National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Section: BCS-4B

Lab # 14

# Problem 01

#include <iostream>

#include <cassert>

using namespace std;

template <class v>

struct HashItem

{

int key;

v value;

short status; // 0: Empty, 1: Occupied, -1: Deleted

};

template <class v>

class HashMap

{

private:

HashItem<v>\* hashArray;

int capacity;

int currentElements;

double loadFactorThreshold = 0.75;

void doubleCapacity();

virtual int getNextCandidateIndex(int key, int i);

public:

HashMap()

: HashMap(10) {}

HashMap(int const capacity);

void insert(int const key, v const value);

bool deleteKey(int const key);

v\* get(int const key);

~HashMap();

};

template <class v>

HashMap<v>::HashMap(int const capacity)

{

assert(capacity > 1 && "Capacity must be greater than 1");

this->capacity = capacity;

hashArray = new HashItem<v>[capacity]();

currentElements = 0;

}

template <class v>

void HashMap<v>::doubleCapacity()

{

int oldCapacity = capacity;

HashItem<v>\* oldArray = hashArray;

capacity \*= 2;

hashArray = new HashItem<v>[capacity]();

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

{

if (oldArray[i].status == 1)

{

int key = oldArray[i].key;

v value = oldArray[i].value;

insert(key, value);

}

}

delete[] oldArray;

}

template <class v>

int HashMap<v>::getNextCandidateIndex(int key, int i)

{

return (key + i) % capacity;

}

template <class v>

void HashMap<v>::insert(int const key, v const value)

{

if (currentElements >= loadFactorThreshold \* capacity)

{

doubleCapacity();

}

int i = 0;

int index = key % capacity;

while (i < capacity)

{

if (hashArray[index].status != 1)

{

hashArray[index].key = key;

hashArray[index].value = value;

hashArray[index].status = 1; // Mark as occupied

currentElements++;

return;

}

i++;

index = getNextCandidateIndex(key, i);

}

cout << "Hash Table is full. Cannot insert." << endl;

}

template <class v>

bool HashMap<v>::deleteKey(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

hashArray[index].status = -1; // Mark as deleted

currentElements--;

return true;

}

i++;

index = getNextCandidateIndex(key, i);

}

return false;

}

template <class v>

v\* HashMap<v>::get(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

return &(hashArray[index].value);

}

i++;

index = getNextCandidateIndex(key, i);

}

return nullptr; // Key not found or table is full

}

template <class v>

HashMap<v>::~HashMap()

{

delete[] hashArray;

}

int main()

{

HashMap<int> hash(10);

hash.insert(1, 10);

hash.insert(2, 20);

hash.insert(3, 30);

hash.insert(4, 40);

hash.insert(5, 50);

hash.insert(6, 60);

hash.insert(7, 70);

hash.insert(8, 80);

hash.insert(9, 90);

hash.insert(10, 100);

for (int i = 1; i <= 10; i++)

{

int\* val = hash.get(i);

if (val != nullptr)

{

cout << "Value at key " << i << ": " << \*val << endl;

}

else

{

cout << "Value at key " << i << " not found" << \*val << endl;

}

}

for (int i = 1; i <= 5; i++)

{

bool deleted = hash.deleteKey(i);

if (deleted)

{

cout << "Key " << i << " deleted successfully" << endl;

}

else

{

cout << "Key " << i << " deletion failed or not found" << endl;

}

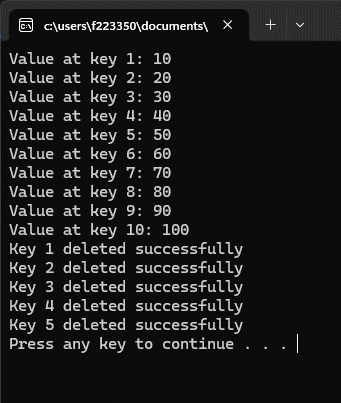
}

system("pause");

return 0;

}

## Output



# Problem 02

#include <iostream>

#include <cassert>

using namespace std;

template <class v>

struct HashItem

{

int key;

v value;

short status; // 0: Empty, 1: Occupied, -1: Deleted

};

template <class v>

class HashMap

{

private:

public:

HashItem<v>\* hashArray;

int currentElements;

int capacity;

double loadFactorThreshold = 0.75;

void doubleCapacity();

virtual int getNextCandidateIndex(int key, int i);

HashMap()

: HashMap(10) {}

HashMap(int const capacity);

void insert(int const key, v const value);

bool deleteKey(int const key);

v\* get(int const key);

~HashMap();

};

template <class v>

HashMap<v>::HashMap(int const capacity)

{

assert(capacity > 1 && "Capacity must be greater than 1");

this->capacity = capacity;

hashArray = new HashItem<v>[capacity]();

currentElements = 0;

}

template <class v>

void HashMap<v>::doubleCapacity()

{

int oldCapacity = capacity;

HashItem<v>\* oldArray = hashArray;

capacity \*= 2;

hashArray = new HashItem<v>[capacity]();

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

{

if (oldArray[i].status == 1)

{

int key = oldArray[i].key;

v value = oldArray[i].value;

insert(key, value);

}

}

delete[] oldArray;

}

template <class v>

int HashMap<v>::getNextCandidateIndex(int key, int i)

{

return (key + i) % capacity;

}

template <class v>

void HashMap<v>::insert(int const key, v const value)

{

if (currentElements >= loadFactorThreshold \* capacity)

{

doubleCapacity();

}

int i = 0;

int index = key % capacity;

while (i < capacity)

{

if (hashArray[index].status != 1)

{

hashArray[index].key = key;

hashArray[index].value = value;

hashArray[index].status = 1; // Mark as occupied

currentElements++;

return;

}

i++;

index = getNextCandidateIndex(key, i);

}

cout << "Hash Table is full, Cannot insert" << endl;

}

template <class v>

bool HashMap<v>::deleteKey(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

hashArray[index].status = -1; // Mark as deleted

currentElements--;

return true;

}

i++;

index = getNextCandidateIndex(key, i);

}

return false;

}

template <class v>

v\* HashMap<v>::get(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

return &(hashArray[index].value);

}

i++;

index = getNextCandidateIndex(key, i);

}

return nullptr; // Key not found or table is full

}

template <class v>

HashMap<v>::~HashMap()

{

delete[] hashArray;

}

template <class v>

class QHashMap : public HashMap<v>

{

private:

virtual int getNextCandidateIndex(int key, int i) override

{

return (key + (i\*i)) % capacity; // Adding square of i

}

public:

HashMap<v>::HashMap;

};

int main()

{

QHashMap<int> hash(10);

hash.insert(1, 10);

hash.insert(2, 20);

hash.insert(3, 30);

hash.insert(4, 40);

hash.insert(5, 50);

hash.insert(6, 60);

hash.insert(7, 70);

hash.insert(8, 80);

hash.insert(9, 90);

cout << "Using QHashMap class" << endl;

for (int i = 1; i <= 9; i++)

{

int\* val = hash.get(i);

if (val != nullptr)

{

cout << "Value at key " << i << ": " << \*val << endl;

}

else

{

cout << "Value at key " << i << " not found" << \*val << endl;

}

}

for (int i = 1; i <= 5; i++)

{

bool deleted = hash.deleteKey(i);

if (deleted)

{

cout << "Key " << i << " deleted successfully" << endl;

}

else

{

cout << "Key " << i << " deletion failed or not found" << endl;

}

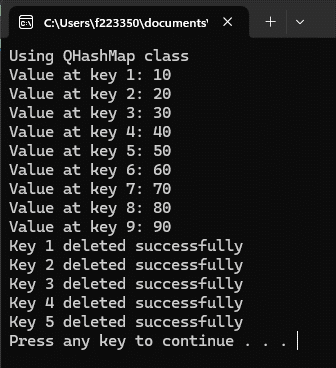
}

system("pause");

return 0;

}

## Output



# Problem 03

#include <iostream>

#include <cassert>

using namespace std;

template <class v>

struct HashItem

{

int key;

v value;

short status; // 0: Empty, 1: Occupied, -1: Deleted

};

template <class v>

class HashMap

{

private:

public:

HashItem<v>\* hashArray;

int currentElements;

int capacity;

double loadFactorThreshold = 0.75;

void doubleCapacity();

virtual int getNextCandidateIndex(int key, int i);

HashMap()

: HashMap(10) {}

HashMap(int const capacity);

void insert(int const key, v const value);

bool deleteKey(int const key);

v\* get(int const key);

~HashMap();

};

template <class v>

HashMap<v>::HashMap(int const capacity)

{

assert(capacity > 1 && "Capacity must be greater than 1");

this->capacity = capacity;

hashArray = new HashItem<v>[capacity]();

currentElements = 0;

}

template <class v>

void HashMap<v>::doubleCapacity()

{

int oldCapacity = capacity;

HashItem<v>\* oldArray = hashArray;

capacity \*= 2;

hashArray = new HashItem<v>[capacity]();

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

{

if (oldArray[i].status == 1)

{

int key = oldArray[i].key;

v value = oldArray[i].value;

insert(key, value);

}

}

delete[] oldArray;

}

template <class v>

int HashMap<v>::getNextCandidateIndex(int key, int i)

{

return (key + i) % capacity;

}

template <class v>

void HashMap<v>::insert(int const key, v const value)

{

if (currentElements >= loadFactorThreshold \* capacity)

{

doubleCapacity();

}

int i = 0;

int index = key % capacity;

while (i < capacity)

{

if (hashArray[index].status != 1)

{

hashArray[index].key = key;

hashArray[index].value = value;

hashArray[index].status = 1; // Mark as occupied

currentElements++;

return;

}

i++;

index = getNextCandidateIndex(key, i);

}

cout << "Hash Table is full, Cannot insert" << endl;

}

template <class v>

bool HashMap<v>::deleteKey(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

hashArray[index].status = -1; // Mark as deleted

currentElements--;

return true;

}

i++;

index = getNextCandidateIndex(key, i);

}

return false;

}

template <class v>

v\* HashMap<v>::get(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

return &(hashArray[index].value);

}

i++;

index = getNextCandidateIndex(key, i);

}

return nullptr; // Key not found or table is full

}

template <class v>

HashMap<v>::~HashMap()

{

delete[] hashArray;

}

template <class v>

class DHashMap : public HashMap<v>

{

private:

virtual int getNextCandidateIndex(int key, int i) override

{

int prime = 5;

int first\_value = key % capacity;

int second\_value = (prime - (key % prime));

return (first\_value + i \* second\_value) % capacity; // Double hashing

}

public:

HashMap<v>::HashMap;

};

int main()

{

DHashMap<int> hash(10);

hash.insert(1, 10);

hash.insert(2, 20);

hash.insert(3, 30);

hash.insert(4, 40);

hash.insert(5, 50);

cout << "Using DHashMap class" << endl;

for (int i = 1; i <= 5; i++)

{

int\* val = hash.get(i);

if (val != nullptr)

{

cout << "Value at key " << i << ": " << \*val << endl;

}

else

{

cout << "Value at key " << i << " not found" << \*val << endl;

}

}

for (int i = 1; i <= 2; i++)

{

bool deleted = hash.deleteKey(i);

if (deleted)

{

cout << "Key " << i << " deleted successfully" << endl;

}

else

{

cout << "Key " << i << " deletion failed or not found" << endl;

}

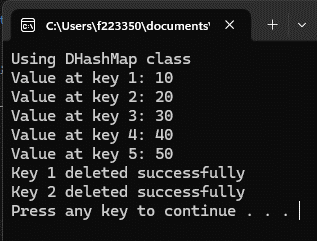
}

system("pause");

return 0;

}

## Output



# Problem 04

#include <iostream>

#include <cassert>

#include <string>

#include <fstream>

#include <sstream>

using namespace std;

template <class v>

struct HashItem

{

int key;

v value;

short status; // 0: Empty, 1: Occupied, -1: Deleted

};

template <class v>

class HashMap

{

private:

HashItem<v>\* hashArray;

int capacity;

int currentElements;

double loadFactorThreshold = 0.75;

void doubleCapacity();

virtual int getNextCandidateIndex(int key, int i);

public:

HashMap()

: HashMap(10) {}

HashMap(int const capacity);

void insert(int const key, v const value);

bool deleteKey(int const key);

v\* get(int const key);

~HashMap();

};

template <class v>

HashMap<v>::HashMap(int const capacity)

{

assert(capacity > 1 && "Capacity must be greater than 1");

this->capacity = capacity;

hashArray = new HashItem<v>[capacity]();

currentElements = 0;

}

template <class v>

void HashMap<v>::doubleCapacity()

{

int oldCapacity = capacity;

HashItem<v>\* oldArray = hashArray;

capacity \*= 2;

hashArray = new HashItem<v>[capacity]();

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

{

if (oldArray[i].status == 1)

{

int key = oldArray[i].key;

v value = oldArray[i].value;

insert(key, value);

}

}

delete[] oldArray;

}

template <class v>

int HashMap<v>::getNextCandidateIndex(int key, int i)

{

return (key + i) % capacity;

}

template <class v>

void HashMap<v>::insert(int const key, v const value)

{

if (currentElements >= loadFactorThreshold \* capacity)

{

doubleCapacity();

}

int i = 0;

int index = key % capacity;

while (i < capacity)

{

if (hashArray[index].status != 1)

{

hashArray[index].key = key;

hashArray[index].value = value;

hashArray[index].status = 1; // Mark as occupied

currentElements++;

return;

}

i++;

index = getNextCandidateIndex(key, i);

}

cout << "Hash Table is full, Cannot insert" << endl;

}

template <class v>

bool HashMap<v>::deleteKey(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

hashArray[index].status = -1; // Mark as deleted

currentElements--;

return true;

}

i++;

index = getNextCandidateIndex(key, i);

}

return false;

}

template <class v>

v\* HashMap<v>::get(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

return &(hashArray[index].value);

}

i++;

index = getNextCandidateIndex(key, i);

}

return nullptr; // Key not found or table is full

}

template <class v>

HashMap<v>::~HashMap()

{

delete[] hashArray;

}

void populateHash(string filename, HashMap<string> \*hash)

{

ifstream dataFile(filename);

if (!dataFile)

{

cout << "Not able to open " << filename << endl;

return;

}

string line;

while (getline(dataFile, line))

{

istringstream iss(line);

string id, name;

if (iss >> id >> name)

{

hash->insert(stoi(id), name);

}

}

dataFile.close();

}

int main()

{

HashMap<string> hash(10);

populateHash("data.txt", &hash);

for (int i = 1; i <= 10; i++)

{

string\* val = hash.get(i);

if (val != nullptr)

{

cout << "Value at key " << i << ": " << \*val << endl;

}

else

{

cout << "Value at key " << i << " not found" << endl;

}

}

for (int i = 1; i <= 5; i++)

{

bool deleted = hash.deleteKey(i);

if (deleted)

{

cout << "Key " << i << " deleted successfully" << endl;

}

else

{

cout << "Key " << i << " deletion failed or not found" << endl;

}

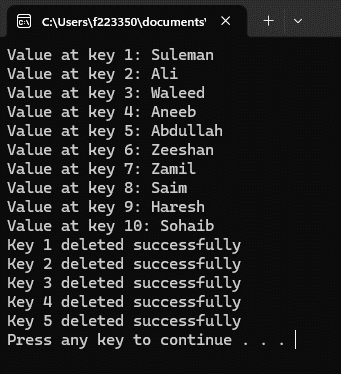
}

system("pause");

return 0;

}

## Output



# Problem 05

#include <iostream>

#include <cassert>

#include <string>

#include <fstream>

#include <sstream>

using namespace std;

template <class v>

struct HashItem

{

int key;

v value;

short status; // 0: Empty, 1: Occupied, -1: Deleted

};

template <class v>

class HashMap

{

public:

HashItem<v>\* hashArray;

int capacity;

int currentElements;

double loadFactorThreshold = 0.75;

void doubleCapacity();

virtual int getNextCandidateIndex(int key, int i);

public:

HashMap()

: HashMap(10) {}

HashMap(int const capacity);

void insert(int const key, v const value);

bool deleteKey(int const key);

v\* get(int const key);

~HashMap();

};

template <class v>

HashMap<v>::HashMap(int const capacity)

{

assert(capacity > 1 && "Capacity must be greater than 1");

this->capacity = capacity;

hashArray = new HashItem<v>[capacity]();

currentElements = 0;

}

template <class v>

void HashMap<v>::doubleCapacity()

{

int oldCapacity = capacity;

HashItem<v>\* oldArray = hashArray;

capacity \*= 2;

hashArray = new HashItem<v>[capacity]();

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

{

if (oldArray[i].status == 1)

{

int key = oldArray[i].key;

v value = oldArray[i].value;

insert(key, value);

}

}

delete[] oldArray;

}

template <class v>

int HashMap<v>::getNextCandidateIndex(int key, int i)

{

return (key + i) % capacity;

}

template <class v>

void HashMap<v>::insert(int const key, v const value)

{

if (currentElements >= loadFactorThreshold \* capacity)

{

doubleCapacity();

}

int i = 0;

int index = key % capacity;

while (i < capacity)

{

if (hashArray[index].status != 1)

{

hashArray[index].key = key;

hashArray[index].value = value;

hashArray[index].status = 1; // Mark as occupied

currentElements++;

return;

}

i++;

index = getNextCandidateIndex(key, i);

}

cout << "Hash Table is full, Cannot insert" << endl;

}

template <class v>

bool HashMap<v>::deleteKey(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

hashArray[index].status = -1; // Mark as deleted

currentElements--;

return true;

}

i++;

index = getNextCandidateIndex(key, i);

}

return false;

}

template <class v>

v\* HashMap<v>::get(int const key)

{

int i = 0;

int index = key % capacity;

while (i < capacity && (hashArray[index].status == 1 || hashArray[index].status == -1))

{

if (hashArray[index].status == 1 && hashArray[index].key == key)

{

return &(hashArray[index].value);

}

i++;

index = getNextCandidateIndex(key, i);

}

return nullptr; // Key not found or table is full

}

template <class v>

HashMap<v>::~HashMap()

{

delete[] hashArray;

}

void populateHash(string filename, HashMap<string> \*hash)

{

ifstream dataFile(filename);

if (!dataFile)

{

cout << "Not able to open " << filename << endl;

return;

}

string line;

while (getline(dataFile, line))

{

istringstream iss(line);

string id, name;

if (iss >> id >> name)

{

hash->insert(stoi(id), name);

}

}

dataFile.close();

}

template <class v>

class QHashMap : public HashMap<v>

{

private:

virtual int getNextCandidateIndex(int key, int i) override

{

return (key + (i\*i)) % capacity; // Adding square of i

}

public:

HashMap<v>::HashMap;

};

template <class v>

class DHashMap : public HashMap<v>

{

private:

virtual int getNextCandidateIndex(int key, int i) override

{

int prime = 5;

int first\_value = key % capacity;

int second\_value = (prime - (key % prime));

return (first\_value + i \* second\_value) % capacity; // Double hashing

}

public:

HashMap<v>::HashMap;

};

int main()

{

HashMap<string> \*map;

map = new HashMap<string>;

populateHash("students.txt", map);

cout << \*map->get(9);

map->deleteKey(9);

assert(map->get(9) == nullptr);

delete map;

cout << endl;

map = new QHashMap<string>;

populateHash("students.txt", map);

cout << \*map->get(98);

map->deleteKey(98);

assert(map->get(98) == nullptr);

delete map;

cout << endl;

map = new DHashMap<string>;

populateHash("students.txt", map);

cout << \*map->get(101);

map->deleteKey(101);

assert(map->get(101) == nullptr);

delete map;

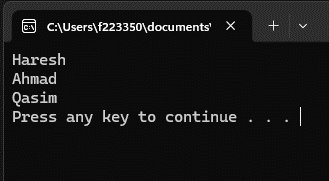
cout << endl;

system("pause");

return 0;

}

## Output



# Problem 06

#include <iostream>

#include <string>

using namespace std;

struct Node

{

int key;

Node \*next;

};

// Re-Hashing

class Rehashing

{

private:

int size;

Node \*\*table;

public:

Rehashing(int size)

: size(size)

{

table = new Node \*[size];

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

{

table[i] = nullptr;

}

}

int hashFunc(int key)

{

return key % size;

}

void insert(int key)

{

int index = hashFunc(key);

Node \*newNode = new Node();

newNode->key = key;

newNode->next = nullptr;

if (table[index] == nullptr)

{

table[index] = newNode;

}

else

{

Node \*curr = table[index];

while (curr->next != nullptr)

{

curr = curr->next;

}

curr->next = newNode;

}

}

void remove(int key)

{

int index = hashFunc(key);

Node \*curr = table[index];

Node \*prev = nullptr;

while (curr != nullptr && curr->key != key)

{

prev = curr;

curr = curr->next;

}

if (curr == nullptr)

{

return;

}

if (prev == nullptr)

{

table[index] = curr->next;

}

else

{

prev->next = curr->next;

}

delete curr;

}

bool search(int key)

{

int index = hashFunc(key);

Node \*curr = table[index];

while (curr != nullptr)

{

if (curr->key == key)

{

return true;

}

curr = curr->next;

}

return false;

}

};

// Double-Hashing

class DoubleHashing

{

private:

int size;

int \*table;

bool \*isOccupied;

public:

DoubleHashing(int size)

: size(size)

{

table = new int[size];

isOccupied = new bool[size];

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

{

table[i] = -1;

isOccupied[i] = false;

}

}

int hashFunc1(int key)

{

return key % size;

}

int hashFunc2(int key)

{

return 3 - (key % 3);

}

void insert(int key)

{

int index = hashFunc1(key);

int step = hashFunc2(key);

while (isOccupied[index])

{

index = (index + step) % size;

}

table[index] = key;

isOccupied[index] = true;

}

void remove(int key)

{

int index = hashFunc1(key);

int step = hashFunc2(key);

while (table[index] != -1)

{

if (table[index] == key)

{

table[index] = -1;

isOccupied[index] = false;

return;

}

index = (index + step) % size;

}

}

bool search(int key)

{

int index = hashFunc1(key);

int step = hashFunc2(key);

while (table[index] != -1)

{

if (table[index] == key)

{

return true;

}

index = (index + step) % size;

}

return false;

}

};

// Bucketing

class Bucketing

{

private:

int size;

Node \*\*table;

public:

Bucketing(int size)

: size(size)

{

table = new Node \*[size];

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

{

table[i] = nullptr;

}

}

int hashFunc(int key)

{

return key % size;

}

void insert(int key)

{

int index = hashFunc(key);

Node \*newNode = new Node();

newNode->key = key;

newNode->next = nullptr;

if (table[index] == nullptr)

{

table[index] = newNode;

}

else

{

Node \*curr = table[index];

while (curr->next != nullptr)

{

curr = curr->next;

}

curr->next = newNode;

}

}

void remove(int key)

{

int index = hashFunc(key);

Node \*curr = table[index];

Node \*prev = nullptr;

while (curr != nullptr && curr->key != key)

{

prev = curr;

curr = curr->next;

}

if (curr == nullptr)

{

return;

}

if (prev == nullptr)

{

table[index] = curr->next;

}

else

{

prev->next = curr->next;

}

delete curr;

}

bool search(int key)

{

int index = hashFunc(key);

Node \*curr = table[index];

while (curr != nullptr)

{

if (curr->key == key)

{

return true;

}

curr = curr->next;

}

return false;

}

};

void searchAndPrint(int val, bool found)

{

cout << "Search result for " << val << ": ";

if (found)

{

cout << "Found" << endl;

}

else

{

cout << "Not Found" << endl;

}

}

int main()

{

// Rehashing

Rehashing reHash(10);

reHash.insert(5);

reHash.insert(15);

reHash.insert(25);

cout << "HashTable-Rehashing: " << endl;

searchAndPrint(5, reHash.search(5));

searchAndPrint(15, reHash.search(15));

searchAndPrint(25, reHash.search(25));

reHash.remove(15);

searchAndPrint(15, reHash.search(15));

// DoubleHashing

DoubleHashing doubleHash(10);

doubleHash.insert(5);

doubleHash.insert(15);

doubleHash.insert(25);

cout << "\nHashTable-DoubleHashing: " << endl;

searchAndPrint(5, doubleHash.search(5));

searchAndPrint(15, doubleHash.search(15));

searchAndPrint(25, doubleHash.search(25));

doubleHash.remove(15);

searchAndPrint(15, doubleHash.search(15));

// Bucketing

Bucketing Bucketing(10);

Bucketing.insert(5);

Bucketing.insert(15);

Bucketing.insert(25);

cout << "\nHashTable-Bucketing: " << endl;

searchAndPrint(5, Bucketing.search(5));

searchAndPrint(15, Bucketing.search(15));

searchAndPrint(25, Bucketing.search(25));

Bucketing.remove(15);

searchAndPrint(15, Bucketing.search(15));

return 0;

}

## Output

