# Solutions to DSA Questions 51-80 (Linked Lists, Stacks, Queues, Trees)

For 1-2 Years Experience Roles at EPAM Compiled on September 26, 2025

## Introduction

This document provides detailed solutions for 30 Data Structures and Algorithms (DSA) problems (questions 51 to 80) from the Linked Lists, Stacks, Queues, and Trees categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity. Dynamic programming problems include both memoization and tabulation approaches where applicable.

## Contents

# 1 Add Two Numbers as Linked Lists

## 1.1 Problem Statement

Given two non-empty linked lists representing non-negative integers (digits in reverse order), add them and return the sum as a linked list.

## 1.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 2->4->3, l2 = 5->6->4 $\rightarrow$ Output: 7->0->8 (342 + 465 = 807)
- **Test Case 2**: l1 = 0, l2 = 0 $\rightarrow$ Output: 0
- **Test Case 3**: l1 = 9->9->9, l2 = 1 $\rightarrow$ Output: 0->0->0->1
- **Test Case 4**: l1 = 1->8, l2 = 0 $\rightarrow$ Output: 1->8

## 1.3 Algorithm

1. Initialize dummy node and current pointer.
2. Traverse both lists, adding digits and carry.
3. Create new nodes for sum digits, handle carry.
4. Return dummy.next.

**Time Complexity**: $O(\max(n, m))$  **Space Complexity**: $O(\max(n, m))$

## 1.4 Python Solution

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

def add_two_numbers(l1, l2):
    dummy = ListNode(0)
    curr = dummy
    carry = 0

    while l1 or l2 or carry:
        x = l1.val if l1 else 0
        y = l2.val if l2 else 0
        total = x + y + carry
        carry = total // 10
        curr.next = ListNode(total % 10)
        curr = curr.next
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None
    return dummy.next
```

# 2 Intersection of Two Linked Lists

## 2.1 Problem Statement

Given two linked lists, find the node where they intersect (same reference) or return None.

## 2.2 Dry Run on Test Cases

- **Test Case 1**: l1 = 4->1->(8->4->5), l2 = 5->6->1->(8->4->5) → Output: Node 8
- **Test Case 2**: l1 = 1->2, l2 = 3->4 → Output: None
- **Test Case 3**: l1 = None, l2 = 1 → Output: None
- **Test Case 4**: l1 = 1->2->3, l2 = 3 → Output: Node 3

## 2.3 Algorithm

1. Traverse both lists, switching to other list when reaching end.
2. If pointers meet, that's the intersection.
3. If both reach None, no intersection.

**Time Complexity**: $O(n + m)$ **Space Complexity**: $O(1)$

## 2.4 Python Solution

```python
def get_intersection_node(headA, headB):
    if not headA or not headB:
        return None

    a, b = headA, headB
    while a != b:
        a = a.next if a else headB
        b = b.next if b else headA
    return a
```

# 3 Reverse Nodes in k-Group

## 3.1 Problem Statement

Given a linked list, reverse every k nodes and return the new head.

## 3.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, k = 2 → Output: 2->1->4->3->5
- **Test Case 2**: head = 1->2->3->4->5, k = 3 → Output: 3->2->1->4->5
- **Test Case 3**: head = 1, k = 1 → Output: 1
- **Test Case 4**: head = None, k = 2 → Output: None

## 3.3 Algorithm

1. Check if k nodes exist.
2. Reverse k nodes using iterative reversal.
3. Recursively process next k-group.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$ or $O(n/k)$ for recursion

## 3.4 Python Solution

```python
def reverse_k_group(head, k):
    def get_kth(curr, k):
        while curr and k > 0:
            curr = curr.next
            k -= 1
        return curr

    dummy = ListNode(0, head)
    prev_group = dummy

    while head:
        kth = get_kth(head, k - 1)
        if not kth:
            break
        next_group = kth.next
        kth.next = None

        # Reverse current group
        prev = next_group
        curr = head
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        # Connect to previous group
        prev_group.next = prev
        prev_group = head
        head = next_group

    return dummy.next
```

# 4 Palindrome Linked List

## 4.1 Problem Statement

Given a linked list, determine if it is a palindrome.

## 4.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->2->1 → Output: True
- **Test Case 2**: head = 1->2 → Output: False
- **Test Case 3**: head = 1 → Output: True
- **Test Case 4**: head = None → Output: True

## 4.3 Algorithm

1. Find middle using slow and fast pointers.
2. Reverse second half.
3. Compare first and second halves.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

## 4.4 Python Solution

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

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

    # Reverse second half
    second_half = slow.next
    slow.next = None
    prev = None
    while second_half:
        next_node = second_half.next
        second_half.next = prev
        prev = second_half
        second_half = next_node

    # Compare
    first_half = head
    while prev:
        if first_half.val != prev.val:
            return False
        first_half = first_half.next
        prev = prev.next
    return True
```

## 5 Remove Linked List Elements

### 5.1 Problem Statement

Given a linked list and a value, remove all nodes with that value.

### 5.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->6->3->4->6, val = 6 → Output: 1->2->3->4
- **Test Case 2**: head = None, val = 1 → Output: None
- **Test Case 3**: head = 7->7->7, val = 7 → Output: None
- **Test Case 4**: head = 1, val = 2 → Output: 1

### 5.3 Algorithm

1. Use dummy node to handle head removal.
2. Traverse list, skip nodes with given value.
3. Return dummy.next.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 5.4 Python Solution

```python
def remove_elements(head, val):
    dummy = ListNode(0, head)
    curr = dummy

    while curr.next:
        if curr.next.val == val:
            curr.next = curr.next.next
        else:
            curr = curr.next
    return dummy.next
```

# 6 Swap Nodes in Pairs

## 6.1 Problem Statement

Given a linked list, swap every two adjacent nodes and return the head.

## 6.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4 → Output: 2->1->4->3
- **Test Case 2**: head = 1 → Output: 1
- **Test Case 3**: head = None → Output: None
- **Test Case 4**: head = 1->2->3 → Output: 2->1->3

## 6.3 Algorithm

1. If less than 2 nodes, return head.
2. Swap current pair, recursively swap rest.
3. Adjust pointers to connect swapped pairs.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$ for recursion

## 6.4 Python Solution

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

    next_node = head.next
    head.next = swap_pairs(next_node.next)
    next_node.next = head
    return next_node
```

# 7 Odd Even Linked List

## 7.1 Problem Statement

Given a linked list, group odd-indexed nodes together followed by even-indexed nodes.

## 7.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5 → Output: 1->3->5->2->4
- **Test Case 2**: head = 2->1->3->5->6->4->7 → Output: 2->3->6->7->1->5->4
- **Test Case 3**: head = 1 → Output: 1

- **Test Case 4**: head = None → Output: None

### 7.3 Algorithm

1. Maintain odd and even pointers.
2. Link odd nodes together, even nodes together.
3. Connect odd list to even list.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 7.4 Python Solution

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

    odd = head
    even = head.next
    even_head = even

    while even and even.next:
        odd.next = even.next
        odd = odd.next
        even.next = odd.next
        even = even.next

    odd.next = even_head
    return head
```

## 8 Partition List

### 8.1 Problem Statement

Given a linked list and value x, partition list so all nodes less than x come before nodes greater than or equal to x.

### 8.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->4->3->2->5->2, x = 3 → Output: 1->2->2->4->3->5
- **Test Case 2**: head = 2->1, x = 2 → Output: 1->2
- **Test Case 3**: head = None, x = 0 → Output: None
- **Test Case 4**: head = 1, x = 2 → Output: 1

### 8.3 Algorithm

1. Maintain two lists: less and greater.
2. Traverse list, append nodes to appropriate list.
3. Connect less to greater.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 8.4 Python Solution

```python
def partition(head, x):
    less_dummy = ListNode(0)
    greater_dummy = ListNode(0)
    less = less_dummy
    greater = greater_dummy

    curr = head
    while curr:
        if curr.val < x:
            less.next = curr
            less = less.next
        else:
            greater.next = curr
            greater = greater.next
        curr = curr.next

    greater.next = None
    less.next = greater_dummy.next
    return less_dummy.next
```

## 9 Rotate List

### 9.1 Problem Statement

Given a linked list, rotate it to the right by k places.

### 9.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4->5, k = 2 → Output: 4->5->1->2->3
- **Test Case 2**: head = 0->1->2, k = 4 → Output: 2->0->1
- **Test Case 3**: head = 1, k = 1 → Output: 1
- **Test Case 4**: head = None, k = 1 → Output: None

### 9.3 Algorithm

1. Find length and last node.

2. Compute effective k = k % length.
3. Find new tail (length - k - 1), set next to None, connect last to head.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 9.4 Python Solution

```python
def rotate_right(head, k):
    if not head or not head.next or k == 0:
        return head

    # Find length and last node
    length = 1
    last = head
    while last.next:
        last = last.next
        length += 1

    k = k % length
    if k == 0:
        return head

    # Find new tail
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next

    new_head = new_tail.next
    new_tail.next = None
    last.next = head
    return new_head
```

## 10 Reorder List

### 10.1 Problem Statement

Given a linked list L0->L1->...->Ln-1->Ln, reorder it to L0->Ln->L1->Ln-1->...

### 10.2 Dry Run on Test Cases

- **Test Case 1**: head = 1->2->3->4 $\rightarrow$ Output: 1->4->2->3
- **Test Case 2**: head = 1->2->3->4->5 $\rightarrow$ Output: 1->5->2->4->3
- **Test Case 3**: head = 1 $\rightarrow$ Output: 1
- **Test Case 4**: head = None $\rightarrow$ Output: None

### 10.3 Algorithm

1. Find middle using slow and fast pointers.
2. Reverse second half.
3. Merge first and second halves alternately.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 10.4 Python Solution

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

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

    # Reverse second half
    second = slow.next
    slow.next = None
    prev = None
    while second:
        next_node = second.next
        second.next = prev
        prev = second
        second = next_node

    # Merge
    first = head
    while prev:
        next_first = first.next
        next_prev = prev.next
        first.next = prev
        prev.next = next_first
        first = next_first
        prev = next_prev
```

## 11 Valid Number

### 11.1 Problem Statement

Given a string, determine if it is a valid number (integer, decimal, or scientific notation).

### 11.2 Dry Run on Test Cases

- **Test Case 1**: Input = "0" → Output: True

- **Test Case 2**: Input = "e" → Output: False
- **Test Case 3**: Input = "2e10" → Output: True
- **Test Case 4**: Input = "-0.1" → Output: True

## 11.3 Algorithm

1. Use regex or manual parsing to check:
   - Optional sign, digits, optional decimal, optional 'e' followed by integer.
2. Ensure proper format (e.g., no multiple decimals).

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

## 11.4 Python Solution

```python
def is_number(s):
    s = s.strip()
    if not s:
        return False

    # Split on 'e' or 'E'
    parts = s.lower().split('e')
    if len(parts) > 2:
        return False

    # Validate base part
    base = parts[0]
    if not base or base == '+' or base == '-':
        return False

    decimal_count = 0
    digit_seen = False
    for i, char in enumerate(base):
        if char == '.':
            decimal_count += 1
            if decimal_count > 1:
                return False
        elif char.isdigit():
            digit_seen = True
        elif char not in ['+', '-'] or i != 0:
            return False

    if not digit_seen:
        return False

    # Validate exponent if present
    if len(parts) == 2:
        exponent = parts[1]
        if not exponent or exponent == '+' or exponent == '-':
            return False
        digit_seen = False
        for i, char in enumerate(exponent):
            if char.isdigit():
                digit_seen = True
            elif char not in ['+', '-'] or i != 0:
                return False
        if not digit_seen:
            return False

    return True

# Example usage
print(is_number("2e10"))  # Output: True
```

# 12 Min Stack

## 12.1 Problem Statement

Design a stack that supports push, pop, top, and retrieving the minimum element in O(1) time.

## 12.2 Dry Run on Test Cases

- **Test Case 1**: push(3), push(5), getMin() $\rightarrow$ 3, push(2), getMin() $\rightarrow$ 2, pop(), getMin() $\rightarrow$ 3
- **Test Case 2**: push(1), pop(), top() $\rightarrow$ None
- **Test Case 3**: push(2), push(1), getMin() $\rightarrow$ 1
- **Test Case 4**: empty stack, getMin() $\rightarrow$ None

## 12.3 Algorithm

1. Use two stacks: one for values, one for minimums.
2. Push: append value, update min stack if needed.
3. Pop: remove from both stacks if popped value was min.
4. Top/GetMin: return top of respective stacks.

**Time Complexity**: $O(1)$ for all operations **Space Complexity**: $O(n)$

## 12.4 Python Solution

```python
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        self.stack.append(val)
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self):
        if not self.stack:
            return
        val = self.stack.pop()
        if val == self.min_stack[-1]:
            self.min_stack.pop()

    def top(self):
        return self.stack[-1] if self.stack else None

    def getMin(self):
        return self.min_stack[-1] if self.min_stack else None

# Example usage
# minStack = MinStack()
# minStack.push(3)
# minStack.push(5)
# print(minStack.getMin())  # Output: 3
```

## 13 Evaluate Reverse Polish Notation

### 13.1 Problem Statement

Given an array of strings representing an RPN expression, evaluate it.

### 13.2 Dry Run on Test Cases

- **Test Case 1**: tokens = ["2", "1", "+", "3", "*"] → Output: 9 ((2 + 1) * 3)
- **Test Case 2**: tokens = ["4", "13", "5", "/", "+"] → Output: 6 (4 + 13/5)
- **Test Case 3**: tokens = ["10"] → Output: 10
- **Test Case 4**: tokens = ["10", "6", "/"] → Output: 1

### 13.3 Algorithm

1. Use a stack to store operands.
2. For each token:
    - If number, push to stack.

    - If operator, pop two operands, compute, push result.

3. Return stack top.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$

### 13.4 Python Solution

```python
def eval_rpn(tokens):
    stack = []
    operators = {
        '+': lambda x, y: x + y,
        '-': lambda x, y: x - y,
        '*': lambda x, y: x * y,
        '/': lambda x, y: int(x / y)
    }

    for token in tokens:
        if token in operators:
            b = stack.pop()
            a = stack.pop()
            stack.append(operators[token](a, b))
        else:
            stack.append(int(token))
    return stack[0]

# Example usage
print(eval_rpn(["2", "1", "+", "3", "*"]))  # Output: 9
```

## 14 Valid Parentheses with Wildcard

### 14.1 Problem Statement

Given a string with '(', ')', and '*', where '*' can be '(', ')', or empty, check if valid.

### 14.2 Dry Run on Test Cases

- **Test Case 1**: s = "()" → Output: True
- **Test Case 2**: s = "(*)" → Output: True
- **Test Case 3**: s = "(*))" → Output: True
- **Test Case 4**: s = "((*)" → Output: True

### 14.3 Algorithm

1. Track min and max open brackets (min can be 0, max can increase with *).
2. For each char:

  - '(': min++, max++
  - ')': min–, max–
  - '*': min–, max++

3. Ensure min >= 0, reset min to 0 if negative.
4. Return True if min == 0 at end.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(1)$

### 14.4 Python Solution

```python
def check_valid_string(s):
    min_open = max_open = 0

    for char in s:
        if char == '(':
            min_open += 1
            max_open += 1
        elif char == ')':
            min_open -= 1
            max_open -= 1
        else:  # '*'
            min_open -= 1
            max_open += 1
        if max_open < 0:
            return False
        if min_open < 0:
            min_open = 0
    return min_open == 0

# Example usage
print(check_valid_string("(*)"))  # Output: True
```

## 15 Next Greater Element

### 15.1 Problem Statement

Given two arrays nums1 and nums2, for each element in nums1, find the next greater element in nums2.

### 15.2 Dry Run on Test Cases

- **Test Case 1**: nums1 = [4,1,2], nums2 = [1,3,4,2] → Output: [-1,3,-1]
- **Test Case 2**: nums1 = [2,4], nums2 = [1,2,3,4] → Output: [3,-1]
- **Test Case 3**: nums1 = [1], nums2 = [1] → Output: [-1]

- **Test Case 4**: nums1 = [], nums2 = [1,2] → Output: []

### 15.3 Algorithm

1. Use stack to find next greater for each element in nums2.
2. Store results in hashmap.
3. Map nums1 elements to their next greater.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$

### 15.4 Python Solution

```python
def next_greater_element(nums1, nums2):
    stack = []
    next_greater = {}

    for num in nums2:
        while stack and stack[-1] < num:
            next_greater[stack.pop()] = num
        stack.append(num)

    while stack:
        next_greater[stack.pop()] = -1

    return [next_greater[num] for num in nums1]

# Example usage
print(next_greater_element([4,1,2], [1,3,4,2]))  # Output:
    [-1,3,-1]
```

## 16 Daily Temperatures

### 16.1 Problem Statement

Given an array of temperatures, return an array where each element is the number of days until a warmer day.

### 16.2 Dry Run on Test Cases

- **Test Case 1**: temperatures = [73,74,75,71,69,72,76,73] → Output: [1,1,4,2,1,1,0,0]
- **Test Case 2**: temperatures = [30,40,50,60] → Output: [1,1,1,0]
- **Test Case 3**: temperatures = [30] → Output: [0]
- **Test Case 4**: temperatures = [30,20,10] → Output: [0,0,0]

### 16.3 Algorithm

1. Use stack to store indices of temperatures.
2. For each temperature, pop stack while current > stack top, calculate days.
3. Push current index to stack.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$

### 16.4 Python Solution

```python
def daily_temperatures(temperatures):
    n = len(temperatures)
    result = [0] * n
    stack = []

    for i in range(n):
        while stack and temperatures[i] > temperatures[stack[-1]]:
            prev = stack.pop()
            result[prev] = i - prev
        stack.append(i)
    return result

# Example usage
print(daily_temperatures([73,74,75,71,69,72,76,73]))  # Output: [1,1,4,2,1,1,0,0]
```

## 17 Implement Stack Using Queues

### 17.1 Problem Statement

Implement a stack using two queues with push, pop, top, and empty operations.

### 17.2 Dry Run on Test Cases

- **Test Case 1**: push(1), push(2), top() $\to$ 2, pop() $\to$ 2, empty() $\to$ False
- **Test Case 2**: push(1), pop(), empty() $\to$ True
- **Test Case 3**: empty() $\to$ True
- **Test Case 4**: push(1), top() $\to$ 1

### 17.3 Algorithm

1. Use two queues; main queue holds stack elements.
2. Push: add to main queue.

3. Pop/Top: move all but last element to second queue, process last, swap queues.

**Time Complexity**: $O(n)$ for pop/top, $O(1)$ for push/empty **Space Complexity**: $O(n)$

### 17.4 Python Solution

```python
from collections import deque

class MyStack:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        self.q1.append(x)

    def pop(self):
        if not self.q1:
            return None
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        result = self.q1.popleft()
        self.q1, self.q2 = self.q2, self.q1
        return result

    def top(self):
        if not self.q1:
            return None
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        result = self.q1[0]
        self.q2.append(self.q1.popleft())
        self.q1, self.q2 = self.q2, self.q1
        return result

    def empty(self):
        return len(self.q1) == 0
```

## 18 Implement Queue Using Stacks

### 18.1 Problem Statement

Implement a queue using two stacks with enqueue, dequeue, peek, and empty operations.

### 18.2 Dry Run on Test Cases

- **Test Case 1**: enqueue(1), enqueue(2), peek() $\rightarrow$ 1, dequeue() $\rightarrow$ 1
- **Test Case 2**: enqueue(1), dequeue(), empty() $\rightarrow$ True

- **Test Case 3**: empty() → True
- **Test Case 4**: enqueue(1), peek() → 1

### 18.3 Algorithm

1. Use two stacks: push$_stackandpop_stack.Enqueue : pushtopush_stack$.

2. Dequeue/Peek: move push$_stacktopop_stackifempty, pop/peekfrompop_stack$. **Time Complexity**: $O(1)$ amortized for all operations **Space Complexity**: $O(n)$

### 18.4 Python Solution

```python
class MyQueue:
    def __init__(self):
        self.push_stack = []
        self.pop_stack = []

    def push(self, x):
        self.push_stack.append(x)

    def pop(self):
        if not self.pop_stack:
            while self.push_stack:
                self.pop_stack.append(self.push_stack.pop())
        return self.pop_stack.pop() if self.pop_stack else None

    def peek(self):
        if not self.pop_stack:
            while self.push_stack:
                self.pop_stack.append(self.push_stack.pop())
        return self.pop_stack[-1] if self.pop_stack else None

    def empty(self):
        return not self.push_stack and not self.pop_stack
```

# 19 Design Circular Queue

## 19.1 Problem Statement

Design a circular queue with enqueue, dequeue, front, rear, isEmpty, and isFull operations.

## 19.2 Dry Run on Test Cases

- **Test Case 1**: k=3, enQueue(1), enQueue(2), enQueue(3), isFull() → True, deQueue() → 1
- **Test Case 2**: k=1, enQueue(1), deQueue(), isEmpty() → True

– **Test Case 3**: k=2, enQueue(1), Front() → 1, Rear() → 1

– **Test Case 4**: k=1, isEmpty() → True

## 19.3 Algorithm

1. Use array of size k with front and rear pointers.
2. Enqueue: if not full, add at rear, increment rear % k.
3. Dequeue: if not empty, increment front % k.
4. Track size for empty/full checks.

**Time Complexity**: $O(1)$ for all operations **Space Complexity**: $O(k)$

## 19.4 Python Solution

```python
class MyCircularQueue:
    def __init__(self, k):
        self.size = k
        self.queue = [None] * k
        self.front = -1  # Index of front element
        self.rear = -1   # Index of last element
        self.count = 0   # Number of elements

    def enQueue(self, value):
        if self.isFull():
            return False
        if self.isEmpty():
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = value
        self.count += 1
        return True

    def deQueue(self):
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.size
        self.count -= 1
        if self.isEmpty():
            self.front = -1
            self.rear = -1
        return True

    def Front(self):
        return self.queue[self.front] if not self.isEmpty
           () else -1

    def Rear(self):
        return self.queue[self.rear] if not self.isEmpty
           () else -1

    def isEmpty(self):
        return self.count == 0

    def isFull(self):
        return self.count == self.size
```

## 20 Sliding Window Maximum

### 20.1 Problem Statement

Given an array and window size k, find the maximum element in each sliding window.

### 20.2 Dry Run on Test Cases

- **Test Case 1**: nums = [1,3,-1,-3,5,3,6,7], k = 3 → Output: [3,3,5,5,6,7]
- **Test Case 2**: nums = [1], k = 1 → Output: [1]
- **Test Case 3**: nums = [1,-1], k = 1 → Output: [1,-1]
- **Test Case 4**: nums = [], k = 1 → Output: []

### 20.3 Algorithm

1. Use deque to store indices of potential max elements.
2. For each element:
   - Remove indices outside window.
   - Remove smaller elements from back.
   - Add current index.
3. After k elements, append max (front of deque) for each window.

**Time Complexity**: $O(n)$   **Space Complexity**: $O(k)$

### 20.4 Python Solution

```
from collections import deque

def max_sliding_window(nums, k):
    if not nums or k == 0:
        return []

```

```python
    result = []
    deq = deque()

    for i in range(len(nums)):
        # Remove indices outside window
        while deq and deq[0] <= i - k:
            deq.popleft()
        # Remove smaller elements
        while deq and nums[deq[-1]] < nums[i]:
            deq.pop()
        deq.append(i)
        # Add max for window
        if i >= k - 1:
            result.append(nums[deq[0]])
    return result

# Example usage
print(max_sliding_window([1,3,-1,-3,5,3,6,7], 3))  #
    Output: [3,3,5,5,6,7]
```

## 21 Largest Rectangle in Histogram

### 21.1 Problem Statement

Given an array of bar heights, find the largest rectangle area in the histogram.

### 21.2 Dry Run on Test Cases

- **Test Case 1**: heights = [2,1,5,6,2,3] → Output: 10 (height 5, width 2)
- **Test Case 2**: heights = [2,4] → Output: 4
- **Test Case 3**: heights = [1] → Output: 1
- **Test Case 4**: heights = [] → Output: 0

### 21.3 Algorithm

1. Use stack to store indices of increasing heights.
2. For each bar, pop stack while current height $<$ stack top height.
3. Calculate area for each popped bar: height * (current index - previous index - 1).
4. Handle remaining bars after loop.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$

### 21.4 Python Solution

```python
def largest_rectangle_area(heights):
    stack = [-1]
    max_area = 0
    heights.append(0)  # Sentinel to process remaining bars

    for i in range(len(heights)):
        while stack[-1] != -1 and heights[i] < heights[stack[-1]]:
            h = heights[stack.pop()]
            w = i - stack[-1] - 1
            max_area = max(max_area, h * w)
        stack.append(i)

    heights.pop()  # Remove sentinel
    return max_area

# Example usage
print(largest_rectangle_area([2,1,5,6,2,3]))  # Output: 10
```

# 22 Minimum Remove to Make Valid Parentheses

## 22.1 Problem Statement

Given a string with '(', ')', and letters, remove minimum characters to make it valid.

## 22.2 Dry Run on Test Cases

- **Test Case 1**: s = "lee(t(c)o)de)" → Output: "lee(t(c)o)de"
- **Test Case 2**: s = "a)b(c)d" → Output: "ab(c)d"
- **Test Case 3**: s = "))((" → Output: ""
- **Test Case 4**: s = "(a(b(c)d)" → Output: "a(b(c)d)"

## 22.3 Algorithm

1. Use stack to track indices of open parentheses.
2. Mark invalid parentheses for removal.
3. Build result string, skipping marked indices.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(n)$

### 22.4 Python Solution

```python
def min_remove_to_make_valid(s):
    s = list(s)
    stack = []

    # Mark invalid parentheses
    for i, char in enumerate(s):
        if char == '(':
            stack.append(i)
        elif char == ')':
            if stack:
                stack.pop()
            else:
                s[i] = ''

    # Mark unmatched open parentheses
    while stack:
        s[stack.pop()] = ''

    return ''.join(s)

# Example usage
print(min_remove_to_make_valid("lee(t(c)o)de)"))  #
    Output: "lee(t(c)o)de"
```

## 23 Binary Tree Inorder Traversal

### 23.1 Problem Statement

Given a binary tree, return its inorder traversal (left, root, right).

### 23.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [1,3,2]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [2,1,3]

### 23.3 Algorithm

1. Use iterative approach with stack.
2. Push all left nodes to stack.
3. Pop node, add to result, process right subtree.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(h)$ (h = tree height)

### 23.4 Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None
        ):
        self.val = val
        self.left = left
        self.right = right

def inorder_traversal(root):
    result = []
    stack = []
    curr = root

    while curr or stack:
        while curr:
            stack.append(curr)
            curr = curr.left
        curr = stack.pop()
        result.append(curr.val)
        curr = curr.right
    return result
```

## 24 Binary Tree Preorder Traversal

### 24.1 Problem Statement

Given a binary tree, return its preorder traversal (root, left, right).

### 24.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [1,2,3]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [1,2,3]

### 24.3 Algorithm

1. Use iterative approach with stack.
2. Push root, pop and process node, push right then left (stack reverses order).

**Time Complexity**: $O(n)$ **Space Complexity**: $O(h)$

### 24.4 Python Solution

```python
def preorder_traversal(root):
    if not root:
        return []

    result = []
    stack = [root]

    while stack:
        node = stack.pop()
        result.append(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return result
```

# 25 Binary Tree Postorder Traversal

### 25.1 Problem Statement

Given a binary tree, return its postorder traversal (left, right, root).

### 25.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,null,2,3] → Output: [3,2,1]
- **Test Case 2**: root = [] → Output: []
- **Test Case 3**: root = [1] → Output: [1]
- **Test Case 4**: root = [1,2,3] → Output: [2,3,1]

### 25.3 Algorithm

1. Use two stacks: first for preorder (root, left, right), second to reverse.
2. Pop from first stack, push to second, add children in reverse order.
3. Pop from second stack for result.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(h)$

### 25.4 Python Solution

```python
def postorder_traversal(root):
    if not root:
        return []

    result = []
    stack1 = [root]
    stack2 = []

    while stack1:
        node = stack1.pop()
        stack2.append(node)
        if node.left:
            stack1.append(node.left)
        if node.right:
            stack1.append(node.right)

    while stack2:
        result.append(stack2.pop().val)
    return result
```

# 26 Binary Tree Level Order Traversal

## 26.1 Problem Statement

Given a binary tree, return its level order traversal (level by level, left to right).

## 26.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: [[3],[9,20],[15,7]]
- **Test Case 2**: root = [1] → Output: [[1]]
- **Test Case 3**: root = [] → Output: []
- **Test Case 4**: root = [1,2,3] → Output: [[1],[2,3]]

## 26.3 Algorithm

1. Use queue to process nodes level by level.
2. For each level, process all nodes, add children to queue.
3. Collect nodes per level in result.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(w)$ (w = max width)

## 26.4 Python Solution

```
from collections import deque

def level_order(root):
```

```python
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(current_level)
    return result
```

# 27 Maximum Depth of Binary Tree

## 27.1 Problem Statement

Given a binary tree, find its maximum depth (number of nodes along longest path from root to leaf).

## 27.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] $\rightarrow$ Output: 3
- **Test Case 2**: root = [1,null,2] $\rightarrow$ Output: 2
- **Test Case 3**: root = [] $\rightarrow$ Output: 0
- **Test Case 4**: root = [1] $\rightarrow$ Output: 1

## 27.3 Algorithm

1. Use recursive DFS: max depth = 1 + max(left depth, right depth).
2. Base case: return 0 for None.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(h)$ for recursion

## 27.4 Python Solution (Recursive)

```python
def max_depth(root):
    if not root:
        return 0
    return 1 + max(max_depth(root.left), max_depth(
        root.right))

# Example usage
# root = TreeNode(3, TreeNode(9), TreeNode(20,
    TreeNode(15), TreeNode(7)))
# print(max_depth(root))  # Output: 3
```

### 27.5 Python Solution (Iterative)

```python
def max_depth_iterative(root):
    if not root:
        return 0

    queue = deque([(root, 1)])
    max_depth = 0

    while queue:
        node, depth = queue.popleft()
        max_depth = max(max_depth, depth)
        if node.left:
            queue.append((node.left, depth + 1))
        if node.right:
            queue.append((node.right, depth + 1))
    return max_depth
```

## 28 Same Tree

### 28.1 Problem Statement

Given two binary trees, check if they are structurally identical and have the same values.

### 28.2 Dry Run on Test Cases

- **Test Case 1**: p = [1,2,3], q = [1,2,3] → Output: True
- **Test Case 2**: p = [1,2], q = [1,null,2] → Output: False
- **Test Case 3**: p = [1,2,1], q = [1,1,2] → Output: False
- **Test Case 4**: p = [], q = [] → Output: True

### 28.3 Algorithm

1. Recursively check:
2. If both None, return True.

3. If one None or values differ, return False.
4. Recurse on left and right subtrees.

**Time Complexity**: $O(\min(n, m))$  **Space Complexity**: $O(h)$

### 28.4 Python Solution

```python
def is_same_tree(p, q):
    if not p and not q:
        return True
    if not p or not q or p.val != q.val:
        return False
    return is_same_tree(p.left, q.left) and
       is_same_tree(p.right, q.right)
```

# 29 Symmetric Tree

## 29.1 Problem Statement

Given a binary tree, check if it is mirror symmetric.

## 29.2 Dry Run on Test Cases

- **Test Case 1**: root = [1,2,2,3,4,4,3] → Output: True
- **Test Case 2**: root = [1,2,2,null,3,null,3] → Output: False
- **Test Case 3**: root = [1] → Output: True
- **Test Case 4**: root = [] → Output: True

## 29.3 Algorithm

1. Recursively check left and right subtrees:
2. If both None, return True.
3. If one None or values differ, return False.
4. Compare left.left with right.right and left.right with right.left.

**Time Complexity**: $O(n)$  **Space Complexity**: $O(h)$

## 29.4 Python Solution

```python
def is_symmetric(root):
    def is_mirror(left, right):
        if not left and not right:
            return True
        if not left or not right or left.val !=
            right.val:
            return False
        return is_mirror(left.left, right.right)
            and is_mirror(left.right, right.left)

    return is_mirror(root, root) if root else True
```

# 30 Balanced Binary Tree

## 30.1 Problem Statement

Given a binary tree, determine if it is height-balanced (difference in heights of left and right subtrees <= 1).

## 30.2 Dry Run on Test Cases

- **Test Case 1**: root = [3,9,20,null,null,15,7] → Output: True
- **Test Case 2**: root = [1,2,2,3,3,null,null,4,4] → Output: False
- **Test Case 3**: root = [] → Output: True
- **Test Case 4**: root = [1] → Output: True

## 30.3 Algorithm

1. Use DFS to compute height of each subtree.
2. Return -1 if unbalanced, else height.
3. Tree is balanced if root's height != -1.

**Time Complexity**: $O(n)$ **Space Complexity**: $O(h)$

## 30.4 Python Solution

```
def is_balanced(root):
    def check_height(node):
        if not node:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
        right_height = check_height(node.right)
        if right_height == -1 or abs(left_height -
            right_height) > 1:
            return -1
        return max(left_height, right_height) + 1

    return check_height(root) != -1
```