Chapter 7

Hashing

7.1 Introduction

- Remember that although in most of our classroom examples we are using simple lists of integers as out data sets, in practice it is not like that. The “elements” or “items” in realistic situations are records where each record might consists of several fields; each field perhaps being of several digits or characters.

- However, one of the fields must be considered to be the way in which a record is identified (hence called the IDENTIFIER). Typically this might be an employee’s payroll number or a stock item’s product code. This identifier is also known as the record KEY.

- Let us revert to our classroom practice where the elements are simple integers representing the identifiers (rather than records with keys). Consider the following sequence of ‘keys’.

Array index	0	1	2	3	4
Key	254	378	631	794	827

- Our search techniques to date are : sequential search or binary search

- Both assumed that the elements were held in contiguous locations in an array (as above) and the binary search required them to be sorted. Search times were O(n) and $O(\log_2 n)$ respectively.

- Suppose we could generate the index of the key from the key itself!!!

- We would then have a search technique with search time O(1)!!! This is what you achieve with HASHING

7.2 Hashing

- With hashing techniques you attempt to find a HASHING FUNCTION that is able to convert the keys (usually integers or short character strings) into integers in the range 0…N-1 where N is the number of records that can fit into the amount of memory available.

- In our examples we will just consider a record that consist solely of a key that is an integer and that we want to convert the key into the index of an array. This may or may not be simple.

- The term HASH and HASHING came from the very fact that the conversion from key to index really does 'hash' the key as in many cases the index resulting from 'hashing' a key bears absolutely no resemblance to the original key at all.

- Suppose memory was limitless then we could have the simplest hash function of all index = key

- In the example above this would mean declaring an array of 1000 elements and putting the keys in those elements with the same index!

- In the example this would mean that 5 elements of the array contained data and that 995 did not! For this to be a viable proposition, memory would have to be limitless (and can be wasted). In practice you can not afford such extravagance.

- The hash function you choose depends very much on the distribution of actually key within the range of the possible keys.

A hash search is a search in which the key, through an algorithmic function, determines the location of the data. Because we are searching an array, we use a hashing algorithm to transform the key into the index that contains the data we need to locate.

7.3 Hash Function – Truncation

- With truncation part of the key is ignored and the remainder used directly.

- Suppose in a certain situation there were just 6 keys and they were:

12302	12303	12305	12307	12308	12309

- The hash function here would be

index = last digit of key or (index = key mod 12300)

- So provided that we have declared an array of 10 elements (index 0 to 9) then the hash function would be suitable and with (only?) 40% wastage. Of course the above hash function would not work if the range of keys was:

	2134	4134	5134	7134	9134
or	2560	4670	6124	8435	9200

- Why wouldn't it work???

- Here you would require a hash function that removes the last 3 digits. This is why it is necessary for you to know something about the distribution of actual keys before deciding on a hash function.

Important points to remember in hashing

When truncating back digits:

- Apply the divide (/) operator
- Divide by 1, then followed by *n* number of zeros (number of zeros equals the number of digits to **truncate at the back**).

e.g. given key 12345, if we are asked to truncated the last **3** digits at the back: **hash value = key / 1000 = 12345 / 1000 = 12**

```
//function definition – truncated last 3 digits int
hash_function(int key)
{
    int hash_value;
    hash_value = key / 1000;
    return hash_value;
}
```

When truncating front digits:

- Apply the modulus (%) operator
- Modulus by 1, then followed by *n* number of zeros (number of zeros equals number of **remainder digits**).

e.g. given key 12345, if we are asked to truncated the first 3 digits (remainder left 2 digits):

hash value = key % 100 = 12345 % 100 = 45

```
//function definition – truncated first 3 digits, remainder left 2 digits int
hash_function(int key)
{
    int hash_value;
    hash_value = key % 100;
    return hash_value;
}
```

When truncating front & back digits:

- Apply the divide (/) & modulus (%) operator

e.g. given key 12345, if we are asked to truncated the first 2 digits and last digit

Truncating last digit (1 digit at the back)

Hash value = 12345 / 10 = 1234

Truncating first 2 digits (remainder left 2 digits)

Hash value = 1234 % 100 = 34

Complete formula: Hash value = (key / 10) % 100

```
//function definition – truncated the first 2 digits and last digit int
hash_function(int key)
{
    int hash_value;
    hash_value = (key / 10) % 100;
    return hash_value;
}
```

7.4 Hash Function – Folding

- Here the key is partitioned and the parts combined in some way (maybe by adding or multiplying them) to obtain the index. Suppose we have 1000 records but 8 digit keys then perhaps.

The 8 digit key such as:	62538194	62538195
May be partitioned into:	625 381 94	625 381 95
The groups now added:	1100	1101
And the result truncated:	100	101

- Since all the information in the key is used in generating the index, folding often achieves a better spread than truncation just by itself.

7.5 Hash Function – Modular Arithmetic

- The key is converted into an integer, the integer is then divided by the size of the index range and the remainder is taken as the index position of the record.

```
index = key mod size
```

- As an example, suppose we have a 5 digit integer as a key and that there are 1000 records (and room for 1000 records)

```
The hash function would then be: index = key mod 1000
```

- If we are every lucky! Our keys might be such that there is only1 key that maps to each index. Of course we might still have the situation where two keys map to the same range (e.g. 23456, 43456) – this is called a COLLISION.

- In practice it always turns out that it is better to have an index range that is a prime number. This way you do not get so many COLLISIONS.

- In the above, it would be better to have an index range of 997 or 1009. However, collisions do and will occur.

7.6 Collision Resolution with Open Addressing

 In hashing collisions do occur. Collisions happened when two different keys result in the same index by the hashing function. Such collisions have to be resolved.

7.6.1 Linear Probing

- With liner probing, you start at the point where the collision occurred and do a sequential search through the table for an empty location (if you are setting up the table) or for the desired key – or empty location (if you are searching for a target). Because the method searches in a straight line it is called linear probing.

- Note that this method should be considered circular to enable the search to be continued at the beginning of the table when the end of the table is reached.

7.6.2 Clustering

 The problem with linear probing is that there is a tendency towards clustering (i.e. when more entries are made, there is a greater chance of a collision and strings of keys result). Thus the searching technique gets reduced almost to a sequential search!

7.6.3 Quadratic Probing

 One way of reducing clustering is to use quadratic probing. Here instead of probing at h+1, h+2, h+3 etc, you probe at h+1, h+4, h+9, h+16 etc. (always mode the hash size).

7.7 Collision Resolutions by Chaining

- It is convenient to store the hash table in contiguous storage (i.e. in an array) as this way we are able to refer quickly to random positions in the table (linked storage in not suitable for this).

- However there is no reason why the records and the collisions could not be stored in linked storage. This is referred to as collision resolution by chaining.

7.8 Example of Collision Resolution

 Consider 6 keys and a hash table of size 7. We will ensure collisions by choosing the hash function **h = key mod 7** and the 7 key as follows:

Keys	12	15	21	36	84	96
Hash index	5	1	0	1	0	5

7.8.1 Resolution with Linear Probe

Index	0	1	2	3	4	5	6
Key						12	
		15				12	
	21	15				12	
	21	15	36			12	
	21	15	36	84		12	
	21	15	36	84		12	96

7.8.2 Resolution with Chaining

Index	0	1	2	3	4	5	6
Key						12	
		15				12	
	21	15				12	
	21	15-36				12	
	21-84	15-36				12	
	21-84	15-36				12-96	