Version: 07/04/2025 07:55

CSC2001F Assignment DS4

Hashing

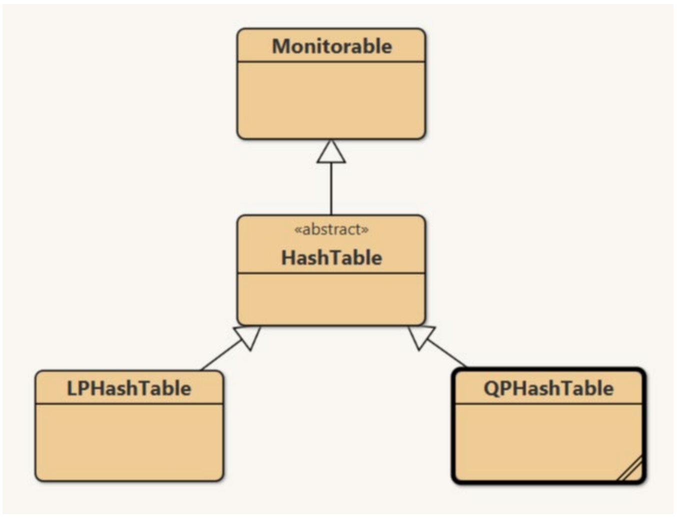
Overview

This assignment concerns (i) the implementation and evaluation of linear probing, and quadratic probing, and (ii) the optimizing of a hash function for a specific data set.

You will use a number of hash table classes given to you. The hash tables that these construct store simple String data items, so can only track if an item is in the data structure or not.

Framework

The classes that you will be using are available on the Amathuba assignment page.



- HashTable is an **incomplete** implementation of a hash table.
- A QPHashTable is an **incomplete** implementation of a Quadratic Probing (QP) hash table.
- An LPHashTable is an **incomplete** implementation of a Linear Probing (LP) hash table.

There are some **additional classes**: AutoTest, LPAutoTest, QPAutoTest, and DataMaker. AutoTest, LPAutoTest and QPAutoTest may be used **to test your work** for the first part of the assignment. DataMaker will be used **to generate personalised test data** for the second part (as explained below).

Also on the assignment page is a data file '`students.txt`' (containing the usernames of everyone on the course).

Design principles

The HashTable class has an array for storing String data items, a hash function, a method for inserting a String in the table, and a method for determining whether the table contains a given String.

The principle of HashTable is that both `insert()` and `contains()` can be implemented using a general purpose '`findIndex()`' function.

Continued

```
/**
 * Find the table index (element number) 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 free slot is not found (such as when the table
 * is full).
 */
protected abstract int findIndex(String word);
```

An implementation of `findIndex()will` depend on the ‘`hashFunction()`’ method:

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

To implement a Linear Probing hash table or a Quadratic Probing hash table, a subclass of HashTable simply needs to provide a suitable implementation of `findIndex()` – and a pair of constructors.

Performance evaluation

One way to evaluate the performance of a hash table implementation is to count the number of probes performed when inserting and searching. This requires that the hash table implementations be ‘instrumented’ – the `findIndex()` function needs code that counts probes.

Thus the HashTable class is defined as a subclass of the Monitorable class. Monitorable provides a variable and methods for maintaining a probe count. The `findIndex()` code simply needs to call the appropriate method at the appropriate point(s).

The methods of Monitorable are:

```
/**

 * Obtain the number of probes of the hash table structure that have
 * occurred since start-up or the last resetProbeCount().
 */
public int getProbeCount()

/**
 * Increment the probe count.
 */
public void incProbeCount()

/**
 * Reset the probe counter to zero.
 */
public void resetProbeCount()
```

Continued

Task One [50 marks]

Complete the implementations of the LPHashTable and QPHashTable classes.

- Complete the implementation of the LPHashTable class using the linear probing technique.
- Complete the implementation of the QPHashTable class using the quadratic probing technique.
    - 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'.

Submission

Submit your completed classes to the automatic marker.

Note

The automatic marker will use programs called LPAutoTest and QPAutoTest, each of which is a subclass of AutoTest. You may use them to evaluate your work if you wish.

The programs are identical except for the type of hash table created. They accept a number of commands: INSERT (insert an item), ISEMPTY (is the table empty), SIZE (how many items in the table), CONTAINS (is item in the table?), DUMP (print the table contents), PROBECOUNT (how many probes have been performed), RESETCOUNT (set probe count to zero), LOAD (insert items into table from file), QUIT.

The first user input sets the size of the hashtable.

Sample I/O (LPAutoTest) with **user input shown in bold** (what's not in bold is the output):

```
11
INSERT BLYBOK010
SIZE
1
DUMP

   0 : null
   1 : null
   2 : BLYBOK010
   3 : null
   4 : null
   5 : null
   6 : null
   7 : null
   8 : null
   9 : null
  10 : null
#Entries: 1.
INSERT BRHTHA016
INSERT BTHAMO046
DUMP

   0 : null
   1 : null
   2 : BLYBOK010
   3 : BRHTHA016
   4 : null
   5 : null
   6 : null
```

Continued

```
   7 : null
   8 : null
   9 : BTHAMO046
  10 : null
#Entries: 3.
QUIT
```

Sample I/O (LPAutoTest) with **user input shown in bold** (what's not in bold is the output):

```
11
load students.txt 9
dump

   0 : mhljoh019
   1 : mtmluv001
   2 : rmkyas002
   3 : gwlhlo001
   4 : null
   5 : vzjjoh002
   6 : nmktha004
   7 : null
   8 : tmbtla001
   9 : dlmsip065
  10 : msslei003
#Entries: 9.
probecount
11
quit
```

Sample I/O (QPAutoTest) with **user input shown in bold** (what's not in bold is the output):

```
11
LOAD students.txt 7
dump

   0 : null
   1 : mtmluv001
   2 : rmkyas002
   3 : gwlhlo001
   4 : null
   5 : null
   6 : nmktha004
   7 : null
   8 : tmbtla001
   9 : dlmsip065
  10 : msslei003
#Entries: 7.
probecount
8
QUIT
```

Sample I/O (QPAutoTest) with **user input shown in bold** (what's not in bold is the output):

```
11
load students.txt 10
Insert failure. Table full?
dump
```

Continued

```
    0 : mhljoh019
    1 : mtmluv001
    2 : rmkyas002
    3 : gwlhlo001
    4 : null
    5 : vzjjoh002
    6 : nmktha004
    7 : null
    8 : tmbtla001
    9 : dlmsip065
   10 : msslei003
#Entries: 9.
quit
```

(Be aware that the `AutoTest` program is not tolerant of incorrect user input since it was designed for the automarker and is offered as a courtesy.)

Task Two [50 marks]

The hash function contained in the HashTable class uses an array of 9 weights (**with values 0..4**) to create a linear combination of the 9 characters in a student ID:

$hashFunction(s) = weights$[0]*s.charAt(0)+weights[1]*s.charAt(1)+ … +weights[8]*s.charAt(8)

The default weights are {1, 2, 3, 4, 1, 2, 3, 4, 1}, however, the values can be changed using the `setWeights()` method e.g.,

```
final HashTable hashTable = new LPHashTable(10);
final int[] newWeights = {0, 1, 2, 3, 4, 3, 2, 1, 0};
hashTable.setWeights(newWeights);
```

The code creates an LPHashTable and sets the hash weights to {0, 1, 2, 3, 4, 3, 2, 1, 0} in that order.

This task concerns **computing weights that optimise insertion** of a specific list of names in an LPHashTable. Insertion is optimised if it uses the least number of probes. There may be more than one combination of weights that achieve this.

1. On the assignment page you will find a program called '`DataMaker.java`'. You will use this to generate your personal custom list, L, of 36 usernames from '`students.txt`'.
2. You will then write a program called '`Optimize.java`' that, for every possible combination of weights, evaluates the number of probes required for insertion of L usernames in an LPHashTable of size 37. It will output a single line containing two integers: (a) the least number of probes required, and (b) the number of weight combinations that achieve this. Thus if the output is
   44 4
   then the least number of probes to insert all the usernames was 44, and there were 4 different combinations (sets of 9 weights) that achieved a probe count of 44.

The `DataMaker` program accepts two command line inputs: (i) YOUR username, and (ii) the number of usernames required in L:

java DataMaker <username> <list length>

It will generate list L, outputting it to the screen. You should use file redirection to pipe this to a file called '`mydata.txt`'. Your `Optimize` program will read in this file.

Continued

Submission

Submit to the automatic marker:

1. Your data file '`mydata.txt`'.
2. Your `Optimize.java` code.
3. A text file called '`results.txt`' consisting of the results of your investigation: a single line containing: the least number of probes required to insert the contents of `mydata.txt`, and the number of weight combinations. The automarker will evaluate the contents of `results.txt` with respect to `mydata.txt`. Marks will be awarded in proportion to proximity to the correct answer. `Optimize.java` will be inspected to ensure that you have fulfilled the task requirements.

End