EECS 214 Fall 2016

HW2: Three Dictionaries

Due: Thursday, October 12, at 11:59 PM, via 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 lookup functions 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 some tests.

Your task

Importing your account code

The entries in our ledgers will be `Accounts` from HW0. The provided source file `dictionaries.rkt` contains the `Account` data definition. You will have to copy your definitions of `account_transfer!` and `account_credit!` from

[1]`http://goo.gl/ZUuKwp`

HW0. If you are working with a partner, you may use either your definitions or your partner's.

The linked-list representation

The linked list data definition is as follows:

```
# A ListLedger is one of:
# - nil()
# - node(Account, ListLedger)
# where the `account.id` values are unique.
defstruct nil()
defstruct node(element, link)
```

That is, a `ListLedger` is either the empty list, or a list node containing one account and a list with (potentially) more accounts. The accounts are stored in no particular order.

Write the function

```
  list_lookup: AccountId ListLedger -> Account or False
```

which takes an account number and a ledger (in linked-list representation) and looks up the account with that number. If the account is found then it returns the account; otherwise it returns `False`.

The binary search tree representation

The binary search tree data definition is as follows:

```
# A BstLedger is one of
# - leaf()
# - branch(BstLedger, Account, BstLedger)
# where for a branch branch(l, acct, r), all the account.ids
# 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.)
defstruct leaf()
defstruct branch(left, element, right)
```

That is, a `BstLedger` 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 the 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).

Write the function

```
  bst_lookup: AccountId BstLedger -> Account or False
```

which takes an account number and a ledger (in binary search tree representation) and looks up the account with that number. The function must use the BST property to find the element rather than searching the whole tree. If the account is found then it returns the account; otherwise it returns `False`.

The sorted vector representation

The sorted vector data definition is as follows:

```
# A VecLedger is VectorOf<Account>
# where the account.id values are strictly ascending (and
# thus unique).
```

That is, a `VecLedger` is a vector of accounts sorted by account number. This means that lookups proceed by binary search, not by a linear scan of the vector, for time complexity $\mathcal{O}(\log n)$.

Write the function

```
  vec_lookup: AccountId VecLedger -> Account or False
```

which takes an account number and a ledger (in sorted vector representation) and looks up the account with that number. The function must binary search the array, not scan element by element. If the account is found then it returns the account; otherwise it returns `False`.

Then write the function

```
  transfer!: Number AccountId AccountId VecLedger -> Void
```

which takes an amount to transfer, two account numbers, and a sorted vector ledger, and transfers the indicated amount from the first account to the second. If either account does not exist, the function should call `error`.

Deliverables

The provided file `dictionaries.rkt`, containing

- definitions of the three lookup functions `list_lookup`, `bst_lookup`, and `vec_lookup`,
- a definition for `transfer!`, and
- sufficient tests to be confident of your code's correctness.