COS 360 Programming Languages

Prof. Briggs Third Programming Exercise

Individual Part due date: 7 October 2017 at 8:00 AM
Team Part due date: 7 October 2017 at 8:00 AM

As we have said, the set that contains all subsets of $\mathcal{N}$ that are either finite themselves or have finite complements(so are *cofinite*), is itself a countable collection of sets. One way to imagine these sets is to consider for any subset S of $\mathcal{N}$, the infinite bit string representation of S $(b_0, b_1, b_2, \ldots)$ where $b_i = 1 \leftrightarrow i \in S$. If S is finite, then $\exists\, m\, \forall\, n \geq m\ b_n = 0$, that is, eventually the string ends with the infinite sequence of 0's. If S is cofinite, then $\exists\, m\, \forall\, n \geq m\ b_n = 1$, that is, the sequence eventually ends with the infinite sequence of 1's.

It is easy to conceptualize the union($\cup$), intersect($\cap$), and absolute complement($\bar{\ }$) operations for these bit string representations as componentwise or($\vee$), componentwise and($\wedge$), and componentwise negation($\neg$) of the bits.

For this assignment, you are to implement a representation of the collection of subsets that are either finite or cofinite as a Java class, `CofinFin.java`, by completing the template file of that name in this archive.

The class has two instance specific data members,

1. `finite` of type `TreeSet<Integer>`
2. `complement` of type `boolean`

The scheme is that `finite` holds some finite subset S of $\mathcal{N}$, and `complement` indicates whether the set being represented is S or the complement of S. If `complement` is true, then the set being represented is the complement of S, that is, the represented set is cofinite. If `complement` is false, then the set being represented is S itself, that is, the represented set is finite. Mathematicians often use the notation $\bar{S}$ or S^c to indicate the unary, absolute complement of a set S, $\mathcal{N} \setminus S$. We will use the former here when we need it.

You will need to code two constructors as specified in the source template (I give you the default constructor), the two mutators, `add` and `remove`, as specified, and also implement the following operations

1. union($\cup$)

2. intersect($\cap$)
3. complement($\bar{\ }$)
4. is subset of($\subseteq$)
5. is in($\in$)
6. equals($=$); you can do this using $\subseteq$
7. maximum value
8. minimum value
9. set difference($\setminus$) I want this to be coded by individuals working alone. There is a separate class file `CF.java` for this purpose.

The details of the specification are given in the template, in particular, exceptions that should be thrown and the messages to go with the exceptions.

These operations are all instance specific, that is, they are not static. The receiver object should always refer to the first set argument, if there are two set arguments. None of these operations, except `add` and `remove`, modify the receiver. None of them modify any other argument. If the operation returns a `CofinFin` result, it should create an entirely new object with its own data members. In particular, it should not share a `TreeSet<Integer>` object with another `CofinFin` object.

Note, `TreeSet` will already have methods for most of these that you can use, but you need to adapt to the cases when one or more of the arguments is cofinite. Suppose A is a variable of type `CofinFin`. We use A_c to refer to its `complement` data member, and A_f to refer to its `finite` data member, and $\langle A \rangle$ for the set value that A represents. The key to remember is that $x \in \langle A \rangle \Leftrightarrow A_c \neq (x \in A_f)$. That is, when A_c is true, $x \in \langle A \rangle \Leftrightarrow \neg(x \in A_f)$. When A_c is false, $x \in \langle A \rangle \Leftrightarrow x \in A_f$.

We discuss the methods that return a `CofinFin` result. The complement method is trivial: just copy the TreeSet and flip the complement. The others are a little harder, but surprisingly easy, if you approach them as follows.

This first thing to do is to break down the input arguments into the distinct combinations of cofinite and finite. For two input arguments, there will be four combinations, and you determine for each specific pair whether the result will be finite or cofinite. For example, for union, if you union two of these sets, it will be cofinite if either one or both is cofinite, and finite else. Think of the bit string representations and what the boolean expression for the bits would be for that set operator. For a particular pair, will that operation eventually lead to all 0's or all 1's?

Next, for each combination, you use the set theoretic definition of the operation to write an expression of the form

$$x \in \text{conceptual result} \Leftrightarrow \text{expression for the operator and conceptual inputs}$$

Now, for a particular combination, each of the three sets, the result and the two input arguments, will be known to be cofinite or finite, so you can replace the expressions $\ldots \in \langle \text{variable} \rangle$ representing membership in the conceptual value of the variable with the correct version of $\in$ or $\notin$ and the `finite` data member as explained above. When it is finite you replace it with $\in$ and for cofinite $\notin$. Then you act on the expression to make the $\Leftrightarrow$ show what the condition should be for membership in the result variable's `finite` data member. This may require you to negate both sides of it and use the DeMorgan identities.

Finally, examine the side that has the input variables and see what set theoretic operator that expression defines on the `finite` data members of the arguments. That indicates the `TreeSet` method you will want to use on the `finite` members of the arguments to obtain the `finite` data member for the result.

We illustrate with an example. Suppose you are calculating the union of two `CofinFin` objects, A and B, into a third `CofinFin` object C(I am just naming them for reference in the discussion). You should split the code into the four cases corresponding to the four possibilities of (A_c, B_c). You need to determine the values of C_c and C_f. C_c is determinable from A_c and B_c as stated above.

You will need to consider each of the four possibilities, but we examine one, when (A_c, B_c) is $(\text{true}, \text{false})$, unioning a cofinite set $\langle A \rangle$ with a finite set $\langle B \rangle$, so the result has to be cofinite, and C_c must therefore be set to true.

The second part is to determine what values should go in C_f. If we write the logical equivalence using the definition of union and the conceptual sets it is

$$x \in \langle C \rangle \Leftrightarrow x \in \langle A \rangle \ \vee \ x \in \langle B \rangle$$

but because A, B, and C are cofinite, finite, and cofinite respectively, we can replace this with

$$x \notin C_f \Leftrightarrow x \notin A_f \ \vee \ x \in B_f$$

Since we want the condition for when $x \in C_f$ to tell us what set operation to use, we negate both sides

$$x \in C_f \Leftrightarrow \neg(x \notin A_f \ \vee \ x \in B_f)$$

which by one of the DeMorgan laws is

$$x \in C_f \Leftrightarrow x \in A_f \ \wedge \ x \notin B_f$$

The expression $x \in A_f \ \wedge \ x \notin B_f$ is the set difference operator, $A_f \setminus B_f$, so we need to find the `Set` method that realizes that.

The other combinations and the other operators all work the same way: use the set theoretic definition of the operator and the specific combination to write an expression involving the `finite` data members of the variables, manipulate it to isolate $x \in C_f$ on one side, and see what set theoretic operator on A_f and B_f is implied by the expression on the right.

Although you could get away with defining just one of union or intersect, since the other can be defined from it and the complement operation, I want you to code both of them from scratch. Similarly, $A \setminus B = A \cap \bar{B}$, but I want you to explicitly code it using the technique I described above. For what combinations will $A \setminus B$ be cofinite? finite? What is the bitwise expression you would use?

The subset method can be handled similarly. The definition is

$$\langle A \rangle \subseteq \langle B \rangle \Leftrightarrow \forall\, x(x \in \langle A \rangle \rightarrow x \in \langle B \rangle)$$

Using the four possibilities for A and B we can rewrite the right hand side. One the four can be dealt with immediately from simple cardinality considerations.

For the (true,true) pair you use $\bar{A} \subseteq \bar{B} \leftrightarrow B \subseteq A$, a fact that can be established from the definition of $\subseteq$ and the contrapositive.

For the (false,true) pair, the right hand side becomes

$$\forall\, x(x \in A_f \rightarrow x \notin B_f)$$

Its negation is

$$\exists\, x(x \in A_f \wedge x \in B_f)$$

so the value is equivalent to $A_f \cap B_f = \emptyset$. The `Set` method for subset is `containsAll`, but you will need to read its documentation to use it correctly. The name sounds like the second argument to $\subseteq$ is the receiver object of the method.

By taking advantage of the `Set` methods already present, you should not need to write many loops in your code. The constructor that takes an array will require one, and finding the minimum value in A when A_c is true will also need one. That will be the smallest value that is *not* in A_f. Everything else should be reworkable into `Set` methods. TreeSet objects have their values ordered, and unless the object is empty the `last` method will return the largest value and the `first` method will return the smallest.

The main method is written to test the other methods, using a `toString` method that we provide. There are some earlier test cases that are commented out because they do not catch any exceptions that are thrown. If your code works for the basic part, you can uncomment the older test cases and see if it works for them as well. Correct output files for both are included. If you see a discrepancy between your output and mine, and you think yours is correct, you should call it to my attention.

I want to be able to do simple file compares, so although you may add additional code while debugging, what you submit should not produce any more output than the initial template does. Of course, you need to replace the stub code of the methods with your implementation.

If you have any questions, get back to me.