# Dynamic Arrays & Linked Lists

## Competitive Programming Training Guide

**Target Audience:** Freshers | **Duration:** 6 Hours | **Language:** Python

---

# 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

### Python Lists - Dynamic by Nature

```
# Python lists are dynamic arrays by default
arr = [1, 2, 3, 4, 5]  # No fixed size!
arr.append(6)  # Automatically grows
```

**Advantages of Python Lists:**

- Dynamic sizing handled automatically

- No manual memory management
- Rich built-in methods
- Flexible with mixed data types

---

# 2. Dynamic Arrays in Python

## Python List Internals

Python lists are implemented as dynamic arrays that automatically resize when needed.

**How it works internally:**

1. Starts with initial capacity
2. When full, creates new larger array (typically ~1.125x growth)
3. Copies elements to new array
4. Continues operations

## Time Complexity Analysis

```
# Python list operations complexity
my_list = [1, 2, 3, 4, 5]

# O(1) operations
my_list.append(6)        # Amortized O(1)
my_list.pop()            # Remove from end
my_list[i]               # Random access

# O(n) operations
my_list.insert(0, 0)     # Insert at beginning
my_list.pop(0)           # Remove from beginning
my_list.remove(3)        # Remove by value
element in my_list       # Search
```

## Hands-On Practice

```python
# Problem 1: Monitor list growth
import sys

my_list = []
for i in range(20):
    my_list.append(i)
    print(f"Size: {len(my_list)}, Memory: {sys.getsizeof(my_list)} bytes")

# Problem 2: Reverse a list in-place
def reverse_list(lst):
    left, right = 0, len(lst) - 1
    while left < right:
        lst[left], lst[right] = lst[right], lst[left]
        left += 1
        right -= 1

# Problem 3: Remove even numbers using list comprehension
def remove_evens(lst):
    return [x for x in lst if x % 2 != 0]

# Alternative using filter with lambda
def remove_evens_lambda(lst):
    return list(filter(lambda x: x % 2 != 0, lst))

# Test the functions
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original:", numbers)
print("After removing evens:", remove_evens(numbers))
```

# 3. Linked Lists Fundamentals

## Node Class Implementation

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

    def __repr__(self):
        return f"ListNode({self.val})"
```

# Singly Linked List Implementation

```python
class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, val):
        """Add node to the end of the list"""
        new_node = ListNode(val)
        if not self.head:
            self.head = new_node
            return

        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def prepend(self, val):
        """Add node to the beginning of the list"""
        new_node = ListNode(val)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Display the linked list"""
        elements = []
        current = self.head
        while current:
            elements.append(current.val)
            current = current.next
        return " -> ".join(map(str, elements)) + " -> None"

    def __repr__(self):
        return self.display()

# Example usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.prepend(0)
print(ll)  # Output: 0 -> 1 -> 2 -> 3 -> None
```

## Doubly Linked List

```
class DoublyListNode:
    def __init__(self, val=0, next=None, prev=None):
        self.val = val
        self.next = next
        self.prev = prev

    def __repr__(self):
        return f"DoublyListNode({self.val})"

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def append(self, val):
        new_node = DoublyListNode(val)
        if not self.head:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node
```

## Comparison: Python List vs Linked List

| Operation | Python List | Linked List |
|---|---|---|
| Access | O(1) | O(n) |
| Insert at head | O(n) | O(1) |
| Delete at head | O(n) | O(1) |
| Memory | Contiguous | Fragmented |
| Built-in methods | Many | Few |

**When to use Linked Lists in Python:**

- Frequent insertions/deletions at ends
- Educational purposes
- Implementing other data structures
- When memory fragmentation isn't a concern

---

# 4. Core Linked List Operations

# Essential Techniques

## 1. Find Middle Node (Tortoise and Hare)

```
def find_middle(head):
    if not head:
        return None

    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow

# Example
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
middle = find_middle(head)
print(f"Middle node: {middle.val}")  # Output: 3
```

## 2. Reverse Linked List (Iterative) - MUST KNOW

```
def reverse_list(head):
    prev = None
    current = head

    while current:
        next_node = current.next  # Store next node
        current.next = prev       # Reverse link
        prev = current            # Move prev forward
        current = next_node       # Move current forward

    return prev  # New head

# Example
original = ListNode(1, ListNode(2, ListNode(3)))
reversed_head = reverse_list(original)
```

## 3. Reverse Linked List (Recursive)

```python
def reverse_list_recursive(head):
    if not head or not head.next:
        return head

    new_head = reverse_list_recursive(head.next)
    head.next.next = head
    head.next = None

    return new_head
```

### 4. Remove Nth Node From End (Two Pointer)

```python
def remove_nth_from_end(head, n):
    dummy = ListNode(0)
    dummy.next = head

    first = second = dummy

    # Move first pointer n+1 steps ahead
    for _ in range(n + 1):
        first = first.next

    # Move both until first reaches end
    while first:
        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)

```python
def has_cycle(head):
    if not head:
        return False

    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True

    return False
```

# Find Cycle Starting Point

```python
def detect_cycle_start(head):
    if not head:
        return None

    slow = fast = head
    has_cycle = False

    # Detection phase
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            has_cycle = True
            break

    if not has_cycle:
        return None

    # Find starting point
    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return slow
```

## Check Palindrome

```python
def is_palindrome(head):
    if not head or not head.next:
        return True

    # Step 1: Find middle
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse second half
    second_half = reverse_list(slow)

    # Step 3: Compare both halves
    first_half = head
    temp = second_half
    result = True

    while second_half:
        if first_half.val != second_half.val:
            result = False
            break
        first_half = first_half.next
        second_half = second_half.next

    # Step 4: Restore the list
    reverse_list(temp)

    return result
```

---

# 6. Advanced Problems & Applications

## Merge Two Sorted Lists

```python
def merge_two_lists(l1, l2):
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # Attach remaining elements
    current.next = l1 if l1 else l2

    return dummy.next
```

## Add Two Numbers (Digits stored in reverse)

```python
def add_two_numbers(l1, l2):
    dummy = ListNode(0)
    current = dummy
    carry = 0

    while l1 or l2 or carry:
        total = carry

        if l1:
            total += l1.val
            l1 = l1.next

        if l2:
            total += l2.val
            l2 = l2.next

        carry = total // 10
        current.next = ListNode(total % 10)
        current = current.next

    return dummy.next
```

# LRU Cache Implementation

```python
class LRUCache:
    class Node:
        def __init__(self, key, val):
            self.key = key
            self.val = val
            self.prev = None
            self.next = None

    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}
        self.head = self.Node(0, 0)
        self.tail = self.Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node):
        """Remove a node from the linked list"""
        prev_node = node.prev
        next_node = node.next
        prev_node.next = next_node
        next_node.prev = prev_node

    def _add(self, node):
        """Add node right after head"""
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add(node)
            return node.val
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self._remove(self.cache[key])

        node = self.Node(key, value)
```

```
        self.cache[key] = node
        self._add(node)

        if len(self.cache) > self.capacity:
            # Remove LRU node
            lru = self.tail.prev
            self._remove(lru)
            del self.cache[lru.key]
```

## Flatten a Multilevel Doubly Linked List

```
def flatten(head):
    if not head:
        return head

    current = head
    while current:
        if current.child:
            next_node = current.next
            child_head = flatten(current.child)

            current.next = child_head
            child_head.prev = current
            current.child = None

            # Find tail of child list
            tail = child_head
            while tail.next:
                tail = tail.next

            # Connect tail to next node
            if next_node:
                tail.next = next_node
                next_node.prev = tail

        current = current.next

    return head
```

---

# 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

## Python List Cheat Sheet

```python
# Basic Operations
lst = []                    # Create empty list
lst = [1, 2, 3]            # Create with elements
lst.append(4)               # Add to end: O(1)
lst.insert(0, 0)            # Insert at index: O(n)
lst.extend([5, 6])          # Add multiple: O(k)
lst.pop()                   # Remove from end: O(1)
lst.pop(0)                  # Remove from front: O(n)
lst.remove(3)               # Remove by value: O(n)
lst[0]                      # Access: O(1)
lst[1:4]                    # Slicing: O(k)

# List Comprehensions
squares = [x**2 for x in range(10)]
evens = [x for x in lst if x % 2 == 0]

# Built-in Functions
len(lst)                    # Length
sum(lst)                    # Sum
min(lst), max(lst)          # Min/Max
sorted(lst)                 # Sorted copy
lst.sort()                  # In-place sort
```

# Linked List Common Patterns

## Dummy Node Technique

```python
# Use when head might change
dummy = 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 is None)
2. Single node list
3. Operations at head

4. Operations at tail
5. Cyclic lists

## Time Complexity Summary

| Operation | Python List | Linked List |
|---|---|---|
| Access | O(1) | O(n) |
| Search | O(n) | O(n) |
| Insert at head | O(n) | O(1) |
| Insert at tail | O(1) | O(1)* |
| Delete head | O(n) | O(1) |
| Delete tail | O(1) | O(n)* |
| Memory | Contiguous | Fragmented |

*With tail pointer

---

# Competitive Programming Tips

1. **Always draw diagrams** before coding linked list problems
2. **Use dummy nodes** to simplify edge cases
3. **Check for None** in every operation
4. **Test with:** empty list, single node, two nodes, large lists
5. **Use Python's built-in list** for most practical problems
6. **Learn linked lists** for interviews and understanding fundamentals

## 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

## Python-Specific Tips

```python
# Use list comprehensions for filtering
evens = [x for x in lst if x % 2 == 0]

# Use built-in functions when possible
reversed_list = lst[::-1]  # Reverse a list

# Use enumerate for index tracking
for i, val in enumerate(lst):
    print(f"Index {i}: {val}")

# Use zip for multiple list iteration
for a, b in zip(list1, list2):
    print(a + b)
```

---

# Python Collections Module

```python
from collections import deque

# deque is optimized for head/tail operations
dq = deque([1, 2, 3])
dq.appendleft(0)    # O(1)
dq.popleft()        # O(1)
```

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

**Remember:** While Python lists are usually sufficient, understanding linked lists is crucial for interviews and advanced data structures!

---

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

---

## Quick Reference: Removing Even Elements in Python

```python
# Method 1: List Comprehension (Recommended)
def remove_evens_comprehension(lst):
    return [x for x in lst if x % 2 != 0]

# Method 2: Filter with Lambda
def remove_evens_filter(lst):
    return list(filter(lambda x: x % 2 != 0, lst))

# Method 3: In-place removal
def remove_evens_inplace(lst):
    lst[:] = [x for x in lst if x % 2 != 0]

# Method 4: Using while loop (educational)
def remove_evens_while(lst):
    i = 0
    while i < len(lst):
        if lst[i] % 2 == 0:
            lst.pop(i)
        else:
            i += 1
    return lst

# Test
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original:", numbers)
print("Comprehension:", remove_evens_comprehension(numbers))
print("Filter:", remove_evens_filter(numbers))
```

**Output:**

```
Original: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Comprehension: [1, 3, 5, 7, 9]
Filter: [1, 3, 5, 7, 9]
```

To convert this markdown to PDF, you can use:

1. **Pandoc:** `pandoc python-training.md -o python-training.pdf`
2. **VS Code** with Markdown PDF extension
3. **Online converters** like markdown-pdf.com

The Python version focuses on practical implementations while maintaining the core data structure concepts!