CSCI 230 – Homework 6

Hashing

Objectives

- Implement hashing.
- Test the implementation thoroughly with tests that cover all possible use cases.

Background information

Hashing from Shaffer textbook

Assignment (from openDSA old pdf version of the book found in https://people.cs.vt.edu/~shaffer/Book/JAVA3elatest.pdf)

Implement a Dictionary ADT with a Hash Table

Details:

Implement the dictionary ADT that is pasted below, by means of a hash table with linear probing as the collision resolution policy. You might wish to begin with the code of Figure 9.9 (see below). Using empirical simulation, determine the cost of insert and delete as α grows (i.e., reconstruct the dashed lines of Figure 9.10, see below). Then, repeat the experiment using quadratic probing and pseudo- random probing. What can you say about the relative performance of these three collision resolution policies?

Dictionary ADT

```
/** The Dictionary abstract class. */
public interface Dictionary<Key, E> {

  /** Reinitialize dictionary */
  public void clear();

  /** Insert a record
      @param k The key for the record being inserted.
      @param e The record being inserted. */
  public void insert(Key k, E e);

  /** Remove and return a record.
      @param k The key of the record to be removed.
      @return A maching record. If multiple records match
      "k", remove an arbitrary one. Return null if no record
      with key "k" exists. */
  public E remove(Key k);

  /** Remove and return an arbitrary record from dictionary.
      @return the record removed, or null if none exists. */
  public E removeAny();

  /** @return A record matching "k" (null if none exists).
      If multiple records match, return an arbitrary one.
      @param k The key of the record to find */
  public E find(Key k);

  /** @return The number of records in the dictionary. */
  public int size();
};
```

Figure 4.29 The ADT for a simple dictionary.

Dictionary Implemented using Hashing

```
/** Dictionary implemented using hashing. */
class HashDictionary<Key extends Comparable<? super Key>, E>
                     implements Dictionary<Key, E> {
  private static final int defaultSize = 10;
  private HashTable<Key,E> T; // The hash table
  private int count;          // # of records now in table
  private int maxsize;        // Maximum size of dictionary

  HashDictionary() { this(defaultSize); }
  HashDictionary(int sz) {
    T = new HashTable<Key,E>(sz);
    count = 0;
    maxsize = sz;
  }

  public void clear() {   /** Reinitialize */
    T = new HashTable<Key,E>(maxsize);
    count = 0;
  }

  public void insert(Key k, E e) {  /** Insert an element */
    assert count < maxsize : "Hash table is full";
    T.hashInsert(k, e);
    count++;
  }

  public E remove(Key k) {  /** Remove an element */
    E temp = T.hashRemove(k);
    if (temp != null) count--;
    return temp;
  }
```

```
  public E removeAny() {  /** Remove some element. */
    if (count != 0) {
      count--;
      return T.hashRemoveAny();
    }
    else return null;
  }

  /** Find a record with key value "k" */
  public E find(Key k) { return T.hashSearch(k); }

  /** Return number of values in the hash table */
  public int size() { return count; }
}
```

Figure 9.9 A partial implementation for the dictionary ADT using a hash table. This uses a poor hash function and a poor collision resolution policy (linear probing), which can easily be replaced. Member functions **hashInsert** and **hashSearch** appear in Figures 9.6 and 9.7, respectively.

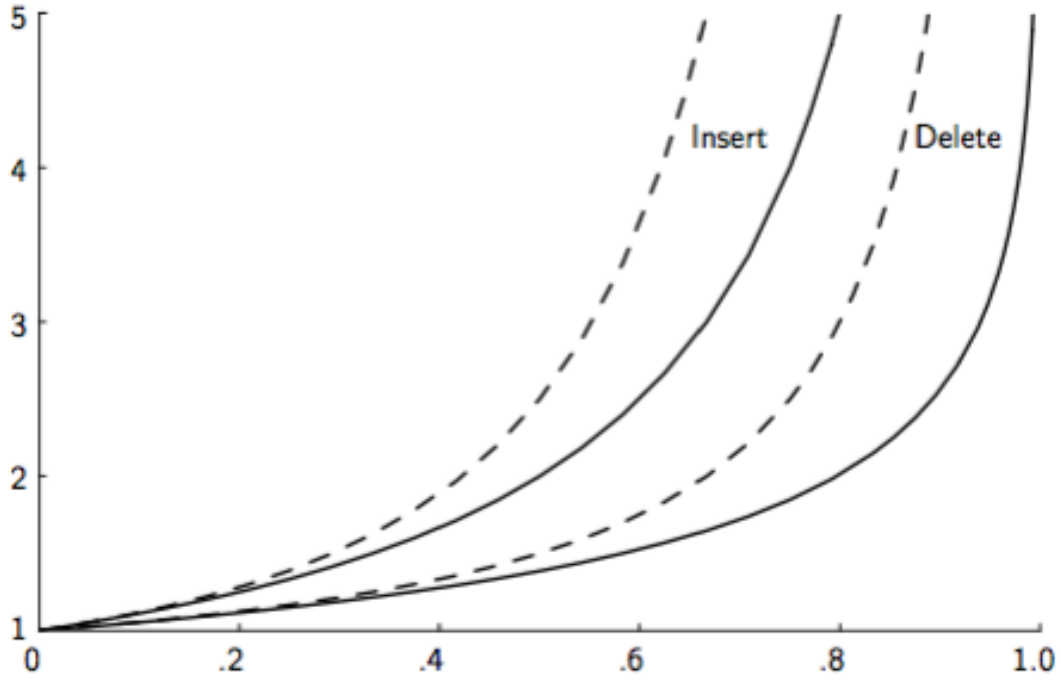

Figure 9.10 Growth of expected record accesses with α. The horizontal axis is the value for α, the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for "random" probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two leftmost lines show the cost for insertion (equivalently, unsuccessful search); the two rightmost lines show the cost for deletion (equivalently, successful search).

What to submit:

- A zipped folder containing all of your java files **and no subdirectories/folders**.
- One of the files must contain the **HashDictionary** class in a file naturally named **HashDictionary.java** The filename and class name must match exactly what is in bold.
- One class named **HW6** stored in a file named **HW6.java** that contains a main method that demos your dictionary and what works and specifies in comments what doesn't.
- Any other java files required of your solution. // There might not be any.
- A README that includes:
    - A plot of the linear probing vs quadratic probing costs, similar to 9.10.
    - An explanation of your experiments and conclusions in one paragraph.
- Put **Hash**Dictionary.java, HW6.java, and any other java files needed in a directory named YourLastNameFirstInitial_HW6. Then compress (zip) that directory. Then you will have a folder YourLastNameFirstInitial_HW6.zip. This is what you need to submit for HW6.