Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Maps and Hashing

Maps/Dictionaries

Examples for this section are in `https://github.com/SienaCSISDataStructuresJDT/map-examples`.

A *map* or *dictionary* is a structure used for storing and looking up items via key-value associations.

An association matches a value with an immutable key. A generic version of such a structure is in `Association.java`. Note that there is no way to modify the key once we construct the `Association`, but the value can be modified.

An example similar to one we saw earlier in the semester is in `SpellsArrayAssociation`. Here, we look up values by key, and print out the corresponding value.

The utility of the `Association` becomes more apparent when we instead use an `ArrayList` as we see in `SpellsArrayListAssociation`.

We have replaced the search loop with a call to the `ArrayList`'s `indexOf` method. The parameter we pass will be compared with the elements of the list using the `Association`'s `equals` method, and that requires only that the keys are equal, regardless of value.

What is the Big-O cost in the best and worst cases for that `indexOf` method call?

What if we were to use an `ArrayList`-like structure that is ordered by keys?

- This only works if our keys are `Comparable` and the `Associations` using those keys are `Comparable` by the same criteria. (See `ComparableAssociation`).
- We would need to add the elements in order by the `Comparable` keys. (What is the best and worst case here?)
- We can then take advantage of the ordering to search in `indexOf`. (What is the best and worst case here?)

Under what circumstances might it be worth keeping these in an ordered array-based list?

Of course, we have seen other, more efficient, ordered data structures since we learned about `ArrayLists`.

Let's use a binary search tree to store our `ComparableAssociations` of key/value pairs. This is done in `SpellsBST`.

Now what are the costs of the operations?

A Quick Summary

There are many possible implementations of dictionary/map data structures. If we think abstractly about a few variants, we can consider their time complexity on the most common operations, and space complexity. In the table below, n denotes the actual number of elements in the map, and N denote the maximum number of elements it could contain (where there is such a restriction).

Structure	Search	Insert	Delete	Space
Linked List	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$	$O(N)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Array[KeyRange] of `EltType`	$O(1)$	$O(1)$	$O(1)$	KeyRange

That last line is very interesting – it represents the payoff for a big space-time tradeoff. If we know the range of possible keys (suppose they're all integers between 0 and KeyRange-1) we can use the key as the index directly into an array. And have constant time search, insert, and delete! But that's potentially a very large array.

Hashing

The map implementation of an array with keys as the subscripts and values as array elements makes a lot of sense. For example, we could store information about members of a team whose uniform numbers are all within a given range by storing each player's information in the array location indexed by their uniform number. If some uniform numbers are unused, there will be some empty slots, but it gives us constant time access to any player's information as long as we know the uniform number.

However, there are some important restrictions to overcome for us to be able to use this representation.

This implementation assumes that the data has a key which is of a type that can be used as an array subscript (some enumerated type in Pascal, non-negative integers in C/C++/Java), which is not always the case. What if we're trying to store strings or doubles or something more complex?

Moreover, the size requirements for this implementation could be prohibitive. Suppose we wanted to store 2000 student records indexed by social security number or a college ID number. We would need an array with 1 billion elements, all but 2000 of which would be empty!

In cases like this, where most of the entries will be empty, we would like to use a smaller array, but come up with a good way to map our elements into the array entries.

Suppose we have a lot of data elements of type `E` and a set of indices I in which we can store data elements. We would like to obtain a function H: `E` $\rightarrow I$, which has the properties:

1. $H(e)$ can be computed quickly

2. If $e1 \neq e2$ then $H(e1) \neq H(e2)$. (*i.e.*, H is a one-to-one function)

This is called a *perfect hashing function*. Unfortunately, they are difficult to find unless you know all possible entries to the table in advance. This is rarely the case.

Instead we use something that behaves well, but not necessarily perfectly.

The goal is to scatter elements through the array randomly so that they won't attempt to be stored in the same location, or "bump into" each other.

So we will define a *hashing function* H: `Keys` $\rightarrow$ `Addresses` (or array indices), and we will call `H(e.key)` the *home address* of `e`.

Assuming no two distinct elements have the same home address (*i.e.*, wind up mapping to the same array location), we have add, search and remove them in $\Theta(1)$ time.

Note, however, that we cannot efficiently visit our elements in order, as is possible with some of the other maps structures.

Since in general, a hashing function is not guaranteed to be one-to-one, it is possible that multiple distinct elements map to the same home address.

This suggests two problems to consider:

1. How to choose a good hashing function to minimize this occurrence?
2. How to resolve the situation where two different elements are mapped to same home address?

Selecting Hashing Functions

The following quote should be memorized and recited by anyone trying to design a hashing function:

"A given hash function must always be tried on real data in order to find out whether it is effective or not."

Data which contains unexpected and unfortunate regularities can completely destroy the usefulness of any hashing function!

We will start by considering hashing functions for integer-valued keys.

Digit selection As the name suggests, this involves choosing digits from certain positions of key (*e.g.*, last 3 digits of a Social Security Number).

To make sense, the size of the table would match the number of digits we select. So if we select 3 digits, the table should have size 1000.

Unfortunately it is easy to make a very unfortunate choice. We would need careful analysis of the expected keys to determine which digits will work best. Patterns in the keys can cause problems. We want to make sure that the expected keys are at least capable of generating all possible home addresses.

For example, the first digits of SSNs often reflect the region of the country in which they were assigned, and hence usually would work poorly as a hashing function.

Even worse, what about the first four digits of a Siena ID number? The first three digits of a phone number?

Division Here, we let `H(key) = key mod TableSize`

This is very efficient and often gives good results if the `TableSize` is chosen properly.

If it is chosen poorly then you can get very poor results. Consider the case where `TableSize` = $2^8 = 256$ and the keys are computed from ASCII equivalent of two letter pairs, i.e. $Key(xy) = 2^8 \cdot ASCII(x) + ASCII(y)$, then all pairs ending with the same letter get mapped to same address. Similar problems arise with any power of 2.

It is usually safest to choose the `TableSize` to be a prime number in the range of your desired table size. The default size in the structure package's hash table, for example, is 997.

Mid-Square Here, the approach is to square the key and then select a subset of the bits from the result. Often, bits from the middle part of the square are taken. The mixing provided by the multiplication ensures that all digits are used in the computation of the hash code.

Example: Let the keys range between 1 and 32000 and let the `TableSize` be $2048 = 2^{11}$.

Square the key and select 11 bits from the middle of the result. Note that such bit manipulations can be done very efficiently using shifting and bitwise and operations.

```
H(Key) = (key >> 10) & 0x07FF;
```

Will pull out the bits:

```
Key         = 123456789abcdefghijklmnopqrstuvw
Key >> 10   = 0000000000123456789abcdefghijklm
& 0x07FF    = 000000000000000000000cdefghijklm
```

Using r bits produces a table of size 2^r.

Folding Here, we break the key into chunks of digits (sometimes reversing alternating chunks) and add them up.

This is often used if the key is too big. Suppose the keys are Social Security numbers, which are 9 digits numbers. We can break it up into three pieces, perhaps the 1st digit, the next 4, and then the last 4. Then add them together.

This technique is often used in conjunction with other methods – one might do a folding step, followed by division.

What if the key is a String?

We can use a formula like - $Key(xy) = 2^8 \cdot (int)x + (int)y$ to convert from alphabetic keys to ASCII equivalents. (Use 2^{16} for Unicode.) This is often used in combination with folding (for the rest of the string) and division.

A very simple hash function for strings is to add together the ordinal equivalent of all letters and take the remainder mod `TableSize`.

```
hash = 0;
for (int charNo = 0; charNo < word.length(); charNo++)
    hash = hash + (int)(word.charAt(CharNo));
hash = hash % tableSize;  /* gives 0 <= hash < tableSize */
```

However, this has a potential problem – words with the same letters get mapped to same places:

miles, slime, smile

This is not going to be a very good hash function, but it's at least simple. It can be improved significantly by taking the letters in pairs before division.

We can see how the designers of Java decided to generate hashing functions for `String` objects. As you may know, all Java objects come equipped with a `hashCode` method (just like they all have a `toString` and a handful of other methods). The `hashCode` for Java `String`s is defined as specified in its Javadoc:

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the i^{th} character of the string, n is the length of the string, and ^ here indicates exponentiation. (The hash value of the empty string is zero.)

`https://github.com/SienaCSISDataStructuresJDT/HashCodes-example`

Handling Collisions

Recall that the home address of a key is the location that the hashing function returns for that key.

A *hash collision* occurs when two different keys have the same home location.

There are two main options for getting out of trouble:

1. *Open addressing*, also confusingly sometimes called *closed hashing*: if the desired slot is full, do something to find an open slot. This must be done in such a way that one can find the element again quickly!

2. *External chaining*, also called *open hashing* or *separate chaining*: make each slot capable of holding multiple items and store all objects that with the same home address in their desired slot.

For our discussion, we will assume our hash table consists of an array of `TableSize` entries of the appropriate type:

```
   T [] table = new T[TableSize];
```

Open Addressing

Here, we find the home address of the key (most commonly using the hashing function mod `TableSize`). If the slot at the home address happens to be occupied, we keep trying new slots until an empty slot is located.

There will be three types of entries in the table:

- an empty slot, that has never been occupied by an element, often represented by `null`
- deleted entries - marked by inserting a special entry indicating "empty after removal" (the need for this will soon become obvious), and
- normal entries which contain a reference to an object in the hash table.

This approach requires a *rehashing*. There are many ways to do this – we will consider a few.

First, *linear probing*. We simply look for the next available slot in the array beyond the home address. If we get to the end, we wrap around to the beginning:

```
Rehash(i) = (i + 1) % TableSize.
```

This is about as simple as possible. Successive rehashing will eventually try every slot in the table for an empty slot.

For example, consider a simple hash table of strings, where `TableSize = 26`, and we choose a (poor) hash function:

`H(key)` = the alphabetic position (zero-based) of the first character in the string.

Strings to be input are `GA`, `D`, `A`, `G`, `A2`, `A1`, `A3`, `A4`, `Z`, `ZA`, `E`

Here's what happens with linear rehashing:

0	1	2	3	4	5	6	7	8	9	10	...	25
A	A2	A1	D	A3	A4	GA	G	ZA	E	..	...	Z

In this example, we see the potential problem with an open addressing approach in the presence of collisions: *clustering*.

Primary clustering occurs when more than one key has the same home address. If the same rehashing scheme is used for all keys with the same home address then any new key will collide with all earlier keys with the same home address when the rehashing scheme is employed.

In the example, this happened with `A`, `A2`, `A1`, `A3`, and `A4`.

Secondary clustering occurs when a key has a home address which is occupied by an element which originally got hashed to a *different* home address, but in rehashing got moved to the address which is the same as the home address of the new element.

In the example, this happened with `E`.

When we search for `A3`, we need to look in 0,1,2,3,4.

What if we search for `AB`? When do we know we will not find it? We have to continue our search to the first empty slot! This is not until position 10!

Now let's delete `A2` and then search for `A1`. If we just put a `null` in slot 1, we would not find `A1` before encountering the `null` in slot 1.

So we must mark elements where deletions have occurred with a special value (not just make them empty). These slots can be reused on a new insertion, but we cannot stop a search when we see a slot marked as deleted. This can be pretty complicated.

Minor variants of linear rehashing would include adding any number k (which should be relatively prime to `TableSize`), rather than adding 1.

If the number k is divisible by any factor of `TableSize` (i.e., k is not relatively prime to `TableSize`), then not all entries of the table will be explored when rehashing. For instance, if `TableSize = 100` and k = 50, the linear rehashing function will only explore two slots no matter how many times it is applied to a starting location.

Often the use of $k = 1$ works as well as any other choice.

Next, we consider *quadratic rehashing*.

Here, we try slot `(home +` j^2`) % TableSize` on the j^{th} rehash.

This variant can help with secondary clustering but not primary clustering.

It can also result in instances where in rehashing you don't try all possible slots in table.

For example, suppose the `TableSize` is 5, with slots 0 to 4. If the home address of an element is 1, then successive rehashings result in 2, 0, 0, 2, 1, 2, 0, 0, ... The slots 3 and 4 will never be examined to see if they have room. This is clearly a disadvantage.

Our last option for rehashing is called *double hashing*.

Rather than computing a uniform jump size for successive rehashes, we make it depend on the key by using a *different* hashing function to calculate the rehash.

For example, we might compute

```
delta(Key) = Key % (TableSize -2) + 1
```

and add this delta for successive rehash attempts.

If the `TableSize` is chosen well, this should alleviate both primary and secondary clustering.

For example, suppose the `TableSize` is 5, and `H(n) = n % 5`. We calculate delta as above. So while `H(1) = H(6) = H(11) = 1`, the jump sizes for the rehash differ since `delta(1) = 2`, `delta(6) = 1`, and `delta(11) = 3`.

External Chaining

We can eliminate some of our troubles entirely by allowing each slot in the hash table hold as many items as necessary.

The easiest way to do this is to let each slot be the head of a linked list of elements.

The simplest way to represent this is to allocate a table as an array of references, with each non-`null` entry referring to a linked list of elements which got mapped to that slot.

We can organize these lists as ordered, singly or doubly linked, circular, etc.

We can even let them be binary search trees if we want.

Of course with good hashing functions, the size of lists should be short enough so that there is no need for fancy representations (though one may wish to hold them as ordered lists).

There are some advantages to this over open addressing:

1. The "deletion problem" doesn't arise.
2. The number of elements stored in the table can be larger than the table size.
3. It avoids all problems of secondary clustering (though primary clustering can still be a problem – resulting in long lists in some buckets).

Hashing Analysis

We can measure the performance of various hashing schemes with respect to the *load factor* of the table.

The load factor, α, of a table is defined as the ratio of the number of elements stored in the table to the size of the table.

$\alpha = 1$, at least for open addressing, means the table is full, $\alpha = 0$ means it is empty.

Larger values of α lead to more collisions.

(Note that with external chaining, it is possible to have $\alpha > 1$).

The table below (from Bailey's Java Structures) summarizes the performance of our collision resolution techniques in searching for an element.

Strategy	Unsuccessful	Successful
Linear probing	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$
Double hashing	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\frac{1}{(1-\alpha)}$
External chaining	$\alpha+e^{-\alpha}$	$1+\frac{1}{2}\alpha$

The value in each slot represents the average number of compares necessary for a search. The first column represents the number of compares if the search is ultimately unsuccessful, while the second represents the case when the item is found.

The main point to note is that the time for linear rehashing goes up dramatically as α approaches 1.

Double hashing is similar, but not so bad, whereas external chaining does not increase very rapidly at all (linearly).

In particular, if $\alpha = .9$, we get:

Strategy	Unsuccessful	Successful
Linear probing	55	$\frac{11}{2}$
Double hashing	10	~ 4
External chaining	3	1.45

The differences become greater with heavier loading.

The space requirements (in words of memory) are roughly the same for both techniques. If we have n items stored in a table allocated with `TableSize` entries:

- open addressing: `TableSize` + $n\cdot$ objectsize
- external chaining: `TableSize` + $n\cdot$ (objectsize + 1)

With external chaining we can get away with a smaller `TableSize`, since it responds better to the resulting higher load factor.

A general rule of thumb might be to use open addressing when we have a small number of elements and expect a low load factor. We have have a larger number of elements, external chaining can be more appropriate.

Hashtable Implementations

Hashing is a central enough idea that all `Objects` in Java come equipped with a method `hashCode` that will compute a hash function. Like `equals`, this is required by `Object`. We can override it if we know how we want to have our specific types of objects behave.

Java has long included hash tables in `java.util.Hashtable`, and more recently in `java.util.HashMap`, which is an extention of the `Map` interface.

It uses external chaining, with the buckets implemented as a linear structure. It allows two parameters to be set through arguments to the constructor:

- `initialCapacity` - how big is the array of buckets
- `loadFactor` - how full can the table get before it automatically grows

Suppose an insertion causes the hash table to exceed its load factor. What does the resize operation entail? Every item in the hash table must be rehashed! If only a small number of buckets get used, this could be a problem. We could be resizing when most buckets are still empty. Moreover, the resize and rehash may or may not help. In fact, shrinking the table might be just as effective as growing it in some circumstances.

There are other variations on the hash map in the Java Collections framework.