Hashing
Computer Science 1321
Spring Semester 2012

0. Introduction:

Ideas associated with searching abound in computer science. Last semester we studied searching sequentially on unordered data sets, and also studied the binary search on ordered data. We learned that the sequential search requires time proportional to the size of the data set (O(n)), whereas the binary search was much better, requiring time proportional to the log of the size of the data set (O(ln(n))). This meant that for a data set of size 1000 elements, the sequential search might require as many as 1000 comparisons to determine whether a particular item was present in the data, whereas if the data were sorted, we could determine whether a particular item was in the data in as few as 10 comparisons.

But, every time you get something it seems like you have to give something up. In order to achieve this remarkable improvement in searching for a particular item using the binary search, we have the overhead of maintaining a sorted collection. Without a sorted collection, we are reduced to searching sequentially.

The overriding question at this point is: can we do any better? Are there other ways of storing the data than in simple lists (arrays) of values in either unsorted or sorted order? The answer of course is yes!

1. Hashing:

The idea is to develop a method of information storage and retrieval that will allow essentially direct access to any element of the collection in constant time (O(1)). How is this possible, or, is it possible? And what sort of scheme should we use to perform this magical storage system?

The ideas behind hashing are actually relatively straightforward. As an example, suppose we have a collection of strings we wish to store. (We could equally as well use numbers). We don't know in advance how many strings there are, but know that there are only finitely many of them. We decide on what is termed a “hash table.” A hash table may, in practice, be nothing more than a python list, or some other similar data structure. Each position in the hash table is referred to as a “slot.” Most often, a slot is merely a cell in the list with a particular index.

We must provide a function, termed a “hash function” which will take one of the strings we wish to store, and return an index in the table as its storage location. If h is the hash function, and s is the string we wish to store, then

index = h(s)

is to provide the index in the hash table where the string is to be stored. Many different hash functions are possible, and an entire theory has been devoted to providing the best kind of hash function. We shall keep things simple, however. The theory of hashing remains the same.

As an example, assume we wish to store strings in a hash table of size 23. As you will see later, having the size of the hash table a prime number can be an advantage. If we wish to store a string, we have to

find an index position in the list to determine a slot to place our string. A simple hash function might be given by the following scheme

```
def h(s):
        v = 0
        for x in s:
                v += ord(x)
        return v % 23
```

This function merely adds the ordinal values of each of the characters in the string, then finds the remainder upon division by the table size. This provides an index in the appropriate range, and we may store the value at that slot.

For example, using the above hash function, if s = 'computer' then the sum of the ordinal values of the letters in s is 879 (check it). 879 % 23 is 5, thus we store the string 'computer' in slot 5 of the hash table (list).

This is only one example of many possible hash functions.

Moreover, the string itself may be a key value in a much more complex data structure we wish to store away. We use the string as a simple example. In most cases the items stored in a hash table will be a key, value pair. The key is used to determine the location in the hash table where the value is stored. We emphasize that the value could be a complex data structure holding many items. The key is hashed to store it, retrieve it, or delete it entirely. (See student experiments).

Retrieving the value from the hash table involves applying the hash function (hashing) the keyword, then going to that slot in the hash table to get the information stored there. (The nice thing about Python lists are the fact that virtually anything can be stored in a list.) Calculating the hash function can be done in constant time (it is essentially the same all the time), thus we have achieved our desired O(1) performance.

This would be the end of the story, except for the fact that it isn't quite that easy. What if two different words have the same hash value? We would then attempt to store them in the same slot of the hash table. A collision has occurred. Now it gets interesting.

2. Collision resolution:

In the first place, if we have a reasonably large amount of data to store in our hash table, we must make the table sufficiently large so that we have a reasonable chance of success when it comes to storing a particular value. We don't want the hash table to be too large, however, since then we would have a bunch of space that wouldn't be used. On the other hand, we don't want to make the hash table too small, since then we would encounter more collisions than we could reliably handle. (This is one of the places where computer science is still somewhat of an art, rather than a science, since we simply have to guess what a good hash table size might be.) Moreover, even though we make the hash table quite large there is still the possibility of a collision. We thus have to have some sort of strategy for handling collisions when they occur. The term for searching for another slot to store a value is rehashing.

None	None	None	None	None	None	None	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10

Initially, for example, we have a hash table with 11 slots, and indices from 0 through 10. Note that the python keyword None is used to mark an empty slot. As we calculate our hash function and begin to store values in the table, certain of the slots will fill. Then the table might look like:

None	None	value1	None	value2	None	value3	None	None	None	None
0	1	2	3	4	5	6	7	8	9	10

The load factor of a hash table is defined to be the ratio of the number of cells filled to the table size. The load factor is generally denoted by the Greek letter lambda (λ). In this case the load factor is

$\lambda = 3/11$.

We would be happy as a clam if everything had its own unique position in the hash table. This utopian world, however, does not exist. There is also theory concerning how high you should let the load factor get before you increase the size of the hash table. (Generally increasing the size of the hash table is undesirable because of the overhead involved).

If a collision does occur, what do we do?

The simplest method of handling collisions is to use a technique called “linear probing.” Using this technique, if we hash value1 and store it in slot x, then hash value2, which also hashes to x, we simply look to the right. If the next slot is empty, we store value2 in that slot. If it isn't, we continue to look to the right and store value2 in the next available empty slot to the right. If we reach the end of the list, we use modular arithmetic and wrap to the beginning of the list and continue to search from there for an open slot. The first available open slot is used to store value2. If value3 hashes to the same slot as value1 and value2, then again we search to the right for the next available slot and store value3 there.

As inelegant as this technique is, it works. Moreover, it is used in practice. However, it has several deficiencies. In particular, if there are many collisions, the fundamental premise of developing hashing is destroyed, since the problem of storing values degenerates to a sequential search for an open slot. Hopefully, however, the hash table is sufficiently large, the hash function is discriminating, and the load factor is not so high as to allow lots of collisions. If that's the case then we may find an open slot in a few looks. Another problem with linear probing is that the data elements being stored tend to cluster. If value1 hashes to slot x and value2 hashes to slot x, then value2 is stored in x+1. But value3 might hash to slot x+1, in which case it is stored in slot x+2. You can see how clustering might happen.

Other techniques have been developed for collision resolution. Your textbook describes a technique called “Plus 3” to avoid the clustering problem. If value1 hashes to slot x, and value2 also hashes to slot x, then value 2 is stored in slot x+3 if it is available. If not, it is stored in slot x+6, etc.

Your textbook also discusses quadratic probing. If value1 hashes to slot x, we then probe slots x+1, x+4, x+9, etc. to resolve collisions.

3. Chaining:

An alternative method to resolve collisions in a hash table is to use chaining. We know by now how valuable Python lists are. Could we make another use of a Python list?

Suppose we envision our empty hash table as follows:

[]	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]
0	1	2	3	4	5	6	7	8	9	10

Each of the slots of the hash table is an empty Python list. As values are hashed and stored in the hash table, values that hash to the same slot are merely **append**ed to the list. We know whether a slot is empty, since we may check for an empty list. In this way, all of the data items with the same hash value are in the list in that slot. Some of the same problems still remain with this technique, however. We wouldn't want the chained lists to become too large, since again the hashing problem would be reduced in many cases to a simple sequential search. (It occurs to me that it may be the case that the lists could be maintained in sorted order, and if they became large a binary search could be used instead. But then a simple append could not be used – we would need an insert).

But what of programming languages (like C and Java for example) that don't have a nice builtin data structure like a Python list? You recall that just before the first test in this class we discussed a "nodes and references" approach to creating our own linked list. This technique could certainly be used to implement chaining. In this case, each of the empty slots in the hash table could contain a None reference. As values are hashed to the same slot, we may simply add values to the linked list at that slot. A bit harder than using Python lists, but the idea is the same.

I encourage you to read the discussion of your textbook on pages 147-158.

4. Student Experiments:

Task One: Create a hash table class. The class should have the functionality:

HashTable(size) returns a new hash table with size empty slots

Store (key, data) stores a data item using the key as the location. That is, we will apply the hash function to the key to find out where the data is to be stored in the hash table.

Search(key) returns the data item stored in the hash table using key. This method should return None if the hash table slot indicated by key is empty.

Delete(key) removes the data hashed by key from the hash table.

LoadFactor() returns the current load factor for the hash table.

Display() shows a view of the hash table. We probably wouldn't do this in the real world, but in this learning environment, we want to see what is happening.

Task Two: Add linear probing to the hash table class you have created. It must have some method for

collision resolution. Linear probing is a standard approach.

Task Three: Implement chaining using Python lists. Modify your Display() function so we can see the effect.

Task Four: (extra) Implement chaining using nodes and references linked lists. This is slightly more difficult but is a nice example of an application of linked lists.

As a simple example, you may assume that the key is a positive integer, and the value to be stored is a string. In this way we have a simple (key, value) pair. This means that the hash function can be defined as

h(key) = key%size

where size is the size of the hash table.

As with the previous modules, make a directory called

```
youruserid.hashing
```

and place everything you wish me to see in this folder. If you have any special tricks or techniques you wish me to be aware of, include a README text file with the directions for how you would like me to exercise your code.

Copy your folder into my public directory using the command

```
cp -r youruserid.hashing /users/meggen/Public
```

on or before the due date.

If you have any questions send your instructor an email or post on Piazza.