Chapter 7- Hashing

7.1 - Introduction to Hashing

7.2 - Examples of Hashing Functions

7.3 –Collision Resolution

7.4 – Perfect hash function

7.1 Hashing

Searching for a key in an array takes O(logn) time using Binary Search if the array is sorted.

Searching in a BST takes O(h) time, and O(h) is O(logn) if the tree is balanced.

Now we want to get more ambitious, and be able to search in O(1) time!

Given a key we want to be able to locate the key in O(1) time!

How should we store the key so that this can be done?

7.1 Hash Tables

We will store the keys in a table called hash-table indexed by the key.

The idea is to compute a function called "hash function" which computes the index in the table, where the key is to be stored.

For example if the hash function *h*, is the number of letters in the key,

h("Larry") = 5, so Larry's record is stored at location 5.

h("Tom") = 3, so Tom's record is stored at location 3.

If we wanted to locate a record, we simply find the hash function of the name, and go to that index.

What is the problem with this technique?

7.1 Collision

There is an issue of *collision* since *h*("Tom") = *h*("Joe"). How we store Tom's record and Joe's record in the same location?

A hash function is called a *perfect hash function* if it maps different key's to different numbers.

Are there perfect hash functions?

Given less than perfect hash functions, how do we resolve collision?

Can we still search in O(1) time?

7.2 Division Hashing

One of the properties of hash function is to make sure that the index it creates is a valid index in the table.

Generally the table size *tsize* is chosen to be a prime number, and then the hash function *h*(*key*) = *key MOD size*

The key is assumed to be a number, but this can always be arranged for instance you can take a key that is a string and create a number out of it using the number of characters in the string, or using the number of vowels or some other count in the string.

Does each key generate a unique number. NO. There is possibility of collision.

But this is a commonly used hash function.

7.2 Folding

Another commonly used hash function is called folding. Basically you take the key and split it in parts and fold it and compute another number using folding.

For example, ss## 123-45-6789 can be folded as

123

456

789

Now, when we add we get 1368. We use 1368 MOD tsize to compute the location for 123-45-6789.

7.2 Folding

Or, the middle number could be folded again as

123

654

789

When added, we get 1566. Now we use 1566 MOD tsize to store the number 123-45-6789.

7.2 Mid Square function

In this technique, the key is squared and mid-part of the result is used as the index to store the key.

For example, to store 3121, we compute $3121^2 = 9740641$.

Assuming that the table size is 1000, we could use the mid-part, 406, as the location to store 3121.

If the key is a string, we first need to convert it to a numeric value using Unicode values of the characters in the string.

7.3 Collision Resolution

Although the techniques we have seen so far, appears to search for a key in time O(1), there is an issue with storing the keys when they have the same hash value.

Since the hash functions of two different keys can be the same, how do we store the keys that hash to the same value. Then, how do we search for a key whose hash value corresponds to more than one key.

To resolve collisions such as above, we use a technique called probing.

7.3 Linear Probing

When an element K is to be inserted into the table, first the hash function h(K) is computed on the key to find the address. If the address is occupied, then the next available address is found, among the following list:

h(K) , h(K) +p(1), h(K) + p(2) , h(K) + p(3) ….

The function p(i) is called the probing function. If the simplest probing function p(i) = i, is used, the hashing technique is called *linear probing*.

That is, in linear probing, if h(K) is occupied, we look for available address in the following order:

h(K) +1, h(K) + 2 , h(K) + 3 ….

Assuming that the keys A_i, B_i, C_i, hashes to i, let us try to use linear probing to add the following to the hash table.

7.3 Linear Probing

Arrivals of elements occurs in the following order:

	A_5, A_2, A_3,	B_5, A_9, B_2,	B_9, C_2,
1			
2	A_2	A_2	A_2
3	A_3	A_3	A_3
4		B_2	B_2
5	A_5	A_5	A_5
6		B_5	B_5
7			C_2
8			
9		A_9	A_9
10			B_9

7.3 Quadratic Probing

The problem with linear probing is that there is 'clustering'. Once there is a collision, the elements tend to cluster around the same locations. This causes table to be dense in some areas and sparse in others and this makes the search time longer.

If we use a quadratic function for p(i), there will not be as much clustering. Let us use the probe sequence as

h(K), h(K) $+1^2$, h(K) -1^2 , h(K) + 2^2, h(K) – 2^2, h(K) + 3^2, h(K) –3^2 etc. ….

Using the quadratic probing, the placement of the elements in the previous example would be as shown below.

7.3 Quadratic Probing

Arrivals of elements occurs in the following order:

	$A_5, A_2, A_3,$	$B_5, A_9, B_2,$	$B_9, C_2,$
1		B_2	B_2
2	A_2	A_2	A_2
3	A_3	A_3	A_3
4			
5	A_5	A_5	A_5
6		B_5	B_5
7			C_2
8			
9		A_9	A_9
10			B_9

7.3 Search by Probing

When searching for a key, we simply rehash the key and then search for key using the same probing sequence as given in linear (or quadratic) hashing.

If we looking for C_2 in the linear probing example above, we will first hit A_2, since there is no match, we will then look at the probe sequence and find A_3, B_2, A_5, B_5 and then find C_2.

When the key is in the table, it is found in one of the locations in the hash sequence. If the key is not in the table, we have to travel the hash sequence until make one complete cycle on the hash table.

7.3 Search Times

Here is the time complexity of search using hashing.

LF = (number of entries in the table)

(size of the table)

Type of search	Linear Probing	Quadratic probing
Successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$
Unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$

7.3 Chaining

Chaining is another technique for collision resolution.

Whenever there is a collision, as linked list is used the store other elements which hash to the same address.