Project #3 Solution Description

Nicholas Mills

2/21/18

UMUC

CMSC 350

Ioan Salomie

**1. Assumptions, main design decisions and error handling**

Assumptions

* Spaces separate tokens.
* Input solely deals with integers and fractions. The program does not support floating point numbers.
* There should be no extra elements in the input other than the digits and slashes for fractions. Any anomalous elements will result in an exception being thrown.
* The input will be no more than 30 characters, and the output will be no more than 20. The program should still calculate properly, but the field will be too small to show the full values.

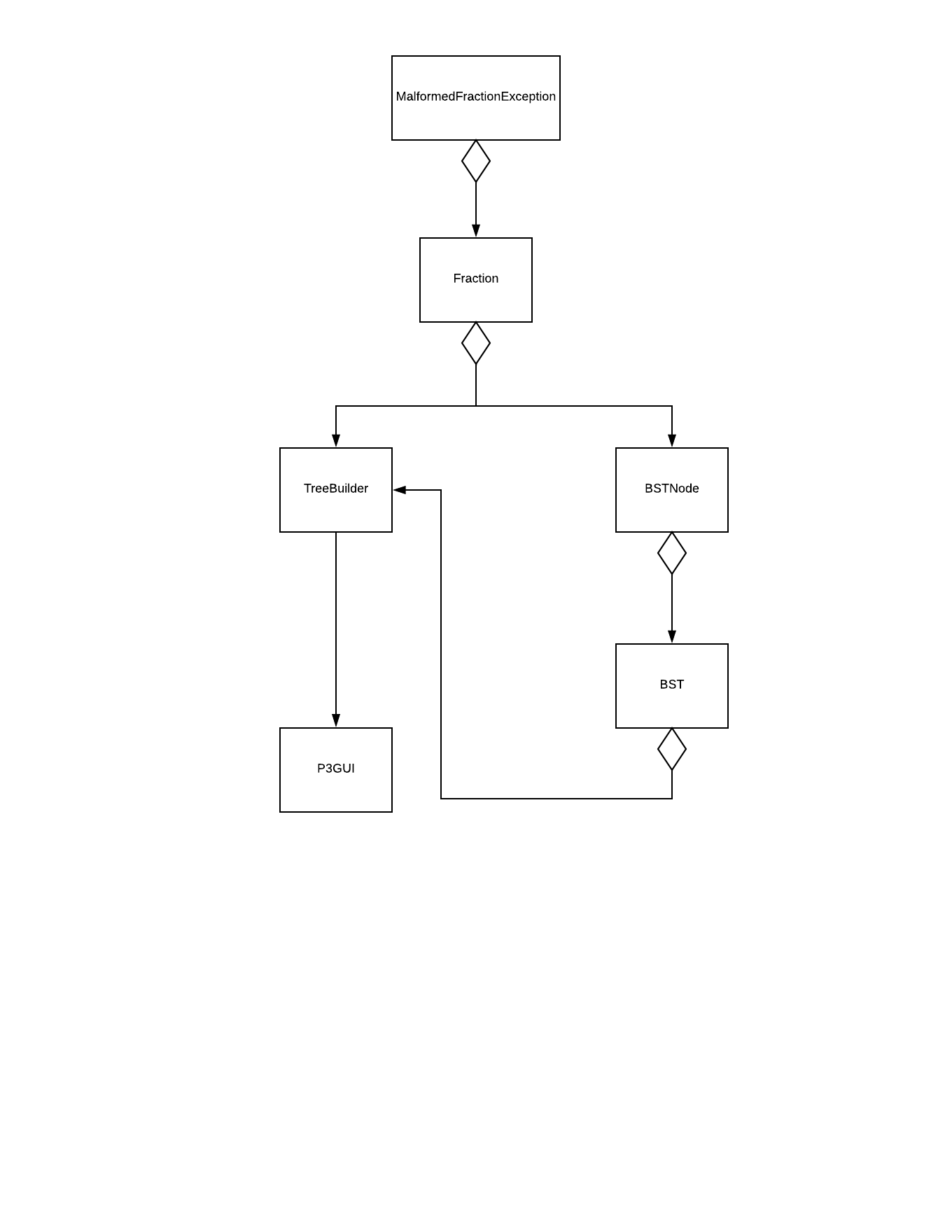
Main design decisions

* For tokenizing the string, I used a Scanner, since it uses space delimiters by default.
* For the GUI, I used a BoxLayout combined with 6 different panels to organize the different elements. In order to get the radio buttons to appear as they did in the prompt, I used a combination of a GridLayout and TitledBorders created in a BorderFactory.
* The most complex and time-consuming part of the project was attempting to parse the tokens without having to repeat large amounts of code. To accomplish this, I retrieved the String-based constructors for Integers and the Fraction class using the Constructor class and passed them to the TreeBuilder. Since these constructors function similarly, creating a numeric object based on a passed string and throwing an exception if said string does not meet the proper format, I was able to cut down on the amount of code used in token parsing considerably. This required me to have a couple of unchecked casts in my code, but since I am guaranteed that the input will be of the proper type through exception handling and minimal options, I decided that it was worth the warnings.
* For the Fraction toString method, I simply kept the passed string. For the comparison, I parsed the Fractions into two parts divided by the slash and then calculated their value accordingly.
* The BST class creates trees based on Comparable Objects passed to it, shuffling lower valued items down the left side of the tree and the higher valued ones to the right using a recursive insertNode structure. It also can read these values forward or backward depending on a passed String variable which dictates sort order through the inOrderTraversal method.

Error handling

* If the string does not fit the proper format, the MalformedFractionException was thrown by the constructor. This is passed by getTokens and eventually caught by P3GUI and shown to the user.
* In the getTokens method of TreeBuilder, I check to make sure that no characters other than slashes and digits were in the input. If there were, I throw a NumberFormatException. Due to the way that the Integer constructor works, it also automatically throws a NumberFormatException if it encounters anomalous characters, so slashes in non-fraction inputs also trigger said exception. NumberFormatExceptions inform the user of non-numeric input via dialog box.
* Because of the way the Constructor class works, it throws exceptions that are not the exceptions that are thrown by the Constructors that they are called. Therefore, I had to figure out which initial Exception caused the Constructor Exception and pass the correct Exception along to the P3GUI.
* The Constructor class uses file names as Strings to access constructors of classes. In case these strings become invalid somehow, I included a stack trace and dialog box to help deal with that issue.

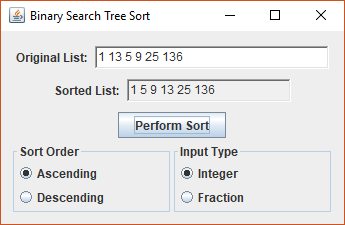
**2. UML class diagrams**

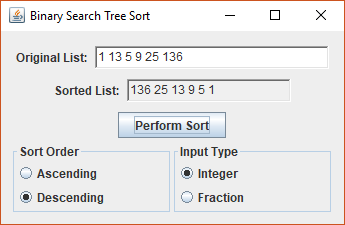
****

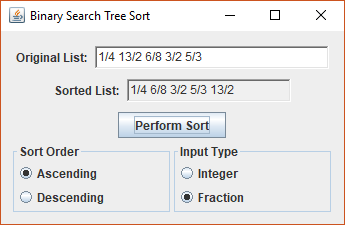
**3. Test cases**

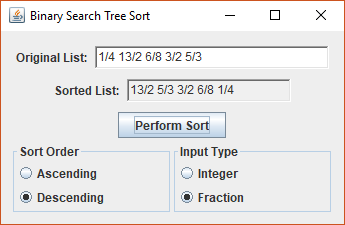
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What aspect is tested** | **Input** | **Expected Output** | **Actual Output** | **Pass / Fail** |
| Integers Ascending | 1 13 5 9 25 136 | 1 5 9 13 25 136 | 1 5 9 13 25 136 | **P** |
| Integers Descending | 1 13 5 9 25 136 | 136 25 13 9 5 1 | 136 25 13 9 5 1 | **P** |
| Fractions Ascending | 1/4 13/2 6/8 3/2 5/3 | 1/4 6/8 3/2 5/3 13/2 | 1/4 6/8 3/2 5/3 13/2 | **P** |
| Fractions Descending | 1/4 13/2 6/8 3/2 5/3 | 13/2 5/3 3/2 6/8 1/4 | 13/2 5/3 3/2 6/8 1/4 | **P** |
| Integer NumberFormatException | 1/4 13/2 6/8 3/2 5/3 | Non-numeric input. | Non-numeric input. | **P** |
| Fraction NumberFormatException w/ Letters | afdasfd | Non-numeric input. | Non-numeric input. | **P** |
| Malformed  Fraction  Exception | 3/6/2 | Malformed fraction. | Malformed fraction. | **P** |

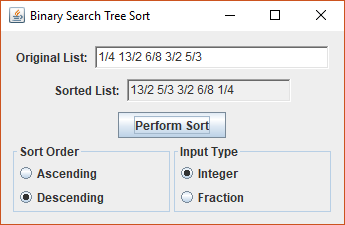
**4. Screenshots**

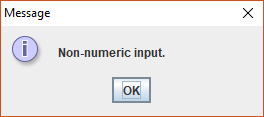


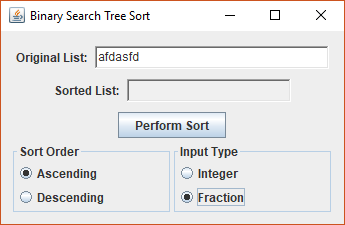


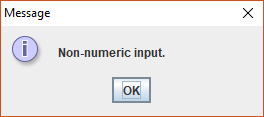


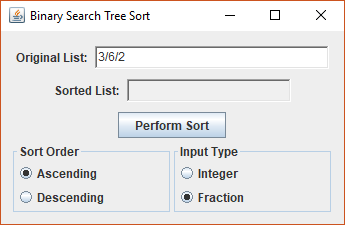


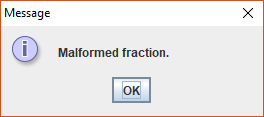












**5. Lessons learned**

**Learn to let go.** I spent days on end thinking about how to figure out how to condense the code for this project. A lot of why I am turning this in so late is due to that fixation. I probably would have been less satisfied with the outcome had I needed to create multiple different string parsing methods, but the project would have been done much, much sooner than it was in this way. What’s worse, as far as I can tell, reflection styles are generally seen in a poor light so the solution that I came up with, while it did suit my whim to cut down the amount of repeated code, is ultimately inelegant. The project still is not where I want it to be even after all this time and refinement but ultimately I *have* to let it go at this point because the longer I work on it the more my grade suffers.

On the bright side, I did learn a lot about reflection and generic classes and the ways they can and cannot, should and should not be used, even though that wasn’t really the main point of this project as far as I can tell.

Another lesson learned is that no matter how much one wants to, it is not always possible to simply brute force one’s way through a problem. Sitting down for hours and staring at the screen without writing anything is not productive. If I’m too fried to think clearly, I should step away for a while and come back to it with fresh eyes. This would have probably ultimately been more efficient than the alternative.

Lastly, the notion of doing things the “right way” can be just as much of a hinderance as it is a helpful guideline. Part of what took me so long to finish was my drive to eliminate all warnings from every part of my code while still being able to pass along the class or constructor to the generic and trying to accomplish both of those at once prevented me from even getting past the token processing part of the issue until this past night. All of the Binary Sort logic took me all of an hour or two to deal with. It’s better to get something that works in time than something that’s more refined but extremely late. Perfection is the enemy of good enough. I need to remember that.