CSC2001F 2016 Practical: Hash Tables

1. Introduction

This assignment concerns the implementation and evaluation of linear probing, quadratic probing, and separate chaining hash tables. It is in the context of a specific application of the technology: the development of an electronic dictionary (a collection of words and their meanings).

You should add a sub directory titled 'DS2_Prac1' to your CSC2001F directory tree.

1.1 Scenario

A simple electronic dictionary is required that should store words and their (possibly multiple) meanings.

Sample I/O for the intended electronic dictionary program:

```
Enter a word (or '#quit'):
bore
No entry for this word.
Enter a word (or '#quit'):
bald
[(adjective) lacking hair]
Enter a word (or '#quit'):
beat
[(noun) pulse, pulsation, heartbeat, (verb) vanquish, come out
better in a competition or race, (verb) hit repeatedly, (verb)
pound, thump, move rhythmically, (verb) be superior to]
Enter a word (or '#quit'):
quit
[(verb) drop out, give up, throw in the towel]
Enter a word (or '#quit'):
#quit
```

A text file, called '`lexicon.txt`', has been provided. This file contains the word definitions that will be used to populate the dictionary. The format (in ad-hoc BNF) is as follows:

<lexicion> ::= {<entry>}
<entry> ::= <word type> ':' <word> ':' [<description>]
<word type> ::= ' a'|'v'|'n'
<word> ::= {<letter>}$^{+}$
<description> ::= {<character>}

The lexicon contains 0 or more entries. An entry consists of word type followed by a colon, followed by the word, followed by a colon, optionally followed by a description. The word type is represented by a single character; either 'a', 'v', or 'n'. A word consists of a sequence of one or more letters. A description consists of 1 or more characters (generally, it's a word phrase).

CONTINUED

1.2 Assignment objective

The dictionary is to be implemented using a hash table. The options are to use (i) linear probing, (ii) quadratic probing, and (iii) sequential chaining. The key question is, 'in this context, what are the relative performance characteristics of each type of table?'.

Your assignment is to develop an implementation for each and then investigate the question. In section 2 we describe the framework within which you will be working. In section 3 we outline performance testing requirements, and in section 4 and 5 we provide precise lists of tasks.

NOTE: You are not required to implement the electronic dictionary program described above. The description simply sets the scene. This assignment concerns the 'under-the-hood' design.

2. Dictionary Framework

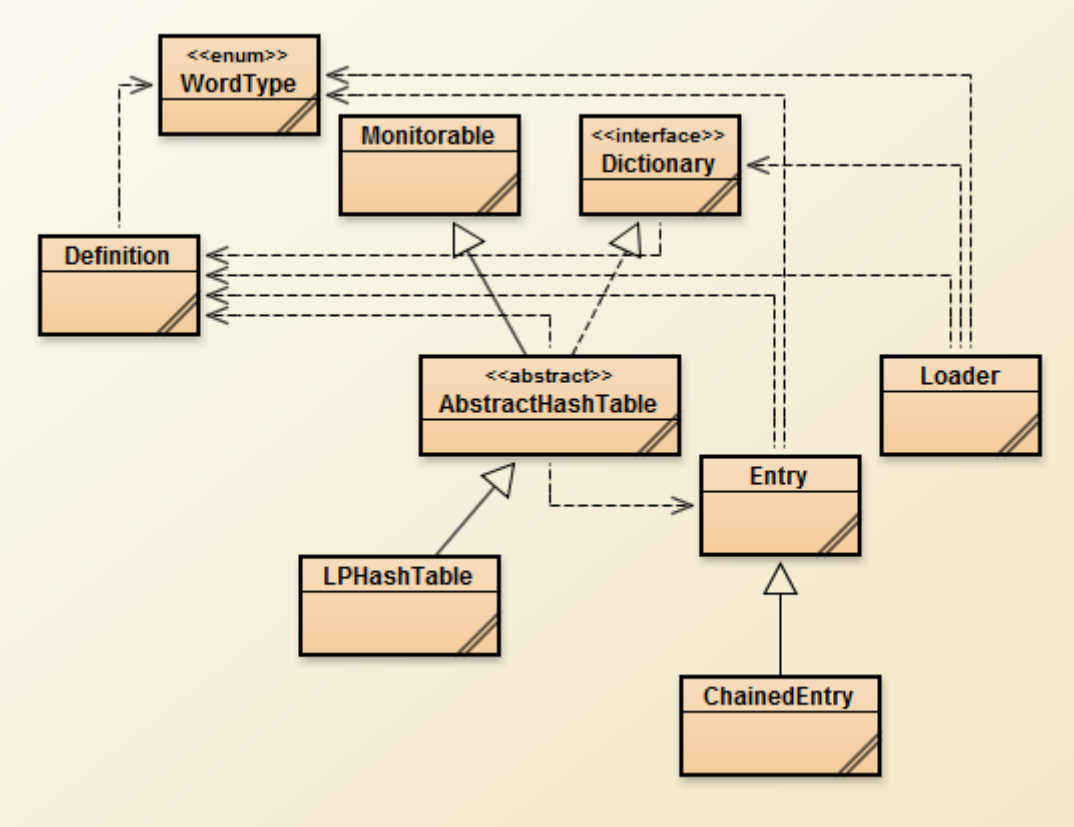
On the Vula assignment page you will find the file '`framework.zip`'. It contains a '`src`' directory, a '`test`' directory, and a '`data`' directory.

- The `src` directory contains a Java package called '`dictionary`'.
- The `test` directory contains components that may assist with performance testing.
- The `data` directory contains the `lexicon.txt` file.

You are expected to construct your own `makefile` for this assignment.

2.1 The dictionary package

The package contains the following (incomplete) framework:



CONTINUED

- Dictionary is an abstract data type that describes the functionality at the heart of the design: a dictionary contains words and their definitions.
- A Definition type of object represents a word definition. It consists of a word type and word description.
- WordType is an enumerated type definition. Values are NOUN, ADJECTIVE, and VERB.
- AbstractHashTable is an (incomplete) base class for hash-table based implementations of Dictionary.
- An AbstractHashTable contains Entry objects. An Entry aggregates a word and its definitions.
- An LPHashTable is an incomplete implementation of a Linear Probing (LP) hash table.
- Loader is a utility module containing a method for loading a dictionary with word definitions contained in a named text file.

The implementation of AbstractHashTable and LPHashTable must be completed, and then further hash table implementations added: a Quadratic Probing (QP) hash table and a Sequential Chaining (SC) hash table.

2.2 Design principles

A Dictionary provides the methods '`containsWord()`', '`getDefinitions()`', and '`insert()`'. The principle of the AbstractHashTable design is that each of these methods can be implemented using a general purpose search method called '`findIndex()`':

```
/**
 * Find the table index for the given word.
 * If the word is in the table, then the method returns its index.
 * if it is not in the table then the method returns the index of
 * the first free slot.
 *
 * Returns -1 if a slot is not found (such as when the table
 * is full under LP).
 */
protected abstract int findIndex(String word);
```

An implementation of `findIndex()will typically` depend on the '`hashFunction()`' method:

```
/**
 * Generate a hash code for the given key.
 * (Used by sub classes to implement findIndex().)
 */
protected int hashFunction(String key) { //...
```

To implement an LP hash table or a QP hash table, a sub class of AbstractHashTable simply needs to provide a suitable implementation of `findIndex()` – along with a pair of constructors.

Implementing an SC hash table, requires that the sub class override the '`containsWord()`', '`getDefinitions()`', and '`insert()`' methods as well.

CONTINUED

3. Performance Testing

To investigate the performance characteristics of the different types of hash table, you will conduct a series of experiments involving loading and searching for word definitions.

There are two aspects to performance: time and space. How long does it take to load/search? How much memory is used to store the data?

The investigation will focus on time. However, rather than obtaining actual timings for operations, you will instead count the number of probes that are performed. (In the case of sequential chaining, this means counting each examination of a link in the chain as a probe.)

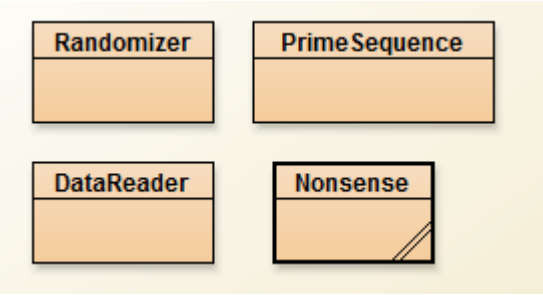
You will obtain answers to the following:

- Given each hash table implementation, how many probes are performed in the course of loading the contents of `lexicion.txt`?
- Given each hash table implementation, what is the average number of total probes performed when searching for the definitions for a random sample of 100 words?

Measurements will be taken for three different load factors: 0.5, 0.75, and 1.

3.1 Test bed

You will write performance test programs that enable the load and search performance of each type of hash table to be measured. The `test` directory contains classes that will assist you in this endeavour.



- The DataReader class provides a means of reading a text file into a list of strings.
- The Randomizer class provides a means of obtaining the elements of a list in a random order. (Alternatively you may wish to use the '`java.utils.Collections.shuffle()`' method.)
- The PrimeSequence class provides a means of obtaining prime numbers in ascending order.
- The Nonsense class provides a means of generating nonsense words.

CONTINUED

4. Hash Table Implementation Tasks [50 marks]

A corresponding JUnit test class is required for each type of hash table class.

4.1 Linear Probing [25 marks]

Complete the AbstractHashTable and LPHashTable classes.

- Implement the `hashFunction()` in the AbstractHashTable class using the algorithm presented by Weiss in "Data Structures and Problem Solving Using Java".
- Complete the implementation of the LPHashTable class using the linear probing technique.

NOTES (for developing and debugging):

- Strictly speaking, you're not developing a pure implementation of a hash table. The Dictionary ADT requires that the insertion of a word definition involves updating an existing Entry should one exist.
- The AbstractHashTable class provides a method for dumping the contents of the table.
- You can create small test files from `lexicon.txt`.

4.2 Quadratic Probing [10 marks]

Create a new sub class of AbstractHashTable that uses quadratic probing and is called 'QPHashTable'.

- Quadratic probing should be implemented efficiently.
- As for the LPHashTable class, you should implement two constructors: one that creates a table of the default size, and the other that accepts the initial table size as a parameter.
- Assume that probing has failed when the number of probes exceeds the table size ($i>M$).
- When probing fails, the `findIndex`() method should return '-1'.

NOTES (for developing and debugging):

- Keep table size prime, and keep the load factor less than or equal to 0.5, while experimenting with core functionality.

4.3 Sequential Chaining [15 marks]

Create a new sub class of AbstractHashTable that uses Sequential Chaining and is called 'SCHashTable'.

- You should use the ChainedEntry data type and override the AbstractHashTable class declaration of table i.e. the SCHashTable class should contain the following:

```
protected ChainedEntry[] table;
```

- The class will override the '`containsWord()`', '`getDefinitions()`', and '`insert()`' methods.
- You should implement two constructors: one that creates a table of the default size, and the other that accepts the initial table size as a parameter.

CONTINUED

5. Performance Testing Tasks [50 Marks]

5.1 Instrumentation

To conduct performance testing, your hash table implementations need to be instrumented – you need to add code that counts probes.

To this end, the AbstractHashTable class is defined as a sub class of the Monitorable class, and the latter provides a variable and methods for maintaining a probe count. Your code simply needs to call the appropriate method at the appropriate point(s).

5.2 Load Test [6 Marks]

Write a program called LoadTest that may be used to perform load performance testing on a hash table implementation of Dictionary.

The program will be executed with the following command:

```
java –cp bin LoadTest <class name> <table size> <file name>
```

e.g.

```
java –cp bin LoadTest dictionary.LPHashTable 100 lexicon.txt
```

The program accepts a number of command line arguments:

- The fully qualified name of the class of Dictionary to be tested.
- The initial table size.
- The name of the lexicon from which word definitions are to be read.

The program will create an instance of the class, insert the definitions in the lexicon, and then print out the total number of probes performed.

NOTE: A skeleton LoadTest may be found in the `test` directory – it contains the code needed to dynamically create a Dictionary object from command line arguments.

5.3 Search Test [14 marks]

Write a program called SearchTest that may be used to perform load performance testing on a hash table implementation of Dictionary.

The program will be executed with the following command:

```
java –cp bin SearchTest <class name> <table size> <file name> <sample size>
<number of trials>
```

e.g.

```
java –cp bin SearchTest dictionary.QPHashTable 1000 lexicon.txt 100 10000
```

The program accepts a number of command line arguments:

- The fully qualified name of the class of Dictionary to be tested.
- The initial table size.

CONTINUED

- The name of the lexicon from which word definitions are to be read.
- The size of test samples (i.e. the number of words to be sought), and the number of trials.

An instance of the dictionary class will be created, the word definitions in the data file inserted, and then a series of trials conducted.

- Each trial will involve searching for the definitions of a random sample (of the given size) of words.
- Random samples will be part computed from the data in the input file.
- 20% of each sample will consist of words not in the lexicon. (This can be achieved by making randomly generated nonsense Strings.)
- The program will sum the total number of probes over all trials and calculate and output the average for the sample size.

5.4 Performance Tests

Use the performance test programs to determine the characteristics of the three hash table implementations.

5.4.1 Load performance

Obtain the number of probes performed in the course of loading the contents of `lexicon.txt`.

- Obtain measurements for three different load factors: 0.5, 0.75, and 1.
- Ensure that, for each load factor, the table size is the next largest prime.
- Note that QPHashTable may fail under extreme load factors.

Note that load factor depends on the size of the table and the size of the lexicon.

5.4.2 Search performance

Calculate the average number of total probes performed when searching for the definitions for a random sample of 100 words.

- Obtain measurements for three different load factors: 0.5, 0.75, and 1.
- You will need a large number of trials to get some stability in your measurements. Focus on the percentage differences and don't worry about decimal places.

5.5 Report [30 marks]

Write up a short report on the performance testing of the three hash table implementations.

Your report should contain a results section, an analysis section, and an evaluation section.

5.5.1 Results section

The results section should present (in tabular form) the sets of results obtained for tasks 5.4.1 and 5.4.2. i.e. the load performance results for a load factor of 0.5, the search performance results for a load factor of 0.5, and so on.

It should also explain how the results were obtained i.e describe the LoadTest and SearchTest inputs used, and explain why they were selected.

Your tests should be plausible and reproducible.

CONTINUED

5.5.2 Analysis

In the analysis section, for each set of results, you should present (in tabular form) the percentage difference between quadratic probing and linear probing, sequential chaining and liner probing, and sequential chaining and quadratic probing.

(Take the percentage difference between *a* and *b* to be *((a-b)/a)*100*.)

5.5.3 Evaluation

Write up a short evaluation of the performance of the three hash table implementations based on your results and analysis.

Consider these sorts of questions:

- If space is not an issue, which produces the fastest insert/search functions, and under what conditions?
- Which technique generally performs best?
- Allowing that the investigation has not considered the amounts of memory used, using estimates, which offers the best combination of speed and efficient storage?

6. Submission and Marking

Following CSC2001F practical work requirements (see the document on Vula), you must:

- Conduct JUnit testing that achieves a given level of coverage (according to JaCoCo).
- Ensure that your makefile functions correctly.
- Maintain your GIT repository: conducting regular commits with comments that describe the changes.

For this assignment you need a minimum of 90% statement coverage and 80% branch coverage for the AbstractHashTable, LPHashTable, QPHashTable, SCHashTable, LoadTest and SearchTest classes. You should prepend your GIT commit comments with the label “P3”.

Your work will be part automatically marked and part manually marked by the tutors.

Submit a zip file of your entire CSC2001F directory to the automatic marker.

6.1 Mark breakdown

The automatic marker will conduct tests of your AbstractHashTable, LPHashTable, QPHashTable and SCHashTable classes.

Automarker ‘Question’	*Target*	*Mark allocation*	
1	AbstractHashTable, LPHashTable	25	
2	QPHashTable	10	
3	SCHashTable	15	
		Total	*50*

CONTINUED

The second part of the assignment will be manually assessed by the tutors.

Task	*Target*	*Mark allocation*	
5.1, 5.2, 5,3	Instrumentation, LoadTest and SearchTest	20	
5.4, 5.5	Report	30	
		Total	*50*

6.2 Work practice

A tutor will assess your adherence to practical work requirements. A failure will result in a ***percentage of your marks being deducted***.

Requirement	*Penalty if not met*	
File/Directory structure: source files in src, test source files in test, compiled files in bin, javadocs in doc, and coverage report in coveragereport.	5%	
Makefile: fully implements the targets 'all', 'test', 'report', 'doc', 'clean'.	5% per target not met (or faulty) to a maximum of 20%. *20%*	
Repository: regular commits, meaningful comments (prepended with label "P3").	5%	
Unit testing: achieves 90% line coverage and 80% branch coverage, and tests are valid i.e. actually check that the outcome is correct.	5% **per coverage target per class** and 5% per invalid test to a maximum of 30%). *30%*	
	Capped Maximum	*30%*

END