Programming Project 3 (50 points)
Due: Thursday, February 28 by midnight

Background:

Most arithmetic expressions we see are written in infix notation, which means they are of the form:

number operator number

For example, 6 + 3 is in infix notation. Notice that the operator appears IN-BETWEEN the two numbers it's applied to. However, there are other ways to write arithmetic expressions. One way is postfix notation, which uses the form:

number number operator

In this notation, the operator appears AFTER (post) the two numbers it's applied to. For example, 6 + 3 becomes 6 3 + in postfix notation.

Another way of writing expressions is prefix notation, which uses the form:

operator number number

Here, the operator comes BEFORE (pre) the two numbers it's applied to. 6 + 3 becomes + 6 3 in prefix notation.

Converting to postfix notation

It's fairly simple to move between infix notation and postfix notation:

1) Find the operator that's applied LAST
2) Find the two expressions that the operator is applied to (expr1 and expr2)
3) The postfix expression will be expr1 expr2 operator
4) Go back to step 1 to divide up both expr1 and expr2

For example, suppose I want to convert the following expression to postfix:

6 * 7 – 4 + (4 / 2 - 3)

First, I identify the operator that's applied last – the +. The two expressions that the + is applied to are 6*7-4 and (4/2-3). So, our final postfix expression will be

(postfix of 6*7-4) (postfix of (4/2-3)) +

Now, let's break apart 6*7-4. The operator applied last is the -, and it's applied to 6*7 and 4. 6*7 is 6 7 * in postfix, so the overall postfix expression is 6 7 * 4 -.

Next, let's break apart 4/2-3. The operator applied last is the -, and it's applied to 4/2 and 3. 4/2 is 4 2 / in postfix, so the overall postfix expression is 4 2 / 3 -.

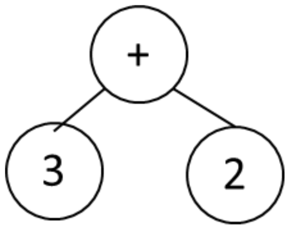
This tells us that 6 * 7 – 4 + (4 / 2 - 3) is:

6 7 * 4 - 4 2 / 3 - + //**postfix of 6*7-4 postfix of 4/2-3**

in postfix.

Notice that our postfix equivalent did not require any parentheses. In fact, no postfix expression needs parentheses.

Creating an Expression Tree
We can also represent arithmetic expressions using an *expression tree*. Here, the root of the tree is an operator, and its two children are the two things being operated on. For example, 3 + 2 becomes:



In general, to create an expression tree from a postfix expression, you will read the postfix expression one item at a time:

1) If the current item is a number, push it onto a stack
2) If the current item is an operator, pop the top two items from the stack. Create the expression tree with the operator as the root, the right subtree as the first item you popped, and the left subtree as the second item you popped. Push that new expression tree onto the stack.

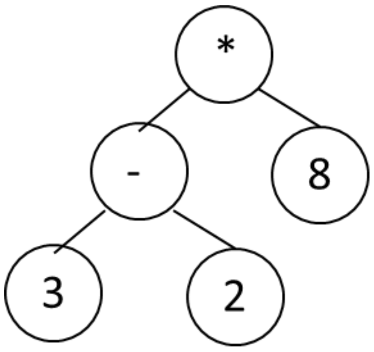
When you finish processing the postfix expression, you will have one item left on the stack – the final expression tree. We will discuss expression trees more during class.

Converting from Postfix to Infix Notation:
To convert from postfix to infix, you first use the algorithm above to create an expression tree. Then you perform an inorder traversal on the tree, which will print the expression in infix notation.

Recall that an inorder traversal recursively visits:
Left
Root

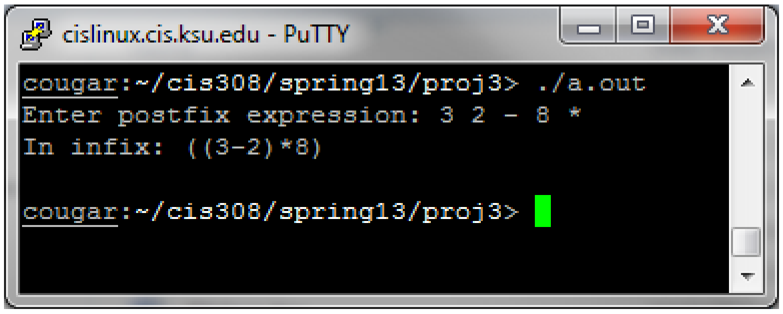
Right

On a tree. For example, the postfix expression 3 2 – 8 * would yield the expression tree:



And an inorder traversal on that tree would give: ((3-2)*8), which is the infix equivalent.

Assignment Description:
You are to write a program in C that asks the user for a postfix expression and uses the algorithm above to first create an expression tree for it and then print the equivalent infix expression by doing an inorder traversal on that expression tree. Here is screenshot of the program running on the example above:



```
cougar:~/cis308/spring13/proj3> ./a.out
Enter postfix expression: 3 2 - 8 *
In infix: ((3-2)*8)

cougar:~/cis308/spring13/proj3>
```

Implementation Requirements:
Your assignment must meet the following requirements:

- Your program should have two structs: one for an expression tree, and one for a node in the stack.
- Your program should contain at least the following separate functions: push (adding an item to your stack), pop (removing an item from your stack), inorder (prints the expression tree using an inorder traversal), freeTree (releases the memory in the expression tree), and the main function
- The inorder and freeTree functions must be recursive
- Your program should implement the algorithm in the background section for first creating an expression tree using a stack and then performing an inorder traversal on that tree.

- You may assume that there will be spaces between each operand (number) and each operator in the user input
- Your program should handle operands with multiple digits
- You should release all allocated memory when the program ends

We will start the project as a class during lab.

Documentation:
Your program must include a comment block at the top of every file, as well as at the top of each function. The function comments should include a brief description of what the function does, and explain any function arguments and return values. You may use the comment block below as a template:

```
/**********************************************
* Name: (YOUR NAME)                           *
* Date: (THE DUE DATE)                        *
* Assignment: Project 3: Postfix to Infix     *
***********************************************
* (WRITE A DESCRIPTION OF THE PROGRAM)        *
***********************************************/
```

Submission:
First, you will need to create a zip file of your project. To create a zip file in Unix, put all your code for this project (probably just one .c file) in a directory called “proj3”. Change directories to one back from the proj1 directory. To create a zip file of your project called “proj3.zip”:

```
zip proj3.zip proj3/*
```

It should list all the files that it included in the zip file.

To create a zip file in Windows, again put all your code for the project in a directory called “proj3”. Then, right-click on the proj3 folder and right-click, select “Send To”, and then select “Compressed (zipped) file”. This will create a zip file with your code called proj3.zip.

To submit your project, find the `proj3.zip` file that was created above. Then, go to “`Files and Content->Modules->File Dropbox`” on K-State Online, and upload the `proj3.zip` file. **Put your name and Project 3 in the description box.**

Grading:

Programs that do not compile will receive a grade of 0. A grading breakdown for programs that do compile appears below:

Correctly parses input postfix expression (assumes spaces, works with multiple digits) and builds expression tree	15
Format of project (struct for stack and tree, push/pop/inorder/free/main functions, inorder/free are recursive)	8

Infix expression conversion is correct, and is implemented using the algorithm in the background section	20
Each node has memory allocated with malloc and released with free (for both the stack and the tree)	5
Documentation and submission	2
Total	**50**