CS 2110 Summer 2011: # tables

Due at the end of class

Work in groups of three or four. Write a single set of answers per group on this OR a separate sheet of paper and turn it in at the end of class, with the names and netids of all group members. It will not be graded, but your involvement in the activities will affect your grade for participation.

The hashtable is the final installment in 3-5 part series on how to implement a Map. The goal of this exercise is to get you thinking about some of the implementation details and how those design choices impact the correctness/efficiency of the map.

1 Open addressing, table doubling

Recall that to add an entry under open addressing, the hash is computed to identify the index \( i \). Then a process of array probing is carried out to identify an available position. With linear probing, each successive position is checked; with quadratic probing, the distance from the start index increases quadratically (i.e., if \( h(key) = i \), then it checks \((i + j^2) \mod m\) for \( j = 0, 1, 2, \ldots \)).

(a) With open addressing, the remove method finds the key to be removed and marks the corresponding entry as “available” rather than “empty” (i.e., \texttt{null}). Why?

(b) How does this affect performance? Hint: consider a hashtable with very high churn (many additions followed by many deletions).

(c) With linear probing, if there is an available spot, probing is guaranteed to find it. Is the same true for quadratic probing? Hint: Consider an array where \( m = 16 \).
(d) When a hashtable gets overloaded, its size is doubled, similar to what happens with an ArrayList. However, before copying entries over to the new array, they are rehashed. Why? Give it concrete example using a (small!) table that shows what can go wrong.

2  hashcode() contract

(a) As you may recall, on the pretest we defined a CornellStudent class with an overriding definition for equals. However, we did not override hashCode – oops! Therefore hashCode was inherited from the Object class. Object.hashCode returns an integer that essentially corresponds to the memory address of the object, so two objects are unlikely to return the same value for hashCode(). Nevertheless, will the statement below always print false? Why or why not?

```java
HashMap<CornellStudent, String> dormDirectory =
    new HashMap<CornellStudent, String>();
...
CornellStudent st1 = new CornellStudent("joe");
CornellStudent st2 = new CornellStudent("joe");
dormDirectory.put(st1, "Baker");
System.out.println(dormDirectory.containsKey(st2));
```

(b) Suppose we implement hashCode to be a function of student name and net id (the same things being compared between student objects in the equals method). Suppose also that a CornellStudent is a mutable object, meaning that it is possible to change its internal state (e.g., with a setter method). What is the following statement likely print? And why do I say “likely”?

```java
```
... CornellStudent st1 = new CornellStudent("joe");
dormDirectory.put(st1, "Baker");
st1.setNetID("byeJoe123");
System.out.println(dormDirectory.containsKey(st1));

3 Implement hashtable with buckets

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a linked list. All elements that hash into the same location will be stored in the list.

Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken after dividing by the table size. (Keep mind hashCode can return a negative number due to overflow.) This yields a table index. Next, the linked list indicated by the table index is examined. For the latter step, you can take advantage of the methods that a linked list provides (though keep efficiency in mind).

The load factor ($\lambda$) is defined as the number of elements divided by the table size. In this structure, the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running times of contains and remove are proportional to the length of the list, they are $O(\lambda)$. Therefore the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address hash table) if the load factor becomes larger than 10.

When the array is doubled, entries must be rehashed. Since recomputing hashCode can be expensive, it often makes sense to cache the value of hashCode with the item so that it does not need to be recomputed. (How might you do this? I.e., where should the cached value be stored?)

Implement a class called HashTable<E> with methods

- void add(E item)
- boolean contains(E item)

---

1Adapted from an exercise by Timothy Budd.
• **void remove(E item)**

You will also probably want a private `resize()` method. You can decide on whatever private variables make the most sense based on the information that you need to keep track of.

(Bonus) How would you implement an `Iterator<E>` for a hashtable?