Lesson 31-32

Objectives

- Hashing
    - Hashing
    - Find the Hash Function
    - Characteristics of good hashing function
    - Collision
    - Solutions for Handling Collision
    - Implementing Hashing

Hashing:

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Hashing is a technique which uses less key comparisons and searches the element in **O(n)** time in the worst case and in an average case it will be done in **O(1)** time. This method generally used the hash functions to map the keys into a table, which is called a **hash table**.

1) Hash table

Hash table is a type of data structure which is used for storing and accessing data very quickly. Insertion of data in a table is based on a key value. Hence every entry in the hash table is defined with some key. By using this key data can be searched in the hash table by few key comparisons and then searching time is dependent upon the size of the hash table.

2) Hash function

Hash function is a function which is applied on a key by which it produces an integer, which can be used as an address of hash table. Hence one can use the same hash function for

accessing the data from the hash table. In this the integer returned by the hash function is called hash key.

Types of Finding Hash Function

There are various types of hash function which are used to place the data in a hash table.

1. Division method

In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.

Then:

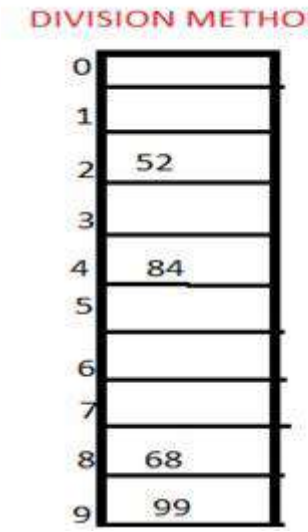
h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10



2. Mid square method

In this method firstly key is squared and then mid part of the result is taken as the index.

For example: consider that if we want to place a record of 3101 and the size of table is 1000. So 3101*3101=9616201 i.e. **h (3101) = 162 (middle 3 digit)**

3. Digit folding method

In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key. For example: consider a record of 12465512 then it will be divided into parts i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.

H(key)=124+655+12

=791

Characteristics of good hashing function

1. The hash function should generate different hash values for the similar string.
2. The hash function is easy to understand and simple to compute.
3. The hash function should produce the keys which will get distributed, uniformly over an array.
4. A number of collisions should be less while placing the data in the hash table.
5. The hash function is a perfect hash function when it uses all the input data.

Collision

It is a situation in which the hash function returns the same hash key for more than one record, it is called as collision. Sometimes when we are going to resolve the collision it may lead to a overflow condition and this overflow and collision condition makes the poor hash function.

Solutions For Handling Collisions:

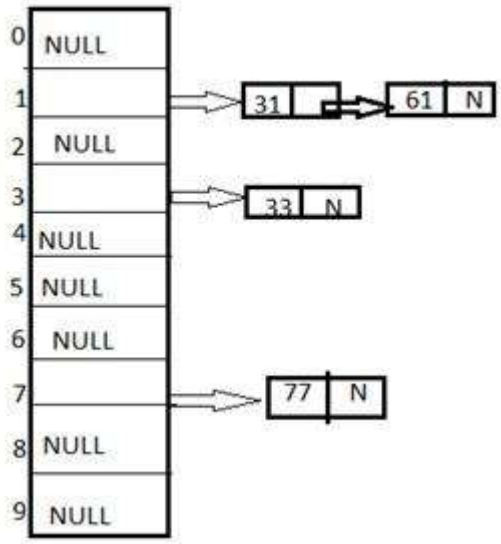
If there is a problem of collision occurs then it can be handled by apply some technique. These techniques are called as collision resolution techniques. There are generally four techniques which are described below.

1) Chaining

It is a method in which additional field with data i.e. chain is introduced. A chain is maintained at the home bucket. In this when a collision occurs then a linked list is maintained for colliding data.

Example:

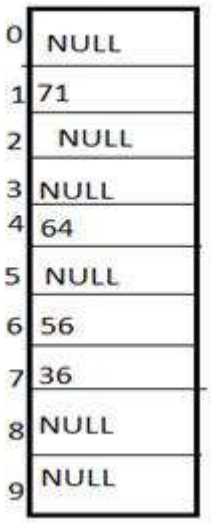
Let us consider a hash table of size 10 and we apply a hash function of H(key)=key % size of table. Let us take the keys to be inserted are 31,33,77,61. In the below diagram we can see at same bucket 1 there are two records which are maintained by linked list or we can say by chaining method.



2) Linear probing

It is very easy and simple method to resolve or to handle the collision. In this collision can be solved by placing the second record linearly down, whenever the empty place is found. In this method there is a problem of clustering which means at some place block of a data is formed in a hash table.

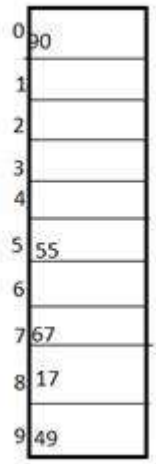
Example: Let us consider a hash table of size 10 and hash function is defined as H(key)=key % table size. Consider that following keys are to be inserted that are 56,64,36,71.



In this diagram we can see that 56 and 36 need to be placed at same bucket but by linear probing technique the records linearly placed downward if place is empty i.e. it can be seen 36 is placed at index 7.

3) Quadratic probing

This is a method in which solving of clustering problem is done. In this method the hash function is defined by the H(key)=(H(key)+x*x)%table size. Let us consider we have to insert following elements that are:-67, 90,55,17,49.



In this we can see if we insert 67, 90, and 55 it can be inserted easily but at case of 17 hash function is used in such a manner that :-(17+0*0)%10=17 (when x=0 it provide the index value 7 only) by making the increment in value of x. let x =1 so (17+1*1)%10=8.in this case bucket 8 is empty hence we will place 17 at index 8.

4) Double hashing

It is a technique in which two hash function are used when there is an occurrence of collision. In this method 1 hash function is simple as same as division method. But for the second hash function there are two important rules which are

1. It must never evaluate to zero.
2. Must sure about the buckets, that they are probed.

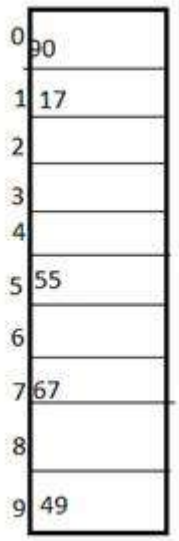
The hash functions for this technique are:

H1(key)=key % table size

H2(key)=P-(key mod P)

Where, **p** is a prime number which should be taken smaller than the size of a hash table.

Example: Let us consider we have to insert 67, 90,55,17,49.



In this we can see 67, 90 and 55 can be inserted in a hash table by using first hash function but in case of 17 again the bucket is full and in this case we have to use the second hash function which is H2(key)=P-(key mode P) here p is a prime number which should be taken smaller than the hash table so value of p will be the 7.

i.e. H2(17)=7-(17%7)=7-3=4 that means we have to take 4 jumps for placing the 17. Therefore 17 will be placed at index 1.

The hash.java Program

```
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
{                          // (could have more data)
private int iData;              // data item (key)
//--------------------------------------------------------------public
DataItem(int ii)          // constructor
{ iData = ii; }
//--------------------------------------------------------------
public int getKey()
{ return iData; }
//--------------------------------------------------------------}
 // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
{
private DataItem[] hashArray;    // array holds hash table

private int arraySize;

private DataItem nonItem;       // for deleted items
// -------------------------------------------------------------
public HashTable(int size)       // constructor
{
```

```
arraySize = size;
hashArray = new DataItem[arraySize];
nonItem = new DataItem(-1);   // deleted item key is -1
}
// -------------------------------------------------------------public
void displayTable()
{
System.out.print("Table: ");
for(int j=0; j<arraySize; j++)
{
if(hashArray[j] != null)
System.out.print(hashArray[j].getKey() + " ");
else
System.out.print("** ");
}
System.out.println("");
}
// -------------------------------------------------------------public
int hashFunc(int key)
{
return key % arraySize;       // hash function
}
// -------------------------------------------------------------public
void insert(DataItem item) // insert a DataItem
//
(assumes table not full)
{
int key = item.getKey();      // extract key
int hashVal = hashFunc(key);  // hash the key
// until empty cell or -1,
while(hashArray[hashVal] != null &&
hashArray[hashVal].getKey() != -1)
{
++hashVal;                // go to next cell
hashVal %= arraySize;     // wraparound if necessary
}
hashArray[hashVal] = item;    // insert item

}  // end insert()

// -------------------------------------------------------------public
DataItem delete(int key)  // delete a DataItem
{
int hashVal = hashFunc(key);  // hash the key
while(hashArray[hashVal] != null)  // until empty cell,
{                             // found the key?
if(hashArray[hashVal].getKey() == key)
{
DataItem temp = hashArray[hashVal]; // save item
hashArray[hashVal] = nonItem;       // delete item
return temp;                        // return item
```

```
}
++hashVal;                // go to next cell
hashVal %= arraySize;     // wraparound if necessary
}
return null;              // can't find item
}  // end delete()
// ---------------------------------------------------------public
DataItem find(int key)    // find item with key
{
int hashVal = hashFunc(key);  // hash the key
while(hashArray[hashVal] != null)  // until empty cell,
{                         // found the key?
if(hashArray[hashVal].getKey() == key)
return hashArray[hashVal];   // yes, return item
++hashVal;                // go to next cell
hashVal %= arraySize;     // wraparound if necessary
}
return null;              // can't find item
}
// ---------------------------------------------------------}
 // end class HashTable
////////////////////////////////////////////////////////////////
class HashTableApp
{
public static void main(String[] args) throws IOException
{
DataItem aDataItem;

int aKey, size, n, keysPerCell;


// get sizes
System.out.print("Enter size of hash table: ");
size = getInt();
System.out.print("Enter initial number of items: ");
n = getInt();
keysPerCell = 10;
// make table
HashTable theHashTable = new HashTable(size);
for(int j=0; j<n; j++)       // insert data
{
aKey = (int)(java.lang.Math.random() *
keysPerCell * size);
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem);
}
while(true)               // interact with user
{
System.out.print("Enter first letter of ");
System.out.print("show, insert, delete, or find: ");
char choice = getChar();
switch(choice)
```

```
{
case ‘s’:
theHashTable.displayTable();
break;
case ‘i’:
System.out.print(“Enter key value to insert: “);
aKey = getInt();
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem);
break;
case ‘d’:
System.out.print(“Enter key value to delete: “);
aKey = getInt();
theHashTable.delete(aKey);
break;
case ‘f’:
System.out.print(“Enter key value to find: “);

aKey = getInt();

aDataItem = theHashTable.find(aKey);
if(aDataItem != null)
{
System.out.println(“Found “ + aKey);
}
else
System.out.println(“Could not find “ + aKey);
break;
default:
System.out.print(“Invalid entry¥n”);
}  // end switch
}  // end while
}  // end main()
//--------------------------------------------------------------public
static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//--------------------------------------------------------------public
static char getChar() throws IOException
{
String s = getString();
return s.charAt(0);
}
//--------------------------------------------------------------public
static int getInt() throws IOException
{
String s = getString();
```

```
return Integer.parseInt(s);
}
```