DSAL Practical’s

Consider the telephone book database of N clients. Make use of a hash table implementation to quickly look up a client’s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers

def run():

    htable\_linear = [None] \* 10

    htable\_quadratic = [None] \* 10

    n = int(input("Enter number of Phone Numbers: "))

    for \_ in range(n):

        key = int(input("Enter Mobile Number: "))

        position = key % 10

        if htable\_linear[position] is None:

            htable\_linear[position] = [key]

        else:

            htable\_linear = linear\_probe(key, htable\_linear, position)

            print("Linear Probing for empty slot..")

        if htable\_quadratic[position] is None:

            htable\_quadratic[position] = [key]

        else:

            htable\_quadratic = quadric\_probe(key, htable\_quadratic, position)

            print("Quadratic Probing for empty slot..")

        display(htable\_linear, "Linear Probing")

        display(htable\_quadratic, "Quadratic Probing")

    return htable\_linear, htable\_quadratic

def linear\_probe(key, htable, pos):

    for i in range(1, 10):

        proble\_position = (pos + i) % 10

        if htable[proble\_position] is None:

            htable[proble\_position] = [key]

            return htable

    print("Hash table is full")

    return htable

def quadric\_probe(key, htable, pos):

    for i in range(1, 10):

        probe\_position = (pos + i \* i) % 10

        if htable[probe\_position] is None:

            htable[probe\_position] = [key]

            return htable

    print("Hash table is full")

    return htable

def display(htable, method):

    print(f"\n{method} Hash Table:")

    for pos in range(10):

        val = htable[pos]

        print(f"Position {pos}: {val}")

def search(key, ht, method):

    comparisons = 0

    pos = key % 10

    if ht[pos] is not None and ht[pos] == key:

        comparisons += 1

        return pos, comparisons

    for i in range(1, 10):

        comparisons += 1

        if method == "linear":

            probe\_position = (pos + i) % 10

        else:

            if method == "quadratic":

                probe\_position = (pos + i \* i) % 10

        if ht[probe\_position] is not None and ht[probe\_position] == key:

            return probe\_position, comparisons

    return -1, comparisons

def main():

    ht\_linear, ht\_quadratic = None, None

    while True:

        print("\nMenu:")

        print("1. Insert Phone Number")

        print("2. Search Phone Number")

        print("3. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:

            if ht\_linear is None and ht\_quadratic is None:

                ht\_linear, ht\_quadratic = run()

            else:

                print("Phone numbers already inserted.")

        elif choice == 2:

            if ht\_linear is None and ht\_quadratic is None:

                print("No phone numbers to search.")

            else:

                key = int(input("Enter Mobile Number to search: "))

                pos, comparisons = search(key, ht\_linear, "linear")

                if pos != -1:

                    print(f"Phone number {key} found at position {pos} in Linear Probing with {comparisons} comparisons.")

                else:

                    print(f"Phone number {key} not found in Linear Probing with {comparisons} comparisons.")

                pos, comparisons = search(key, ht\_quadratic, "quadratic")

                if pos != -1:

                    print(f"Phone number {key} found at position {pos} in Quadratic Probing with {comparisons} comparisons.")

                else:

                    print(f"Phone number {key} not found in Quadratic Probing with {comparisons} comparisons.")

        elif choice == 3:

            print("Exiting...")

            break

        else:

            print("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Experiment No. 6

Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and perform DFS and BFS on that.

#include<iostream>

#include<stack>

#include<queue>

#include<vector>

#include<map>

#include<iomanip>

using namespace std;

class Graph {

    private:

        int vertices;

        map<string, int> landmarkIndex;

        map<int, string> indexLandmark;

        vector<vector<int>> adjMatrix;

        map<int, vector<int>> adjList;

    public:

        Graph(int v){

            vertices = v;

            adjMatrix.resize(vertices, vector<int>(vertices, 0));

        }

        void addLandmark(string name, int index){

            landmarkIndex[name] = index;

            indexLandmark[index] = name;

        }

        void addEdge(string src, string dest){

            if(landmarkIndex.find(src) == landmarkIndex.end() || landmarkIndex.find(dest ) == landmarkIndex.end()){

                cout << "Invalid landmarks entered!\n";

                return;

            }

            int u = landmarkIndex[src];

            int v = landmarkIndex[dest];

            adjMatrix[u][v] = 1;

            adjMatrix[v][u] = 1; // For undirected graph

            adjList[u].push\_back(v);

            adjList[v].push\_back(u); // For undirected graph

        }

        void dfs()

        {

            cout << "Enter starting landmark for DFS: ";

            string startLandmark;

            getline(cin, startLandmark);

            if (landmarkIndex.find(startLandmark) == landmarkIndex.end()) {

                cout << "Invalid starting landmark!" << endl;

                return;

            }

            int start = landmarkIndex[startLandmark];

            vector<bool> visited(vertices, false);

            stack<int> s;

            s.push(start);

            visited[start] = true;

            cout << "DFS Traversal: ";

            while(!s.empty())

            {

                int node = s.top();

                s.pop();

                cout << indexLandmark[node] << " ";

                for( int i = 0; i < vertices; i++)

                {

                    if(adjMatrix[node][i] != 0 && !visited[i])

                    {

                        s.push(i);

                        visited[i] = true;

                    }

                }

            }

        }

        void bfs()

        {

            cout << "Enter starting landmark for BFS: ";

            string startLandmark;

            getline(cin, startLandmark);

            if (landmarkIndex.find(startLandmark) == landmarkIndex.end()) {

                cout << "Invalid starting landmark!" << endl;

                return;

            }

            int start = landmarkIndex[startLandmark];

            vector<bool> visited(vertices, false);

            queue<int> q;

            q.push(start);

            visited[start] = true;

            cout << "BFS Traversal: ";

            while(!q.empty())

            {

                int node = q.front();

                q.pop();

                cout << indexLandmark[node] << " ";

                for( int neighbor : adjList[node])

                {

                    if(!visited[neighbor]){

                        visited[neighbor] = true;

                        q.push(neighbor);

                    }

                }

            }

        }

       void displayGraph()

       {

         int maxwidth = 0;

         for( int i = 0; i < vertices; i++){

            if(indexLandmark[i].length() > maxwidth){

                maxwidth = indexLandmark[i].length();

            }

         }

         maxwidth += 2; // For padding

         cout << "Graph Adjacency Matrix:" << endl;

         cout << setw(maxwidth) << " ";

         for( int i = 0; i < vertices; i++){

            cout << setw(maxwidth) << indexLandmark[i];

         }

         cout << endl;

         for ( int i = 0; i < vertices; i++){

            cout << setw(maxwidth) << indexLandmark[i];

            for( int j = 0; j < vertices; j++){

                cout << setw(maxwidth) << adjMatrix[i][j];

            }

            cout << endl;

         }

         cout << "Graph Adjacency List:" << endl;

         for( auto &pair : adjList){

            cout << setw(maxwidth) << indexLandmark[pair.first] << "-> ";

            for( int neighbor : pair.second){

                cout << indexLandmark[neighbor] << " ";

            }

            cout << endl;

         }

        }

};

int main()

{

    int num\_landmarks;

    int choice;

    cout << "Enter the number of landmarks: ";

    cin >> num\_landmarks;

    cin.ignore();

    Graph g(num\_landmarks);

    cout << "Enter the landmarks (name): " << endl;

    for (int i = 0; i< num\_landmarks; i++){

        string name;

        getline(cin, name);

        g.addLandmark(name, i);

    }

    //menu driven

    do{

        cout<< endl << "Menu: " << endl;

        cout<< "1. Add Edge" << endl;

        cout<< "2. Display Graph" << endl;

        cout<< "3. DFS Traversal" << endl;

        cout<< "4. BFS Traversal" << endl;

        cout<< "5. Exit" << endl;

        cout<< "Enter your choice: ";

        cin >> choice;

        cin.ignore(); // To ignore the newline character after the integer input

        switch (choice) {

            case 1: {

                string src, dest;

                cout << "Enter source landmark: ";

                getline(cin, src);

                cout << "Enter destination landmark: ";

                getline(cin, dest);

                g.addEdge(src, dest);

                break;

            }

            case 2:

                g.displayGraph();

                break;

            case 3:

                g.bfs();

                break;

            case 4:

                g.dfs();

                break;

            case 5:

                cout << "Exiting program.\n";

                return 0;

            default:

                cout << "Invalid choice! Try again.\n";

        }

    }while(choice != 5);

    return 0;

}

Experiment No. 7

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

Experiment No. 8

Given sequence k = k1<K2…< Kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key?

Experiment No. 9

A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide a facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

#include <iostream>

#include <algorithm>

#include <queue>

#include <climits>

using namespace std;

struct Node {

    string keyword, meaning;

    Node\* left;

    Node\* right;

    int height;

    Node(string key, string m)

    {

        keyword = key;

        meaning = m;

        left = right = nullptr;

        height = 1;

    }

};

class AVLTree {

public:

    Node\* root = nullptr;

    AVLTree(){

        root = nullptr;

    }

    int getHeight(Node\* node) {

        if (node == nullptr) {

            return 0;

        }

        return node->height;

    }

    void updateHeight(Node\* node) {

        if (node != nullptr) {

            node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        }

    }

    int calculateBalanceFactor(Node\* node) {

        if (node == nullptr) {

            return 0;

        }

        return getHeight(node->left) - getHeight(node->right);

    }

    Node\* rightRotate(Node\* y) {

        cout << "Performing Right Rotation at node: " << y->keyword << endl;

        Node\* x = y->left;

        Node\* T2 = x->right;

        x->right = y;

        y->left = T2;

        updateHeight(y); // Update y first (now child)

        updateHeight(x); // Update x last (new root)

        return x;

    }

    Node\* leftRotate(Node\* x) {

        cout << "Performing Left Rotation at node: " << x->keyword << endl;

        Node\* y = x->right;

        Node\* T2 = y->left;

        y->left = x;

        x->right = T2;

        updateHeight(x); // Update x first (now child)

        updateHeight(y); // Update y last (new root)

        return y;

    }

    Node\* findMinNode(Node\* node) {

        while (node != nullptr && node->left != nullptr) {

            node = node->left;

        }

        return node;

    }

    Node\* balanceTree(Node\* node) {

        if (node == nullptr) {

            return node;

        }

        updateHeight(node);

        int balance = calculateBalanceFactor(node);

        // Left Heavy Cases

        if (balance > 1) {

            // Left-Right Case

            if (calculateBalanceFactor(node->left) < 0) {

                cout << "LR rotation needed at " << node->keyword << endl;

                node->left = leftRotate(node->left);

            }

            // Left-Left Case (or after LR's first step)

             cout << "LL rotation needed at " << node->keyword << endl;

            return rightRotate(node);

        }

        // Right Heavy Cases

        else if (balance < -1) {

            // Right-Left Case

            if (calculateBalanceFactor(node->right) > 0) {

                 cout << "RL rotation needed at " << node->keyword << endl;

                node->right = rightRotate(node->right);

            }

            // Right-Right Case (or after RL's first step)

             cout << "RR rotation needed at " << node->keyword << endl;

            return leftRotate(node);

        }

        return node;

    }

    Node\* insert(Node\* node, const string& keyword, const string& meaning, bool& inserted) {

        if (node == nullptr) {

            inserted = true;

            return new Node(keyword, meaning);

        }

        if (keyword < node->keyword) {

            node->left = insert(node->left, keyword, meaning, inserted);

        } else if (keyword > node->keyword) {

            node->right = insert(node->right, keyword, meaning, inserted);

        } else {

            cout << "Keyword '" << keyword << "' already exists. Use update option." << endl;

            inserted = false;

            return node;

        }

        if (!inserted) {

            return node;

        }

        return balanceTree(node);

    }

    Node\* remove(Node\* node, const string& keyword, bool& deleted) {

        if (node == nullptr) {

            deleted = false;

            return node;

        }

        if (keyword < node->keyword) {

            node->left = remove(node->left, keyword, deleted);

        } else if (keyword > node->keyword) {

            node->right = remove(node->right, keyword, deleted);

        } else {

            deleted = true;

            // Case 1 & 2: Node with only one child or no child

            if (node->left == nullptr || node->right == nullptr) {

                Node\* temp = node->left ? node->left : node->right;

                if (temp == nullptr) {

                   temp = node;

                   node = nullptr;

                } else {

                   \*node = \*temp;

                }

                delete temp;

                temp = nullptr;

            } else {

                // Case 3: Node with two children - replace with inorder successor

                Node\* temp = findMinNode(node->right);

                node->keyword = temp->keyword;

                node->meaning = temp->meaning;

                bool dummy\_deleted = true;

                node->right = remove(node->right, temp->keyword, dummy\_deleted);

            }

        }

        if (node == nullptr) {

            return node;

        }

        if (deleted) {

             return balanceTree(node);

        } else {

             return node;

        }

    }

    void printInOrder(Node\* node) {

        if (node != nullptr) {

            printInOrder(node->left);

            cout << node->keyword << " (BF: " << calculateBalanceFactor(node) << ") : " << node->meaning << endl;

            printInOrder(node->right);

        }

    }

    void printReverseInOrder(Node\* node) {

         if (node != nullptr) {

            printReverseInOrder(node->right);

            cout << node->keyword << " (BF: " << calculateBalanceFactor(node) << ") : " << node->meaning << endl;

            printReverseInOrder(node->left);

        }

    }

    Node\* findNode(const string& keyword) {

         Node\* current = root;

         while (current != nullptr) {

             if (keyword == current->keyword) {

                 return current;

             } else if (keyword < current->keyword) {

                 current = current->left;

             } else {

                 current = current->right;

             }

         }

         return nullptr;

    }

    string search(const string& keyword, int& comparisons) {

        Node\* current = root;

        comparisons = 0;

        while (current != nullptr) {

            comparisons++;

            if (keyword == current->keyword) {

                return current->meaning;

            } else if (keyword < current->keyword) {

                current = current->left;

            } else {

                current = current->right;

            }

        }

        return "Keyword not found.";

    }

    bool updateMeaning(const string& keyword, const string& newMeaning) {

        Node\* node = findNode(keyword);

        if (node != nullptr) {

            node->meaning = newMeaning;

            return true;

        }

        return false;

    }

    void printLevelOrder() {

        if (root == nullptr) {

            cout << "Tree is empty." << endl;

            return;

        }

        queue<Node\*> q;

        q.push(root);

        int level = 0;

        cout << "--- Level Order Traversal ---" << endl;

        while (!q.empty()) {

            int levelSize = q.size();

            cout << "Level " << level << ": ";

            bool level\_has\_nodes = false;

            for (int i = 0; i < levelSize; ++i) {

                Node\* current = q.front();

                q.pop();

                if (current != nullptr) {

                    cout << current->keyword << "(BF:" << calculateBalanceFactor(current) << ") ";

                    level\_has\_nodes = true;

                    q.push(current->left);

                    q.push(current->right);

                } else {

                     cout << "null ";

                }

            }

            cout << endl;

            // Check if remaining nodes are all null

            bool all\_null = true;

            queue<Node\*> temp\_q = q;

            while (!temp\_q.empty()) {

                if (temp\_q.front() != nullptr) {

                    all\_null = false;

                    break;

                }

                temp\_q.pop();

            }

            if (all\_null) break;

            level++;

        }

        cout << "-----------------------------" << endl;

    }

    int getTreeHeight() {

        return getHeight(root);

    }

    bool isEmpty() const {

        return root == nullptr;

    }

};

int main() {

    AVLTree dictionary;

    string keyword, meaning, newMeaning;

    int comparisons = 0;

    char addMore;

    int choice;

    bool inserted, deleted;

    while (true) {

        cout << "\n==== AVL Dictionary Menu ====\n";

        cout << "1. Add new keyword(s)\n";

        cout << "2. Print Ascending\n";

        cout << "3. Print Descending\n";

        cout << "4. Delete a keyword\n";

        cout << "5. Search a keyword\n";

        cout << "6. Update meaning\n";

        cout << "7. Get Tree Height\n";

        cout << "8. Display Tree Level Order\n";

        cout << "9. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                addMore = 'y';

                while (addMore == 'y' || addMore == 'Y') {

                    cout << "Enter keyword: ";

                    cin >> keyword;

                    cout << "Enter meaning: ";

                    cin.ignore();

                    getline(cin, meaning);

                    inserted = false;

                    dictionary.root = dictionary.insert(dictionary.root, keyword, meaning, inserted);

                    if (inserted) {

                        cout << "Keyword '" << keyword << "' inserted." << endl;

                    }

                    dictionary.printLevelOrder();

                    cout << "Add more keywords? (y/n): ";

                    cin >> addMore;

                }

                break;

            case 2:

                if (dictionary.isEmpty()) {

                    cout << "Dictionary is empty." << endl;

                } else {

                    cout << "--- Dictionary Entries (Ascending) ---" << endl;

                    dictionary.printInOrder(dictionary.root);

                    cout << "--------------------------------------" << endl;

                }

                break;

            case 3:

                if (dictionary.isEmpty()) {

                    cout << "Dictionary is empty." << endl;

                } else {

                    cout << "--- Dictionary Entries (Descending) ---" << endl;

                    dictionary.printReverseInOrder(dictionary.root);

                    cout << "---------------------------------------" << endl;

                }

                break;

            case 4:

                if (dictionary.isEmpty()) {

                    cout << "Dictionary is empty. Cannot delete." << endl;

                } else {

                    cout << "Enter keyword to delete: ";

                    cin >> keyword;

                    deleted = false;

                    dictionary.root = dictionary.remove(dictionary.root, keyword, deleted);

                    if (deleted) {

                        cout << "Keyword '" << keyword << "' deleted." << endl;

                        dictionary.printLevelOrder();

                    } else {

                        cout << "Keyword '" << keyword << "' not found, cannot delete." << endl;

                    }

                }

                break;

            case 5:

                 if (dictionary.isEmpty()) {

                     cout << "Dictionary is empty." << endl;

                } else {

                    cout << "Enter keyword to search: ";

                    cin >> keyword;

                    comparisons = 0;

                    string result = dictionary.search(keyword, comparisons);

                    if (result == "Keyword not found.") {

                         cout << "Keyword '" << keyword << "' not found. (Comparisons: " << comparisons << ")" << endl;

                    } else {

                         cout << "Meaning: " << result << " (Comparisons: " << comparisons << ")" << endl;

                    }

                }

                break;

            case 6:

                 if (dictionary.isEmpty()) {

                     cout << "Dictionary is empty." << endl;

                } else {

                    cout << "Enter keyword to update: ";

                    cin >> keyword;

                    cout << "Enter new meaning: ";

                    cin.ignore();

                    getline(cin, newMeaning);

                    if (dictionary.updateMeaning(keyword, newMeaning)) {

                        cout << "Meaning updated successfully." << endl;

                    } else {

                        cout << "Keyword '" << keyword << "' not found, cannot update." << endl;

                    }

                }

                break;

            case 7:

                cout << "Tree Height: " << dictionary.getTreeHeight() << endl;

                break;

            case 8:

                cout << "Current Tree State:" << endl;

                dictionary.printLevelOrder();

                break;

            case 9:

                 cout << "Exiting program." << endl;

                return 0;

            default:

                cout << "Invalid choice. Please choose a valid option." << endl;

        }

    }

}