Lecture 15.1

Topics:

1. Ideal Hashing
2. Hash Functions and Techniques
3. Collisions and Overflows
4. Hashing By Division – Digression
5. Hashing with Chains (Linked Lists)

---

1. Ideal Hashing

1.1 Definition

> In hashing, there is a hash function that maps keys into positions in a table called the hash table. In ideal situation, if the element **e** has the key **k** and **f()** is the hash function, then **e** is stored in the position **f(k)** of the table.

To search for an element with key `k`, one compute `f(k)` and see whether an element exists at position `f(k)` of the table. If so, the element is found; if not, the dictionary contains no element with this key. Also, if the element was found, it may be deleted (if required) my making the position `f(k)` of the table empty. If the element was not found, it may be inserted by placing the element in position `f(k).`

There are many applications that would require some kind of searching for the data given some key information. A quickest search (with an indicated result) may need large amount of memory space to become inappropriate for use. Other search may allow trade off to get data a bit slower.

The hash table will provide the search in constant time for any given element. Depending on the design of the table, the access time may not be affected by the numbers of elements stored in a table. Note also that the operations required by any hash table only use the "equality comparison" operator.

1.2 Hash Functions and Tables

When the key range is too large to use the ideal approach of above, then one can use a hash table whose length is smaller than the key range and a hash function `f(k)` that map several different keys into the same position of the hash table.

Each position of the table is a **bucket**; `f(k)` is the **home bucket** for the element whose key is `k.` The number of buckets in a table equals the table length.

Since a hash function may map several keys into the same bucket, one may consider designing buckets that can hold more than one element. The number of elements that a bucket may hold equals the number of slots in the bucket.

There are issues that need to be considered when working with a hash table. They are collisions, overflows, and methods to avoid them. These will be presented in the next lecture(s).

Let's look at several terms involved in hashing as below.

`(k, e)` : The pair of key and element

`r`	: The range of keys
`b`	: The table size or number of buckets
`f(k)`	: The hash function
`search()`	: The method to find the element using the key
`insert()`	: The method to insert the element to some location determined by the key
`remove()`	: The method to remove the element at location determined by the key

In general, the expected occurrence of collisions and overflows is minimized when approximately the same number of keys from the key range hashes into any bucket of the table. That means we may have a distribution of keys to be relatively the same for all buckets. A uniform hash function is one of the examples.

In practice, a hash function may not result to a uniform distribution. It is apparent that some hash functions will work better than the others in providing an approximately equal distribution of keys to home buckets. These uniform hash functions that produce good distribution of keys are called **good hash functions**. Further discussion on the good hash functions can be found in many texts.

2. Hash Functions & Techniques

In a hashing structure,

(i) If the element **e** has the key **k** and **f()** is the hash function, then **e** is stored in the position **f(k)** of the table.

(ii) When the key range is too, then a hash table whose length is smaller than the key range and a hash function `f(k)` (which provides a mapping to take several different keys into the same position of the hash table) would be employed.

2.1 Tables -- Buckets, Home Buckets, Slots

Each position of the table is a **bucket**; `f(k)` is the **home bucket** for the element whose key is `k.` The number of buckets in a table equals the table length.

Since a hash function may map several keys into the same bucket, one may consider designing buckets that can hold more than one element. The number of elements that a bucket may hold equals the number of **slots** in the bucket.

In general, a hash table that is kept in main memory of a computer will have either 0 or 1 slot per bucket. Hash table that is stored on disk may have several slots per bucket.

2.2 Hash Functions/Techniques

There are two groups of mapping functions/techniques that one can use: linear and nonlinear. A linear mapping will give a one to one value (position) with one key. A nonlinear mapping may assign two or more keys to the same position.

The linear mapping is not practical and not considered. There are many different nonlinear mapping that would map a larger range of k to a smaller range in `f(k).`

3. Collision & Overflow Handling – Probing Functions

Let's consider again the case where a bucket has only one single slot and there is a collision at **$f(k_h)$.** The general formulation for finding the alternative cell **`a(k)`** will be as follows

$$a(i) = (f(k_h) + p(i)) \% \text{TABLESIZE} \quad \text{with } p(0) = 0 \qquad \text{(Eqn. 3)}$$

where `p(i)` is the collision resolution strategy (or, probing function).

- A linear probing function `p(i)`, which is a linear function (a function with order 1 for the variable **`i`**), of `i`.
- A quadratic probing function p(i) will have second order for i.

4. Hashing By Division –Digression

One would like to choose **`D`** so that the hash function **`f(k)`** is a good hash function, i.e., a function that would produce a good distribution of keys (an approximately equal distribution of keys to home buckets).

Some choices of **`D`** result in good hash function and others would result in a bad hash function. Recall that the divisor function **`f(k) = k%D`** is a uniform hash function for every key range **`[0, r]`** where **`r`** is positive integer. Let's take a closer look to the behavior of **`f(k)`** based on **`D`**.

Suppose that **`D`** is an even integer. Then **`f(k)`** is even whenever **`k`** is even, and **`f(k)`** is odd whenever **`k`** is odd. Thus, if the data set has many more even keys than odd then several keys get even home buckets than odd home buckets. This would result in a non-uniform assignment of home buckets. The same is also true for odd-weighed keys in a data set. Thus, choosing **`D`** to be even would produce a bad hash function.

Also so in practice, when **`D`** is divisible by small odd numbers such as 3, 5, and 7, hashing by division does not distribute the use of home buckets uniformly. Therefore, **`D`** should not be chosen as even or divisible by small odd numbers.

The ideal choice for **`D`** is a prime number. Whenever, one cannot find a prime number close to the table length (known, selected, or estimated a-priori), then **`D`** should be chosen to be not divisible by any number between 2 and 19. Best results are obtained when **`D`** (and subsequently the number of buckets **`b`**) is either a prime number or has no prime factor less than 20.

5. Hashing with Chains (Linked Lists)

Chains (or linked lists) provide a good option to solve the overflow problem in hashing. One can certainly create an array of linked lists associating with the "home buckets". That means several elements can share the same home bucket. **Figure 1** depicts a hash table in which overflows are handled by chaining.

To search for an element with key **`k`**, one should compute the home bucket (says **`k%D`**) for the key correspondence and then search the chain this bucket points to. To insert an element, one must verify that the table has no element with the same key. To delete an element with key **`k`**, one accesses the home bucket chain, search this chain for an element with the given key, and delete the element.

Note that each insert would be preceded with a search. So it may be a good preprocessing step to maintain the chains in some ordering fashion of the key values, such as ascending order.

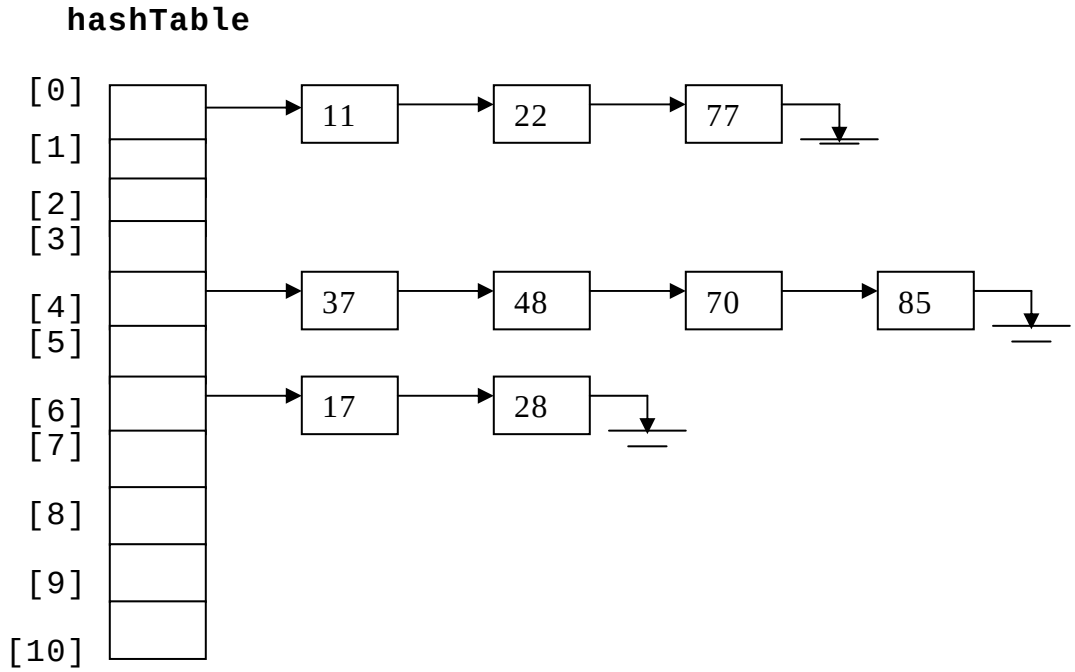

Figure 5. Hashing with chains using D = 11