Homework 2: Three Dictionaries

Code due: Tue., Oct. 15 at 11:59 PM (via GSC)
Self-eval due: Thu., Oct. 17 at 11:59 PM (on GSC)

You may work on your own or with one (1) partner.

The *dictionary* is a common abstract data type for representing a table of key-value pairs. For example, suppose you wanted a data structure for keeping track of your friends' birthdays:

Key	**Value**
Anne	Aug. 27, 1997
Bob	Mar. 5, 1997
Carol	Dec. 29, 1996

To represent such a table, you could use a dictionary with names as keys and birthdays as values. In this assignment, we consider dictionaries as bank ledgers mapping account numbers to account information.

There are many possible concrete implementations of the dictionary ADT, and in this assignment you will implement operations for three:

- a linked list,
- a binary search tree, and
- a sorted vector.

In `dictionaries.rkt`[1], I've supplied headers for the functions that you'll need to write, along with an insufficient number of tests.

Your task

The `Account` struct

In Homework 1, we used an `Account` class, but to keep things simple in Homework 2, we use a plain `Account` struct instead:

[1] `https://bit.ly/2p7Mm4f`

```
struct Account:
    let id: nat?
    let owner: str?
    let balance: num?
```

The annotations after the colons are contracts—predicates, in this case, that must hold for each field. That is, the `id` field must be a natural number, the `owner` field must be a string, and the `balance` field can be any kind of number.

The starter code also defines six example accounts, a function for cloning accounts (so that you can make copies of the example accounts to modify when testing), and an `account_transfer` function (used later).

The `LEDGER` interface

A ledger is, for our purposes, a collection of `Account`s that supports lookup by account ID. That is, it's a kind of read-only dictionary where the keys are account IDs and the values are the `Account`s themselves.

We plan to represent a ledger with three different concrete data structures, so first we express the ledger abstract data type (ADT) as an interface:

```
interface LEDGER:
    # How many accounts are in this ledger?
    def len(self) -> nat?

    # Looks up an account by ID, returning None if not found.
    def lookup(self, id: nat?) -> OrC(Account?, NoneC)
```

You will complete the implementations of three classes that satisfy the `LEDGER` interface.

The linked-list representation

The linked list data definition is as follows. First, the list itself is a singly-linked list of nodes:

```
# A BareList is one of:
#  - cons(Account, BareList)
#  - None
# where the `account.id` values are unique.
struct cons:
    let element: Account?
    let link:    OrC(cons?, NoneC)
```

As with the `Account` struct, the `cons` struct uses contracts to express constraints on its fields: The `element` field must be an `Account` struct, and the `link` field can be either `None` (the end of the list) or another `cons`.

We encapsulate the singly-linked list in a `ListLedger` class, whose one field, `contents`, is head of the `BareList` as described above.

```
class ListLedger (LEDGER):
    let contents: OrC(cons?, NoneC)

    def __init__(self):
        self.contents = None

    …
```

In addition to the constructor for `ListLedger`, the starter code contains a definition of a `push_back` method, which adds an account to the end of the list. Your job is to implement four additional methods:

1. `ListLedger.len(self) -> nat?` returns the number of accounts in the ledger, or equivalently, the number of nodes in the list.
2. `ListLedger.push_front(self, account: Account?) -> NoneC` adds an account to the front of the list.
3. `ListLedger.pop_front(self) -> Account?` removes the first node from the list and returns its account, or errors on empty.
4. `ListLedger.lookup(self, id: nat?) -> OrC(Account?, NoneC)` looks up an account by ID, returning `None` if no `Account` with the given ID can be found.

Note that accounts are stored in an order determined by the sequence of push and pop operations, which isn't expected to be related to any ordering

of their values. While it's possible that the linked list inside a `ListLedger` will contain multiple accounts having the same `id`, only one account object of a given `id` will be observable to clients of the `LEDGER` interface between modifications involving that same `id`.

A tiny example test is provided, but you need to write more.

The binary search tree representation

```
# A BareBst is one of
#  - branch(BareBst, Account, BareBst)
#  - None
# where for each branch `branch(l, acct, r)`, all the account IDs in
# `l` are less than `acct.id` and all the account.ids in `r` are
# greater than `acct.id`. (This is the binary search tree property.)
struct branch:
    let left:    OrC(branch?, NoneC)
    let element: Account?
    let right:   OrC(branch?, NoneC)
```

That is, a `BareBst` is either the empty tree, or a tree node containing one account and two subtrees with (potentially) more accounts. The accounts are stored in order of increasing account number à la binary search tree property. This means that lookups needs only proceed down a single path in the tree rather than searching everywhere, for time complexity $\mathcal{O}(\log n)$ (when the tree is balanced).

As with `ListLedger` above, the tree nodes are encapsulated in a class, `BstLedger`, which has one field to hold the root of the tree:

```
class BstLedger (LEDGER):
    let contents: OrC(branch?, NoneC)

    def __init__(self):
        self.contents = None

    …
```

In addition to the constructor for `BstLedger`, the starter code contains a

definition of an `insert` method, which adds an account to the bottom of the tree. The `insert` method does not necessarily produce a balanced tree—we'll learn how to do that later—but it does produce a predictable tree, which can be useful for testing.

Your job is to implement two additional methods:

5. `BstLedger.len(self) -> nat?` returns the number of accounts in the tree, or equivalently, the number of internal nodes.
6. `BstLedger.lookup(self, id: nat?) -> OrC(Account?, NoneC)` looks up an account by ID, returning `None` if no `Account` with the given ID can be found. This method must use the BST property to find the element rather than searching the whole tree.

Very few tests are provided, so you must write more.

The sorted vector representation

The third class uses a sorted vector. Unlike the linked list and binary search tree above, there are no auxiliary structs to define. The `VecLedger` class has one field, which points to a vector of accounts sorted by ascending ID:

```
class VecLedger (LEDGER):
    let contents: VecC[Account?]

    def __init__(self, v: VecC[Account?]):
        self._assert_sorted(v)
        self.contents = [ acct for acct in v ]

    …
```

Rather than provide a method for adding accounts to the vector, the constructor for `VecLedger` takes an already-sorted vector of accounts. Thus, the caller is responsible for supplying such a thing. Before initializing the `contents` field, the initializer method calls a private helper method, `VecLedger._assert_sorted`, to check that the vector meets the class's invariant. If so, then it stores a copy of the vector in `self.contents`.

In addition to the constructor for `VecLedger`, the starter code contains a definition of a `len` method. Your job is to implement two additional methods:

7. `VecLedger._assert_sorted(self, v)` scans vector `v`, checking that the account IDs are strictly increasing; it calls `error` if it discovers that they are not.

8. `VecLedger.lookup(self, id: nat?) -> OrC(Account?, NoneC)` looks up an account by ID, returning `None` if no `Account` with the given ID can be found. This method must take advantage of the invariant that the `contents` vector is sorted by account ID. This means that lookups proceed by binary search, not by a linear scan of the vector, for time complexity $\mathcal{O}(\log n)$.

A small number of tests are provided, but you should write more.

Working with the `LEDGER` ADT

Because all three ledger classes implement the `LEDGER` interface, we can write code that works with instances of all three of them. The starter code provides a function header:

```
def ledger_transfer(amount: num?,
                    from_id: nat?,
                    to_id: nat?,
                    ledger: LEDGER!):
```

This function should look up the accounts specified by `from_id` and `to_id` in `ledger`, and transfer `amount` from `from_id` to `to_id`. The contract annotation `LEDGER!` limits `ledger` to be used only according to the `LEDGER` interface, which is all that `ledger_transfer` needs in order to do its work. It should look up both accounts, calling `error` if either does not exist. Then it should perform the transfer using the provided `account_tranfer` function.

9. Complete the definition of the `ledger_transfer` function.

Deliverables

The provided file `dictionaries.rkt`[2], containing

- definitions of the eight methods and one function described above, and
- sufficient tests to be confident of your code's correctness.

[2] `https://bit.ly/2p7Mm4f`