Project 2: Stacks, Queues, and Trees

DUE: Sunday, March 1st at 11:59pm

Setup

- Download the `project2.zip` and unzip it. This will create a folder `section-yourGMUUserName-p2`.
- Rename the folder replacing `section` with the `s001`, `s003`, `s004`, `s005` based on the lecture section you are in.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address.
- After renaming, your folder should be named something like: `s001-krusselc-p2`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p2.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
s001-krusselc-p2.zip --> s001-krusselc-p2 --> JavaFile1.java
                                              JavaFile2.java
                                              JavaFile3.java
                                              ...
```

- Submit to blackboard. DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED IS THE RIGHT THING. ***Submitting the wrong files will result in a 0 on the assignment!***

Basic Procedures

You must:

- Have code that compiles with the command: **`javac *.java`** in your user directory.
- Have code that runs with the command: **`java SimpleCompiler [filename] [true|false]`**

You may:

- Add additional methods and variables, however these methods **must be private**.
- Add additional nested, local, or anonymous classes, but any nested classes must be private.

You may NOT:

- Make your program part of a package.
- Add additional *public* methods, variables, or classes. You may have public methods in private nested classes.
- Use any built in Java Collections Framework classes in your program (e.g. no **`LinkedList`**, **`ArrayList`**, etc.).
- Create any arrays anywhere in your program, except the **`toArray()`** method of the **`CallStack`** class. You *may not* call the **`toArray()`** method of the stack to bypass this requirement.
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided classes or methods that are complete (**`Node`**, **`runCompiler()`** in **`SimpleCompiler`**, etc.).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

By the end of this project, you will have defined a stack class, a *binary search tree* (more information below), and a simple compiler that takes as input a postfix program and not only compiles it, but also evaluates it and prints the output!. You will use file I/O to read programs from files and evaluate them using your "SimpleCompiler". If you're unfamiliar with postfix notation, the following programs (and their more familiar infix versions) are shown below:

Postfix Program	Equivalent Infix Program	Output	Notes
3 2 +	3 + 2		This program takes 3 and 2, then adds them.
3 2 + print	print 3 + 2	5	This program takes the output from "3 2 +" and prints it. Print is an operator with one parameter.
x 3 2 + = x print	x = 3 + 2 print x	5	This program adds 3 and 2, then stores it in a variable called x. Later, it is asked to print x.

How the SimpleCompiler Works

When you have completed the assignment, your "SimpleCompiler" will have two modes: normal and debug. In normal mode, the SimpleCompiler will do all the computations and display the output of any print statements. In debug mode, the SimpleCompiler will perform one "step" at a time. The SimpleCompiler is run in the following ways:

```
java SimpleCompiler [path-to-program] [debug-mode-true|false]
```

For example, when I run the SimpleCompiler *without* debug mode on one of the provided sample programs (sample1.txt in this case), I type and see the following:

```
> java SimpleCompiler ../prog/sample1.txt false

Program: 1 2 + 3 4 - 5 6 + 4 2 / * * + print
-19
```

And when I run the SimpleCompiler *with* debug mode, I type and see the following:

```
> java SimpleCompiler ../prog/sample1.txt true

Program: 1 2 + 3 4 - 5 6 + 4 2 / * * + print

######### Step 1 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
1
----------Program Remaining----
2 + 3 4 - 5 6 + 4 2 / * * + print

Press Enter to Continue
```

When I press enter I see:

```
######### Step 2 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
1 2
----------Program Remaining----
+ 3 4 - 5 6 + 4 2 / * * + print
```

`Press Enter to Continue`

The debug mode and printing have already been done for you, but your job will be to build the supporting data structures and processing the program. You will do this in stages:

- Step 1: Read the input from a file into a queue (a linked list queue!)
- Step 2: Create a stack to support the SimpleCompiler's computations (a linked list stack!) and process simple math programs (e.g. "**`3 2 + print`**").
- Step 3: Create a binary search tree (more information below) which would serve as a symbol table to support the SimpleCompiler when looking up variable names and processing math/printing with variables.

More information on these stages is given in the later parts of this document.

tl;dr You're building a postfix compiler with a debug-mode command. Commands for running the compiler are above.

Postfix Program Processing with a Stack and No Variables

The basic procedure is as follows:

1. Treat the incoming program as a "queue" (the left side in the front of the queue).
2. When you encounter a value, push it on the stack (i.e., Call Stack).
3. When you encounter an operator, pop what you need off the stack and perform the operation.
4. Depending on the operator, you might need to push the result back on the stack.

Example:

Input Queue	Current Symbol	Call Stack	Notes
3 2 + 4 2 - * 1 + print			We start with our input in a queue and nothing on the stack.
2 + 4 2 - * 1 + print	3		We examine the first input in the queue and discover it is a value (not an operator), so we will put it on the stack.
+ 4 2 - * 1 + print	2	3	The next input is also a value, so we will put it on the stack (top of the stack is to the right).
4 2 - * 1 + print	+	3 2	Now we have an operator. Addition takes two values, so we pop twice from the stack and add those two. The result of addition goes back on the stack, the "+" sign is thrown away.
2 - * 1 + print	4	5	Value, goes on the stack.
- * 1 + print	2	5 4	Value, goes on the stack.
* 1 + print	-	5 4 2	Subtraction operator, pop twice, subtract, push result.
1 + print	*	5 2	Multiplication operator, pop twice, multiply, push result.
+ print	1	10	Value, goes on the stack.
Print	+	10 1	Addition operator, pop twice, add, push result.
	print	11	The print operator takes only one value, so we pop once and print that value. Additionally, printing does not push anything new onto the stack!
			All done, nothing in queue!

Note: Chapter 11 of your textbook has some useful information about postfix calculations if you want more info!

tl;dr The compiler will take programs like "1 2 + print" and print the number 3. See Chapter 11 for some more info.

Postfix Program Processing with a Stack and Variables

The basic procedure is as follows:

1. Treat the incoming program as a "queue" (the left side in the front of the queue).
2. When you encounter a non-operator, push it on the stack.

3. When you encounter an operator, pop what you need off the stack. If you pop a variable off the stack, look it up in the "LookUpBST" which is a binary search tree. Perform the operation with the values you've discovered.
5. Depending on the operator, you might need to push the result back on the stack.

Example:

Input Queue	Current Symbol	Call Stack	LookUp BST	Notes
x 3 2 + = y 4 = x y + print				We start with our input in a queue and nothing on the stack.
3 2 + = y 4 = x y + print	x			x is not an operator, so push it on the stack.
2 + = y 4 = x y + print	3	x		Not an operator, goes on the stack.
+ = y 4 = x y + print	2	x 3		Not an operator, goes on the stack.
= y 4 = x y + print	+	x 3 2		Operator; addition takes two values, pop twice and push result.
y 4 = x y + print	=	x 5		This is an assignment operator, so start by popping twice. The first thing you pop with be the value of the variable and the second thing you pop will be the variable name. These will go in the LookUpBST, and nothing new goes on the stack.
4 = x y + print	y		x:5	Not an operator, goes on the stack.
= x y + print	4	y	x:5	Not an operator, goes on the stack.
x y + print	=	y 4	x:5	Assignment, pop twice and add to LookUpBST.
y + print	x		x:5 y:4	Not an operator, goes on the stack.
+ print	y	x	x:5 y:4	Not an operator, goes on the stack.
Print	+	x y	x:5 y:4	Addition operator, pop twice, add, push result. When you try to add variables to each other (or variables to values like 5), you need to look them up in the LookUpBST.
	print	9	x:5 y:4	Print operator, so pop 9 off the stack and print it. Nothing new goes on the stack.
			x:5 y:4	All done, nothing in queue!

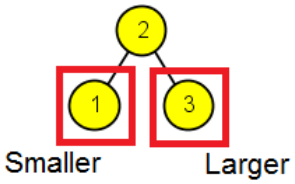
tl;dr The final version of the compiler will be able to store variables in a "LookUpBST" which is a binary search tree. Then the compiler will be able to take programs like "x 1 = x print" and print the number 1 (the value of x).

Learn a New Data Structure and Related Algorithms!

This project will require you to learn a new data structure all on your own! (But not really, we're going to give you a lot of support ☺.)

Binary *Search* Trees (BSTs)

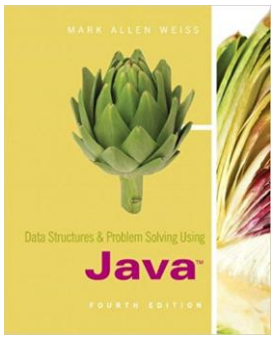
We've covered (or will cover soon) *binary* trees in class (trees where nodes can have at most two children – left and right), but you're now going to do binary *search* trees. These are binary trees where there is a search property associated with it:



For any given node:

- The left child's value is smaller than the given node's value
- The right child's value is larger than the given node's value
- Both the left and right subtrees are binary search trees (meaning this property holds all the way down)

Tools for Learning Binary Search Trees



1) **Textbook** –The textbook has an entire chapter on binary search trees! Who knew? Chapter 19 (specifically 19.1-19.3) has all the information you need to understand how BSTs work and the implementation details (the textbook even includes code). Chapter 18 also has a lot of useful information and code for tree traversals (walks).

   Note that you do not need to do *balancing* for this assignment (so at the moment you don't need sections 19.4-19.8; we'll cover balancing trees later in the semester).

   [WARNING] You are not allowed to copy and paste the textbook code into your assignment because: (1) it won't work as-is – the structure on your assignment is a little different and the problems you are addressing are different as well, (2) you won't learn a lot by copy-and-pasting, and (3) we said so – see 2. You should: *read* the textbook, use it to *understand* what you are trying to do, then *write* your own solution.

2) **Gnarley Trees** – We've uploaded the stand-alone jar of Gnarley Trees to: Piazza -> Resources -> Resources Tab -> Tools and Resources -> gt.jar (https://piazza.com/gmu/spring2020/cs310/resources)

   Gnarley Trees is a visualizer for trees you can use to get a better understanding of BSTs and how they work. It's especially helpful when trying to determine what your code should be doing when inserting values. Likely Gnarley Trees will be demoed in class, and you will find this a valuable tool for more complicated trees later in the semester.

   Gnarley Trees original jar source from: https://people.ksp.sk/~kuko/gnarley-trees/

3) **Videos** – Prefer to sit back and watch someone else explain? Here are some videos:
   - BST Basics and Motivation: https://www.youtube.com/watch?v=pYT9F8_LFTM
   - BST Insertion and Deletion Algorithms: https://www.youtube.com/watch?v=wcIRPqTR3Kc

   [WARNING] These videos are the only authorized outside resources for this assignment, the rest of the internet (including related videos, Google searches, etc.) are still not authorized. Please understand that you will be referred to the honor court if you are found to have copied/reproduced code from the internet. We don't mind you *learning* from other resources, but often students become confused between learning and copying, so we are only authorizing the videos above which do not contain code. Use your judgment when viewing other resources.

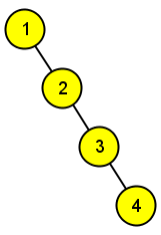
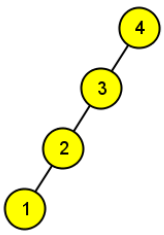
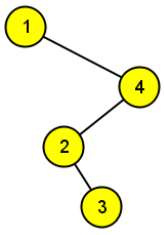
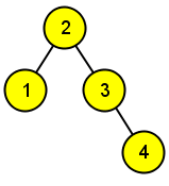
Binary Search Tree Common Mistakes

Once you think you're comfortable with BST basics, take a look at the list of common mistakes below to ensure you won't make them yourself!

1) **Binary Trees and Binary Search Trees are Different** – The word "search" is really important, there are many types of binary trees (and we'll actually cover several in class later in the semester), but the *search property* is what defines BSTs. Make sure you've got these two straight in your head.

2) **Special treatment for duplicates** – Since BSTs are often used to represent sets of comparable things, duplicate keys are not generally allowed. There are ways to make BSTs that allow duplicates, but you will not need to support duplicate keys for the current assignment. Follow the instructions in the provided template files for what to do if a duplicate key is offered to the BST. The instructions will tell you to replace the associated value if you are to inset a key more than once!

3) **Insertion Order Matters** – Inserting the same values into a BST in different orders will result in different trees. Examples below show what happens when inserting the values 1-4 in different orders:

Insertion Order:	1 2 3 4	4 3 2 1	1 4 2 3	2 3 1 4
Resulting Tree:				

This happens because the first value inserted becomes the root, the second value is placed in relation to the root (following the search property rules of BSTs), and so on. There is no correct tree for a given set of values, but there is a correct tree for a given set of values *and* a given insertion order. Deletion has a similar effect: the order in which you delete items will shape the resulting tree.

tl;dr You're doing Binary Search Trees for this project. You can learn about them from your textbook (ch. 19.1-19.3) and visualize them with gt.jar (on Piazza). Read the common mistakes section above after you think you've got it.

LookUpBST

`LookUpBSTBST` class is your implementation of a binary search tree. It needs to support the following methods:

- Operations supported by general binary trees and not relying on the search property: **`height()`**, **`numLeaves()`**, **`toString()`**. Look for guidance in Chapter 18 of your textbook.
- Operations utilizing the search property of binary search trees: **`contains()`**, get(), **`put()`**, **`findBiggestKey()`**. Look for guidance in Chapter 19 of your textbook.

Check the provided template file for the detailed requirements on the behavior, return, and big-O of those methods. The references listed above provide very helpful guidance as for which algorithms you can follow to implement them.

[WARNING] One last time… while the textbook, Gnarley Trees, and the videos can help you understand the general algorithm for various operations, don't mimic them exactly; follow the instructions!

How To Handle a Multi-Week Project

While this project is given to you to work on over several weeks, you are unlikely to be able to complete this is one weekend. It has been specifically designed such that the "stages" correspond with each of a set of lectures. We recommend the following schedule:

- Step 1 (**fileToQueue()**, and some of **CallStack** methods): Before the first weekend and first weekend (2/15-2/16)
  - Your class has finished talking about linked lists, maybe is still talking about stacks and queues, and will finish discussing stacks and queues before the weekend. You should have enough background on linked lists, stacks, and queues at this point to do the file I/O portion of this assignment (**fileToQueue()**) and some of the **CallStack** methods .
- Step 2 (**CallStack**, and 1/2 **LookUpBST** methods): Before the second weekend
  - At this point you've learned about stacks and queues in class, have some introduction to trees, and done your readings, so implement **CallStack** and try out some basic math processing. Attempt one or two **LookUpBST** methods.
- Step 3 (**LookUpBST** and Assignment Operators): Second weekend (2/22-2/23)
  - Your class has finished discussing trees at this point, so you should have enough to do binary search tree (**LookUpBST**) on your own if you're following your readings.

This schedule will leave you with an entire week + weekend to perform additional testing, get additional help, and otherwise handle the rest of life ☺ Also, notice that if you get it done early in the week, you can get extra credit! Otherwise you'll have plenty of time to test and debug your implementation before the due date.

tl;dr Don't try to do this project all at once. Above is a schedule you can use to keep on track.

Step 1: Supporting Program Input

Programs will be provided to the "SimpleCompiler" in text files. In the provided zip you will find a number of sample programs in a folder called **`prog`**. Once you complete this assignment you'll be able to run all these programs!

Your first step is to read in a program for the compiler to compile and evaluate from a file into a "queue" of nodes (the "input queue" in the above examples). You have been given a node class (**`Node`**) and do not need to edit this is any way.

However, if you open **`SimpleCompiler.java`** you will see a method called **`fileToQueue()`** which needs to be completed. Using the Java Scanner class, you'll need to turn an input text file into a **`Node`** queue of symbols that the SimpleCompiler can compile and evaluate (symbols are space separated; you may assume only one space between each symbol if that is helpful).

Once you have completed this stage, you should be able to run your program as follows and see the program displayed. The compiler can't evaluate the program yet, but this is a good start!

```
> java SimpleCompiler ../prog/sample1.txt false

Program: 1 2 + 3 4 - 5 6 + 4 2 / * * + print
```

If you don't remember how to do file I/O, please look over the Java review materials provided to you on Piazza and in the textbook, or go for a refresher in the TA/Prof office hours. Ask early! You don't want to be stuck on material from previous semesters and not be able to get to the cool stuff!

tl;dr Implement **`fileToQueue()`** using the provided Node class and test with the sample programs provided. The compiler won't work yet, that's Step 2.

Step 2: Supporting Simple Math

In order for the SimpleCompiler to work, we're going to need to have a stack as well as a queue. As discussed in class, technically all you need to make a linked list are node, and linked lists can be used as the basis of both stacks and queues, but it is nice to use Java's object oriented focus to build much nicer interfaces to our data structures. Therefore the **`CallStack`** you'll be writing will have nice encapsulation/abstraction/etc. for a stack rather than just the bare nodes.

If you look at **`CallStack.java`** you will see some standard methods for a stack (**`push`**, **`pop`**, **`isEmpty`**, etc.) as well as a few other useful methods. Completing this class will allow your SimpleCompiler to keep a Call stack.

> Note: You must make a *linked list* stack, you may not use a dynamic array or any other type of array for the **`CallStack`** class (except as a local variable to return in the **`toArray()`** method). You may not call the **`toArray()`** method to bypass this requirement either.

Once you've completed the **`CallStack`**, look back in SimpleCompiler at the **`compile()`** method. This method takes an input queue and processes a specified number of items from the queue. There is an instance of the **`CallStack`** class in SimpleCompiler (**`callStack`**) which you can use in your math processing. For this step, you need to be able to process: (1) numbers, (2) the math symbols +, -, *, and /, (3) the print command. Division is "integer division", so no decimals.

Once you are done with this section, you should be able to run **`sample1.txt`** in both normal and debug mode and see the *exact same* output as the example run shown later in this document. If you are not seeing the same output, for example if your stack or queue is printing backwards, fix this before continuing.

tl;dr Implement the **`CallStack`** class and the **`compile()`** method. You only need to support **`+`**, **`-`**, **`*`**, **`/`**, and **`print`**. Try running your program with **`sample1.txt`** to verify it works!

Step 3: Supporting Variables

When a SimpleCompiler looks up a variable, it needs to do it fast. Binary search trees offer a O(log n) lookup as long as the tree is not a degenerate one or not highly unbalanced. If you open **`LookUpBST.java`** you'll find the outline of a binary search tree (with typical methods like put, get, delete, etc.).

Once binary search tree data structure is completed, look back at the **`compile()`** method in SimpleCompiler and at the overview section of this document outlining how to process variables when you encounter them in the queue. An instance

of **`LookUpBST`** (symbols) has already been setup for you in the **`SimpleCompiler`** class, which you need to update when variables are assigned values.

When everything looks good, try **`sample2.txt`** (and create other sample files for yourself) and test the SimpleCompiler in both debug and non-debug modes.

tl;dr Implement the **`LookUpBST`** class and enhance the **`compile()`** method to support the = operator. Try running your program with **`sample2.txt`** to verify it works!

[Extra Credit, 5 pts] Step 4: Challenge

Support the additional operators defined in the SimpleCompiler's ASSIGN_OPS (**`+=`**, **`-=`**, **`*=`**, **`/=`**). The **`sample3.txt`** file should work after you do this, but you'll want to do a lot more testing than just that!

tl;dr Really? Ok… no extra credit for you :P

Requirements Summary

An overview of the requirements are listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes and fill the provided template files.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods.
- **Big-O** - Template files provided to you contain instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

Input Format/Samples

We provide an initial set of input files in **`prog`** folder of the project package. Feel free to write more for your own testing. You can assume the following with the postfix program you need to process:

- All inputs are well-structured postfix programs with no syntax errors;
- There will not be any division by zero;
- There is always a single space between different numbers, operators, and names in an input file;
- The postfix program only include integer numbers;
- You only need to support **`+`**, **`-`**, **`*`**, **`/`**, **`=`**, and **`print`**; all arithmetic operations are binary. You will need to support additionally **`+=`**, **`-=`**, **`*=`**, **`/=`** if you attempt for extra credit.

Testing

The main methods provided in the template files contain useful code to test your project as you work. You can use command like "**`java CallStack`**" to run the testing defined in **`main()`**. You could also edit **`main()`** to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

tl;dr You need to: write code, document it, meet the big-O reqs, and test your code.

Example Runs* (sample1.txt)

*Uses IN-ORDER traversal to convert the tree to string

```
> java SimpleCompiler ../prog/sample1.txt
false

Program: 1 2 + 3 4 - 5 6 + 4 2 / * * + print
-19



> java SimpleCompiler ../prog/sample1.txt true

Program: 1 2 + 3 4 - 5 6 + 4 2 / * * + print
```

```
######### Step 1 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
1
----------Program Remaining----
2 + 3 4 - 5 6 + 4 2 / * * + print
```

```
######### Step 2 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
1 2
----------Program Remaining----
+ 3 4 - 5 6 + 4 2 / * * + print


######### Step 3 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3
----------Program Remaining----
3 4 - 5 6 + 4 2 / * * + print


######### Step 4 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 3
----------Program Remaining----
4 - 5 6 + 4 2 / * * + print


######### Step 5 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 3 4
----------Program Remaining----
- 5 6 + 4 2 / * * + print


######### Step 6 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1
----------Program Remaining----
5 6 + 4 2 / * * + print


######### Step 7 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 5
----------Program Remaining----
6 + 4 2 / * * + print


######### Step 8 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 5 6
----------Program Remaining----
+ 4 2 / * * + print


######### Step 9 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 11
----------Program Remaining----
4 2 / * * + print


######### Step 10 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 11 4
----------Program Remaining----
2 / * * + print


######### Step 11 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 11 4 2
----------Program Remaining----
/ * * + print


######### Step 12 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 11 2
----------Program Remaining----
* * + print


######### Step 13 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -1 22
----------Program Remaining----
* + print
```

```
######### Step 14 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
3 -22
----------Program Remaining----
+ print


######### Step 15 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
-19
----------Program Remaining----
print


######### Step 16 ###############

----------Step Output----------
-19
----------Lookup BST---------

----------Call Stack--------
```

Example Runs (sample2.txt)

```
>java SimpleCompiler ../prog/sample2.txt false

Program: x 3 2 + = y 4 = x y + print
9

>java SimpleCompiler ../prog/sample2.txt true

Program: x 3 2 + = y 4 = x y + print

######### Step 1 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
x
----------Program Remaining----
3 2 + = y 4 = x y + print


######### Step 2 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
x 3
----------Program Remaining----
2 + = y 4 = x y + print


######### Step 3 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
x 3 2
----------Program Remaining----
+ = y 4 = x y + print


######### Step 4 ###############

----------Step Output----------
----------Lookup BST---------

----------Call Stack--------
x 5
----------Program Remaining----
= y 4 = x y + print


######### Step 5 ###############

----------Step Output----------
----------Lookup BST---------
x:5
----------Call Stack--------

----------Program Remaining----
y 4 = x y + print


######### Step 6 ###############

----------Step Output----------
----------Lookup BST---------
x:5
----------Call Stack--------
y
----------Program Remaining----
4 = x y + print


######### Step 7 ###############

----------Step Output----------
----------Lookup BST---------
x:5
----------Call Stack--------
y 4
----------Program Remaining----
= x y + print

######### Step 8 ###############

----------Step Output----------
----------Lookup BST---------
x:5 y:4
----------Call Stack--------

----------Program Remaining----
x y + print

######### Step 9 ###############

----------Step Output----------
----------Lookup BST---------
x:5 y:4
----------Call Stack--------
x
----------Program Remaining----
y + print


######### Step 10 ###############

----------Step Output----------
----------Lookup BST---------
x:5 y:4
----------Call Stack--------
x y
----------Program Remaining----
+ print


######### Step 11 ###############

----------Step Output----------
----------Lookup BST---------
x:5 y:4
----------Call Stack--------
9
----------Program Remaining----
Print
```

```
######### Step 12 ###############

----------Step Output----------
9
----------Lookup BST---------
x:5 y:4
----------Call Stack--------
```