Linked Lists

Nishant - Notes

December 7, 2024

Contents

1 Introduction to Linked Lists

1.1 What is a Linked List?

A linked list is a linear data structure resembling a chain, where each node is connected to the next, and each node represents an individual element. Unlike arrays, the elements in a linked list are not stored in contiguous memory locations. This non-contiguous nature allows for efficient insertion and deletion of elements without the need to shift other elements, a limitation present in arrays.

1.2 Why Linked Lists over Arrays?

Linked lists offer several advantages over arrays:

- **Dynamic Size**: Unlike arrays, the size of a linked list can be increased or decreased at any location and at any point in time efficiently.
- **Efficient Insertions/Deletions**: Inserting or deleting elements does not require shifting elements, making these operations more efficient, especially for large datasets.
- **Memory Utilization**: Linked lists use memory more efficiently as they do not require a predefined size and can expand as needed.

1.3 Types of Linked Lists

- **Singly Linked Lists**: Each node points to the next node in the sequence.
- **Doubly Linked Lists**: Each node points to both the next and the previous nodes, allowing bidirectional traversal.
- **Circular Linked Lists**: The last node points back to the first node, forming a circle.

2 Implementing a Linked List in Java

2.1 Node Class Definition

The fundamental building block of a linked list is the `Node` class, which contains two primary components: the data and a pointer to the next node.

2.1.1 Java Code: Node Class

```java
class Node {
    public int data;
    public Node next;

    // Constructor to initialize a new node
    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

Listing 1: Java: Node Class Definition

2.2 Creating and Linking Nodes

To create a linked list, instantiate nodes and link them together by setting the `next` pointers.

2.2.1 Java Code: Creating and Linking Nodes

```java
public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<>();
        arr.add(2);
        arr.add(5);
        arr.add(8);
        arr.add(7);

        /*
         * Assigning values to
         * the nodes
         */
        Node y1 = new Node(arr.get(0), null);
        Node y2 = new Node(arr.get(1), null);
        Node y3 = new Node(arr.get(2), null);
        Node y4 = new Node(arr.get(3), null);

        /*
         * Linking of
         * Nodes
         */
        y1.next = y2;
        y2.next = y3;
        y3.next = y4;

        /*
         * Printing Nodes with their
         * values and data
         */
        System.out.println(y1.data + " " + y1.next);
        System.out.println(y2.data + " " + y2.next);
        System.out.println(y3.data + " " + y3.next);
        System.out.println(y4.data + " " + y4.next);
    }
}
```

Listing 2: Java: Creating and Linking Nodes

2.3 Converting an Array to a Linked List

Converting an array to a linked list involves creating nodes for each element and linking them sequentially.

2.3.1 Java Code: Array to Linked List

```java
import java.util.*;

class Node {
    int data;
    Node next;

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        next = null;
    }
}

public class LinkedList {
    // Function to convert an array to a linked list
    public static Node arrayToLinkedList(int[] arr) {
        int size = arr.length;
        if (size == 0) return null;

        // Create head of the linked list
        Node head = new Node(arr[0]);
        Node current = head;

        /* Iterate through the array
           and create linked list nodes */
        for (int i = 1; i < size; i++) {
            current.next = new Node(arr[i]);
            current = current.next;
        }

        return head;
    }

    // Function to print the linked list
    public static void printLinkedList(Node head) {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
```

```java
        int[] arr = {1, 2, 3, 4, 5};

        // Convert array to linked list
        Node head = arrayToLinkedList(arr);

        // Print the linked list
        printLinkedList(head);
    }
}
```

Listing 3: Java: Array to Linked List Conversion

2.4 Traversing a Linked List

Traversal involves visiting each node in the linked list sequentially from the head to the end.

2.4.1 Java Code: Traversing and Printing Linked List

```java
import java.util.*;

class Node {
    int data;
    Node next;

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        next = null;
    }
}

class Solution {
    // Function for Linked List Traversal
    public List<Integer> LLTraversal(ListNode head) {
        // Storing a copy of the linked list
        ListNode temp = head;
        // To store the values sequentially
        List<Integer> ans = new ArrayList<>();

        // Keep traversing until null is encountered
        while (temp != null) {
            // Storing the values
            ans.add(temp.val);
            // Storing the address of the next node
            temp = temp.next;
        }
        // Return answer
        return ans;
    }
}

```

```java
public class Main {
    public static void main(String[] args) {
        // Manual creation of nodes
        ListNode y1 = new ListNode(2);
        ListNode y2 = new ListNode(5);
        ListNode y3 = new ListNode(8);
        ListNode y4 = new ListNode(7);

        // Linking the nodes
        y1.next = y2;
        y2.next = y3;
        y3.next = y4;

        // Creating an instance of Solution class
        Solution solution = new Solution();

        // Calling LLTraversal method to get the values
        List<Integer> result = solution.LLTraversal(y1);

        // Printing the result
        System.out.println("Linked List Values:");
        for (int val : result) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```

Listing 4: Java: Linked List Traversal

2.5 Calculating Length of a Linked List

Determining the number of nodes in a linked list.

2.5.1 Java Code: Calculating Length

```java
import java.util.*;

class Node {
    int data;
    Node next;

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        next = null;
    }
}

public class LinkedList {
    // Function to convert an array to a linked list
```

```java
    public static Node arrayToLinkedList(int[] arr) {
        int size = arr.length;
        if (size == 0) return null;

        // Create head of the linked list
        Node head = new Node(arr[0]);
        Node current = head;

        /* Iterate through the array
           and create linked list nodes */
        for (int i = 1; i < size; i++) {
            current.next = new Node(arr[i]);
            current = current.next;
        }

        return head;
    }

    // Function to print the linked list
    public static void printLinkedList(Node head) {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    // Function to calculate the length of the linked list
    public static int lengthOfLinkedList(Node head) {
        int length = 0;
        Node current = head;

        // Count the nodes
        while (current != null) {
            length++;
            current = current.next;
        }

        return length;
    }

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

        // Convert array to linked list
        Node head = arrayToLinkedList(arr);

        // Print the linked list
        printLinkedList(head);

```

```
        // Calculate the length of the linked list
        int length = lengthOfLinkedList(head);
        System.out.println("Length of the linked list: " + length);
    }
}
```

Listing 5: Java: Calculating Length of Linked List

2.6 Searching for an Element in a Linked List

Determining whether a specific value exists within the linked list.

2.6.1 Java Code: Searching an Element

```
import java.util.*;

class Node {
    int data;
    Node next;

    // Constructor to initialize a new node
    Node(int val) {
        data = val;
        next = null;
    }
}

public class LinkedList {
    // Function to print the linked list
    public static void printLinkedList(Node head) {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    // Function to search for an element in the linked list
    public static boolean searchElement(Node head, int target) {
        Node current = head;

        // Traverse the linked list
        while (current != null) {
            if (current.data == target) {
                return true;
            }
            current = current.next;
        }

        return false;
```

```java
    }

    public static void main(String[] args) {
        // Create a linked list manually
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(4);
        head.next.next.next.next = new Node(5);

        // Print the linked list
        printLinkedList(head);

        // Search for an element in the linked list
        int target = 3;
        if (searchElement(head, target)) {
            System.out.println("Element " + target + " found in the linked list.");
        } else {
            System.out.println("Element " + target + " not found in the linked list.");
        }
    }
}
```

Listing 6: Java: Searching for an Element in Linked List

2.7 Deleting the k-th Node in a Linked List

Removing the node at the specified position.

2.7.1 Java Code: Deleting the k-th Node

```java
import java.io.*;

class ListNode {
    int val;
    ListNode next;
    ListNode(int data1) {
        val = data1;
        next = null;
    }

    ListNode(int data1, ListNode next1) {
        val = data1;
        next = next1;
    }
}

class Solution {
    // Function to delete the k-th node of a linked list
```

```
    public ListNode deleteKthNode(ListNode head, int k) {
        // If the list is empty, return null
        if (head == null)
            return null;

        // If k is 1, delete the head node
        if (k == 1) {
            ListNode temp = head;
            head = head.next;
            return head;
        }

        // Initialize a temporary pointer
        ListNode temp = head;

        // Traverse to the (k-1)th node
        for (int i = 0; temp != null && i < k - 2; i++) {
            temp = temp.next;
        }

        /* If k is greater than the number of nodes,
           return the unchanged list */
        if (temp == null || temp.next == null)
            return head;

        // Delete the k-th node
        ListNode next = temp.next.next;
        temp.next = next;

        // Return head
        return head;
    }
}

public class Main {
    // Function to print the linked list
    private static void printLL(ListNode head) {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + "␣");
            current = current.next;
        }
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
        // Initialize an array with values for the linked list
        int[] arr = {12, 5, 8, 7};

        // Create a linked list with the values from the array
```

```java
        ListNode head = new ListNode(arr[0]);
        head.next = new ListNode(arr[1]);
        head.next.next = new ListNode(arr[2]);
        head.next.next.next = new ListNode(arr[3]);

        // Print the original linked list
        System.out.print("Original list: ");
        printLL(head);

        // Creating an instance of Solution class
        Solution sol = new Solution();

        // Call the deleteKthNode function to delete the k-th node
        int k = 2;
        head = sol.deleteKthNode(head, k);

        // Print the linked list after deletion
        System.out.print("List after deleting the kth node: ");
        printLL(head);
    }
}
```

Listing 7: Java: Deleting the k-th Node in Linked List

2.8 Inserting at the k-th Position in a Linked List

Adding a new node at a specified position.

2.8.1 Java Code: Inserting at k-th Position

```java
import java.util.*;

// Definition of singly linked list
class ListNode {
    int val;
    ListNode next;

    ListNode(int data1) {
        val = data1;
        next = null;
    }

    ListNode(int data1, ListNode next1) {
        val = data1;
        next = next1;
    }
}

// Solution class
class Solution {
    // Function to insert a new node at the k-th position
```

```java
    public ListNode insertAtKthPosition(ListNode head, int X, int K) {
        /* If the linked list is empty
           and k is 1, insert the
           new node as the head */
        if (head == null) {
            if (K == 1)
                return new ListNode(X);
            else
                return head;
        }

        /* If K is 1, insert the new
           node at the beginning
           of the linked list */
        if (K == 1)
            return new ListNode(X, head);

        int cnt = 0;
        ListNode temp = head;

        /* Traverse the linked list
           to find the node at position k-1 */
        while (temp != null) {
            cnt++;
            if (cnt == K - 1) {
                /* Insert the new node after the node
                   at position k-1 */
                ListNode newNode = new ListNode(X, temp.next);
                temp.next = newNode;
                break;
            }
            temp = temp.next;
        }

        return head;
    }
}

// Main class
public class Main {
    // Helper Method to print the linked list
    private static void printLL(ListNode head) {
        while (head != null) {
            System.out.print(head.val + " ");
            head = head.next;
        }
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
```

```java
        // Create a linked list from an array
        int[] arr = {10, 30, 40};
        int X = 20, K = 2;
        ListNode head = new ListNode(arr[0]);
        head.next = new ListNode(arr[1]);
        head.next.next = new ListNode(arr[2]);

        // Print the original list
        System.out.print("Original List: ");
        printLL(head);

        // Create a Solution object
        Solution sol = new Solution();
        head = sol.insertAtKthPosition(head, X, K);

        // Print the modified linked list
        System.out.print("List after inserting the given value at the Kth 
            position: ");
        printLL(head);
    }
}
```

Listing 8: Java: Inserting at the k-th Position in Linked List

2.9 Reversing a Linked List

Reversing a linked list can be done either iteratively or recursively. Below are both approaches with detailed explanations and Java implementations.

2.9.1 Iterative Approach

Intuition To reverse a linked list without using extra space, we change the direction of the links between the nodes. Think of it like flipping the arrows between the nodes. This means each node will point to the one before it instead of the one after it. By doing this, the last node in the original list becomes the first node in the reversed list. This way, we efficiently reverse the list without needing any extra memory.

Approach

1. **Initialize Pointers**: Start by setting two pointers, `temp` and `prev`, at the head of the linked list and `NULL` respectively. The `temp` pointer will be used to traverse the list, while the `prev` pointer will help reverse the direction of the links.

2. **Traverse and Reverse**: Move through the linked list with the `temp` pointer. For each node:

   - Save the next node in a variable called `front`. This ensures you don't lose track of the remaining list.
   - Change the next pointer of the current node (`temp`) to point to the previous node (`prev`). This action reverses the link.

- Move the `prev` pointer to the current node (`temp`). This prepares `prev` for the next iteration.
- Move the `temp` pointer to the next node (`front`). This continues the traversal.

3. **Complete the Reversal**: Continue the process until the `temp` pointer reaches the end of the list (`NULL`). At this point, the `prev` pointer will be at the new head of the reversed list.
4. **Return the New Head**: Finally, return the `prev` pointer as it now points to the head of the reversed linked list.

2.9.2 Java Code: Iterative Reversal

```java
import java.util.*;

// Definition of singly linked list
class ListNode {
    int val;
    ListNode next;
    ListNode() {
        val = 0;
        next = null;
    }
    ListNode(int data1) {
        val = data1;
        next = null;
    }
    ListNode(int data1, ListNode next1) {
        val = data1;
        next = next1;
    }
}

class Solution {
    /* Function to reverse a linked list using iteration */
    public ListNode reverseList(ListNode head) {
        /* Initialize 'temp' at
           head of linked list */
        ListNode temp = head;

        /* Initialize pointer 'prev' to NULL,
           representing the previous node */
        ListNode prev = null;

        /* Traverse the list, continue till
           'temp' reaches the end (NULL) */
        while (temp != null) {
            /* Store the next node in
               'front' to preserve the reference */
            ListNode front = temp.next;

            /* Reverse the direction of the
```

```java
               current node's 'next' pointer
               to point to 'prev' */
            temp.next = prev;

            /* Move 'prev' to the current
               node for the next iteration */
            prev = temp;

            /* Move 'temp' to the 'front' node
               advancing the traversal */
            temp = front;
        }

        /* Return the new head of
           the reversed linked list */
        return prev;
    }
}

public class Main {
    // Function to print the linked list
    public static void printLinkedList(ListNode head) {
        ListNode temp = head;
        while (temp != null) {
            System.out.print(temp.val + "␣");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Create a linked list with values 1, 3, 2, and 4
        ListNode head = new ListNode(1);
        head.next = new ListNode(3);
        head.next.next = new ListNode(2);
        head.next.next.next = new ListNode(4);

        // Print the original linked list
        System.out.print("Original␣Linked␣List:␣");
        printLinkedList(head);

        // Solution instance
        Solution sol = new Solution();
        // Reverse the linked list
        head = sol.reverseList(head);

        // Print the reversed linked list
        System.out.print("Reversed␣Linked␣List:␣");
        printLinkedList(head);
    }
}
```

Listing 9: Java: Iterative Reversal of Linked List

2.9.3 Recursive Approach

Intuition Recursion enables us to decompose a problem into more manageable, smaller subproblems, which we can then solve one at a time until we get to the base case, or most straightforward answer. After that, we solve the initial problem by combining the outcomes of these smaller solutions.

When recursively reversing a linked list, we start by considering the complete list with N nodes. We can break this down recursively by starting with $N-1$ nodes, moving on to $N-2$ nodes, and so on, until we reach a single node.

In the base case, reversing a list with one node is straightforward because the list is already in reverse. We simply return this node. When we return from each recursive call, we flip the pointers to reverse the linkages between nodes, thereby reversing the entire list.

This method effectively manages the reversal process by using the power of recursion to break down the task into smaller, more manageable parts.

Approach

1. **Base Case**: Check if the linked list is empty or has only one node. In these cases, the list is already reversed, so simply return the head.

2. **Recursive Function**: The main part of the algorithm is a recursive function that handles the reversal of the linked list.

   - If the base case is not met, the function calls itself recursively. This process continues until the base case is reached, effectively reversing the list starting from the second node onwards.

3. **Returning the New Head**: After the recursion completes, the function returns the new head of the reversed linked list. This new head was the last node of the original list before the reversal.

2.9.4 Java Code: Recursive Reversal

```
import java.util.*;

// Definition of singly linked list
class ListNode {
    int val;
    ListNode next;
    ListNode() {
        val = 0;
        next = null;
    }
    ListNode(int data1) {
        val = data1;
```

```java
        next = null;
    }
    ListNode(int data1, ListNode next1) {
        val = data1;
        next = next1;
    }
}

class Solution {
    /* Function to reverse a singly linked list using recursion */
    public ListNode reverseList(ListNode head) {
        /* Base case:
           If the linked list is empty or has only one node,
           return the head as it is already reversed. */
        if (head == null || head.next == null) {
             return head;
        }

        /* Recursive step:
           Reverse the linked list starting
           from the second node (head.next). */
        ListNode newHead = reverseList(head.next);

        /* Save a reference to the node following
           the current 'head' node. */
        ListNode front = head.next;

        /* Make the 'front' node point
           to the current
           'head' node in the
           reversed order. */
        front.next = head;

        /* Break the link from
           the current 'head' node
           to the 'front' node
           to avoid cycles. */
        head.next = null;

        /* Return the 'newHead,'
           which is the new
           head of the reversed
           linked list. */
        return newHead;
    }
}

public class Main {
    // Function to print the linked list
    public static void printLinkedList(ListNode head) {
        ListNode temp = head;
```

```java
        while (temp != null) {
            System.out.print(temp.val + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Create a linked list with values 1, 3, 2, and 4
        ListNode head = new ListNode(1);
        head.next = new ListNode(3);
        head.next.next = new ListNode(2);
        head.next.next.next = new ListNode(4);

        // Print the original linked list
        System.out.print("Original Linked List: ");
        printLinkedList(head);

        // Solution instance
        Solution sol = new Solution();
        // Reverse the linked list recursively
        head = sol.reverseList(head);

        // Print the reversed linked list
        System.out.print("Reversed Linked List: ");
        printLinkedList(head);
    }
}
```

Listing 10: Java: Recursive Reversal of Linked List

2.10 Complexity Analysis

- **Time Complexity**: $O(n)$, where n is the number of nodes in the linked list.
- **Space Complexity**: $O(n)$ due to the recursive call stack.

2.11 Advantages

- **Iterative Approach**: Efficient with $O(1)$ space complexity.
- **Recursive Approach**: Elegant and easier to understand; aligns well with the divide-and-conquer paradigm.

2.12 Limitations

- **Iterative Approach**: Requires careful handling of pointers to avoid errors.
- **Recursive Approach**: Uses additional space for the call stack, which can lead to stack overflow for very large lists.

3 Complexity Analysis

3.1 Time Complexity

Linked list operations have varying time complexities based on the specific operation:

- **Traversal**: $O(n)$
- **Insertion at Head**: $O(1)$
- **Insertion at Tail**: $O(n)$
- **Deletion at Head**: $O(1)$
- **Deletion at Tail**: $O(n)$
- **Searching**: $O(n)$
- **Reversal**: $O(n)$

3.2 Space Complexity

Linked lists generally require $O(n)$ space to store n elements, as each node holds additional pointers besides the data.

4 Advantages and Limitations

4.1 Advantages

- **Dynamic Size**: Can easily grow or shrink as needed.
- **Efficient Insertions/Deletions**: Adding or removing elements does not require shifting elements.
- **Memory Utilization**: Does not require contiguous memory allocation.

4.2 Limitations

- **Random Access**: Does not allow direct access to elements; requires traversal from the head.
- **Memory Overhead**: Each node requires extra memory for pointers.
- **Cache Performance**: Poorer cache performance compared to arrays due to non-contiguous memory allocation.

5 Conclusion/Summary

In this section, we delved into the concept of linked lists, exploring their structure, advantages over arrays, and various operations such as creation, traversal, insertion, deletion, and reversal. Linked lists offer a dynamic and flexible way to store data, allowing efficient insertions and deletions without the constraints of fixed-size arrays. However, they come with trade-offs, including lack of random access and additional memory overhead for pointers.

Understanding linked lists is fundamental for mastering more complex data structures and algorithms. They serve as the backbone for various applications, including implementing stacks, queues, and other abstract data types. Mastery of linked lists also enhances problem-solving skills, enabling the development of efficient and optimized code.

6 References

- GeeksforGeeks: Linked List
- GeeksforGeeks: Introduction to Linked Lists
- Wikipedia: Linked List
- YouTube: Linked List Tutorial
- Programiz: Linked List in Java