LINKED LISTS

Prerequisites: Arrays, pointers (conceptual understanding)

Table of Contents

1. Basics of Linked Lists
2. Singly Linked List
3. Doubly Linked List
4. Circular Linked List
5. Floyd's Cycle Detection Algorithm
6. Merge Sort on Linked Lists
7. LRU Cache Implementation
8. Skip Lists (Advanced)
9. Practice Problems
10. Interview Questions

BASICS OF LINKED LISTS

What is a Linked List?

A **linked list** is a linear data structure where each element (node) points to the next one. Unlike arrays, elements in a linked list are not stored in contiguous memory locations. Each node consists of:

- Data
- Pointer (reference) to the next node

Advantages:

- Dynamic size (no need to predefine size like arrays)
- Easy insertions/deletions (no shifting of elements needed)

Disadvantages:

- Sequential access only (no direct indexing)
- Extra memory for pointers

Real-life Analogy:
A chain of paperclips, each clip (node) points to the next.

Node Structure in Java:

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

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

2. SINGLY LINKED LIST

A **Singly Linked List** is the most basic form of a linked list, where each node contains data and a pointer to the next node.

Structure:
[Data | Next] → [Data | Next] → [Data | Next] → null

Operations:

1. **Insertion at Beginning**
2. **Insertion at End**
3. **Insertion at Position**
4. **Deletion (from Beginning, End, Position)**
5. **Traversal**
6. **Search**

Java Implementation:

```
class SinglyLinkedList {
    Node head;

    // Insert at beginning
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    // Insert at end
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
```

```
            return;
        }
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }

    // Delete from beginning
    public void deleteFromBeginning() {
        if (head != null) {
            head = head.next;
        }
    }

    // Delete from end
    public void deleteFromEnd() {
        if (head == null || head.next == null) {
            head = null;
            return;
        }
        Node temp = head;
        while (temp.next.next != null) {
            temp = temp.next;
        }
        temp.next = null;
    }

    // Traverse and print list
    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " → ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

Example:

```
public class Test {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insertAtBeginning(3);
```

```
        list.insertAtBeginning(2);
        list.insertAtBeginning(1);
        list.insertAtEnd(4);
        list.insertAtEnd(5);

        list.printList(); // 1 → 2 → 3 → 4 → 5 → null

        list.deleteFromBeginning();
        list.printList(); // 2 → 3 → 4 → 5 → null

        list.deleteFromEnd();
        list.printList(); // 2 → 3 → 4 → null
    }
}
```

Time Complexity:

- Insertion (beginning): O(1)
- Insertion (end): O(n)
- Deletion (beginning): O(1)
- Deletion (end): O(n)
- Traversal: O(n)

3. DOUBLY LINKED LIST

A **Doubly Linked List (DLL)** is a type of linked list in which each node contains three parts:

- Data
- Pointer to the **next** node
- Pointer to the **previous** node

Structure:

null ← [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] → null

null ← [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] → null

Advantages:

- Allows bidirectional traversal
- Easier deletion of a given node

Disadvantages:

- Extra memory for one additional pointer

Node Structure in Java:

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

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

Java Implementation:

```
class DoublyLinkedList {
    DNode head;

    // Insert at beginning
    public void insertAtBeginning(int data) {
```

```java
    DNode newNode = new DNode(data);
    newNode.next = head;
    if (head != null) head.prev = newNode;
    head = newNode;
}

// Insert at end
public void insertAtEnd(int data) {
    DNode newNode = new DNode(data);
    if (head == null) {
        head = newNode;
        return;
    }
    DNode temp = head;
    while (temp.next != null) temp = temp.next;
    temp.next = newNode;
    newNode.prev = temp;
}

// Delete from beginning
public void deleteFromBeginning() {
    if (head == null) return;
    head = head.next;
    if (head != null) head.prev = null;
}

// Delete from end
public void deleteFromEnd() {
    if (head == null || head.next == null) {
        head = null;
```

```
            return;
        }
        DNode temp = head;
        while (temp.next != null) temp = temp.next;
        temp.prev.next = null;
    }

    // Traverse list forward
    public void printForward() {
        DNode temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ⇄ ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

Example:

```
public class DLLTest {
    public static void main(String[] args) {
        DoublyLinkedList dll = new DoublyLinkedList();
        dll.insertAtBeginning(2);
        dll.insertAtBeginning(1);
        dll.insertAtEnd(3);
        dll.insertAtEnd(4);

        dll.printForward(); // 1 ⇄ 2 ⇄ 3 ⇄ 4 ⇄ null

        dll.deleteFromBeginning();
```

```
        dll.printForward(); // 2 ⇄ 3 ⇄ 4 ⇄ null

        dll.deleteFromEnd();
        dll.printForward(); // 2 ⇄ 3 ⇄ null
    }
}
```

4. CIRCULAR LINKED LIST

A **Circular Linked List (CLL)** is a linked list in which:

- The last node's next pointer points to the head (in singly CLL)
- Both next and prev pointers are circular (in doubly CLL)

Structure (Singly CLL):

```
[Data|Next] → [Data | Next] → [Data | Next] ┐
     ↑                  ↓
     └──────────────── HEAD ◄──────────┘
```

Java Implementation (Singly CLL):

```
class CircularLinkedList {
    Node head = null;
    Node tail = null;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
            newNode.next = head;
        } else {
            tail.next = newNode;
```

```
            tail = newNode;
            tail.next = head;
        }
    }

    public void printList() {
        if (head == null) return;
        Node temp = head;
        do {
            System.out.print(temp.data + " → ");
            temp = temp.next;
        } while (temp != head);
        System.out.println("HEAD");
    }
}
```

Example:

```
public class CircularTest {
    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();
        cll.insert(1);
        cll.insert(2);
        cll.insert(3);
        cll.insert(4);
        cll.printList(); // 1 → 2 → 3 → 4 → HEAD
    }
}
```

Use Cases:

- Round-robin scheduling

- Repeated buffering (music, streaming)

Floyd's Cycle Detection (Tortoise and Hare Algorithm)

Explanation:

Detecting a cycle in a linked list is a classic problem. A cycle exists if some node in the linked list can be reached again by continuously following the next pointers.

Floyd's Cycle Detection algorithm uses two pointers:

- **Slow pointer (tortoise):** Moves one step at a time.
- **Fast pointer (hare):** Moves two steps at a time.

If the linked list has a cycle, the fast pointer will eventually meet the slow pointer inside the cycle. If there is no cycle, the fast pointer will reach the end (null).

Steps:

1. Initialize slow and fast pointers to the head of the list.
2. Move slow by one node and fast by two nodes in each iteration.
3. If slow == fast at any point, a cycle exists.
4. If fast reaches null (end), no cycle.

Code (Java):

```
class Node {
  int data;
  Node next;
  Node(int d) { data = d; next = null; }
}

public class LinkedListCycle {

  public static boolean hasCycle(Node head) {
    if (head == null) return false;
```

```
        Node slow = head;
        Node fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;          // move slow by 1
            fast = fast.next.next;     // move fast by 2

            if (slow == fast) {        // cycle detected
                return true;
            }
        }
        return false; // no cycle
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);
        head.next.next.next = new Node(4);
        // Create cycle for testing
        head.next.next.next.next = head.next;

        System.out.println("Has cycle: " + hasCycle(head)); // Output: true
    }
}
```

Merge Sort on Linked Lists

Explanation:

Merge Sort is a divide and conquer algorithm which works well for linked lists because:

- It doesn't require random access.
- Splitting and merging linked lists can be done efficiently.

Steps:

1. Find the middle of the linked list (using slow and fast pointer).
2. Divide the list into two halves.
3. Recursively sort each half.
4. Merge the two sorted halves.

Code (Java):

```
public class LinkedListMergeSort {

    static class Node {
        int data;
        Node next;
        Node(int d) { data = d; next = null; }
    }

    // Function to merge two sorted lists
    public static Node sortedMerge(Node a, Node b) {
        if (a == null) return b;
        if (b == null) return a;

        Node result;
        if (a.data <= b.data) {
            result = a;
            result.next = sortedMerge(a.next, b);
        } else {
            result = b;
```

```
        result.next = sortedMerge(a, b.next);
    }
    return result;
}

// Function to find the middle node
public static Node getMiddle(Node head) {
    if (head == null) return head;

    Node slow = head;
    Node fast = head.next;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}

// Merge Sort function
public static Node mergeSort(Node head) {
    if (head == null || head.next == null)
        return head;

    Node middle = getMiddle(head);
    Node nextOfMiddle = middle.next;

    middle.next = null;  // Split the list into two halves

    Node left = mergeSort(head);
```

```java
    Node right = mergeSort(nextOfMiddle);

    Node sortedList = sortedMerge(left, right);
    return sortedList;
}

// Helper function to print list
public static void printList(Node head) {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    Node head = new Node(4);
    head.next = new Node(2);
    head.next.next = new Node(1);
    head.next.next.next = new Node(3);

    System.out.println("Original List:");
    printList(head);

    head = mergeSort(head);

    System.out.println("Sorted List:");
    printList(head);
}
```

}

LRU Cache Implementation using Doubly Linked List and HashMap

Explanation:

An LRU (Least Recently Used) cache evicts the least recently used item when it reaches capacity.

Key data structures:

- **Doubly Linked List:** To maintain order of usage — head is most recently used, tail is least recently used.
- **HashMap:** For O(1) access to nodes in the list by key.

Operations:

- **get(key):** Return value if key exists, move node to head (mark as recently used).
- **put(key, value):** Add a new node or update existing node and move to head. If capacity is exceeded, remove tail node.

Code (Java):

```
import java.util.*;


class LRUCache {


    class Node {
        int key, value;
        Node prev, next;
        Node(int k, int v) { key = k; value = v; }
    }


    private void addNode(Node node) {
        node.prev = head;
        node.next = head.next;
```

```
    head.next.prev = node;
    head.next = node;
}

private void removeNode(Node node) {
    Node prevNode = node.prev;
    Node nextNode = node.next;

    prevNode.next = nextNode;
    nextNode.prev = prevNode;
}

private void moveToHead(Node node) {
    removeNode(node);
    addNode(node);
}

private Node popTail() {
    Node res = tail.prev;
    removeNode(res);
    return res;
}

private Map<Integer, Node> cache = new HashMap<>();
private int size;
private int capacity;
private Node head, tail;

public LRUCache(int capacity) {
```

```
        this.size = 0;
        this.capacity = capacity;

        head = new Node(0, 0);
        tail = new Node(0, 0);

        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        Node node = cache.get(key);
        if (node == null) return -1;

        // Move accessed node to head
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        Node node = cache.get(key);

        if (node == null) {
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            addNode(newNode);
            size++;

            if (size > capacity) {
                Node tailNode = popTail();
```

```
            cache.remove(tailNode.key);
            size--;
          }
        } else {
          node.value = value;
          moveToHead(node);
        }
      }

    public static void main(String[] args) {
        LRUCache lruCache = new LRUCache(2);


        lruCache.put(1, 1);
        lruCache.put(2, 2);
        System.out.println(lruCache.get(1)); // returns 1
        lruCache.put(3, 3);              // evicts key 2
        System.out.println(lruCache.get(2)); // returns -1 (not found)
        lruCache.put(4, 4);              // evicts key 1
        System.out.println(lruCache.get(1)); // returns -1 (not found)
        System.out.println(lruCache.get(3)); // returns 3
        System.out.println(lruCache.get(4)); // returns 4
    }
}
```

Skip Lists (Advanced)

Explanation:

Skip Lists are a probabilistic data structure that allow fast search, insertion, and deletion operations — expected O(log n) time complexity. They can be seen as a multi-level linked list with "express lanes" that skip several nodes.

- Each node has multiple forward pointers.
- Higher levels allow skipping more nodes.

- Level assignment is random (usually coin toss).

Skip lists are used as an alternative to balanced trees.

Basic Idea:

- Level 0: regular linked list.
- Higher levels: sparse linked lists skipping nodes.
- Searching starts from top level and moves forward, dropping levels as needed.

Simplified Code Outline (Java):

Due to complexity, here’s a basic structure for a skip list node and insertion logic:

```
import java.util.Random;

class SkipListNode {
    int value;
    SkipListNode[] forward;

    public SkipListNode(int level, int value) {
        forward = new SkipListNode[level + 1];
        this.value = value;
    }
}

public class SkipList {
    private final int MAX_LEVEL = 16;
    private final float P = 0.5f;
    private int level = 0;
    private SkipListNode header = new SkipListNode(MAX_LEVEL, Integer.MIN_VALUE);
    private Random random = new Random();

    private int randomLevel() {
```

```
    int lvl = 0;
    while (random.nextFloat() < P && lvl < MAX_LEVEL) {
        lvl++;
    }
    return lvl;
}

public void insert(int value) {
    SkipListNode[] update = new SkipListNode[MAX_LEVEL + 1];
    SkipListNode x = header;

    for (int i = level; i >= 0; i--) {
        while (x.forward[i] != null && x.forward[i].value < value) {
            x = x.forward[i];
        }
        update[i] = x;
    }

    x = x.forward[0];
    if (x == null || x.value != value) {
        int lvl = randomLevel();
        if (lvl > level) {
            for (int i = level + 1; i <= lvl; i++) {
                update[i] = header;
            }
            level = lvl;
        }

        SkipListNode newNode = new SkipListNode(lvl, value);
        for (int i = 0; i <= lvl; i++) {
```

```
            newNode.forward[i] = update[i].forward[i];
            update[i].forward[i] = newNode;
        }
    }
}

public boolean search(int value) {
    SkipListNode x = header;
    for (int i = level; i >= 0; i--) {
        while (x.forward[i] != null && x.forward[i].value < value) {
            x = x.forward[i];
        }
    }
    x = x.forward[0];
    return x != null && x.value == value;
}

// Additional delete and display methods can be added similarly.

public static void main(String[] args) {
    SkipList list = new SkipList();
    list.insert(3);
    list.insert(6);
    list.insert(7);
    list.insert(9);
    list.insert(12);
    list.insert(19);
    list.insert(17);

    System.out.println("Search 9: " + list.search(9));   // true
```

```
        System.out.println("Search 15: " + list.search(15)); // false
    }
}
```

ADVANCED LINKED LIST CODING PROBLEMS

1. Detect and Remove Cycle in a Linked List

Given a linked list that might contain a cycle, detect the cycle and remove it, making the list linear.

- Input: Head of linked list
- Output: Head of linear linked list (no cycle)
- Constraints: Do it without using extra space.

---

2. Flatten a Multilevel Doubly Linked List

Each node may have a child pointer to another doubly linked list. Flatten the list so all nodes appear in a single-level doubly linked list.

- Input: Head of a multilevel doubly linked list
- Output: Head of flattened list

---

3. Merge k Sorted Linked Lists

Given k sorted linked lists, merge them into a single sorted linked list efficiently.

- Hint: Use a min-heap (priority queue) to achieve O(N log k) time where N is total nodes.

---

4. LRU Cache with Time-to-Live (TTL)

Extend LRU Cache to support TTL for each key — items expire after given time. Ensure efficient get, put, and expiry checks.

- Design your own data structures to handle TTL.

---

5. Copy List with Random Pointer

Given a linked list where each node has an additional random pointer, create a deep copy of the list.

- The random pointer can point to any node or null.

- O(n) time and O(1) extra space solution.

---

6. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of the list k at a time and return its modified list.

- If the number of nodes is not a multiple of k, leave the last nodes as is.

---

7. Add Two Numbers Represented by Linked Lists

Each linked list represents a number in reverse order; add the two numbers and return the sum as a linked list.

---

8. Find the Intersection Point of Two Linked Lists

Given two singly linked lists, find the node at which the two lists intersect.

- Optimize for O(m + n) time and O(1) space.

---

9. Skip List Implementation with Delete and Range Search

Implement a fully functional skip list supporting insertion, deletion, and range search efficiently.

---

10. Merge Sort on Linked List without Recursion

Implement the merge sort on linked list iteratively (bottom-up approach).

Bonus Challenge:

Design a Data Structure for a Playlist

- Support adding songs, removing songs, playing next/previous song, and random shuffle — all in average O(1) time.
- Hint: Combine linked list and hash map.

RECOMMENDED YOUTUBE LINKS

1. https://youtu.be/oAja8-Ulz6o?si=OOXWChdClisb0zJo
2. https://youtu.be/58YbpRDc4yw?si=UQsDkWsWYludK3It
3. https://youtu.be/SMIq13-FZSE?si=px1hkwN3d4lVF-AR
4. https://youtu.be/j5hWoyjrl14?si=drLOkrPgT9Il0_i
5. https://youtu.be/Nq7ok-OyEpg?si=XoF7FKr9kj_AfJhV

6. https://youtu.be/cL4gHVuFOvk?si=RqHg2xh2fnzyO6dD
7. https://youtu.be/Hj_rA0dhr2I?si=u-FRRCZAtMrd238X
8. https://youtube.com/playlist?list=PLGjplNEQ1it-OKRcYlCEDpTiIB1YOcvn6&si=XNwXhQP9ItnYci4k
9. https://youtube.com/playlist?list=PLgUwDviBIf0rAuz8tVcM0AymmhTRsfaLU&si=IU6_BH-3wlb0WqXg
10. https://youtube.com/playlist?list=PLnccP3XNVxGrks-guEVjE1xj9V9YC5oQ7&si=rNnzQHDDm96WMEqQ