Lab #14

Separate chaining:

#include <iostream>

using namespace std;

class HashTable

{

private:

int tableSize;

struct Node

{

int data;

Node\* next;

Node(int val)

{

data=val;

next = nullptr;

}

};

Node\*\* hashTable;

public:

HashTable(int size) : tableSize(size)

{

hashTable = new Node\*[tableSize];

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

{

hashTable[i] = nullptr;

}

}

// Hash function to convert a key into an index

int hashFunction(int key)

{

return key % tableSize;

}

// Insert an element into the hash table

void insert(int key)

{

int index = hashFunction(key);

Node\* newNode = new Node(key);

newNode->next = hashTable[index];

hashTable[index] = newNode;

}

// Delete an element from the hash table

bool remove(int key)

{

int index = hashFunction(key);

Node\* prev = nullptr;

Node\* curr = hashTable[index];

while (curr) {

if (curr->data == key) {

if (prev)

prev->next = curr->next;

else

hashTable[index] = curr->next;

delete curr;

return true;

}

prev = curr;

curr = curr->next;

}

return false;

}

// Search for an element in the hash table

bool search(int key)

{

int index = hashFunction(key);

Node\* curr = hashTable[index];

while (curr)

{

if (curr->data == key)

{

return true;

}

curr = curr->next;

}

return false;

}

~HashTable() {

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

Node\* curr = hashTable[i];

while (curr) {

Node\* temp = curr;

curr = curr->next;

delete temp;

}

}

delete[] hashTable;

}

};

int main() {

int tableSize;

cout << "Enter the size of the hash table: ";

cin >> tableSize;

HashTable obj(tableSize);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter the key to insert: ";

cin >> key;

obj.insert(key);

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

if (obj.remove(key)) {

cout << "Element " << key << " deleted successfully.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 3:

cout << "Enter the key to search: ";

cin >> key;

if (obj.search(key)) {

cout << "Element " << key << " found in the hash table.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

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

}

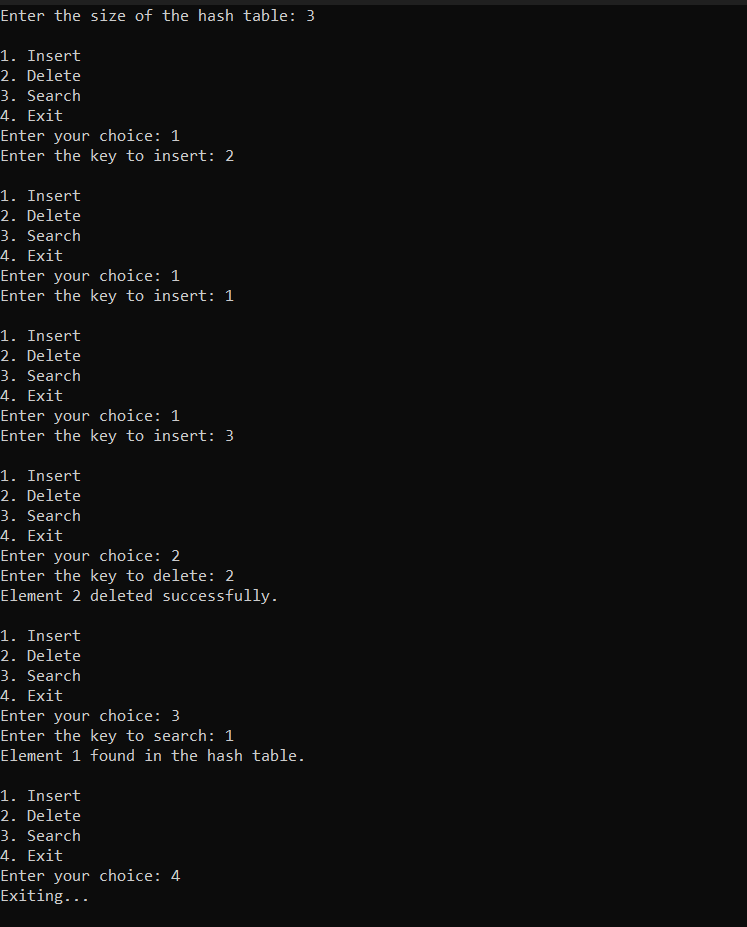
}

system("pause");

return 0;

}

Output:



Linear probing:

#include <iostream>

using namespace std;

class HashTable

{

private:

int tableSize;

int\* hashTable;

bool\* isOccupied;

public:

HashTable(int size) : tableSize(size)

{

hashTable = new int[tableSize];

isOccupied = new bool[tableSize];

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

{

hashTable[i] = -1;

isOccupied[i] = false;

}

}

// Hash function to convert a key into an index

int hashFunction(int key) {

return key % tableSize;

}

// Insert an element into the hash table using linear probing

void insert(int key)

{

int index = hashFunction(key);

while (isOccupied[index]) {

index = (index + 1) % tableSize;

}

hashTable[index] = key;

isOccupied[index] = true;

}

// Delete an element from the hash table

bool remove(int key) {

int index = hashFunction(key);

int initialIndex = index;

while (isOccupied[index]) {

if (hashTable[index] == key) {

hashTable[index] = -1;

isOccupied[index] = false;

return true;

}

index = (index + 1) % tableSize;

if (index == initialIndex) {

break; // Element not found

}

}

return false;

}

// Search for an element in the hash table

bool search(int key) {

int index = hashFunction(key);

int initialIndex = index;

while (isOccupied[index]) {

if (hashTable[index] == key)

{

return true;

}

index = (index + 1) % tableSize;

if (index == initialIndex) {

break; // Element not found

}

}

return false;

}

~HashTable() {

delete[] hashTable;

delete[] isOccupied;

}

};

int main() {

int tableSize;

cout << "Enter the size of the hash table: ";

cin >> tableSize;

HashTable hashTable(tableSize);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the key to insert: ";

cin >> key;

hashTable.insert(key);

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

if (hashTable.remove(key)) {

cout << "Element " << key << " deleted successfully.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 3:

cout << "Enter the key to search: ";

cin >> key;

if (hashTable.search(key)) {

cout << "Element " << key << " found in the hash table.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

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

}

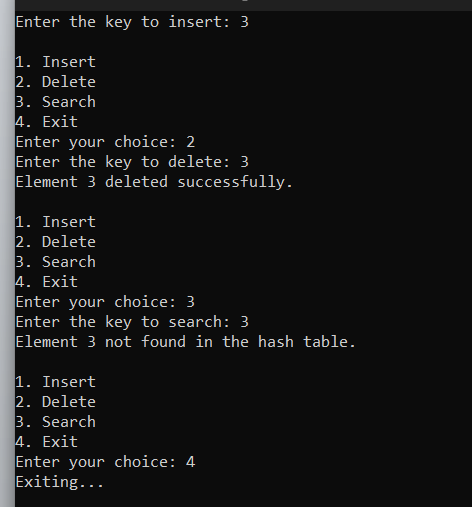
}

system("pause");

return 0;

}

Output:



Quadratic probing:

#include <iostream>

using namespace std;

class HashTable

{

private:

int tableSize;

int\* hashTable;

bool\* var;

public:

HashTable(int size) : tableSize(size)

{

hashTable = new int[tableSize];

var = new bool[tableSize];

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

hashTable[i] = -1;

var[i] = false;

}

}

// Hash function to convert a key into an index

int hashFunction(int key) {

return key % tableSize;

}

// Quadratic probing function

int quadraticProbing(int index, int i) {

return (index + i \* i) % tableSize;

}

// Insert an element into the hash table using quadratic probing

void insert(int key) {

int index = hashFunction(key);

int i = 0;

while (var[index]) {

i++;

index = quadraticProbing(index, i);

if (i == tableSize) {

cout << "Hash table is full. Cannot insert more elements.\n";

return;

}

}

hashTable[index] = key;

var[index] = true;

}

// Delete an element from the hash table

bool remove(int key) {

int index = hashFunction(key);

int i = 0;

while (var[index]) {

if (hashTable[index] == key) {

hashTable[index] = -1;

var[index] = false;

return true;

}

i++;

index = quadraticProbing(index, i);

}

return false;

}

// Search for an element in the hash table

bool search(int key) {

int index = hashFunction(key);

int i = 0;

while (var[index]) {

if (hashTable[index] == key) {

return true;

}

i++;

index = quadraticProbing(index, i);

}

return false;

}

~HashTable() {

delete[] hashTable;

delete[] var;

}

};

int main() {

int tableSize;

cout << "Enter the size of the hash table: ";

cin >> tableSize;

HashTable hashTable(tableSize);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the key to insert: ";

cin >> key;

hashTable.insert(key);

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

if (hashTable.remove(key)) {

cout << "Element " << key << " deleted successfully.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 3:

cout << "Enter the key to search: ";

cin >> key;

if (hashTable.search(key)) {

cout << "Element " << key << " found in the hash table.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

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

}

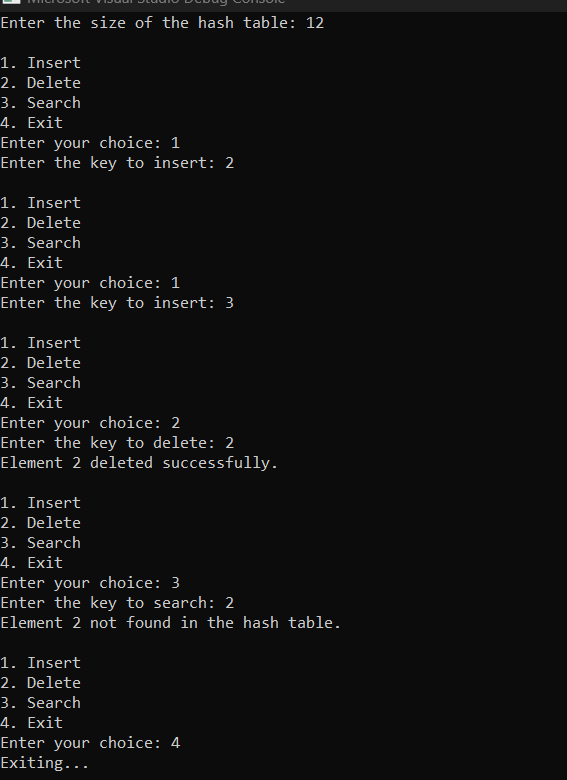
}

system("pause");

return 0;

}

Output:



Rehashing:

#include <iostream>

using namespace std;

class HashTable {

private:

int tableSize;

int\* hashTable;

bool\* isOccupied;

int numElements;

const double loadFactorThreshold = 0.7;

public:

HashTable(int size) : tableSize(size), numElements(0) {

hashTable = new int[tableSize];

isOccupied = new bool[tableSize];

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

hashTable[i] = -1;

isOccupied[i] = false;

}

}

// Hash function to convert a key into an index

int hashFunction(int key) {

return key % tableSize;

}

// Quadratic probing function

int quadraticProbing(int index, int i) {

return (index + i \* i) % tableSize;

}

// Insert an element into the hash table using quadratic probing and rehashing

void insert(int key) {

if (numElements >= loadFactorThreshold \* tableSize) {

rehash();

}

int index = hashFunction(key);

int i = 0;

while (isOccupied[index]) {

i++;

index = quadraticProbing(index, i);

}

hashTable[index] = key;

isOccupied[index] = true;

numElements++;

}

// Delete an element from the hash table

bool remove(int key) {

int index = hashFunction(key);

int i = 0;

while (isOccupied[index]) {

if (hashTable[index] == key) {

hashTable[index] = -1;

isOccupied[index] = false;

numElements--;

return true;

}

i++;

index = quadraticProbing(index, i);

}

return false;

}

// Search for an element in the hash table

bool search(int key) {

int index = hashFunction(key);

int i = 0;

while (isOccupied[index]) {

if (hashTable[index] == key) {

return true;

}

i++;

index = quadraticProbing(index, i);

}

return false;

}

// Rehashing function to resize the hash table

void rehash() {

int newSize = tableSize \* 2;

int\* newHashTable = new int[newSize];

bool\* newIsOccupied = new bool[newSize];

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

newHashTable[i] = -1;

newIsOccupied[i] = false;

}

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

if (isOccupied[i]) {

int key = hashTable[i];

int newIndex = key % newSize;

int j = 0;

while (newIsOccupied[newIndex]) {

j++;

newIndex = quadraticProbing(newIndex, j);

}

newHashTable[newIndex] = key;

newIsOccupied[newIndex] = true;

}

}

delete[] hashTable;

delete[] isOccupied;

tableSize = newSize;

hashTable = newHashTable;

isOccupied = newIsOccupied;

}

~HashTable() {

delete[] hashTable;

delete[] isOccupied;

}

};

int main() {

int tableSize;

cout << "Enter the size of the hash table: ";

cin >> tableSize;

HashTable obj(tableSize);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the key to insert: ";

cin >> key;

obj.insert(key);

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

if (obj.remove(key)) {

cout << "Element " << key << " deleted successfully.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 3:

cout << "Enter the key to search: ";

cin >> key;

if (obj.search(key)) {

cout << "Element " << key << " found in the hash table.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

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

}

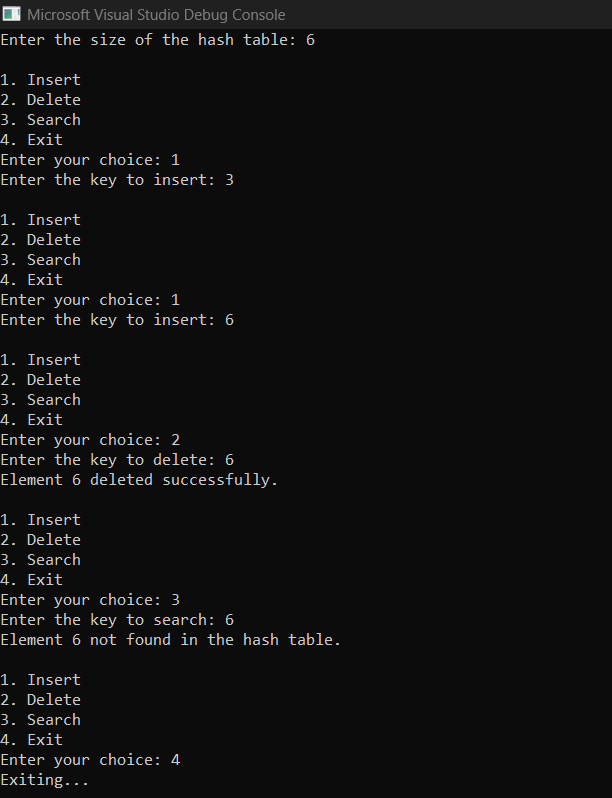
}

system("pause");

return 0;

}

Output:



Bucket:

#include <iostream>

using namespace std;

class Node {

public:

int key;

Node\* next;

Node(int key) : key(key), next(nullptr) {}

};

class HashTable

{

private:

int tableSize;

Node\*\* hashTable;

// Hash function to convert a key into an index

int hashFunction(int key) {

return key % tableSize;

}

public:

HashTable(int size) : tableSize(size) {

hashTable = new Node\*[tableSize];

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

hashTable[i] = nullptr;

}

}

// Insert an element into the hash table using the Bucket method

void insert(int key) {

int index = hashFunction(key);

Node\* newNode = new Node(key);

if (hashTable[index] == nullptr) {

hashTable[index] = newNode;

}

else {

Node\* curr = hashTable[index];

while (curr->next != nullptr) {

curr = curr->next;

}

curr->next = newNode;

}

}

// Delete an element from the hash table

bool remove(int key) {

int index = hashFunction(key);

Node\* curr = hashTable[index];

Node\* prev = nullptr;

while (curr != nullptr) {

if (curr->key == key) {

if (prev == nullptr) {

hashTable[index] = curr->next;

}

else {

prev->next = curr->next;

}

delete curr;

return true;

}

prev = curr;

curr = curr->next;

}

return false;

}

// Search for an element in the hash table

bool search(int key) {

int index = hashFunction(key);

Node\* curr = hashTable[index];

while (curr != nullptr) {

if (curr->key == key) {

return true;

}

curr = curr->next;

}

return false;

}

// Destructor to deallocate memory

~HashTable() {

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

Node\* curr = hashTable[i];

while (curr != nullptr) {

Node\* next = curr->next;

delete curr;

curr = next;

}

}

delete[] hashTable;

}

};

int main() {

int tableSize;

cout << "Enter the size of the hash table: ";

cin >> tableSize;

HashTable hashTable(tableSize);

int choice, key;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the key to insert: ";

cin >> key;

hashTable.insert(key);

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

if (hashTable.remove(key)) {

cout << "Element " << key << " deleted successfully.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 3:

cout << "Enter the key to search: ";

cin >> key;

if (hashTable.search(key)) {

cout << "Element " << key << " found in the hash table.\n";

}

else {

cout << "Element " << key << " not found in the hash table.\n";

}

break;

case 4:

cout << "Exiting...\n";

return 0;

default:

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

}

}

system("pause");

return 0;

}

Output:

