**DS LAB 11**

**Task 01**

#include <iostream>

const int TABLE\_SIZE = 10;

class Node{

public:

    int key;

    int value;

    Node\* next;

    Node(int k, int v){

        key = k;

        value = v;

        next = NULL;

    }

};

class hash\_table{

public:

    Node\* table[TABLE\_SIZE];

    hash\_table(){

        for(int i = 0; i < TABLE\_SIZE; i++) {

            table[i] = NULL;

        }

    }

    int hash(int key) {

        return key % TABLE\_SIZE;

    }

    void insert(int key, int value) {

        int index = hash(key);

        Node\* newNode = new Node(key, value);

        if(table[index] == NULL) {

            table[index] = newNode;

            std::cout << "Inserted " << key << " with Value " << value << " at Index " << index << std::endl;

        }

        else{

            Node\* current = table[index];

            while(current != NULL) {

                current = current->next;

            }

            newNode->next = table[index];

            table[index] = newNode;

            std::cout << "Inserted " << key << " with Value " << value << " at Index " << index << " (Collision Handled)" << std::endl;

        }

    }

    void search(int key) {

        int index = hash(key);

        Node\* current = table[index];

        while(current != NULL) {

            if(current->key == key) {

                std::cout << std::endl << key << " Found in Hash Table\n";

                return;

            }

            current = current->next;

        }

        std::cout << std::endl << key << " not Found in Hash Table\n";

    }

    void delete\_ht(int key) {

        int index = hash(key);

        Node\* current = table[index];

        Node\* prev = NULL;

        while(current != NULL) {

            if(current->key == key) {

                if(prev == NULL) {

                    table[index] = current->next;

                }

                else{

                    prev->next = current->next;

                }

                delete current;

                std::cout << "Deleted key " << key << " from Index " << index << std::endl;

                return;

            }

            prev = current;

            current = current->next;

        }

        std::cout << std::endl << "Key " << key << " not Found in Hash Table" << std::endl;

    }

    void print\_table(){

        for(int i = 0; i < TABLE\_SIZE; i++) {

            std::cout << i << ": ";

            Node\* current = table[i];

            while(current != NULL){

                std::cout << "(" << current->key << ", " << current->value << ") -> ";

                current = current->next;

            }

            std::cout << "NULL" << std::endl;

        }

    }

};

int main() {

    hash\_table hash\_table;

    hash\_table.insert(20, 100);

    hash\_table.insert(34, 200);

    hash\_table.insert(45, 300);

    hash\_table.insert(70, 400);

    hash\_table.insert(56, 500);

    std::cout << "Hash Table After Insertion: " << std::endl;

    hash\_table.print\_table();

    hash\_table.search(34);

    hash\_table.delete\_ht(34);

    hash\_table.delete\_ht(70);

    hash\_table.delete\_ht(100);

    std::cout << std::endl << "Hash Table After Deletions: " << std::endl;

    hash\_table.print\_table();

    return 0;

}

A screenshot of a computer program

Description automatically generated

**Task 02**

#include <iostream>

void count\_pairs(int arr[], int N, int K) {

    int hash\_table[10000] = {0};

    int pair\_count = 0;

    for(int i = 0; i < N; i++) {

        int complement = K - arr[i];

        if(complement >= -1000 && complement <= 1000) {

            pair\_count += hash\_table[complement + 1000];

        }

        hash\_table[arr[i] + 1000]++;

    }

    std::cout << "Number of Pairs with Sum " << K << " is: " << pair\_count << std::endl;

}

int main() {

    int arr[] = {1, 5, 7, 1};

    int n = 4, k = 6;

    count\_pairs(arr, n, k);

    return 0;

}

A screen shot of a computer

Description automatically generated

**Task 03**

#include <iostream>

void find\_triplet(int arr[], int size) {

    bool found = false;

    for(int i=0; i < size - 2; i++) {

        int hash[2001] = {0};

        for(int j = i + 1; j < size; j++) {

            int complement = -(arr[i] + arr[j]);

            if(complement >= -1000 && complement <= 1000 && hash[complement + 1000] > 0) {

                std::cout << "Triplet found: " << arr[i] << " " << arr[j] << " " << complement << std::endl;

                found = true;

            }

            hash[arr[j] + 1000] = 1;

        }

    }

    if(!found) {

        std::cout << "Triplet not found" << std::endl;

    }

}

int main() {

    int arr[] = {0, -1, 2, -3, 1};

    int size = 5;

    find\_triplet(arr, size);

    return 0;

}

