CS2112—Fall 2012

Homework 3

Data Structures and Web Filtering

Due: September 28, 2012 PM

Implementing spam blacklists and web filters requires matching candidate domain names and URLs very rapidly against a long list of blacklisted domains. In this assignment you will be implementing core data structures and algorithms for such a filter. The first part focuses on implementing a generic hash table, a prefix tree, and a Bloom filter. The second part requires you to create an application that can determine whether an input string matches a *blacklist* of known bad strings. Finally, there is a written problem for you to turn in.

0 Instructions

0.1 Grading

Solutions will be graded on both correctness and style. A correct program compiles without errors or warnings, and behaves according the the requirements given here. A program with good style is clear, concise, and easy to read.

For this assignment, we are especially looking for good documentation of the interfaces implemented by your data structures. Write comments that crisply explain what all the methods do at a level of abstraction that enables a client to use your data structure effectively, while leaving out unnecessary details.

0.2 Partners

You *must* work alone for this assignment. But remember that the course staff is happy to help with problems you run into. Use Piazza for questions, attend office hours, or set up meetings with any course staff member for help. Read all Piazza posts, because often your questions have already been asked by someone else, even before you think to ask them.

0.3 Restrictions

Your use of `java.util` will be restricted again for this assignment. *Classes* from `java.util` cannot be used anywhere in your code except in the JUnit test harness (but `Scanner` may be used anywhere). *Interfaces* from `java.util` can be used anywhere in your code to guide your internal data structures.

While we require that you respect any interfaces we release, you should note that you are allowed to (and even expected to) come up with your own classes for portions of the assignment. An example from HW2 of a simple class that is nice to have is `FactorResult`.

1 Hash Tables

1.1 Overview

For an overview of how hash tables work, you should refer to the lecture notes.

You may use `hashCode()`, Java's default hash function. Your hash table should implement the latest `java.util.Map` interface which uses generics.

For the Set that must be returned by the `keyset()` method, you are not required to implement the full Set interface. We will require only that it implements `isEmpty()`, `size()`, `contains()`, and `toArray()` returning an Object array. Other methods may throw an UnsupportedOperationException.

1.1.1 Collisions

When different keys map to the same bucket, it is known as a *collision*.

Chaining involves putting keys and elements into a linked list at the bucket. The linked list is then searched for the key. We recommend you use chaining to handle collisions.

You may want to consider keeping track of the load factor, resizing your table whenever it crosses a threshold. A smart choice of load factor will keep memory usage reasonable while avoiding collisions.

1.2 Implementation Requirements

Hashing aside, the operations `containsKey`, `put`, `get`, and `remove` should have expected $O(1)$ runtime. Your hash table should use $O(n)$ space, where n is the number of elements being held.

2 Prefix Trees

2.1 Overview

A prefix tree, also known as a *trie*, is a data structure tailored for storing and retrieving strings. The root node represents the empty string. Each possible next letter branches to a different child node. For each node where a string terminates, that node may contain either the value of the string or a flag indicating the string termination. In the latter representation, every string in the data structure is determined by the path along the trie. Figure 1 shows an example of a trie. Your trie should implement the `Trie` interface that we provide.

2.2 Implementation Requirements

Your implementation should support `insert`, `delete`, and `contains` in $O(k)$ time, where k is the length of the string.

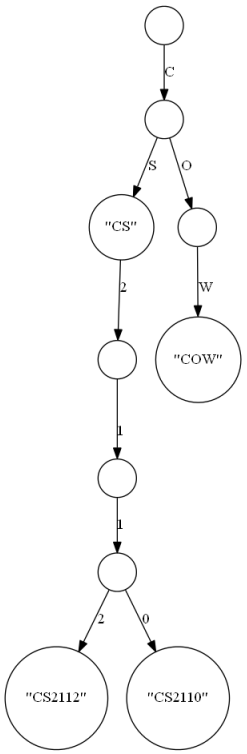

Figure 1: A trie containing the strings `CS`, `COW`, `CS2110`, and `CS2112`, where string terminations are represented by the value of the strings at the corresponding nodes.

3 Bloom Filters

A Bloom filter is a probabilistic constant-space data structure for testing whether an element is in a set. It is probabilistic in the sense that a false positive may be returned (i.e., an element is determined to be in the set when it is not) but false negatives cannot occur. A Bloom filter with nothing in it is a bit array of 0s.

To insert an element into a Bloom filter, put the element through k different hash functions. Use the integer results of those hash functions as indices into the bit array. Set those k bits in the bit array to 1.

To simulate k different hash functions for String objects, you may append single characters to the end of your Strings before hashing (e.g. `a`, `b`, `c`, ...). To determine if an element is not in the Bloom filter, check if all of its hash indices. If any of them is zero, the element must not be in the Bloom filter.

3.1 An Example of a False Positive

Suppose you have a Bloom filter for String objects represented by a bit array of length 2, initially empty, using only one hash function. To insert `CS2112`, we compute its hash value (suppose it hashes to 0), and we set that bit to 1 in the bit array. Now, to check whether `CS2110` is in the Bloom filter, we compute its hash value (suppose it also hashes to 0). We see that the bit at position

0 is set to 1, and so we conclude the Bloom filter may contain the String `CS2110`.

3.2 Implementation Requirements

The size of the bit array backing your Bloom filter and the number of hash functions you use affects the probability of false positive results.

Your Bloom filter should implement the `BloomFilter` interface that we provide.

4 Web Filter

Your web filter will take in URL(s) as input parameters and determine whether they match a blacklist of bad URLs. Your web filter should implement the `WebFilter` interface that we provide. You should assume that the input source for blacklists is a newline-separated file containing millions of URLs.

Console interface

For Homework 3, we have provided a sample interface but you are welcome to change it as you see fit:

- `clearFilter`: empty the web filter blacklist
- `addBlacklist <blacklist_file>`: add the URLs from the file specified to the web filter.
- `filter <input_urls> <filtered_urls>`: read URLs from the `input_urls` file. For each URL that is in the web filter, add it to a newline-separated output file specified by `filtered_urls`.
- `perf <input_urls> <n>`: read up to `n` URLs from the `input_urls` file. Determine whether each URL is in the filter and report how many passed the filter, along with the total time taken in milliseconds. If `n` is larger than the number of URLs in the input, the existing URLs are reused repeatedly until `n` total URLs have been tested.

5 Performance

Performance analysis is a component of the grade for this assignment. You should choose data structure(s) wisely to be efficient in both memory usage and performance. Justify your design in `README.txt`. We are looking for quantified comparisons of performance when you use different data structures to back the web filter. Feel free to make use of `System.currentTimeMillis()` for timing and VisualVM for memory profiling. Correctness and performance are both important when we evaluate how well the web filter works.

6 Testing

In addition to the code you write for data structures and the web filter, you should also submit any tests that you write. Testing is a component of the grade for this assignment.

You may write your own custom test harness, but you are encouraged to use JUnit, an automated test suite with excellent Eclipse integration. We have included a small example to demonstrate how to write JUnit tests.

We will not directly test any internal data structures that you implement but it is highly recommended that you write JUnit test harnesses for them.

7 Submission

You should compress exactly these files into a zip file that you will then submit on CMS:

- *Source code*: Because this assignment is more open than the last, you should include all source code required to compile and run the project.
- *Tests*: You should include code for all your test cases.
- `README.txt`: This file should contain your name, your NetID, all known issues you have with your submitted code, and the names of anyone you have discussed the homework with. In addition, you should briefly describe and justify your design, noting any interesting design decisions you encountered.
- `Solution.txt`: Include the solution to the written problem.
- `perf.txt` or `perf.pdf`: This file should include your analysis of performance.

Do not include any files ending in `.class`. We expect you to stick to Java 6 features and avoid features found only in Java 7.

All `.java` files should compile and conform to the prototypes we gave you. We write our own classes that use your classes' public methods to test your code. *Even if you do not use a method we require, you should still implement it for our use.* You may add your own additional methods.

Par for this assignment is about 700 lines of new code.

8 Written Problem

The standard Java interface `SortedSet` describes a set whose elements have an ordering. Abstractly, the set keeps its elements in sorted order. Here is a simplified version of the interface:

```
/** A set of unique elements kept sorted in ascending order. */
interface SortedSet<T extends Comparable<T>> {
   /** Add x to the set. */ % should have said: if it is not already there.
   void add(T x);
   /** Tests whether x is in the set. */
   boolean contains(T x);
   /** Remove element x. */
   void remove(T x);
   /** Return the first element in the set. */
   T first();
}
```

8.1 Part 1

The specification of `remove` has at least one serious problem. Clearly identify a problem and write a better specification. You may change the signature if you justify it. (**Note:** We are not considering a failure to produce nice javadoc a serious problem here.)

8.2 Part 2

You are given a compiled implementation of the `SortedSet` interface but don't have the code. Design four black-box test cases for the `remove` method that give reasonable coverage. A very short explanation for why each test case adds coverage is reasonable. Let us assume that testing on `SortedSet<Integer>` is adequate. You may use the notation `[1,2,3]` to represent a sorted set containing the elements 1, 2, and 3. For example, one test case might written as:

```
On [3,5,6,10] : remove(6) // a typical case
```

Now give four more that improve coverage.

8.3 Part 3

Suppose we want a different implementation `UnsortedList` that is just like `SortedList` except that it has no data structure invariant. But it should still implement the `SortedSet` interface. Implement the `add` and `first` methods as concisely as you can. (Hint: It should be easier to implement `add` since there is no invariant to maintain.)