COMP 215 Assignment #3

Due October 4th (note this is a Thursday) at 11:55PM

1 Description

In this assignment, you will be asked to provide two implementations for the `ISparseArray<T>` interface. Note that you already used this interface when you used the `SparseArray<T>` class in the 2^{nd} assignment.

Here are the specific requirements for the assignment:

1. The two `ISparseArray<T>` classes you implement should be called `TreeSparseArray<T>` and `LinearSparseArray<T>`, respectively. The latter class should have a constructor that takes a single integer, which is the initial number of slots to allocate (this will make more sense when you read on). The former class should have a no-argument constructor.

2. You need to implement iterators for both of these classes. The iterators will both implement the `Iterator<IIndexedData<T>>` interface. The iterator for the `TreeSparseArray<T>` class should be called `TreeIndexedIterator<T>`, and the iterator for `LinearSparseArray<T>` should be called `LinearIndexedIterator<T>`. Note that in order to do this, you will need a concrete implementation of the `IIndexedData<T>` interface. You can use the same implementation for both iterators.

3. `TreeSparseArray<T>` must be based upon the `TreeMap` generic, and it is almost trivial to implement. You might find that most/all methods are one line long. As an aside, the `SparseArray<T>` class we provided you in A2 was also based upon a `TreeMap`.

4. `LinearSparseArray<T>` is a bit more complicated. It should have the following two private member variables:

   ```
   private int [] indices;
   private Vector<T> data;
   ```

   The first is an array that should store the indices of all of the data that have been added into the `LinearSparseArray`. The second is a vector that stores all of the data objects that have been added into the `LinearSparseArray`. It is assumed that the ith item in `indices` is the index for the ith item in `data`. In addition to these two member variables, you can add whatever other member variables you want.

2 Optimizing Performance

This assignment primarily is meant to provide you with more practice building class hiearchies so that you can gain a better understanding of object-oriented design principles. One reason, among many, that such class hierarchies can be valuable is that they allow you to swap in different implementations of the same abstraction for performance optimization without modifying the code that uses the abstraction. A small part of this assignment (2 test cases) deals with exploring such performance optimizations. We encourage you not to begin optimizing performance until you have a functional implementation of the required classes. You may, however, want to keep in mind the optimizations discussed below as you implement `LinearSparseArray` so that you don't do something that will make such optimization more difficult.

At first glance, the required `LinearSparseArray` implementation may seem odd. The reason for this somewhat strange setup is that it will allow you to come up with an implementation that is very, very fast for certain access patterns—raw arrays (such as `indices`) are very fast when they store primitive types and are accessed sequentially. As such, this particular setup can be used to implement `myArray` (a `LinearSparseArray`) so that the following code runs very quickly (much faster than the `SparseArray` implementation we provided you in A2):

```
LinearSparseArray<Double> myArray;
...
double total = 0.0;
for (int i = 0; i < 100000000; i++) {
  Double temp = myArray.get (i);
  if (temp != null)
    total += temp;
}
```

At first glance, this loop may seem silly. For vectors of length n (100000000 in this example) with m elements actually contained in the sparse array, this loop would have a running time of $O(n \log m)$ with a `TreeSparseArray`. (You should convince yourself this is true.) Using the iterator that all `ISparseArray` implementations must provide, you could trivially reduce the running time of this computation to $O(m)$ by only iterating over the elements actually contained in the sparse array. (Again, you should convince yourself this is true.)

However, there are many real-world situations in which you cannot successfully use an iterator to traverse the sparse array. As we all know by now, an instance of the `SparseDoubleVector` class contains an `ISparseArray` object. It is common for a user of a `SparseDoubleVector` to scan the array from front to back, perhaps skipping a few entries here and there, calling `SparseDoubleVector.getItem` to access each element in sequence. The effect of using the `SparseDoubleVector` class in this way is that a sequence of calls to `ISparseArray.get` must be executed. Those calls will look very similar to the calls in the loop given above, motivating this as an important test case.

A naive implementation of `LinearSparseArray.get` will yield a running time of $O(nm)$ on the above loop. By getting that down to $O(n)$, you will be able to get good performance on the aforementioned real use cases.

Because we will be using this in the overall application we are building throughout the semester, you will be graded on not only the correctness, but also the performance of your `LinearSparseArray` implementation. So, a few of the test cases focus on the speed of your implementation for certain operations. One test checks to see if `LinearSparseArray<Double>` runs is less than 2/3 of the time of the tree-based `SparseArray<Double>` we gave you for A2, for a series of very well-behaved `put` and `get` operations. Another checks whether `LinearSparseArray<Double>` runs in less than 1/2 of the time for the same test.

3 Optimization Hints

The required speedups require careful optimization, but they are not pushing the limits of what is possible. For example, on my machine, my reference implementation of `LinearSparseArray<Double>` takes less than 1/5 as long as `SparseArray<Double>` for this particular test. You can do whatever you need to do in order to make your `LinearSparseArray<T>` implementation fast for these speed tests, as long as you use the `indices` and `data` member variables described above. Here are some suggestions that you might consider:

1. Always maintain `indices` and `data` so that positions stored in `indices` are always in sorted order. But don't make any calls to `sort` to do this. If someone `put`s an item with a position that is greater than you've ever seen before, just append the index and the data to the end of `indices` and `data`, respectively. This will take $O(1)$ time. It is this situation where the test cases require you to be super fast, and you will! If, however, someone adds to a position that is not at the end, then just use a slow, $O(m)$ algorithm to add the item in, sliding everything down to maintain a sorted order for `indices`. You'll be slow for out-of-order insertions, but that's not what you'll be timed on.

2. Always keep some empty space at the end of `indices` and `data` so that you can very quickly append new data. Then, when you run out of space, use a "doubling" algorithm. Just make both `indices` and `data` twice as long as they were before—now they'll have a bunch of empty space at the end. So each time they fill up, you double them. That way, the number of times you need to increase the size of both is logarithmic in the number of insertions, and you allocate more space very infrequently.

3. When someone calls `get`, the naive thing would be for you to just linearly search through `indices` to find the item they are looking for. Instead, you could remember the last slot in `indices` that you retrieved a value from, and start your search from there—if you don't find the position you are looking for, then you do an $O(m)$ search. The great thing about this is that if someone goes through the array in order, you will have a super-fast $O(1)$ retrieval time for each element.

4 Grading

Passing all of the test cases is 70% of your grade. Your design (that is, your class hierarchy) is 10% of your grade. And documentation counts for 20% of your grade. The guidelines from A1 for commenting your code apply to A3 as well.

As before, keep in mind that we will be looking over your code. If you don't meet the requirements outlined in the previous section, we reserve the right to not give credit for some of the test cases that you passed.

5 What You Need To Do

You should first download the `.jar` file, the file of test cases (`SparseArrayTester.java`), and the two interface files `IIndexedData.java` and `ISparseArray.java`. Note that every file except for the test code is identical to what you used for A2. Once you do this, create a project, name it `HW3.drjava`, add all of the interfaces to it, and add the `.jar` file (which contains the `SparseArray` class implementation) to it. When you are ready to test, you can add the test code to your project.

Just like last time, you may not modify any of the interfaces or the `sparsearray.jar` file in any way.

6 Turnin

You should add all of your source files and the `.jar` file into a DrJava project as described above. On Owlspace, you should turn in the `HW3.drjava` project file that DrJava creates when you create your project, as well as any and all `.java` source files or `.jar` files that you create or use in your project. The goal here is for us to be able to simply fire up DrJava using your `HW3.drjava` project, and everything should work without having to move files around or add files to the project. Please comment out any debugging output from your code before you turn it in.

Please upload each file directly. Do not create an archive (zip, rar, etc.).

7 Academic Honesty

Please keep the academic honesty policy in mind as you work on the assignment.