CSE 241 Algorithms and Data Structures **Lab 3**

Skip Lists for Ordered Data Collections

Assigned: 10/31/2012 *Due Date: 11/28/2012*

1 Overview

This lab is intended to give you some practice with the skip list data type. We'll use the skip list as a database indexing strategy that lets us make useful queries quickly. The specific application will be an interface to a good-sized database of historical events collected from several sources around the Web.

Our event database will contain objects of type `Event`. Events are pairs of the form (*year*, *description*), where *year* is an integer value denoting the date of the event (negative numbers for BC/BCE dates), and *description* is an arbitrary text string. I have provided you with code to read an event database from a file into an array of `Event` objects. Your job is to implement an `EventList`, a collection type that organizes the events to support queries.

Because many events can happen in the same year, your collection will need to deal intelligently with **duplicate keys**. The interface specs below will describe the desired behaviors in the case of duplicates.

2 Part One: Implement the Skip List (70%)

Your first order of business is to implement insertion, deletion, and search for your `EventList` type. For search, we will replace the usual `find` operation by a pair of methods more suited to our application.

2.1 Interface Specs

The Java interface specs for the required `EventList` operations are as follows:

- `void insert(Event e)` – insert the specified event into the list. If multiple events with the same year are inserted, you need to store *all* of them.
- `void remove(int year)` – remove *all* events in the list with the specified year.
- `Event [] findRange(int first, int last)` – return an array containing all events with years between `first` and `last` inclusive, in increasing order by date (with ties broken arbitrarily).
- `Event [] findMostRecent(int year)` – return an array containing all events with date x, where x is the *latest date in the list* that is $\leq$ *year*. In other words, given a date in history, what's the "latest news" as of that date? Note that if there are multiple most recent events, you must return *all* of them.

For both the `find` methods above, you should return `null` if no events are found matching the specified criteria. C++ users will have a slightly different interface: the elements of the list are `Event *`'s instead of `Event`s, and the return type of the `find` methods is a `std::vector<Event *> *`.

You must use a skip list to implement the collection. Your `find` methods should run in expected time $O(\log n + m)$, where n is the total number of events in the list and m is the number of events returned. The `insert` and `remove` methods should be expected $O(\log n)$.

For this part of the lab, you can allocate static head and tail pillars that have "large enough" height (say, 1000, which is very unlikely to be exceeded). However, you **must** keep track of the highest level used so far and use that level as the starting point for your operations. Otherwise, you'll end up doing a lot of useless work to walk down the pillars before you even get started.

I have provided a geometric random number generator `randomHeight()` as part of the `EventList` class. You should initialize the generator in your constructor as shown. Any random seed is fine – the randomness in a skip list affects running time but not correctness.

2.2 Testing Your Event List

To run the Lab3 test code, you should run

```
java Lab3 <event file> <query file>
```

The event file contains one or more lines, each of which contains one event. Each line has the form

```
<year> <description>
```

where the year is an integer, and the description is an arbitrary string. The query file contains one or more lines, each of which contains one query. The following types of query lines are valid:

```
F <year1> <year2>
M <year>
D <year>
```

these query types correspond to the "findRange", "findMostRecent", and "remove" operations, respectively.

I have posted the event and query files that the autograder uses along with the lab document, in case you want to play with them locally yourself.

What to Turn In

To complete this part, you must ensure that your SVN repository contains all of the following:

- Your code, in particular your implementation of the `EventList` class.
- A correctly edited *control file* that specifies the implementation language you used and indicates that the lab is ready for grading. The file `lab3/control.txt` is the control file for this lab; see that file and the Labs page on the course website for editing instructions.

- A brief document describing how you implemented support for multiple events with the same key. This is important! As usual, you should also describe anything interesting you noticed and any problems remaining with the code.

  Put this document in the `lab3` subdirectory of your repository. It should be named `README`, with a file extension matching its format. The file may be plain text (`README.txt`), Word (`README.doc` or `README.docx`), or PDF (`README.pdf`).

3 Part Two: Space Improvements (30%)

The basic skip list type can be made more space-efficient in two ways. You should implement both of the following optimizations:

- **Dynamically resize the head and tail pillars** (15%). With this optimization, your head and tail always start out at height 1. Whenever you allocate a new pillar taller than the head and tail, double the head and tail heights one or more times until the resulting pillars are at least as tall as the new nodes.

- **Singly-linked skip list** (15%). We can nearly halve the space used by the skip list by making it *singly linked* rather than doubly linked. After completing this part, your list should contain *only* `next` pointers, not `prev` pointers. For simplicity, you may keep the tail pillar around (though you can get rid of it if you like by inserting appropriate null pointer checks).

  The only procedure seriously affected by this change should be your `findMostRecent()` method. You'll need to rethink how you implement this procedure without `prev` pointers. (Hint: it's very similar to the basic `find` method.)

If your code works with these extensions, you may use it *instead of* the Part One code to satisfy all the turn-in requirements of Part 1. Document how you did the extensions in your `README`.

4 Part Three: Text Search (10%)

For extra credit, augment your `EventList` to permit searching for events based on keywords in their description, rather than by their year. I have provided an interface for a method `findByKeyword()` that takes a string containing a keyword and returns all events containing that keyword in their description. Keywords are *case-insensitive* and will be converted to lower-case before your code sees them.

The `Event` class provides a method `toKeywords()` that converts its description to an array of keyword strings; see the comment at the top of the file for more info. Note that keywords contain only lower-case alphabetic characters (all upper-case chars are converted to lower case), numbers, and dashes. Any other characters are removed from the string before keyword conversion. Hence, the string

```
Alas, poor Yorick's head!  I sort-of-knew him, Horatio!
```

is split into the following list of keywords:

```
alas poor yoricks head i sort-of-knew him horatio
```

Your keyword search *must* run in time sublinear in the number of events! For your implementation, you are free to use any Java library or C++ STL functions and classes (e.g., a hash table or Map) you like to maintain the mapping from keywords to the events that contain them. To test your extension, the Lab3 driver also accepts commands of the form

```
K <keyword>
```

If you implement this extension, **please enable the extra directive** in your control file so that the autograder tests it. Please add a brief description of your implementation to your README.