Dynamic Arrays & Linked Lists

Competitive Programming Training Guide

Target Audience: Freshers | Duration: 6 Hours

Table of Contents

---

1. Introduction to Data Structures

Why Data Structures Matter in Competitive Programming

- Efficient data organization
- Optimized time and space complexity
- Better algorithm design
- Essential for technical interviews

Static Arrays - The Limitations

```java
int[] arr = new int[100]; // Fixed size - problems if we need more!
```

Problems: - Fixed size determined at compile time - Wasted memory if size overestimated - Insufficient space if size underestimated - Costly resizing requires creating new array and copying elements

---

2. Dynamic Arrays

Concept & Implementation

Dynamic arrays grow automatically when needed (e.g., `ArrayList` in Java, `Vector`).

How it works: 1. Start with initial capacity (e.g., 10 elements) 2. When full, create new array with 1.5x the size (in Java) 3. Copy all elements to new array 4. Continue operations

Amortized Analysis

Key Insight: While occasional resizing is expensive (O(n)), most operations are cheap (O(1))

```java
// Java ArrayList example
import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>(); // Initial capacity 10
list.add(1); // Size: 1
list.add(2); // Size: 2
// ... continues until capacity 10 is reached
list.add(11); // Capacity: 15 (increased), copies elements
```

Time Complexity: - ✅ Access: O(1) - ✅ Append (amortized): O(1) - ❌ Insert at middle: O(n) - ❌ Delete from middle: O(n)

Hands-On Practice

```java
// Problem 1: Monitor behavior
ArrayList<Integer> list = new ArrayList<>();
for(int i = 0; i < 20; i++) {
    list.add(i);
    System.out.println("Size: " + list.size());
    // Note: Capacity is not directly visible in ArrayList
}

// Problem 2: Reverse a list in-place
void reverseList(ArrayList<Integer> list) {
    int left = 0, right = list.size() - 1;
    while(left < right) {
        int temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
        left++;
        right--;
    }
}

// Problem 3: Remove even numbers
ArrayList<Integer> removeEvens(ArrayList<Integer> list) {
    ArrayList<Integer> result = new ArrayList<>();
    for(int num : list) {
```

```
        if(num % 2 != 0) {
            result.add(num);
        }
    }
    return result;
}
```

---

3. Linked Lists Fundamentals

Singly Linked List

Node Structure:

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }

    ListNode(int val, ListNode next) {
        this.val = val;
        this.next = next;
    }
}
```

Visual Representation:

```
Head → [1|•] → [2|•] → [3|•] → [4|•] → NULL
```

Key Characteristics: - Non-contiguous memory allocation - Dynamic size - Efficient insertions/deletions at head

Basic Operations

```
class LinkedList {
    private ListNode head;

    public LinkedList() {
        head = null;
    }

    // Insert at head - O(1)
    public void insertAtHead(int data) {
```

```
        ListNode newNode = new ListNode(data);
        newNode.next = head;
        head = newNode;
    }

    // Traversal - O(n)
    public void printList() {
        ListNode current = head;
        while(current != null) {
            System.out.print(current.val + " → ");
            current = current.next;
        }
        System.out.println("NULL");
    }
}
```

Doubly Linked List

```
class DoublyListNode {
    int val;
    DoublyListNode next;
    DoublyListNode prev;

    DoublyListNode(int val) {
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}
```

Advantages: - Bidirectional traversal - O(1) deletion of known nodes - Easier reverse traversal

Disadvantages: - More memory overhead (extra pointer) - Slightly more complex implementation

Comparison: Arrays vs Linked Lists

Operation	Array/ArrayList	Linked List
Access	O(1)	O(n)
Insert at head	O(n)	O(1)
Delete at head	O(n)	O(1)
Memory	Contiguous	Fragmented
Size	Dynamic	Dynamic

When to use Linked Lists: - Frequent insertions/deletions at ends - Unknown size requirements - Implementing stacks, queues, hash tables

---

4. Core Linked List Operations

Essential Techniques

1. Find Middle Node (Tortoise and Hare)

```
public ListNode findMiddle(ListNode head) {
    if(head == null) return null;

    ListNode slow = head;
    ListNode fast = head;

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

    return slow;
}
```

2. Reverse Linked List (Iterative) - MUST KNOW

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode current = head;
    ListNode next = null;

    while(current != null) {
        next = current.next;  // Store next node
        current.next = prev;  // Reverse link
        prev = current;       // Move prev forward
        current = next;       // Move current forward
    }

    return prev; // New head
}
```

3. Delete Node Given Only Pointer to It

```
public void deleteNode(ListNode nodeToDelete) {
    // Copy next node's data to current node
    nodeToDelete.val = nodeToDelete.next.val;
```

```
    // Point to next's next
    nodeToDelete.next = nodeToDelete.next.next;
    // Note: In Java, garbage collection handles deletion
}
```

4. Remove Nth Node From End (Two Pointer)

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);  // Dummy node to handle edge cases
    dummy.next = head;

    ListNode first = dummy;
    ListNode second = dummy;

    // Move first pointer n+1 steps ahead
    for(int i = 0; i <= n; i++) {
        first = first.next;
    }

    // Move both until first reaches end
    while(first != null) {
        first = first.next;
        second = second.next;
    }

    // Remove the node
    second.next = second.next.next;

    return dummy.next;
}
```

---

5. Problem Solving Techniques

Cycle Detection (Floyd's Algorithm)

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

    ListNode slow = head;
    ListNode fast = head;

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

```
        if(slow == fast) {
            return true;  // Cycle detected
        }
    }

    return false;
}
```

Find Cycle Starting Point

```
public ListNode detectCycleStart(ListNode head) {
    if(head == null) return null;

    ListNode slow = head;
    ListNode fast = head;
    boolean hasCycle = false;

    // Detection phase
    while(fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        if(slow == fast) {
            hasCycle = true;
            break;
        }
    }

    if(!hasCycle) return null;

    // Find starting point
    slow = head;
    while(slow != fast) {
        slow = slow.next;
        fast = fast.next;
    }

    return slow;
}
```

Check Palindrome

```
public boolean isPalindrome(ListNode head) {
    if(head == null || head.next == null)
        return true;
```

```
    // Step 1: Find middle
    ListNode slow = head;
    ListNode fast = head;

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

    // Step 2: Reverse second half
    ListNode secondHalf = reverseList(slow.next);

    // Step 3: Compare both halves
    ListNode firstHalf = head;
    ListNode temp = secondHalf;
    boolean result = true;

    while(secondHalf != null) {
        if(firstHalf.val != secondHalf.val) {
            result = false;
            break;
        }
        firstHalf = firstHalf.next;
        secondHalf = secondHalf.next;
    }

    // Step 4: Restore the list (optional)
    reverseList(temp);

    return result;
}
```

---

6. Advanced Problems & Applications

Merge Two Sorted Lists

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    while(l1 != null && l2 != null) {
        if(l1.val <= l2.val) {
            current.next = l1;
```

```
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    // Attach remaining elements
    if(l1 != null) current.next = l1;
    if(l2 != null) current.next = l2;

    return dummy.next;
}
```

Add Two Numbers

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;
    int carry = 0;

    while(l1 != null || l2 != null || carry != 0) {
        int sum = carry;

        if(l1 != null) {
            sum += l1.val;
            l1 = l1.next;
        }

        if(l2 != null) {
            sum += l2.val;
            l2 = l2.next;
        }

        carry = sum / 10;
        current.next = new ListNode(sum % 10);
        current = current.next;
    }

    return dummy.next;
}
```

LRU Cache Implementation

```java
import java.util.*;

class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode prev;
        DLinkedNode next;
    }

    private void addNode(DLinkedNode node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(DLinkedNode node) {
        DLinkedNode prev = node.prev;
        DLinkedNode next = node.next;
        prev.next = next;
        next.prev = prev;
    }

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

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

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

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        head = new DLinkedNode();
```

```
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if(node == null) return -1;
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);
        if(node == null) {
            DLinkedNode newNode = new DLinkedNode();
            newNode.key = key;
            newNode.value = value;
            cache.put(key, newNode);
            addNode(newNode);
            ++size;
            if(size > capacity) {
                DLinkedNode tail = popTail();
                cache.remove(tail.key);
                --size;
            }
        } else {
            node.value = value;
            moveToHead(node);
        }
    }
}
```

---

7. Practice Problems

Easy Level

1. Reverse Linked List (LeetCode #206)
2. Merge Two Sorted Lists (LeetCode #21)
3. Linked List Cycle (LeetCode #141)
4. Middle of the Linked List (LeetCode #876)
5. Remove Duplicates from Sorted List (LeetCode #83)

Medium Level

1. Add Two Numbers (LeetCode #2)
2. Remove Nth Node From End of List (LeetCode #19)
3. Copy List with Random Pointer (LeetCode #138)
4. LRU Cache (LeetCode #146)
5. Flatten a Multilevel Doubly Linked List (LeetCode #430)

Hard Level

1. Merge k Sorted Lists (LeetCode #23)
2. Reverse Nodes in k-Group (LeetCode #25)
3. First Missing Positive (LeetCode #41)

---

8. Cheat Sheets

Dynamic Arrays Cheat Sheet

```java
// Java ArrayList
ArrayList<Integer> list = new ArrayList<>();
list.add(1);                    // Add to end
list.remove(list.size()-1);     // Remove from end
list.add(0, 0);                 // Insert at beginning
list.remove(0);                 // Remove from beginning
list.size();                    // Current size
list.get(0);                    // Random access
list.set(0, 5);                 // Update element

// Java LinkedList (Doubly Linked List)
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.addFirst(1);         // Add to head
linkedList.addLast(2);          // Add to tail
linkedList.removeFirst();       // Remove from head
linkedList.removeLast();        // Remove from tail
```

Linked List Common Patterns

Dummy Node Technique

```java
// Use when head might change
ListNode dummy = new ListNode(0);
dummy.next = head;
// ... operations
return dummy.next;
```

Two Pointer Patterns

- Fast & Slow: Middle detection, cycle detection
- First & Second: Nth node from end, intersection point

Edge Cases to Always Check

1. Empty list (head == null)
2. Single node list
3. Operations at head
4. Operations at tail
5. Cyclic lists

Time Complexity Summary

Operation	SLL	DLL	Dynamic Array
Access	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)
Insert at head	O(1)	O(1)	O(n)
Insert at tail	O(n)	O(1)	O(1) am.
Delete head	O(1)	O(1)	O(n)
Delete tail	O(n)	O(1)	O(1)
Memory	O(n)	O(n)	O(n)

Competitive Programming Tips

1. Always draw diagrams before coding linked list problems
2. Use dummy nodes to simplify edge cases
3. Check for null pointers in every operation
4. Test with: empty list, single node, two nodes, large lists
5. For interviews: Explain your approach before coding

Common Mistakes to Avoid

- Forgetting to update all relevant pointers
- Not handling the head pointer correctly
- Infinite loops in cyclic lists
- Off-by-one errors in traversal
- Not considering edge cases

Key Java Collections to Know

- `ArrayList<E>` - Dynamic array implementation
- `LinkedList<E>` - Doubly linked list implementation
- `Vector<E>` - Thread-safe dynamic array (legacy)
- `Stack<E>` - LIFO stack (extends Vector)

Practice Recommendation: Solve at least 10-15 problems from each difficulty level to build strong intuition for linked list manipulations.

Remember: Understanding reference manipulation is key to mastering linked lists. Practice visualizing the connections and pointer movements!

This material covers the essential concepts for the 6-hour training session. Focus on understanding the patterns rather than memorizing code. Happy coding! "'