CURTIN UNIVERSITY (CRICOS number: 00301J)

Department of Computing, Faculty of Engineering and Science

Data Structures and Algorithms (COMP1002)

PRACTICAL 8 - HASHING

AIMS

- To implement a hash table
- To make the above hash table automatically resize.
- To save to and reload from file.

BEFORE THE PRACTICAL:

- Read this practical sheet fully before starting.

ACTIVITY 1: HASH TABLE IMPLEMENTATION

You are going to write a hash table with a simple hash function. Create a new class called DSAHashTable, and a companion class called DSAHashEntry (see the lecture notes). Assume the keys are strings and the values are Objects.

Following are a few notes on the implementation details of the hash table.

- hashArray stores the key, value and state (used, free, or previously-used) of every hash entry. We *must* store both key and value since we need to check hashArray to tell if there is a collision and we should keep probing until we find the right key.
- put(), hasKey() and get() all must take the passed-in key and call hash() to convert the key into an integer. This integer is then used as the index into hashArray.
- Java: If you use a private inner class for DSAHashEntry, put(DSAHashEntry) will need to be private, otherwise it will be public.
- There are many, many hash functions in existence, but all hash functions must be repeatable (*i.e.,* the same key will always give the same index). A good hash function is fast and will distribute keys evenly inside hashArray. Of course, the latter depends on the distribution of the keys as well, so it's not easy to say what a good hash function will be without knowing the keys! So for this prac, you just use a one of the hash functions from the lecture notes.
- Use linear probing or double-hashing to handle collisions when inserting. Use linear probing first since it is easier to think about, then convert to double-hashing.
- Note that hasKey(), get() and remove() will also need to use the same approach since they also need to find the right item – it's probably a good idea to try to make a private find() method that does the probing for these three functions and returns the index to use. Use the DSAHashEntry state to tell you when to stop probing.
- Be aware that remove with probing methods adds the problem that it can break probing unless additional measures are taken.

- In particular, say we added Key1, then Key2 which collides with Key1, so we linearly probe and add Key2 to the next entry. But if we remove Key1, later attempts to get Key2 will fail because Key2 maps to where Key1 used to be. Since it is now null, probing will abort and imply that Key2 doesn't exist.
- The solution is to use the 'state' field in DSAHashEntry that tracks whether the entry has been used before or not (again, see the Lecture notes)

Testing: Write a test harness to test your classes with simple (numerical) data.

ACTIVITY 2: HASH RESIZE

There are various ways to determine when to, and how to, re-size a hash table.

The simplest way to determine **when** is to set an upper and lower threshold value for the load factor. When the number of elements is outside of this, the **put()** or **remove()** method should call **reSize(size)** automatically. Remember, this will be computationally expensive (what is it in Big-O?), so it is important not to set the threshold too low. Also, collisions occur more frequently at higher load factors, thus it is equally important not to set the threshold to high. Do some research to find "good" values.

One simple way to resize is to create the new array, then iterate over the list (ignoring unused and previously used slots), re-hashing (using put()). To select a suitable size for the new array, you can either use a "look-up" list of suitable primes (web search for this), or recalculate a new prime after doubling/halving the previous size.

Testing: Test this with very low hash table size, just so you know it will work when you increase the size of the table.

ACTIVITY 3: FILES

To truly test your implementation, you will need a large dataset. Use the random names csv file as input to insert values into your hash table. There are some duplicates in the file so your program should be able to handle them.

It is also useful to be able to save the hash table. The save order is not important, so just go through the keys and values in the order they are stored in the hash table.

SUBMISSION DELIVERABLE:

Your classes (all that are required for this program) are due before the beginning of your next tutorial. Also include any other relevant classes and input files that you have used.

We will need at least a driver program to test your implementation. Test Harnesses are even better.

SUBMIT ELECTRONICALLY VIA BLACKBOARD, under the *Assessments* section.

MARKING GUIDE

Your submission will be marked as follows:

- [2] You have submitted UML diagrams for all classes and code.
- [2] Your DSAHashTable and DSAHashEntry are implemented properly and you can show they work.
- [2] Your hash function is well thought out and properly implemented. This means that it meets at least the first three criteria of a good hash function and you can argue that it at least partially meets the last.
- [2] Your hash table resizes as it grows and shrinks.
- [2] You can read in and save csv files.