Contents

SR.NO.	Group	Topic
1	A	Consider a telephone book database with N clients. Implement a hash table to quickly look up a client's telephone number, using two collision handling techniques and compare them based on the number of comparisons required to find a set of telephone numbers. (Telephone Book Database)
2	A	Implement a dictionary using hashing, handling collisions with chaining (with and without replacement). The dictionary should have standard operations like Insert, Find, and Delete, where keys are unique and comparable. (Hash Table Implementation)
3	B	Construct a tree to represent a book with chapters, sections, and subsections. Print the nodes and analyze the time and space requirements of the method. (Tree Construction Method)
4	B	Create a binary search tree from an empty tree by inserting values in a given order. Then, perform operations like inserting a new node, finding the number of nodes in the longest path, finding the minimum data value, swapping left and right pointers, and searching for a value.\ (Binary Search Tree)
5	B	Construct an expression tree from a given prefix expression and traverse it using post order traversal (non-recursive). Then, delete the entire tree. (Expression Tree Construction)
6	C	Represent a graph using adjacency matrix/list to perform Depth-First Search (DFS) and adjacency list to perform Breadth-First Search (BFS). Use a map of the area around a college as the graph, identifying prominent landmarks as nodes. (Graph Representation Algorithm)
7	C	Represent a graph of flight paths between cities, where the cost of an edge is the time or fuel used for the journey. Use adjacency list or matrix representation and check if the graph is connected. Justify the storage representation used. (Flight Path Graph)
8	D	Given a sequence of sorted keys with search probabilities, build a binary search tree with the least search cost. Consider the access probability for each key. (Binary Search Optimization)
9	D	Create a dictionary that stores keywords and their meanings. Provide facilities for adding, deleting, and updating entries, as well as displaying the data in sorted order. Use a height-balanced tree and analyze the complexity of finding a keyword. (Dictionary Management System)
10	E	Read the marks obtained by students in an online examination and find the maximum and minimum marks using a heap data structure. Analyze the algorithm. (Marks Sorting Algorithm)
11	F	Maintain a student information system using a sequential file, allowing users to add, delete, and display information. Handle cases where a student's record does not exist. (Student Information System)
12	F	Maintain an employee information system using an index sequential file, allowing users to add, delete, and display information. Handle cases where an employee's record does not exist. (Employee Data Management)

Group A

Assignment 1

Title: Implementation of a Hash Table for a Telephone Book Database

Problem Statement: Develop a telephone book database for N clients using a hash table to facilitate quick lookups of client telephone numbers. Implement two collision resolution techniques and evaluate their performance based on the number of comparisons needed to retrieve a set of telephone numbers.

Objectives:

- Create a hash table for storing and retrieving telephone numbers.
- Implement two collision resolution techniques (e.g., chaining and open addressing).
- Assess the efficiency of both techniques by comparing the number of comparisons required.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* Python.

Theory Explanation:
A hash table is a data structure that associates keys with values through a hash function. Collisions happen when multiple keys hash to the same index. Two prevalent collision resolution methods are:

1. **Chaining:** Each bucket in the hash table contains a linked list of entries. When a collision occurs, the new entry is added to the list.
2. **Open Addressing:** Collisions are resolved by searching for the next available slot on the hash table.

- ***Time Complexity:***
- Insertion: *O(1)*
- Search: *O(1)*
- Deletion: *O(1)*

- ***Space Complexity:*** *O*(*n*)

Algorithms:

1. **Chaining**

- Resolve collisions by maintaining a linked list at each bucket.
- **Steps:**
  1. Calculate the hash index for the key.
  2. If the bucket is empty, insert the key-value pair.

3. If a collision occurs, add the key-value pair to the linked list.

2. **Open Addressing (Linear Probing)**

- Resolve collisions by probing for the next available slot in the hash table.
- **Steps:**
  1. Calculate the hash index for the key.
  2. If the bucket is empty, insert the key-value pair.
  3. If a collision occurs, probe the next slot until an empty slot is found.

Input/Output Format Examples:

- **Input:** Insert ("Alice", "1234567890"), Insert ("Bob", "9876543210"), Lookup ("Alice")
- **Output:** "1234567890"
- **Input:** Insert ("Charlie", "5555555555"), Lookup ("Bob")
- **Output:** "9876543210"

Code Examples and Syntax Details:

```
class HashTableChaining:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]
    def hash_function(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self.hash_function(key)
        self.table[index].append((key, value))
    def lookup(self, key):
        index = self.hash_function(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None
class HashTableOpenAddressing:
```

```
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
    def hash_function(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self.hash_function(key)
        while self.table[index] is not None:
            index = (index + 1) % self.size
        self.table[index] = (key, value)
    def lookup(self, key):
        index = self.hash_function(key)
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + 1) % self.size
        return None
```

Procedure:

1. Implement the hash table using chaining.
2. Implement the hash table using open addressing.
3. Insert a collection of telephone numbers into both hash tables.
4. Conduct lookups and tally the number of comparisons required for each technique.

Conclusion: The implementations of hash tables using chaining and open addressing yield efficient lookup operations. Chaining is easier to implement and effectively manages collisions, while open addressing may be more space-efficient but can require more comparisons in the event of collisions.

Questions for Review:

1. What is the time complexity of lookup operations in chaining and open addressing?
2. How does the choice of hash function affect the performance of the hash table?

Assignment 2

Title: Implementation of Dictionary ADT using Hashing with Collision Handling

Problem Statement: Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with/without replacement. Data consists of a set of (key, value) pairs. Keys are mapped to values, must be unique, and comparable. Standard operations to be supported include Insert(key, value), Find(key), and Delete(key).

Objectives:

- Implement dictionary operations using hashing.
- Handle collisions using chaining (with and without replacement).
- Ensure efficient storage and retrieval of key-value pairs.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* Python

Theory Explanation:
A dictionary Abstract Data Type (ADT) maps keys to values. Hashing is used for efficient storage and retrieval of data. Collisions occur when multiple keys map to the same hash index. To handle collisions, chaining can be used by maintaining a linked list at each bucket. In chaining with replacement, elements are inserted based on a calculated index even if another bucket position is occupied.

- ***Average Time Complexity:***
- Insertion: O(1)
- Search: O(1)
- Deletion: O(1)

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

Algorithms:

1. *Insert Operation (with Chaining)*
   - Add a new (key, value) pair to the dictionary.
   - Steps:
     1. Compute the hash index for the key.
     2. Check if the key already exists; if it does, update the value.
     3. If no collision, insert the key-value pair.

4. If a collision occurs, append the new pair to the linked list at the index.

2. *Find Operation*
   - Retrieve the value associated with a given key.
   - Steps:
     1. Compute the hash index for the key.
     2. Traverse the linked list at the index.
     3. If the key is found, return the associated value.
     4. If not found, return an error message.
3. *Delete Operation*
   - Remove a key-value pair from the dictionary.
   - Steps:
     1. Compute the hash index for the key.
     2. Traverse the linked list at the index.
     3. If the key is found, delete the key-value pair.
     4. If not found, return an error message.

Input/Output Format Examples:

- Input: Insert ("Alice", "12345"), Find ("Alice")
- Output: "12345"
- Input: Insert ("Bob", "67890"), Delete ("Bob"), Find ("Bob")
- Output: "Key not found"

Code Examples and Syntax Details:

```
class HashTableChaining:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def hash_function(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
```

```
        index = self.hash_function(key)
        # Check if the key already exists; update if found
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        # Append the new key-value pair
        self.table[index].append([key, value])
    def find(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return "Key not found"
    def delete(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                self.table[index].remove(pair)
                return f"Key '{key}' deleted"
        return "Key not found"
# Example Usage
dict_adt = HashTableChaining()
dict_adt.insert("Alice", "12345")
dict_adt.insert("Bob", "67890")
print("Find Alice:", dict_adt.find("Alice"))
print(dict_adt.delete("Bob"))
print("Find Bob:", dict_adt.find("Bob"))
```

Procedure:

1. Implement dictionary operations using hashing.
2. Handle collisions using chaining.
3. Test all operations including insert, find, and delete.
4. Analyze the performance of the dictionary operations.

Conclusion: The implementation of a dictionary using hashing with collision handling techniques demonstrates efficient storage and retrieval of key-value pairs. Chaining provides flexibility and prevents data loss due to collisions.

Questions for Review:

1. What are the advantages and disadvantages of using chaining for collision handling?
2. How can we improve the efficiency of hash table operations?

Group B

Assignment 3

Title: Tree Representation and Node Printing in a Book Structure

Problem Statement: A book consists of chapters, chapters consist of sections, and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

Objectives:

- Construct a hierarchical tree to represent the structure of a book.
- Traverse and print all nodes (chapters, sections, and subsections).
- Analyze the time and space requirements of the method.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
A tree is a hierarchical data structure consisting of nodes. Each node contains data and references to child nodes. The root node represents the book, while its children represent chapters. Chapters contain sections, and sections contain subsections. Trees are efficient for representing hierarchical data structures like book content.

Time Complexity: O(n) for traversal, where n is the total number of nodes.
Space Complexity: O(h), where h is the height of the tree for recursive traversal.

Algorithms:

1. *Tree Construction*
   - Construct a hierarchical tree to represent the book structure.
   - Steps:
     1. Create a root node for the book.
     2. Add child nodes for chapters.
     3. Add child nodes for sections and subsections under each chapter.
2. *Tree Traversal and Node Printing*
   - Traverse the tree and print all nodes in a hierarchical manner.
   - Steps:

1. Start from the root node.
2. Recursively visit and print each child node.

Input/Output Format Examples:

- Input: Book -> Chapter 1 -> Section 1.1 -> Subsection 1.1.1
- Output:

Book

Chapter 1

Section 1.1

Subsection 1.1.1

Code Examples and Syntax Details:

```
#include <iostream>
#include <vector>
using namespace std;
class Node
{public:
    string data;
    vector<Node*> children;
    Node(string data) {
        this->data = data; }
    void addChild(Node* child) {
        children.push_back(child); }
    void printTree(int level = 0) {
        for (int i = 0; i < level; i++) {
            cout << "  "; }
        cout << data << endl;
        for (Node* child : children) {
            child->printTree(level + 1); }}};
int main() {
    Node* book = new Node("Book");
```

```
Node* chapter1 = new Node("Chapter 1");
Node* section1 = new Node("Section 1.1");
Node* subsection1 = new Node("Subsection 1.1.1");
section1->addChild(subsection1);
chapter1->addChild(section1);
book->addChild(chapter1);
book->printTree();
return 0; }
```

Procedure:

1. Create a root node for the book.
2. Add nodes for chapters, sections, and subsections.
3. Implement a tree traversal function to print the nodes.
4. Analyze the time and space complexity.

Conclusion: The hierarchical representation of a book using a tree structure is efficient for storing and traversing book content. The recursive traversal method ensures all nodes are printed in a structured manner.

Questions for Review:

1. What is the time complexity of traversing a tree?
2. How can we improve space efficiency in tree traversal?

Assignment 4

Title: Binary Search Tree (BST) Construction and Operations

Problem Statement: Beginning with an empty binary search tree (BST), construct a BST by inserting values in the given order. After constructing the tree:

1. Insert a new node.
2. Find the number of nodes in the longest path from the root.
3. Find the minimum data value in the tree.
4. Change the tree so that the roles of left and right pointers are swapped at every node.
5. Search for a specific value.

Objectives:

- Construct a binary search tree by inserting values in a specified order.
- Perform various operations such as insertion, path length calculation, minimum value finding, swapping child pointers, and searching for a specific value.
- Analyze the efficiency of the implemented operations.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
A Binary Search Tree (BST) is a hierarchical data structure in which each node has at most two children. For each node, the left child contains values less than the node, and the right child contains values greater than the node. BSTs support efficient operations like insertion, deletion, and search in O(log n) time on average.

Algorithms:

1. *BST Construction and Insertion*
   - Insert nodes while maintaining the BST property.
   - Steps:
     1. If the tree is empty, create the root node.
     2. Compare the value to be inserted with the current node.

3. Recursively insert the value in the left or right subtree.

2. *Longest Path Calculation*
   - Find the depth of the tree by calculating the longest path from the root to a leaf node.
   - Steps:
     1. Compute the depth of the left and right subtrees recursively.
     2. Return the maximum depth plus one.
3. *Finding Minimum Value*
   - Find the node with the smallest value.
   - Steps:
     1. Traverse the left subtree until the leftmost node is reached.
4. *Swapping Child Pointers*
   - Swap left and right child pointers at every node.
   - Steps:
     1. Recursively swap left and right pointers for each node.
5. *Search Operation*
   - Search for a specific value in the BST.
   - Steps:
     1. Compare the target value with the current node.
     2. Recursively search the left or right subtree based on the comparison.

Input/Output Format Examples:

- Input: Insert 50, 30, 70, 20, 40, 60, 80; Find minimum; Search for 70
- Output:

  Minimum value: 20

  Value 70 found

- Input: Swap child pointers; Find longest path
- Output:

  Longest path length: 3

Code Examples and Syntax Details:

```
#include <iostream>
using namespace std;
class Node {
public:
    int data;    Node* left;
    Node* right;    Node(int value) {
        data = value;
        left = right = nullptr;}};
class BST {
public:
    Node* insert(Node* root, int key) {
        if (root == nullptr) {
            return new Node(key);}
        if (key < root->data) {
            root->left = insert(root->left, key);}
            else {
            root->right = insert(root->right, key);}
        return root;}
    int findMin(Node* root) {
        while (root->left != nullptr) {
            root = root->left;}
        return root->data;}
    int longestPath(Node* root) {
        if (root == nullptr) {
            return 0;}
        int leftDepth = longestPath(root->left);
        int rightDepth = longestPath(root->right);
        return max(leftDepth, rightDepth) + 1;}
    void swapChildren(Node* root) {
```

```
        if (root) {
            swap(root->left, root->right);  swapChildren(root->left);
            swapChildren(root->right);}}
    bool search(Node* root, int key) {
        if (root == nullptr) return false; if (root->data == key) return true;
        if (key < root->data) return search(root->left, key);
        return search(root->right, key); } };
int main() {
    BST tree;
    Node* root = nullptr;
    root = tree.insert(root, 50);
    tree.insert(root, 30);    tree.insert(root, 70);
    tree.insert(root, 20);    tree.insert(root, 40);    tree.insert(root, 60);
    tree.insert(root, 80);
    cout << "Minimum value: " << tree.findMin(root) << endl;
    cout << "Longest path length: " << tree.longestPath(root) << endl;
    tree.swapChildren(root);
    cout << "Search for 70: " << (tree.search(root, 70) ? "Found" : "Not Found") <<
endl;
      return 0;}
```

Procedure:

1. Construct the BST by inserting the given values.
2. Insert a new node and search for a specific value.
3. Find the longest path and the minimum value.
4. Swap the child pointers of all nodes.

Conclusion: The BST operations such as insertion, search, and traversal were successfully implemented. The tree's hierarchical structure allows for efficient data management and retrieval.

Questions for Review:

1. What is the time complexity of search and insertion operations in a BST?
2. How does swapping child pointers affect the BST structure?

Assignment 5

Title: Construction and Traversal of an Expression Tree

Problem Statement: Construct an expression tree from a given prefix expression (e.g., +--a*bc/def). Traverse the tree using postorder traversal (non-recursive) and delete the entire tree.

Objectives:

- Construct an expression tree from a given prefix expression.
- Perform non-recursive postorder traversal of the tree.
- Delete the entire expression tree after traversal.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
An expression tree is a binary tree used to represent mathematical expressions. Internal nodes represent operators, and leaf nodes represent operands. Traversals like postorder are used to evaluate or display the expression in different forms.

Traversal Techniques:

- **Postorder Traversal:** Left subtree → Right subtree → Root
- **Non-recursive Traversal:** Using a stack to maintain nodes explicitly.

Algorithms:

1. *Expression Tree Construction*
   - Build the expression tree from a prefix expression.
   - Steps:
     1. Scan the prefix expression from right to left.
     2. If the character is an operand, push it onto the stack as a node.
     3. If the character is an operator, pop two nodes from the stack, create a new node with the operator, and attach the popped nodes as children.
     4. Push the new node back onto the stack.
2. *Non-Recursive Postorder Traversal*

- Traverse the tree in postorder without recursion.
- Steps:
  1. Use two stacks—one for nodes and another for output.
  2. Push nodes to the first stack and record in the second stack in reverse postorder.
  3. Display the contents of the second stack.

3. *Delete Entire Tree*

- Deallocate all nodes in the tree.
- Steps:
  1. Perform a postorder traversal and delete each node.

Input/Output Format Examples:

- Input: Prefix expression "+*ab/-cde"
- Output:

  *Postorder Traversal: a b * c d e / - +*

  Tree Deleted
- Input: Prefix expression "-+abc"
- Output:

  Postorder Traversal: a b + c -

  Tree Deleted

Code Examples and Syntax Details:

```
#include <iostream>
#include <stack>
using namespace std;
class Node {
public:
    char data;
    Node* left;
    Node* right;
    Node(char value) {
```

```
        data = value;
        left = right = nullptr;}};
class ExpressionTree {
public:
    Node* buildTree(string prefix) {
        stack<Node*> st;
        for (int i = prefix.length() - 1; i >= 0; i--) {
            if (isalnum(prefix[i])) {
                st.push(new Node(prefix[i]));
            } else {
                Node* node = new Node(prefix[i]);
                node->left = st.top(); st.pop();
                node->right = st.top(); st.pop();
                st.push(node);}}
        return st.top();}
    void postorderTraversal(Node* root) {
        if (!root) return;
        stack<Node*> st1, st2;
        st1.push(root);
        while (!st1.empty()) {
            Node* node = st1.top(); st1.pop();
            st2.push(node);
            if (node->left) st1.push(node->left);
            if (node->right) st1.push(node->right);}
        cout << "Postorder Traversal: ";
        while (!st2.empty()) {
            cout << st2.top()->data << " ";
            st2.pop();}
        cout << endl;}
    void deleteTree(Node* root) {
```

```
        if (!root) return;
        deleteTree(root->left);
        deleteTree(root->right);
        delete root; }};
int main() {
    ExpressionTree tree;
    string prefix = "+*ab/-cde";
    Node* root = tree.buildTree(prefix);
    tree.postorderTraversal(root);
    tree.deleteTree(root);
    cout << "Tree Deleted" << endl;
    return 0;}
```

Procedure:

1. Read the prefix expression.
2. Construct the expression tree.
3. Perform non-recursive postorder traversal.
4. Delete the entire tree.

Conclusion: The expression tree was successfully constructed and traversed using non-recursive postorder traversal. The tree was deleted after the traversal, demonstrating efficient memory management.

Questions for Review:

1. What is the advantage of using non-recursive traversal methods?
2. How does an expression tree help in evaluating complex expressions?

Group C

Assignment 6

Title: Graph Representation and DFS/BFS Traversal

Problem Statement: Represent a given graph using adjacency matrix/list. Perform DFS using an adjacency matrix and BFS using an adjacency list. Use the map of the area around the college as the graph, identifying prominent landmarks as nodes.

Objectives:

- Represent a graph using adjacency matrix and list.
- Perform DFS using adjacency matrix representation.
- Perform BFS using adjacency list representation.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
A graph is a collection of vertices (nodes) and edges connecting pairs of vertices. Graph traversal techniques help explore all nodes in the graph.

Traversal Techniques:

1. **Depth First Search (DFS)**: Explores as far as possible along each branch before backtracking.
2. **Breadth First Search (BFS)**: Explores all neighbors at the current depth before moving to the next level.

Graph Representation Techniques:

1. **Adjacency Matrix:** A 2D array where matrix[i][j] = 1 if there is an edge between node i and node j.
2. **Adjacency List:** A list where each node points to a list of its adjacent nodes.

Algorithms:

1. *DFS using Adjacency Matrix*
    - Recursively traverse the graph using an adjacency matrix.
    - Steps:
        1. Start at the initial node.

2. Mark the node as visited.
3. Recursively visit all adjacent unvisited nodes.

2. *BFS using Adjacency List*
   - Traverse the graph level by level using a queue.
   - Steps:
     1. Start at the initial node.
     2. Mark the node as visited and enqueue it.
     3. Dequeue a node, visit it, and enqueue all its unvisited neighbors.

Input/Output Format Examples:

- Input: Graph with nodes {A, B, C, D}, Edges {A-B, A-C, B-D}
- DFS Output: A → B → D → C
- BFS Output: A → B → C → D
- Input: Graph with nodes {X, Y, Z}, Edges {X-Y, Y-Z}
- DFS Output: X → Y → Z
- BFS Output: X → Y → Z

Code Examples and Syntax Details:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
class Graph {
    int V;
    vector<int> *adj;
public:
    Graph(int V) {
        this->V = V;
        adj = new vector<int>[V];}
    void addEdge(int u, int v) {
```

```
        adj[u].push_back(v);}
    void DFSUtil(int v, vector<bool> &visited) {
        visited[v] = true;
        cout << v << " ";
        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited);}}}
    void DFS(int start) {
        vector<bool> visited(V, false);
        cout << "DFS Traversal: ";
        DFSUtil(start, visited);
        cout << endl;}
    void BFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;
        visited[start] = true;
        q.push(start);
        cout << "BFS Traversal: ";
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            cout << node << " ";
            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);}}}
        cout << endl;}};
int main() {
    Graph graph(4);
    graph.addEdge(0, 1);    graph.addEdge(0, 2);
```

```
graph.addEdge(1, 3);    graph.addEdge(2, 3);
graph.DFS(0);    graph.BFS(0);
return 0;}
```

Procedure:

1. Represent the graph using adjacency matrix and list.
2. Perform DFS using the adjacency matrix.
3. Perform BFS using the adjacency list.
4. Analyze the traversal results.

Conclusion: The graph was successfully represented using both adjacency matrix and list. DFS and BFS traversals were implemented and demonstrated effectively.

Questions for Review:

1. What is the time complexity of DFS and BFS for adjacency list representation?
2. How does the choice of graph representation affect memory usage and traversal performance?

Assignment 7

Title: Graph Representation and Connectivity Check

Problem Statement: There are flight paths between cities. If there is a flight between city A and city B, then there is an edge between the cities. The cost of the edge can represent the time taken or fuel consumed for the journey. Represent this as a graph using adjacency list or adjacency matrix. Check whether the graph is connected or not, and justify the storage representation used.

Objectives:

- Represent the graph using adjacency list or adjacency matrix.
- Implement methods to check whether the graph is connected.
- Analyze the storage efficiency of the chosen representation.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
A graph is a collection of vertices and edges representing relationships between entities. In this problem, the vertices are cities, and the edges are flight paths between them, with associated costs such as time or fuel consumption.

Graph Representation Techniques:

1. **Adjacency Matrix:** Suitable for dense graphs but uses more memory.
2. **Adjacency List:** Suitable for sparse graphs and more memory-efficient.

Graph Connectivity:
A graph is connected if there is a path between any two nodes. This can be verified using DFS or BFS traversal.

Algorithms:

1. *Graph Representation using Adjacency List*
    - Use an adjacency list to represent the graph, where each city points to a list of its connected cities and edge costs.
    - Steps:
        1. Initialize an empty list for each city.

2. For each edge, update the list of both connected cities.

2. *Graph Connectivity Check (DFS)*
    - Traverse the graph using DFS to check if all nodes can be visited from any starting node.
    - Steps:
        1. Start DFS from an initial node.
        2. Mark nodes as visited.
        3. Check if all nodes have been visited after the traversal.

Input/Output Format Examples:

- Input: Cities {A, B, C, D}, Flight paths {(A, B, 2 hours), (B, C, 3 hours), (C, D, 1 hour)}
- Output: Graph is connected
- Input: Cities {X, Y, Z}, Flight paths {(X, Y, 5 hours)}
- Output: Graph is not connected

Code Examples and Syntax Details:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
class Graph {
    int V; vector<int> *adj;
public:
    Graph(int V) {
        this->V = V;    adj = new vector<int>[V];}
    void addEdge(int u, int v) {
        adj[u].push_back(v);  adj[v].push_back(u);}
    void DFS(int v, vector<bool> &visited) {
        visited[v] = true;
        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFS(neighbor, visited);}}}
```

```
    bool isConnected() {
        vector<bool> visited(V, false);
        DFS(0, visited);
        for (bool visit : visited) {
            if (!visit) return false;}
        return true;}};
int main() {
    Graph graph(4);
    graph.addEdge(0, 1);    graph.addEdge(1, 2);    graph.addEdge(2, 3);
    if (graph.isConnected()) {
        cout << "Graph is connected" << endl;
    } else {
        cout << "Graph is not connected" << endl;}
    return 0;}
```

Procedure:

1. Represent the graph using an adjacency list or adjacency matrix.
2. Implement the DFS algorithm to check graph connectivity.
3. Analyze the storage efficiency of the chosen representation.

Conclusion: The graph was successfully represented and checked for connectivity. The adjacency list representation was more efficient for sparse graphs, and DFS was effective in verifying connectivity.

Questions for Review:

1. What is the time complexity of checking connectivity using DFS?
2. When would an adjacency matrix be more efficient than an adjacency list?

Group D

Assignment 8

Title: Construction of Optimal Binary Search Tree

Problem Statement: Given a sequence of sorted keys $k_1 < k_2 < \cdots < k_n$, with a search probability p_i for each key k_i build a binary search tree that has the least search cost given the access probabilities.

Objectives:

- Understand the concept of an optimal binary search tree (OBST).
- Implement the algorithm to build an OBST based on dynamic programming.
- Analyze the search cost efficiency of the constructed OBST.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:

A **Binary Search Tree (BST)** is a tree in which each node has at most two children, and the left child contains values less than the parent node, while the right child contains values greater than the parent node. An **Optimal Binary Search Tree (OBST)** minimizes the expected search cost based on the search probabilities of keys.

The algorithm uses dynamic programming to compute the cost of subtrees and identify the root nodes that lead to the least overall search cost.

Algorithms:

1. *Algorithm for OBST Construction Using Dynamic Programming*
    - Compute the optimal structure by minimizing search costs for all possible subtrees using a bottom-up approach.
    - Steps:
        1. Initialize cost and weight matrices.
        2. Compute the weight of all subtrees.
        3. Fill the cost table by considering all subtrees and possible roots.
        4. Record the root nodes for constructing the final OBST.

Input/Output Format Examples:

- Input: Keys = [10, 20, 30], Probabilities = [0.4, 0.2, 0.4]
- Output: OBST with root key 20
- Input: Keys = [5, 15, 25, 35], Probabilities = [0.1, 0.2, 0.4, 0.3]
- Output: OBST with root key 25

Code Examples and Syntax Details:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
struct OBSTNode {
    int key;    OBSTNode* left;
    OBSTNode* right;};
int sum(const vector<int>& p, int i, int j) {
    int total = 0;
    for (int k = i; k <= j; k++) {
        total += p[k];}
    return total;}
int optimalBST(const vector<int>& keys, const vector<int>& p) {
    int n = keys.size();
    vector<vector<int>> cost(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++) {
        cost[i][i] = p[i];}
    for (int length = 2; length <= n; length++) {
        for (int i = 0; i <= n - length; i++) {
            int j = i + length - 1;
            cost[i][j] = INT_MAX;
      for (int r = i; r <= j; r++) {
int c = (r > i ? cost[i][r - 1] : 0) + (r < j ? cost[r + 1][j] : 0) + sum(p, i, j);
                cost[i][j] = min(cost[i][j], c);}}}
    return cost[0][n - 1];}
```

```
int main() {
    vector<int> keys = {10, 20, 30};
    vector<int> probabilities = {40, 20, 40};
    cout << "Minimum cost of OBST: " << optimalBST(keys, probabilities) << endl;
    return 0;}
```

Procedure:

1. Define the keys and their search probabilities.
2. Use dynamic programming to compute the optimal BST cost and root structure.
3. Construct the OBST and analyze its search cost efficiency.

Conclusion: The optimal binary search tree was successfully constructed with minimum search cost using dynamic programming. The approach efficiently reduced search time by balancing the tree based on access probabilities.

Questions for Review:

1. What is the time complexity of constructing an OBST using dynamic programming?
2. Why is it important to use dynamic programming for OBST construction?

Assignment 9

Title: Dictionary Implementation Using Height Balanced Tree

Problem Statement: A dictionary stores keywords and their meanings. Implement operations to add new keywords, delete keywords, and update the values of any entry. Display the entire data in sorted order (ascending/descending). Find the maximum number of comparisons required to find a keyword and analyze the complexity using a height-balanced tree (AVL Tree).

Objectives:

- Understand the concept of height-balanced trees (AVL Trees).
- Implement dictionary operations using AVL Tree.
- Analyze the complexity for keyword search in height-balanced trees.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
An **AVL Tree** is a self-balancing Binary Search Tree (BST) where the difference between heights of the left and right subtrees of any node is at most one. The dictionary operations—insert, delete, and update—can be performed efficiently in O(logn)O(\log n)O(logn) time using AVL trees due to their balanced structure.

Key Operations:

- **Insertion:** Maintain balance by performing rotations if the AVL property is violated.
- **Deletion:** Similar to insertion, rotations are used to restore balance.
- **Updating:** Search for the key and modify its value.

Algorithms:

1. *Algorithm for Insertion in AVL Tree*
    - Insert a key and rebalance the tree if necessary.
    - Steps:
        1. Perform standard BST insertion.
        2. Update the height of the current node.

3. Compute the balance factor to check if the node is unbalanced.
4. Perform rotations (left, right, or double rotations) to rebalance.

2. *Algorithm for Deletion in AVL Tree*
   - Delete a key and rebalance the tree if necessary.
   - Steps:
     1. Perform standard BST deletion.
     2. Update the height of the current node.
     3. Compute the balance factor to check if the node is unbalanced.
     4. Perform rotations to rebalance the tree.

Input/Output Format Examples:

- Input: Insert ("Apple", "A fruit"), Insert ("Ball", "A toy"), Search ("Apple")
- Output: "A fruit"
- Input: Delete ("Ball"), Display in Ascending Order
- Output: "Apple: A fruit"

Code Examples and Syntax Details:

```
#include <iostream>
#include <string>
using namespace std;
struct Node {
    string key;
    string value;
    Node* left;
    Node* right;
    int height;};
int height(Node* N) {
    return (N == nullptr) ? 0 : N->height;}
Node* newNode(string key, string value) {
    Node* node = new Node();
```

```
    node->key = key;    node->value = value;
    node->left = node->right = nullptr;
    node->height = 1;    return node;}
Node* rightRotate(Node* y) {
    Node* x = y->left;    Node* T2 = x->right;
    x->right = y;    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;}
Node* leftRotate(Node* x) {
    Node* y = x->right;    Node* T2 = y->left;
    y->left = x;    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;}
Node* insert(Node* node, string key, string value) {
    if (node == nullptr) return newNode(key, value);
    if (key < node->key)
        node->left = insert(node->left, key, value);
    else if (key > node->key)
        node->right = insert(node->right, key, value);
    else {
        node->value = value; // Update the existing key
        return node;}
    node->height = 1 + max(height(node->left), height(node->right));
    int balance = height(node->left) - height(node->right);
    if (balance > 1 && key < node->left->key) return rightRotate(node);
    if (balance < -1 && key > node->right->key) return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
```

```
        return rightRotate(node);}
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);}
    return node;}
void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->key << ": " << root->value << " ";
        preOrder(root->left);
        preOrder(root->right);}}
int main() {
    Node* root = nullptr;
    root = insert(root, "Apple", "A fruit");
    root = insert(root, "Ball", "A toy");
    root = insert(root, "Cat", "An animal");
    preOrder(root);
    return 0;
}
```

Procedure:

1. Initialize the AVL Tree.
2. Implement dictionary operations (insert, delete, update, search).
3. Perform operations and display results in ascending/descending order.

Conclusion: The dictionary was successfully implemented using an AVL Tree. The height-balanced property ensured efficient search, insertion, and deletion operations with a time complexity of $O(\log n)$.

Questions for Review:

1. What is the role of rotations in maintaining the balance of an AVL Tree?
2. How does the search complexity of an AVL Tree compare to that of an unbalanced BST?

Group E

Assignment 10

Title: Graph Representation and Traversal Techniques

Problem Statement: Implement a graph using an adjacency list and perform Depth-First Search (DFS) and Breadth-First Search (BFS). Analyze their time complexities and compare their performances for different graph structures.

Objectives:

- Understand graph representations using adjacency lists.
- Implement graph traversal algorithms (DFS and BFS).
- Analyze and compare traversal complexities.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:

A **graph** is a collection of vertices (nodes) and edges (connections). Graphs can be represented using:

1. **Adjacency Matrix:** A 2D array where each entry (i,j)(i, j)(i,j) indicates the presence of an edge between vertex iii and vertex jjj.
2. **Adjacency List:** An array of lists, where each list contains the neighbors of a vertex.

Traversal Techniques:

- **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking. Implemented using recursion or a stack.
- **Breadth-First Search (BFS):** Explores all neighbors of a node before moving to the next level. Implemented using a queue.

Algorithms:

1. *DFS Algorithm*
    - Mark the current node as visited.
    - Recursively visit all its unvisited neighbors.
2. *BFS Algorithm*
    - Initialize a queue and enqueue the starting node.

- Mark the node as visited.
- Dequeue a node, explore its neighbors, and enqueue unvisited nodes.

Input/Output Format Examples:

- Input: Graph with edges (0, 1), (0, 2), (1, 2), (2, 0), (2, 3), (3, 3)
- Output: DFS from node 2: 2 0 1 3
- Output: BFS from node 2: 2 0 3 1

Code Examples and Syntax Details:

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;
class Graph {
    int V; // number of vertices
    list<int>* adj; // adjacency list
public:
    Graph(int V);
    void addEdge(int v, int w);
    void DFSUtil(int v, bool visited[]);
    void DFS(int v);
    void BFS(int s);
};
Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v, int w) {
    adj[v].push_back(w);
}
void Graph::DFSUtil(int v, bool visited[]) {
```

```
    visited[v] = true;
    cout << v << " ";

    for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}
void Graph::DFS(int v) {
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    DFSUtil(v, visited);
    delete[] visited;}
void Graph::BFS(int s) {
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
    queue<int> queue;
    visited[s] = true;
    queue.push(s);
    while (!queue.empty()) {
        s = queue.front();
        cout << s << " ";
        queue.pop();
        for (auto i = adj[s].begin(); i != adj[s].end(); ++i) {
            if (!visited[*i]) {
                visited[*i] = true;
                queue.push(*i);}}}}
int main() {
    Graph g(4);
```

```
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
    cout << "DFS starting from vertex 2:\n";
    g.DFS(2);
    cout << "\nBFS starting from vertex 2:\n";
    g.BFS(2);
    return 0;
}
```

Procedure:

1. Initialize the graph.
2. Implement DFS and BFS algorithms.
3. Test with different graph structures.
4. Analyze and compare their complexities.

Conclusion: The graph was successfully represented using an adjacency list, and both DFS and BFS were implemented. DFS was observed to follow a deeper exploration path, while BFS explored level-wise. Both techniques provided efficient traversal with time complexities of O(V+E)O(V + E)O(V+E).

Questions for Review:

1. In what scenarios is DFS more suitable than BFS?
2. Why is BFS better for finding the shortest path in an unweighted graph?

Group F

Assignment 11

Title: Student Information Management System Using Sequential File Handling

Problem Statement: Department maintains student information, including roll number, name, division, and address. Implement a system that allows users to:

1. Add student information
2. Delete student information
3. Display the information of a particular student
4. Display an appropriate message if the record does not exist. Use sequential file handling to maintain the data.

Objectives:

- Create and maintain a student database using sequential file handling.
- Allow the user to perform CRUD operations (Create, Read, Update, Delete).
- Handle cases when a student record is not found gracefully.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
Sequential file handling refers to storing and accessing records in a linear, sequential manner. In this method, the file is read from the beginning to the end to search or update specific records. Sequential file handling is simple but less efficient for large datasets compared to indexed or direct access methods.

Algorithms:

1. *Add Student Information*
    - Open the file in append mode.
    - Write the student details in the file.
    - Close the file.
2. *Delete Student Information*
    - Open the file in read mode and a temporary file in write mode.
    - Copy all records except the one to be deleted to the temporary file.

- Replace the original file with the temporary file.

3. *Display Student Information*
    - Open the file in read mode.
    - Search for the record by roll number.
    - If found, display the student details; otherwise, display an appropriate message.

Input/Output Format Examples:

- Input: Add Student (Roll No: 101, Name: Alice, Division: A, Address: "Street 1")
- Output: Student information added successfully
- Input: Display Student with Roll No 101
- Output: Roll No: 101, Name: Alice, Division: A, Address: Street 1
- Input: Display Student with Roll No 999
- Output: "Student record not found"

Code Examples and Syntax Details:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
struct Student {
    string roll_no, name, division, address;
};
void addStudent() {
    ofstream file("student_data.txt", ios::app);
    Student s;
    cout << "Enter Roll No: ";
    cin >> s.roll_no;
    cout << "Enter Name: ";
    cin >> s.name;
    cout << "Enter Division: ";
    cin >> s.division;
```

```
    cout << "Enter Address: ";
    cin >> s.address;
    file << s.roll_no << "," << s.name << "," << s.division << "," << s.address <<
endl;
    file.close();
    cout << "Student information added successfully.\n";
}
void deleteStudent() {
    ifstream file("student_data.txt");
    ofstream temp("temp.txt");
    string roll_no;
    bool found = false;
    cout << "Enter Roll No to delete: ";
    cin >> roll_no;
    string line;
    while (getline(file, line)) {
        if (line.find(roll_no) != 0) {
            temp << line << endl;
        } else {
            found = true;
        }
    }
    file.close();
    temp.close();
    remove("student_data.txt");
    rename("temp.txt", "student_data.txt");
    if (found) {
        cout << "Student information deleted successfully.\n";
    } else {
        cout << "Student record not found.\n";
    }
```

```
}
void displayStudent() {
    ifstream file("student_data.txt");
    string roll_no;
    bool found = false;
    cout << "Enter Roll No to search: ";
    cin >> roll_no;
    string line;
    while (getline(file, line)) {
        if (line.find(roll_no) == 0) {
            cout << "Student Details: " << line << endl;
            found = true;
            break;
        }
    }
    file.close();
    if (!found) {
        cout << "Student record not found.\n";}}
int main() {
    int choice;
    do {
        cout << "\n1. Add Student\n2. Delete Student\n3. Display Student\n4.
Exit\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1: addStudent(); break;
            case 2: deleteStudent(); break;
            case 3: displayStudent(); break;
            case 4: break;
            default: cout << "Invalid choice!\n";}}
while (choice != 4);
```

```
return 0;}
```

Procedure:

1. Create a file to store student records.
2. Implement functions to add, delete, and display student information.
3. Test the system by performing various operations.
4. Handle cases where the student record is not found.

Conclusion: The student information system was successfully implemented using sequential file handling. CRUD operations were handled efficiently, and appropriate messages were displayed for non-existent records.

Questions for Review:

1. What are the limitations of sequential file handling?
2. How can file handling be optimized for large datasets?

Assignment 12

Title: Employee Information Management System Using Indexed Sequential File Handling

Problem Statement: A company maintains employee information, including employee ID, name, designation, and salary. Implement a system that allows users to:

1. Add employee information
2. Delete employee information
3. Display the information of a particular employee
4. Display an appropriate message if the record does not exist. Use indexed sequential file handling to maintain the data.

Objectives:

- Create and maintain an employee database using indexed sequential file handling.
- Allow the user to perform CRUD operations (Create, Read, Update, Delete).
- Handle cases when an employee record is not found gracefully.

Experimental Setup:

- *Operating System:* Ubuntu
- *Programming Language:* C++
- *Compiler:* GCC

Theory Explanation:
Indexed sequential file handling involves maintaining a main file sorted by a key (like employee ID) and an index file for faster search operations. This method provides efficient access compared to purely sequential file handling by reducing search time.

Algorithms:

1. *Add Employee Information*
    - Open the file in append mode.
    - Write the employee details in the file.
    - Maintain the index file by sorting entries based on employee ID.
2. *Delete Employee Information*
    - Open the main and index files.
    - Copy all records except the one to be deleted to a temporary file.

- Replace the original file with the temporary file. Update the index file accordingly.

3. *Display Employee Information*
    - Open the index file and main file.
    - Perform a binary search on the index file to find the record.
    - Display the employee details if found, or an appropriate message if not.

Input/Output Format Examples:

- Input: Add Employee (Employee ID: 101, Name: John, Designation: Manager, Salary: 50000)
- Output: Employee information added successfully
- Input: Display Employee with ID 101
- Output: ID: 101, Name: John, Designation: Manager, Salary: 50000
- Input: Display Employee with ID 999
- Output: "Employee record not found"

Code Examples and Syntax Details:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <sstream>
using namespace std;
struct Employee {
    string id, name, designation;
    float salary;};
vector<Employee> employees;
void loadEmployees() {
    ifstream file("employee_data.txt");
    string line;
    employees.clear();
    while (getline(file, line)) {
```

```
        stringstream ss(line);
        Employee e;
        getline(ss, e.id, ',');
        getline(ss, e.name, ',');
        getline(ss, e.designation, ',');
        ss >> e.salary;
        employees.push_back(e);}
    file.close();}
void saveEmployees() {
    ofstream file("employee_data.txt");
    for (const auto& e : employees) {
        file << e.id << "," << e.name << "," << e.designation << "," << e.salary <<
endl;}
    file.close();}
void addEmployee() {
    Employee e;
    cout << "Enter Employee ID: ";
    cin >> e.id;
    cout << "Enter Name: ";
    cin >> e.name;
    cout << "Enter Designation: ";
    cin >> e.designation;
    cout << "Enter Salary: ";
    cin >> e.salary;
    employees.push_back(e);
    sort(employees.begin(), employees.end(), [](Employee a, Employee b) { return a.id
< b.id; });
    saveEmployees();
    cout << "Employee information added successfully.\n";}
void deleteEmployee() {
    string empId;
```

```
    bool found = false;
    cout << "Enter Employee ID to delete: ";
    cin >> empId;
    auto it = remove_if(employees.begin(), employees.end(), [&](Employee e) { return
e.id == empId; });
    if (it != employees.end()) {
        employees.erase(it, employees.end());
        found = true;}
    saveEmployees();
    if (found)
        cout << "Employee information deleted successfully.\n";
    else
        cout << "Employee record not found.\n";}
void displayEmployee() {
    string empId;
    bool found = false;
    cout << "Enter Employee ID to search: ";
    cin >> empId;
    for (const auto& e : employees) {
        if (e.id == empId) {
            cout << "ID: " << e.id << ", Name: " << e.name << ", Designation: " <<
e.designation << ", Salary: " << e.salary << endl;
            found = true;
            break;}}
    if (!found)
        cout << "Employee record not found.\n";
}
int main() {
    loadEmployees();
    int choice;
    do {
```

```
        cout << "\n1. Add Employee\n2. Delete Employee\n3. Display Employee\n4.
Exit\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1: addEmployee(); break;
            case 2: deleteEmployee(); break;
            case 3: displayEmployee(); break;
            case 4: break;
            default: cout << "Invalid choice!\n";}}
while (choice != 4);
    return 0;}
```

Procedure:

1. Create a file to store employee records.
2. Implement functions to add, delete, and display employee information.
3. Maintain an index file for efficient searching.
4. Handle cases where the employee record is not found.

Conclusion: The employee information system was successfully implemented using indexed sequential file handling. The index file allowed for faster search operations compared to sequential search alone.

Questions for Review:

1. How does indexed sequential file handling improve search efficiency?
2. What are the limitations of this approach compared to database systems?