<E> walk = head;
80     while (walk != null) {
81         sb.append(walk.getElement());
82         if (walk.getNext() != null)
83             sb.append(", ");
84         walk = walk.getNext();
85     }
86     sb.append("");
87     return sb.toString();
88 }
89 }

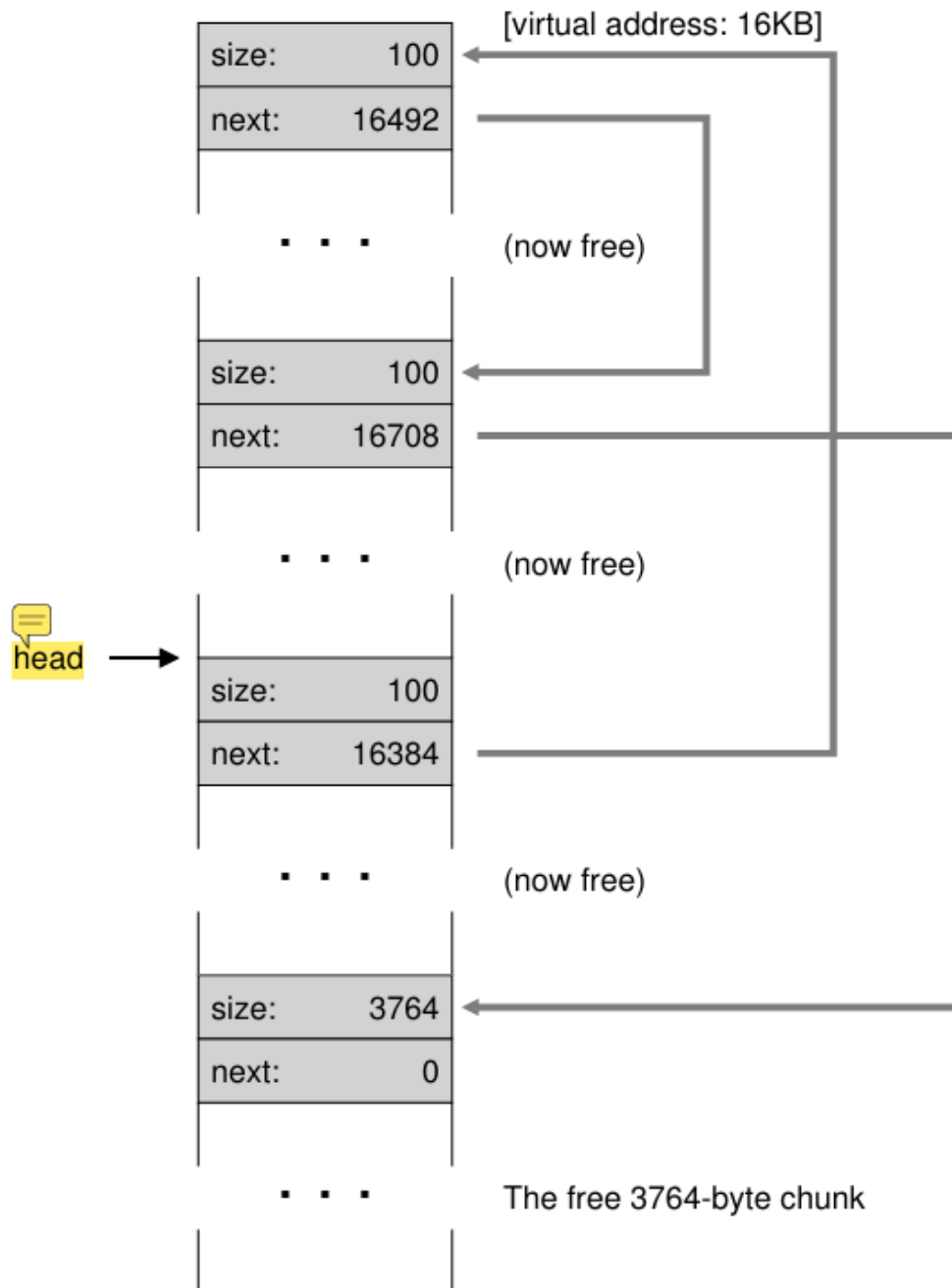
```

Some Real World Applications

Singly linked lists are widely used to solve a diverse set of problems. One of their main use cases is to implement other data structures such as stacks, which we will see in upcoming labs.

Furthermore, In operating systems, imagine your computer has a pool of free memory blocks, and you need to keep track of which parts of memory are available for use. We can implement a **free list** structure, where each node represents a free memory block.

Each node stores the size of the memory block, and a pointer to the next free memory block:



Doubly Linked List

A doubly linked list differs from a singly linked list in one key aspect, which is that each node has **2 pointers**: `next`, which points to the next node, and `previous`, which points to the previous node. This allows traversing the list in both directions, however, this doesn't mean we should use it always over a singly linked list, because it requires extra memory for the additional pointer and slightly more complex pointer updates during insertions and deletions.

Implementation in Java


```
1  class DoublyLinkedList<E> {
2
3      public static class Node<E> {
4
5          private E element;
6          private Node<E> prev;
7          private Node<E> next;
8
9          public Node(E e, Node<E> p, Node<E> n) {
10             element = e;
11             prev = p;
12             next = n;
13         }
14
15         public E getElement() {
16             return element;
17         }
18
19         public void setElement(E element) {
20             this.element = element;
21         }
22
23         public Node<E> getPrev() {
24             return prev;
25         }
26
27         public Node<E> getNext() {
28             return next;
29         }
30
31         public void setPrev(Node<E> p) {
32             prev = p;
33         }
34
35         public void setNext(Node<E> n) {
36             next = n;
37         }
38     }
39
40     private Node<E> header;
41     private Node<E> trailer;
42     private int size = 0;
43
44     public DoublyLinkedList() {
45         header = new Node<>(null, null, null);
46         trailer = new Node<>(null, header, null);
47         header.setNext(trailer);
48     }
49
50     public int size() {
51         return size;
52     }
53
54     public boolean isEmpty() {
55         return size == 0;
56     }
57
58     public Node<E> getHead() {
```

```
59     if (isEmpty()) {
60         return null;
61     }
62     return header.getNext();
63 }
64
65 public Node<E> getTail() {
66     if (isEmpty()) {
67         return null;
68     }
69     return trailer.getPrev();
70 }
71
72 public E first() {
73     if (isEmpty()) {
74         return null;
75     }
76     return header.getNext().getElement();
77 }
78
79 public E last() {
80     if (isEmpty()) {
81         return null;
82     }
83     return trailer.getPrev().getElement();
84 }
85
86 public void addFirst(E e) {
87     addBetween(e, header, header.getNext());
88 }
89
90 public void addLast(E e) {
91     addBetween(e, trailer.getPrev(), trailer);
92 }
93
94 public E removeFirst() {
95     if (isEmpty()) {
96         return null;
97     }
98     return remove(header.getNext());
99 }
100
101 public E removeLast() {
102     if (isEmpty()) {
103         return null;
104     }
105     return remove(trailer.getPrev());
106 }
107
108 private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
109     Node<E> newest = new Node<>(e, predecessor, successor);
110     predecessor.setNext(newest);
111     successor.setPrev(newest);
112     size++;
113 }
114
115 public E remove(Node<E> node) {
116     Node<E> predecessor = node.getPrev();
```

```

116     Node<E> predecessor = node.getPrev();
117     Node<E> successor = node.getNext();
118     predecessor.setNext(successor);
119     successor.setPrev(predecessor);
120     size--;
121     return node.getElement();
122 }
123
124 @Override
125 public String toString() {
126     StringBuilder sb = new StringBuilder("(");
127     Node<E> walk = header.getNext();
128     while (walk != trailer) {
129         sb.append(walk.getElement());
130         if (walk.getNext() != trailer) {
131             sb.append(", ");
132         }
133         walk = walk.getNext();
134     }
135     sb.append(")");
136     return sb.toString();
137 }
138 }
139

```

Real World Example

Doubly linked lists are used in a wide range of applications. For example, in text editors, a doubly linked list can be used to store characters or lines of text.

Each node represents a character (or line) and contains pointers to both its previous and next nodes. This structure allows the cursor to move efficiently in both directions and makes insertions or deletions at arbitrary positions extremely fast, only pointer adjustments are needed.

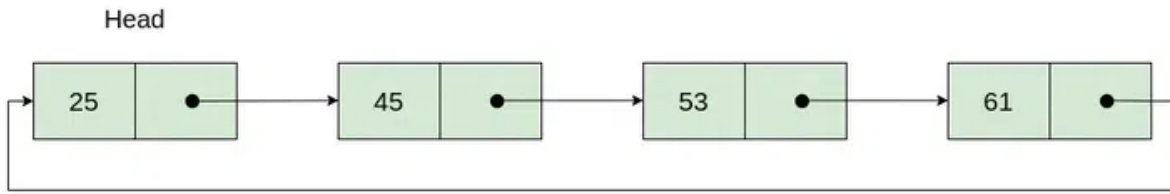
For instance, when a user types or deletes a character, the editor can simply insert or remove a node at the cursor position without having to shift large blocks of text, as would be required in an array-based approach.

Circular Linked List

A circular linked list is a variation of a linked list in which the last node's pointer does not point to `null`, but instead points back to the first node.

This creates a circular chain of nodes, allowing continuous traversal through the list without ever reaching an end.

Circular Linked List



Circular linked lists can be singly or doubly linked, and they are often used in applications that require cyclic iteration, such as round-robin scheduling or buffering systems.

Implementation in Java


```
1 public class CircularlyLinkedList<E> {
2
3     private static class Node<E> {
4         private E element;
5         private Node<E> next;
6
7         public Node(E e, Node<E> n) {
8             element = e;
9             next = n;
10        }
11
12        public E getElement() {
13            return element;
14        }
15
16        public Node<E> getNext() {
17            return next;
18        }
19
20        public void setNext(Node<E> n) {
21            next = n;
22        }
23    }
24
25    private Node<E> tail = null;
26    private int size = 0;
27
28    public CircularlyLinkedList() {}
29
30    public int size() {
31        return size;
32    }
33
34    public boolean isEmpty() {
35        return size == 0;
36    }
37
38    public E first() {
39        if (isEmpty()) return null;
40        return tail.getNext().getElement(); // head
41    }
42
43    public E last() {
44        if (isEmpty()) return null;
45        return tail.getElement();
46    }
47
48    public void rotate() {
49        if (tail != null)
50            tail = tail.getNext();
51    }
52
53    public void addFirst(E e) {
54        if (isEmpty()) {
55            tail = new Node<>(e, null);
56            tail.setNext(tail);
57        } else {
58            Node<E> newest = new Node<>(e, tail.getNext());
```

```

59     tail.setNext(newest);
60 }
61 size++;
62 }
63
64 public void addLast(E e) {
65     addFirst(e);
66     tail = tail.getNext();
67 }
68
69 public E removeFirst() {
70     if (isEmpty()) return null;
71     Node<E> head = tail.getNext();
72     if (head == tail)
73         tail = null;
74     else
75         tail.setNext(head.getNext());
76     size--;
77     return head.getElement();
78 }
79
80 @Override
81 public String toString() {
82     if (isEmpty()) return "()";
83     StringBuilder sb = new StringBuilder("");
84     Node<E> walk = tail.getNext();
85     for (int i = 0; i < size; i++) {
86         sb.append(walk.getElement());
87         if (i < size - 1)
88             sb.append(", ");
89         walk = walk.getNext();
90     }
91     sb.append(")");
92     return sb.toString();
93 }
94 }

```

Some Real world applications

Circular linked lists are commonly used in situations where data needs to be processed in a continuous, cyclic manner.

A classic example is round-robin scheduling in operating systems, where each process is given an equal time slice in a repeating cycle.

Another example could be a game where players take turns. Each player is a node in a circular linked list, and the turn rotates continuously from one player to the next, looping back to the first player after the last one.

Tasks

Task 1 (6 points)

For this task, you are going to build a class called `LinkedListString` which represents a string using a doubly linked list.

Your class should have the following 2 constructors:

- `LinkedListString()` Initializes an empty linked string
- `LinkedListString(String str)` Initializes the linked string from a given `String` object

Your class should have the following methods:

- `size()` Returns the length of the linked string
- `toString()` Converts the linked string into a regular Java `String` object
- `reverse()` Reverses the linked string in-place, without creating a new list
- `deleteCharAt(index i)` Deletes the character at the specified index `i`
- `append(String str)` Appends the given `String str` to the linked string
- `append(LinkedListString str)` Appends the given `LinkedListString str` to the linked string

Use the following code to implement your solution:

```
1
2  /* make sure to add both this package identifier
3   * and the import statement before you submit your solution
4   * to GradeScope. Remove them while testing locally */
5  package com.gradescope.labTwo;
6  import com.gradescope.labTwo.tests.DoublyLinkedList;
7
8  public class LinkedString {
9
10     private final DoublyLinkedList<Character> list;
11
12     public LinkedString(){
13         this.list = new DoublyLinkedList<>();
14     }
15
16     public LinkedString(String str){
17         this.list = new DoublyLinkedList<>();
18         // continue implementing this constructor
19     }
20
21     public int size(){
22         // implement this method
23         return 0;
24     }
25
26     public void append(String str){
27         // implement this method
28     }
29
30     public void append(LinkedString str){
31         // implement this method
32     }
33
34     public DoublyLinkedList.Node<Character> getFirstCharacterNode(){
35         // implement this method
36         return null; // just a dummy return
37     }
38
39     public void reverse(){
40         // implement this method
41     }
42
43     public void deleteCharAt(int index){
44         // implement this method
45     }
46
47     @Override
48     public String toString(){
49         // implement this method
50         return null; // just a dummy return
51     }
52 }
```

Use the following code to test your solution:


```
1 public class LinkedStringTests {
2     public static void main(String[] args) {
3
4         System.out.println("==== TEST 1: Creating empty LinkedString =====");
5         LinkedString empty = new LinkedString();
6         System.out.println("Expected size = 0");
7         System.out.println("Actual size   = " + empty.size());
8         System.out.println();
9
10        System.out.println("==== TEST 2: Creating from normal String =====");
11        LinkedString hello = new LinkedString("Hello");
12        System.out.println("Expected size = 5");
13        System.out.println("Actual size   = " + hello.size());
14        System.out.println("Expected string = Hello");
15        System.out.println("Actual string  = " + hello.toString());
16        System.out.println();
17
18        System.out.println("==== TEST 3: append(String) =====");
19        hello.append(" World");
20        System.out.println("Expected string = Hello World");
21        System.out.println("Actual string   = " + hello.toString());
22        System.out.println("Expected size = 11");
23        System.out.println("Actual size    = " + hello.size());
24        System.out.println();
25
26        System.out.println("==== TEST 4: append(LinkedString) =====");
27        LinkedString exclamation = new LinkedString("!");
28        hello.append(exclamation);
29        System.out.println("Expected string = Hello World!");
30        System.out.println("Actual string   = " + hello.toString());
31        System.out.println("Expected size = 12");
32        System.out.println("Actual size    = " + hello.size());
33        System.out.println();
34
35        System.out.println("==== TEST 5: reverse() =====");
36        hello.reverse();
37        System.out.println("Expected string = !dlroW olleH");
38        System.out.println("Actual string   = " + hello.toString());
39        System.out.println();
40
41        System.out.println("==== TEST 6: deleteCharAt(middle) =====");
42        hello.deleteCharAt(1);
43        System.out.println("Expected string = !lroW olleH");
44        System.out.println("Actual string   = " + hello.toString());
45        System.out.println("Expected size = 11");
46        System.out.println("Actual size    = " + hello.size());
47        System.out.println();
48
49        System.out.println("==== TEST 7: deleteCharAt(first) =====");
50        hello.deleteCharAt(0);
51        System.out.println("Expected string = lroW olleH");
52        System.out.println("Actual string   = " + hello.toString());
53        System.out.println();
54
55        System.out.println("==== TEST 8: deleteCharAt(last) =====");
56        hello.deleteCharAt(hello.size() - 1);
57        System.out.println("Expected string = lroW olle");
58        System.out.println("Actual string   = " + hello.toString());
```

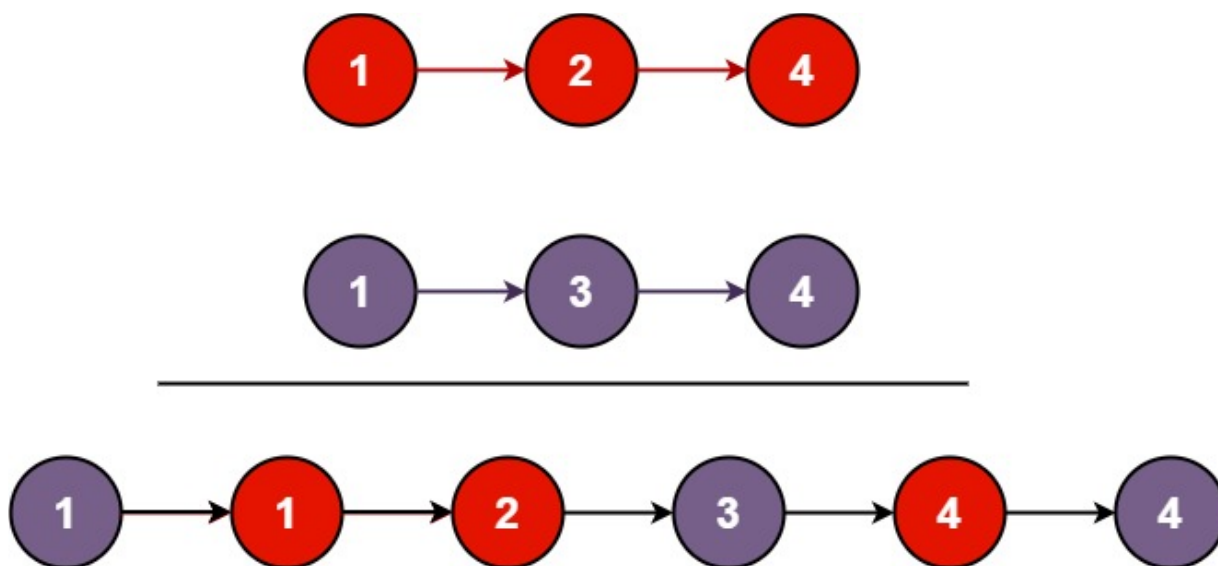
```
59     System.out.println();
60
61     System.out.println("==== TEST 9: reverse() again (odd size) =====");
62     hello.reverse();
63     System.out.println("Expected string = ello Worl");
64     System.out.println("Actual string  = " + hello.toString());
65     System.out.println();
66
67     System.out.println("==== ALL TESTS EXECUTED =====");
68 }
69 }
```

Task 2 (4 points)

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.



Question link on [LeetCode](#)

Example 1:

Input: list1 = [1,2,4] , list2 = [1,3,4]

Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [] , list2 = []

Output: []

Example 3:

Input: list1 = [], list2 = [0]

Output: [0]

Constraints:

- The number of nodes in both lists is in the range [0,50] .
- $-100 \leq \text{Node.val} \leq 100$
- Both list1 and list2 are sorted in non-decreasing order.

Use the following code to implement your solution:

```
1  /* make sure to add this package identifier before you
2  * submit your solution to GradeScope. Remove it while testing locally */
3  package com.gradescope.labTwo;
4  import com.gradescope.labTwo.tests.SinglyLinkedList;
5
6  public class TaskTwo {
7      public static SinglyLinkedList.Node<Integer> mergeTwoLists(
8          SinglyLinkedList.Node<Integer> l1,
9          SinglyLinkedList.Node<Integer> l2
10     ) {
11         // implement your solution here
12     }
13 }
```

You are going to solve this question on LeetCode so you get more used to it :)

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

💬 0



👁 118

**Read more****Data Structures and Algorithms Lab 1 - Arrays**

In this lab, we are going to dive into Arrays, one of the most basic yet fundamental data structures that we us...

Nov 15, 2025

Data Structures and Algorithms Lab 0 - Introduction

Welcome one and all to the Data Structures and Algorithms lab! This lab is one of the most important...

Nov 8, 2025

Read more from Ahmad Albarasy

Published on  HackMD