Linked List Data Structure

A **linked list** is a fundamental data structure in computer science. It mainly allows efficient **insertion** and **deletion** operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque. Here's the comparison of Linked List vs Arrays

Linked List:

- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

- **Data Structure:** Contiguous
- **Memory Allocation:** Typically allocated to the whole array
- **Insertion/Deletion:** Inefficient
- **Access:** Random

Singly Linked List

A **singly linked list** is a fundamental data structure, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the linked list. The next of the last node is **null**, indicating the end of the list. Linked Lists support efficient insertion and deletion operations.

Understanding Node Structure

In a singly linked list, each node consists of two parts: data and a pointer to the next node. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.

// Definition of a Node in a singly linked list

public class Node {

// Data part of the node

int data;

// Pointer to the next node in the list

```
    Node next;

    // Constructor to initialize the node with data
    public Node(int data){
        this.data = data;
        this.next = null;
    }
}
```

In this example, the Node class contains an integer data field (**data**) to store the information and a pointer to another Node (**next**) to establish the link to the next node in the list.

Creating an Example Linked List of Size 3 to Understand Working

Create the first node

- Allocate memory for the first node and Store data in it.
- Mark this node as head.

Create the second node

- Allocate memory for the second node and Store data in it.
- Link the first node's next to this new node.

Create the third node

- Allocate memory for the third node and Store data in it.
- Link the second node's next to this node.
- Set its next to NULL to ensure that the next of the last is NULL.

```
public static void main(String[] args) {
        // Create the first node (head of the list)
        Node head = new Node(10);

        // Link the second node
        head.next = new Node(20);

        // Link the third node
```

```
        head.next.next = new Node(30);


        // Link the fourth node
        head.next.next.next = new Node(40);
    }
}
```

Applications of Linked List

Advantage

- Dynamic size (no fixed limit like arrays)
- Efficient insertion and deletion (especially in the middle)
- Can implement complex data structures like stack, queue, graph

Disadvantage

- Extra memory required for storing pointers
- No direct/random access (need traversal)
- Cache unfriendly (not stored in contiguous memory)

Doubly Linked List

A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)**Node Definition**

Here is how a node in a Doubly Linked List is typically represented:

```
class Node {

  // To store the Value or data.
  int data;

  // Reference to the Previous Node
  Node prev;

  // Reference to the next Node
  Node next;

  // Constructor
  Node(int d) {
    data = d;
    prev = next = null;
  }
};
```

*Each node in a **Doubly Linked List** contains the **data** it holds, a pointer to the **next** node in the list, and a pointer to the **previous** node in the list. By linking these nodes together through the **next** and **prev** pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List.*

Creating a Doubly Linked List with 4 Nodes

Create the head node.

- Allocate a node and set head to it. Its prev and next should be null/None.

Create the next node and link it to head.

- head.next = new Node(value2)
- head.next.prev = head

Create further nodes the same way.

- For the third node:
  => head.next.next = new Node(value3)
  => head.next.next.prev = head.next
- Repeat until you have the required nodes.

Ensure the tail's next is null.
The last node you created must have next == null

Set / keep track of head (and optionally tail).
Use head to access the list from the front. Keeping a tail pointer simplifies appends.

```
class Node {
    int data;
    Node prev;
    Node next;

    Node(int value) {
        data = value;
        prev = null;
        next = null;
    }
}

class GfG {
    public static void main(String[] args) {
        // Create the first node (head of the list)
        Node head = new Node(10);

        // Create and link the second node
        head.next = new Node(20);
        head.next.prev = head;
```

```
        // Create and link the third node
        head.next.next = new Node(30);
        head.next.next.prev = head.next;

        // Create and link the fourth node
        head.next.next.next = new Node(40);
        head.next.next.next.prev = head.next.next;

        // Traverse the list forward and print elements
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data);
            if (temp.next != null) {
                System.out.print(" <-> ");
            }
            temp = temp.next;
        }
    }
}
```

Output

10 <-> 20 <-> 30 <-> 40

Introduction to Circular Linked List

A circular linked list is a data structure where the last node points back to the first node, forming a closed loop.

- **Structure:** All nodes are connected in a circle, enabling continuous traversal without encountering NULL.
- Difference from Regular Linked List: In a regular linked list, the last node points to NULL, whereas in a circular linked list, it points to the first node.

- **Uses:** Ideal for tasks like scheduling and managing playlists, where smooth and repeated.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked list and doubly linked list . So, circular linked lists are basically of two types:

1. Circular Singly Linked List

In Circular Singly Linked List, each node has just one pointer called the "next" pointer. The next pointer of the last node points back to the first node and this results in forming a circle. In this type of Linked list, we can only move through the list in one direction.

2. Circular Doubly Linked List:

In circular doubly linked list, each node has two pointers prev and next, similar to doubly linked list. The prev pointer points to the previous node and the next points to the next node. Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.

Note: Here, we will use the singly linked list to explain the working of circular linked lists.

Representation of a Circular Singly Linked List

Let's take a look on the structure of a circular linked list.

Create/Declare a Node of Circular Linked List

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

   Node(int data){
      this.data = data;
      this.next = null;
   }
}
```

In the code above, each node has **data** and a **pointer** to the next node. When we create multiple nodes for a circular linked list, we only need to connect the last node back to the first one.

Example of Creating a Circular Linked List

Here's an example of creating a circular linked list with three nodes (10, 20, 30, 40, 50):

Why have we taken a pointer that points to the last node instead of the first node?

For the insertion of a node at the beginning, we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of the start pointer, we take a pointer to the last node, then in both cases there won't be any need to traverse the whole list. So insertion at the beginning or at the end takes constant time, irrespective of the length of the list.

Operations

Length of a Linked List

Given a Singly Linked List, the task is to find the Length of the Linked List.

Examples:

Input: *LinkedList = 1->3->1->2->1*
Output: *5*
Explanation: *The linked list has 5 nodes.*

Input: *LinkedList = 2->4->1->9->5->3->6*
Output: *7*
Explanation: *The linked list has 7 nodes.*

Input: *LinkedList = 10->20->30->40->50->60*
Output: *6*
Explanation: *The linked list has 6 nodes.*

Iterative Approach to Find the Length of a Linked List

The idea is similar to traversal of linked list with an additional variable to count the number of nodes in the Linked List.

Steps to find the length of the Linked List:

- Initialize count as 0.
- Initialize a node pointer, curr = head.
- Do following while curr is not NULL
    - curr = curr -> next
    - Increment count by 1.

- Return count.

```
// Iterative Java program to count the number of
// nodes in a linked list

// Node class to define a linked list node
class Node {
    int data;
    Node next;

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

// Class to define methods related to the linked list
public class GFG {

    // Counts number of nodes in linked list
    public static int countNodes(Node head) {

        // Initialize count with 0
        int count = 0;

        // Initialize curr with head of Linked List
        Node curr = head;
```

```
    // Traverse till we reach null
    while (curr != null) {

        // Increment count by 1
        count++;

        // Move pointer to next node
        curr = curr.next;
    }
    // Return the count of nodes
    return count;
}

// Driver code
public static void main(String[] args) {

    // Create a hard-coded linked list:
    // 1 -> 3 -> 1 -> 2 -> 1
    Node head = new Node(1);
    head.next = new Node(3);
    head.next.next = new Node(1);
    head.next.next.next = new Node(2);
    head.next.next.next.next = new Node(1);

    // Function call to count the number of nodes
    System.out.println("Count of nodes is "
                       + countNodes(head));
}
```

```
}
```

Output

Count of nodes is 5

Time complexity: O(n), Where n is the size of the linked list
Auxiliary Space: O(1), As constant extra space is used.

Recursive Approach to Find the Length of a Linked List:

*The idea is to use recursion by maintaining a function, say **countNodes(node)** which takes a node as an argument and calls itself with the next node until we reach the end of the Linked List. Each of the recursive call returns **1 + count of remaining nodes**.*

```
// Recursive Java program to find length
// or count of nodes in a linked list

// Link list node
class Node {
    int data;
    Node next;

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

// Recursively count number of nodes in linked list
public class GFG {
    public static int countNodes(Node head) {
```

```
        // Base Case
        if (head == null) {
            return 0;
        }

        // Count this node plus the rest of the list
        return 1 + countNodes(head.next);
    }

    public static void main(String[] args) {

        // Create a hard-coded linked list:
        // 1 -> 3 -> 1 -> 2 -> 1
        Node head = new Node(1);
        head.next = new Node(3);
        head.next.next = new Node(1);
        head.next.next.next = new Node(2);
        head.next.next.next.next = new Node(1);

        // Function call to count the number of nodes
        System.out.println("Count of nodes is "
                           + countNodes(head));
    }
}
```

Output

Count of nodes is 5

Time Complexity: O(n), where **n** is the length of Linked List.
Auxiliary Space: O(n), Extra space is used in the recursion call stack.