Data Structures Lab 11

Course: Data Structures (CL2001) **Semester:** Spring 2024
Instructor: Bushra Sattar **SLA:** M Anus

Note:

- Maintain discipline during the lab.
- Topics: **{Hash Table, Insertion, Searching, Deletion, Collision, Linear Probing, Double Hashing and Changing }**
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Hash Table

```
class HashTableEntry {
        public:
          int k;
          int v;
          HashTableEntry(int k, int v) {
            this->k= k;
            this->v = v;
          }
};
class HashMapTable {
        private:
          HashTableEntry **t;
        public:
          HashMapTable() {}
          int HashFunc(int k) {}
          void Insert(int k, int v) {}
          int SearchKey(int k) {}
          void Remove(int k) {}
          ~HashMapTable() {}

};
```

Hash Functions

```
int K_Mod_N(int key,int N)
{
 return K % N;
}


int mid_square_hash(int key)
{
int value = key*key;
int middle_value= middle_value(value);
Return middle_value;
}

int  Folding_hash(int  value1, int  value2, int  value3)
{
   return value1 + value2 + value3;
}
```

Linear Probing

```
void Table::insert( const RecordType& entry )
{
  bool alreadyThere;
  int index;

  assert( entry.key >= 0 );

  findIndex( entry.key, alreadyThere, index );
  if( !alreadyThere )
  {
    assert( size( ) < CAPACITY );
    used++;
  }
  data[index] = entry;
}
```

Formulla:
R(K,I) = (H(K)+I)) MOD 10

Chaining with Singly Linked Lists

```
class HashNode
{
   public:
  .    int key;
          int value;
          HashNode* next;
       HashNode(int key, int value)
       {
          this->key = key;
             this->value = value;
             this->next = NULL;
       }
};
 class HashMap
{
   private:
       HashNode** htable;
   public:
       HashMap()
       {
          htable = new HashNode*[TABLE_SIZE];
          for (int i = 0; i < TABLE_SIZE; i++)
             htable[i] = NULL;
       }
  }
```

Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

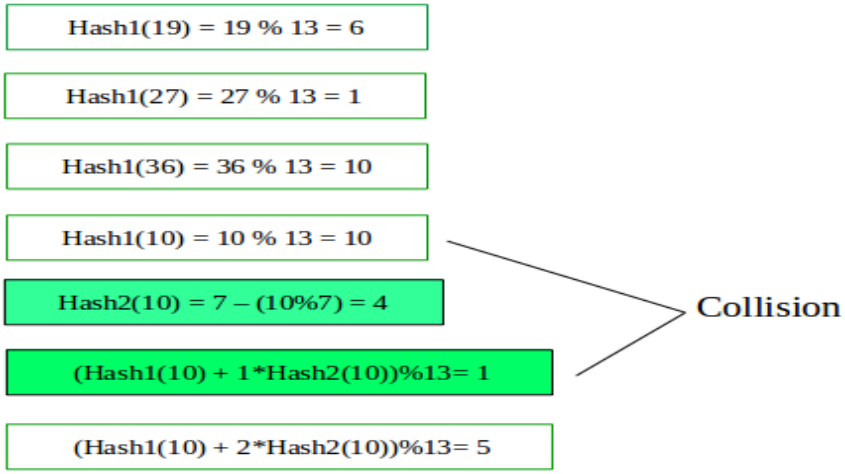
Double hashing can be done using :
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE
is size of hash table.
(We repeat by increasing i when collision occurs)

First hash function is typically hash1(key) = key % TABLE_SIZE
A popular second hash function is : hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.
A good second Hash function is:

Formulla:
H1(K)=KMOD11 //here 11 can be n
H2(K)=7-(KMOD7) //Can be any value but the most common taken is 7
(H1(k)+H2(k)) mod 11 //again here 11 can be n

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



Lab Tasks

Task-1:
Write a Program to implement Hash table and implement the following task. Via linked lists and any hash calculation method.
Keys= (20,34,45,70,56)
a. Insert element into the table
b. Search element from the key
c. Delete element at a key

Task-2:
Given an array of N integers, and an integer K, find the number of pairs of elements in the array whose sum is equal to K. Use Hashing (time complexity should not be more than N worst case)
Input:
N = 4, K = 6
arr[] = {1, 5, 7, 1}
Output: 2
Explanation:
arr[0] + arr[1] = 1 + 5 = 6
and arr[1] + arr[3] = 5 + 1 = 6.

Task-3:
Given an array arr[] of n integers. Check whether it contains a triplet that sums up to zero and time complexity should not exceed(n^2). Use hashing with any method
Note: Return 1, if there is at least one triplet following the condition else return 0.
Input: n = 5, arr[] = {0, -1, 2, -3, 1}
Output: 1
Explanation: 0, -1 and 1 forms a triplet with sum equal to 0

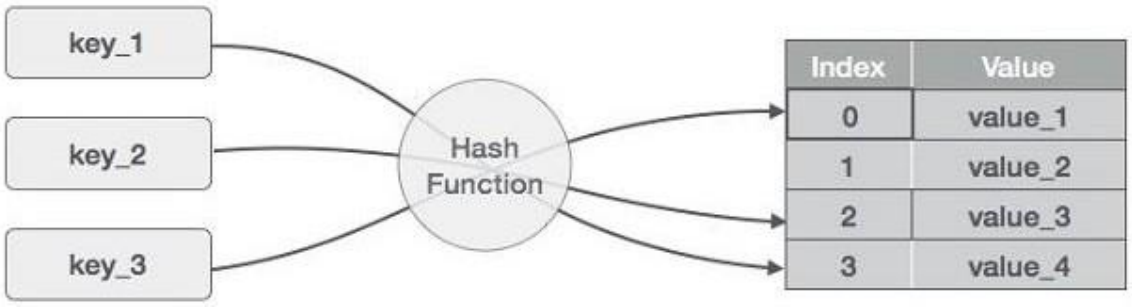
Task-4:
Given a set of N nuts of different sizes and N bolts of different sizes. There is a one-one mappingbetween nuts and bolts. Match nuts and bolts efficiently.
Comparison of a nut to another nut or a bolt to another bolt is not allowed. It means nut can only becompared with bolt and bolt can only be compared with nut to see which one is bigger/smaller.
The elements should follow the following order! # $ % & * @ ^ ~ .
Example 1:
Input:
N = 5
nuts[] = {@, %, $, #, ^} //You can use any symbols they are not exclusive to thisbolts[] = {%, @, #, $ ^}
Output:
$ % @ ^# $ % @ ^

Note: Might sound easy but the max allowed time complexity is N*log(N) with hashes.

Task-5:
Consider a hash table that uses the linear probing technique with the following hash function f(x) = (5x+4)%11. (The hash table is of size 11.) If we insert the values 3, 9, 2, 1, 14, 6 and 25 into the table, in that order, display where these values would end up in the table?

1. Hash Table

A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. It is an array into which data is inserted using a hash function.



2. Hash Functions

i. Division Method

This is the easiest method to create a hash function. The hash function can be described as h(k) = k mod n. Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder. It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

ii. Multiplication Method

The hash function used for the multiplication method is h(k) = floor(n(kA mod 1)) Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value.

iii. Mid Square Value Method

In Mid square, the key is squared and the address is selected from the middle of the result.

iv. Folding Method

Divide the key into several parts with same length (except the last part) • Then sum up these parts (drop the carries) to get the hash address Two method of folding:

Shift folding — add up the last digit of all the parts with alignment

Boundary folding — each part doesn't break off, fold to and fro along the boundary of parts, then add up these with alignment, the result is a hash address

v. Radix Method

Regard keys as numbers using another radix then convert it to the number using the original radix. Pick some digits of it as a hash address, usually choose a bigger radix as converted radix, and ensure that they are inter-prime

3. Collisions

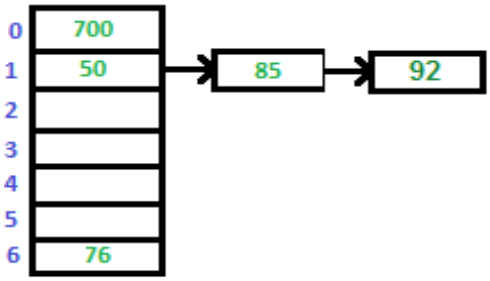
The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value.

4. Linear Probing

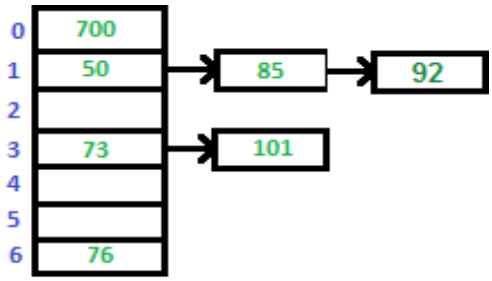
A hash table in which a collision is resolved by putting the item in the next empty place in the array following the occupied place. The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location. Here the idea is to place a value in the next available position. Because in this approach searches are performed sequentially so it's known as linear probing. Here array or hash table is considered circular becausewhen the last slot reached an empty location not found then the search proceeds to the first locationof the array.

5. Chaining

A chained hash table fundamentally consists of an array of linked lists. Each list forms a bucket in which we place all elements hashing to a specific position in the array. To insert an element, we first pass its key to a hash function in a process called hashing the key. This tells us in which bucketthe element belongs. We then insert the element at the head of the appropriate list. To look up or remove an element, we hash its key again to find its bucket, then traverse the appropriate list untilwe find the element we are looking for. Because each bucket is a linked list, a chained hash tableis not limited to a fixed number of elements. However, performance degrades if the table becomestoo full.



Inser 92 Collision Occurs, add to chain



Insert 73 and 101

Linear Probing

In linear probing, we linearly probe for next slot. It is the simplest way to avoid collisions. But the main problem withthis technique is clustering. There is the possibility that many consecutive elements form groups and it starts takingtime to find a free slot or to search an element.

Let hash(x) be the slot index computed using a hash function and S be the table

sizeIf 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
……..
……..

Algorithm

Insertion operation in Linear Probing

- Calculate a hash code from the key
- Access that hash element
    - If the hash element is empty, add straight away
    - If not, probe through subsequent elements, trying to find a free place
        - If a free place is found, add the data at that position
        - If no free place is found, the add will fail

Search operation in linear probing

- Calculate the hash code for the given search key
- Access the hash element
- 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:
        - A match is found between the search and data key
        - A completely empty hash element is found

Deletion operation in linear probing

- Calculate the hash code for the given target key
- Access the hash element
- 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:

- A 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.

Example Program

implementation of hashing in using open addressing (linear probing) for collision resolution

```
#include<iostream>
using namespace std;
const int SIZE = 10;

class HashTable {
private:
   int arr[SIZE];
public:
HashTable() {
      for (int i = 0; i < SIZE; i++) {
         arr[i] = -1;
      }
   }
   int hashFunction(int key) {
      return key % SIZE;
   }

   void insert(int key) {
      int index = hashFunction(key);
      int i = 0;

      while (arr[(index + i) % SIZE] != -1 && arr[(index + i) % SIZE] != -2 && arr[(index + i) %
SIZE] != key) {
         i++;
      }

      if (arr[(index + i) % SIZE] == key) {
         cout << "Element already exists in the hash table" << endl;
      }
      else {
arr[(index + i) % SIZE] = key;
      }
   }
   void remove(int key) {
      int index = hashFunction(key);
      int i = 0;
```

```
        while (arr[(index + i)
        % SIZE] != -1) {
if (arr[(index + i) % SIZE] == key) {
arr[(index + i) % SIZE] = -2;
cout << "Element deleted from the hash table" << endl;
return;
}
i++;
}

    cout << "Element not found in the hash table" << endl;
}
void display() {
    for (int i = 0; i < SIZE; i++) {
        if (arr[i] == -1 || arr[i] == -2) {
            continue;
        }
        cout << "Index " << i << ": " << arr[i] << endl;
    }
}
};

int main() {
HashTable ht;
ht.insert(5);
ht.insert(15);
ht.insert(25);
ht.insert(35);
ht.insert(45);
ht.display();
ht.remove(15);
ht.display();
ht.remove(10);
ht.display();
ht.insert(55);
ht.display();

return 0;
}
```