Joanna Klukowska
joannakl@cs.nyu.edu

Project 6: San Francisco Movies - One More Time

Due date: Decemeber 5, 11:55PM EST.

You may discuss any of the assignments with your classmates and tutors (or anyone else) but all work for all assignments must be entirely your own . Any sharing or copying of assignments will be considered cheating (this includes posting of partial or complete solutions on Piazza, GitHub or any other public forum). If you get significant help from anyone, you should acknowledge it in your submission (and your grade will be proportional to the part that you completed on your own). You are responsible for every line in your program: you need to know what it does and why. You should not use any data structures and features of Java that have not been covered in class (or the prerequisite class). If you have doubts whether or not you are allowed to use certain structures, just ask your instructor.

In this project you will revisit one last time the program from project 2 and project 3. The front-end (i.e., the user interface) part of the project is exactly the same as in the previous projects. The back-end part (i.e., the internal implementation and storage) will be different.

Objectives

The goal of this programming project is for you to master (or at least get practice on) the following tasks:

- revising previously written code
- using and implementing a binary search tree
- using and implementing iterators
- extending existing classes (inheritance)
- implementing existing interfaces (`Collection`, `Iterable`, `Iterator`)

Much of the code for this project has been already implemented in projects 2 and 3, but you need to implement a binary search tree for the first time and this may turn out to be time consuming. It is also your chance to fix any probelms that your previous implementations might have had.

Make sure to ask questions during recitations, in class and on Piazza.

Start early! This project may not seem like much coding, but debugging always takes time.

For the purpose of grading, your project should be in the package called `project6`. This means that each of your submitted source code files should start with a line:

```
package project6;
```

Keep in mind that spelling and capitalization are important! The package declaration line has to be the first line in your file!

Changes

In project 2, the `MovieList` class inherited from the `ArrayList` class. In project 3, it inherited from your own implementation of a linked list class. In this project, your `MovieList` class should inherit from your own implementation of a binary search tree class called `BST`. It should be a generic implementation.

Data Storage and Organization

You need to provide all the classes that you implemented for project 2. The `MovieList` class from project 2/3 might require some modification (in addition to the change from which class it inherits), but most of the other class can remain identical (you are encouraged to fix any errors you might have discovered in your previous implementations).

In addition you need to implement a generic `BST<E>` class.

You may implement additional classes and additional methods in the required classes, if you wish.

Joanna Klukowska
joannakl@cs.nyu.edu

`BST<E>` Class

This class should implement the Java's `Collection<E>` interface, https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html. This means that the class needs to provide all of the methods from the interface. It also means that the class needs to implement the `Iterable<E>` interface (since it is a superinterface of the `Collection<E>` interface).

Your class should not allow duplicate elements and it should not allow `null` elements.

For the purpose of this assignment you can leave some of the method **unimplemented** - this means that the method signature is there, but the body simply throws an `UnsupportedOperationException`. There are also **default** methods that are implemented in the interface itself and you can leave them as they are (unless you decide that you need to modify the default code).

All methods marked with arrows, ⇒ , below have be to **implemented** by your code - this means that there is code for those methods that actually performs the functionality required by the documentation.

In addition to the methods required by the `Collection<E>` interface, the class should provide the following public methods.

- ⇒ `E get( E value)`

  Returns the reference to the element stored in the tree with a value equal to the `value` passed as the parameter or `null` if no equal element exist in this tree.

  Parameters:

  `value` - an element to search for in this tree

  Returns: the reference to the element equal to the parameter `value` or `null` if not such element exists

- ⇒ `String toString()`

  Returns a string representation of this collection. The string representation consists of a list of the collection's elements in the order they are returned by its iterator (in this class this is the order of the inorder traversal), enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (comma and space). Elements are converted to strings as by `String.valueOf(Object)`.

  Returns: a string representation of this collection

- ⇒ `String toStringTreeFormat()`

  Returns a string representation of this collection in a tree-format. The details of this implementation are up to you and you are free to use the methods that we looked at in class.

```
/**
* Produces tree like string representation of this BST.
* @return string containing tree-like representation of this BST.
*/
public String toStringTreeFormat() {

    StringBuilder s = new StringBuilder();
    preOrderPrint(root, 0, s);
    return s.toString();
}

/*
* Uses pre-order traversal to produce a tree-like representation of this BST.
* @param tree the root of the current subtree
* @param level level (depth) of the current recursive call in the tree to
* determine the indentation of each item
* @param output the string that accumulated the string representation of this
* BST
*/
private void preOrderPrint(Node<E> tree, int level, StringBuilder output) {
    if (tree != null) {
        String spaces = "\n";
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
        if (level > 0) {
            for (int i = 0; i < level - 1; i++)
                spaces += "    ";
            spaces += "|--";
        }
        output.append(spaces);
        output.append(tree.data);
        preOrderPrint(tree.left, level + 1, output);
        preOrderPrint(tree.right , level + 1, output);
    }
    else {   //print the null children
        String spaces = "\n";
        if (level > 0) {
            for (int i = 0; i < level - 1; i++)
                spaces += "    ";
            spaces += "|--";
        }
        output.append(spaces);
        output.append("null");
    }
}
```

- ⇒ **`Iterator<E>  preorderIterator()`**
  Returns an iterator over the elements in this collection. The elements should be returned in the order determined **by the preorder traversal of the tree**.

- ⇒ **`Iterator<E>  postorderIterator()`**
  Returns an iterator over the elements in this collection. The elements should be returned in the order determined **by the postorder traversal of the tree**.

The class should also provide the following methods that are described in the **`TreeSet<E>`** class specification (https://docs.oracle.com/javase/10/docs/api/java/util/TreeSet.html), but are not part of the **`Collection<E>`** interface.

⇒ **`E  ceiling (E e)`**
Returns the least element in this set greater than or equal to the given element, or null if there is no such element.

⇒ **`Object  clone()`**
Returns a shallow copy of this tree instance.

⇒ **`E  first()`**
Returns the first (lowest) element currently in this set.

⇒ **`E  floor (E e)`**
Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.

⇒ **`E  higher (E e)`**
Returns the least element in this set strictly greater than the given element, or null if there is no such element.

⇒ **`E  last()`**
Returns the last (highest) element currently in this set.

⇒ **`E  lower (E e)`**
Returns the greatest element in this set strictly less than the given element, or null if there is no such element.

Methods from the `Collection` interface

⇒ **`boolean  add (E e)`**
Ensures that this collection contains the specified element (optional operation). Note: **You need to implement this method.**

Joanna Klukowska
joannakl@cs.nyu.edu

`boolean addAll (Collection<? extends E> c)`
Adds all of the elements in the specified collection to this collection (optional operation).

⇒ `void clear()`
Removes all of the elements from this collection (optional operation). Note: **You need to implement this method.**

⇒ `boolean contains (Object o)`
Returns true if this collection contains the specified element.

⇒ `boolean containsAll (Collection<?> c)`
Returns true if this collection contains all of the elements in the specified collection.

⇒ `boolean equals (Object o)`
Compares the specified object with this collection for equality.

`int hashCode()`
Returns the hash code value for this collection. Note: **You can use the method inherited from the Object class.**

⇒ `boolean isEmpty()`
Returns true if this collection contains no elements.

⇒ `Iterator<E> iterator()`
Returns an iterator over the elements in this collection. The elements should be returned in the order determined **by the inorder traversal of the tree**.

`default Stream<E> parallelStream()`
Returns a possibly parallel Stream with this collection as its source.

⇒ `boolean remove (Object o)`
Removes a single instance of the specified element from this collection, if it is present (optional operation). Note: **You need to implement this method.**

`boolean removeAll (Collection<?> c)`
Removes all of this collection's elements that are also contained in the specified collection (optional operation).

`default boolean removeIf (Predicate<? super E> filter)`
Removes all of the elements of this collection that satisfy the given predicate.

`boolean retainAll (Collection<?> c)`
Retains only the elements in this collection that are contained in the specified collection (optional operation).

⇒ `int size()`
Returns the number of elements in this collection.

`default Spliterator<E> spliterator()`
Creates a Spliterator over the elements in this collection.

`default Stream<E> stream()`
Returns a sequential Stream with this collection as its source.

Joanna Klukowska
joannakl@cs.nyu.edu

⇒ `Object[]  toArray()`
Returns an array containing all of the elements in this collection.

⇒ `<T> T[]  toArray (T[] a)`
Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Nodes

Your program should provide and use an internal class that provides nodes for your tree. The details of the implementation of that class are up to you. The internal class should be private! (It needs to be documented!)

Iterator

Your `BST<E>` class implements `Iterable<E>` interface. This means that its `iterator()` method needs to return an instance of a class that implements the `Iterator<E>` interface. The `iterator()` method should return an iterator instance that accesses the values in the tree according to the inorder traversal of the binary search tree. The two additional methods `preorderIterator()` and `postOrederIterator()` need to return iterators that access the values in the tree according to the preorder and postorder traversals, respectively.

The details of the implementation are up to you and you are free to implement more than one internal private iterator class. The `remove` method in the `Iterator<E>` interface is optional and you do not need to provide the actual remove functionality. (This means that the method has to exist, but instead of performing its function, it throws an instance of `UnsopportedOperationException`.)

Programming Rules

You should follow the rules outlined in the document *Code conventions* posted on the course website at https://cs.nyu.edu/~joannakl/cs102_f18/notes/CodeConventions.pdf.

The data file should be read only once! Your program needs to store the data in memory resident data structures.

You may use any exception-related classes.

You may use any classes to handle the file I/O, but probably the simplest ones are `File` and `Scanner` classes. You are responsible for knowing how to use the classes that you select.

For the binary search tree implementation you should use the textbook, lecture material and source code of Java built-in classes as a guide. You are free to look at the `TreeSet` class implemented in Java, but be warned that that class implements a red-black balanced binary search tree.
As usual, you should give credit to any sources you are using.

Working on This Assignment

You should start right away!

You should modularize your design so that you can test it regularly. Make sure that at all times you have a working program. You can implement methods that perform one task at a time. This way, if you run out of time, at least parts of your program will be functioning properly.

You should make sure that you are testing the program on much smaller data set for which you can determine the correct output manually. You can create a test input file that contains only a few rows.

You should make sure that your program's results are consistent with what is described in this specification by running the program on carefully designed test inputs and examining the outputs produced to make sure they are correct. The goal in doing this is to try to find the mistakes you have most likely made in your code.

Joanna Klukowska
joannakl@cs.nyu.edu

Each class that you submit will be tested by itself without the context of other classes that you are implementing for this assignment. This means that you need to make sure that your methods can perform their tasks correctly even if they are executed in situations that would not arise in the context of this specific program.

You should backup your code after each time you spend some time working on it. Save it to a flash drive, email it to yourself, upload it to your Google drive, do anything that gives you a second (or maybe third copy). Computers tend to break just a few days or even a few hours before the due dates - make sure that you have working code if that happens.

Grading

If your program does not compile or if it crashes (almost) every time it is run, you will get a zero on the assignment. Make sure that you are submitting functioning code, even if it is not a complete implementation.

If the program does not adhere to the specification, the grade will be low and will depend on how easy it is to figure out what the program is doing.

50 points class correctness: correct behavior of methods of the required classes and correct behavior of the program (the emphasis will be placed on the `BST<E>` class correctness)

30 points design and the implementation of the required classes and any additional classes

20 points proper documentation, program style and format of submission

How and What to Submit

For the purpose of grading, your project should be in the package called `project6`. This means that each of your submitted source code files should start with a line:

```
package project6;
```

Your should submit all your source code files (the ones with .java extensions only) in a single **zip** file to Gradescope.

You can produce a zip file directly from Eclipse:

- right click on the name of the package (inside the `src` folder) and select Export...
- under General pick Archive File and click Next
- in the window that opens select appropriate files and settings:
  - in the right pane pick ONLY the files that are actually part of the project, but make sure that you select all files that are needed
  - in the left pane, make sure that no other directories are selected
  - click Browse and navigate to a location that you can easily find on your system (Desktop or folder with the course materials or ...)
  - in Options select ”Save in zip format”, ”Compress the contents of the file” and ”Create only selected directories”
- click Finish