Exercise 1 – HashSet \ CLIDS 2013

1. Goals

1. Implementing a data structure that will be learned in class (HashSet)
2. Experimenting with this data structure, getting familiar with specific concepts of hashing (chaining, load factor, etc).

2. Submission Details

- Submission deadline: **Thursday, 18/4/2012, 23:55**
- This exercise will be done alone (**not in pairs**).
- Note: In this exercise you may use classes and interfaces from the java.lang package in standard java 1.7 distribution. You may also use java.util.LinkedList. You may not use any other classes that you didn't write yourself and you should not import any other existing java classes.

3. Introduction

A Set is a collection that contains no duplicate elements. More formally, for each two elements in the set, *a* and *b*, *a.equals(b)* returns *false*. This can be useful, for example, for representing a deck of cards or a mathematical set.

Hash set is a common Set implementation that provides efficient *add()*, *contains()* and *delete()* operations. Hash sets use a special function called *hash function*, which maps any objects to a single integer, an index in the hash structure which is based on the object's *hash code* (see lecture 4 for more details). In a few weeks you will get to know the theoretical background of hash sets in the Data Structures (DaSt) course; in this exercise you will implement a HashSet class that can stores Strings.

4. API

You are required to submit a class called *MyHashSet* that implements the following API. You are allowed to write and submit more class(es) as you see fit.

4.1 Package

Both the *MyHashSet* class and any other class that you implement in this exercise should be placed in the *clids.ex1.data_structures* package. That is, you should create a package called *clids*. In this package there should be a sub-package called *ex1*, and inside it another sub-package (*data_structures*). Your file(s) should be found in the last sub-package (*data_structures*).

4.2 Methods

```
/**
 * Add a new element with value newValue into the table.
 * @param newValue new value to add to the table.
 * @return false iff newValue already exists in the table.
 */
public boolean add(String newValue);
```

```
/**
 * Look for an input value in the table.
 * @param searchVal value to search for.
 * @return true iff searchVal is found in the table.
 */
 public boolean contains(String searchVal);

/**
 * Remove the input element form the table.
 * @param toDelete value to delete.
 * @return true iff toDelete is found and deleted.
 */
 public boolean delete(String toDelete);

/**
 * @return the number of elements in the table.
 */
public int size()

/**
 * @return the capacity of the table.
 */
 public int capacity();
```

4.3 Constructors

```
/**
 * A default constructor.
 * Constructs a new, empty table with default initial capacity (16)
 * and load factor (0.75) and lower load factor (0.25).
 */
public MyHashSet();


/**
 * Constructs a new, empty table with the specified initial capacity
 * and the specified load factors.
 * @param initialCapacity - the initial capacity of the hash table.
 * @param upperLoadFactor - the upper load factor of the hash table.
 * @param lowerLoadFactor - the lower load factor of the hash table.
 */
public MyHashSet(int initialCapacity, float upperLoadFactor,
                 float lowerLoadFactor);
```

```
/**
 * Data constructor - builds the hash set by adding the elements
 * into the input array one-by-one.
 * If the same value appears twice (or more) in the list, it is
 * ignored.
 * The new table has the default values of initial capacity
 * (16), upper load factor (0.75), and lower load factor (0.25).
 * @param data Values to add to the set.
 */
public MyHashSet(String[] data);
```

5. Implementation notes

Hashing strategy

Each object in *Java* has a default hash code. In order to map a String element into a bucket in the hash set, use the default hash code of the instance and use the *modulo (%)* operation in order to convert it to a valid bucket index (as described in lecture 4).

A "hash collision" describes the situation where two or more elements are mapped to the same bucket. You are required to handle such cases using the chaining strategy. Thus, a single bucket may store multiple entries, and must be searched sequentially. For this purpose you may use a LinkedList object (which will be used to store several values in the same bucket). You may either use the LinkedList class supplied by Java (in the java.util package) or implement a linked list of your own.

Performance considerations and re-hashing

The performance of a hash table instance is affected by three parameters: *initial capacity*, *upper load factor* and *lower load factor*. The *capacity* is the number of cells (aka buckets) in the table. The *initial capacity* is the *capacity* when the hash is created. *Load factor* is defined as $\frac{M}{capacity}$ where *M* is the preferred number of entries which can be inserted before an increment or a decrement in the capacity is required. Thus, the *upper* \ *lower load factor* measure how full \ empty the table is allowed to get before its capacity is automatically increased \ decreased, respectively. More specifically, when the number of elements in the table exceeds the product of the *upper load factor* and the current *capacity*, the capacity is increased, and when the number of elements in the table is less than the product of the *lower load factor* and the current *capacity*, the capacity is decreased. After changing the capacity (i.e. after adding\deleting an element), re-hashing (mapping all of the entries to new buckets in the new hash set) is required.

6. Analyzing the HashSet

Hash-based data structures offer constant time performance for the basic operations (add, delete, contains and size), assuming the hash function disperses the elements properly among the buckets.

a. Analyze the time complexity of iterating over the elements in a HashSet in terms of capacity and number of elements. What is the complexity of the different operations (add, contains, remove) when the size is larger than the capacity? What is the complexity when the capacity is much larger than the size? Describe how the complexity depends on the capacity and on the size in the general case.

b. Compare the performance of three different data structures. You are provided with two data files, data1.txt and data2.txt (supplied in the attached *ex1_files.zip*). Create three different data structure instances: java.util.LinkedList (a standard linked list), java.util.TreeSet (a binary search tree)[1] and MyHashSet.
The file data1.txt contains a list of 99,999 words such that after adding all of them into a MyHashSet object, they are all mapped to the same cell. The file data2.txt contains a mixture of different 99,999 words (that should be mapped ~uniformly across a hash table after adding all of the words.

In this part you are required to use the data file and the different data structures and report the time that is required for performing several operations as described below:

1. Add all the words in the data files to each of the created data structures. You should perform this separately for data1.txt and data2.txt.
    i. Compare the times required for this operation for **the different data structures** (for each file separately). I.e., compute the required times for adding all of the file content for a LinkedList, a TreeSet and a MyHashSet (use the *add()* method of each implementation in order to add all of the elements).
    ii. Compare the times required for the operation **between the two data files**, for each data structure separately (i.e., the time required for adding data1's content into a MyHashSet instance vs. adding data2's content). Do you expect to see a significant difference in all of the data structures? Explain your answer.
2. Perform the *contains* operation for the data structures that are initialized by data1's content. Apply the *contains* operation, when looking for the string "-13170890158" and when looking for the word "hi". Note that whereas the first word will be mapped to the same index as the rest of the data, "hi" will have a different position. Compare between the times required for each contains operation:
    i. For the same word compare **the different data structure** separately (i.e., for the word "hi", compare between LinkedList, TreeSet and HashSet, and do this also for the word "-13170890158").
    ii. For each data structure separately compare the times **between the two words** (i.e., compare the time of looking for "hi" and for "-13170890158" in a LinkedList, in a TreeSet and in a MyHashSet instances).
3. Perform the *contains* operation for the data structures that are initialized by data2's content. Apply the *contains* operation, when looking for the word "23" and when looking for the word "hi". Whereas "hi" does not exist in the data, "23" exists. For each data structure separately compare **between the two words** (for example, compare the time of looking for "hi" and for "23" in a LinkedList, etc).

You should submit the java files that you used for the analysis. These files should be in the package *clids.ex1.analysis.* The file that contains main method for running the analysis for this part should be called *MyAnalysis.java.* Note that some of the operations may take a few minutes.
The answers to these questions should appear in the README file (see below). In each answer specify which results you expect to get (and why) and whether the actual results match your expectations.

[1] TreeSet is actually a red-black tree, which is a special kind of binary tree (more details about it will come later in the DaSt course). For the purposes of this exercise, assume this is a regular binary search tree.

Notes:

a. In the file MyAnalysis.java (or supplementary files that you used for the analysis) you will have to import the used data structures. In this case you will have to import java.util.LinkedList, java.util.TreeSet and clids.ex1.data_structures.MyHashSet (this should be imported this as well because it resides in a different package).
b. In order to fetch the time required for performing the operation, use the *Date.getTime()* method. The command *new Date().getTime()* creates a new Date objects and returns the time in milliseconds that matches this object. You can use it twice (before and after the operation that you wish to time) and report the difference.
   Usage example:

```
long timeBefore = new Date().getTime();

// … some operations that we wish to time

long timeAfter = new Date().getTime();

// the number that you should report and use in the analysis:
long difference = timeAfter – timeBefore;
```

c. In order to have the content of the data files as Java objects, you are supplied with a method that reads a content of a text file and returns a String[] with that content.

The method is *Ex1Utils.file2array(fileName)* (the file *Ex1Utils.java* is supplied in *ex1_files.zip*)
You should locate the *Ex1Utils.java* in the same package of your analysis code (*clids.ex1.analysis*)

Usage example:

```
String[]  data1array = Ex1Utils.file2array("data1.txt")
```

Note that you should use the full path of the file that you send to the method, unless it resides in the same directory with the file that runs this code.

7. Submission Requirements

7.1 README

In each exercise in this course you are required to hand in a text file called "README". **It is important that the file name will be "README", and not, for example, "readme" or "README.txt".**
This text file is a crucial part of each exercise since it contains all the relevant information about your design and implementation decisions. Submitting an exercise without a well-written README file may result in a serious loss of points, since the graders will rely heavily on it in order to understand your implementation. Specific guidelines for writing the README file are provided in the Submission Regulations guidelines.

Specifically, please address the following points on your README file:

1. In case you built additional classes (other than *MyHashSet*) in order to implement the Hash Set, describe them. The description should include the purpose of each class, its main methods and its interaction with the *MyHashSet* class.

2. Describe your implementation of the methods *add()* and *delete()*. The description should include the general workflow in each of these methods. You should also indicate which helper function(s) did you implement for each of them, and which of these helper functions are shared by these two methods (if any).
3. Your README file should also answer the questions from Section 6.

7.2 Automatic Testing

Similarly to the intro2cs/p course, we will use automatic tests in order to evaluate the quality of your program. Our tests will do the following:

1. Make sure your code compiles and contains a README file. This includes verifying that your *MyHashSet* class follows the API defined in Section 4 (including the correct package names).
2. Test each of your methods on simple inputs, in order to test their basic functionality.
3. Test your hash on interesting scenarios (collision of values, re-hashing).
4. Test your code on complicated settings that include large inputs and multiple add/delete operations.

Each test is defined by an input file, which creates a hash set and performs a set of operations on it. The format of the input files is described in Appendix A.

Important note: when you submit the exercise via the course web site, a report of results of running the tests on you exercises is generated and will be sent to your email. When many students submit at the same time a delay of the report mail is expected. I**nstead of re-submitting the exercise every time, you can run an external file that generates the same results.** Running this script **BEFORE** submitting the exercise via the website (and waiting for an email each time) is a much better alternative.
In order to do that, simply run on the command line the following command (*ex1.jar* is your jar file):

~clids/bin/ex1AutoTests.sh ex1.jar

7.3 Submission Guidelines

You should submit a file named ex1.jar containing all the .java files of your program, as well as the README file and the files used for the analysis of section 6. Please note the following:

- Files should be submitted along with their original packages.
  - You should submit the files from *clids.ex1.data_structures* as well as from *clids.ex1.analysis*
- No *.class* files should be submitted.
- Your program must compile without any errors or warnings.
- javadoc should accept your program and produce documentation.

You may use the following unix command to create the jar files:

*jar –cvf ex1.jar README clids/ex1/data_structures/*java clids/ex1/analysis/*java*

This command should be run from the main project directory (that is, where the *clids* directory is found).

8. Misc.

8.1 Grading

- Working according to Coding style guidelines – 10%
- Analyzing the hash set (see section 6) – 20%
- Questions 1,2 in the README file section (see Section 7.1) – 10%
- Software design and implementation details – 10%
- Correct implementation – 50%

8.2 Forums and Questions

An ex1 discussion forum is available in the course website. You may use it to post questions regarding the exercise, as well as to hold relevant discussions. We will review it every 2-3 days and answer some of the questions.

In order to make the best of the forum, we advise you, before posting questions, to carefully review both the exercise definition and previously asked questions. The search option in the forum may come in handy for querying previously asked questions. Asking questions that were already asked by other students will result in (a) slower response time on our behalf (due to irrelevant questions) and (b) difficulties on the students' behalf to extract the relevant issues from the forum.

In addition to the discussion forum, an ex1 official announcement forum is also available on the course website. We will use it to post important updates and clarifications. Messages in this forum are obligatory.

GOOD LUCK!

Appendices

A. Test Input File Format

We will run automatic tests on your code. In this appendix, we describe the format of these tests. You may find this appendix useful, so that during the debug process, you will be able to understand what the problems in your code are (if any). Please note that you are **not required to write any code** in this section. It is all implemented by the course staff.

Each of the automatic tests is incorporated in a file of the following format:

- The first line starts with a '#' sign, and is a comment describing the test (i.e., this line is ignored).

- The second line creates an MyHashSet object (denoted myHash) using one of the constructors defined in Section 4.3:
  - An empty line calls the default constructor.
  - A line starts with "*N:*" (N for "numbers") followed by three integers (e.g., *N: 20 0.8 0.1*) calls the constructor that gets an initial capacity, an upper load factor and a lower load factor respectively.
  - A line starts with "A:" (A for "array") followed by a sequence of words (e.g., *A: Welcome to ex1*) calls the data constructor that initializes a new instance with the given data.

- The next lines call the methods of the created set. The lines are called one after the other. They can be one of the following:
  - *add some_string return_value*
    call *myHash.add(some_string)*. Verify that the method's return value matches *return_value* (either *true* or *false*).

  - *delete somes_string return_value*
    call myHash.delete(some_string). Verify that the method's return value matches *return_value* (either *true* or *false*).

  - *contains some_string return_value*
    call *myHash.contains(some_string)*. Verify that the method's return value matches *return_value* (either *true* or *false*).

  - *size return_value*
    call *myHash.size().* Verify that the method's return value matches *return_value* (a non-negative integer).

  - *capacity return_value*
    call *myHash.capacity().* Verify that the method's return value matches *return_value* (a non-negative integer).

A.1 Examples

A.1.1

Consider the following example test file:

```
# a simple creation of a set and adding the word Hi

add Hi true
size 1
```

In this example, the first line is a comment describing the test. The second line is empty, which means calling the empty constructor. The third line calls *myHash.add("Hi")*, and then verifies that the command succeeded (by verifying that the return value of the *add* method is *true*). The last line verifies that the tree size is now 1 (after adding a single element).

A.1.2

Consider another example:

```
# calling the constructor with the value 20, adding "Hi", adding "Hello", deleting "Hi", and then searching for "Bye".
N: 20 0.8 0.1
add Hi true
add Hello true
delete Hi true
contains Bye false
```

The first line is a comment describing the test. The second line starts with "N:" which means calling the constructor that gets the initial capacity, upper load factor and lower load factor parameter (20, 0.8 and 0.1 respectively in this case). The third line calls *myHash.add("Hi"),* and then verifies that the command succeeded. The forth line calls *myHash.add("Hello")*, and then verifies that the command succeeded. The fifth line calls *myHash.delete("Hi")*, and then verifies that the command succeeded. The last line calls *myHash.contains("Bye")* and verifies that "Bye" is not found in the tree (return value should be *false*).

A.1.3

Consider a 3rd example:

```
# calling the data constructor with the values {"DaSt", "CLIDS", "Infi"}, and checking the size and the capacity.
A: DaSt CLIDS Infi
add Infi false
capacity 16
size 3
```

The first line is, again, a comment describing the test. The second line starts with "A:" which means calling the data constructor that gets the initial data that the set contains (*DaSt CLIDS Infi*). The third line validates that an insertion trial of *Infi* has failed (return value is *false).* Finally, the last two lines check that the created set has a capacity of 16 and a total of 3 elements.

A.2 Some Clarifications

Please note the following:

- Your code passes a given test only if your methods return the correct values in each of the method calls.
- A test file may include numerous calls to *add, delete, contains, size* and *capacity*. However, only a single hash set is constructed in every test.
- The test files themselves, are supplied by the course staff.

A.3 Technical Issues

A file called *ex1_files.zip* is found in the course website. In contains the test files, on which your program will be tested. If any problem was discovered during a test (i.e., your program returned a different value than expected in one of the method calls, or an exception was thrown and not caught), an error message will be printed to the screen (with the matching test information, for the test that failed). Otherwise, if you passed successfully all of the tests, the message *"Perfect!"* will be printed and that there are no errors.

As in the intro2cs/p courses, after you submit your exercise, we will run your code against all the input files, and you will receive an email with the tests you failed on (if any). Error messages will contain some information about the problem. For example, you may receive the following error:

runTests[0](Ex1Tester): /cs/course/2012/clids/auto_tests/ex1/tests/009.txt (# simple delete of a non-existing value): line number 3: expected:<false> but was:<true>

This means that your program failed the test in the file 009.txt, when trying to execute the delete operation. More specifically, line 3 of the test file tests the delete operation, and the test failed at this stage because the return value of calling delete() was true instead of false.

For your convenience, you may **(and are strongly advised to!)** run the automatic tests from the terminal by running the following command:

~clids/bin/ex1AutoTests.sh ex1.jar

(where *ex1.jar* is your jar file)