**Zubair Ali**

**22P-9232**

**BCS-3C**

**Data Structures Assignment#4**

Task#1:

Output:

**Task#2:**

#include<iostream>

using namespace std;

// Define a class for Linear Hashing

class LinearHash{

public:

int size; // Size of the hash table

int \*arr; // Pointer to the array that will store the hash table

// Constructor to initialize the hash table

LinearHash(int s)

{

arr=new int[size]; // Allocate memory for the hash table

size=s; // Set the size of the hash table

for(int i=0;i<size;i++) // Initialize all elements of the hash table to -2

{

arr[i]=-2;

}

}

// Hash function to calculate the index for a given key

int HashFunction (int key)

{

int x = (key + 7) \* (key + 7);

x = x / 16;

x = x + key;

x = x % 11;

return x;

}

// Function to insert a key into the hash table

void insert(int key)

{

cout<<"\n "<<key;

int ind=HashFunction(key); // Calculate the index for the key

cout<<" "<<ind;

if(arr[ind]!=-2) // If the calculated index is already occupied

{

cout<<" ";

while(arr[ind]!=-2) // Find the next available index

{

ind=(ind+1)%size;

cout<<ind<<",";

}

}

arr[ind]=key; // Insert the key at the found index

}

// Function to search for a key in the hash table

int search(int key)

{

int ind=HashFunction(key); // Calculate the index for the key

if(arr[ind]!=key) // If the key is not found at the calculated index

{

while(arr[ind]!=-2) // Find the next index

{

ind=(ind+1)%size;

}

}

if(arr[ind]==key) // If the key is found

{

cout<<"\nValue "<<key<<" found at index "<<ind;

}

else // If the key is not found

{

cout<<"\nValue "<<key<<" Not found!";

}

}

// Function to delete a key from the hash table

void deleteKey( int key)

{

int index = HashFunction(key); // Calculate the index for the key

while (arr[index] != key) // Find the index where the key is stored

{

index = (index + 1) % size;

}

if(arr[index]==key) // If the key is found

{

cout<<"\nValue Deleted!";

arr[index]=-2; // Delete the key

}

else // If the key is not found

{

cout<<"\nValue "<<key<<" Not found!";

}

}

// Function to print the hash table

void Print()

{

cout<<"\n\nFinal Hash Table:";

cout<<"\nSlot:";

for(int i=0;i<size;i++) // Print the index

{

cout<<" "<<i;

}

cout<<endl<<"Key ";

for(int i=0;i<size;i++) // Print the keys

{

if(arr[i]==-2) // If the index is empty

{

cout<<" \_";

}

else // If the index is occupied

{

cout<<" "<<arr[i];

}

}

}

};

// Main function

int main()

{

cout<<"\nKey Value Initial Slot Probe Sequence";

LinearHash h1(11); // Create a hash table of size 11

h1.insert(43); // Insert keys into the hash table

h1.insert(23);

h1.insert(1);

h1.insert(0);

h1.insert(15);

h1.insert(31);

h1.insert(4);

h1.insert(7);

h1.insert(11);

h1.insert(3);

h1.Print(); // Print the final hash table

}

**Output:**

A screenshot of a computer

Description automatically generated

**Task#3:**

**1.Linear Probing:**

#include<iostream>

using namespace std;

class LinearHash{

public:

int size; // Size of the hash table

int \*arr; // Pointer to the array that will store the hash table

float count;

// Constructor to initialize the hash table

LinearHash(int s)

{

size = s;

arr = new int[size];

count = 0;

// Initialize all elements of the hash table to -2

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

{

arr[i] = -2;

}

}

// Hash function to calculate the index for a given key

int HashFunction(int key)

{

int x = (key + 7) \* (key + 7);

x = x / 16;

x = x + key;

x = x % size;

return x;

}

// Function to insert a key into the hash table

void insert(int key)

{

count++;

float lf = count / size;

lf = lf \* 100;

// If the load factor is greater than or equal to 70, rehash the table

if (lf >= 70)

{

cout << "\nLoad factor " << lf;

reHash();

}

cout << "\n " << key;

int ind = HashFunction(key);

cout << " " << ind;

// If the calculated index is already occupied

if (arr[ind] != -2)

{

cout << " ";

// Find the next available index

while (arr[ind] != -2)

{

ind = (ind + 1) % size;

cout << ind << ",";

}

}

// Insert the key at the found index

arr[ind] = key;

}

// Function to search for a key in the hash table

int search(int key)

{

int ind = HashFunction(key);

// If the key is not found at the calculated index

if (arr[ind] != key)

{

// Find the next index

while (arr[ind] != -2)

{

ind = (ind + 1) % size;

}

}

// If the key is found

if (arr[ind] == key)

{

cout << "\nValue " << key << " found at index " << ind;

}

else // If the key is not found

{

cout << "\nValue " << key << " Not found!";

}

}

// Function to delete a key from the hash table

void deleteKey(int key)

{

int index = HashFunction(key);

// Find the index where the key is stored

while (arr[index] != key)

{

index = (index + 1) % size;

}

// If the key is found

if (arr[index] == key)

{

cout << "\nValue Deleted!";

arr[index] = -2;

}

else // If the key is not found

{

cout << "\nValue " << key << " Not found!";

}

}

// Function to print the hash table

void Print()

{

cout << "\nHash table:\n";

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

{

if (arr[i] != -2)

{

cout << " " << i << " | " << arr[i];

cout << endl;

}

else

{

cout << " " << i << " | " << "Empty";

cout << endl;

}

}

}

// Function to rehash the table when the load factor is too high

// Function to rehash the table when the load factor is too high

void reHash()

{

// Reset the count of elements in the hash table

count = 0;

// Print a message indicating the start of rehashing

cout << "\n Rehashing Started";

// Declare a pointer for the new array

int \*newArray;

// Declare a boolean variable to check if a number is prime

bool result = true;

// Calculate the new size of the hash table, which is twice the current size

int newsize = size \* 2;

// Find the next prime number after the new size

for (int i = newsize + 1; i < newsize \* 2; i++)

{

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

{

// If the number is divisible by any number other than 1 and itself, it is not prime

if (i % j == 0)

{

result = false;

}

}

// If the number is prime, set the new size to this number and break the loop

if (result == true)

{

newsize = i;

cout << "\n\n" << i;

break;

}

result = true;

}

// Create a new array to store the old hash table

int \*old = new int[size];

int oldsize = size;

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

{

old[i] = arr[i];

}

// Set the size of the hash table to the new size

size = newsize;

// Allocate memory for the new array

newArray = new int[newsize];

// Set the pointer of the hash table to the new array

arr = newArray;

// Initialize all elements of the new hash table to -2

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

{

arr[i] = -2;

}

// Reinsert the keys from the old hash table into the new hash table

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

{

if (old[i] != -2)

{

insert(old[i]);

}

}

// Delete the old array

delete[] old;

}

};

// Main function

int main() {

// Print the header for the output

cout << "\nKey Value Initial Slot Probe Sequence";

// Create a hash table with an initial size of 15

LinearHash h1(15);

// Insert keys into the hash table

h1.insert(17);

h1.insert(26);

h1.insert(15);

h1.insert(9);

h1.insert(11);

h1.insert(43);

h1.insert(75);

h1.insert(19);

h1.insert(35);

h1.insert(45);

h1.insert(55);

h1.insert(9);

h1.insert(10);

h1.insert(21);

h1.insert(61);

h1.insert(23);

// Print the hash table

h1.Print();

return 0;

}

**Output:**

A screenshot of a computer

Description automatically generated

**Quadratic Probing:**

#include<iostream>

using namespace std;

class QuadraticHash {

public:

int size; // Size of the hash table

int \*arr; // Pointer to the array that stores the hash table

float count; // Number of elements in the hash table

// Constructor

QuadraticHash(int size) {

count = 0; // Initialize count to 0

this->size = size; // Set the size of the hash table

arr = new int[size]; // Allocate memory for the hash table

for (int i = 0; i < size; i++) { // Initialize all elements to -2

arr[i] = -2;

}

}

// Hash function using quadratic probing

int HashFunction(int key) {

int x = (key + 7) \* (key + 7); // Quadratic function

x = x / 16; // Divide by 16

x = x + key; // Add the key

x = x % size; // Modulo the size of the hash table

return x; // Return the hash value

}

// Insert a key into the hash table

void insert(int key) {

count++; // Increment the count

float lf = count / size \* 100; // Calculate the load factor

// Check load factor and rehash if needed

if (lf >= 70) {

cout << "\nLoad factor " << lf;

reHash(); // Rehash the table

}

cout << "\n " << key;

int ind = HashFunction(key); // Calculate the hash value

int hval = ind;

int i = 1;

cout << " " << ind;

// Handle collisions using quadratic probing

if (arr[ind] != -2) {

cout<<" ";

while (arr[ind] != -2) {

ind = (hval + i \* i) % size; // Quadratic probing

i++;

cout << ind << ",";

}

}

arr[ind] = key; // Insert the key

}

// Search for a key in the hash table

void search(int key) {

int ind = HashFunction(key); // Calculate the hash value

int hval = ind;

int i = 1;

// Handle collisions using quadratic probing

if (arr[ind] != key) {

while (arr[ind] != -2) {

ind = (hval + i \* i) % size; // Quadratic probing

i++;

}

}

// Check if the key is found

if (arr[ind] == key) {

cout << "\nValue " << key << " found at index " << ind;

} else {

cout << "\nValue " << key << " Not found!";

}

}

// Delete a key from the hash table

void deleteKey(int key) {

int index = HashFunction(key); // Calculate the hash value

int hval = index;

int i = 1;

// Find the key using quadratic probing

while (arr[index] != key) {

index = (hval + i \* i) % size; // Quadratic probing

i++;

}

// Check if the key is found and delete it

if (arr[index] == key) {

cout << "\nValue Deleted!";

arr[index] = -2; // Mark the slot as empty

} else {

cout << "\nValue " << key << " Not found!";

}

}

// Print the hash table

void Print() {

cout << "\nHash table:\n";

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

if (arr[i] != -2) { // If the slot is not empty

cout << " " << i << " | " << arr[i];

cout << endl;

} else { // If the slot is empty

cout << " " << i << " | " << "Empty";

cout << endl;

}

}

}

// Rehash the hash table

void reHash() {

count = 0; // Reset the count

cout << "\n Rehashing Started";

int \*newArray;

bool result = true;

// Find the next prime number after twice the current size

int newsize = size \* 2;

for (int i = newsize + 1; i < newsize \* 2; i++) {

for (int j = 2; j < i; j++) {

if (i % j == 0) {

result = false;

}

}

if (result == true) {

newsize = i;

cout << "\n\n" << i;

break;

}

result = true;

}

// Store the old hash table

int \*old = new int[size];

int oldsize = size;

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

old[i] = arr[i];

}

// Create a new hash table with the new size

size = newsize;

newArray = new int[newsize];

arr = newArray;

// Initialize all elements to -2

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

arr[i] = -2;

}

// Reinsert the keys from the old hash table into the new hash table

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

if (old[i] != -2) {

insert(old[i]);

}

}

// Delete the old hash table

delete[] old;

}

};

int main() {

cout << "\nKey Value Initial Slot Probe Sequence";

QuadraticHash h1(15); // Create a hash table with an initial size of 15

h1.insert(17);

h1.insert(26);

h1.insert(15);

h1.insert(9);

h1.insert(11);

h1.insert(43);

h1.insert(75);

h1.insert(19);

h1.insert(35);

h1.insert(45);

h1.insert(55);

h1.insert(9);

h1.insert(10);

h1.insert(21);

h1.insert(61);

h1.insert(23); // Insert keys into the hash table

h1.Print(); // Print the hash table

// Example of searching for a key

//h1.search(45); // Search for the key 45

return 0;

}

**Output:**

A screenshot of a computer

Description automatically generated

**Double Hashing:**

#include<iostream>

using namespace std;

class DoubleHash{

public:

int size;

int \*arr;

float count;

// Constructor

DoubleHash(int size)

{

count = 0; // Initialize count to 0

this->size = size; // Set the size of the hash table

arr = new int[size]; // Allocate memory for the hash table

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

{ // Initialize all elements to -2

arr[i] = -2;

}

}

// First hash function

int firstHash(int key)

{

int x = (key + 7) \* (key + 7); // Quadratic function

x = x / 16; // Divide by 16

x = x + key; // Add the key

x = x % size; // Modulo the size of the hash table

return x; // Return the hash value

}

// Second hash function

int SecondHash(int key)

{

return 1 + key % (size - 2); // Return the hash value

}

// Insert a key into the hash table

void insert(int key)

{

count++; // Increment the count

float lf = count / size \* 100; // Calculate the load factor

// Check load factor and rehash if needed

if (lf >= 70)

{

cout << "\nLoad factor " << lf;

reHash(); // Rehash the table

}

cout << "\n " << key;

int ind = firstHash(key); // Calculate the first hash value

int h1 = ind;

int h2 = SecondHash(key); // Calculate the second hash value

int i = 1;

cout << " " << ind;

// Handle collisions using double hashing

if (arr[ind] != -2)

{

cout << " ";

while (arr[ind] != -2)

{

ind = (h1 + i \* h2) % size; // Double hashing

i++;

cout << ind << ",";

}

}

arr[ind] = key; // Insert the key

}

// Search for a key in the hash table

int search(int key)

{

int ind = firstHash(key); // Calculate the first hash value

int h1 = ind;

int h2 = SecondHash(key); // Calculate the second hash value

int i = 1;

// Handle collisions using double hashing

if (arr[ind] != key)

{

cout << " ";

while (arr[ind] != key)

{

ind = (h1 + i \* h2) % size; // Double hashing

i++;

}

}

// Check if the key is found

if (arr[ind] == key)

{

cout << "\nValue " << key << " found at index " << ind;

}

else

{

cout << "\nValue " << key << " Not found!";

}

}

// Delete a key from the hash table

void deleteKey(int key)

{

int ind = firstHash(key); // Calculate the first hash value

int h1 = ind;

int h2 = SecondHash(key); // Calculate the second hash value

int i = 1;

// Find the key using double hashing

if (arr[ind] != key)

{

while (arr[ind] != key)

{

ind = (h1 + i \* h2) % size; // Double hashing

i++;

}

}

// Check if the key is found and delete it

if (arr[ind] == key)

{

cout << "\nValue Deleted!";

arr[ind] = -2; // Mark the slot as empty

}

else

{

cout << "\nValue " << key << " Not found!";

}

arr[ind] = -2; // Mark the slot as empty

}

// Print the hash table

void Print()

{

cout << "\nHash table:\n";

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

{

if (arr[i] != -2)

{ // If the slot is not empty

cout << " " << i << " | " << arr[i];

cout << endl;

}

else

{ // If the slot is empty

cout << " " << i << " | " << "Empty";

cout << endl;

}

}

}

// Rehash the hash table

void reHash()

{

count = 0; // Reset the count

cout << "\n Rehashing Started";

int \*newArray;

bool result = true;

// Find a new size for the hash table, which is a prime number

int newsize = size \* 2;

for (int i = newsize + 1; i < newsize \* 2; i++)

{

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

{

if (i % j == 0)

{

result = false;

}

}

if (result == true)

{

newsize = i;

cout << "\n\n" << i;

break;

}

result = true;

}

// Save the old hash table

int \*old = new int[size];

int oldsize = size;

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

{

old[i] = arr[i];

}

// Create a new hash table with the new size

size = newsize;

newArray = new int[newsize];

arr = newArray;

// Initialize the new hash table

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

{

arr[i] = -2;

}

// Reinsert the keys from the old hash table to the new hash table

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

{

if (old[i] != -2)

{

insert(old[i]);

}

}

delete[] old; // Delete the old hash table

}

};

int main()

{

DoubleHash h1(15); // Create a hash table with an initial size of 15

cout << "\nKey Value Initial Slot Probe Sequence";

// Insert keys into the hash table

h1.insert(17);

h1.insert(26);

h1.insert(15);

h1.insert(9);

h1.insert(11);

h1.insert(43);

h1.insert(75);

h1.insert(19);

h1.insert(35);

h1.insert(45);

h1.insert(55);

h1.insert(9);

h1.insert(10);

h1.insert(21);

h1.insert(61);

h1.insert(23);

h1.Print(); // Print the hash table

// Example of searching for a key

// h1.search(45); // Search for the key 45

return 0;

}

**Output:**

A screenshot of a computer

Description automatically generated

**Task #4:**

**Bucketing:**

#include<iostream>

using namespace std;

class Bucket{

public:

int \*\*arr;

int size;

float count;

// Constructor

Bucket(int s)

{

count=0;

size=s;

arr=new int\*[3]; // Initialize the hash table with 3 buckets

for(int i=0;i<3;i++)

{

arr[i]=new int[size]; // Initialize each bucket with the given size

}

for(int i=0;i<3;i++)

{

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

{

arr[i][j]=-2; // Initialize all slots as empty

}

}

}

// Hash function

int HashFunc(int key)

{

return key%3; // Return the remainder of the key divided by 3

}

// Function to insert a key into the hash table

void insert(int key)

{

count++; // Increase the count of keys

float bucketsize =3\*size; // Calculate the total number of slots

float lf=count/bucketsize; // Calculate the load factor

lf=lf\*100; // Convert the load factor to a percentage

if(lf>=70) // If the load factor is greater than or equal to 70%

{

Print(); // Print the hash table

//cout<<"count:"<<count;

cout<<"\nLoad factor "<<lf;

reHash(); // Rehash the hash table

}

int h=HashFunc(key); // Calculate the hash value of the key

bool result=false;

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

{

if(arr[h][i]==-2) // If the slot is empty

{

arr[h][i]=key; // Insert the key into the slot

result=true;

break;

}

}

}

// Function to rehash the hash table

void reHash()

{

count=0;

cout<<"\n Rehashing Started";

int \*\*newArray=new int\*[3]; // Create a new hash table

bool result = true;

int newsize = size \* 2; // Double the size of the hash table

for(int i=0;i<3;i++)

{

newArray[i]=new int[newsize]; // Initialize the new hash table

}

cout<<newsize;

int \*\*old=new int\*[3]; // Create a temporary hash table to store the old keys

int oldsize = size;

for(int i=0;i<3;i++)

{

old[i]=new int[size];

}

for (int i = 0; i < 3; i++)

{

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

{

old[i][j]=arr[i][j]; // Copy the keys from the old hash table to the temporary hash table

}

}

size = newsize;

arr = newArray; // Set the new hash table as the current hash table

for(int i=0;i<3;i++)

{

for(int j=0;j<newsize;j++)

{

arr[i][j]=-2; // Initialize all slots as empty

}

}

for (int i = 0; i < 3; i++)

{

for(int j=0;j<oldsize;j++)

{

if (old[i][j] != -2) // If the slot is not empty

{

insert(old[i][j]); // Insert the key into the new hash table

}

}

}

for(int i=0;i<3;i++)

{

delete []old[i]; // Delete the temporary hash table

}

delete[] old;

}

// Function to search for a key in the hash table

void search(int key)

{

int h=HashFunc(key); // Calculate the hash value of the key

bool result=false;

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

{

if(arr[h][i]==key) // If the key is found

{

result=true;

cout<<"\nValue "<<key <<" found!";

break;

}

}

if(result != true)

{

cout<<"\n Value not found";

}

}

// Function to delete a key from the hash table

void deleteKey(int key)

{

int h=HashFunc(key); // Calculate the hash value of the key

bool result=false;

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

{

if(arr[h][i]==key) // If the key is found

{

arr[h][i]=-2; // Delete the key

result=true;

cout<<"\n\nValue "<<key <<" found and deleted";

break;

}

if(result!=true)

{

cout<<"\n Value not found";

}

}

}

// Function to print the hash table

void Print()

{

for(int i=0;i<3;i++)

{

cout<<endl;

cout<<"Bucket "<<i<<": ";

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

{

if(arr[i][j]==-2) // If the slot is empty

{

cout<<"Empty ";

}

else

{

cout<<arr[i][j]<<" "; // Print the key

}

}

}

}

};

int main()

{

Bucket h1(5); // Create a hash table with 5 slots per bucket

// Insert keys into the hash table

h1.insert(17);

h1.insert(26);

h1.insert(15);

h1.insert(9);

h1.insert(11);

h1.insert(43);

h1.insert(75);

h1.insert(19);

h1.insert(35);

h1.insert(45);

h1.insert(55);

h1.insert(9);

h1.insert(10);

h1.insert(21);

h1.insert(61);

h1.insert(23);

h1.Print(); // Print the hash table

return 0;

}

**Output:**

A black background with white text

Description automatically generated

**Chaining:**

#include "iostream"

using namespace std ;

class Node{

public:

int key;

Node\* next;

Node(int k)

{

key=k;

next=NULL;

}

};

//remove the errors if any

class Chain {

public:

Node\* arr[3];

int size;

// Constructor

Chain()

{

for(int i=0;i<3;i++)

{

arr[i]=NULL; // Initialize all lists as empty

}

}

// Hash function

int Hash(int key)

{

return key%3; // Return the remainder of the key divided by 3

}

// Function to insert a key into the hash table

void insert(int key)

{

int h=Hash(key); // Calculate the hash value of the key

Node\* newnode=new Node(key); // Create a new node with the key

Node\* temp=arr[h]; // Get the head of the linked list for the hash value

if(temp==NULL) // If the list is empty

{

arr[h]=newnode; // Insert the node at the head of the list

}

else // If the list is not empty

{

while(temp->next!=NULL) // Traverse to the end of the list

{

temp=temp->next;

}

temp->next=newnode; // Insert the node at the end of the list

}

}

// Function to print the hash table

void print()

{

for(int i=0;i<3;i++)

{

cout<<"\nBucket "<<i<<":"; // Print the bucket number

Node\* temp=arr[i]; // Get the head of the linked list for the bucket

while(temp!=NULL) // Traverse the list

{

cout<<"->"<<temp->key; // Print the key

temp=temp->next; // Go to the next node

}

}

}

// Function to delete a key from the hash table

void deletion(int key)

{

int h=Hash(key); // Calculate the hash value of the key

Node\* temp=arr[h]; // Get the head of the linked list for the hash value

Node\* prev=temp; // Node to keep track of the previous node

bool result=false;

if(arr[h]->key==key) // If the key is at the head of the list

{

arr[h]=arr[h]->next; // Delete the head of the list

cout<<"\nValue Deleted";

return;

}

while(temp!=NULL) // Traverse the list

{

if(temp->key==key) // If the key is found

{

result=true;

break;

}

prev=temp; // Keep track of the previous node

temp=temp->next; // Go to the next node

}

if(result==true) // If the key was found

{

cout<<"\nValue Deleted";

prev->next=temp->next; // Delete the node

}

else // If the key was not found

{

cout<<"\nValue not found";

}

}

// Function to search for a key in the hash table

void Search(int key)

{

int h=Hash(key); // Calculate the hash value of the key

Node\* temp=arr[h]; // Get the head of the linked list for the hash value

bool result=false;

if(arr[h]->key==key) // If the key is at the head of the list

{

cout<<"\nValue Found";

return;

}

while(temp!=NULL) // Traverse the list

{

if(temp->key==key) // If the key is found

{

result=true;

break;

}

temp=temp->next; // Go to the next node

}

if(result==true) // If the key was found

{

cout<<"\nValue found";

}

else // If the key was not found

{

cout<<"\nValue not found";

}

}

};

int main()

{

Chain h1; // Create a hash table

// Insert keys into the hash table

h1.insert(17); // Insert the key 17

h1.insert(26); // Insert the key 26

h1.insert(15); // Insert the key 15

h1.insert(9); // Insert the key 9

h1.insert(11); // Insert the key 11

h1.insert(43); // Insert the key 43

h1.insert(75); // Insert the key 75

h1.insert(19); // Insert the key 19

h1.insert(35); // Insert the key 35

h1.insert(45); // Insert the key 45

h1.insert(55); // Insert the key 55

h1.insert(9); // Insert the key 9

h1.insert(10); // Insert the key 10

h1.insert(21); // Insert the key 21

h1.insert(61); // Insert the key 61

h1.insert(23); // Insert the key 23

h1.print(); // Print the hash table

h1.deletion(17); // Delete the key 17 from the hash table

h1.print(); // Print the hash table again to check if the key has been deleted

h1.Search(45); // Search for the key 45 in the hash table

return 0;

}

**Output:**

A black background with white text

Description automatically generated