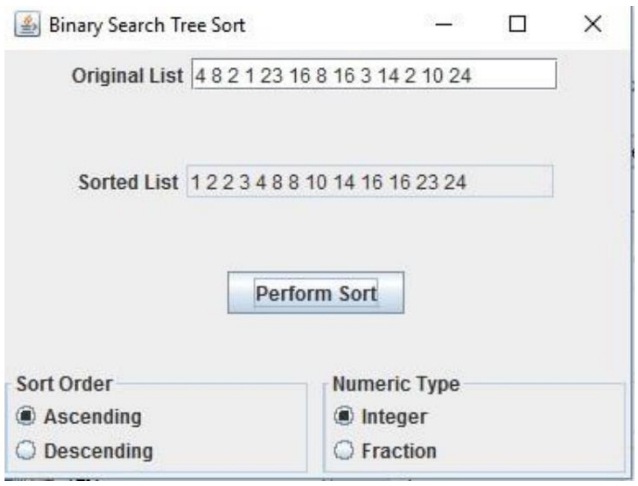
UMUC

CMSC 350 Project 3

1. Specification

Design, write and test a program that performs a sort by using a binary search tree. The program should be able to sort lists of integers or lists of fractions in either ascending or descending order. One set of radio buttons should be used to determine whether the lists contain integers or fractions and a second set should be used to specify the sort order.
The main class **P3GUI** should create the Swing based GUI shown below. The GUI must be generated by code that you write. You may not use a drag-and-drop GUI generator.



Original List: 4 8 2 1 23 16 8 16 3 14 2 10 24

Sorted List: 1 2 2 3 4 8 8 10 14 16 16 23 24

Sort Order: [✓ Ascending] Descending

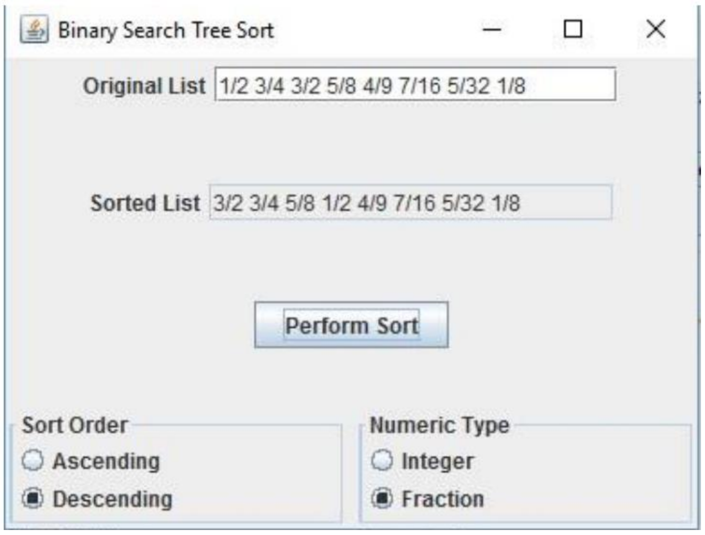
Numeric Type: [✓ Integer] Fraction

Pressing the *Perform Sort* button should cause all the numbers in the original list to be added to a binary search tree. Then, an inorder traversal of the tree should be performed to generate the list in sorted order and that list should then be displayed in the *Sorted List* text field.

In addition to the main class that defines the GUI, you should have a generic class for the binary search tree. That class needs a method to insert a new value in the tree and a method that performs an inorder tree traversal that generates and returns a string that contains the tree elements in sorted order. The insert method does not need to rebalance the tree if it becomes unbalanced. It should allow duplicate entries and it must be written using recursion. Other methods may be defined if needed.

The third class required for this project is one that defines fractions. It should have a constructor that accepts a string representation of a fraction and a `toString` method. It must implement the `Comparable` interface, which means a `compareTo` method is also required.

A second example of a run of this program is shown below that sorts fractions in descending order:



Original List: 1/2 3/4 3/2 5/8 4/9 7/16 5/32 1/8

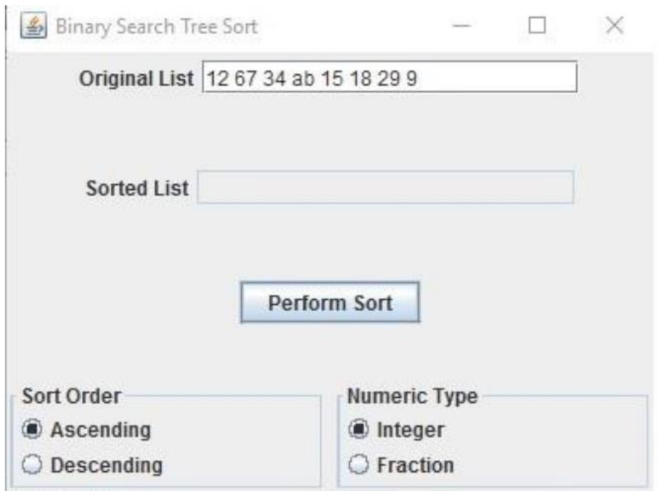
Sorted List: 3/2 3/4 5/8 1/2 4/9 7/16 5/32 1/8

Sort Order: Ascending [✓ Descending]

Numeric Type: Integer [✓ Fraction]

Note that fractions are to be written with a slash separating the numerator and denominator with no spaces on either side of the slash. Spaces should be used to separate the fractions.

The only error checking required of this program is to check for nonnumeric input or improperly formatted fractions such as `3/4/8`. A nonnumeric input should cause a `NumberFormatException` to be thrown. A malformed fraction should cause a custom exception `MalformedFractionException` to be thrown. The main class must catch these exceptions and display an appropriate error message as shown below for `NumberFormatException` (for `MalformedFractionException` the displayed message should be "Malformed Fraction").

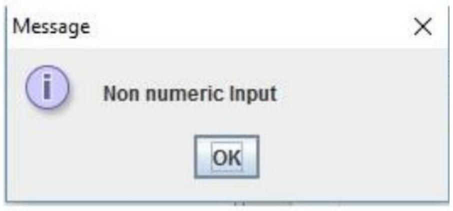


Original List: 12 67 34 ab 15 18 29 9

Sorted List:

Sort Order: [✓ Ascending] Descending

Numeric Type: [✓ Integer] Fraction



Non numeric Input

Your program should compile without errors.

The Google recommended Java style guide (https://google.github.io/styleguide/javaguide.html) should be used to format and document your code. Specifically, the following style guide attributes should be addressed: header comments include filename, author, date and brief purpose of the program; In-line comments used to describe major functionality of the code; the meaning and the role of variables and constants are indicated as code comments; meaningful variable names and prompts applied; class names are written in UpperCamelCase; variable names are written in lowerCamelCase; constant names are in written in All Capitals; braces use K&R style.

In addition the following design constraints should be followed: declare all instance variables private; avoid the duplication of code.

2. Submission requirements

Submit the following to the Project 3 assignment area no later than the due date listed in your LEO classroom.

1. All **.java** source files (**no other file types should be submitted)**. The source code should use Java code conventions and appropriate code layout (white space management and indents) and comments. All submitted files may be included in a .zip file.

2. The solution description document **P3SolutionDescription** (.pdf or .doc / .docx) containing the following:
(1) Assumptions, main design decisions, error handling;
(2) A UML class diagram that includes all classes you wrote. Do not include predefined classes. You need only include the class name for each individual class, not the variables or methods;
(3) A table of test cases including the test cases that you have created to test the program. The table should have 5 columns indicating (i) what aspect is tested, (ii) the input values, (iii) the expected output, (iv) the actual output and (v) if the test case passed or failed. Each test case will be defined in a table row.
(4) Relevant screenshots of program execution;
(5) Lessons learned from the project;

Grading Rubric:

Criteria	Meets	Does Not Meet
Design	**5 points**	**0 points**
	GUI is hand coded and matches required design	GUI is generated by a GUI generator or does not match required design
	Includes generic class for binary search tree	Does not include generic class for binary search tree
	All instance variables are private	Instance variables are not private
	Insert method uses recursion	Insert method does not use recursion
	Contains Fraction class that implements Comparable	Does not contain Fraction class that implements Comparable
	Uses good object-oriented design practice regarding code efficiency, encapsulation and information hiding, class and code reuse, high cohesion of classes, avoiding code duplication.	Does not use good object-oriented design practice regarding code efficiency, encapsulation and information hiding, class and code reuse, high cohesion of classes, avoiding code duplication.
Functionality	**10 points**	**0 points**
	Correctly sorts all test cases in ascending order	Does not correctly sort all test cases in ascending order
	Correctly sorts all test cases in descending order	Does not correctly sort all test cases in descending order
	Correctly sorts all test cases involving integers	Does not correctly sort all test cases involving integers
	Correctly sorts all test cases involving fractions	Does not correctly sort all test cases involving fractions
	Handles the exceptions and generates error messages for nonnumeric and malformed fraction input	Does not handle the exceptions and generate error message for nonnumeric and malformed fraction input
Test Cases	**5 points**	**0 points**
	Test cases table is defined and included in the P3SolutionDescription document	Test cases table is not defined and included in the P3SolutionDescription document
	Test cases include integers	Test cases do not include fractions
	Test cases include fractions	Test cases do not include a case to test invalid token beginning with a digit
	Test cases include nonnumeric and malformed fraction input	Test cases do not include nonnumeric and malformed fraction input
Documentation	**5 points**	**0 points**
	Solution description document P3SolutionDescription includes all the required sections (appropriate titled).	No solution description document is included
	Source code follows Google recommendation Java style	Source code does not follow Google recommendation Java style
	Comment blocks with class description included with each class	Comment blocks with class description not included with each class
	Source code is commented and indented	Source code is not commented and indented
Overall Score	**Meets**	**Does not meet**
	16 or more	15 or less