Q-1: Write a CPP program to implement hashing with chaining.

Sample test case:

|  |
| --- |
| Input: arr= {15, 11, 27, 8, 12}  bucket=7  Output:  0  1 --> 15 --> 8  2  3  4 --> 11  5  6 --> 27 |

Solution:

#include<bits/stdc++.h>

using namespace std;

class Hash

{

int BUCKET; // No. of buckets

// Pointer to an array containing buckets

list<int> \*table;

public:

Hash(int V); // Constructor

// inserts a key into hash table

void insertItem(int x);

// deletes a key from hash table

void deleteItem(int key);

// hash function to map values to key

int hashFunction(int x)

{

return (x % BUCKET);

}

void displayHash();

};

Hash::Hash(int b)

{

this->BUCKET = b;

table = new list<int>[BUCKET];

}

void Hash::insertItem(int key)

{

int index = hashFunction(key);

table[index].push\_back(key);

}

void Hash::deleteItem(int key)

{

// get the hash index of key

int index = hashFunction(key);

// find the key in (index)th list

list <int> :: iterator i;

for (i = table[index].begin();

i != table[index].end(); i++)

{

if (\*i == key)

break;

}

// if key is found in hash table, remove it

if (i != table[index].end())

table[index].erase(i);

}

// function to display hash table

void Hash::displayHash()

{

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

{

cout << i;

for (auto x : table[i])

cout << " --> " << x;

cout << endl;

}

}

int main()

{

// array that contains keys to be mapped

int a[] = {15, 11, 27, 8, 12};

int n = sizeof(a)/sizeof(a[0]);

// insert the keys into the hash table

Hash h(7); // 7 is count of buckets in

// hash table

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

h.insertItem(a[i]);

// delete 12 from hash table

h.deleteItem(12);

// display the Hash table

h.displayHash();

return 0;

}

Q-2: Given an array of n non-negative integers and a series of queries, each defined by a range within the array, the task is to determine the frequency of a specific element within each specified range. The provided ranges are given in terms of positions (not 0-based indexes) within the array. The goal is to process multiple queries of this type and calculate the frequency of the specified element in each range.

Sample test case:

|  |
| --- |
| Input : arr[] = {2, 8, 6, 9, 8, 6, 8, 2, 11};  left = 2, right = 8, element = 8  left = 2, right = 5, element = 6  Output : 3  1 |

Solution:

#include<bits/stdc++.h>

using namespace std;

unordered\_map< int, vector<int> > store;

// Returns frequency of element in arr[left-1..right-1]

int findFrequency(int arr[], int n, int left,

int right, int element)

{

// Find the position of first occurrence of element

int a = lower\_bound(store[element].begin(),

store[element].end(),

left)

- store[element].begin();

// Find the position of last occurrence of element

int b = upper\_bound(store[element].begin(),

store[element].end(),

right)

- store[element].begin();

return b-a;

}

int main()

{

int arr[] = {2, 8, 6, 9, 8, 6, 8, 2, 11};

int n = sizeof(arr) / sizeof(arr[0]);

// Storing the indexes of an element in the map

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

store[arr[i]].push\_back(i+1); //starting index from 1

// Print frequency of 2 from position 1 to 6

cout << "Frequency of 2 from 1 to 6 = "

<< findFrequency(arr, n, 1, 6, 2) <<endl;

// Print frequency of 8 from position 4 to 9

cout << "Frequency of 8 from 4 to 9 = "

<< findFrequency(arr, n, 4, 9, 8);

return 0;

}

Q-3: Write a CPP program for handling of collision via open addressing Method-Double Hashing.

Sample test case:

|  |
| --- |
| Input: values={19, 27, 36, 10, 64}, table\_size=11  Output:  Enter a key that you want to search 36  36 found  The hash table looks like:  0  1  2  3 --> 36  4  5 --> 27  6  7  8 --> 19  9 --> 64  10 --> 10 |

Solution:

#include <bits/stdc++.h>

using namespace std;

// Predefining the size of the hash table.

#define SIZE\_OF\_TABLE 11

// Used for the second hash table.

// 8 and 11 are coprime numbers.

#define CO\_PRIME 8

// Defining the hashTable vector.

vector<int> hashTable(SIZE\_OF\_TABLE);

class DoubleHash

{

int size;

public:

// To check whether the table is full or not.

bool isFull()

{

// In case the table becomes full.

return (size == SIZE\_OF\_TABLE);

}

// Calculating the first hash.

int hash1(int key)

{

return (key % SIZE\_OF\_TABLE);

}

// Calculating the second hash.

int hash2(int key)

{

return (CO\_PRIME - (key % CO\_PRIME));

}

DoubleHash()

{

size = 0;

for (int i = 0; i < SIZE\_OF\_TABLE; i++)

hashTable[i] = -1;

}

// Function for inserting a key into the hash table.

void insertHash(int key)

{

// We first check whether the hash is full or not.

if (isFull())

return;

// Obtaining the index from the first hash table.

int index = hash1(key);

// In case a collision occurs.

if (hashTable[index] != -1)

{

// Obtaining the index from second hash table.

int index2 = hash2(key);

int i = 1;

while (1)

{

// Obtaining the new index.

int newIndex = (index + i \* index2) % SIZE\_OF\_TABLE;

//If no collision occurs, the key is stored.

if (hashTable[newIndex] == -1)

{

hashTable[newIndex] = key;

break;

}

i++;

}

}

//If no collision occurs, the key is stored.

else

hashTable[index] = key;

size++;

}

// For searching a key in the hash table.

void search(int key)

{

int i1 = hash1(key);

int i2 = hash2(key);

int i = 0;

while (hashTable[(i1 + i \* i2) % SIZE\_OF\_TABLE] != key)

{

if (hashTable[(i1 + i \* i2) % SIZE\_OF\_TABLE] == -1)

{

cout << key << " does not exist" << endl;

return;

}

i++;

}

cout << key << " found" << endl;

}

// For displaying the complete hash table.

void displayHash()

{

for (int i = 0; i < SIZE\_OF\_TABLE; i++)

{

if (hashTable[i] != -1)

cout << i << " --> "

<< hashTable[i] << endl;

else

cout << i << endl;

}

}

};

int main()

{

// Assuming the number of initial values to be 5.

int n = 5, i, k;

int a[100];

// We first need to insert keys into the table.

DoubleHash hash;

cout << "Enter 5 values: \n";

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

{

cin >> a[i];

hash.insertHash(a[i]);

}

// First, we search for a key that is present in the table.

cout << "Enter a key that you want to search \n";

cin >> k;

hash.search(k);

// Lets display the hash table.

// Since we took the SIZE\_OF\_TABLE as 11 we will get the indices from 0-10.

cout << "\n The hash table looks like:\n";

hash.displayHash();

return 0;

}