CS 474: Object Oriented Programming Languages and Environments
Spring 2018

First Smallalk project

Due time: 9:00 pm on Monday 2/19/2018

You work for the company *Big and small talks "R" us*, a software outfit that writes code exclusively in VisualWorks Smalltalk. Unfortunately, some inexperienced programmer in the compan y has wiped out the predefined *Dictionary* class of VisualWorks from your archives. Smalltalk dictionaries are data structures associating keys with values. For instance the key "politician" may have value "pathological liar". Now your company has to re-implement dictionaries. Alice and Bob, the two principals at *Big and small talks "R" us*, disagree as to how the new dictionaries should be implemented in order to support fast storage and retrieval of dictionary data. Alice thinks that dictionaries should be implemented using the predefined Smalltalk class *SortedCollection*, whereas Bob thinks that the implementation of dictionaries should use binary search trees. Since Smalltalk does not contain a binary search tree class, this class will have to be coded as well. You are now asked to implement dictionaries both ways. You are specifically responsible for implementing 3 classes named *NewDictionary, TreeDictionary* and *SortedDictionary*.

NewDictionary is an abstract class that inherits directly from class *Collection*. This class defines a common protocol for subclasses that support storage, insertion and retrieval of *(key, value)* pairs. Keys are Smalltalk strings; values are arbitrary Smalltalk objects. For this preliminary implementation, you may assume that values will be just numbers. *NewDictionary* declares the basic dictionary functionality, including deferred methods *at:*, *at:put:*, *allPairs*, *keys* and *values*.

Method *at:put:* takes a key and a value as arguments. It stores the corresponding (key, value) pair in the dictionary. If a pair with the same key was already present in the dictionary, the pair simply disappears. The receiver is returned. This method is also deferred.

Method *at:* takes a string (i.e., a *key*) as an argument and returns the corresponding *value* as an output. This method returns *nil* if the argument key is not contained in the dictionary. This method is deferred in *NewDictionary*.

Method *add:*, which is inherited from *Collection*, takes a (key, value) pair as an argument. It inserts the pair into the dictionary. If a pair with the same key was already stored in the dictionary, that previous pair is returned; the pair will be superseded by the new pair in the dictionary. This method is not deferred; it must be implemented in class *NewDictionary*. The method returns the pair with the same key that was already present in the dictionary or *nil*, if the key was not present in the dictionary before *add:* was called.

Method *allPairs* returns a new *OrderedCollection* containing all (key, value) pairs current stored in the dictionary. This method is implemented in class *NewDictionary* (i.e., it is not deferred).

Method *keys* returns a new *OrderedCollection* containing all keys (strings) current stored in the dictionary. This method is deferred in *NewDictionary*.

Method *values* returns a new *OrderedCollection* containing all values (numbers) current stored in the dictionary. This method is deferred in *NewDictionary*.

Finally, class *NewDictionary* rejects method *remove:ifAbsent:*, which is inherited from *Collection*. For this preliminary implementation, you are not required to remove pairs from dictionaries.

NewDictionary suclasses *SortedDictionary* and *TreeDictionary* are responsible for implementing the methods deferred in *NewDictionary*. In addition, these classes must implement deferred method *do:* which is inherited from *Collection*. This method takes as input a one-argument block and executes the method on each pair contained in the receiver (a dictionary). The receiver is returned. *SortedDictionary* uses the predefined Smalltalk class *SortedCollection* to hold the pairs in the dictionary. (Pairs will be sorted by key.)

For *TreeDictionary* you must implement a binary search tree class called *BST*. Recall that the root and every internal node in a BST has two children, subject to the property that a node *n*'s left child holds a value less than *n*'s value, and the right child holds a value greater than *n*'s value. You are not required to keep the tree balanced or to enforce the property that every internal node have two children. Each tree node will hold exactly one (key, value) pair; tree nodes are compared using the key of each pair only.

Your implementation should only maintain one dictionary at a time, whether this is a tree dictionary or a sorted dictionary. You must include a graphical user interface supporting the following functionality:

1. Creating a new empty tree dictionary–The previous dictionary is deleted.
2. Creating a new empty tree dictionary–The previous dictionary is deleted.
3. Adding a new (key, value) pair to the dictionary.
4. Retrieving a value stored in the dictionary by its key.
5. Applying a block to all elements of the current dictionary.
6. Printing all pairs in the dictionary in alphabetical order by keys.
7. Printing all keys in the dictionary in alphabetical order.
8. Printing all values in the dictionary in random order.

You must work alone on this project. Your project code should be in a special package called `CS474`. Save all your code by filing out that package in the file `xxx.st`, where `xxx` denotes your last name. Submit the file using the submit link in the assignment page of the Blackboard course web site. No late submissions will be accepted.

Hints. Define auxiliary classes as needed in order to make it easy to code and maintain your project. For instance the *BST* class may conveniently use a *Node* class for defining each tree node. Also, you may consider using a *DictionaryPair* class for working with (key, value pairs). Put all your code definitions in a new package and save your image often.