Lecture 14.2

Topics:

1. Hash Function & Table – Introduction
2. Hash Functions & Techniques
3. Collision & Overflow Handling – Linear Probing

---

1. Hash Function & Table – Introduction

Hashing is a technique that may be used to search, insert, and remove elements belonging to table (data structure). Hashing is a randomization scheme that will not force the entire collection of elements to be sequentially checked out. Note that the data structures like linked list, stack, queue, tree do provide means to access their elements but only with restricted order.

Hashing deals with a collection that mostly means **dictionary**. What is a **dictionary**?

1.1 Dictionary

A dictionary is a collection of pairs of the form `(k, e)`, where **k** is a key and **e** is the element associated with the key (equivalently, **e** is the element whose key is **k**). No two pairs in a dictionary have the same key.

Certain operations can be performed on a dictionary. They are,

i. Getting/finding the element associated with a specific key from the dictionary.
ii. Inserting/putting an element with a specified key into the dictionary.
iii. Deleting/removing an element with a specified key.

For example, the class list for CIS 27 class is a dictionary with as many elements as students registered for the course. When a new student registered, an element/record corresponding to this student is inserted/added to the dictionary. When a student drops the course, his/her record may be deleted. The instructor may query (search) the dictionary to determine the record corresponding to a particular student and make changes to the record (e.g., add/change test or assignment scores). The student name may be used as the element key.

A dictionary with duplicates is similar to a dictionary as defined above. However, it allows two or more `(k, e)` pairs to have the same key.

For example, a word dictionary is a collection of elements. Each element consists of a word, the meaning of the word, the pronunciation, etc. Webster's dictionary contains an element (or entry) for the word "**`letter`**". Part of this element reads "A written or printed message to a person or a group, usually sent by mail in an envelope." Webster also has another entry for "**`letter`**" to say "A person who lets, or rents out, property. Webster may add new elements as new words created or as words take on new meaning. In data structures terminology, a word dictionary is a dictionary with duplicates.

In a dictionary with duplicates, there are rules to eliminate the ambiguity in the search and remove operations. That means, if one is to search (or remove) an element with a specified key, then which of the several elements with this key is to be returned (or removed)?

For examples, one can provide all values for selection then carry out the operations, or just operate on any one or all values. Some applications require a different form of the remove operation in which all elements inserted after a particular time are to be removed.

1.2 Ideal Hashing

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).`

Example 1

Consider the class list of CIS 27 class. Instead of using student names as the key, let's use student ID numbers as keys, which are nine (9) digits. Assume that there are at most 35 students and their IDs will be in the range 999999000 to 999999100.

The function `f(k) = k – 999999000` maps student IDs into table positions 0 to 100. Thus, we may use an array called `table[101]` to store pairs of the form `(k, e).` The array `table[]` is initialized so that `table[i]` is null for `i` is in `[0, 100].`

To search for an element with key `k`, we compute `f(k) = k – 999999000`. The element is at `table[f(k)]` provided that `table[f(k)]` is not `null`; the element can be removed if desired for this case. If `table[f(k)]` is null, the class list does not contain any element with key `k`; the element can be inserted if desired.

In the ideal situation as given in the above example, the table (or dictionary) needs to be initialized and appropriate values should be assigned after performing the insertion (**put**), searching and finding (**get**), and removal (**remove**) operations.

In many cases, the range in the key values is very large that a table would either not make sense or not be practical, or both.

Example 2

Suppose that in the class list of **Example 1** the student IDs are in the range of `[999000000, 999999999]` and the hash function `f(k) = k – 999000000`. That means the value of `f(k)` is in the range `[0, 999999].`

It does not make sense to use a table of this size for only 35 students. Besides wasting space, it takes time to initialize 1000000 entries to null.

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.

Example 3

An example is the compiler's symbol table that will be generated for every program to be compiled. The compiler records the information for all declared identifiers (variables, method names, class names, interface names, etc.) such as types, scope, and memory assignment. When it sees an identifier other than the declaration statements, the compiler will check to see if the name was declared. If it was declared, then the compiler would look up the appropriate information from in the symbol table; otherwise an error message would be issued.

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.

1.3 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.

Example 4 – Uniform/Non-Uniform Hash Function

Assume that our hash table has `b > 1` buckets numbered `0` through `(b-1)`. The hash function is defined as,

`f(k) = 0` for all `k = 0, 1, 2,` (Eqn. 1)

This is not a uniform hash function because all keys hash into the same bucket, which is bucket 0. When this hash is used, we get the maximum number of all-possible collisions and overflows.

Suppose that `b = 10` and the key range is `[0, 100].` A uniform hash function will hash approximately 10 keys into each bucket. Or, when the key range is `[0, 1000]` then the approximately 100 keys are hashed into each bucket.

The division function of `f(k) = k%b` is a uniform hash function for every key range `[0, r]` where `r` is a positive integer. When `r` is 22 and `b` is 10, some buckets will get two and some buckets will get three. When `r` is 50 and `b` is 11, some buckets will get 5 and some buckets will get 4. No matter what `r` and `b > 1` are, when the division is used, some buckets get `( r / b)` keys and the others get `( r / b + 1 )` keys.

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 (**Examples 1 & 2**) 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).` Let's consider one simple function using modulo approach -- division hash.

2.2.1 Division Hash Function

Hashing by division is the most common choice (others are multiplication, folding, etc.). Its hash function has the following form

```
f(k) = k%D                                                    (Eqn. 2)
```

where `k` is the key, `D` is the length (i.e., number of buckets) of the hash table. The positions in the hash table are indexed from 0 through `(D – 1)`.

For examples, when `D = 11`, the home buckets for the keys `3, 26, 18, 32` are `f(3) = 3, f(26) = 5, f(18) = 7, f(32) = 10`. This is shown is **Figure 1** below.

In this example, the values were also used as keys to determine the corresponding buckets. Each bucket has only one slot. If the value 63 is about to be inserted to the table, the home bucket is 3. This bucket is already occupied by a different value. Thus, a **collision** has occurred.

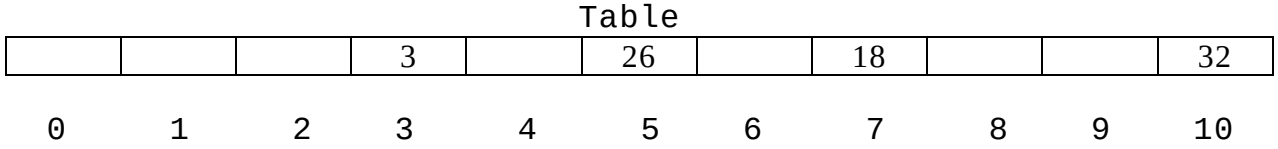

Figure 1. A division hash table

A collision occurs whenever two or more keys result in the same home bucket. If there are enough slots to hold the all values, then the collision is not a problem. However, if there are not enough spaces to hold the values, then we have an overflow.

Normally, we would try to avoid both the collisions and overflows at the same time, if possible. But this may not be feasible and we have to come up with a technique to handle these situations. Note that collisions may not be a big deal as long as we still have enough slots to store new values, but an overflow would be a problem. Where should we put the value?

2.2.2 Collision & Overflow-Handling Mechanisms -- Brief

Handling the overflow would mean that we would try to come up with a way to insert the value to some slot. Let's assume that the table only has one slot for every bucket. There are several techniques that would provide the resolutions to the collisions: Linear Probing, Quadratic Probing, chaining, etc. The discussion of the linear probing will be given next.

3. Collision & Overflow Handling -- Linear Probing

One approach to handle this collision is called linear probing. In this, it is to probe for an available slot (or, an address of an opening slot) that might be used to store the element; thus it also has the name open address hashing.

Let's consider the case where a bucket has only one single slot and there is a collision at $\mathbf{f(k_h)}$. 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).

Note that, in general, the open address hashing approach will have a larger table size (`TABLESIZE`) than the chaining because all data will be placed directly in a table. Let's consider the linear probing and quadratic probing approaches.

3.1 Insertion in Linear Probing

Let's consider the division hash function with the divisor of 11 (i.e., `D = 11`) below.

```
f(k) = k%D                                                    (Eqn. 4)
```

A single slot per bucket may have the placement as in **Figure 2** with the collision (and overflow) to occur for the next key of 69. Note that $\mathbf{k_h}$ is 69 and $f(k_h)$ is 3.

			3		26		18			32
0	1	2	3	4	5	6	7	8	9	10

Figure 2. A division hash table

Consider a linear probing function `p(i)`, which is a linear function (a function with order 1 for the variable **`i`**),of `i`. Let's assume a simple linear probing function as follows,

```
p(i) = i       i = 0, 1, 2,…                                  (Eqn. 5)
```

This would be similar to test the slots sequentially for an available (maybe empty!) slot. Thus, 69 will be placed in the next available slot shown in **Figure 3**.

			3	69	26		18			32
0	1	2	3	4	5	6	7	8	9	10

Figure 3. A division hash table with `p(k) = k`

Let's add 45 and 92 to the table. The result should look as in **Figure 4**.

	45		3	69	26	92	18			32
0	1	2	3	4	5	6	7	8	9	10

Figure 4. A division hash table

The next steps would be to consider the search and removal processes.

3.2 Searching

The search begins at the home bucket f(k) of the key k and continues by examining successive buckets in the table (a wrapped around or circular table) until on **`eof`** the following happens:

(1) A bucket containing the element with key k is reached, in which case we have found the element being searched for.
(2) An empty bucket is reached, in which case the table does not contain the element with key k.
(3) We return to the home bucket, in which case the table does not contain the element with key k.

3.3 Removal

The removal of an element must leave behind a table on which the search method described above should work correctly.

	45		3	69	26	93	18			32
0	1	2	3	4	5	6	7	8	9	10

For example, if we are to remove the element with key 69 from table of **Figure 3**, we cannot simple make position 4 of the table null. Doing so will result in the search method failing to find the element with key 92 (as given in condition 2 above).

A removal may require us to move several elements. The search for elements to move begins just after the bucket vacated by the removed element and continues to successive buckets until we either reach an empty bucket or we return to the bucket from which the deletion took place. When elements are moved up on a table following a removal of an element, we must take care not to move an element to a position before its home bucket, because making such element move would cause the search for this element to fail.

3.4 Removal – Alternative

An alternative to this cumbersome deletion strategy is to introduce the field `neverUsed` in each bucket. Let's look at the process.

(1) When the table initialized, this field is set to `true` for all buckets.
(2) When an element is placed into a bucket, its `neverUsed` field is set to `false`. With this, the condition (2) for the search above should be replaced as: a bucket with its `neverUsed` field equal to true is reached.
(3) We accomplish a removal by setting the table position occupied by the removed element to `empty`.

There are several issues that must be resolved such as increasing table size, selecting the divisor, data conversion, etc. The discussion can be found in many textbooks. The next few lectures will consider some of them.