# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 11 : Hash Tables**

**Date: 21th December, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# 

# Lab 11 : Hash Tables

**Introduction**

This lab is based on the hashing with separate chaining and hashing with open addressing using linear probing , Quadratic probing and double hashing.

**Objectives**

Objective of this lab is to make students familiar with hash tables.

**Tools/Software Requirement**

Visual Studio 2012 or gcc or g++

**Tasks**

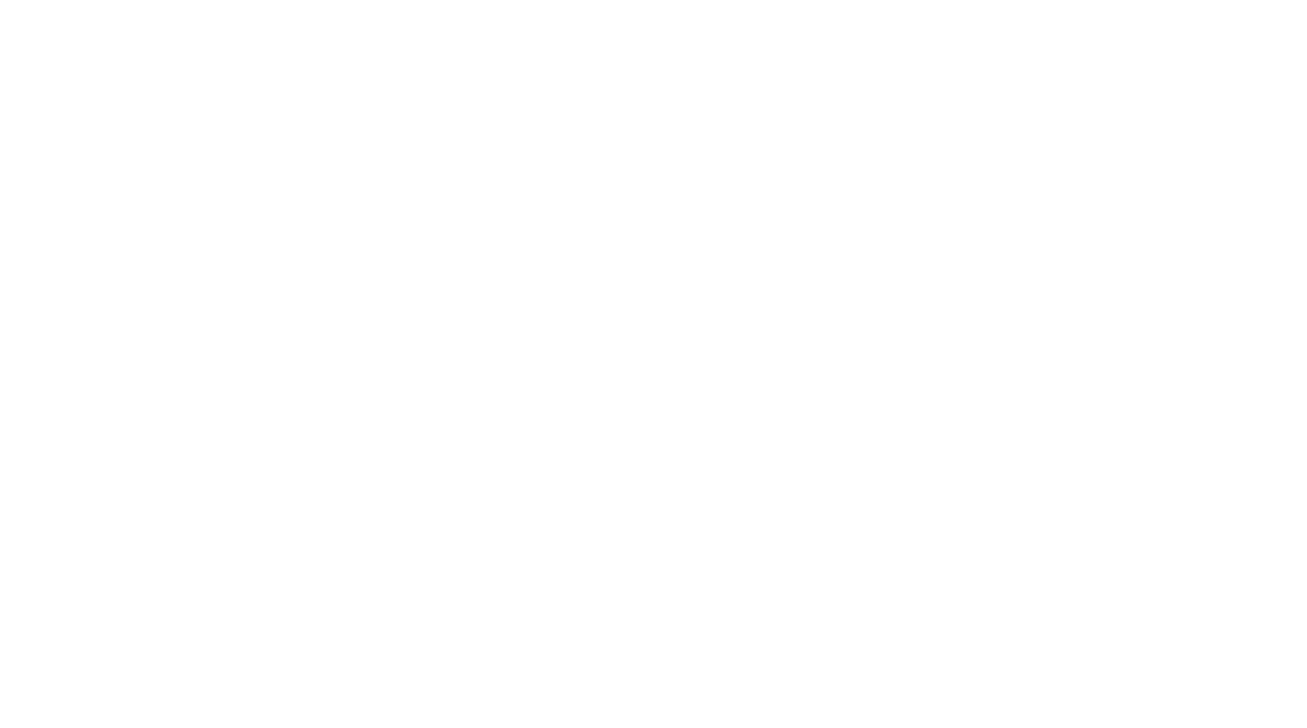
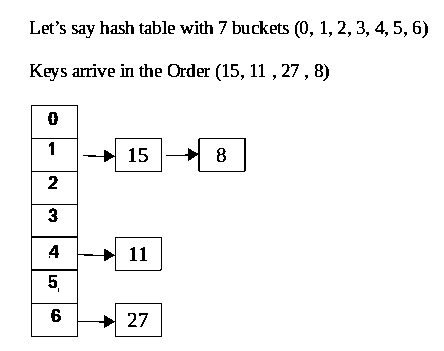
**Part A :** **Hashing With Seperate Chaining**

To write a C++ program to implement the concept of hashing using separate chaining.

**Description**

In [hashing](http://www.geeksforgeeks.org/hashing-data-structure/) there is a hash function that maps keys to some values. But these hashing function may lead to collision that is two or more keys are mapped to same value. Chain hashing resolves collisions. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let’s create a hash function, such that our hash table has ‘N’ number of buckets. To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.



Example: hashIndex = key % noOfBuckets

Insert: Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.

**ALGORITHM**

Step 1:Start

Step 2: Create Table size Step 3: Create hash function

Step 4: To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

Step 5: Display hash entry.   
Step 6: Stop

**Code:**

#include<iostream>

using namespace std;

#define sizeoflist 7

class Hash;

class ListNode;

class LinkedList;

class Hash

{

public:

int data;

ListNode\* next;

};

Hash Htable[sizeoflist];

int HashAlgo(int number)

{

return number % sizeoflist;

}

class ListNode

{

public:

int data;

ListNode\* next;

};

ListNode\* n = new ListNode;

void HashInitialize()

{

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

{

Htable[i].data = -1;

Htable[i].next = n;

}

}

class LinkedList

{

private:

ListNode\* start; // special variable which stores address of the head node.

ListNode\* last; // special variable which stores address of the last node.

ListNode\* PredLoc\_; //to be used by Search(value) method to store address of logical predecessor of value in a list.

ListNode\* Loc\_; //to be used by Search(value) method to store address of current pointer of value in a list.

ListNode\* t = new ListNode;

public:

LinkedList()

{

start = NULL;

last = NULL;

Loc\_ = NULL;

PredLoc\_ = NULL;

}

int length = 0;

void refresh()

{

Loc\_ = NULL;

PredLoc\_ = NULL;

}

bool isEmpty()

{

return start == NULL;

}

void insertLast(int value)

{

ListNode\* newnode = new ListNode;

newnode->data = value;

if (!isEmpty())

{

last->next = newnode; // prevoius last node point now to new node added

last = newnode; // newnode becomes last

last->next = t;// used in print operation

}

else

{

start = newnode; //first value so last and first are equal to new node

last = newnode;//first value so last and first are equal to new node

last->next = t;

}

length++;

}

void HashInsert(int value, int index)

{

if (Htable[index].data == -1)

{

Htable[index].data = value;

}

else

{

insertLast(value);

if (Htable[index].next == n)

Htable[index].next = start;

}

}

void PrintHash(int index)

{

ListNode\* temp = new ListNode;

temp = start;

if (Htable[index].data != -1)

{

cout << index << ". " << Htable[index].data; if (Htable[index].next != n)cout << " -> ";

}

else

{

cout << index << ". " << "Empty";

}

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

{

cout << temp->data;

if (temp->next != t)cout << " -> ";

temp = temp->next;

}

}

};

int main()

{

LinkedList\* obj = new LinkedList[sizeoflist];

HashInitialize();

int arry[sizeoflist] = { 15,10,27,12,3,24,41 };

cout <<" Size In Bytes: " << sizeof(arry) << endl;

cout << " Inserting Elements... " << endl << endl;

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

{

int index = HashAlgo(arry[i]);

obj[index].HashInsert(arry[i], index);

}

cout << " Printing Elements... " << endl;

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

{

obj[i].PrintHash(i);

cout << endl;

}

}

**Screenshot:**

Text

Description automatically generated

**VIVA (PRE & POST LAB) QUESTIONS**

* 1. If several elements are competing for the same bucket in the hash table, what is it called?
     + Ans: Standard Chaining.
  2. How to insert a node in hash table?
     + Ans: use the value and hash function to calculate index ,and then place the value in the table using index calculated.
  3. What is sizeof()function?
     + Ans: is returns, the size (in bytes) of the data container.
     + Example:
     + int arry[sizeoflist] = { 15,10,27,12,3,24,41 };
     + cout <<" Size In Bytes: " << sizeof(arry) << endl;
     + Text

       Description automatically generated
  4. How to delete a node from hash table.
     + First, we will search the value in the relevant chain after calculation the index of the table.
     + Second, we will find the node in the list where the node value is equal to search value. Delete the node by updating the pointer of the previous node. Previous node will point to the next node of the current/deleted node.
     + Third, once pointer of the previous and successor node is updated. Delete the searched/current node.

# Part B: Hashing With Open Addressing Using Linear Probing, Quadratic Probing And Double Hashing

To write a C++ program to implement the concept of hashing using the following open addressing based collision resolution techniques: a) linear probing, b) quadratic hashing, and c) double hashing.

**Description**

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored within the hash table. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data, if needed).

**Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k. **Search(k):** Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

**Delete(k):** Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.

Insert can insert an item in a deleted slot, but the search doesn’t stop at a deleted slot.

Open Addressing is done following ways:

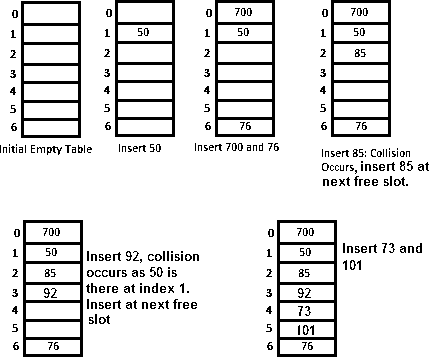
***a)* Linear Probing:** In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



**Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Code:

#include<iostream>

#define SIZE 20

using namespace std;

class HashTable

{

int hashtable[SIZE];

int Create\_Hash\_Table(int value)

{

return value % SIZE;

}

public:

HashTable()

{

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

hashtable[i] = INT\_MIN;

}

void insert(int value)

{

int key = Create\_Hash\_Table(value);

if (hashtable[key] == INT\_MIN)

hashtable[key] = value;

else

{

int temp = key;

key++;

while (hashtable[key] != INT\_MIN && key != temp)

key = Create\_Hash\_Table(key + 1);

if (key == temp)

{

cout << "\nArray is full" << endl;

return;

}

hashtable[key] = value;

}

}

void delete\_data(int value)

{

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

{

if (hashtable[i] == value)

hashtable[i] = INT\_MIN;

}

}

bool Search(int value)

{

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

{

if (hashtable[i] == value)

return true;

}

return false;

}

void Display()

{

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

cout << "Key: " << i << "\tValue: " << hashtable[i] << endl;;

}

};

void main()

{

int option = 0;

HashTable table;

while (true) {

cout << "1.Insert value\n";

cout << "2.Search value\n";

cout << "3.Delete value\n";

cout << "4.Display value\n";

cout << "5.Exit\n";

cout << "Option: ";

cin >> option;

if (option == 1) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.insert(value);

}

else if (option == 2) {

int value;

cout << "\nEnter a value: ";

cin >> value;

if (table.Search(value)) {

cout << "Value exist\n";

}

else {

cout << "Value does not exist\n";

}

}

else if (option == 3) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.delete\_data(value);

cout << "\nValue is deleted\n";

}

else if (option == 4) {

cout << "Hash Table is as follows\n";

table.Display();

}

else if (option == 5) {

break;

}

else {

cout << "\nInvalid! please enter valid command\n";

}

}

}

Screenshot:

A screenshot of a computer

Description automatically generated with medium confidence

***b) Quadratic Probing :*** We look for i2‘th slot in i’th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*1) %S

If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S

If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S

Code:

#include<iostream>

#define Table\_Size 10

using namespace std;

class HashTable

{

int hashtable[Table\_Size];

int Create\_table(int Value)

{

return Value % Table\_Size;

}

public:

HashTable()

{

for (int i = 0; i < Table\_Size; i++)

hashtable[i] = INT\_MIN;

}

void insert(int value)

{

int hash = Create\_table(value);

if (hashtable[hash] == INT\_MIN)

hashtable[hash] = value;

else

{

int temp = hash;

hash++;

while (hashtable[hash] != INT\_MIN && hash != temp)

hash = Create\_table(pow(Create\_table(hash + 1), 2));

if (hash == temp)

{

cout << "\nERROR: Array is full" << endl;

return;

}

hashtable[hash] = value;

}

}

void delete\_data(int value)

{

for (int i = 0; i < Table\_Size; i++)

{

if (hashtable[i] == value)

hashtable[i] = INT\_MIN;

}

}

bool Search(int value)

{

for (int i = 0; i < Table\_Size; i++)

{

if (hashtable[i] == value)

return true;

}

return false;

}

void Display()

{

for (int i = 0; i < Table\_Size; i++)

cout << "Index Number=" << i << "\tValue=" << hashtable[i] << endl;;

}

};

void main()

{

int option = 0;

HashTable table;

while (true) {

cout << "1.Insert value\n";

cout << "2.Search value\n";

cout << "3.Delete value\n";

cout << "4.Display value\n";

cout << "5.Exit\n";

cout << "Option: ";

cin >> option;

if (option == 1) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.insert(value);

}

else if (option == 2) {

int value;

cout << "\nEnter a value: ";

cin >> value;

if (table.Search(value)) {

cout << "Value exist\n";

}

else {

cout << "Value does not exist\n";

}

}

else if (option == 3) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.delete\_data(value);

cout << "\nSuccessfully deleted\n";

}

else if (option == 4) {

cout << "Whole table\n";

table.Display();

}

else if (option == 5) {

break;

}

else {

cout << "\nEnter a valid command\n";

}

}

}

Screenshot:

Graphical user interface, text

Description automatically generated

***c)*** [***Double Hashing***](https://www.geeksforgeeks.org/double-hashing/)***:*** we use another hash function hash2(x) and look for i\*hash2(x) slot in i’th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S

If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S   
 If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S

**ALGORITHM**

Step 1: Start

Step 2: Create hash table and hash function

Step 3: Assigning INT\_MIN indicates that cell is empty

Step4: INT\_MIN and INT\_MAX indicates that cell is empty. So if cell is empty loop will break and goto bottom of the loop to insert element. If table is full we should break, if not check this, loop will go to infinite loop.

Step 5: If the hash element is empty, the deletion has immediately failed.

Otherwise, check for a match between the target and data key

* If there is a match, delete the hash element (and set the flag)
* If there is no match, probe the table until either:
  + An match is found between the target key and the data's key; the data can be deleted, and the deleted flag set.
  + A completely empty hash element is found

Step 6: If the hash element is empty, the search has immediately failed.

Otherwise, check for a match between the search and data key

* If there is a match, return the data.
* If there is no match, probe the table until either:
  + An match is found between the search and data key
  + A completely empty hash element is found.

Step 7: Stop

Code:

#include<iostream>

#define Table\_max\_Size 15

using namespace std;

class HashTable

{

int Hash\_table[Table\_max\_Size];

int Create\_table(int n)

{

return n % Table\_max\_Size;

}

public:

HashTable()

{

for (int i = 0; i < Table\_max\_Size; i++)

Hash\_table[i] = INT\_MIN;

}

void insert(int value)

{

int hash = Create\_table(value);

if (Hash\_table[hash] == INT\_MIN)

Hash\_table[hash] = value;

else

{

hash;

for (size\_t i = 0; i < 300; i++)

{

if (Hash\_table[hash] != INT\_MIN)

{

hash = Create\_table(2 \* Create\_table(hash + 1));

break;

}

if (i == 299) {

cout << "\nERROR: Array is full" << endl;

return;

}

}

if (Hash\_table[hash] == INT\_MIN)

Hash\_table[hash] = value;

else

cout << "\nERROR: Array is full" << endl;

}

}

void delete\_data(int value)

{

for (int i = 0; i < Table\_max\_Size; i++)

{

if (Hash\_table[i] == value)

Hash\_table[i] = INT\_MIN;

}

}

bool Search(int value)

{

for (int i = 0; i < Table\_max\_Size; i++)

{

if (Hash\_table[i] == value)

return true;

}

return false;

}

void Display()

{

for (int i = 0; i < Table\_max\_Size; i++)

cout << "Key: " << i << "\tValue: " << Hash\_table[i] << endl;;

}

};

void main()

{

int option = 0;

HashTable table;

while (true) {

cout << "1.Insert value\n";

cout << "2.Search value\n";

cout << "3.Delete value\n";

cout << "4.Display value\n";

cout << "5.Exit\n";

cout << "Option: ";

cin >> option;

if (option == 1) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.insert(value);

}

else if (option == 2) {

int value;

cout << "\nEnter a value: ";

cin >> value;

if (table.Search(value)) {

cout << "Value exist\n";

}

else {

cout << "Value does not exist\n";

}

}

else if (option == 3) {

int value;

cout << "\nEnter a value: ";

cin >> value;

table.delete\_data(value);

cout << "\nValue is deleted\n";

system("pause");

}

else if (option == 4) {

cout << "Whole Table data\n";

table.Display();

}

else if (option == 5) {

break;

}

else {

cout << "\nEnter a valid command\n";

}

}

}

Screenshot:

Graphical user interface, text

Description automatically generated