Laboratory 10: Dictionaries and Hashing

A dictionary is a data structure that records a list of associations between two different objects, such as a word and its definition, or a name and a phone number. This week we will explore applications of dictionaries as well as writing a method for an array-based dictionary. We will also look at how hashing can speed access to items stored in a dictionary.

Dictionary Operations
- **add(key, value)**
- **remove(key)**
- **getValue(key)**
- **contains(key)**
- **getKeyIterator()**
- **getValueIterator()**
- **isEmpty()**
- **getSize()**
- **clear()**

Part A: Dictionaries and Client Applications Of Them

Begin by opening the lab10a project folder.

1. To begin, open the TestDriver program and create a sample dictionary of names (String) and phone numbers (also String). Experiment with adding names and numbers, and retrieving and updating them as well.
2. Then uncomment the call to testDictionary and run the tests, verifying proper operation of the SortedArrayDictionary
3. Open EZ_Dictionary and define a dictionary class by completing the method definitions. Use two separate ArrayLists to store the keys and values.
4. Back in TestDriver, modify the testDictionary method to test your EZ_Dictionary.
5. Finally, open the WordFrequency program, and complete the code according to the instructions so it computes a word frequency table for chapter 1 of Moby Dick. Use the SortedArrayDictionary to hold the words.
6. [Optional] Add a method getKey() to your EZ_Dictionary class that returns the key associated with a given value. Test this method in the demo section of TestDriver. Using your original phone book dictionary, you should be able to do a “reverse” lookup of a person’s name given their phone number.
7. [Optional] Modify EZ_Dictionary to maintain either keys or values in sorted order. The constructor should take a parameter so when you create a dictionary you can indicate how you want it sorted. For example, `new EZ_Dictionary(EZ_Dictionary.ASCENDING_KEYS)` will make a dictionary where the keys will always be in ascending order, while
   `new EZ_Dictionary(EZ_Dictionary.DESCENDING_VALUES)` will make a dictionary where the values will always be in desceinding order. Your add method should to use sequential search to locate the position to insert an item, and use the same position to add both key and value. This way, the using iterators to list the key/value pairs will show them in the order requested. THEN switch the WordFrequency program to use EZ_Dictionary instead, so you can show the words sorted by most frequent to least frequent.

Part B. Hashing and Dictionaries

In Part B you will work on a Hashed implementation of a Dictionary. Code for this implementation is in the lab10b folder. There is also a Name class and a Driver to verify its performance. The Name class has a hashCode method that computes a hash code based on the first letter of the first and last names. Take a look at the Driver class and notice how it executes.

The HashedDictionary class uses linear probing in both the locate and the probe methods. However, there is code present in these methods but commented out that can be activated to switch to quadratic probing.

8. Modify the hashCode method for Name so that it implements the algorithm discussed in section 21.8:

$$h = u_0 g^{n-1} + u_1 g^{n-2} + \ldots + u_{n-2} g + u_{n-1}$$

where the value of g is 31. You can use the following Java code to implement the formula in an efficient manner (you should prove to yourself that the code below gives the same h as the formula above).

```
int hash = 0;
int n = s.length();
for (int i = 0; i < n; i++)
      hash = g * hash + s.charAt(i);
```

The *i*th character of the string is s.charAt(i). Adding this character to the product g * hash actually adds the character's Unicode value. An explicit cast of s.charAt(i) to int is not necessary and would not affect the result.

This computation can cause an overflow, particularly for long strings. Java ignores these overflows and, for an appropriate choice of g, the result will be a reasonable hash code. Current implementations of the method hashCode in Java's class String use this computation with 31 as the value of g. Realize, however, that the overflows can produce a negative result. You can deal with that when you compress the hash code into an appropriate index for the hash table.

In order to provide unique hashCodes for people with the same last (or first) names, you should use letters from both the first and the last names to build the hash code.

9. Then write your own driver, LoadFactorTest. This program will compare the performance of linear probing and quadratic probing. It will read a list of 1000 names from the file phonebook1k.txt and insert these into the table. Create a hashed dictionary of size 2000, and use the hash code described above. Count the number of collisions that occur while inserting, for both linear probing and quadratic probing, when 1000 names are added to the table. The average number of collisions will be the total number counted divided by 1000. Enter this information into the spreadsheet LoadFactorTest.xls.

Note that a collision occurs whenever a cell being examined in the table already contains an entry. You will also need to disable calls to "rehash" since that will increase the table size. Repeat the experiment for tables of size 1900, 1800, 1700, 1600, 1500, 1400, 1300, and 1200. The load factor lambda conveys how full the table is. (lambda = [number of items]/[table size]). In other words, lambda ranges from 1000/2000 (or 0.5) to 1000/1200 (or 0.833)). Plot your results in excel (load factor on x-axis, average collisions on y-axis) and compare to fig 22-4 in the Chapter 22 lecture pdf.