The goal of this lab is 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. 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 the details of your implementation. Part of the goal of this lab is for you to work through the design process.

Here is the specification of what you are required to support (if you submit all parts). For the purposes 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”.

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`.

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 will be asked 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 `startDate ≤ d ≤ 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 the text. Give yourself some time to get familiar with what support the CD provides, and in the end 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. 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.
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 collection, 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 is allocated using:

```java
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:

```java
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.

It is recommended 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).
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 projects web page for this lab includes an updated version of the TaggedBucketCollection-Wrapper that has two capabilities that are not supported in the version on the book’s CD.

- It includes an additional 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 be easily added.

- It replaces the occurrence of Collection\<E\> in the header by ? extends Collection\<E\> which allows you to use Java generics to specify a more specific type of Collection for the bucket.

Observe that 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 can just extend TaggedBucketCollectionWrapper to include the methods you want to add. You should include the ranch search method and range iterator within this class since it will have access to the wrapped tagged collection. If you want to access the comparator defined over the tags that is used by a TaggedCollection in goldman.collection.tagged.TaggedCollection.java add the line:

```java
public Comparator getTagComparator();
```

Then in goldman.collection.tagged.TaggedCollectionWrapper.java, add the method

```java
public Comparator getTagComparator() {return pairs.getComparator();}
```

Then for any tagged collection, you can call getTagComparator() to get the comparator that is used to compare the tags.

To define the way to compare historical events, HistoricalEvent should implement Comparable and define the compareTo method to provide the desired semantics. It is important that you define the equals method (in which two historical events are equivalent when their date and descriptions are equivalent), and the compareTo method (in which the date is used to order them) for the HistoricalEvent class. As pointed out in the Java documentation, “Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals.” This is what is being suggested here, but it will achieve what you want for this application. If you are using a Mapping where the tag is a historical event, don’t forget that you must also define a hashCode method for HistoricalEvent.

**Part 1: Design (10 points)**

Part 1 is due in one week. 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 method listed on page 1 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.
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).

Part 3: Implement the Range Search (20 points)
This part also includes your implementation for the RangeIterator interface. Both this and the range
search method should be part of the class that extends TaggedBucketCollectionWrapper. For the
inner class that supports the range iterator, think about what iterators you can wrap within it to
support the needed functionality.

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).

```java
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).

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.

- A completed and signed cover sheet with your name written legibly on the top.
- A write-up along the lines for Problem of Homework 4 describing how you designed your lab.
  You must include the time complexity analysis for all methods listed on page 1. Remember this
  part is due after one week. The bonus for submitting early only applies to the other parts.
- The code your 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 runtime 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.