Session – 9

Linked List

CONTENTS

01

Introduction to Linked List

Introduction to Linked List

What is a Linked List?

- A linear data structure
- Consists of nodes where each node contains:
  - Data
  - Reference (or pointer) to the next node
- Unlike arrays, memory is not continuous

Why Use Linked Lists?

- Dynamic memory allocation
- Easy insertion/deletion
- No need to define size in advance
- Efficient memory usage

Introduction to LinkedList

What is a Linked List?

- A linear data structure
- Unlike arrays, memory is not continuous
- Consists of nodes where each node contains:
  - Data
  - Reference (or pointer) to the next node
- What is Node in a Linked List?
  - Building Block
  - Node itself is not a linked list

02

Types of Linked List

Types of Linked List

- **Singly Linked List**: One pointer (next)
- **Doubly Linked List**: Two pointers (next, prev)
- **Circular Linked List**: Last node points to the head

Singly Linked List

- **Structure**: Each node has **two parts**: `data` and `next` (pointer to the next node).
- **Navigation**: Only **forward** from head to end.
- **Last node** points to `NULL`.

Advantages:

- Simple to implement
- Uses less memory than doubly or circular lists

Limitations:

- Cannot traverse backwards

Doubly Linked List

- Each node has **three parts**: `prev`, `data`, and `next`.
- Can move **both forward and backward**.
- **First node's prev** is `NULL` and **last node's next** is `NULL`.

Advantages:

- Easy to reverse
- Bi-directional traversal

Limitations:

- Uses more memory (extra pointer)

Circular Linked List

- In this list, the **last node points to the first node**, forming a **loop**.
- Can be **singly or doubly circular**.

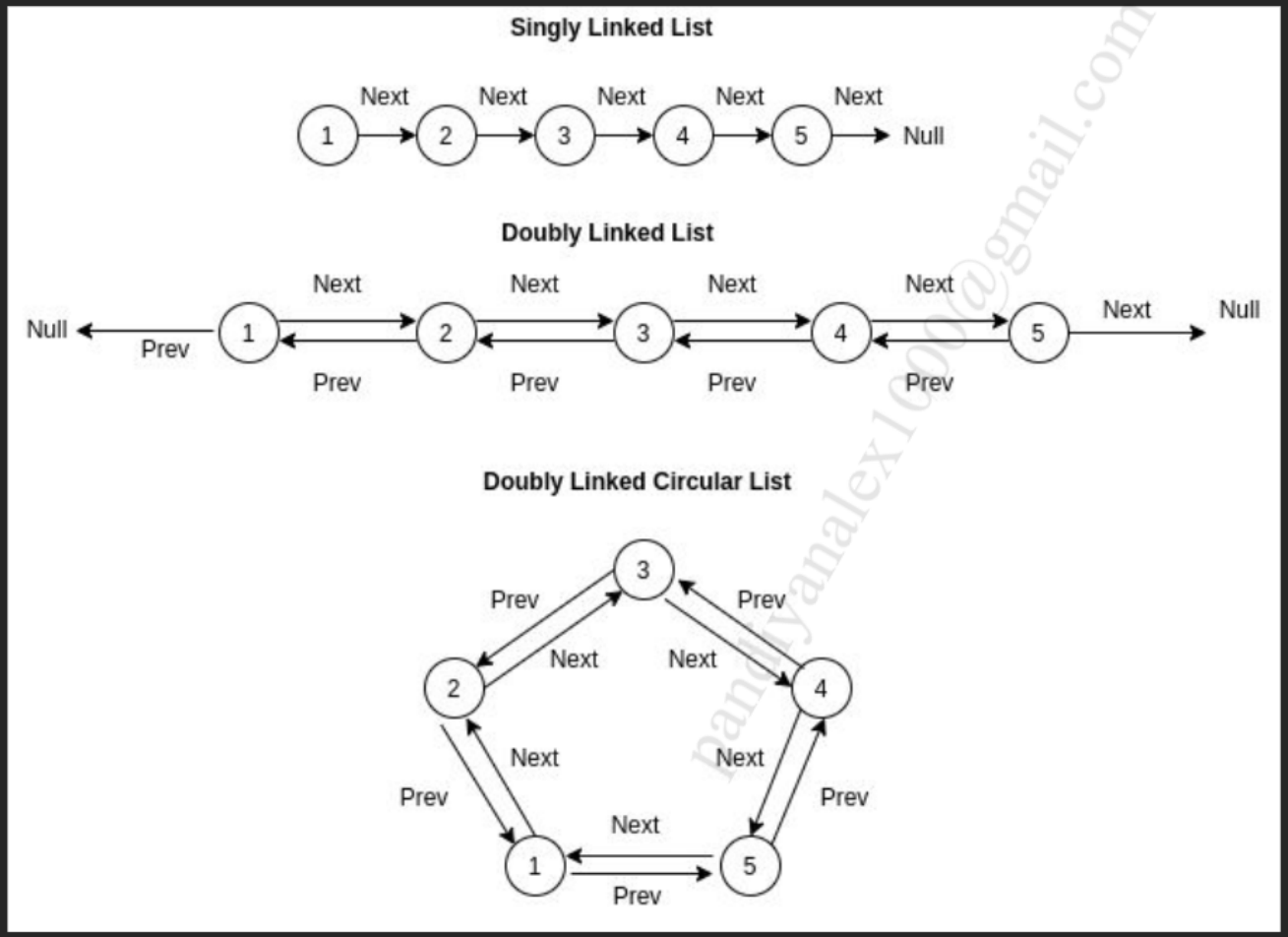
Advantages:

- Efficient for cyclic traversal (like in a game, round-robin scheduling.
- No NULL checks needed.

Limitations:

- Infinite loops possible if not handled properly

Types of Linked List



03

Operations on Linked List

Operations on Linked List

1. **Insertion** (Beginning, Middle, End)
2. **Deletion** (Beginning, Middle, End)
3. **Traversal**
4. **Searching**

04

Linked List - Important Patterns

Linked List - Important Patterns

- Implementation pattern
- Slow and fast pointer
- Reversing a Linked List
- Partition a LL
- Two LL patterns
- Mark and verify / Modify Node
- Loop pattern
- Sorting pattern

Implementation Patterns

Concept

Simply use the basic loop (next pointer) modification to achieve the expected result. Do the expected stuff inside of the while/for loop when iterating over a linked list.

Example Problems

- Add 1 to a number represented by a linked list
- Insert an element into a sorted Circular List
- Delete Kth to last element in a Linked List
- Rotate a Linked List

Fast and Slow Pointer Pattern

Concept

- We use two pointers simultaneously to iterate over a linked list
- One pointer (slow) takes 1 step at a time
- Second pointer (fast) takes 2 steps at time
- By the time the fast pointer reaches the end of the of the node, slow pointer is expected to be in the middle
- This can also be used as an intermediate step in many linked list problems

Example Problems

- Delete the middle element of a linked list
- Detect cycle in a given linked list

Reversing a Linked List

Concept

- Reversing a linked list with the prev and the next pointer
- This can also be done recursively
- Reversing a linked list can be used as an auxiliary step in many problems

Example Problems

- Check if linked list is a palindrome or not
- Reverse the second half of a given linked list

Partition a Linked List

Concept

- Create new empty linked lists (two or more if required, one serving each partition)
- Maintain the heads and/or tails of these new lists
- While iterating over the original linked list, check which new linked list the current node should be added to, simply append that node to the end of that linked list and keep iterating.
- Merge the separate linked lists into one, if required.

Example Problems

- Segregate odd and even nodes in a linked list
- Segregate odd and even values in a linked list
- Clone a linked list with random pointer
- Sort a linked list of 0s 1s and 2s

Two Linked List Patterns

Concept

- Two linked lists will be given as an input.
- Generally we use two pointers, one at the start of the first linked list and other at the start of the second linked list.
- Move the pointers together OR One at a time depending on the question and do the needful (modify value, remove, add to a new list, etc.)

Example Problems

- Merge two sorted linked lists
- Add two numbers represented by linked lists
- Multiply two numbers represented by linked lists

Mark and Verify / Modify node pattern

Concept

- There will be cases where we have to mark and node and visit it later, to check whether we have visited it or not
- Either we modify a structure to maintain a boolean variable namely isVisited
- Or if the given values of a nodes are positive integers then we mark it negative while iterating.
- When we revisit the nodes, either by checking the boolean or by looking at the value of the integer, we can tell whether its been visited or not.

Example Problems

- Intersection of two Linked lists.

• Loop pattern

Concept

- ➢ We detect a loop with the help of slow and fast pointer, if we move them simultaneously and there is a loop present, they are bound to meet at some node.
- ➢ Once the loop is detected, we then try to remove the loop. To remove the loop we need to find the node pointing to the head of the loop.
- ➢ To find the head of the loop, we point any 1 pointer to the head of the linked list and other pointer at the meeting point.
- ➢ Move both the pointers at the same pace now, until they both point to a same node. That same node is the starting point of loop.

Example Problems

- ➢ Detect if loop/cycle exists in a given linked list
- ➢ Find length of a linked list cycle
- ➢ Detect the head of the cycle in a given linked list
- ➢ Remove the loop in a given linked list

Sorting pattern

Concept

- We are asked to sort a given linked list .
- The idea is to use the concept same as merge sort.
- First find the middle of the LL (using slow and fast pointer)
- Then partition them and recursively sort both the linked lists.
- Now simply merge the two sorted linked lists (using the two linked list pattern)

Example Problems

- Sort a given linked list

05

Applications of Linked List

Application of Linked List

Real-World Examples

- Music playlists
- Browser history (back/forward)
- Image viewer (next/prev)

Implementation of:

- Stacks and Queues
- Graphs (Adjacency List)
- Hash tables (with chaining)
- Undo features (text editors)

Other Information

- The nodes are **not contiguously stored in memory**; unlike an array
- Linked lists are suitable for Stacks and Queues since they need addition and deletion at the ends
- Data structures like Binary Tree are related to linked lists; they have a Left and Right pointer instead of Next

Linked Lists are used to **implement**

- DS like **Stacks, Queues and Trees**
- Implement **Graphs** - Adjacency list representation of Graph
- **Hash Tables** - Each Bucket of the hash table can be a linked list (Open chain hashing)

06

Advantages & Disadvantages

Advantages & Disadvantages

Advantages	Disadvantages
Dynamic Size	More memory per node
Easy Insertion/Deletion	No random Access
Efficient use of memory	Extra Pointer overhead

07

Activity

Activity

- **Merge Two Sorted Lists** (2 linked list pattern)
- **Reverse Linked List** (Reverse pattern)
- **Middle of Linked List** (Fast & slow pointer)
- **Remove nth Node from the end of List**
- **Linked List Cycle**
- **Sort List** (sorting pattern)
- **Partition List** (Partition pattern)
- **Palindrome Linked list**

Activity

- Swap nodes in Pairs
- Odd Even Linked List
- Rotate List
- Add Two Numbers
- Reorder List
- Reverse nodes in K Group
- Find the Duplicate Number

https://leetcode.com/problems/lru-cache/description/ ☑️ - Check MICS