

Lab 3

March 18, 2008

Due Date: April 1 (early date: March 25)

The goals of this lab are:

- To help you gain experience in using the ADTs and data structures we have been studying to implement a basic system that allows a user to pose simple queries about a collection of historical events.
- To provide you with experience in the design process from the application to a working implementation for an application that requires multiple data structures to satisfy its needs.
- To help you become more familiar with API and use for the libraries on the CD included with the book so that you can use it to help develop applications more efficiently.

For this application, each historical event is specified by a date (just the year) and a textual description. There are typically many events in the collection that occur on any given date. In this lab there is a good amount of flexibility in your design. Part of the goal of this lab is for you to work through the design process.

For the purpose of this lab, you do not need to include any exception handling in your implementation. You are expected to have a `HistoricalEvent` class that holds each historical event. Please provide a `toString` method that shows the event as “date: description.” In addition, it is **very important** that you provide an `equals` method that defines when two historical events should be considered to be equivalent. You will also find it helpful to have the `HistoricalEvent` class implement `Comparable`, and then use the `compareTo` method to define how events are to be ordered. The other alternative, is to pass a comparator into the constructor (for the collection or tagged collection data structure that you plan to use). Finally, if your design includes a Set of historical events, you will want to define a `hashCode` method for historical events.

You must also provide an implementation for the provided `RangeIterator` interface which extends Java’s `Iterator` interface to modify the semantics of `next` and `hasNext` so it can be used to iterate over all events with a date in a given range. For `remove` you can throw an `UnsupportedOperationException` (or just have it do nothing) – it will not be used for this lab. So you just need to support a constructor, `hasNext` and `next`. (*Hint:* Your implementation here will wrap an iterator over dates, and also wrap an iterator within the bucket associated with the current date.)

You must create a class `HistoricalEventApplication` that includes the following methods. You are expected to create a design that would be efficient even when the number of events gets much larger. Points will be associated with the efficiency of your solution. You are expected to analyze the asymptotic time complexity of your solution

- `void add(HistoricalEvent e)` inserts the given event into the collection.
- `boolean contains(int date)` returns `true` when there is some event in the collection with the given date, and otherwise returns `false`.
- `Collection<HistoricalEvent> dateSearch(int date)` returns a collection of all historical events with the given date.
- `Collection<HistoricalEvent> removeDate(int date)` removes all historical events that occurred on the given date, and returns a collection of the removed events.
- `boolean removeEvent(HistoricalEvent e)` removes a historical event (if one exists) that is equivalent (i.e., has the same date and description) to the given historical event. It returns `true` if an equivalent event was found and removed, and otherwise returns `false`.
- `RangeIterator<HistoricalEvent> rangeSearch(int startDate, int endDate)` returns a range iterator that will iterate over all historical events in the collection (in chronological order) that occurred on a date d where $\text{startDate} \leq d \leq \text{endDate}$.
- `boolean contains(String word)` returns `true` when there is some event in the collection that includes the given word in the description, and otherwise returns `false`.

- `Collection<HistoricalEvent> wordSearch(String word)` returns a collection (for which the iteration order is chronological order) of all events that include the given word within the description of the event.

You are welcome to use anything provided within the Java Library or on the CD provided in our text (which is on reserve in Olin). I highly recommend you read the portion of this lab assignment that reviews the support provided on the CD in our text. I expect you to spend 2-3 hours looking over this material to gain familiarity with it. So give yourself some time to get familiar with what we have provided, and in the end I think you will spend less time overall.

To help you in debugging, there is a very simple program `InteractiveDriver.java` that queries the user with a very simple textual interface. (Please change the argument to `useDelimiter` to whatever your system uses for the end-of-line character.) You may also want to use `20Events` instead of `allEvents` when first debugging. However, you do not need to submit any output when using `20Events` unless you are using it to help demonstrate a problem with your lab. Hopefully, that won't be necessary.

Part 1: Design (10 points)

Part 1 is due by the start of class on Tuesday March 25th. You should provide a design with enough detail that we can understand how you plan to implement your lab. Analyze the asymptotic time complexity for each of the historical event application methods in terms of d the number of dates associated with some event in the collection, n the number of events in the collection, w the number of distinct words in the descriptions of the events in the collection, and any other parameters that are relevant. Just introduce variables for any other aspects of the collection relevant to doing the analysis. You can use the solution for Problem 4 of Homework 4 to give you a sense of the level of detail expected. Anyone who wants to submit Part 1 earlier may do so, and we will grade it so that you can get feedback sooner than March 27th.

Part 2: Implement the Basic Functionality (45 points)

The points will be divided up as follows: supporting structure (5 points), adding an event (10 points), date search (10 points), deleting all events with a given date (10 points), and deleting an event with a given date and description (10 points). You may submit whichever methods you have completed on March 25 for a 5% bonus.

Part 3: Implement the Range Search (20 points)

This part also includes your implementation for the `RangeIterator` interface. You will receive a 5% bonus if you submit this part by March 25.

Part 4: Implement the Word Search (25 points)

In this part you will support the `wordSearch` method and the `contains` method that takes a `String` as its argument. Both of these methods should not be case sensitive. You can use `word.toLowerCase()` to convert the `String word` to lower case. When an event is removed, your data structure used for the word search must also be updated. There will be 10 points associated for properly handling this aspect of the word search. Each event should be included only once for a given word (even if the word occurs more than once in the event description). There will be 5 points associated with making the adjustments necessary so that an event only occurs once for a given word. Here's some code that you can use to take a description (a `String`) and parse it into words (converted to lower case).

```
StringTokenizer st = new StringTokenizer(description.toLowerCase());
while (st.hasMoreTokens()){
    String word = st.nextToken();
    //now do whatever you want with word
}
```

Note that the above uses spaces as a delimiter. So as an example for the description "Peter Dirichlet proves Fermat's Last Theorem for n=14." the words would be "peter", "dirichlet", "proves", "fermat's", "last", "theorem", "for", "n=14." Although for a real application you might want to be more careful about how

punctuation is handled, this way of parsing the description into words is fine for this lab. There are about 4000 words (the exact number varies depending on if you handle punctuation). As with Part 2, you are welcome to turn in a working subset of Part 4 for a 5% bonus.

Overview of Implementation for Collections, TaggedCollections, and TaggedBucketCollections

There are two orthogonal decisions for applications that need an algorithmically positioned collection. The first is whether the application can directly organize the elements using their values (**untagged**) or whether it is necessary, or perhaps just more convenient, to add a tag to organize the elements (**tagged**). The choice between the corresponding tagged and untagged collection is determined by whether the equivalence tester (and if appropriate the comparator) is a function of field(s) in the object, or a function of externally imposed information.

When the elements are organized based on a tag associated with each element, the next decision is whether or not the elements with the same associated tag should be stored as individual tagged elements (**ungrouped**) or combined into one bucket associated with the shared tag (**grouped**). The advantage of grouping elements with a shared tag is that the search cost depends only on the number of unique tags, versus the number of elements, and grouping the elements supports efficient access to all elements with a given tag.

A **Collection** is an untagged collections, a **TaggedCollection** is a tagged ungrouped collection, and a **TaggedBucketCollection** is a tagged grouped collection. In our book, every data structure is implemented for an (untagged) collection. For example, the red-black tree implementation is designed to maintain a collection of elements where the comparator used to order the elements is defined over the elements themselves.

To implement the tagged version of a data structure, we wrap the corresponding untagged version. In particular, we define a **TaggedElement** class in which each tagged element consists of a tag, and its associated data (the element). We also define a **TaggedElementComparator** that wraps the application provided (or default) comparator defined over the tags, to define a comparator over tagged elements. Then, the data structure implementation for the corresponding Collection ADT is used where the elements inserted into the collection are tagged elements, and the comparator provided is the tagged element comparator. All methods of the **TaggedCollectionWrapper** that implements the **TaggedCollection** interface mostly delegate the work to the underlying collection data structure.

The implementation of a tagged bucket collection is in **TaggedBucketCollectionWrapper**. The **TaggedBucketCollectionWrapper** takes two arguments for its constructor: an empty tagged collection that is to be wrapped, and the class to be used for the bucket. For example, for Lab 2, the bucket mapping could have been allocated using:

```
TaggedBucketCollection<String,Integer> table =
    new TaggedBucketCollectionWrapper<String,Integer>
        (new OpenAddressingMapping<String, Collection<Integer>>(), DynamicArray.class);
```

First observe, that each tagged bucket collection is parameterized by two types, the first is the type of the tag and the second is the type of the elements in the bucket. For Lab 2, the tag was a string, and the bucket held integers. Let's now look at the second line which contains the arguments to the constructor. Observe that:

```
new OpenAddressingMapping<String, Collection<Integer>>()
```

allocates a new open addressing mapping (a tagged version of open addressing) where the tag is a string and the associated data is a collection of integers. If desired, an argument to this constructor could have been provided. So this allocates an empty tagged collection that is to be wrapped. The second argument to the **TaggedBucketCollectionWrapper** constructor indicates that a dynamic array is to be used for the bucket. Any data structure can be used for the bucket. If the bucket is large, then just as you think about what data structure you need for an application, you need to think about how the data within each bucket is going to be accessed, and from that determine which ADT, and then which data structure will be best.

In order to do this lab, I strongly recommend that you look over the following interfaces either in the book or using the javadoc provided with the implementation (just as you use the javadoc for the Java libraries).

- `Locator` Interface (Section 5.8, page 64–65)
- `Collection` Interface (Section 7.1, page 90–91)
- `TaggedCollection` Interface¹ (Section 49.2, page 789–791)
- `TaggedBucketCollection` Interface (Section 50.4, pages 829–830)

You will also want to read the interfaces for any ADTs that you choose to use. Note that a list of all interfaces can be found on page 1015 in the index. (The text and CD are in reserve in Olin.)

The `TaggedBucketCollection` interface only includes the methods that are appropriate for all collections. If the tagged collection wrapped has additional functionality (e.g., the `max` method of the ordered collection or priority queue) that is needed by an application, then you need to extend `TaggedBucketCollectionWrapper` to include any methods you want to add. I **strongly advise** you to create an `TaggedBucketOrderedCollectionWrapper` that wraps the `TaggedBucketCollectionWrapper`, and in this method include a range search method using the appropriate methods on the wrapped tagged ordered collection. There is no need, for the purposes of this lab, to provide any of the other methods that can be supported by an ordered collection. We'll spend some time in class talking about this on Thursday, and you can talk to us about this during any of our office hours.

Note that the version of the `TaggedBucketCollectionWrapper` on the CD does not support some options that you might want to use, though there are good designs that don't require you to need any of these extra features. On the course web page you can download a revised version of the `TaggedBucketCollectionWrapper` that includes a constructor with a third argument that is a comparator to pass to the bucket constructor when a bucket is created. This constructor requires that the class used for the bucket has a single-parameter constructor where the parameter is a `Comparator`. If such a constructor does not exist, it can easily be added. In addition, the updated `TaggedBucketCollectionWrapper` allows some specific `Collection` class (e.g., `DynamicArray`) to be used in the type information of Java Generics instead of just giving a type of `Collection` when creating a tagged bucket collection.

What to Submit

For this lab you are expected to submit the following. Please include these items in the given order so that the TAs can find everything easily. Remember that Part 1 is due on March 25, the other portions are not due until April 1.

- A completed and signed cover sheet with your name written legibly on the top.
- A write-up along the lines for Problem 4 of Homework 4 describing how you designed your lab. You must include the time complexity analysis for all historical event application methods. Remember this part is due after one week (on March 25). The bonus for submitting early only applies to the other parts.
- The code you wrote or modified for Lab 3. You do not need to include code used from the book that is not modified. If you used code from any source other than the Java libraries or the book, you must include this on your cover sheet (and if possible give a URL for the code you used). You may not use code given to you by anyone (whether or not they are in the class) or found on the web that has been written for this application. All code/modifications must be your own work. As always, you must indicate anyone who you discussed this lab with (whether you gave or received help).
- The output generated by `DriverForFinalOutput.java`. Please comment out any methods that you did not support. Please clearly specify which methods are supported. There is no need to submit the modified driver. Also, some of the event descriptions are long – do not worry if the way you print the output causes some of them to be cut off.

If there were any problems with your implementation (e.g., wrong output was obtained, a run-time error occurred, ...) then clearly indicate that in your write-up and give as much information as you can as to what you think is causing the problem.

¹This interface did not include a `getTagComparator` method, which you might find useful (though it is not necessary). Feel free to add this, and anything else you feel would be useful to any of the provided classes. You can implement this method (in the `TaggedCollectionWrapper`) by `return pairs.getComparator();`.