CURTIN UNIVERSITY (CRICOS number: 00301J)
Faculty of Engineering and Science
Department of Computing
Data Structures and Algorithms

# Practical 5

## 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: Write Your Own Hash Table

You are going to write a hash table with a simple hash function. Create a new Java class called DSAHashTable, and a companion class called DSAHashEntry (see the lecture notes). Assume the keys are strings and the values are Objects. DSAHashTable should have *at minimum* the following methods:

```
public class DSAHashTable {
   private DSAHashEntry[] m_hashTable;    // Define DSAHashEntry somewhere!

   public DSAHashTable(int maxSize) { ... }

   public void put(String key, Object value) { ... }
   public void put(DSAHashEntry) { ... }
   public Object get(String key) { ... }
   public Object remove(String key) { ... }
   public boolean containsKey(String key) { ... }
   private void reSize(size) { ... }

   private int hash(String key) { ... }

}
```

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

- m_hashTable 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 m_hashTable to tell if there is a collision and we should keep probing until we find the right key.
- put(), containsKey() 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 m_hashTable.
- There are many, many hash functions in existence, but all hash functions must be repeatable (ie: the same key will always give the same index). A good hash function is fast and will distribute keys evenly inside m_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 containsKey(), get() and remove() will also need to use the same 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 a 'state' field in DSAHashEntry that tracks whether the entry has been used before or not (again, see the Lecture notes)

**Testing**: Use the data from your assignment to test each method.

## Activity 2: DSAHashTable Re-Size

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 (`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**: Use the data from RandomNames7000.csv. Read small parts at a time, and put print statements suitable to see when reSize() is called.

## Activity 3: File I/O

To write to and read from a file, you should serialize your `DSAHashEntry` objects. Points to consider:

- Writing: will Java's serialization write the entire list for you, or will you have to iterate over the list?
- Reading: how will you know the size of the table to create?

## Submission Deliverable:

Your `DSAHashTable` class is due at the beginning of your next tutorial.

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

If you finish early, use the rest of the practical to start the next worksheet, because that will be due later on.