CS 202 Programming Project # 1

RSet: A Class For Representing and Manipulating Sets

Part I: Due Wed, Feb 6 by 8:59AM
Part II: Due Wed, Feb 13 by 8:59AM

Overview

For this project you implement a group of Java classes for representing and manipulating finite sets in a rather general way. As you know from CS201, an element of a set may itself be a set. Thus, you may have "sets of sets of sets..." – i.e., a recursive structure. The java classes you implement will enable this kind of representation.

The project may be a bit different from past projects you have done in that I will specify for you "templates" for each class including the public method signatures and their required semantics, but (in most cases) no implementation of the body of the method. Another type of "template" is represented by abstract methods that you will have to implement in a concrete subclass (that you must create) that extends the abstract class **SetElem**. This subclass must be called **RSet**.

One way to think of this is that we have already completed the first phase of the design process and determined how we want our classes to interface with the outside world (or clients of the classes). Now the job is to complete the implementation to meet the specifications.

Objectives

You will gain experience and practice in the following areas.

- Inheritance, Abstract Classes, and Interfaces in Java.
- Linked list implementation and manipulation
- Recursion, recursion, recursion.
- Meeting specified runtime requirements
- Mutability vs. Immutability

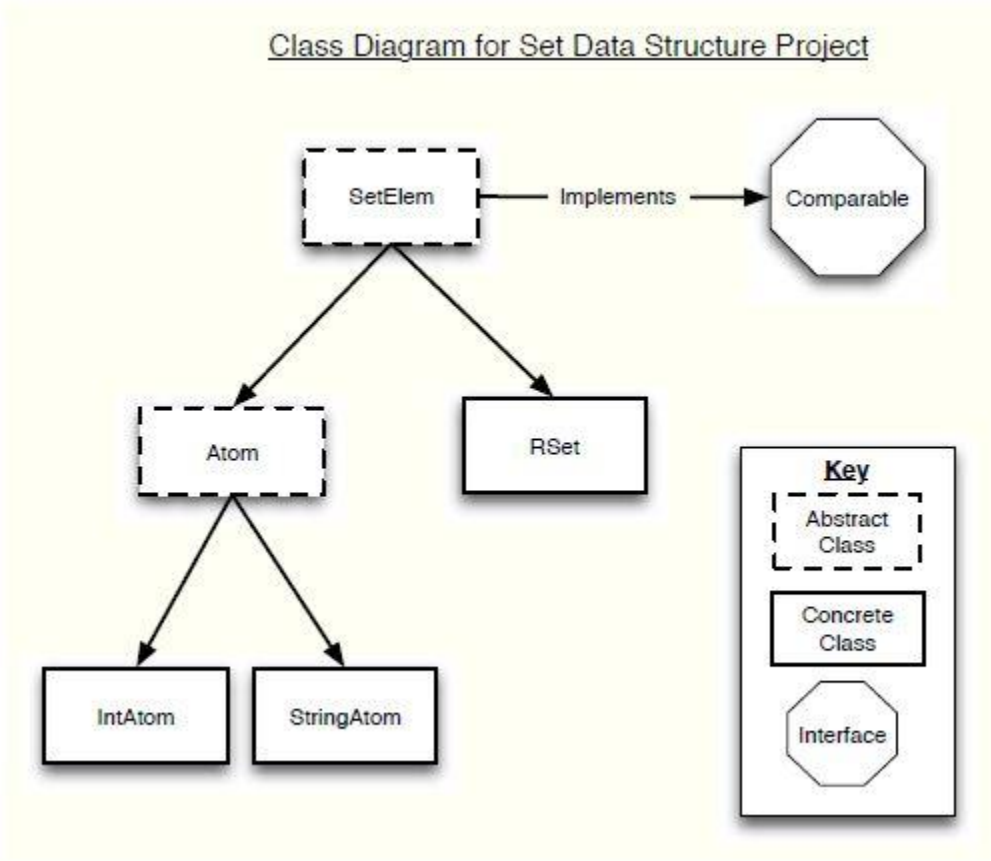
Elements of a Set

An element of a set can be either an atom or another set. An atom is either an integer or a string. For example, the set {12, "xyz", "zzz", {}, {1, 2, {3, 4}}, {{}, {"xyz"}}} has 6 elements: 1 integer, two strings and 3 sets.

We will use the term leaf for any element which itself contains no other elements – i.e., leaves are either atoms or the empty set {}.

Class Organization

In order to represent such sets, I have specified a class organization summarized in the following figure.



The only class you will modify/implement is RSet. But you will need to understand the others. You will be provided a template for the RSet class containing stubs of methods you need to implement.

Rules

- You may **not** change the signatures of any of the methods in the given files.
- You may not add any public data members, except for when implementing abstract methods in a concrete subclass.
- You may **not** add private data members.
- You may add private helper methods.
- You may add inner classes.
- You may create additional classes which your implementation uses although it shouldn't be necessary. Most non-trivial existing Java classes are disallowed for this project – i.e., I expect you to build most everything from scratch. As an example, you cannot use the LinkedList class for this project. You may ultimately thank me for this requirement as you will probably find that some natural ways to approach this project become cumbersome (maybe impossible) if you use the LinkedList class instead of your own implementation. Exceptions to the "No java classes" rule: String and all of the boxing classes (Integer, etc.).

Canonical Ordering

As you know, the order of elements in a set does not matter in a mathematical sense. However, this does not mean that enforcing a particular ordering scheme in the representation of sets is not useful as a convention. In this project, you will adopt such an ordering scheme in part to enable you to meet some of the runtime requirements and also to give some uniformity when displaying sets to the screen. The ordering on set elements you will follow is defined as follows.

- **Atom**s come before sets.
- **IntAtom**s come before StringAtoms
- The ordering of two **IntAtom**s is determined by their respective values.
- The ordering of two **StringAtom**s is determined by their alphabetical ordering.
- The empty set {} comes before all other sets.(but after all atoms).
- When comparing two non-empty sets, we apply recursive lexicographic rules. Suppose the two sets are:

  $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$ (both in canonical order)

  If $a_1 \neq b_1$then the relative ordering of A and B is the same as the relative ordering of a_1 and b_1.. If they are equal, then we move to a_2 and b_2 – precisely in the same way that we do alphabetical ordering. Also, in the same way we do alphabetical ordering, if we exhaust one of the sequences, the exhausted sequence **precedes** the other (e.g., "abc" comes before "abcd" in dictionary ordering).

These ordering rules will be encoded in your implementations of the compareTo() method (note that SetElem implements the Comparable interface).

The invariant that all sets are represented with this ordering will be key in meeting some of the runtime requirements.

Immutability

All of the classes you will implement must be **immutable**. The Wikipedia definition of immutability:

> *"In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created."*

The String class in Java is an example of an immutable class; the boxed types like Integer are also immutable.

Note for example that the union operation creates a new set – the calling object and the parameter are unchanged. You can and will exploit the fact that set elements are immutable to safely save some storage. For example, suppose two sets have an element x in common (x may be an atom or a set itself). Since we know that the object representing x is immutable, both sets can actually refer to the same object without worrying about it being modified through the other set. Thus you can avoid some deep copying.

The EMPTY_SET

If you take a look at the file RSet.java, you will find a static, private and constant data member EMPTY_SET that is pre-initialized.

This may be puzzling at first, but consider the following:

- Since the class is immutable, if a client has multiple **RSet** instances that are the empty set, then why not just let them all point to the same object?
- Notice that an **RSet** instance is really a linked list node. This is possible because the empty set is represented by this special node. Otherwise, what would the empty set be? We could use **null**, but that isn't an actual class instance. So we would then have to add another layer of complexity by defining **RSet** to be an object containing a data member giving us access to our linked list (which could then be **null**).
- This allows us to see more naturally the recursive structure of a set: if **s** is a non-empty **RSet** $\{\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}\}$ then s.next is also an RSet: $\{\mathbf{a_2}, \ldots, \mathbf{a_n}\}$. Kind of nice don't you think?
- By terminating all **RSet** instances with the EMPTY_SET node, we also reduce the number of cases we have to handle. We are using EMPTY_SET as a sentinel. This also makes sense: a singleton set {a}, then this set without its first element is the empty set {}; using the sentinel approach mimics this property: if **RSet s** is a singleton set, then s.next is **EMPTY_SET**.

Phases

You will complete the project in two phases. See the source file for the breakpoint between the phase-1 and phase-2 methods.