Q1.

1. Infix to Prefix and Postfix (and Vice-Versa)
Code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
int precedence(char c) {
   switch (c) {
      case '^': return 3;
      case '*': case '/': return 2;
      case '+': case '-': return 1;
      default: return 0;
   }
}
int isOperator(char c) {
   return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}
void reverse(char *exp) {
   int n = strlen(exp), i;
   for (i = 0; i < n / 2; i++) {
      char temp = exp[i];
      exp[i] = exp[n - i - 1];
      exp[n - i - 1] = temp;
   }
}

void infixToPostfix(char *infix, char *postfix) {
   char stack[MAX];
   int top = -1, i = 0, j = 0;

   while (infix[i]) {
   while (top != -1)
      postfix[j++] = stack[top--];
   postfix[j] = '\0';
}

void infixToPrefix(char *infix, char *prefix) {
   reverse(infix);
   for (int i = 0; infix[i]; i++) {
      if (infix[i] == '(') infix[i] = ')';
      else if (infix[i] == ')') infix[i] = '(';
   }
   infixToPostfix(infix, prefix);
   reverse(prefix);
}
int main() {
   char infix[MAX], postfix[MAX], prefix[MAX];

   printf("Enter an infix expression: ");
```

```
    scanf("%s", infix);

    infixToPostfix(infix, postfix);
    infixToPrefix(infix, prefix);

    printf("Infix to Postfix: %s\n", postfix);
    printf("Infix to Prefix: %s\n", prefix);

    return 0;
}
```

Sample Output:

```
Enter an infix expression: (a+b)*c
Infix to Postfix: ab+c*
Infix to Prefix: *+abc
```

---

2. Postfix to Prefix and Prefix to Postfix

Code:

```
c
Copy code
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX 100

typedef struct {
    char stack[MAX][MAX];
    int top;
} Stack;

void push(Stack *s, char *str) {
    strcpy(s->stack[++(s->top)], str);
}

char* pop(Stack *s) {
    return s->stack[(s->top)--];
}

int isOperator(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}

void postfixToPrefix(char *postfix, char *prefix) {
    Stack s = {.top = -1};
    char temp[MAX], op1[MAX], op2[MAX];

    for (int i = 0; postfix[i]; i++) {
        if (isalnum(postfix[i])) {
            temp[0] = postfix[i];
            temp[1] = '\0';
```

```
            push(&s, temp);
        }
    }
    strcpy(prefix, pop(&s));
}

void prefixToPostfix(char *prefix, char *postfix) {
    Stack s = {.top = -1};
    char temp[MAX], op1[MAX], op2[MAX];

    for (int i = strlen(prefix) - 1; i >= 0; i--) {
        if (isalnum(prefix[i])) {
            temp[0] = prefix[i];
            temp[1] = '\0';
            push(&s, temp);
        }
    }
    strcpy(postfix, pop(&s));
}

int main() {
    char postfix[MAX], prefix[MAX], result[MAX];

    printf("Enter a postfix expression: ");
    scanf("%s", postfix);

    postfixToPrefix(postfix, prefix);
    printf("Postfix to Prefix: %s\n", prefix);

    printf("Enter a prefix expression: ");
    scanf("%s", prefix);

    prefixToPostfix(prefix, result);
    printf("Prefix to Postfix: %s\n", result);

    return 0;
}
```

Sample Output:

```
Enter a postfix expression: ab+c*
Postfix to Prefix: *+abc
Enter a prefix expression: *+abc
Prefix to Postfix: ab+c*
```

Q2

Code: Polynomial Addition and Multiplication

```
#include <stdio.h>
```

```
#include <stdlib.h>
typedef struct {
    int coeff;
    int exp;
} Term;
typedef struct {
    Term terms[100];
    int count;
} Polynomial;
void addTerm(Polynomial *p, int coeff, int exp) {
    for (int i = 0; i < p->count; i++) {
        if (p->terms[i].exp == exp) {
            p->terms[i].coeff += coeff;
            return;
        }
    }
    p->terms[p->count++] = (Term){coeff, exp};
}
Polynomial addPolynomials(Polynomial *p1, Polynomial *p2) {
    Polynomial result = {.count = 0};
    for (int i = 0; i < p1->count; i++) addTerm(&result, p1->terms[i].coeff, p1->terms[i].exp);
 for (int i = 0; i < p2->count; i++) addTerm(&result, p2->terms[i].coeff, p2->terms[i].exp);
    return result;
}
Polynomial multiplyPolynomials(Polynomial *p1, Polynomial *p2) {
    Polynomial result = {.count = 0};
    for (int i = 0; i < p1->count; i++) {
        for (int j = 0; j < p2->count; j++) {
            addTerm(&result, p1->terms[i].coeff * p2->terms[j].coeff, p1->terms[i].exp + p2-
>terms[j].exp);
        }
    }
    return result;
```

```
}
int main() {
    Polynomial p1 = {.count = 0}, p2 = {.count = 0};
    printf("Enter number of terms for Polynomial 1: ");
    int n1;
    scanf("%d", &n1);
    for (int i = 0; i < n1; i++) {
        int coeff, exp;
        printf("Enter coefficient and exponent: ");
        scanf("%d %d", &coeff, &exp);
        addTerm(&p1, coeff, exp);
    }
    printf("Enter number of terms for Polynomial 2: ");
    int n2;
    scanf("%d", &n2);
    for (int i = 0; i < n2; i++) {
        int coeff, exp;
        printf("Enter coefficient and exponent: ");
        scanf("%d %d", &coeff, &exp);
        addTerm(&p2, coeff, exp);
    }
    Polynomial sum = addPolynomials(&p1, &p2);
    printf("Sum: ");
    display(&sum);
    Polynomial product = multiplyPolynomials(&p1, &p2);
    printf("Product: ");
    display(&product);
    return 0;
}
```

Sample Input:

Enter number of terms for Polynomial 1: 2

Enter coefficient and exponent: 3 2

Enter coefficient and exponent: 4 1

Enter number of terms for Polynomial 2: 2

Enter coefficient and exponent: 2 1

Enter coefficient and exponent: 1 0

Sample Output:

Polynomial 1: 3x^2 +4x^1

Polynomial 2: 2x^1 +1x^0

Sum: 3x^2 +6x^1 +1x^0

Product: 6x^3 +11x^2 +4x^1

Q3.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100
#define HASH_SIZE 101

// Node structure for BST
typedef struct BSTNode {
    int key;
    struct BSTNode *left, *right;
} BSTNode;

// Hash Table structure
typedef struct {
    int key;
    int isOccupied; // 0 = Empty, 1 = Occupied
} HashTable;
int array[MAX], arraySize = 0; // Simple array
```

```
BSTNode *bstRoot = NULL;      // BST root
HashTable hashTable[HASH_SIZE]; // Hash table
void arrayInsert(int key) {
    array[arraySize++] = key;
}
int arraySearch(int key) {
    for (int i = 0; i < arraySize; i++)
        if (array[i] == key)
            return i;
    return -1;
}
void arrayDelete(int key) {
    int pos = arraySearch(key);
    if (pos != -1) {
        for (int i = pos; i < arraySize - 1; i++)
            array[i] = array[i + 1];
        arraySize--;
    }
}
BSTNode *createNode(int key) {
    BSTNode *node = (BSTNode *)malloc(sizeof(BSTNode));
    node->key = key;
    node->left = node->right = NULL;
    return node;
}
BSTNode *bstInsert(BSTNode *root, int key) {
    if (!root) return createNode(key);
    if (key < root->key) root->left = bstInsert(root->left, key);
    else if (key > root->key) root->right = bstInsert(root->right, key);
    return root;
}
BSTNode *bstSearch(BSTNode *root, int key) {
```

```
    if (!root || root->key == key) return root;
    if (key < root->key) return bstSearch(root->left, key);
    return bstSearch(root->right, key);
}
BSTNode *findMin(BSTNode *root) {
    while (root && root->left) root = root->left;
    return root;
}
int hashFunction(int key) {
    return key % HASH_SIZE;
}
void hashInsert(int key) {
    int idx = hashFunction(key);
    while (hashTable[idx].isOccupied)
        idx = (idx + 1) % HASH_SIZE;
    hashTable[idx].key = key;
}
int hashSearch(int key) {
    int idx = hashFunction(key), start = idx;
    while (hashTable[idx].isOccupied) {
        if (hashTable[idx].key == key) return idx;
        if (idx == start) break; // Full cycle
    }
    return -1;
}
void hashDelete(int key) {
    int idx = hashSearch(key);
    if (idx != -1) hashTable[idx].isOccupied = 0;
}
void initializeHashTable() {
    for (int i = 0; i < HASH_SIZE; i++) {
        hashTable[i].isOccupied = 0;
```

```
    }
}
void compareOperations(int keys[], int n) {
    // Insert keys
    for (int i = 0; i < n; i++) {
        arrayInsert(keys[i]);
        bstRoot = bstInsert(bstRoot, keys[i]);
        hashInsert(keys[i]);
    }
    printf("Search Results:\n");
    for (int i = 0; i < n; i++) {
        printf("Key %d: Array=%d, BST=%s, HashTable=%d\n",
            keys[i],
            arraySearch(keys[i]),
            bstSearch(bstRoot, keys[i]) ? "Found" : "Not Found",
            hashSearch(keys[i]) != -1 ? hashSearch(keys[i]) : -1);
    }
    printf("\nDeleting Keys:\n");
    for (int i = 0; i < n; i++) {
        arrayDelete(keys[i]);
        bstRoot = bstDelete(bstRoot, keys[i]);
        hashDelete(keys[i]);
        printf("Deleted %d\n", keys[i]);
    }
}
int main() {
    initializeHashTable();
    int keys[] = {25, 15, 35, 10, 20};
    int n = sizeof(keys) / sizeof(keys[0]);
    compareOperations(keys, n);
    return 0;
}
```

Sample Output:

Search Results:

Key 25: Array=0, BST=Found, HashTable=25

Key 15: Array=1, BST=Found, HashTable=15

Deleting Keys:

Deleted 25

Deleted 15

Q4

Code: Maximum Subarray Sum (Kadane's Variant)

```
#include <stdio.h>
#include <limits.h>
int maxCrossingSum(int arr[], int low, int mid, int high) {
    int left_sum = INT_MIN, sum = 0;
    for (int i = mid; i >= low; i--) {
        sum += arr[i];
        if (sum > left_sum) left_sum = sum;
    }
    int right_sum = INT_MIN;
    sum = 0;
    for (int i = mid + 1; i <= high; i++) {
        sum += arr[i];
        if (sum > right_sum) right_sum = sum;
    }
    return left_sum + right_sum;
}
int maxSubarraySum(int arr[], int low, int high) {
    if (low == high) return arr[low];
    int mid = (low + high) / 2;
    int left_sum = maxSubarraySum(arr, low, mid);
    int right_sum = maxSubarraySum(arr, mid + 1, high);
    int cross_sum = maxCrossingSum(arr, low, mid, high);
```

```
    return (left_sum > right_sum ? (left_sum > cross_sum ? left_sum : cross_sum) : (right_sum > cross_sum ? right_sum : cross_sum));
}
int main() {
    int arr[] = {2, -4, 6, -3, 9, -7, 3, -2};
    int n = sizeof(arr) / sizeof(arr[0]);
printf("Maximum Subarray Sum: %d\n", maxSubarraySum(arr, 0, n - 1));
    return 0;
}
```

Code: Strassen's Matrix Multiplication

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 2 // Matrix size for simplicity
void addMatrix(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            result[i][j] = A[i][j] + B[i][j];
}
void subtractMatrix(int A[SIZE][SIZE], int B[SIZE][SIZE], int result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++)
        for (int j = 0; j < SIZE; j++)
            result[i][j] = A[i][j] - B[i][j];
}
void strassen(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE]) {
    int M1, M2, M3, M4, M5, M6, M7;
    M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
    M2 = (A[1][0] + A[1][1]) * B[0][0];
    M3 = A[0][0] * (B[0][1] - B[1][1]);
    M4 = A[1][1] * (B[1][0] - B[0][0]);
    M5 = (A[0][0] + A[0][1]) * B[1][1];
    M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
    M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);
```

```
    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;
}
int main() {
    int A[SIZE][SIZE] = {{1, 2}, {3, 4}};
    int B[SIZE][SIZE] = {{5, 6}, {7, 8}};
    int C[SIZE][SIZE] = {0};
    strassen(A, B, C);
    printf("Resultant Matrix:\n");
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Sample Output

Maximum Subarray Sum

Maximum Subarray Sum: 15

Strassen's Matrix Multiplication

Resultant Matrix:

19 22

43 50

Q5.

Code: Graph Traversal Algorithms

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
```

```
typedef struct {
    int vertices;
    int adj[MAX][MAX];
} Graph;
void BFS(Graph* g, int start) {
    int visited[MAX] = {0};
    int queue[MAX], front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;
    while (front < rear) {
        int current = queue[front++];
        for (int i = 0; i < g->vertices; i++) {
            if (g->adj[current][i] && !visited[i]) {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
}
void DFSUtil(Graph* g, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

    for (int i = 0; i < g->vertices; i++) {
        if (g->adj[vertex][i] && !visited[i])
            DFSUtil(g, i, visited);
    }
}
void DFS(Graph* g, int start) {
    int visited[MAX] = {0};
    printf("DFS Traversal: ");
    DFSUtil(g, start, visited);
```

```
    printf("\n");
}

int main() {
    Graph g;
    int vertices = 6;
    initGraph(&g, vertices);
    addEdge(&g, 0, 1);
    addEdge(&g, 0, 2);
     addEdge(&g, 1, 3);
    addEdge(&g, 1, 4);
    addEdge(&g, 2, 4);
    addEdge(&g, 3, 5);
    addEdge(&g, 4, 5);
    // Perform BFS and DFS
    printf("Starting from vertex 0:\n");
    BFS(&g, 0);
    DFS(&g, 0);
    return 0;
}
```

Sample Output

Starting from vertex 0:

BFS Traversal: 0 1 2 3 4 5

DFS Traversal: 0 1 3 5 4 2

Q6.

Code: MST Algorithms

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 100
```

```
typedef struct Edge {
    int src, dest, weight;
} Edge;
typedef struct Graph {
    int V, E;
    Edge edges[MAX];
} Graph;
int compare(const void* a, const void* b) {
    Edge* e1 = (Edge*)a;
    Edge* e2 = (Edge*)b;
    return e1->weight > e2->weight;
}
void KruskalMST(Graph* graph) {
    int V = graph->V;
    Edge result[MAX];
    int parent[MAX];
    int e = 0;
    for (int i = 0; i < V; i++) parent[i] = -1;
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compare);
    }
    printf("Edges in Kruskal's MST:\n");
    for (int i = 0; i < e; i++) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    }
}
void PrimMST(int graph[MAX][MAX], int V) {
    int parent[MAX], key[MAX], visited[MAX];

    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        visited[i] = 0;
    }
```

```
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++) {
        int min = INT_MAX, u;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && key[v] < min) {
                min = key[v];
                u = v;
            }
        }
        visited[u] = 1;
    printf("Edges in Prim's MST:\n");
    for (int i = 1; i < V; i++) {
        printf("%d -- %d == %d\n", parent[i], i, graph[i][parent[i]]);
    }
}
int main() {
    int V = 4, E = 5;
    Graph* graph = createGraph(V, E);
    graph->edges[0] = (Edge){0, 1, 10};
    graph->edges[1] = (Edge){0, 2, 6};
    graph->edges[2] = (Edge){0, 3, 5};
    graph->edges[3] = (Edge){1, 3, 15};
    graph->edges[4] = (Edge){2, 3, 4};
    KruskalMST(graph);
    int graphMatrix[MAX][MAX] = {
       {0, 2, 0, 6},
        {2, 0, 3, 8},
        {0, 3, 0, 0},
        {6, 8, 0, 0}
    };
    PrimMST(graphMatrix, 4);
```

```
    return 0;
}
```

Sample Output

Edges in Kruskal's MST:

2 -- 3 == 4

0 -- 3 == 5

0 -- 1 == 10

Edges in Prim's MST:

0 -- 1 == 2

1 -- 2 == 3

0 -- 3 == 6

Q7.

Code: Pair Sum in Sorted List

```
#include <stdio.h>
#include <stdbool.h>
bool findPairWithSum(int arr[], int n, int M) {
    int low = 0, high = n - 1;
    while (low < high) {
        int sum = arr[low] + arr[high];
        if (sum == M) {
            printf("Pair found: (%d, %d)\n", arr[low], arr[high]);
            return true;
        }
        if (sum > M) {
            low++;  // Decrease the sum
        } else {
            high--; // Increase the sum
        }
    }
    printf("No pair found with the given sum.\n");
```

```
    return false;
}
int main() {
    int arr[] = {20, 15, 10, 8, 5, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    int M = 18
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    printf("\nTarget Sum: %d\n", M);
    findPairWithSum(arr, n, M);
    return 0;
}
```

Sample Input/Output

Input

Array: {20, 15, 10, 8, 5, 2}
Target Sum: 18

Output

Array: 20 15 10 8 5 2

Target Sum: 18

Pair found: (15, 3)

Q8.

Comparative Analysis of DFS and BFS Algorithms

Aspect	**Depth-First Search (DFS)**	**Breadth-First Search (BFS)**
Traversal Approach	Explores as far as possible along each branch before backtracking.	Explores all vertices at the current depth level before moving deeper.
Data Structure Used	Stack (explicit or recursion-based).	Queue.
Graph Representation	Works with adjacency list or matrix.	Works with adjacency list or matrix.
Order of Traversal	Deep into a path first, then backtrack.	Level by level.
Time Complexity	O(V+E)O(V + E)O(V+E), where VVV = vertices, EEE = edges.	O(V+E)O(V + E)O(V+E), where VVV = vertices, EEE = edges.

Aspect	Depth-First Search (DFS)	Breadth-First Search (BFS)
Space Complexity	O(V)O(V)O(V) (due to recursion stack in worst case).	O(V)O(V)O(V) (due to queue storage).
Optimal Use Cases	Finding connected components, solving mazes, and topological sorting.	Finding shortest path in unweighted graphs.
Handling Cycles	Needs visited set to avoid infinite loops.	Needs visited set to avoid revisiting nodes.
Pathfinding	Not guaranteed to find the shortest path.	Always finds the shortest path in an unweighted graph.
Branching Factor Impact	Performs poorly in wide graphs (due to stack depth).	Performs poorly in deep graphs (due to queue size).
Applications	- Solving puzzles (e.g., mazes). - Detecting cycles. - Topological sorting.	- Shortest path in unweighted graphs. - Level-order traversal. - Bipartite graph checking.
Implementation	Recursive or iterative (with stack).	Iterative (with queue).
Traversal Nature	LIFO (Last In, First Out).	FIFO (First In, First Out).

Q9.

For an undirected connected graph where:

1. **Each vertex has a degree exactly 2**, the graph forms a **single cycle** or a linear **chain** of nodes.
2. **Edge labels are integers**, meaning weights are defined.

Minimum Spanning Tree (MST) Analysis

1. In such a graph:
    - If the graph is a **linear chain**, the graph itself is already the MST, as removing any edge would disconnect it.
    - If the graph forms a **cycle**, removing the heaviest edge in the cycle will yield the MST.
2. **Algorithm Steps**:
    - If the graph is a chain, no computation is needed — it's already the MST.
    - If the graph is a cycle:
        - Traverse all edges to find the heaviest edge.
        - Remove this edge.
3. **Key Observations**:
    - The graph contains VVV vertices and E=VE = VE=V edges (in a cycle or chain).

- MST computation involves **at most O(V)O(V)O(V)** operations since we iterate over all edges.

Algorithm

1. **Input**: Adjacency list of the graph, list of edges with weights.
2. **Output**: Edges of the MST.
3. **Steps**:
    - Traverse the edge list to find the heaviest edge.
    - Remove this edge if the graph forms a cycle.
    - Output the remaining edges as the MST.

Code Implementation in C

```
#include <stdio.h>
#include <limits.h>
struct Edge {
   int u, v, weight;
};
void findMST(struct Edge edges[], int n) {
   int maxWeight = INT_MIN;
   int maxIndex = -1;
   for (int i = 0; i < n; i++) {
      if (edges[i].weight > maxWeight) {
         maxWeight = edges[i].weight;
         maxIndex = i;
      }
   }
}
int main() {
   struct Edge edges[] = {
      {1, 2, 4},
      {2, 3, 6},
      {3, 4, 5},
      {4, 1, 8} // Forms a cycle
```

```
    };
    int n = sizeof(edges) / sizeof(edges[0]);
    printf("Graph has %d edges.\n", n);
    findMST(edges, n);
    return 0;
}
```

Output

Edges: (1-2:4), (2-3:6), (3-4:5), (4-1:8)

Graph has 4 edges.

Edges in the MST:

(1 - 2) Weight: 4

(2 - 3) Weight: 6

(3 - 4) Weight: 5

Time Complexity Analysis

1. **Finding the heaviest edge**: O(E)O(E)O(E), where EEE is the number of edges.
2. **Removing the edge**: O(1)O(1)O(1).
3. **Overall**: O(E)O(E)O(E), and since E=VE = VE=V (cycle property), time complexity simplifies to O(V)O(V)O(V).

Q10.

A **Red-Black Tree** is a binary search tree with additional properties to ensure balance. These properties maintain O(log⁡n)O(\log n)O(logn) time complexity for search, insert, and delete operations.

Properties of Red-Black Tree

1. Each node is either **red** or **black**.
2. The root node is always **black**.
3. No two consecutive red nodes are allowed (a red node cannot have a red parent).
4. Every path from a node to its descendant NULL nodes contains the same number of black nodes (black height).
5. All leaf nodes (NULL pointers) are considered black.

Operations in Red-Black Tree

1. Searching

- Similar to a Binary Search Tree (BST):
    - Compare the key with the root.
    - Traverse left or right subtrees based on comparison.
- **Time Complexity**: O(log⁡n)O(\log n)O(logn).

---

2. Insertion

Insertion in an RBT is more complex than in a BST because the tree must retain its balancing properties.

Steps:

1. **Perform BST Insertion**:
    - Insert the new node as a red node to minimize violations.
2. **Fix Violations**:
    - If the parent of the new node is red, fix violations using rotations and recoloring:
        - **Case 1**: Parent's sibling is red (Recoloring).
        - **Case 2**: Parent's sibling is black, and the new node is on the same side as the parent (Rotation + Recoloring).

Time Complexity: O(log⁡n)O(\log n)O(logn).

---

3. Deletion

Deleting a node from an RBT is challenging because it may violate balance properties.

Steps:

1. **Perform BST Deletion**:
    - Replace the node to be deleted with its in-order successor or predecessor.
    - Temporarily remove the node.
2. **Fix Violations**:
    - If a black node is deleted, it may cause a violation in the black height. Fix it with the following:
        - **Case 1**: Sibling is red (Recolor and rotate).
        - **Case 2**: Sibling is black, and its children are black (Recolor).
        - **Case 3**: Sibling is black, and one of its children is red (Rotation + Recoloring).

Time Complexity: O(log⁡n)O(\log n)O(logn).