Linked List Cheatsheet

Topic Overview

Linked Lists in Java are dynamic data structures where elements (nodes) are linked via pointers, offering flexibility over arrays. This cheatsheet covers techniques for manipulation and problem-solving.

Prerequisites

Arrays

List of Subtopics

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Insertion
- Deletion
- Reverse Linked List
- Detect Loop
- Merge Two Sorted Lists
- Find Middle Node
- Remove Nth Node from End

Key Concepts Explained

- **Singly Linked List**: Each node contains data and a pointer to the next node, suitable for sequential access.
- **Doubly Linked List**: Nodes have pointers to both next and previous nodes, enabling bidirectional traversal.
- **Detect Loop**: Identifies cycles using techniques like Floyd's cycle-finding algorithm.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Singly Linked List**:
  - **Algorithm**:
    1. Create a node class with data and a next pointer.
    2. Initialize the head to null for an empty list.
    3. Traverse by starting at head and following next pointers until null.
    4. Access or modify nodes based on the current pointer.
  - **Context**: Basic structure for dynamic size, with O(n) traversal time.
- **Doubly Linked List**:
  - **Algorithm**:
    1. Define a node with data, next, and previous pointers.
    2. Set head and tail pointers, initializing both to null.
    3. Traverse forward using next or backward using previous.
    4. Update both pointers during insertions or deletions.
  - **Context**: Allows bidirectional access, with O(n) traversal time.
- **Circular Linked List**:
  - **Algorithm**:
    1. Create nodes with data and next pointers, linking the last node to the head.
    2. Initialize head and ensure the last node's next points to head.
    3. Traverse by starting at head and continuing until back at head.
    4. Handle insertions by adjusting the circular link.
  - **Context**: Useful for cyclic data, with O(n) traversal time.
- **Insertion**:
  - **Algorithm**:
    1. For head insertion, create a new node, set its next to head, and update head.
    2. For middle insertion, traverse to the node before the position, adjust pointers.
    3. For tail insertion, traverse to the last node, set its next to the new node.
    4. Update pointers and handle edge cases like empty list.
  - **Context**: O(1) for head, O(n) for middle or tail due to traversal.
- **Deletion**:
  - **Algorithm**:
    1. For head deletion, update head to head.next and free the old head.

2. For middle deletion, traverse to the previous node, adjust its next to skip the target.
3. For tail deletion, traverse to the second-to-last node, set its next to null.
4. Handle cases where the list becomes empty.

- **Context**: O(1) for head, O(n) for middle or tail.

- **Reverse Linked List**:
  - **Algorithm**:
    1. Initialize three pointers: prev (null), current (head), next (null).
    2. For each node, save next = current.next, set current.next = prev.
    3. Move prev to current, current to next, repeat until current is null.
    4. Set head to prev as the new head.
  - **Context**: In-place reversal in O(n) time.

- **Detect Loop**:
  - **Algorithm**:
    1. Use two pointers: slow moves one step, fast moves two steps.
    2. If they meet, a loop exists; if fast reaches null, no loop.
    3. To find loop start, move slow to head, move both one step until they meet again.
  - **Context**: Floyd's algorithm, O(n) time, O(1) space.

- **Merge Two Sorted Lists**:
  - **Algorithm**:
    1. Create a dummy node to build the result list.
    2. Compare heads of both lists, add the smaller value to the result.
    3. Move the pointer of the used list and repeat comparison.
    4. Attach the remaining nodes of the non-empty list.
    5. Return dummy.next as the merged list.
  - **Context**: O(n+m) time, where n and m are list lengths.

- **Find Middle Node**:
  - **Algorithm**:
    1. Use two pointers: slow moves one step, fast moves two steps.
    2. Continue until fast reaches the end or null.
    3. Slow will be at the middle node when fast finishes.
  - **Context**: O(n) time, O(1) space using the two-pointer technique.

- **Remove Nth Node from End**:
  - **Algorithm**:
    1. Use two pointers: fast starts n nodes ahead of slow.

2. Move both pointers until fast reaches the end.
3. Slow will be before the node to delete; adjust its next pointer.
4. Handle edge case where n equals list length.

– **Context**: O(n) time, O(1) space.

Common LeetCode Problems with Approaches

- **Reverse Linked List (206)**: Use the three-pointer method to reverse pointers iteratively.
- **Detect Cycle (141)**: Apply Floyd's cycle-finding algorithm with slow and fast pointers.
- **Merge Two Sorted Lists (21)**: Compare and merge nodes using a dummy node approach.
- **Middle of the Linked List (876)**: Use the two-pointer technique to find the middle.
- **Remove Nth Node From End of List (19)**: Use two pointers with an n-step offset.
- **Add Two Numbers (2)**: Traverse both lists, adding digits and handling carry.

Time & Space Complexities

- Access: O(n)
- Search: O(n)
- Insert/Delete: O(1) for head, O(n) for middle
- Space: O(1) for in-place, O(n) for new list

Important Tips & Tricks

- Use dummy nodes to simplify head insertions or deletions.
- Handle edge cases like empty lists or single nodes.
- Leverage two-pointer techniques for efficient traversal.
- Check for loops before operations on potentially cyclic lists.
- Optimize space by reusing nodes where possible.