CS 2112—Spring 2014

Assignment 3

Data Structures and Web Filtering

Due: March 4, 2014 11:59 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 Javadoc-compliant 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 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. Other external Java libraries will generally not be allowed, either.

While we require that you respect any interfaces we release, you should note that you are allowed to (and even expected to) introduce your own classes to solve portions of the assignment.

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 called 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. That is, run time should be proportional to the string length.

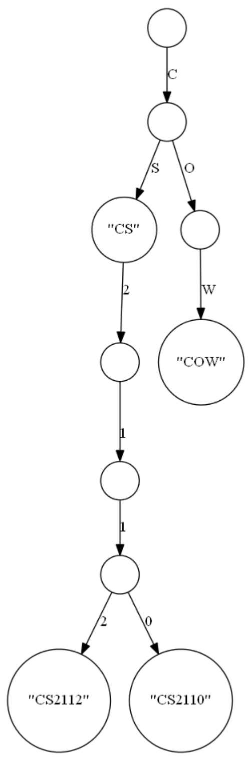


Figure 1: A trie containing the strings `CS`, `COW`, `CS2110`, and `CS2112`, where string terminations are represented by the value of the strings at the corresponding nodes.

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. $\texttt{a}, \texttt{b}, \texttt{c}, \ldots$). To determine if an element is not in the Bloom filter, check 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 should implement your test cases using JUnit, a framework for writing test suites. JUnit has excellent Eclipse integration that makes it easy to use. We have included a small example to demonstrate how to write JUnit tests, and we have scheduled a lab about using JUnit.

You should not only test whether the program works correctly from the command line interface, but also should write test cases for each of the the data structures you implement. There are several good strategies for writing test cases.

In black-box functional testing, the tester defines input–output pairs in which the inputs provide good coverage of the input space. Each input is accompanied by the expected result as defined by the specification. We expect you to define traditional functional test cases for your program as a whole and for each data structure you implement.

A second approach to testing is random testing, in which the inputs are generated randomly, though in a way that satisfies the preconditions. A random test case might generate single randomly chosen method calls or a sequence of randomly chosen method calls against an object of the tested class. This form of testing can catch bugs simply when the code fails with an exception or assertion error. An often effective way to randomly test functional correctness is to test whether the behavior of the code matches that of a simple *reference implementation* on which the same operations are performed. For example, you could use the `java.util` libraries to build simple reference implementations for the abstractions.

We expect you to use random testing on at least one abstraction you develop in this assignment. JUnit has some support for random testing in its Theories module, but use of this feature of JUnit is optional.

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. All source code should be located in the src/ directory.
- *Tests*: You should include code for all your test cases, in a clearly marked directory separate from the rest of your source code.
- `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, and briefly discuss your testing strategy.
- `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`.

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.

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 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

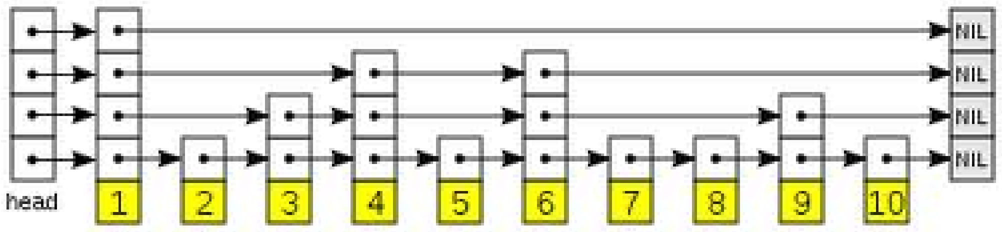
Suppose we want a different implementation `UnsortedList` that is just like `SortedList` except that it has no data structure invariant. But it should still implement the `SortedSet` interface. Implement the `add` and `first` methods as concisely as you can. (Hint: It should be easier to implement `add`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

9 Karma

Karma questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

9.1 Skip Lists

A skip list is an interesting data structure which supports quickly searching through a collection of ordered elements. It is represented as a hierarchy of linked lists in which each linked list holds a subset of the elements of its predecessors. Fast lookup is achieved by using the linked lists on higher levels as express lanes which bypass many of the elements in lower lists thus decreasing the time it takes to find a value in the sorted list at the lowest level. One challenge in designing the skip list is determining which linked lists should contain a given value. We will use a randomized approach presented in the section on Inserting below. An example of a skip list is displayed below and it's operations are addressed in further detail. A great resource for really understanding skip lists can be found here: `http://www.youtube.com/watch?v=kBwUoWpeH_Q` Watching this video before diving into an implementation will make your job substantially easier!



9.1.1 Initialization

When a skip list is created, 1 or more linked lists are created. We do not need to worry about the exact number of initial linked lists as this number will grow as our skip list increases in size (described in the Inserting section). The first element in each of these linked lists should hold a value that is smaller than any element that will be stored in the skip list. Since we are writing generic code and do not know the type of the elements in our skip list, it is best to have the user of the skip list provide an element that is smaller than any other element that could be placed in the skip list. This will ensure that all element lookups will only move down and to the right.

9.1.2 Lookup

The search for an element x begins at the head of the highest list in the collection of linked lists (upper left in the diagram above). This list is traversed to the right until there are no further elements in the list or until the value at the next location is greater than x. At this point a pointer is followed from the current node to it's equivalent in the list one level down where the process continues. Looking up the value 6 in the list above would yield the traversal $1_3 \to 1_2 \to 4_2 \to 6_2$ where x_y represents the node containing value x on level y. We say that the lowest list is on level 0. This traversal takes 3 steps instead of the 5 steps that would be required if only the linked list on level 0 was traversed.

9.1.3 Inserting

To insert an element x into the skip list, find the pair of consecutive nodes n_1, n_2 on level 0 such that the value at n_1 is less than or equal to x and the value at n_2 is greater than or equal to x. Create a new node containing x and insert it into the list at level 0 between n_1 and n_2. Now we must decide whether this element should also be included in the list at level 1. We now flip a coin, if heads, we will insert x in the list at level 1 otherwise we will leave it only at level 0. This process continues with x moving up to the next level with probability 0.5 until a tails is flipped. If x was just inserted in the linked list with maximum level and a heads is flipped, create a new list with level $max + 1$ that now contains only x.

9.1.4 Deleting

To delete an element x from the skip list, find a node containing x on level 0 and remove it from the linked list. Then remove all nodes immediately above the node containing x from their respective linked lists.

Karma: ● ● ● ● ○

9.2 Cuckoo Hashing

Cuckoo hashing is an interesting alternative to chaining when implementing a hash table. Under Cuckoo hashing, two (or more) hash functions are used instead of one and collisions are resolved by switching the current hash function.

9.2.1 Collisions

Assuming a hash table T and a pair of hash functions f_1 and f_2, insertion of an element x works as follows. Compute $f_1(x)$. If $T[f_1(x)]$ is empty then x is placed at that location and the algorithm terminates. If $T[f_1(x)]$ is inhabited by some element y, then x is placed at that location and the newly displaced y is re-inserted using the hash function f_2. This process continues as $T[f_2(y)]$ may be inhabited as well. A risk with cuckoo hashing is that this process could enter a cycle such as $T[f_2(T[f_2(y)]) = y$. In this case, the table is resized and all elements in it are rehashed. Further information about Cuckoo hashing can be found in this paper (http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf).

Karma: ● ● ○ ○ ○