PROJECT BASED ASSIGNMENT REPORT

DATA STRUCTURES

Submitted By

Sr#	Student Name	Student Roll No	Section
1	Abdullah Imran	398	9 (I)
2	Laiba Butt	266	
3 (If any)			

Instructor:
Mr. Humayun Majeed _____ **(**Signature**)**

Department of Computer Science
Lahore Garrison University
Lahore

PROJECT EVALUATION SHEET

(For Instructor Use Only)

Rubrics	[Student-1]	[Student-2]	[Student-3]
	[Roll No]	[Roll No]	[Roll No]
PBA Implementation (6)	**(3)**	**(3)**	**(3)**
1. Completeness & Accuracy (3)			
2. Dry Run & Output Verification (3)			
Coding Standards (6)	**(3)**	**(3)**	**(3)**
3. Code Quality (3)			
4. Logic & Efficiency (3)			
Project Report (6)	**(3)**	**(3)**	**(3)**
5. Structure and Explanation (3)			
6. Results and Discussion (3)			
Project Viva (6)	**(3)**	**(3)**	**(3)**
7. Responsiveness to Questions (3)			
8. Understanding of Concepts (3)			
Project Presentation (6)	**(3)**	**(3)**	**(3)**
9. Confidence (3)			
10. Responsiveness to Audience (3)			
Teamwork (6)	**(3)**	**(3)**	**(3)**
11. Sharing Information (3)			
12. Fulfilling Team Duties (3)			
TOTAL MARKS: 36			

Department of Computer Science

Lahore Garrison University

Project Report

1. INTRODUCTION

This project-based assignment focuses on implementing fundamental data structures and algorithms in C++. The project encompasses a comprehensive exploration of Arrays, Linked Lists, Stacks, Queues, AVL Trees, and Graph data structures. Through this assignment, we aimed to develop a deep understanding of how these data structures operate, their complexities, and their real-world applications.

The primary objective was to implement 28 distinct problems covering various aspects of data structure manipulation, from basic operations like insertion and deletion to advanced algorithms such as Floyd's Cycle Detection, AVL Tree balancing, and Graph traversals. Each implementation was designed to solve practical problems while maintaining optimal time and space complexity.

This project serves as a practical bridge between theoretical knowledge and hands-on implementation, enhancing our problem-solving capabilities and strengthening our foundation in data structures and algorithms.

---

2. ASSIGNMENT OVERVIEW

The assignment was divided into five major parts, each targeting specific data structures and algorithmic concepts:

Part 1: Array & Linked List (Questions 1-10)

This section focused on array manipulation techniques including rotation, rearrangement, and linked list operations such as palindrome detection, merging sorted lists, cycle detection using Floyd's algorithm, and implementing priority queues and deques.

Part 2: Stack & Queue (Questions 11-15)

We implemented advanced stack and queue operations including recursive stack sorting, balanced bracket checking with multiple bracket types, implementing stacks using queues, queue sorting using stacks, and a Min-Stack with constant-time minimum retrieval.

Part 3: Cracking the Code (Questions 16-20)

This section contained classic interview-style problems including removing duplicates from unsorted lists, finding the Nth element from the end, deleting middle nodes, adding numbers represented as linked lists, and detecting loop beginnings in circular lists.

Part 4: AVL Trees (Questions 21-23)

We implemented self-balancing AVL trees with insertion, deletion, and rebalancing operations, working with both integer and floating-point values while maintaining balance factors.

Part 5: Graphs (Questions 24-27)

The final section involved creating directed and undirected weighted graphs representing university buildings, student clubs, and shuttle routes, implementing adjacency lists and matrices, finding Minimum Spanning Trees, and performing BFS and DFS traversals.

Part 6: Main Menu (Question 28)

Integration of all implementations into a single menu-driven program for easy execution and testing.

The work was distributed among team members based on expertise, with regular code reviews and collaborative debugging sessions. We used C++ as our primary programming language with standard libraries for implementation.

---

3. IMPLEMENTATION SUMMARY

3.1. Array Implementation

Operations Implemented:

- Recursive array reversal
- Right rotation by k positions using three-reversal technique
- Rearranging negative numbers before positive numbers

Key Logic:

The rotation function uses a three-step reversal approach: reverse the entire array, reverse the first k elements, and reverse the remaining elements. This achieves O(n) time complexity with O(1) space complexity. The rearrangement function uses a two-pointer technique to partition negatives and positives efficiently.

Challenges:

Ensuring that the recursive reversal function handled edge cases correctly (empty arrays, single elements) and maintaining the relative order during rearrangement required careful pointer manipulation.

3.2. Linked List Implementation

Operations Implemented:

- List reversal (iterative)
- Palindrome detection using mid-split and reverse
- Merging two sorted lists
- Cycle detection and finding cycle start (Floyd's algorithm)
- Removing consecutive absent nodes
- Deleting a node given only its pointer
- Merge sort implementation
- Finding Nth node from end
- Removing duplicates using hash set
- Adding two numbers represented as lists

Key Logic:

Floyd's Cycle Detection algorithm uses slow and fast pointers to detect cycles in O(n) time. The palindrome detection splits the list at the middle, reverses the second half, compares both halves, and restores the original structure. Merge sort uses recursive splitting and merging for O(n log n) sorting.

Challenges:

Managing pointer integrity during list manipulation was critical. The palindrome function required careful restoration of the original list structure. Edge cases like empty lists, single-node lists, and tail node deletions required special handling.

3.3. Stack Implementation

Operations Implemented:

- Recursive stack sorting
- Balanced bracket checking with multiple bracket types
- Min-Stack with O(1) minimum retrieval

Key Logic:

The recursive sort uses a helper function to insert elements in sorted order. Each recursive call pops an element, sorts the remaining stack, then inserts the popped element at its correct position. The Min-Stack maintains the current minimum at each node, updating it during push operations.

Challenges:

Implementing purely recursive sorting without auxiliary data structures required careful design. The bracket checker needed to distinguish between different error types (extra closing, unclosed, type mismatch) and report accurate positions.

3.4. Queue Implementation

Operations Implemented:

- Priority queue insertion based on emergency levels
- Queue sorting using one stack and one auxiliary queue
- Stack implementation using two queues

Key Logic:

The priority queue maintains sorted order by traversing to find the correct insertion position. Queue sorting repeatedly finds the minimum element using a stack for temporary storage. The Stack-using-Queues implementation transfers all elements to the second queue before each push to maintain LIFO order.

Challenges:

Ensuring O(n) push and O(1) pop for the stack implementation required careful queue manipulation. The sorting algorithm needed to handle duplicate values correctly while maintaining stability.

3.5. Cracking the Code

Operations Implemented:

- Duplicate removal using unordered_set for O(n) time
- Finding Nth to last using two-pointer technique
- In-place middle node deletion by copying data
- Adding numbers in linked list format with carry propagation

Key Logic:

The two-pointer technique for finding Nth from end moves one pointer N steps ahead, then moves both pointers until the first reaches the end. The addition algorithm handles varying list lengths and carry propagation similar to manual addition.

Challenges:

The middle node deletion without head pointer access required copying the next node's data, which fails for tail nodes. Adding numbers required careful carry handling and creating new nodes for overflow.

3.6. AVL Tree Construction

Operations Implemented:

- Node insertion with automatic balancing
- Left and right rotations
- Double rotations (LR, RL)
- Balance factor calculation and maintenance
- Deletion with rebalancing

Key Logic:

After each insertion or deletion, balance factors are checked for all ancestors. If any node becomes unbalanced (balance factor > 1 or < -1), appropriate rotations restore balance. Left-Left case requires single right rotation, Left-Right requires left then right rotation, and mirror cases apply for right-heavy trees.

Challenges:

Tracking parent pointers for double linked list implementation added complexity. Handling edge cases during deletion, especially when the node to be deleted has two children, required careful successor/predecessor selection and pointer updates.

3.7. Graph Traversals and Operations

Operations Implemented:

- Adjacency list and matrix representations
- Breadth-First Search (BFS) using queues
- Depth-First Search (DFS) using recursion/stack
- Minimum Spanning Tree using Kruskal's or Prim's algorithm

Key Logic:

BFS uses a queue to explore nodes level by level, marking visited nodes to avoid cycles. DFS uses recursion or explicit stack for depth-first exploration. Kruskal's algorithm sorts edges by weight and uses Union-Find to detect cycles, adding edges that don't create cycles until MST is complete.

Challenges:

Implementing graph representations efficiently for both sparse and dense graphs. Ensuring visited node tracking prevented infinite loops. MST algorithms required careful edge sorting and cycle detection using disjoint set data structures.

3.8. Integrated Menu-Driven Program

Operations Implemented:

- Dynamic menu display with all 28 questions
- User input validation
- Execution of selected functionality
- Option to run multiple questions sequentially
- Clean exit functionality

Key Logic:

A do-while loop maintains the menu until the user exits. A switch statement routes to the appropriate function based on user choice. After each execution, the program prompts whether to run another question, providing seamless testing of all implementations.

Challenges:

Organizing 28 different implementations into a coherent structure. Ensuring clean memory management across all operations. Providing clear output formatting for easy verification of results.

---

4. DRY RUN DEMONSTRATIONS

Hand Written

---

5. CODE SNIPPETS AND STRUCTURE EXPLANATION

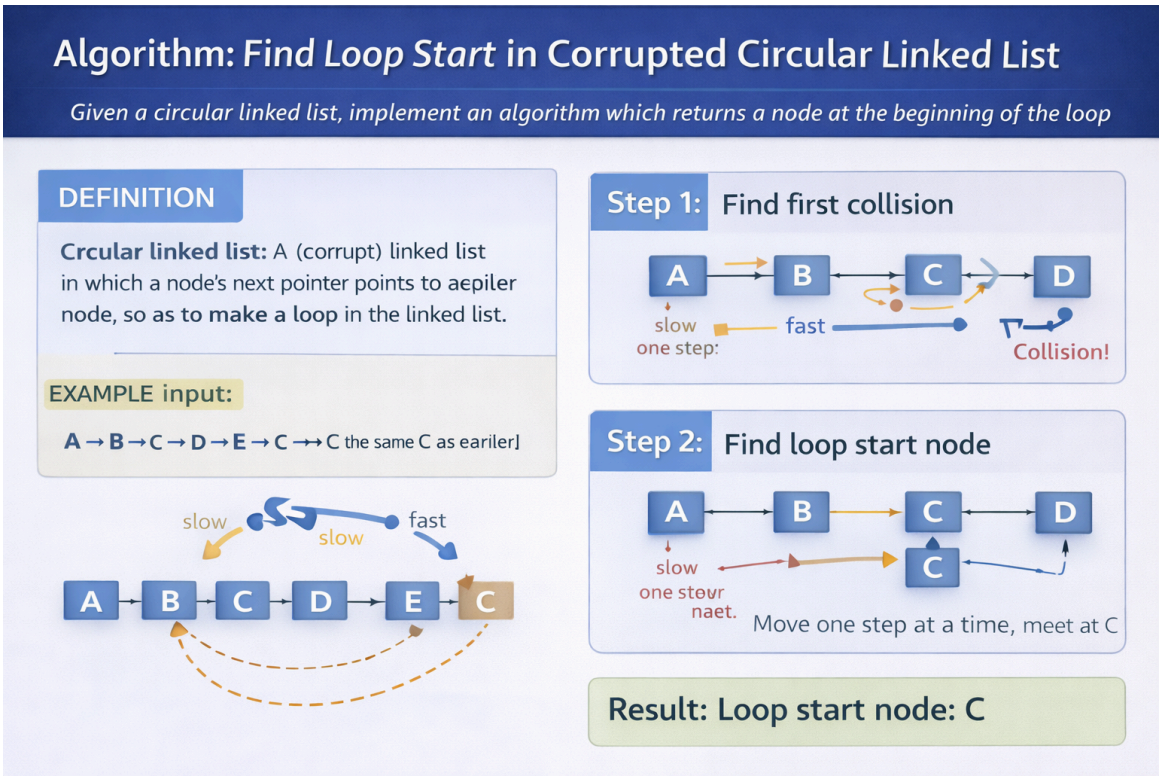
Array Rotation Function

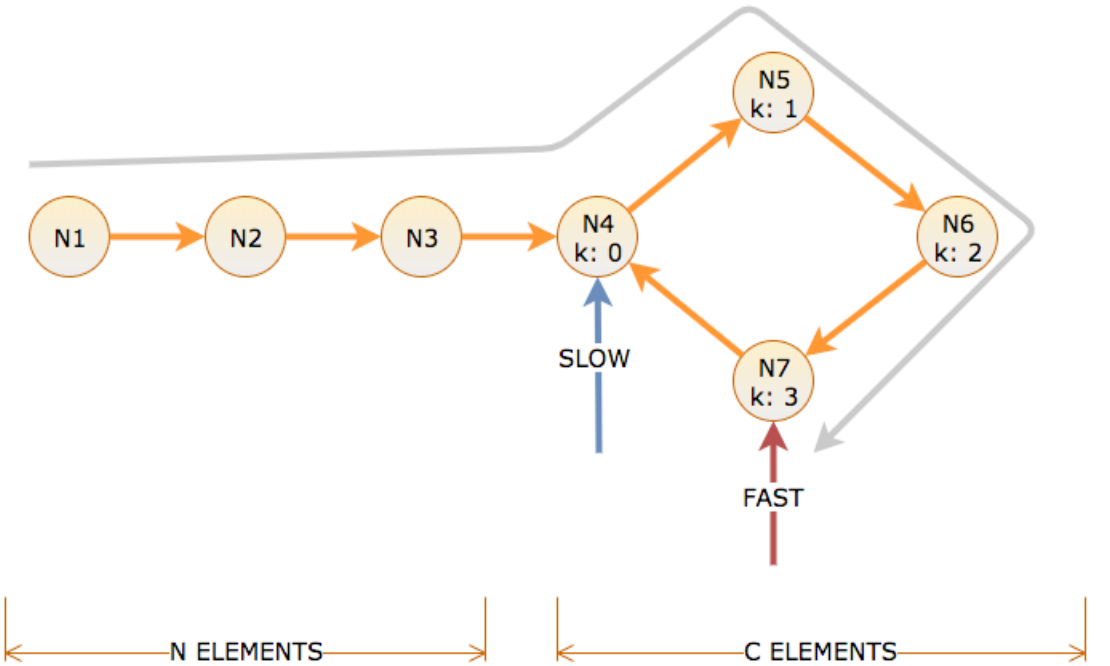
```
void rotateRight(int arr[], int n, int k) {
    k = k % n;
    if (k == 0) return;

    reverse(arr, 0, n - 1);
    reverse(arr, 0, k - 1);
```

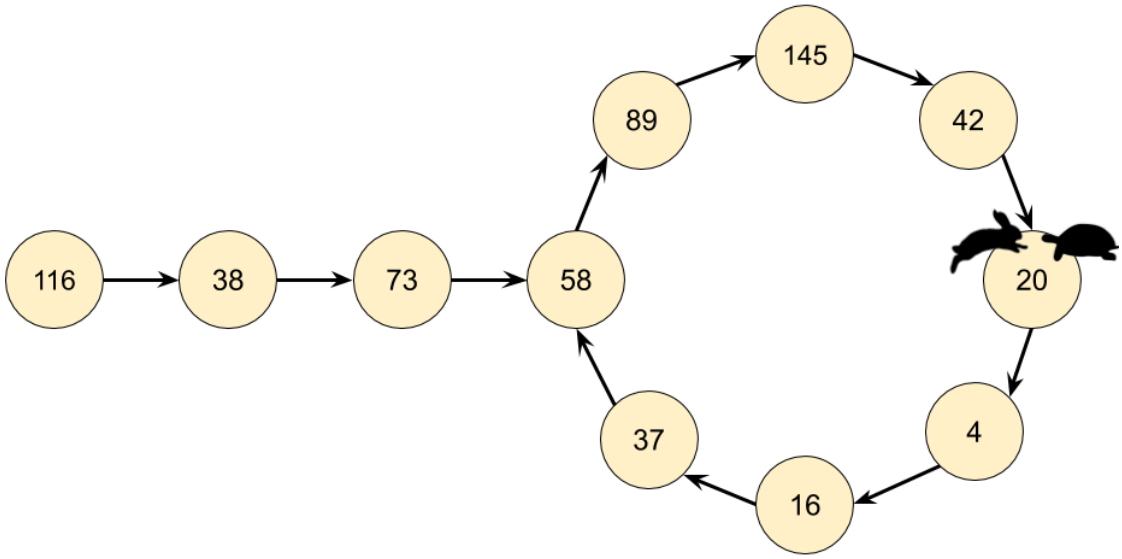
```
    reverse(arr, k, n - 1);
}
```

Explanation: This function rotates an array right by k positions using an efficient three-reversal technique. First, the entire array is reversed, then the first k elements, and finally the remaining elements. This achieves O(n) time complexity with O(1) space, avoiding the need for temporary arrays.

Floyd's Cycle Detection








```
Node* detectCycleStart(Node* head) {
    if (!head || !head->next) return nullptr;

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

```
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {

            slow = head;
            while (slow != fast) {
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return nullptr;
}
```

Explanation: This implements Floyd's Cycle Detection algorithm. The slow pointer moves one step while fast moves two steps. If they meet, a cycle exists. To find the cycle start, reset one pointer to head and move both one step at a time until they meet again—that's the cycle beginning.

Recursive Stack Sort

```
void sortStack(Stack& s) {
    if (s.isEmpty()) return;

    int temp = s.pop();
    sortStack(s);
    insertSorted(s, temp);
}

void insertSorted(Stack& s, int val) {
    if (s.isEmpty() || val >= s.peek()) {
        s.push(val);
        return;
    }

    int temp = s.pop();
    insertSorted(s, val);
    s.push(temp);
}
```

Explanation: This recursively sorts a stack without using auxiliary data structures. The base case is an empty stack. For each element, we pop it, recursively sort the remaining stack, then insert the element in its correct sorted position using another recursive function.

Palindrome Detection

```
bool isPalindrome(Node* head) {
    if (!head || !head->next) return true;

    Node* slow = head;
    Node* fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    Node* second = reverseList(slow->next);
    slow->next = nullptr;

    Node* first = head;
    bool isPal = true;
    while (first && second) {
        if (first->data != second->data) {
            isPal = false;
            break;
        }
        first = first->next;
        second = second->next;
    }

    slow->next = reverseList(second);
    return isPal;
}
```

Explanation: This detects palindromes in O(n) time with O(1) space. It finds the middle, reverses the second half, compares both halves element by element, then restores the original structure. This approach modifies and restores the list without using extra space.

Min-Stack Implementation

```
void push(int val) {
    int currentMin = top ? top->min : INT_MAX;
    int newMin = min(currentMin, val);
    MinNode* newNode = new MinNode(val, newMin);
    newNode->next = top;
    top = newNode;
}
```

```
int getMin() {
    return top ? top->min : -1;
}
```

Explanation: Each node stores both the value and the minimum value up to that point in the stack. When pushing, we compare the new value with the current minimum and store the smaller value. This allows O(1) minimum retrieval at any time without additional data structures.

Priority Queue Insertion

```
void InsertPriority(string name, int emergencyLevel) {
    Node* newNode = new Node{name, emergencyLevel, nullptr};

    if (!head) {
        head = tail = newNode;
        return;
    }

    if (emergencyLevel < head->emergencyLevel) {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* curr = head;
    while (curr->next && curr->next->emergencyLevel <= emergencyLevel) {
        curr = curr->next;
    }

    newNode->next = curr->next;
    curr->next = newNode;
}
```

Explanation: This maintains a sorted priority queue where lower emergency levels have higher priority. New patients are inserted before any patient with a numerically higher emergency level. The function handles edge cases for empty queues and insertion at the front.

Merge Two Sorted Lists

```
Node* mergeSortedLists(Node* list1, Node* list2) {
    if (!list1) return list2;
    if (!list2) return list1;

    Node* head;
```

```
    if (list1->data < list2->data) {
        head = list1;
        list1 = list1->next;
    } else {
        head = list2;
        list2 = list2->next;
    }

    Node* curr = head;
    while (list1 && list2) {
        if (list1->data < list2->data) {
            curr->next = list1;
            list1 = list1->next;
        } else {
            curr->next = list2;
            list2 = list2->next;
        }
        curr = curr->next;
    }

    if (list1) curr->next = list1;
    if (list2) curr->next = list2;

    return head;
}
```

Explanation: This merges two sorted linked lists by comparing heads and selecting the smaller value, advancing the corresponding pointer. It continues until one list is exhausted, then appends the remaining list. No new nodes are created—only pointers are relinked.

Queue Sorting Using Stack

```
void sortQueue(SimpleQueue& input, SimpleQueue& output) {
    SimpleStack S;
    SimpleQueue tempQ;

    while (!input.isEmpty()) {
        int minVal = INT_MAX;


        while (!input.isEmpty()) {
            int x = input.dequeue();
            S.push(x);
            if (x < minVal) minVal = x;
```

```
        }


        bool found = false;
        while (!S.isEmpty()) {
            int x = S.pop();
            if (x == minVal && !found) {
                output.enqueue(x);
                found = true;
            } else {
                tempQ.enqueue(x);
            }
        }


        while (!tempQ.isEmpty()) {
            input.enqueue(tempQ.dequeue());
        }
    }
}
```

Explanation: This function sorts a queue using only one stack and one auxiliary queue. It repeatedly finds the minimum element by transferring all elements to a stack (which allows finding min), extracts the minimum to the output queue, and puts remaining elements back to input for the next iteration. Time complexity is $O(n^2)$.

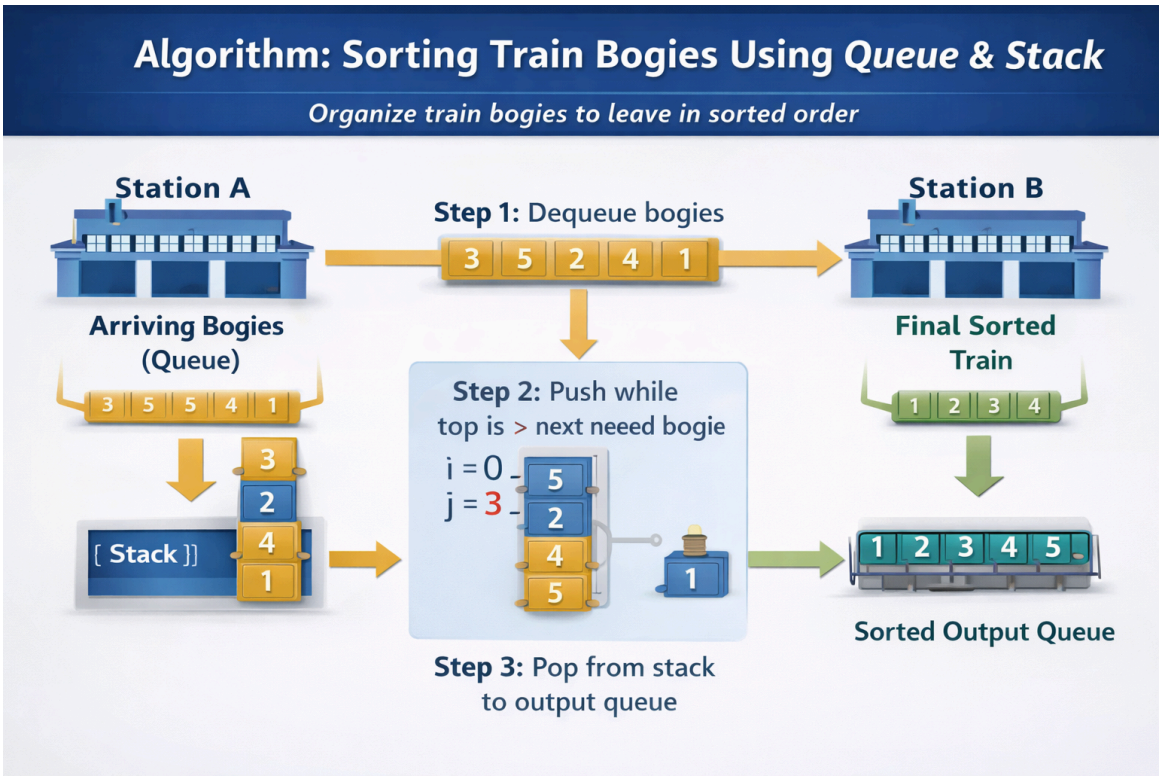
Stack Using Two Queues

```
class StackUsingQueues {
private:
    Queue q1, q2;

public:
    void push(int val) {

        while (!q1.isEmpty()) {
            q2.enqueue(q1.dequeue());
        }


        q1.enqueue(val);


        while (!q2.isEmpty()) {
```

```
            q1.enqueue(q2.dequeue());
        }
    }

    int pop() {
        if (q1.isEmpty()) return -1;
        return q1.dequeue();
    }

    int top() {
        if (q1.isEmpty()) return -1;
        return q1.front->data;
    }
};
```

Explanation: This implements a stack using two queues where push operation has O(n) complexity and pop has O(1). During each push, we transfer all existing elements to q2, add the new element to q1, then move everything back. This ensures the newest element is always at the front of q1, ready for O(1) pop.

Deque Implementation



```
class Deque {
private:
    struct Node {
        int data;
        Node* prev;
        Node* next;
    };
    Node* front;
    Node* rear;

public:
    void InsertFront(int val) {
        Node* newNode = new Node{val, nullptr, nullptr};
        if (!front) {
            front = rear = newNode;
        } else {
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
        }
    }

    void InsertRear(int val) {
        Node* newNode = new Node{val, nullptr, nullptr};
        if (!rear) {
            front = rear = newNode;
        } else {
            newNode->prev = rear;
            rear->next = newNode;
            rear = newNode;
        }
    }

    int RemoveFront() {
        if (!front) return -1;
        int val = front->data;
        Node* temp = front;
        front = front->next;
        if (front) {
            front->prev = nullptr;
        } else {
            rear = nullptr;
        }
        delete temp;
```

```
        return val;
    }

    int RemoveRear() {
        if (!rear) return -1;
        int val = rear->data;
        Node* temp = rear;
        rear = rear->prev;
        if (rear) {
            rear->next = nullptr;
        } else {
            front = nullptr;
        }
        delete temp;
        return val;
    }
};
```

Explanation: A double-ended queue (deque) implemented using a doubly-linked list allows O(1) insertion and deletion from both ends. Each node has prev and next pointers. InsertFront and InsertRear update the respective end pointers and maintain bidirectional links. RemoveFront and RemoveRear handle empty deque cases and update pointers accordingly.

Balanced Bracket Checker

```
string checkBrackets(string s) {
    stack<char> st;

    for (size_t i = 0; i < s.size(); i++) {
        char c = s[i];


        if (c == '(' || c == '{' || c == '[' || c == '<') {
            st.push(c);
        }

        else if (c == ')' || c == ']' || c == '}' || c == '>') {
            if (st.empty()) {
                return "Extra closing bracket '" + string(1, c) +
                       "' at position " + to_string(i + 1);
            }

            char top = st.top();
            st.pop();
```

```
            char expected;
            if (c == ')') expected = '(';
            else if (c == ']') expected = '[';
            else if (c == '}') expected = '{';
            else if (c == '>') expected = '<';


            if (top != expected) {
                return "Wrong-type mismatch: expected '" + string(1, expected) +
                       "' but found '" + string(1, c) + "' at position " +
                       to_string(i + 1);
            }
        }
    }


    if (!st.empty()) {
        string unclosed;
        while (!st.empty()) {
            unclosed = st.top() + unclosed;
            st.pop();
        }
        return "Unclosed brackets: " + unclosed;
    }

    return "Balanced";
}
```

Explanation: This extended bracket checker supports four types of brackets: (), {}, [], <>. It uses a stack to track opening brackets and validates each closing bracket against the stack top. The function detects three error types: extra closing brackets (stack empty when closing bracket found), unclosed brackets (stack not empty at end), and type mismatches (wrong closing bracket type). Position tracking helps identify exact error locations.

Remove Consecutive Absents

```
Node* RemoveConsecutiveAbsents(Node* head, int k) {
    if (!head || k <= 0) return head;

    Node* dummy = new Node(-1);
    dummy->next = head;
    Node* prev = dummy;
```

```
    while (prev->next) {
        Node* curr = prev->next;

        if (curr->data == 0) {

            int count = 0;
            Node* temp = curr;
            while (temp && temp->data == 0) {
                count++;
                temp = temp->next;
            }


            if (count >= k) {
                prev->next = temp;
                Node* toDel = curr;
                while (toDel != temp) {
                    Node* next = toDel->next;
                    delete toDel;
                    toDel = next;
                }
            } else {

                for (int i = 0; i < count; i++) {
                    prev = prev->next;
                }
            }
        } else {
            prev = prev->next;
        }
    }

    head = dummy->next;
    delete dummy;
    return head;
}
```

Explanation: This function removes sequences of k or more consecutive zeros (absents) from a linked list representing student attendance. A dummy node simplifies head deletion cases. The function scans for zero sequences, counts consecutive zeros, and removes the entire sequence if count >= k. Otherwise, it skips the sequence. This maintains O(n) time complexity with single-pass scanning.

Merge Sort for Linked List

```
Node* mergeSort(Node* head) {
    if (!head || !head->next) return head;


    bool sorted = true;
    Node* curr = head;
    while (curr->next) {
        if (curr->data > curr->next->data) {
            sorted = false;
            break;
        }
        curr = curr->next;
    }
    if (sorted) return head;


    Node* mid = findMid(head);
    Node* left = head;
    Node* right = mid->next;
    mid->next = nullptr;


    left = mergeSort(left);
    right = mergeSort(right);


    return mergeSortedLists(left, right);
}

Node* findMid(Node* head) {
    if (!head) return nullptr;
    Node* slow = head;
    Node* fast = head->next;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

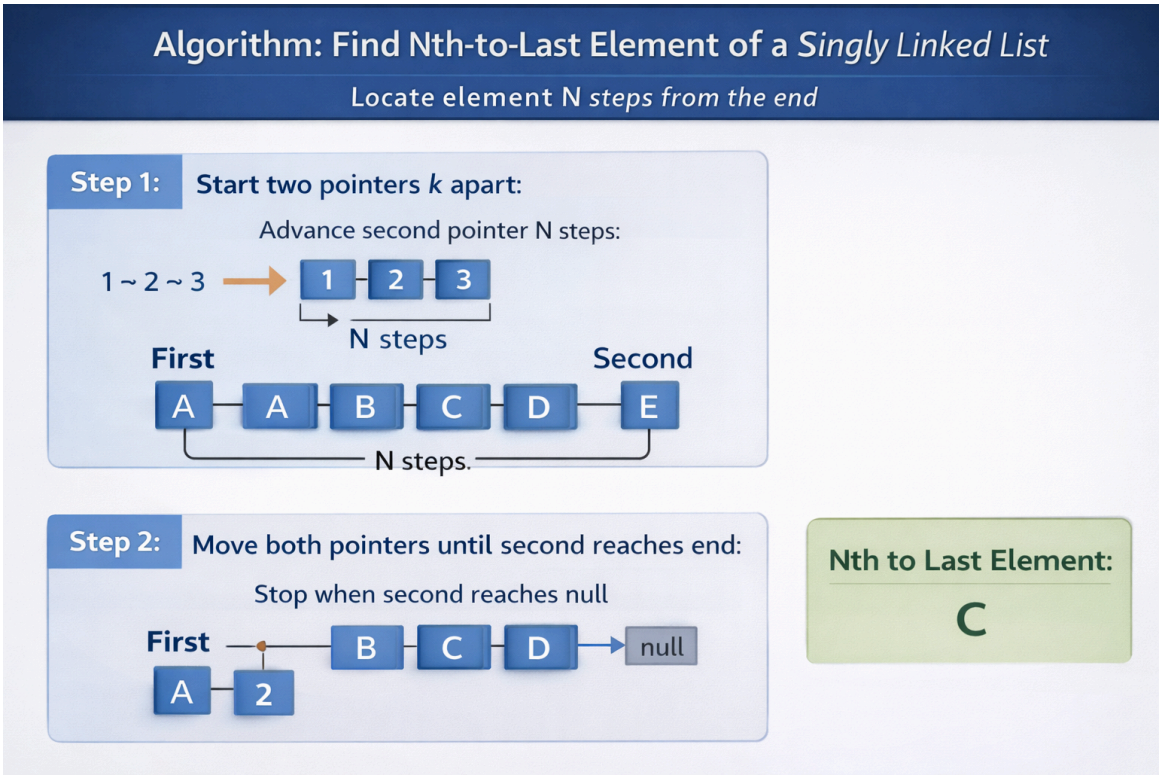
Explanation: This implements merge sort on a linked list with an optimization to detect pre-sorted lists. It uses the slow-fast pointer technique to find the middle, splits the list, recursively sorts both halves, and merges them. The optimization skips sorting if the list is already in order, improving best-case performance from O(n log n) to O(n). Worst-case remains O(n log n).

Remove Duplicates (Unsorted List)

```
Node* removeDuplicates(Node* head) {
    if (!head) return head;

    unordered_set<int> seen;
    Node* curr = head;
    Node* prev = nullptr;

    while (curr) {
        if (seen.find(curr->data) != seen.end()) {

            prev->next = curr->next;
            delete curr;
            curr = prev->next;
        } else {

            seen.insert(curr->data);
            prev = curr;
            curr = curr->next;
        }
    }

    return head;
}
```

Explanation: This removes duplicates from an unsorted linked list in O(n) time using a hash set. As we traverse, we check if each value has been seen before. If yes, we delete the node and link the previous node to the next. If no, we add the value to the set and continue. Space complexity is O(n) for the hash set.

Find Nth Node From End




```
Node* findNthFromEnd(Node* head, int n) {
    if (!head || n <= 0) return nullptr;

    Node* fast = head;
    Node* slow = head;


    for (int i = 0; i < n; i++) {
        if (!fast) return nullptr;
        fast = fast->next;
    }


    while (fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

Explanation: This uses the two-pointer technique to find the Nth node from the end in a single pass. First, move the fast pointer n steps ahead. Then move both pointers together until fast reaches the

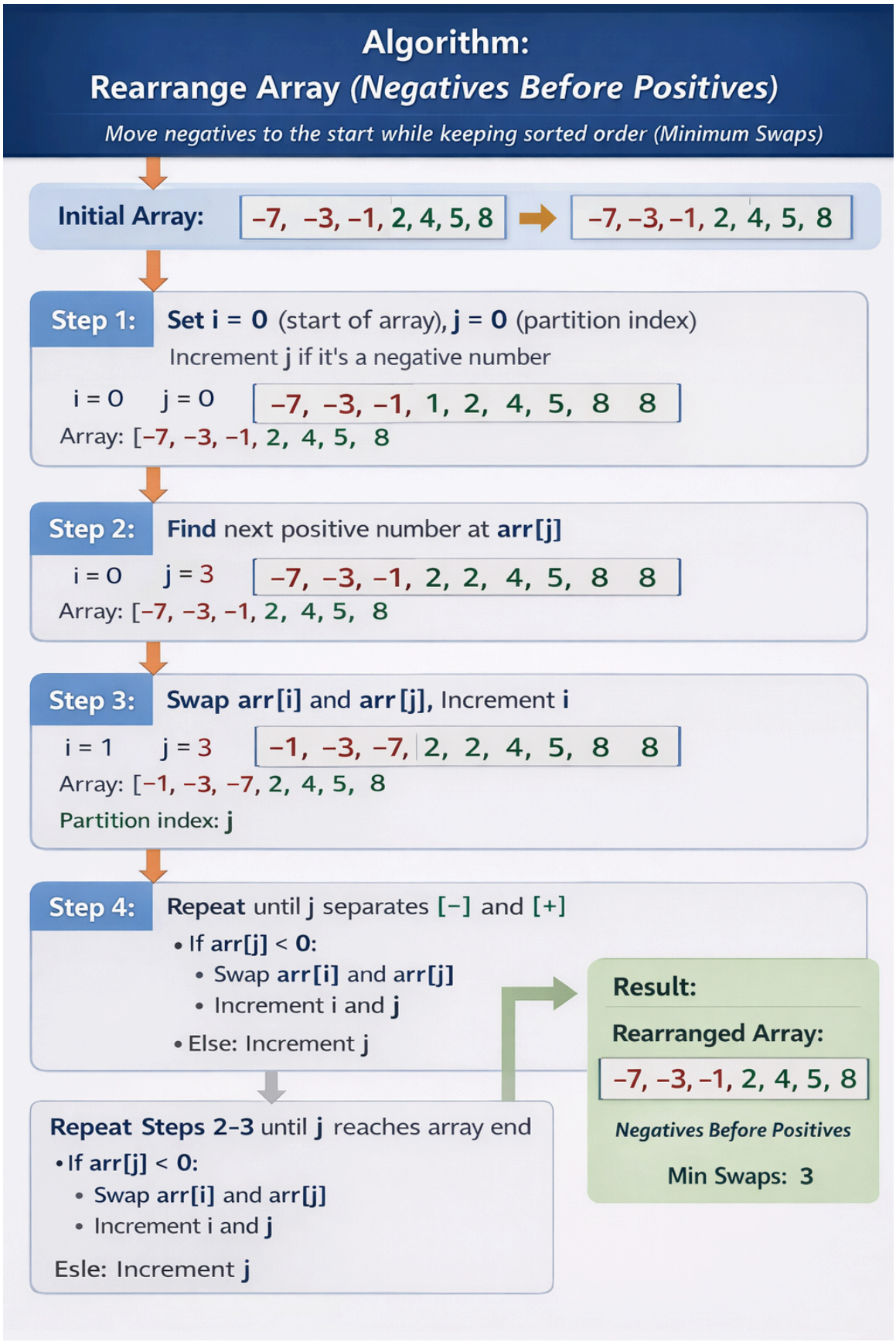
end. At this point, slow is at the Nth node from the end. Time complexity is O(n), space complexity is O(1).

Add Two Numbers (Linked Lists)

```
Node* addTwoNumbers(Node* l1, Node* l2) {
    Node* dummy = new Node(0);
    Node* curr = dummy;
    int carry = 0;

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

        if (l1) {
            sum += l1->data;
            l1 = l1->next;
        }

        if (l2) {
            sum += l2->data;
            l2 = l2->next;
        }

        carry = sum / 10;
        curr->next = new Node(sum % 10);
        curr = curr->next;
    }

    Node* result = dummy->next;
    delete dummy;
    return result;
}
```

Explanation: This adds two numbers represented as linked lists in reverse order (1's digit at head). It traverses both lists simultaneously, adding corresponding digits plus any carry from the previous position. The sum's last digit becomes the new node's value, and sum/10 becomes the new carry. The loop continues until both lists are exhausted and no carry remains. Time complexity is O(max(m,n)) where m and n are list lengths.

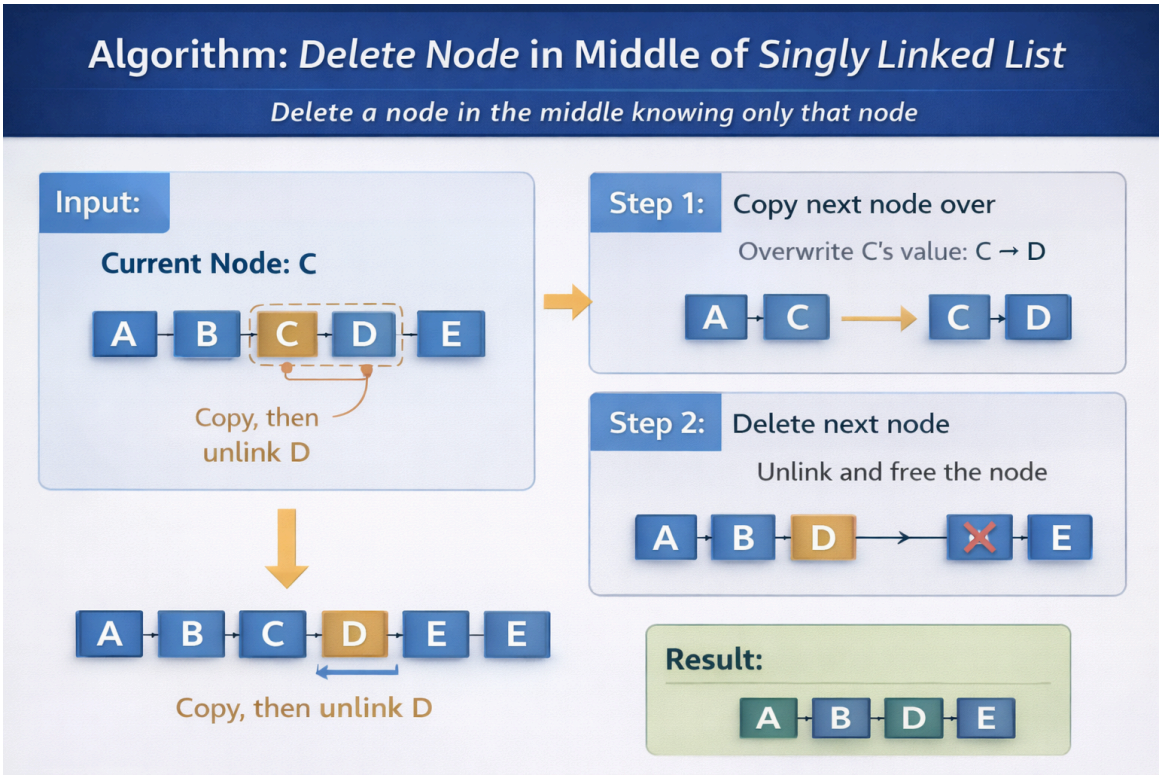
Rearrange Negatives First


Algorithm:
Rearrange Array (Negatives Before Positives)
Move negatives to the start while keeping sorted order (Minimum Swaps)
Initial Array:
-7, -3, -1, 2, 4, 5, 8
-7, -3, -1, 2, 4, 5, 8
Step 1:
Set i = 0 (start of array), j = 0 (partition index)
Increment j if it's a negative number
i = 0 j = 0
-7, -3, -1, 1, 2, 4, 5, 8 8
Array: [-7, -3, -1, 2, 4, 5, 8
Step 2:
Find next positive number at arr[j]
i = 0 j = 3
-7, -3, -1, 2, 2, 4, 5, 8 8
Array: [-7, -3, -1, 2, 4, 5, 8
Step 3:
Swap arr[i] and arr[j], Increment i
i = 1 j = 3
-1, -3, -7, 2, 2, 4, 5, 8 8
Array: [-1, -3, -7, 2, 4, 5, 8
Partition index: j
Step 4:
Repeat until j separates [-] and [+]
• If arr[j] < 0:
• Swap arr[i] and arr[j]
• Increment i and j
• Else: Increment j
Repeat Steps 2-3 until j reaches array end
• If arr[j] < 0:
• Swap arr[i] and arr[j]
• Increment i and j
Esle: Increment j
Result:
Rearranged Array:
-7, -3, -1, 2, 4, 5, 8
Negatives Before Positives
Min Swaps: 3

```
void rearrangeNegativesFirst(int arr[], int n) {
    int i = 0;
    int j = n - 1;

    while (i < j) {
        if (arr[i] < 0) {

            i++;
        } else if (arr[j] >= 0) {

            j--;
        } else {

            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
}
```

Explanation: This rearranges an array to place all negative numbers before positive numbers using a two-pointer approach. The left pointer i finds positive numbers, the right pointer j finds negative numbers, and they swap when both conditions are met. This partitioning runs in O(n) time with O(1) space, similar to the partition step in quicksort.

Delete Node (Given Pointer Only)




```
bool deleteNode(Node* node) {
    if (!node || !node->next) return false;

    Node* next = node->next;
    node->data = next->data;
    node->next = next->next;
    delete next;

    return true;
}
```

Explanation: When only a pointer to the node is given (not the head), we cannot traverse to find the previous node. The solution is to copy the next node's data into the current node and delete the next node instead. This works for all nodes except the tail (no next node to copy from), which is why the function returns false for tail nodes.

---

6. OUTPUT AND SCREENSHOTS

Sample Outputs:

Question 1 - Array Rotation:

```
========================================
Enter your choice: 1

========================================
   Question 1: Rotate Array (Right K Positions)
========================================
Company stores daily revenue of 365 days.
Rotate array by k positions using recursion.

Original Array: [1, 2, 3, 4, 5, 6, 7]
Rotate right by: 3 positions
Rotated Array:  [5, 6, 7, 1, 2, 3, 4]
```

Question 2 - Rearrange Negatives:

```
========================================
Enter your choice: 2

========================================
   Question 2: Rearrange Negatives First
========================================
Rearrange negative numbers before positive.
Keep original sorted order with minimum swaps.

Original Array: [-12, 11, -13, -5, 6, -7, 5, -3, -6]
Rearranged:     [-12, -6, -13, -5, -3, -7, 5, 6, 11]
(All negatives moved to the left)
```

Question 3 - Priority Queue

```
=======================================
Enter your choice: 3

=======================================
   Question 3: Priority Queue (Hospital)
=======================================
Hospital queue with InsertPriority(name, emergencyLevel).
Insert before any patient with higher emergencyLevel.

Inserting patients into emergency queue:
  - John (Priority: 3)
  - Sarah (Priority: 1)
  - Mike (Priority: 2)
  - Emma (Priority: 1)

Patients in priority order:
  1. Sarah (Priority: 1)
  2. Emma (Priority: 1)
  3. Mike (Priority: 2)
  4. John (Priority: 3)
```

Question 4 - Palindrome

```
=======================================
Enter your choice: 4

=======================================
   Question 4: Detect Palindrome (Linked List)
=======================================
Check if linked list is palindrome (e.g., 1→3→7→3→1).
O(n) time, O(1) space using mid-split + reverse.

List: [1 -> 2 -> 3 -> 2 -> 1]
Is Palindrome: Yes
```

Question 5 - Merge Sorted Lists

```
========================================
Enter your choice: 5

========================================
   Question 5: Merge Two Sorted Lists
========================================
Merge two sorted lists (stock prices A & B).
Only pointer relinking, no new nodes created.

List 1: [1 -> 3 -> 5]
List 2: [2 -> 4 -> 6]
Merged: [1 -> 2 -> 3 -> 4 -> 5 -> 6]
```

Question 6 - Detect Cycle Start

```
========================================
Enter your choice: 6

========================================
   Question 6/20: Detect Cycle Start (Floyd's)
========================================
Use Floyd's Cycle Algorithm to find loop start.
Example: A→B→C→D→E→C (returns C)

List with cycle: 1 -> 2 -> 3 -> 4 -> 5 -> (back to 3)
Cycle detected! Start node value: 3
```

Question 7 - Remove Consecutive Absents

```
======================================
Enter your choice: 7

======================================
   Question 7: Remove Consecutive Absents
======================================
Student attendance: 1=present, 0=absent.
RemoveConsecutiveAbsents(k): Delete ≥k consecutive 0's.

Original List (1=Present, 0=Absent): [1 -> 0 -> 0 -> 0 -> 1 -> 0 -> 1]
Remove consecutive absents >= 3
Result List:                         [1 -> 1 -> 0 -> 1]
```

Question 8 - Delete Node (pointer only)

```
======================================
Enter your choice: 8

======================================
   Question 8/18: Delete Node (Given Pointer Only)
======================================
Delete a node when only pointer to that node is given.
Example: Delete 'c' from a→b→c→d→e → a→b→d→e

Original List: [1 -> 2 -> 3 -> 4 -> 5]
Delete node with value: 3
Result List:   [1 -> 2 -> 4 -> 5]
```

Question 9 - Deque

```
========================================
Enter your choice: 9

========================================
   Question 9: Double-Ended Queue (Deque)
========================================
Linked List based Deque supporting:
InsertFront, InsertRear, RemoveFront, RemoveRear

InsertFront(10)
InsertRear(20)
InsertFront(5)
InsertRear(30)

Operations:
RemoveFront: 5
RemoveRear:  30
RemoveFront: 10
RemoveFront: 20
```

Question 10 - Merge Sort List

```
========================================
Enter your choice: 10

========================================
   Question 10: Merge Sort Linked List
========================================
Sort linked list using Merge Sort.
Optimized to detect when already sorted.

Original List: [5 -> 2 -> 8 -> 1 -> 9 -> 3]
Sorted List:   [1 -> 2 -> 3 -> 5 -> 8 -> 9]
```

Question 11 - Sort Stack

```
========================================
Enter your choice: 11

========================================
   Question 11: Sort Stack (Recursion)
========================================
Sort stack in ascending order recursively.
No extra stack/array allowed.

Pushing elements: 5 2 8 1 9 3
Sorted Stack (top to bottom): 3 9 1 8 2 5
```

Question 12 - Check Brackets

```
========================================
Enter your choice: 12

========================================
   Question 12: Check Brackets Balance (Extended)
========================================
Validate expression with mixed brackets: () {} [] <>
Detects: extra closing, unclosed, wrong-type mismatch

Test 1: "{[()]}"
Result: Balanced

Test 2: "{[(])}"
Result: Wrong-type mismatch: expected '[' but found ']' at position 4

Test 3: "{{[[(())]]}}"
Result: Balanced

Test 4: "{[}"
Result: Wrong-type mismatch: expected '{' but found '}' at position 3
```

Question 13 - Stack using Queues

```
========================================
Enter your choice: 13

========================================
   Question 13: Stack Using Two Queues
========================================
Implement Stack where Push is O(n) and Pop is O(1).
Using 2 Queues only.

Pushing elements: 10, 20, 30
Top element: 30
Pop: 30
Top element: 20
Pop: 20
Push: 40
Top element: 40
```

Question 14 - Sort Queue

```
========================================
Enter your choice: 14

========================================
   Question 14: Sort Queue Using Stack
========================================
Train bogies arranged in Queue must leave sorted.
Use one extra queue + one stack.

Original Queue: [5, 1, 4, 2, 3]
Sorted Queue:   [1, 2, 3, 4, 5]
```

Question 15 - Min Stack

```
========================================
Enter your choice: 15

========================================
   Question 15: Min Stack Implementation
========================================
Every push/pop maintains minimum in constant time.
No extra arrays/lists allowed.

Push(5)
  Top: 5, Min: 5
Push(3)
  Top: 3, Min: 3
Push(7)
  Top: 7, Min: 3
Push(2)
  Top: 2, Min: 2
Pop()
  Top: 7, Min: 3
Pop()
  Top: 3, Min: 3
```

Question 16 - Remove Duplicates

```
========================================
Enter your choice: 16

========================================
   Question 16: Remove Duplicates (Unsorted List)
========================================
Remove duplicates from an unsorted linked list.

Original List: [1 -> 2 -> 3 -> 2 -> 4 -> 1]
After Removal: [1 -> 2 -> 3 -> 4]
```

Question 17 - Nth from End

```
========================================
Enter your choice: 17

========================================
   Question 17: Nth to Last Element
========================================
Find the Nth to last element of singly linked list.

List: [1 -> 2 -> 3 -> 4 -> 5]
Find 2nd node from end
Result: 4
```

Question 18 - Delete Middle Node

```
========================================
Enter your choice: 18

========================================
   Question 8/18: Delete Node (Given Pointer Only)
========================================
Delete a node when only pointer to that node is given.
Example: Delete 'c' from a→b→c→d→e → a→b→d→e

Original List: [1 -> 2 -> 3 -> 4 -> 5]
Delete node with value: 3
Result List:   [1 -> 2 -> 4 -> 5]
```

Question 19 - Add Two Numbers

```
========================================
Enter your choice: 19

========================================
   Question 19: Add Two Numbers (Lists)
========================================
Numbers represented as linked lists (reverse order).
Example: (3→1→9→9) + (5→9→2) = (8→0→2→0→1)

Number 1: [2 -> 4 -> 3] (represents 342)
Number 2: [5 -> 6 -> 4] (represents 465)
Sum:      [7 -> 0 -> 8] (represents 807)
```

Question 20 - Find Loop Beginning

```
========================================
Enter your choice: 20

========================================
   Question 6/20: Detect Cycle Start (Floyd's)
========================================
Use Floyd's Cycle Algorithm to find loop start.
Example: A→B→C→D→E→C (returns C)

List with cycle: 1 -> 2 -> 3 -> 4 -> 5 -> (back to 3)
Cycle detected! Start node value: 3
```

Question 28 - Menu

```
~/University/Data Structures/project-assignment

./project


=======================================
   DATA STRUCTURES PROJECT ASSIGNMENT
=======================================
PART-1 (ARRAY & LINKED LIST):
1.  Rotate Array (Right K Positions)
2.  Rearrange Negatives First
3.  Priority Queue (Hospital)
4.  Detect Palindrome (Linked List)
5.  Merge Two Sorted Lists
6.  Detect Cycle Start (Floyd's)
7.  Remove Consecutive Absents
8.  Delete Node (Given Pointer Only)
9.  Double-Ended Queue (Deque)
10. Merge Sort Linked List

PART-2 (STACK & QUEUE):
11. Sort Stack (Recursion)
12. Check Brackets Balance (Extended)
13. Stack Using Two Queues
14. Sort Queue Using Stack
15. Min Stack Implementation

PART-3 (CRACKING THE CODE):
16. Remove Duplicates (Unsorted List)
17. Nth to Last Element
18. Delete Middle Node
19. Add Two Numbers (Lists)
20. Find Loop Beginning

0.  Exit
=======================================
Enter your choice:
```

7. DISCUSSION AND LEARNING REFLECTION

This project significantly enhanced our understanding of data structures and algorithmic thinking. Through hands-on implementation, we discovered the intricate relationship between theoretical concepts and practical coding challenges.

The Floyd's Cycle Detection algorithm was particularly enlightening, demonstrating how clever pointer manipulation can solve complex problems efficiently. Understanding why resetting one pointer to the head finds the cycle start deepened our appreciation for mathematical proofs in algorithms.

Implementing recursive stack sorting without auxiliary structures challenged our recursive thinking. We learned that recursion can elegantly solve problems that seem to require extra space by utilizing the call stack itself.

The AVL Tree implementation taught us the importance of maintaining invariants. Each rotation must preserve BST properties while restoring balance, requiring meticulous attention to pointer updates. This experience emphasized the value of thorough testing with edge cases.

Graph algorithms revealed the power of different traversal strategies. BFS's level-order exploration versus DFS's depth-first approach illustrated how the same problem can be approached from different perspectives, each with unique advantages.

Most importantly, this project improved our debugging skills. Tracking pointer references, visualizing data structure states, and systematically testing edge cases became second nature. The integration phase taught us the value of modular code design and the importance of consistent interfaces across different implementations.

---

8. CHALLENGES FACED

Memory Management: Properly deallocating dynamically allocated nodes in linked lists and trees was challenging initially. We had to ensure every `new` operation had a corresponding `delete` to prevent memory leaks, especially in error paths.

Pointer Manipulation: Maintaining correct pointer relationships, particularly in doubly-linked structures and during list reversal operations, required careful tracking. Drawing diagrams on paper proved essential for visualizing pointer changes.

Edge Cases: Handling boundary conditions like empty lists, single-node lists, and operations on tail nodes required special attention. The node deletion function, for example, fails when the node to delete is the tail, requiring a special check.

Recursion Depth: For very large data structures, recursive implementations like merge sort and stack sorting risked stack overflow. We learned to recognize when iterative approaches might be more practical despite their complexity.

AVL Tree Rotations: Determining which rotation to apply (LL, RR, LR, RL) based on balance factors was initially confusing. Creating a decision flowchart helped systematize the rotation selection process.

Testing Complexity: With 28 different implementations, comprehensive testing was time-consuming. We developed a systematic approach, creating test cases for normal operation, edge cases, and stress tests with large inputs.

Integration Challenges: Merging all implementations into a single program while maintaining clean code organization required restructuring. We used classes and namespaces to avoid naming conflicts and improve modularity.

These challenges enhanced our problem-solving abilities and taught us the importance of systematic debugging, thorough testing, and clear documentation.

---

9. CONCLUSION

This comprehensive project successfully demonstrated mastery of fundamental data structures including Arrays, Linked Lists, Stacks, Queues, AVL Trees, and Graphs. Through implementing 28 distinct problems, we gained practical experience in algorithm design, complexity analysis, and code optimization.

The project reinforced that understanding data structures goes beyond memorizing operations, It requires deep insight into when and why to use specific structures. We learned that the choice of data structure significantly impacts program efficiency, and that seemingly simple operations can hide complex algorithmic challenges.

The integration of all implementations into a cohesive menu-driven program demonstrated our ability to design scalable, maintainable software. This project serves as a strong foundation for advanced topics in algorithms and system design.

Most importantly, this assignment cultivated our problem-solving mindset. We learned to break complex problems into manageable components, design clean interfaces, and test thoroughly. These skills transcend specific programming languages or technologies and will prove invaluable throughout our careers in software engineering.

The experience of collaborative coding, peer code reviews, and collective debugging sessions prepared us for real-world software development environments where teamwork and clear communication are essential.

---

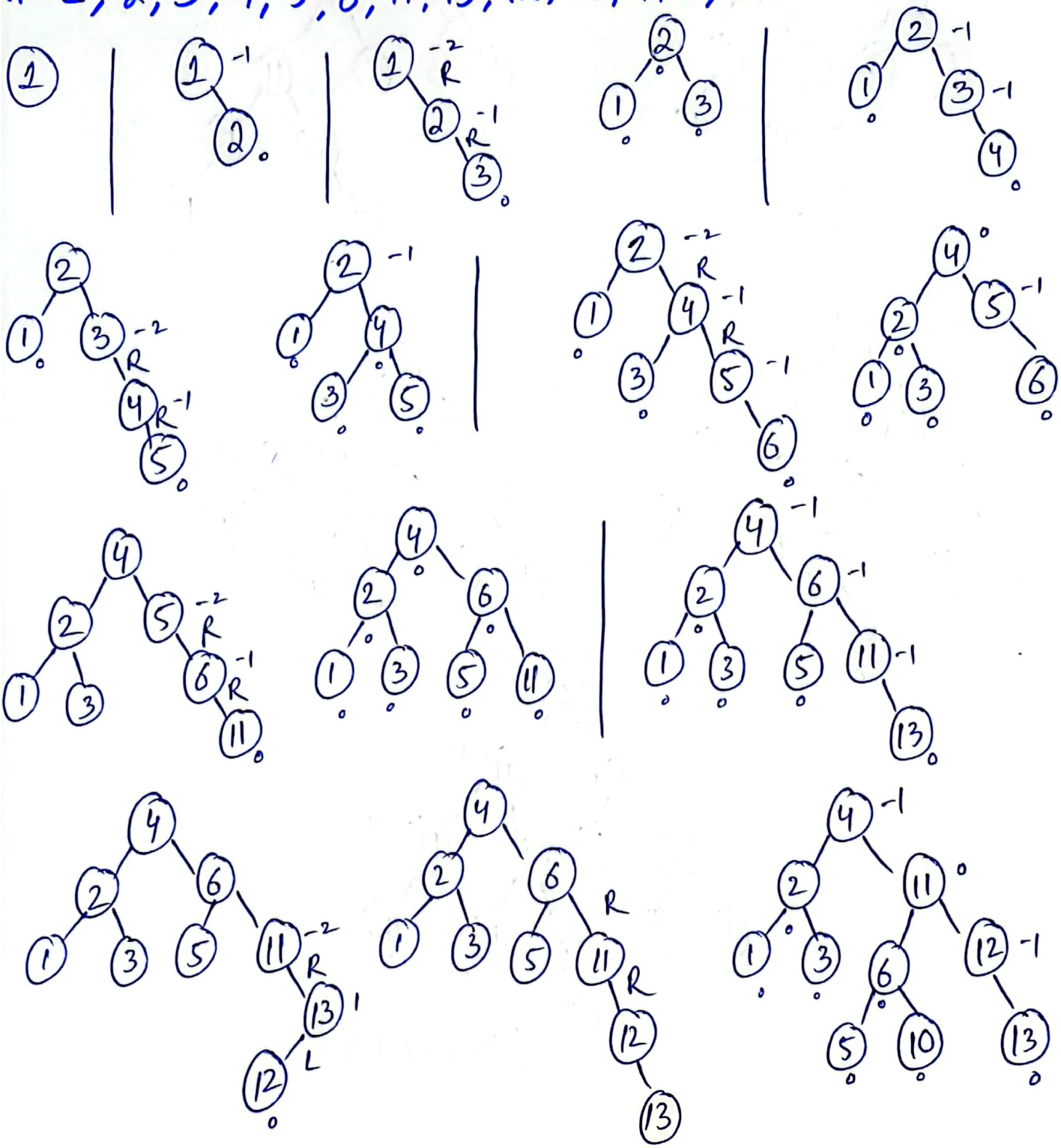
10. REFERENCES

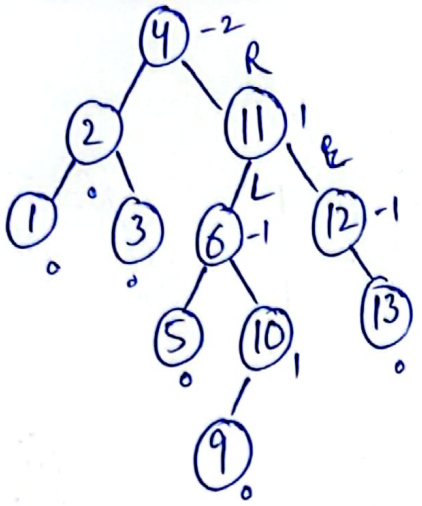
1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
3. Karumanchi, N. (2016). *Data Structures and Algorithms Made Easy*. CareerMonk Publications.
4. GeeksforGeeks. (2024). Data Structures. Retrieved from https://www.geeksforgeeks.org/data-structures/
5. LeetCode. (2024). Problems Archive. Retrieved from https://leetcode.com/problemset/
6. Cracking the Coding Interview by Gayle Laakmann McDowell (6th Edition).
7. Class lecture notes and lab demonstrations by Mr. Humayun Majeed, Data Structures Lab, Fall 2025.
8. C++ Documentation. (2024). Standard Library Reference. Retrieved from https://en.cppreference.com/
9. Visualgo. (2024). Algorithm Visualizations. Retrieved from https://visualgo.net/
10. Stack Overflow Community discussions on data structure implementations and debugging techniques.

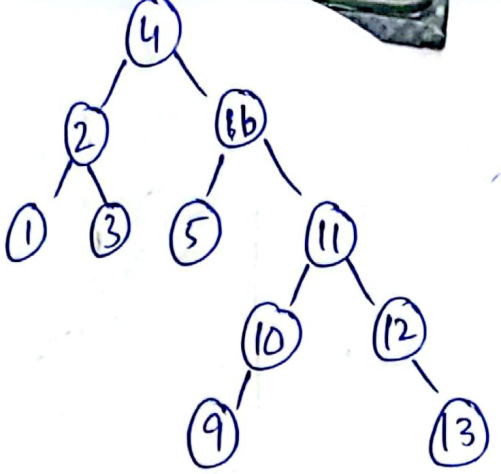
Questions 21 - 27 Continues (Then Dry Runs)

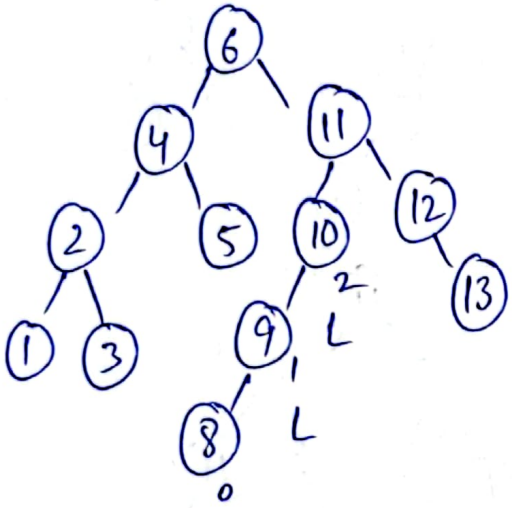
AVL- Tree

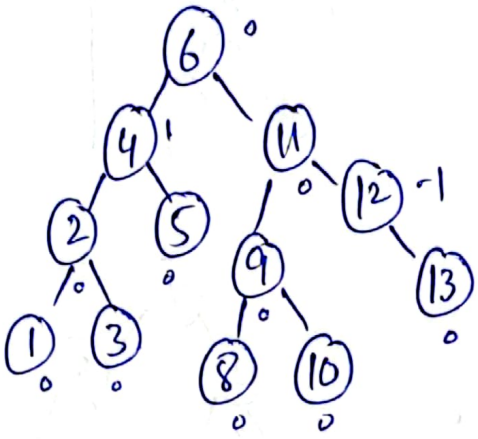
21 - 1, 2, 3, 4, 5, 6, 11, 13, 12, 10, 9, 8, 7.

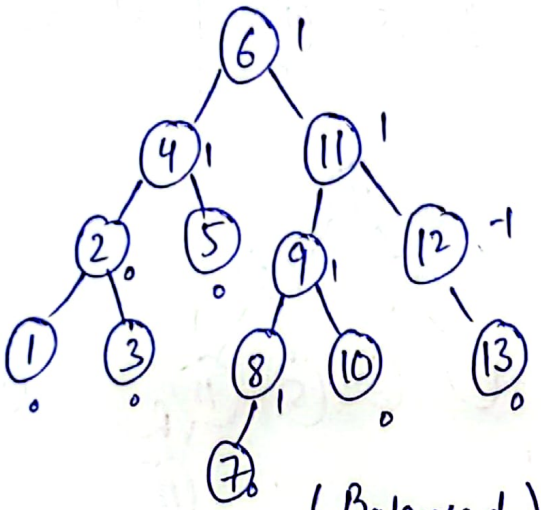






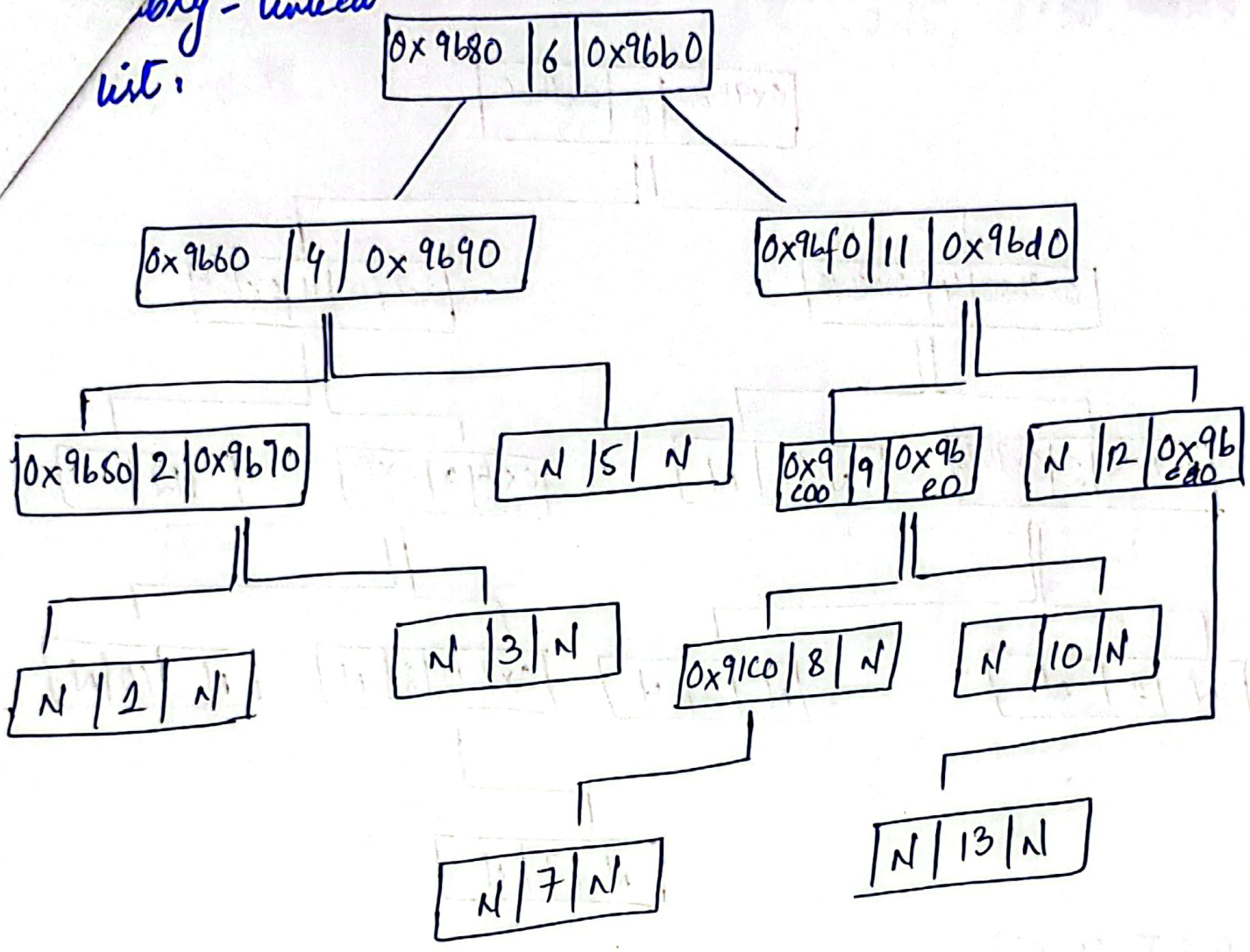




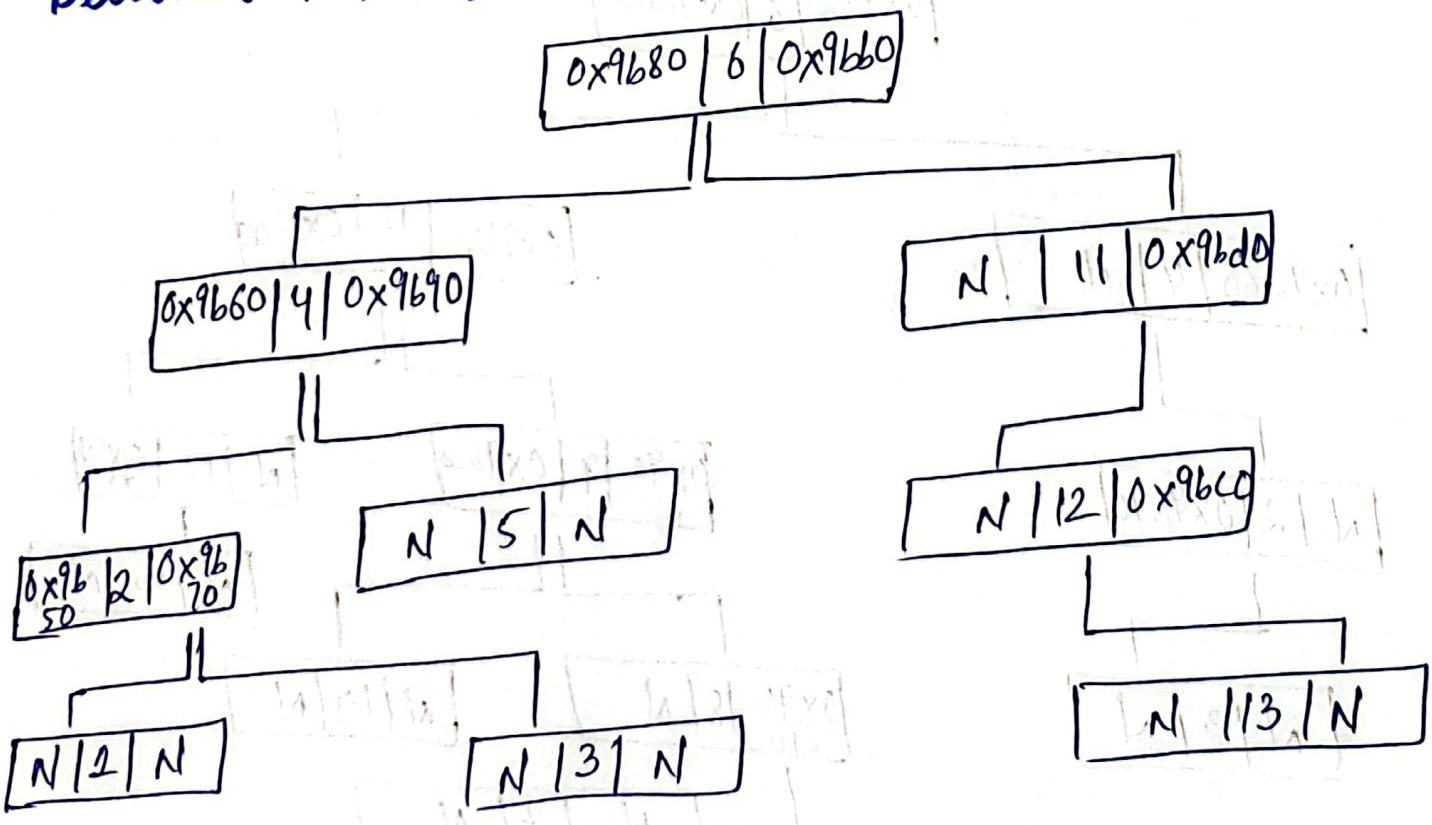


(Balanced)

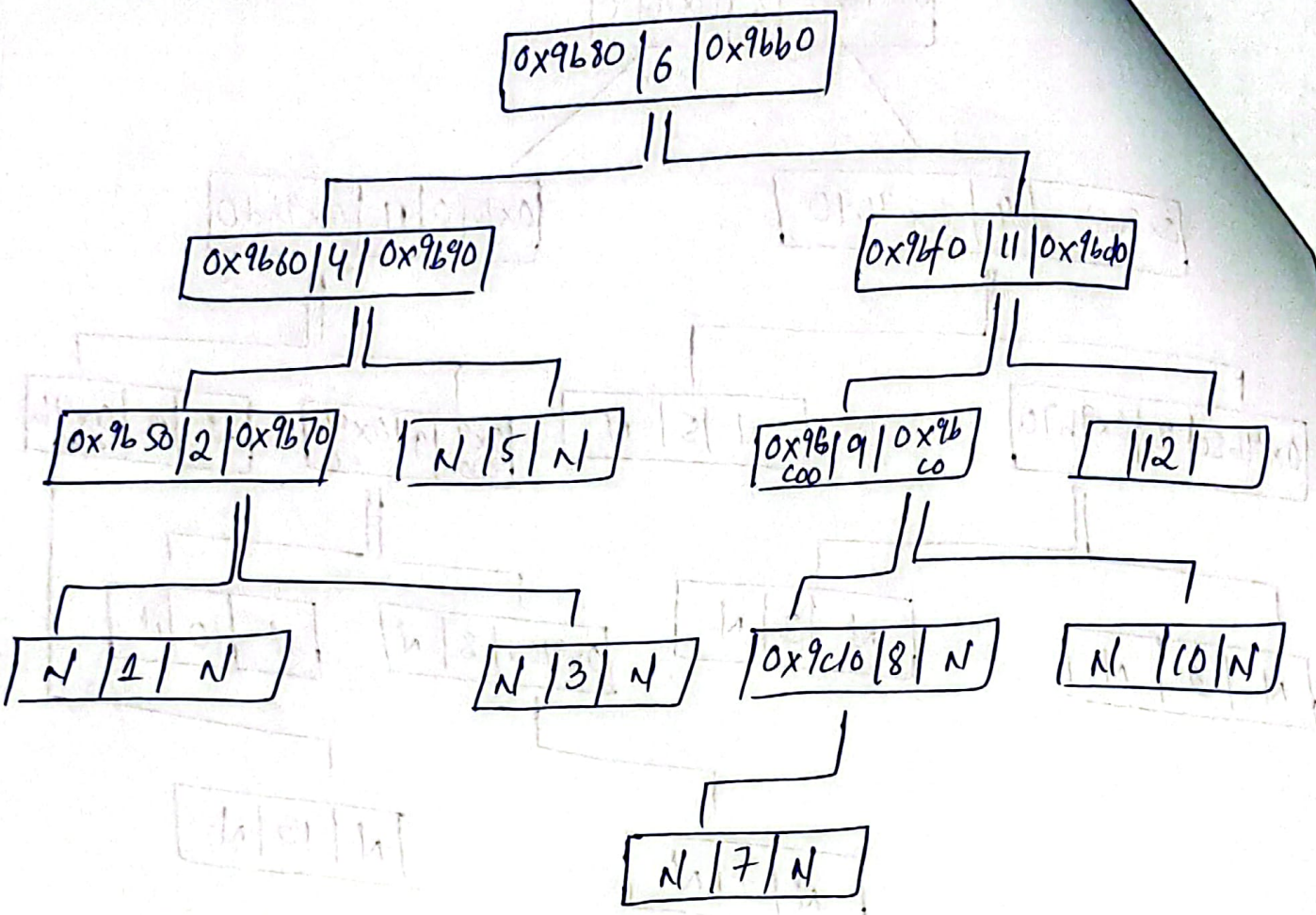
bly - linked
list:



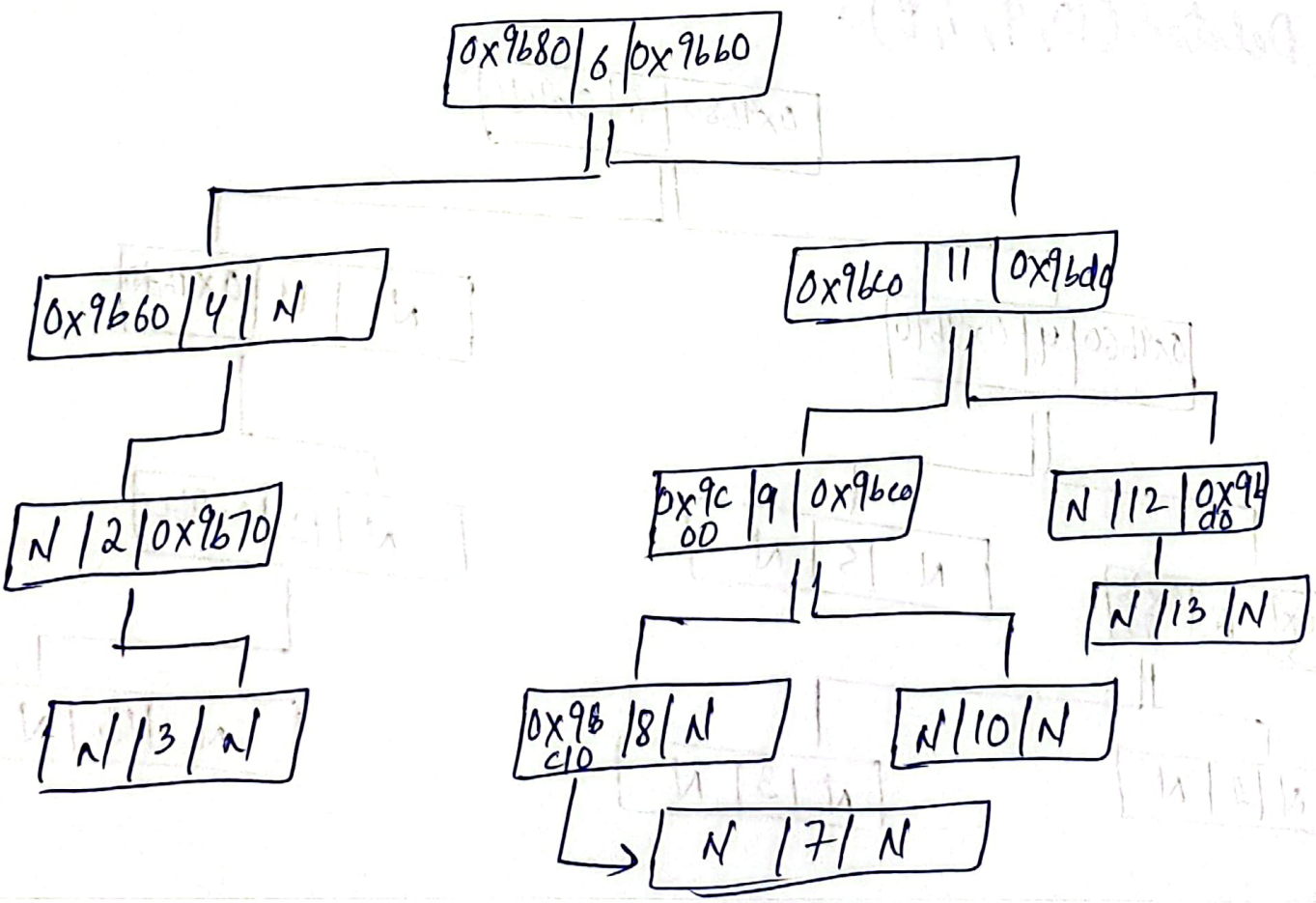
Deleted (10, 9, 7, 8):



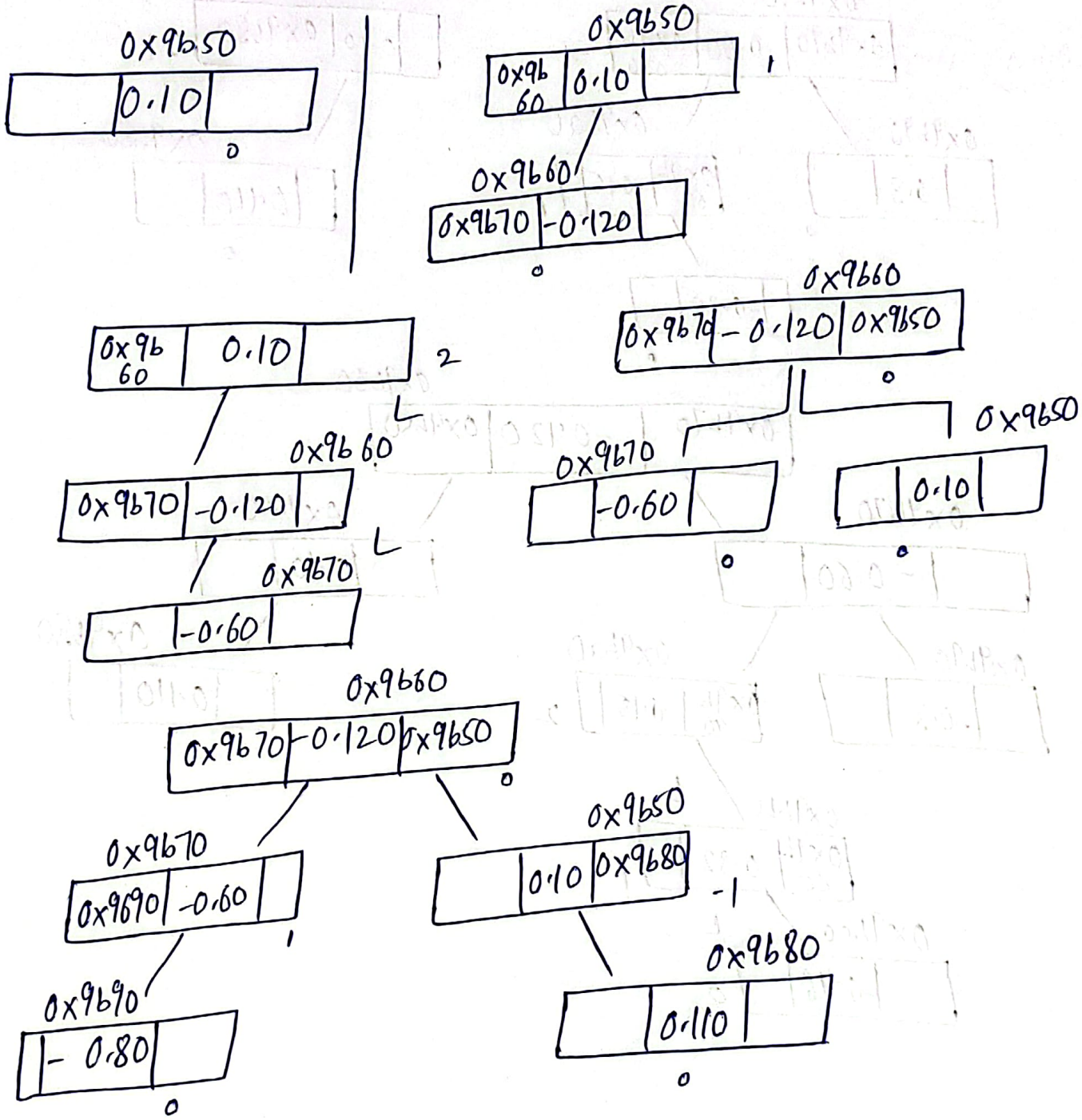
Delete 13:

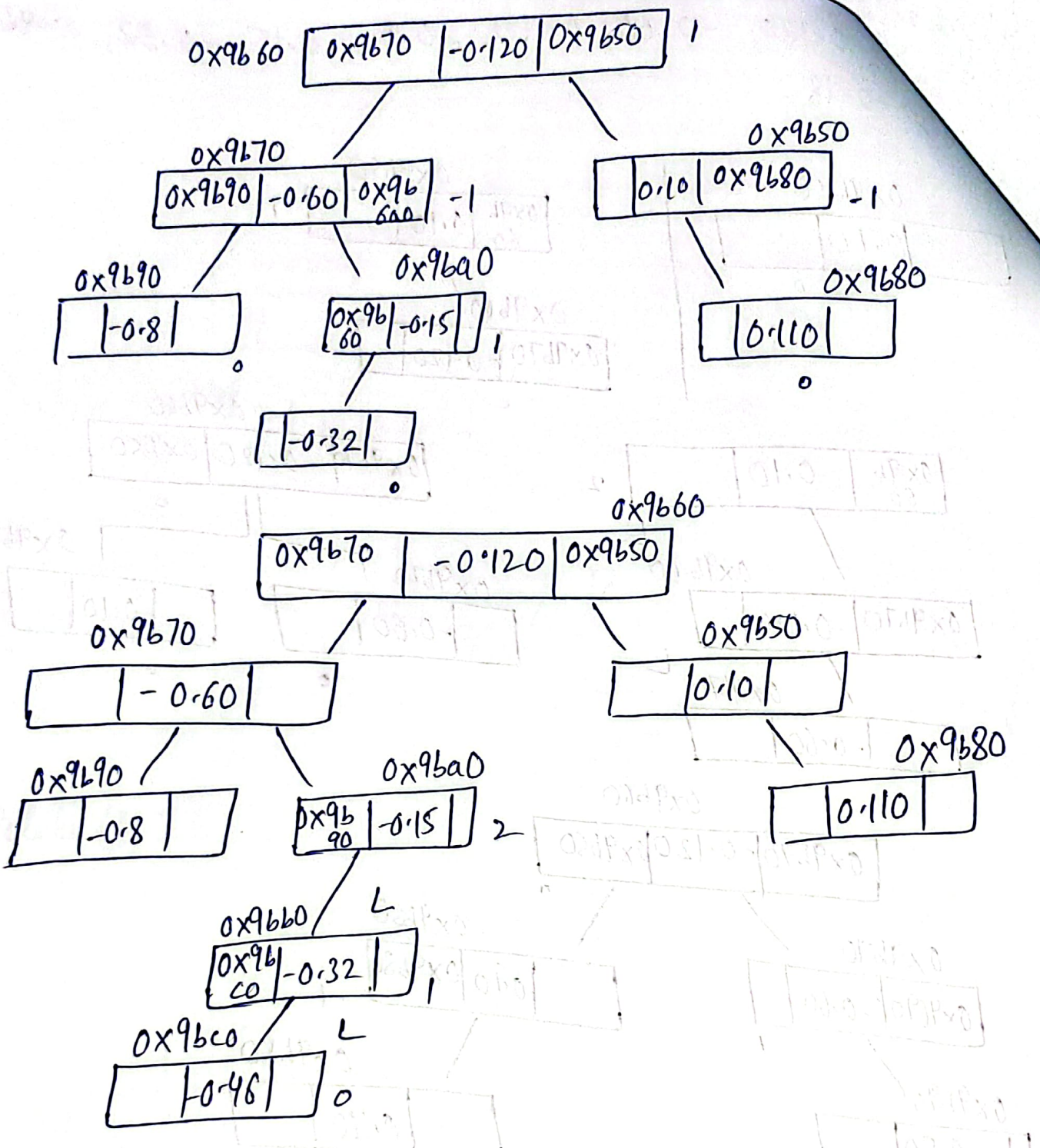


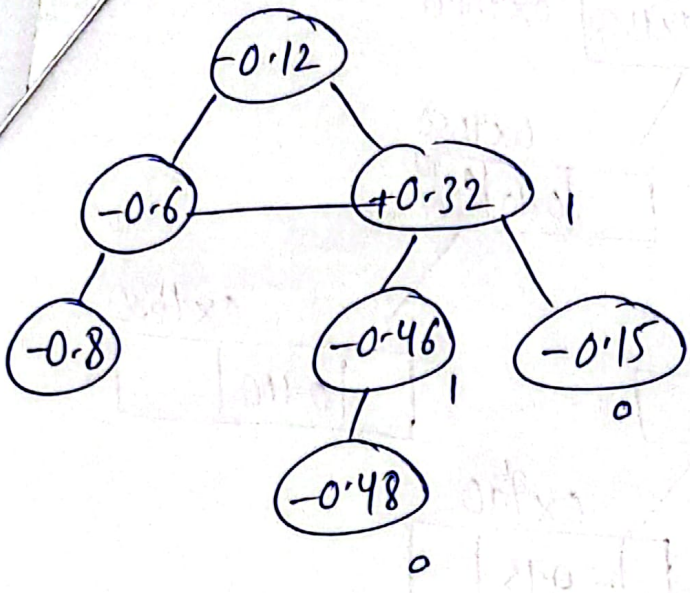
Delete (1, 5):

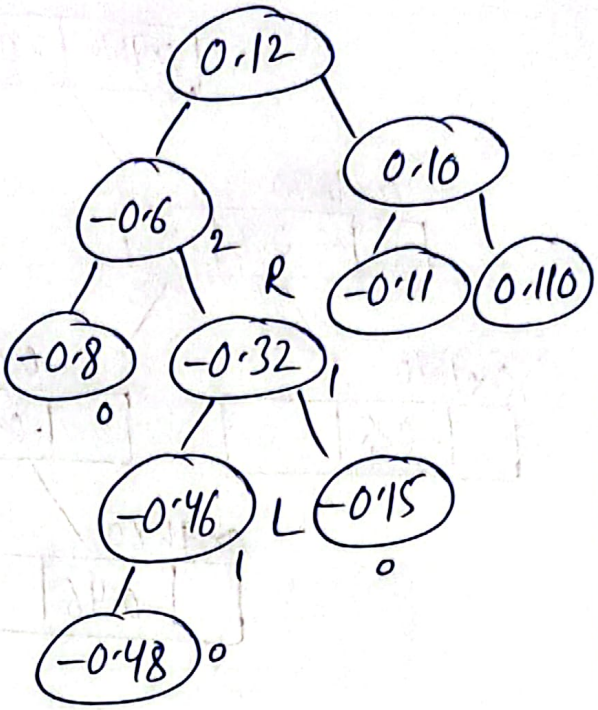


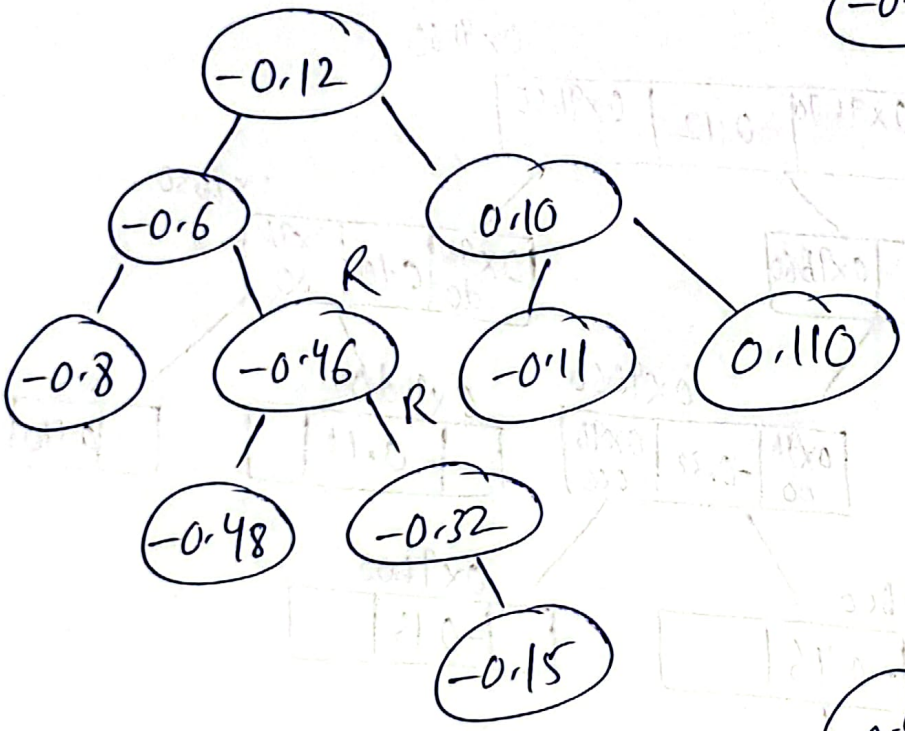
.10, -0.120, -0.60, 0.110, -0.8, -0.15, -0.32, -0.46, 0.11, -0.48.

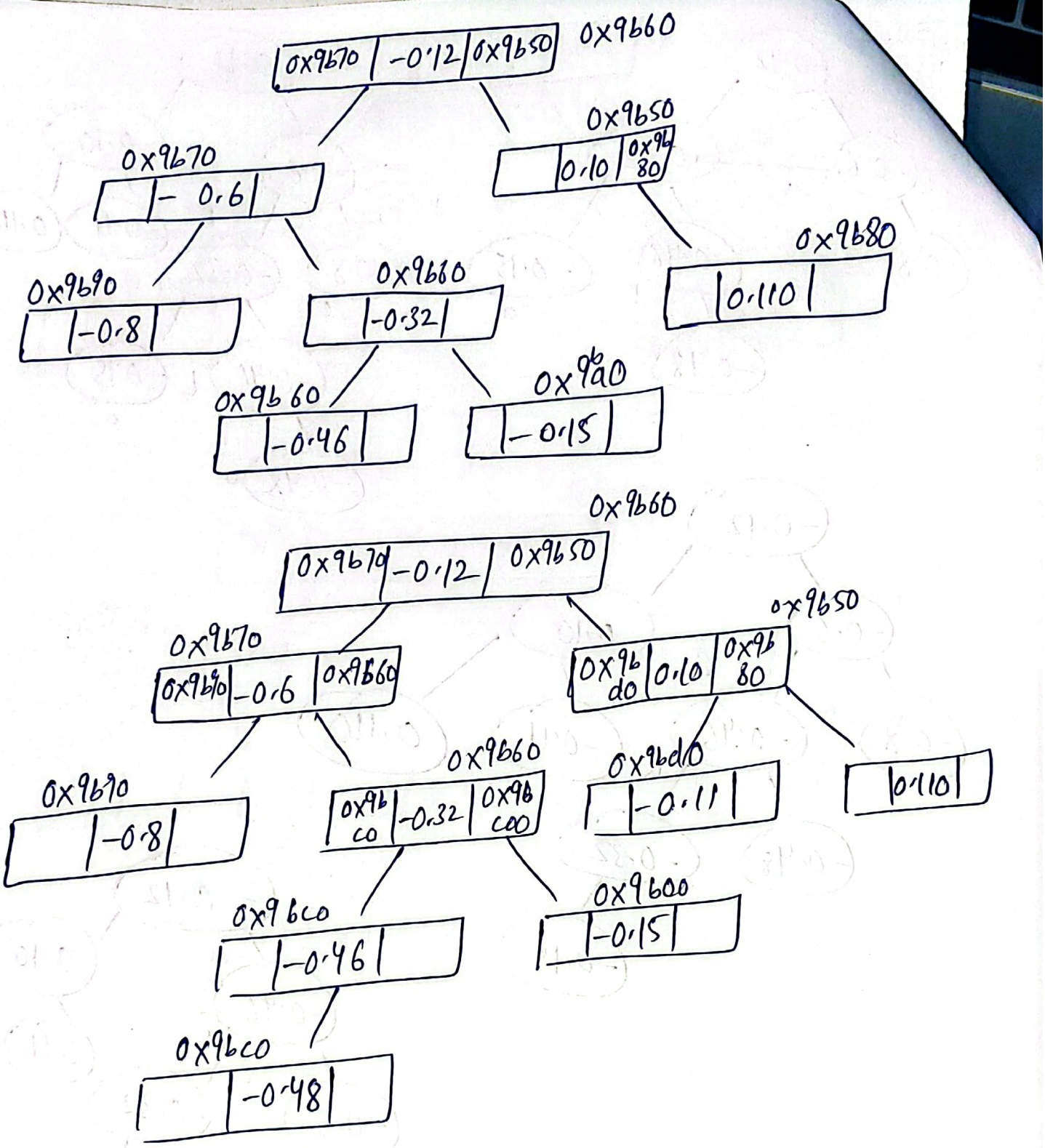


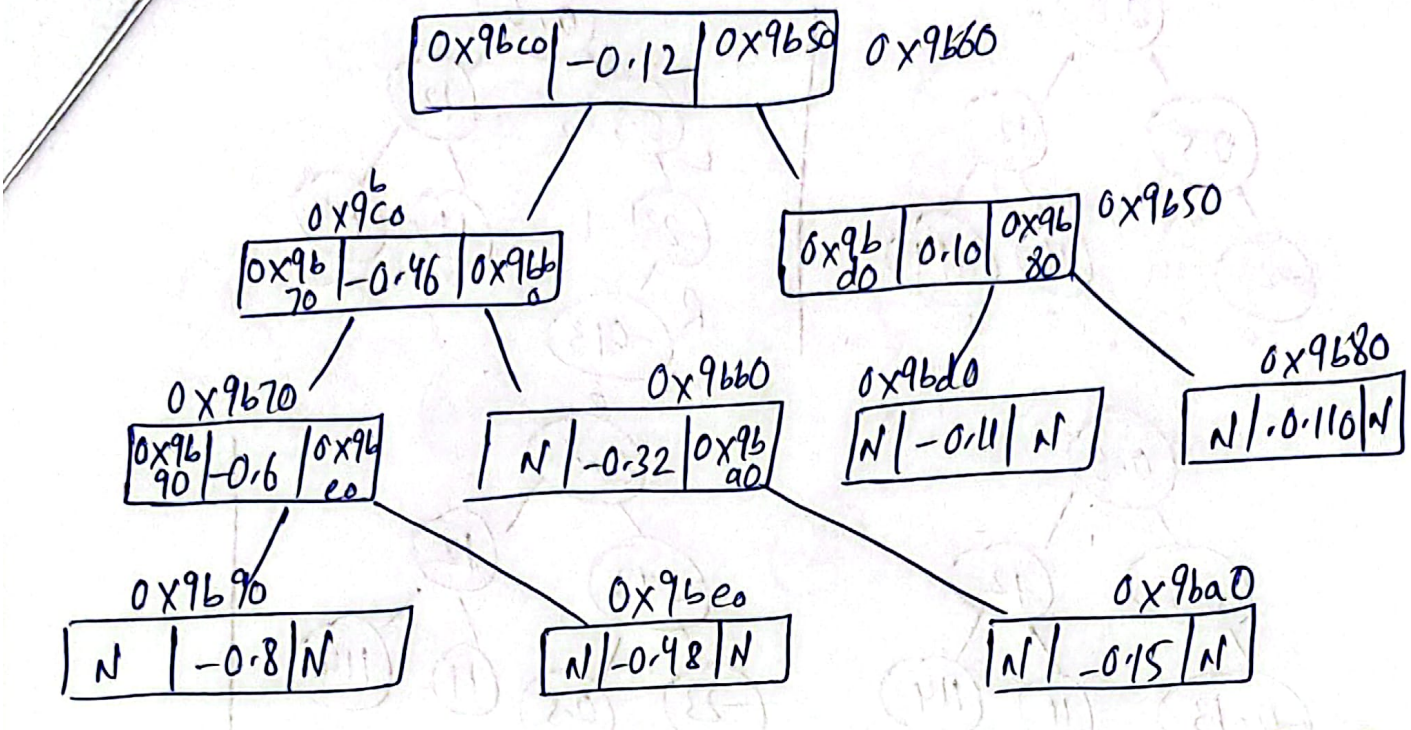




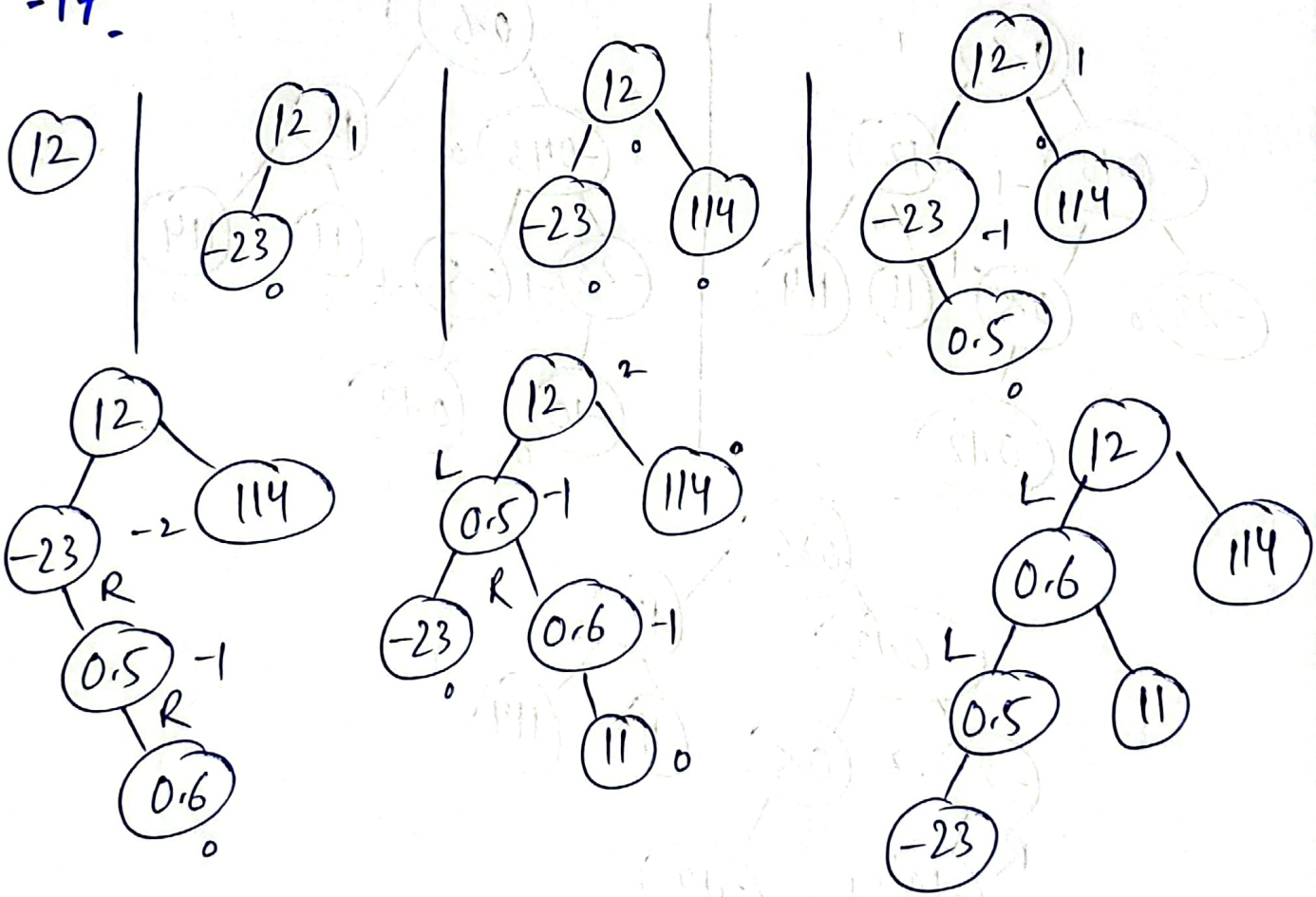


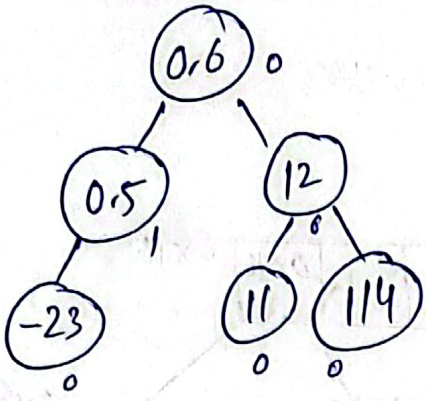


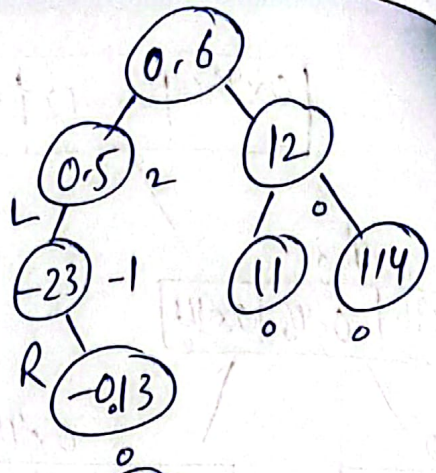


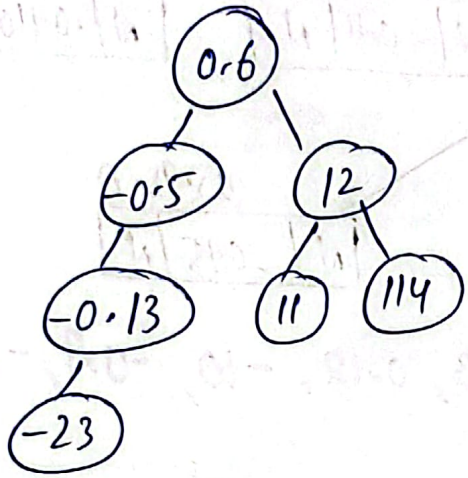


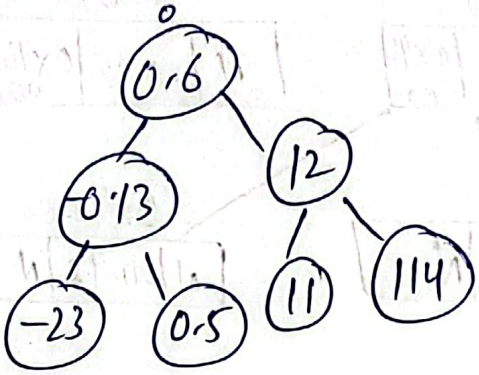
23 - 12, -23, 114, 0.5, 0.6, 11, -0.13, 0.12, -10, -0.9, -28, -17.

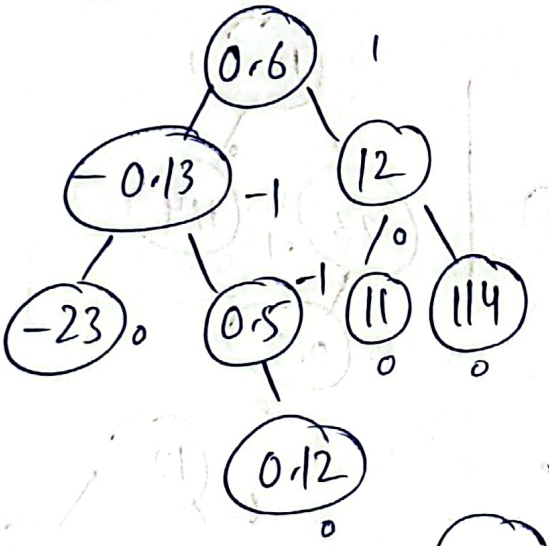


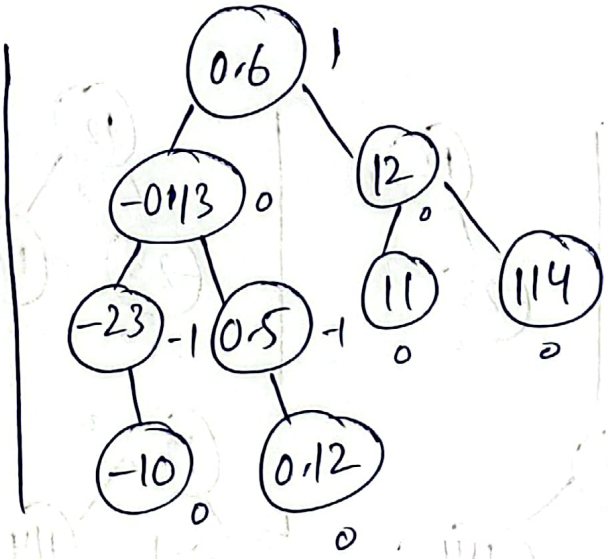


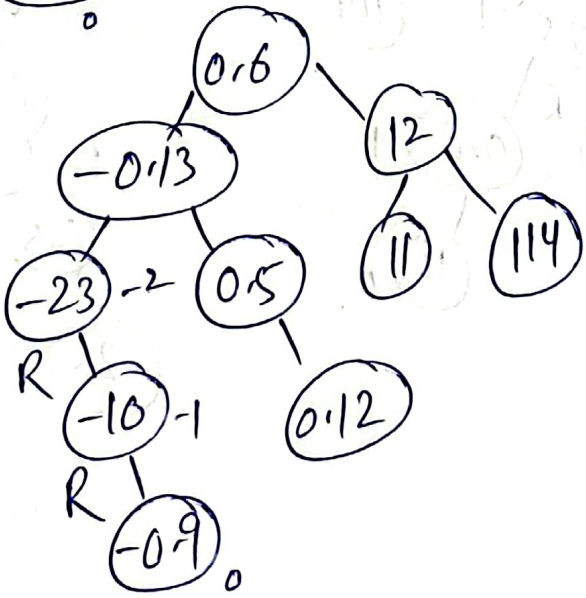


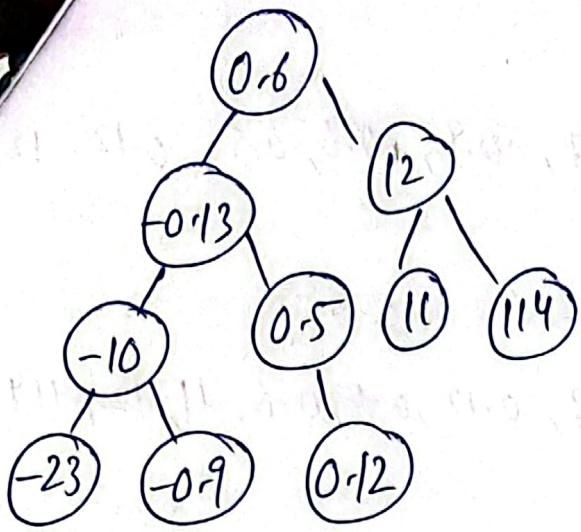


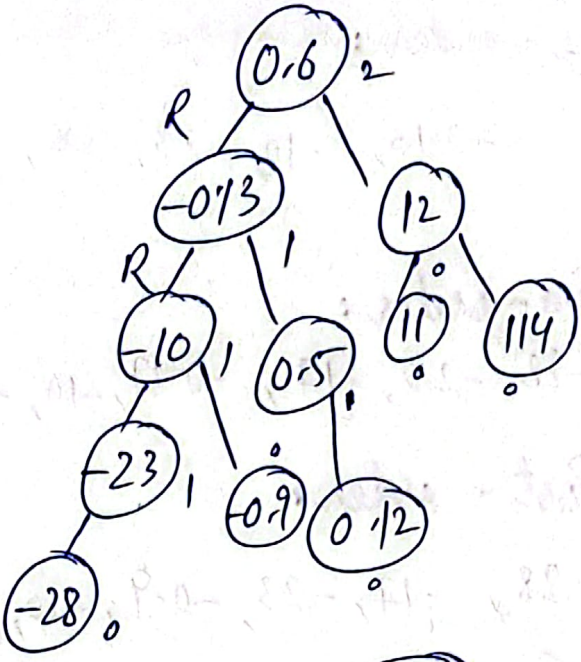


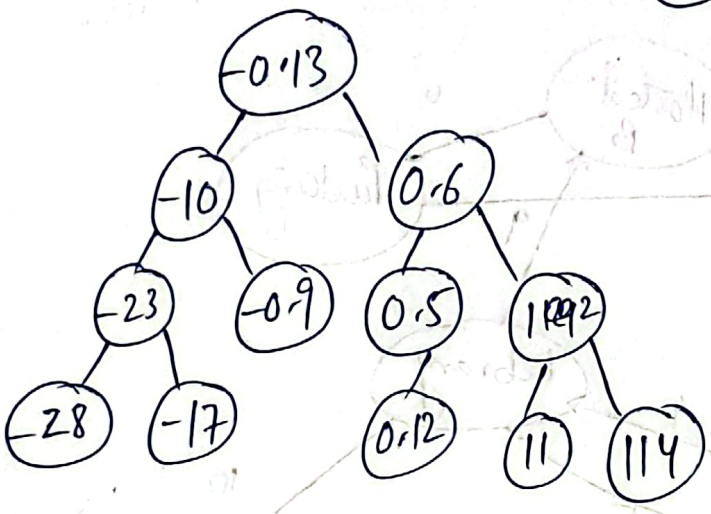


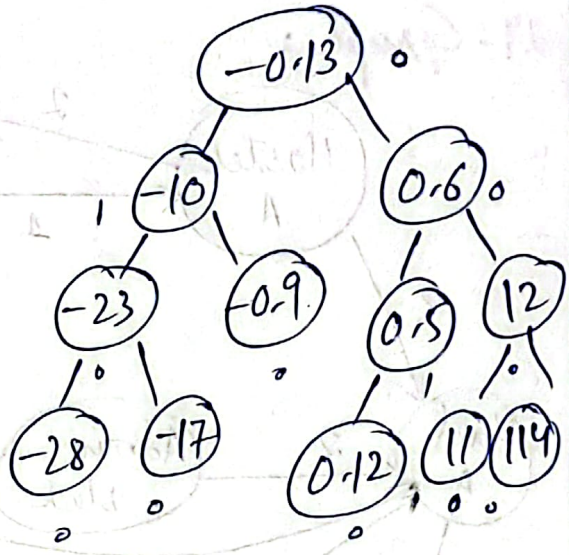


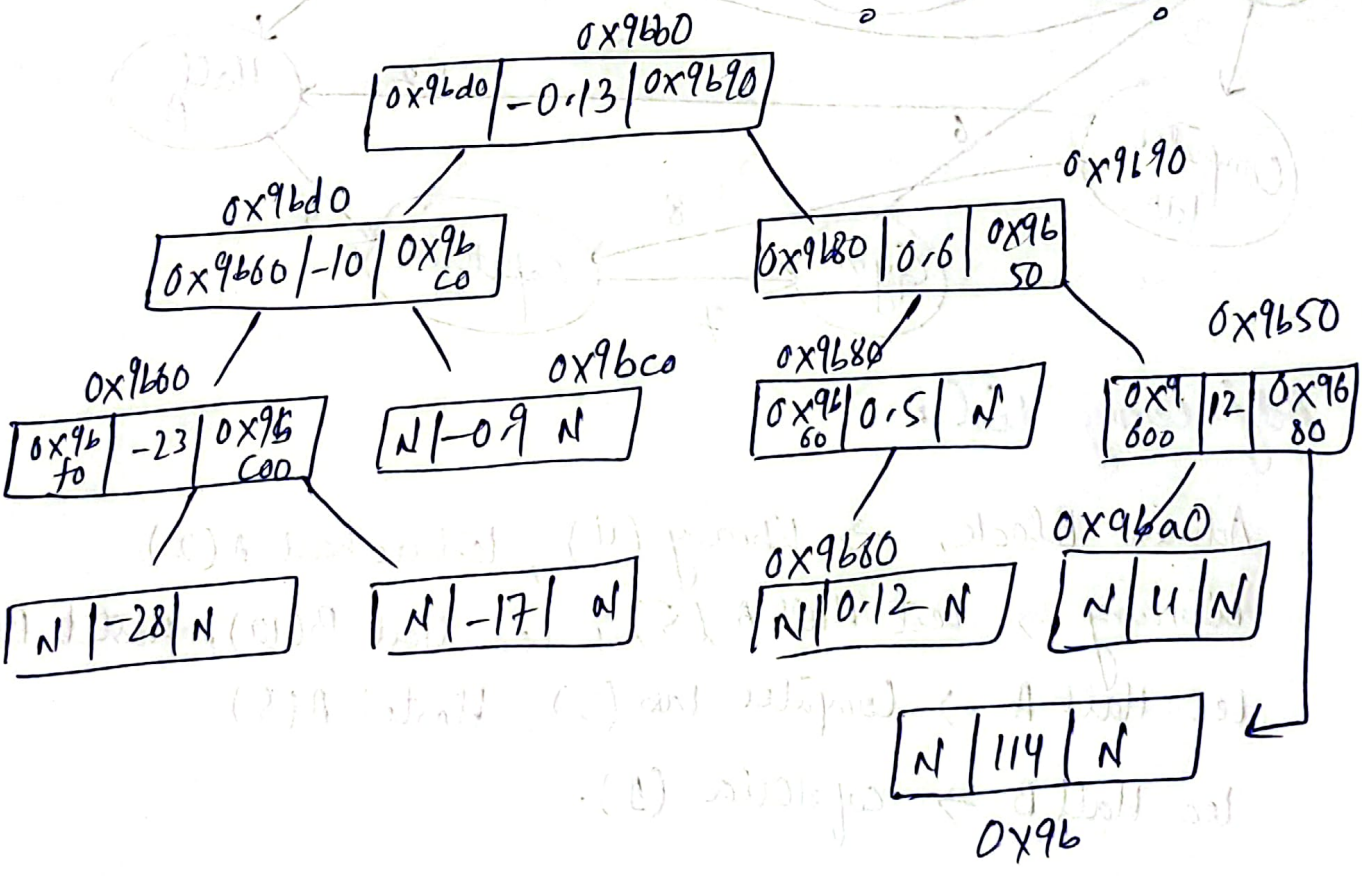












Pre-order:

-0.13, -10, -23, -28, -17, -0.9, +0.6, 0.5, 0.12, 11, 12, 114.

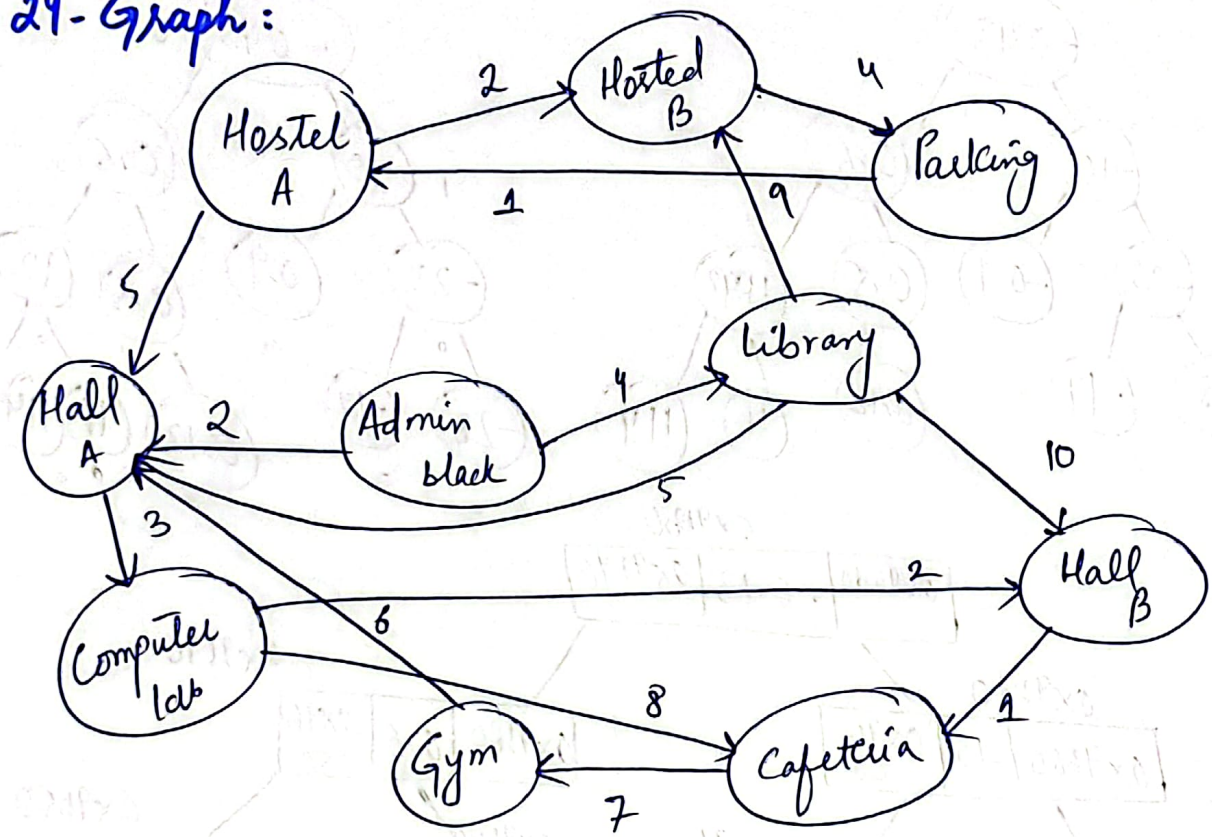
In-order:

-28, -23, -17, -0.89, -10, -0.13, 0.12, 0.5, 0.6, 11, 12, 114.

Post-order:

-28, -17, -23, -0.9, -10, -0.12, 0.5, 11, 114, 12, 0.6, -0.13.

24- Graph:



Adjacency list:

Admin Block → library (4), lecture hall A(2)

Library → Lect Hall A (5), Lec Hall B(10), Hostel B(9)

Lec Hall A → Computer lab (3), Hostel A(5)

Lec Hall B → cafeteria (1).

Computer lab → lec Hall B (2), cafeteria (8)

cafeteria → Gym (7)

Gym → lec Hall A (6)

Hostel A → Hostel B (2)

Hostel B → Parking (4)

Parking → Hostel A (1)

Adjacency matrix:

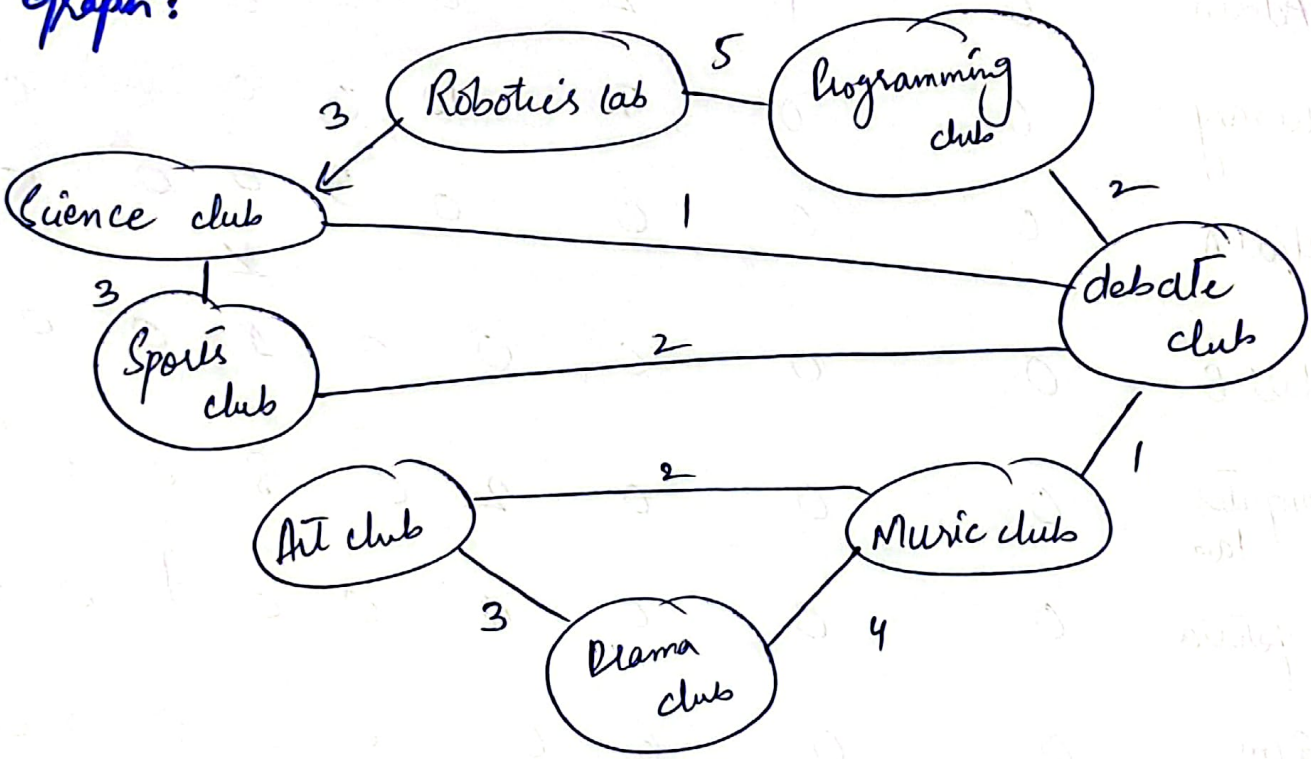
	Admin	library	hall A	B	C.L	cf	Gym	Hostel A	B
Admin	0	4	2	0	0	0	0	0	0
library	0	0	5	10	0	0	0	0	9
Hall A	0	0	0	0	3	0	0	5	0
Hall B	0	0	0	0	0	1	0	0	0
Computer lab	0	0	0	2	0	8	0	0	0
cafeteria	0	0	0	0	0	0	7	0	0
Gym	0	0	6	0	0	0	0	0	0
Hostel A	0	0	0	0	0	0	0	0	0
Hostel B (4) P.L	0	0	0	0	0	0	0	0	0
Parking	0	0	0	0	0	0	0	1	0

MST: The concept of minimum spanning is only undirected graphs to find the path with minimum cost. In graph, it is directed so MST cannot be searched.

BFS: Admin, block, library, Lec hall A, Computer lab, hostel A, lec hall B, hostel B, cafe, Parking, Gym.

DFS: Admin. block, library, lec hall A, lab, lec hall B, cafe, Gym, hostel A, hostel B, Parking.

25- Graph:



Robotics club – Programming club (5), Science club (3)

Programming club – debate club (2)

Dramma club – music club (4), art club (3).

Music club – debate club (1)

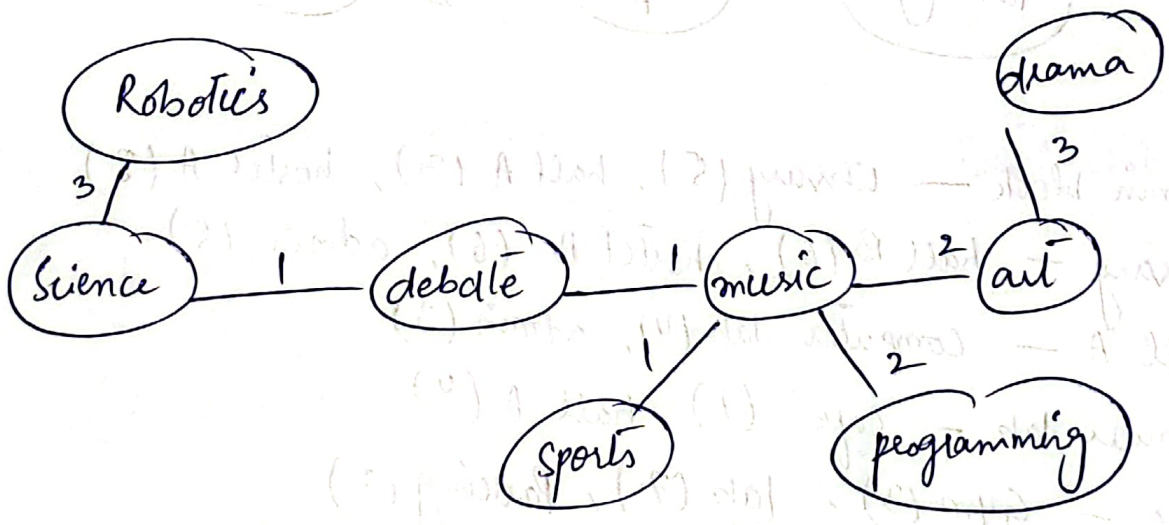
Sports club – debate club (2), science club (3)

club – music club (2)

ence club – debate club (1)

Adjacency matrix:

	Robotics	Program.	Drama	Music	Sports	Art	Sci	Deb
Robotics	0	5	0	0	0	0	3	
Programming	0	0	0	0	0	0	0	0
Drama	0	0	0	4	0	0	0	
Music	0	0	0	0	0	0	3	
Sports	0	0	0	0	0	0	3	
Art	0	0	0	2	0	0	0	
Science	0	0	0	0	0	0	0	
Debate	0	2	0	1	2	0	1	

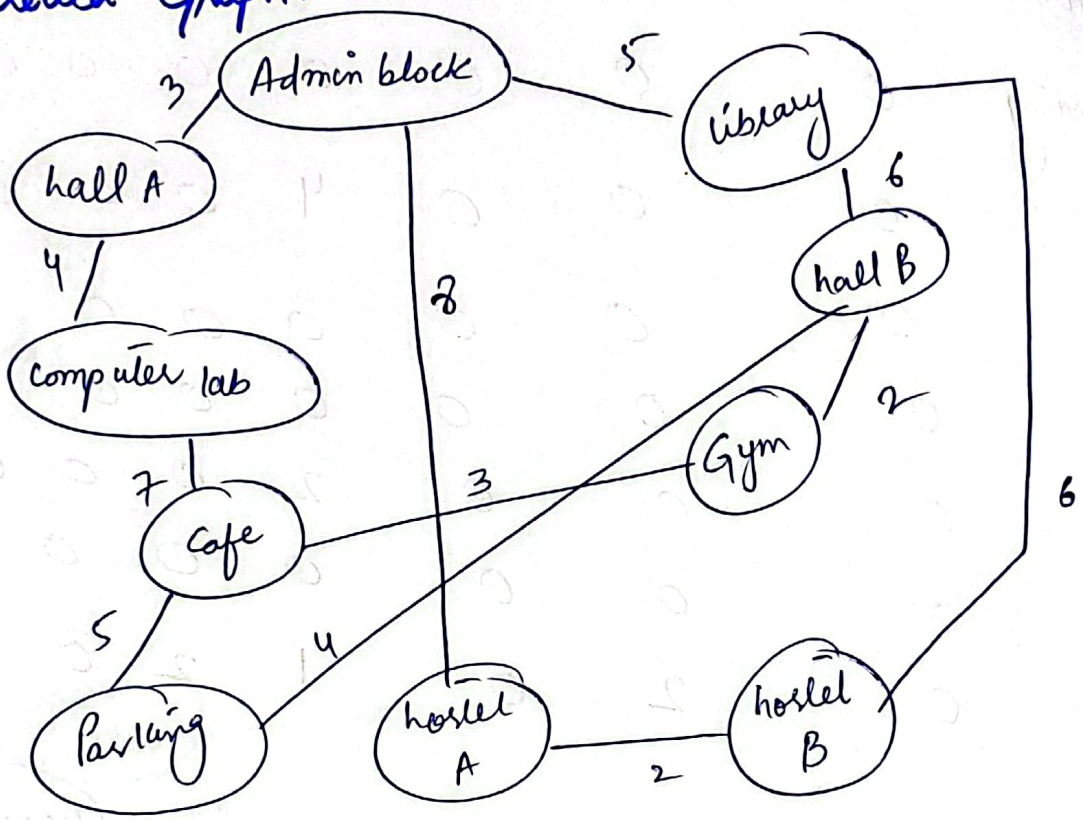


Total cost = 14

BFS: Robotics, programming, Science, Debate, Sports, drama, art.

DFS: Robotics, programming, debate, music, drama, art, Sports science.

26 - Undirected Graph:



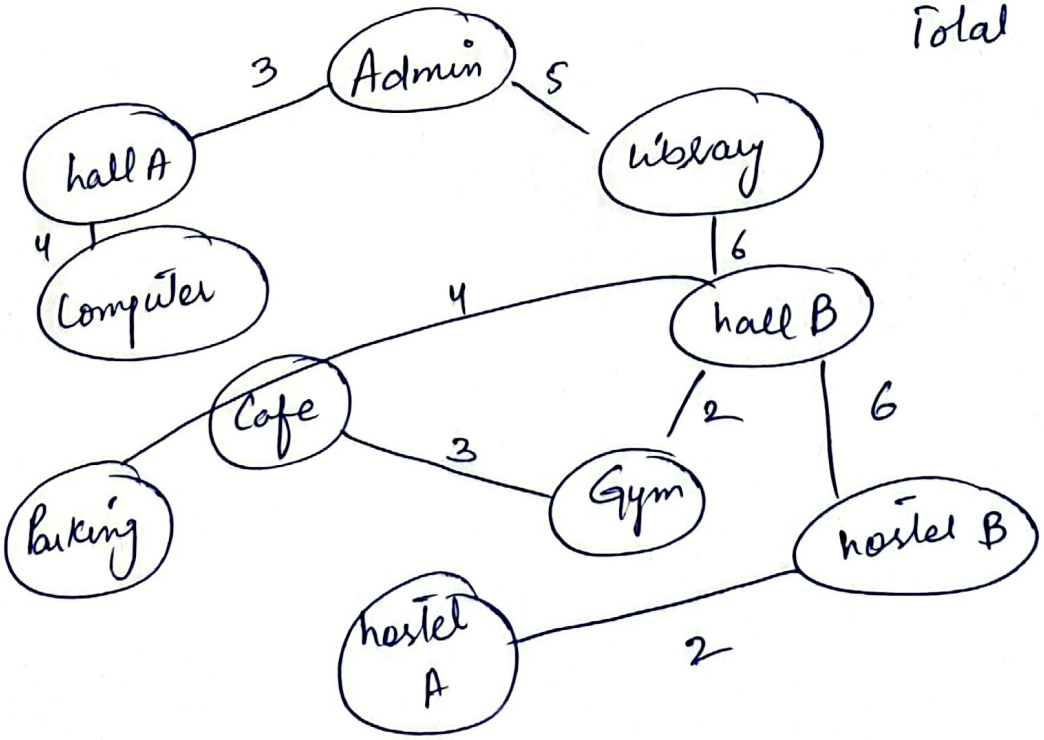
Admin block — library (5), hall A (3), hostel A (8)
library — hall B (6), hostel B (6), admin (5).
Hall A — computer lab (4), admin (3)
computer lab — cafe (7), hall A (4)
cafe — Gym (3), lab (7), Parking (5)
hostel A — admin (8), hostel B (2)
hostel B — library (6), hostel A (2)
Parking — cafe (5)

Hall B — Parking (4), Gym (2), Library (3).

Gym — hall B (2).

	Admin	Lib.	Hall A	C.L	Cafe	Hostel A	Hostel B	Park	Hall B	Gym.
Admin	0	5	3	0	0	8	0	0	0	0
Library	5	0	0	0	0	0	6	0	6	0
Hall A	3	0	0	4	0	0	0	0	0	0
Computer	0	0	4	0	7	0	0	0	0	0
cafe	0	0	0	7	0	0	6	5	0	3
Hostel A	8	0	0	0	0	0	2	0	0	0
Hostel B	0	6	8	0	0	0	0	0	0	0
Parking	0	0	0	0	5	0	0	0	0	0
Hall B	0	3	0	0	0	0	0	4	0	2
Gym.	0	0	0	0	0	0	0	0	2	0

Total cost = 35



BFS: Admin, library, lec hall A, Lec hall B, hostel B, Computer, Gym, Parking, cafe.

DFS: Admin, library, lec hall B, Gym, cafe, computer, lec hall B, Parking, hostel B, hostel A.

DRY- RUN

→ **Arrays:**

Original Array = [1, 2, 3, 4, 5, 6, 7]

Rotate right by K = 3

[5, 6, 7, 1, 2, 3, 4]

arr = [1, 2, 3, 4, 5, 6, 7] ; n = 7 ; K = 3

K = K % n = 3 % 7 = 3

Reverse (arr, 0, 6) → reverse whole array.

1	2	3	4	5	6	7	after swapping	7	2	3	4	5	6	1

7	2	3	4	5	6	1	after swapping	7	6	3	4	5	2	1

7	6	3	4	5	2	1	after swapping	7	6	5	4	3	2	1

Final output is: [7, 6, 5, 4, 3, 2, 1]

Reverse (arr, 0, K - 1)

7	6	5	4	3	2	1	after swapping	5	6	7	4	3	2	1

Final output is: [5, 6, 7, 4, 3, 2, 1]

Reverse (arr, K, n - 1)

5	6	7	4	3	2	1	after swapping	5	6	7	1	3	2	4

5	6	7	1	3	2	4

after swapping

5	6	7	1	2	3	4

Rotated array and the final output is:

5	6	7	1	2	3	4

2 - Arrays:

Original Array is: [-12, 11, -13, -5, 6, -7, 5, -3, -6]

```
i = 0
j = n - 1
while (i < j);
    if (arr[i] < 0)
        i++;
    else if arr[j] >= 0
        j--;
    else
        swap(arr[i], arr[j]), i++, j--.
```

i = 0; j = 8

array = [-12, 11, -13, -5, 6, -7, 5, -3, -6]

arr[i] = -12 < 0 → move i

i = 1 ; j = 8 (Array unchanged)

arr[i] = 11 ≥ 0

arr[j] = -6 < 0. (swapping)

swap (arr[1], arr[8])

array = [-12, -6, -13, -5, 6, -7, 5, -3, 11]

i = 2 ; j = 7

arr[i] = -13 < 0 → move i

i = 3 ; j = 7 (Array unchanged)

arr[i] = -5 < 0 → move i

i = 4 ; j = 7 (Array unchanged)

arr[i] = 6 ≥ 0

arr[j] = -3 ≤ 0 (swapping)

swap (arr[4], arr[7])

array = [-12, -6, -13, -5, -3, -7, 5, 6, 11]

i = 5 ; j = 6

arr[i] = -7 < 0 → move i

i = 6 ; j = 6

loops end (i >= j)

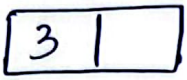
Final output is: [-12, -6, -13, -5, -3, -7, 5, 6, 11]

(All negative moved to the left).

3- Single linked list :

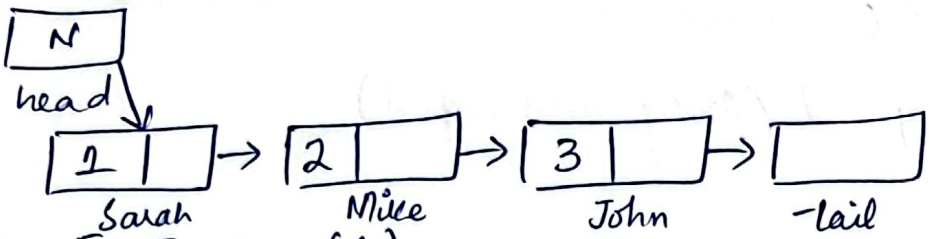
head = null ; tail = null ;

Insert John (3) :

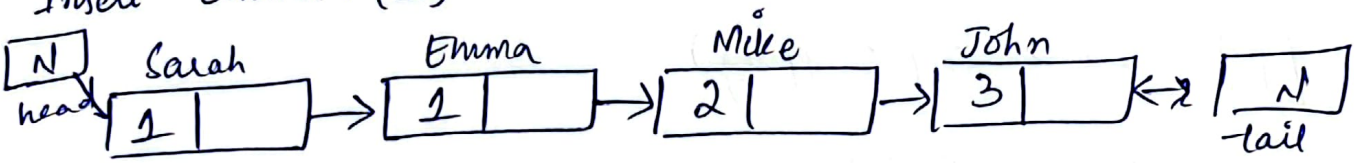


Insert Sarah (1) :

Insert Mike (2) :



Insert Emma : (1)



Final Priority order (matches)

Sarah (Priority 1)

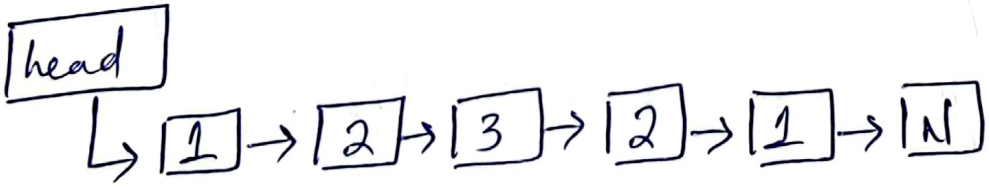
Emma (Priority 1)

Mike (Priority 2)

John (Priority 3).

4- Doubly linked list :

Initial State :



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

slow -> 1
fast -> 1

slow = slow -> next
fast = fast -> next -> next

slow -> 2
fast -> 3

slow = slow -> next
fast = fast -> next -> next

slow -> 3
fast -> 1

middle node = 3

→ Reverse Second Half

```
Node *second = reverseList(slow -> next);
slow -> next = null;
```

Before Reverse:
Second half = [2] -> [1] -> null

After Reverse:
second -> [1] -> [2] -> null

Split list:
first half: [1] -> [2] -> [3] -> null
second half: [1] -> [2] -> null

```
Node *first = head;
bool isPal = true;
```

first -> [1] ; second -> [1]

Compare Both Halves :

Comparison 1 :

first → data == second → data

1 == 1

first → [2]

second → [2]

first → data == second → data

2 == 2

first → [3]

second → null (loops end (second == null)).

isPal = true.

Restore original list.

slow → next = reverse list (second) :

[2] → [1] → null.

[1] → [2] → [3] → [2] → [1] → null.

return isPal ;

Output : Yes.

Final output is :

[1, 2, 3, 42, 1]

isPalindrome : Yes.

curr -> next = list 2;

list 2 = list 2 -> next;

curr = curr -> next;

Merged: [1] -> [2]

curr -> [2]

list 1 -> [3] -> [5] -> null

list 2 -> [4] -> [6] -> null

3 < 4

curr -> next = list 1;

list 1 = list 1 -> next;

curr = curr -> next;

Merged: [1] -> [2] -> [3]

curr -> [3]

list 1 -> [5] -> null

list 2 -> [4] -> [6] -> null

5 < 4

curr -> next = list 2;

list 2 = list 2 -> next;

curr = curr -> next;

5- Linked list:

list 1 : 1 -> 3 -> 5 -> null

list 2: 2 -> 4 -> 6 -> null.

list 1 : [1] -> [3] -> [5] -> null.

list 2: [2] -> [4] -> [6] -> null.

head = ?

if (!list1) return list2;

if (!list2) return list1;

if (list1 -> data < list2 -> data)

1 < 2

head = list1;

list1 = list1 -> next;

head -> [1]

list1 -> [3] -> [5] -> null

list2 -> [2] -> [4] -> [6] -> null.

Node * curr = head;

curr -> [1]

while (list1 && list2)

list1 -> data = 3

list2 -> data = 2

3 < 2.

Merged : [1] → [2] → [3] → [4]

curr → [4]

list1 → [5] → null

list2 → [6] → null

5 < 6

curr → next = list1;

list1 = list1 → next;

curr = curr → next;

Merged : [1] → [2] → [3] → [4] → [5]

curr → [5]

list1 → null

list2 → [6] → null

list1 == null

if (list1) curr → next = list1;

if (list2) curr → next = list2;

curr → next = [6]

head

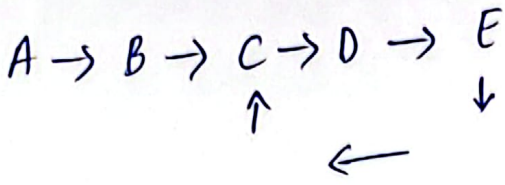
↓

[1] → [2] → [3] → [4] → [5] → [6] → null

Merged list :

1 → 2 → 3 → 4 → 5 → 6.

6- Floyd's Cycle Algorithm:



Head = A

Node *slow = head;

Node *fast = head;

slow → A

fast → A

while (fast && fast → next)

slow = slow → next

fast = fast → next → next

slow → B

fast → C

slow → C

fast → E

slow → D

fast → D

Meeting point = D.

slow = head;

slow → A

fast → D

while (slow != fast)

slow = slow → next

fast = fast → next

slow → C

fast → C

return slow;

cycle start node = C.

cycle detected

loop starts at node : C.

2(slow distance) = fast distance

$2(x+y) = x+y+n-K$

$x = n'K - y$.

7- Remove Consecutive Absents (K = 3)

dummy = node (-1)

dummy → next = head (points to node with value 1)

prev = dummy.

head points to first node (value 1).

dummy (-1) → 1 → 0 → 0 → 0 → 1 → 0 → 1

↑

prev.

curr = prev → next = node (1)

curr → data = 1 (not 0), so skip the if block.

move to else : prev = prev → next = node (1)

dummy (-1) → 1 → 0 → 0 → 0 → 1 → 0 → 1

↑

prev

curr = prev → next = node (0)

curr → data == 0

count = 0

temp = curr = node (0)

temp = node (0, first) ; count = 1 , temp = temp → next = node (0)

temp = node (0, second) = count = 2, temp = temp → next = (second)

temp = node (0, third) = count = 3 , temp = temp → next = node (0, third)
= node (1)

temp = node (1) : temp → data != 0, exit loop.

count = 3 which is >= K (3).

→ Delete Sequence:

prev → next = temp = node (1) (skip the 3 zeroes)

toDel = curr = node (0, first)

• Delete loop runs 3 times:

→ next = toDel → next = node (0, second) , delete node (0, first),

(toDel = node (0, second))

→ next = toDel → next = node (0, third) , delete node (0, second),

toDel = node (0, third)

→ next = toDel → next = node (1) , delete node (0, third) , toDel

= node (1).

dummy (-1) → 1 → 1 → 0 → 1

↑

prev

curr = prev → next = node (0)

curr → data = = 0

count = 0

temp = curr = node (0)

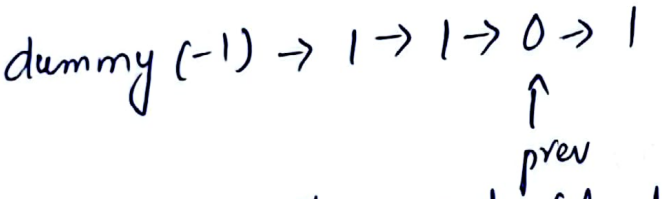
temp = node (0): count = 1, temp = temp → next = node (1)

temp = node (1): temp → data != 0, exit loop.

count = 1 which is < K (3)

Skip deletion, go to else.

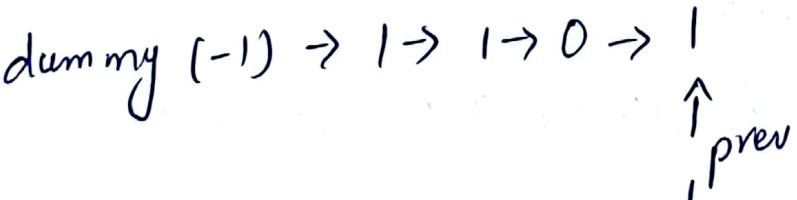
prev = prev → next = node (0)



curr = prev → next = node (1, last)

curr → data = 1 (not 0)

prev = prev → next = node (1, last)



prev → next = null, loops end

head = dummy → next = node (1, first)

delete dummy

return head

list: 1 → 1 → 0 → 1

Time Complexity: O(n)

Space Complexity: O(1)

8- Deleting node:

We have a linked list: 1 → 2 → 3 → 4 → 5 → null

node pointer = pointing to node (3)

1 → 2 → 3 → 4 → 5 → null

↑

node

node points to node (3)

no, node is not null.

node → next points to node (4)

no, node is not null.

1 → 2 → 3 → 4 → 5 → null

↑ ↑

node node → next

We create a temporary pointer called next that points to the same nodes as node → next.

next = node → next

next → now points to node (4).

node → still points to node (3).

1 → 2 → 3 → 4 → 5 → null

↑ ↑

node next

data = 3 data = 4

Copy data from next node to current node.

node → data = next → data.

Before:

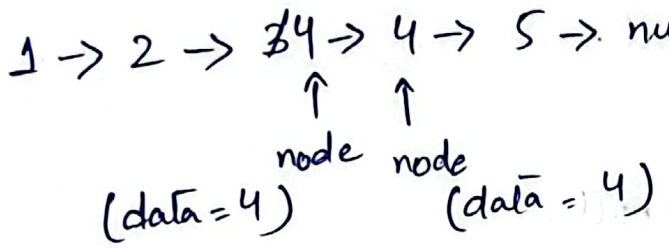
node (3) has data 3.

Node (4) has data 4.

After:

node (3) has data 4.

node (4) still has data 4.

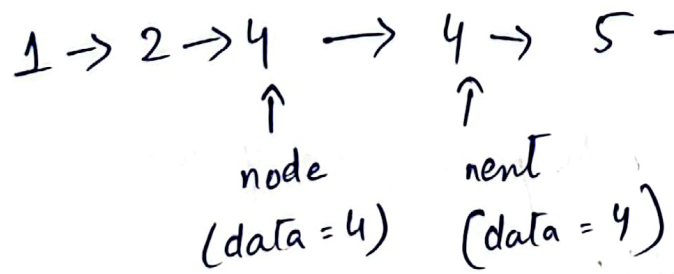


→ We update the current node's next pointer to skip over the next node.

node → next = next → next.

next → next points to node (5).

node → next now points to node (5) (skipping node (4)).



→ Delete the bypass node:

We free the memory of the node that next points too.

delete next.

The original node (4) is removed from memory.

The memory is free.

1 → 2 → 3 → 4 → 5 → null.

node
(data = 4)

→ Return Success:

The function return success (true) indicating successful deletion.

Original list:

1 → 2 → 3 → 4 → 5 → null.

Final list:

1 → 2 → 4 → 5 → null.

9. Queue Implementation:

front = null;

rear = null;

Queue is empty.

void Insert / Front (int val)

val = 10.

Insert a new node with value 10 at the front of the queues.

Node *newNode = new Node (val);

newNode → [10 | null]

if (rear == null)

rear = null;

front = null.

rear = null.

Queue is empty.

front = newNode;

rear = newNode;

front now points to newNode.

rear now points to newNode.

Both, front and rear points to the same node.

front → [10 | null] ← rear.

front points to node (10)

rear points to node (10)

queue : 10.

void Insert | front (int val) → val = 20

Node * newNode = new Node (val);

newNode → data = 20

newNode → next = null

front → [10 | null] ← rear

new node → [20 | null]

if (rear == null)

rear points to node (10)

newNode → next = front;

newNode → [20 | •] → [10 | null]

↑ front, rear.

front = newNode;

front → [20 | ·] → [10 | null] ← rear

front points to node (20)

rear points to node (10)

queue : 20 → 10.

void InsertRear (int val)

val = 30

Node * newNode = newNode (val);

newNode → data = 30

newNode → next = null

front → [20 | ·] → [10 | null] ← rear

newNode → [30 | null]

if (front == null)

front points to node (20)

rear → next = newNode;

front → [20 | ·] → [10 | ·] → [30 | null]

rear

newNode

rear = newNode;

front → [20 | ·] → [10 | ·] → [30 | null] ← rear

Node *temp = front;

while (temp -> next != rear)

front -> [10 | o] -> [30 | null] <- rear

temp

front -> [10 | o] -> deleted

rear, temp

front -> [10 | null] <- rear

return data;

10 - Merge Sort for Linked List:

merge Sort -> 5 -> 2 -> 8 -> 1 -> 9 -> 3.

head = node (5)

head -> next exists

sorted = true

curr = head = node (5)

curr = node (5): 5 > 2 Yes -> sorted, false = break

mid = node (8)

left = head = node (5)

right = mid -> next = node (1)

mid -> next = null.

left half: 5 -> 2 -> 8.

int Remove / front ()

if (front == null)

Node * temp = front;

int data = front -> data;

front -> [20| ·] -> [10| ·] -> [30| null] <- rear

↑

Temp

front = front -> next;

temp -> [20| ·] -> [10 | ·] -> [30| null] <- rear

↑

front

if (front == null)

delete temp;

front -> [10 | ·] -> [30| null] <- rear

return data;

-> Remove Rear ();

int Remove / Rear ()

Remove and return the element from the rear of the queue.

if (front == null)

if (front == rear)

left : 9 , right : 3.

11- Recursive Sort Stack:

Initial Stack (Top to Bottom): [-3, 14, 18, -5, 30]

Popping all Elements

Sort - Stack → Pop → 3 → Recurse for remaining

[14, 18, -5, 30]

Sort - Stack → Pop 14 → Recurse for remaining

[18, -5, 30]

Sort - Stack → Pop 18 → Recurse for remaing

[-5, 30]

Sort - Stack → Pop -5 → Recurse for remaing.

[30]

Sort - Stack → Pop 30 → Recurse for remaining

[ ] Empty

Inserting back in sorted arrays / stack:

Sorted Insert → Push 30 → [30]

Sorted Insert → Push -5 → [-5, 30]

Sorted Insert → Push 18 → [-5, 18, 30]

Sorted Insert → Pop -5, push 14 → [-5, 14, 18, 30]

Sorted Insert → pop -5, push -3, push → [-5, -3, 14, 18, 30]

-5 back

Right half : 1 → 9 → 3.

left = mergeSort (5 → 2 → 8)

right = mergeSort (1 → 9 → 3)

5 > 2 → Yes (Sorted) = false

Returns node (2)

Left (5)

Right : 2 → 8

Base case : Returns node (5)

2 → 8 → no → sorted remains true.

Compare : 2 < 5 → take 2

Compare : 5 < 8 → take 5

Take remaining : 8

Result : 2 → 5 → 8.

1 > 9 → No

9 > 3 → Yes → sorted = false

Returns node (9)

Left : 1

Right : 9 → 3

Base case : returns node (1)

9 > 3 → Yes → sorted = false

returns node (9)

Final output is:

-6
-3
14
18
30

12 - Balanced Bracket Checker:

Test Expression : { [( ) < >] }, length = 8

{ → push ('{') → opening bracket found.

[→ push ('[') → opening bracket found.

(→ push ('(') → opening bracket found.

) → pop ( ) → Top is matching → found correct!

< → push ('<') → opening bracket found.

> → pop ( ) → Top is match → correct!

] → pop ( ) → Top is match → found correct!

} → pop ( ) → Top is found / match → correct.

Final check: Stack is empty and no errors were found.

Result: Balanced.

13 - Stack using 2 Queues (Push : 5, 7, 1, 3)

q1 = [], q2 = []

Push (5) → 5 is added to empty q2 → [5]

Push (7) → 7 is added to empty q2 → [7, 5]

push (1) → is added to q2 → [1, 7, 5]
push (3) → is added to q2 → [3, 1, 7, 5]
display () → q1 is front is 3 → [3, 1, 7, 5]
pop() → removes the front of q1 → [1, 7, 5]
Display → new front is 1 → [1, 7, 5]

14. Train Bogie Sorting:

Original queue : [3 → 1 → 4 → 2] (front to rear)
Initial States : S1 = empty ; q2 = empty..

current = 3 → q1. dequeue () returns 3 → [3]
current = 1 → q1. dequeue () returns 1 → [3, 1]
current = 4 → q1. dequeue () returns 4 → [4, 3, 1]

Pop 1 and 3 from s1 and move them q2 .[] (Q2 : 1 → 3)
Push 4 to s1 → [4]
move everything from q2 back to s1 → [4, 3, 1].

current = 2 → q1. dequeue () returns 2.

Pop 1 from s1 and move to q2 → [4, 3] (Q2 : 1)
Push 2 to s1 → [4, 3, 2]
move everything from q2 back to s1 → [4, 3, 2, 1]
move all elements from s1 back to q1 → [1, 2, 3, 4]

Original Bogie Order : [3, 1, 4, 2]
Sorted Bogie Order : [1, 2, 3, 4].

15- Min Stack with O(1) minimum retrieval:

Initial state = topnode - null, min val = LLONG- MAX

push(10) → first element → min val becomes 10 → top: 10, min:10

push (5) → new min (5 < 10) → top: 5, min: 5.

push (15) → not a new (15 > 5) → top: 15, min 5.

push (2) → new min (2 < 5) → top: 2, min 2.

pop ( ) → top data (-1) is < min val (2) → top: 15, min 5.

pop ( ) → top data (15) is > min val (5) → top: 5, min: 5.

pop ( ) → no data → stack empty.

16- Removing Duplicates:

list: [10 → 20 → 130 → 30 → 20]

Seen: { } (Empty (seen)),

prev= null, current= head (10)

10 → 10 is not in seen. add 10 to set → {10} →

[10, 20, 10, 30, 20]

20 → 20 is not in seen → add 20 to set {10, 20} →

[10 → 20 → 10 → 30 → 20]

10 → duplicate found → {10, 20} → [10 → 20 → 30 → 20]

30 → 30 is not in seen → add 30 to set → {10, 20, 30

[10, 20, 30, 20]

20 → duplicate found → {10, 20, 30} → [10 → 20 → 30]

Original list : 10 → 20 → 10 → 30 → 20 → null

list after removing duplicates : 10 → 20 → 30 → null.

17 - Find nth to last element :

list : [10 → 20 → 30 → 40 → 50] , n = 2

Initial state : slow = 10, fast = 10.

Gap creation → move fast pointer n(2) steps forward → slow at 10, fast at 30.

Simultaneous move → move both pointers 1 move → slow at 20, fast at 40.

Simultaneous move → move both pointers 1 step, fast becomes null → slow at 40, fast at null.

Result → fast is null, slow is now pointing at the result and the result is 40.

Final output is :

list : 10 → 20 → 30 → 40 → 50

2nd from last element : 40.

18 - Delete node with only pointer access :

list : [a → b → c → d → e]

Target node to delete : c.

Access target → pointer is directly on node c → (a → b → (c → d → e).

Copy data → node → data = nextnode → data → (a → b → [d] → d → e)

Relink pointer → node → next = nextnode → next = (a → b → [d] → e)

Cleanup → delete nextnode → (a → b → d → e).

Final output is: (a → b → d → e).

19- Add two numbers (reversed order)

list 1 → (3 → 1 → 9 → 9) (Represents 9913)

list 2 → (5 → 9 → 2) (Represents 295)

Carry = 0, dummynode = 0.

3 + 5 + 0 → sum = 8, node data = 8 % 10 → carry = 0 → 8

1 + 9 + 0 → sum = 10, node data = 10 % 10 → carry = 1 → 8 → 0

9 + 2 + 1 → sum = 12, node data = 12 % 10 → carry = 1 → (8 → 0 → 2)

9 + (null) + 1 → sum = 10, node data = 10 % 10 → carry 1 → (8 → 0 → 2 → 0)

(null) + (null) + 1 → only carry left, node data → carry 0 → = 1 (8 → 0 → 2 → 0 → 1)

Final output is:

(8 → 0 → 2 → 0 → 1) (Represents 10208)

20- Detect loop start (Floyd's Algorithm):

Input: A → B → C → D → E → (back to C)

slow = A, fast = A.

move pointers → slow moves 1 step, moves fast 2 steps → slow B, fast C.

move pointers → slow moves 1 step, fast moves 2 steps → slow C, fast E.

move pointers → slow moves 1 step, fast moves 2 steps → slow D, fast D.

Collision → slow == fast → Cycle found.