Homework # 8

due Monday, April 3, 10:00 PM

In this homework assignment, you will implement an Account ADT using a binary search tree (BST) data structure. A BST permits a more efficient lookup mechanism (comparing to a linked list) so there is an efficiency test to ensure your code can handle a quarter of a million entries.

1 The Binary Search Tree Data Structure

Please read section 9.5 in the textbook for a description of the binary search tree data structure. Alternatively, there are many webpages/lecture notes on BST, the wiki page maybe a good starting point (https://en.wikipedia.org/wiki/Binary_search_tree). In the textbook (as well as some online sources), a separate BTNode class is used; we will not do that. Use a nested node class as before.

Linked lists and arrays support insertion, removal or finding an entry in time proportional to the number of entries in the container. Trees, on the other hand, offer a significant efficiency advantage in that they generally support these operations in time proportional to the log of the number of entries (assuming the tree is kept balanced).

In order to achieve the potential time efficiency of binary search trees, one needs to keep the tree roughly balanced. In CompSci 535, you will learn some of the techniques used to do this (as it happens, the tree-based containers in Java’s libraries use “red-black” trees). But in this course, we will ignore the problems of unbalanced trees. Do not make any attempt to re-balance your tree. The efficiency tests we run will make sure to construct a balanced tree.

2 Concerning the Transaction ADT

Account transactions are implemented using a public class Transaction packaged with the Account class (described below). A “transaction” consists of a date/time, an non-negative amount, and a payee.

We override the equals method to define that two transactions are the same only if they have exactly the same date/time, amount, and payee. We override the hash code computation as well; hash codes will be discussed later in the course.

2.1 How is Date/Time stored?

We will store the Date/Time of a transaction using the java.time.LocalDateTime class. This class provides the ability to compare two Date/Times which will be helpful when creating our tree. You can check if one LocalDateTime is before or after another using date1.isBefore(date2) and date1.isAfter(date2).

We also provide a Util class which gives the functionality to convert a date/time to a string of our preferred format, and back again. This helper class also defines the minimum and maximum dates we will allow in an account.

3 Concerning the Account ADT

Account transactions are most often sorted by date because this allows people to view when they spent money, what they spent it on, and create a budget around that information.

The Account ADT is a sorted collection of Transaction objects. It supports the following public operations:

- int size() Return number of transactions in the account.
- Transaction getEarliest() Return the earliest transaction in the account, or null if none.
- Transaction getNext(LocalDateTime) Return the transaction with closest Date/Time later than the argument, or null if no such transaction in the account.
boolean add(Transaction) Add a transaction, returning whether it was possible (it is not allowed to have two transactions with the exact same Date/Time).

int addAll(Transaction[], int lo, int hi) Add all transactions from the array within the index range [lo, hi) into the account, returning the number successfully added.

Transaction[] toArray(Transaction[]) Copy the transactions (in sequential Date/Time order) into an array and return it.

We omit “remove” functionality from the ADT for simplicity. As mentioned earlier, the account is not allowed to have two transactions with the same Date/Time. (In practice, one would use the payee as a secondary key, but that simply complicates the programming without you learning more about binary search trees.)

The addAll method takes extra parameters so that it can be its own recursive helper function. The reason why one wishes to use recursion is that if addAll were simply to add all the listings in the order they are listed and if the listings are already sorted (which frequently happens if one uses the interactive driver described next), one would have a highly unbalanced tree. Thus addAll is supposed to add elements starting from the middle and then to add two remaining halves recursively.

The efficiency tests check to see that you built the tree correctly. If you wrote the code efficiently, this test shouldn’t take more than 15 seconds or so. If you wrote the inefficient (but easy) technique, the test will take much longer.

4 Concerning the “Account Editor” Driver Program

We provide a text-based interactive program for interacting with Account objects. You may wish to use this program to run your own tests. It also demonstrates the bare-bones functionality a program using Account would have.

5 What you need to do

First, you will need to implement the private recursive helper function (checkInRange) that checks if the tree is well formed. Next, you need to implement all the public methods for the Account ADT using efficient binary search tree algorithms. You may wish also to define private (recursive) helper functions. We provide a skeleton of one such helper function: copyInto, which helps toArray.

6 Files

The git repository for this homework contains the following files:

src/UnlockTest.java Unlock all the locked tests without running them.
src/TestAccount.java JUnit test case for the Account ADT.
src/TestInternals.java JUnit test case for the Account invariant.
src/TestEfficiency.java JUnit test case to check the efficiency of your code. If it takes longer than thirty seconds, you are doing something wrong.
src/edu/uwm/cs351/Account.java The skeleton of Account.
src/edu/uwm/cs351/Transaction.java The Transaction class.
src/edu/uwm/cs351/DateUtil.java Helper utility for working with LocalDateTime.
src/edu/uwm/cs351/Driver.java Driver to see Account class in action.

transactions_data Provided data for the driver.