Data Structures Lab 11

Course: Data Structures (CL2001) **Semester:** Fall 2023
Instructor: Shafique Rehman **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Topics: {Hash Table, Insertion, Searching, Deletion, Collision, Linear Probing, Double Hashing, Chaining, Simple graphs and traversals}
- 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.

Hast Table

```
class HashTableEntry {
   public int k;
   public int v;

 public HashTableEntry(int k, int v) {
       this.k = k;
       this.v = v;
           }
}

class HashMapTable {
   HashTableEntry[] t;

public HashMapTable(int size) {}

private int HashFunc(int k) {
       // Your hash function logic here}

public void Insert(int k, int v) { }

public int SearchKey(int k) {
        // Return -1 if key not found (you can change this logic as needed)}


public void Remove(int k) {// after removing element assign null to that index to completly remove the
elements from memory}
```

Hash Functions

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

    public static int mid_square_hash(int key) {
        int value = key * key;
        int middleValue = middleValue(value);
        return middleValue;
    }

    public static int middleValue(int value) {
        // Extract middle digits; logic might differ depending on desired approach
        // This is a simple example considering a 4-digit number
        String valStr = String.valueOf(value);
        int midIndex = valStr.length() / 2;
        String middleDigits = valStr.substring(midIndex - 1, midIndex + 1);
        return Integer.parseInt(middleDigits);
    }

    public static int foldingHash(int value1, int value2, int value3) {
        return value1 + value2 + value3;
    }
```

Linear Probing

```
public class Table {
    private static final int CAPACITY = 100; // Adjust capacity as needed
    private RecordType[] data;
    private int used;

    public Table() {
        data = new RecordType[CAPACITY];
        used = 0;
    }

    public void insert(RecordType entry) {
        boolean alreadyThere = false;
        int index = 0;

        assert entry.key >= 0;

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

    // Dummy implementation for findIndex and size methods
    // Replace these with actual logic for your question
    private void findIndex(int key, boolean alreadyThere, int index) {
        // Implement your findIndex logic here
        // Set alreadyThere and index values accordingly
    }

    private int size() {
        // Implement your size calculation logic here
        return used;
    }

    // Define RecordType class or structure as needed
    // Example:
    private static class RecordType {
        int key;
        // Other fields relevant to your question/requirements
    }
}
```
Formulla: R(K,I) = (H(K)+I)) MOD 10

Chaining with Singly Linked Lists

```
class HashNode {
    public int key;
    public int value;
    public HashNode next;

    public HashNode(int key, int value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }
}

class HashMap {
    private HashNode[] htable;
    private static final int TABLE_SIZE = 100; // Adjust table size as needed

    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.

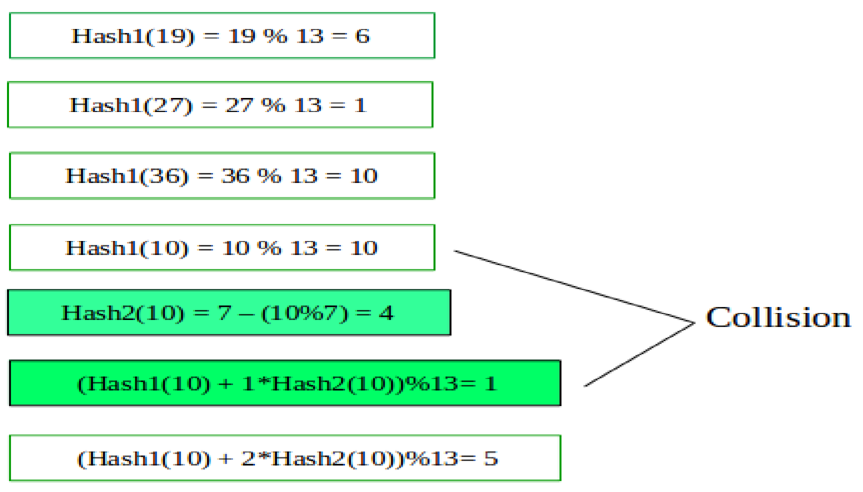
Double hashing can be done using :
(hash1(key) + i * hash2(key)) % TABLE_SIZE
Here hash1() and hash2() are hash functions and TABLE_SIZE
is size of hash table.
(We repeat by increasing i when collision occurs)

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

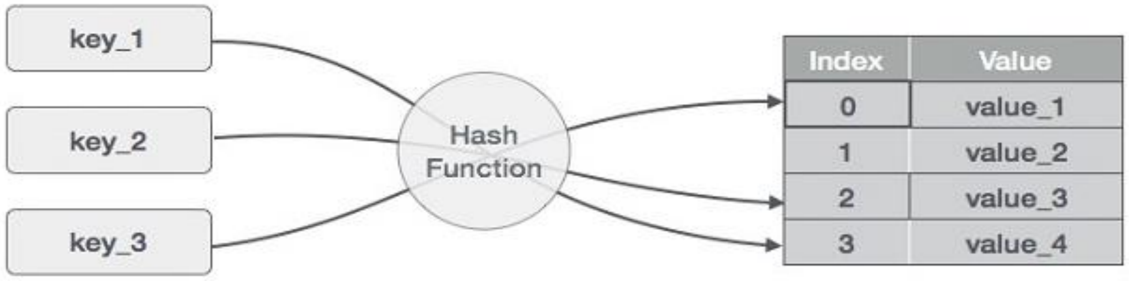
Hash2 (key) = 7 – (key % 7)



References:

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

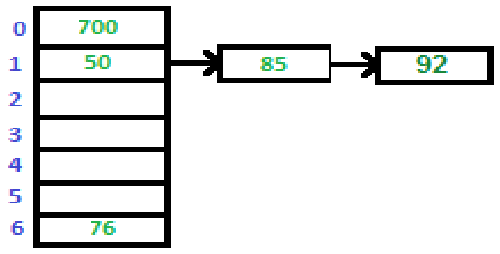
The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result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

4. Linear Probing

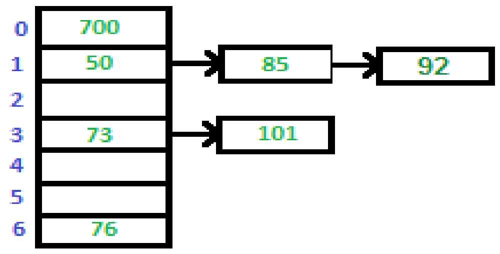
A hash table in which a collision is resolved by putting the item in the next empty place in the array following the occupied place. The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred 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 because when the last slot reached an empty location not found then the search proceeds to the first location of 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 bucket the 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 until we find the element we are looking for. Because each bucket is a linked list, a chained hash table is not limited to a fixed number of elements. However, performance degrades if the table becomes too full.



Inser 92 Collision Occurs, add to chain



Insert 73 and 101