Hash Tables

9/17/12 2:27 AM

Assignment

In this assignment, you will be implementing a hash table with linear probing. You are encouraged to use Junits for testing.

What are Hash Tables?

Suppose we have a dictionary, you know those things that have a set of words and their definitions. You want to be able to look up the definition of any word, very quickly, so you flip to the letter tab and start searching. Well hash tables work the same way; we use keys to access values. In the example above the word is the key and the value is the definition.

Now Going back to the dictionary example, one of the annoying things that happens when we flip to the tab we want is that we have to search until we find the word we are looking for. This happens because there are a lot of collisions for said lettered word at that tab. To make this faster we need a collision handling protocol. There are two main types separate/external chaining and linear probing. For this assignment we will be doing linear probing.

Linear probing

Linear probing says that when a collision occurs (two unique keys hash and compress to the same index), add the entry to the next available spot.
This is just a brief introduction to hash tables with linear probing. If you are already comfortable with linear probing you can skip to the next section (the requirements for this assignment).

Adding to a Hash Table

Consider a hash table of length 5 mapping integer keys to string values. Our hash function will simply be the key itself.

0	
1	
2	
3	
4	

Let's add the key-value pair (3, "Hello") to the hash table. The key, 3, has a hash code of 3. We compress the key by taking the hash code modulo 5, the size of the table, and get an index of 3. Slot 3 is open in the array, so we can just add the pair:

0	
1	
2	
3	(3,"Hello")
4	

Next let's try to add (6, "World"). The key, 6, has a hash code of 6. We compress that to 6 % 5 = 1. The slot at index 1 is open, so we can just add the pair:

0	
1	(6,"World")
2	
3	(3,"Hello")
4	

Finally, let's try adding (8, "foo"). The key, 8, has a hash code of 8. We compress that to 8 % 5 = 3. The slot at index 3 is already occupied, so we will traverse the array until we find an open slot. We find

0	
1	(6,"World")
2	
3	(3,"Hello")
4	(8, "foo")

Removing from a Hash Table

Removing from a hash table with linear probing works similarly. Let's remove the key 6. We calculate the hash code of 6, which is 6, and compress that to the size of our table. This gives us the index 6 % 5 =1. We look at index 1 to see if the key there is indeed 6. It is, so we remove it.

0	
1	
2	
3	(3,"Hello")
4	(8, "foo")

But actually removing the element from the table can cause problems elsewhere. Consider the case of removing 3:

0	
1	
2	
3	
4	(8, "foo")

Now, if we try to remove 8 we will calculate the index to be 3, but when we look at index 3 in the table we will see that it is empty, and therefore an entry with key

8 must not exist. However, we know that there was a collision previously, and that the entry that caused the collision is no longer there.

We can solve this problem by simply iterating over every slot in the array, but that would not give us the O(1) runtime that we are trying to obtain. So instead of actually removing the data, we just want to mark the entry as being available for overwriting.

This is how the table would look when using an "available" flag: (— means available)

0	
1	----
2	
3	----
4	(8, "foo")

Now if we try to remove 8 we can calculate its index to be 3 and see that there was once an entry at index 3. So we should continue looking until we either find the entry we are looking for, or find an empty slot.

0	
1	----
2	
3	----
4	----

Now if we try to add the pair (13, "bar"), the available flag in slot 3 tells us that that slot is available for overwriting, so (13, "bar") will go there:

0	
1	----
2	
3	(13, "bar")
4	----

Requirements

HashTableEntry

The first step to implementing *java.util.Map* is to implement the *java.util.Map.Entry* interface. Implementations of this interface store the individual key-value pairs. Implement this interface in the **HashTableEntry** class.

Because we are storing data of unknown types, this class should have generics for the key and value.

The declaration for this class should look like this:

```
import java.util.Map;

public class HashTableEntry<K, V> implements Map.Entry<K, V>
```

This class should have three instance variables, the key, the value, and a boolean available to flag whether or not this entry is available. You will need the following methods:

1. A constructor to take in the key and value (in that order). available should be set to false by default.

2. isAvailable() and setAvailable(), the getter and setter for available.

3. equals() and hashCode(), as defined by the documentation in the *Map.Entry* API entry.

4. The other methods required by *Map.Entry*. Two of these are simple getters, but setValue is unique in that it must return the old value. Notice that there is no setter for the key instance variable.

LinearProbingHashTable

Create a class called **LinearProbingHashTable** that implements **GradableMap**. The **GradableMap** interface is a sub-interface of *java.util.Map* and defines additional methods that are required for us to grade your hash table.

In previous homework assignments, we often indicated "bad inputs" or other exceptional conditions by returning null, -1, or something similar. However, in this assignment we are writing our class according to the specifications of *java.util.Map*, so we will be throwing exceptions to signal these conditions.

LinearProbingHashTable should only require three instances variables:

1. An array of entries
2. The max load factor (as a double)
3. The size (as an int).

It will need the following methods:

1. Three constructors:
    a. One to take in a size and max load factor.
    b. One to take in just a size and use .75 as the max load factor.
    c. One to take in no parameters and use 11 as the size and .75 as the max load factor.

2. void clear(), to reset the size to 0 and the entries array to an empty array.

3. boolean containsKey(Object key), to check if the Object (not generic!) key is in the table. On average, this should happen in O(1) time.

4. boolean containsValue(Object value), to check if the Object (not generic!) value is in the table. See we don't know where this value might be in the table (we don't know the value's key), we have no choice but to iterate over the entire entries array in O(n) time.

5. Set<Map.Entry<K, V>> entrySet(), which returns a set containing all of the **HashTableEntry** objects from the entries table.

6. V get(K key), to look for a value in the table:
    a. If the key is in the table, return the value associated with that key.
    b. If the key is not in the table, return null.
    c. If key is null, throw a **NullPointerException** immediately. On average, this should happen in O(1) time.

7. boolean isEmpty(), which returns true if the size of the table is 0, and false otherwise.

8. Set<K> keySet(), which returns a *java.util.Set* containing the keys currently in the table. This is the first difference between our implementation and the contract of *java.util.Map*. In our implementation, you may simply create an instance of *java.util.HashMap* and add all of the keys. In the fully-compliant implementation you would have to write a new class that implements *java.util.Set* and is backed by the hash table. See the Java API for more information about this, if you are interested.

9. V put(K key, V value), which adds a new value to the hash table:
    a. If key is non-null and was not previously in the table, add the entry and return null.
    b. If key is non-null and was previously in the table, add the entry and

return the old value (the one that was just overwritten).

   c. If key is null, throw a **NullPointerException** immediately. On average, this should happen in O(1) time. Remember that you will have to resize when the max load factor is exceeded; that is, if the max load factor is .75 and the current load factor is also .75, do not resize.

10. void putAll(java.util.Map<? extends K, ? extends V> map), which simply iterates over the entries in the given Map and adds those values to your hash table. Keep in mind that this parameter may not be an instance of your hash table class, so you cannot cast the Map to **LinearProbingHashTable**. This operation should run in O(n) time.

11. V remove(Object key), which removes the entry with the given key:

   a. (a) If key is non-null and the entry exists, remove the entry and return the associated value.
   b. (b) If key is non-null and the entry does not exist, return null.
   c. (c) If key is null, throw a **NullPointerException**. On average, this should happen in O(1) time.

12. Collection<V> values(), which returns an implementation of *java.util.Collection* containing all of the values in the hash table. This is the second difference between our implementation and the contract of *java.util.Map*. In our implementation, you may simply create an instance of *java.util.ArrayList* and add all of the keys. In the fully-compliant implementation you would have to write a new class that implements *java.util.Collection* and is backed by the hash table. See the Java API for more information about this, if you are interested.

13. The other getters and setters, as required by the **GradableMap** interface.

Deliverables

Submit all the files required to run your solution to T-Square. For this assignment, you should submit (at a minimum):

1. GradableMap.java
2. HashTableEntry.java
3. LinearProbingHashTable.java
4. Any other files needed to run your program (if necessary).