Framework-Based Software Development for Hand-Held Devices
Fall 2023
Project (Dart)

This goal of this project is to implement a portion of a library for collection classes in Dart. The library contains an abstract *Collection* class and two concrete subclasses. Your collections will contain values of an arbitrary type *T*. No duplicate values will be allowed.

1. Abstract superclass *Collection<T>* is the root of all concrete collection subclasses, where *T* is the type of the elements stored in the collection. This class defers instance implementation to its concrete subclasses; however, it does define some concrete methods that are will not be redefined in the subclasses. You may assume that, upon instantiation, data type *T* will support the needed relational operators (e.g., $\leq$, $<$, $\geq$, $==$, etc.)
2. Concrete *Collection<T>* subclass *BST<T>* will implement a collection as a binary search tree.
3. Concrete *Collection<T>* subclass *LinkedList<T>*, will implement a collection as a linked list.

Class *Collection<T>* declares the following public, deferred (abstract) methods:

- *add()*—This method takes as input an element of type *T* to be inserted in the receiver (i.e., an instance of a concrete *Collection* subclass). This method is defined in the subclasses. The modified receiver is returned.
- *copy()*—This no argument method returns a *Collection* subclass instance that is a deep copy of the receiver. The returned collection must be of the same type as the receiver.
- *operator[]()*—This is the indexing operator. This function takes as input an integer index and returns the element at the index position in the receiver. If the index is out of bounds, an error message is printed on the standard error stream and an exception is thrown. (Do not worry about catching the exception; you can let program execution terminate as a result of this exception.) When indexing a BST, use the in-order traversal of the tree to find the *i*-th element.
- *printString()*—This function prints all the elements in the receiver collection in their order on the console.

In addition the *Collection<T>* class defines the following concrete functions. The two concrete subclasses of *Collection<T>* are not allowed to override (redefine) these functions.

- A no-arg constructor.
- *map()*—This public function takes as input a function parameter *fn*. The parameter function *fn* takes as input a type-*T* object and returns also a *T* object. Function *map()* applies function *fn* to all elements contained in the receiver. The function stores the values returned by each execution of function *fn* in a new concrete collection of the same type as the receiver.
- *contains()*—This public function takes as input an arbitrary object (not necessarily of type *T*) and returns a boolean indicating whether the receiver contains the argument object or not.

- *equals()*—This the logical equivalence operator. This function takes as input an arbitrary object. The function returns a boolean indicating whether the receiver and the argument are logically equivalent. The two objects will be logically equivalent if (1) they are exactly the same data type, and (2) they contain the same values in the same order. (When comparing two trees, return true if the trees are isomorphic).

Finally, class *Collection<T>* declares a private, integer data member *_size*, which returns the number of elements contained in the receiver. *Collection<T>* defines a public getter for *_size*; however, only functions in *Collection<T>* are allowed to modify this data member.

Class *BST<T>* is a binary search tree implementation of abstract class *Collection<T>*. *BST<T>* implements all the deferred *Collection<T>* methods and the class constructor. The *copy()* method must return a deep copy of the receiver. You are required to allow clients of *BST<T>* to invoke the inherited methods *map()* and *contains()* but you cannot redefine these inherited methods in *BST<T>*.

When coding the indexing operator for the *BST<T>* class, return the i-th element you encounter in an in-order tree traversal. The first element you encounter will be at position 0. Finally, recall that a binary search tree is a tree subject to the following two conditions: (1) each node can have at most 2 children, and (2) nodes in the left subtree of each node n have values less than n's value and nodes in n's right subtree have values greater than n.

Class *LinkedList<T>* is similar to *BST<T>*; however, it is implemented as a traditional, singly-linked *LinkedList* class.

Implementation requirements. Define class *Collection<T>* to be a subclass of Dart class *Object*. You can use data type *Object* for objects of arbitrary classes. Except for no-arg functions, all your methods must include at least keyword (aka named) parameter. Finally, method *add()* must be coded in such a way that their invocations can be cascaded. You are not allowed to use Dart data-structure libraries, e.g., for your BST and linked list implementations.

Hint. I uploaded the *Collection* class that I presented in class in Week 4 in Blackboard. Feel free to reuse my code if that helps.

Your must work alone on this project. Students are not allowed to discuss detailed designs or share code with each other. You can discuss high-level designs with your colleagues but stop when you get down to designing actual algorithms and data structures. Make sure to test your code on a standard Dart 2.9 or later-version compiler before you submit. You are encouraged to use Piazza to post or answer questions about specific aspects of the project. Just remember never, never, ever to post code publicly on Piazza.