# Hash Tables

Updated: 7th December, 2021

## Aims

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

## Before the Practical

- Read this practical sheet fully before starting.

## Activities

### 1. UML Diagrams

Following the pseudocode from the lecture slides as a guide, draw up the UML diagrams for `DSAHashTable`, `DSAHashEntry` and their test harnesses.

### 2. Hash Table Implementation

Following the lecture slides as a guide, Create `DSAHashTable` class and a companion class called `DSAHashEntry` to implement a hash table with a simple hash function. Use linear probing first since it's easier to think about, then convert to double-hashing. Assume the *keys are strings* and the *values are Objects*. Write a test harness to test each method thoroughly, be sure to test all cases.

**Note:**

- `hashArray` stores the key, value and state (used, free, or previously-used) of every `hashEntry`.
  We *must* store both the 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()` must take the passed-in key and call `hash()` to convert the key into an integer. This integer is then used as the index for `hashArray`.
- Java Students: If you use a private inner class for `DSAHashEntry`, then `put(DSAHashEntry` will need to be private, otherwise it will be public.
- There are 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`.

**Note:**

- 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.
  For the purpose of this practical, just use one of the hash functions from the lecture notes.
- Use *linear probing* or *double-hashing* to handle collisions when inserting.
- `hasKey()`, `get()` and `remove()` will need to use the same approach since they also need to find the right item.
  It's probably a good idea to try 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.
    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* filed in `DSAHashEntry` that tracks whether the entry has been used before or not.

## 3. Resizing a Hash Table

Modify your `DSAHashTable` to allow it to resize. There are various ways to determine when to and how to resize 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()` methods should call `resize(size)` automatically.

- Remember, this will be computationally expensive (*what is it 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 to not set the threshold too high. Do some research to find "good" values.

A simple way to resize is to create a new array, then iterate over `hashArray` (ignoring unused and previously used slots) and re-hashing (using `put()`.

- To select a suitable size for the new array, you can either use a "look up" table of suitable primes or re-calculate a new prime after doubling/halving the previous size.

Test your resize functionality with a small hash table size, just so you know it will work when you increase the size of the table.

## 4. File I/O

To truly test your hash table implementation, you will need a large dataset. Read in the `RandomNames7000.csv` file from previous practicals 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 iterate through the keys and values in the order they are stored in the hash table and write it to a .csv.

## Submission Deliverable

- Your code and UML diagrams are <u>due 2 weeks from your current tutorial session</u>.
  - You will demonstrate your work to your tutors during that session
  - If you have completed the practical earlier, you can demonstrate your work during the next session
- You must **submit** your code and any test data that you have been using **electronically via Blackboard** under the *Assessments* section before your demonstration.
  - Java students, please do not submit the *.class files

## Marking Guide

Your submission will be marked as follows:

- [2] Your UML diagram for all implemented classes and methods.
- [2] Your DSAHashTable and DSAHashEntry are implemented correctly - your test harness will show this.
- [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 you put and remove hash entries.
- [2] You can read in and save .csv files.

**End of Worksheet**