Notice: No lab assignments for the weeks of Apr. 7-11 and 14-18. (The current lab 6 assignment on PostScript is due Apr. 14.)

The first objective of this project is to develop a simple binary tree linked node structure, C/C++ functions to help build expression trees out of these nodes and to traverse them, and to demonstrate the use of these functions.

Here is an alternative, object oriented description: Develop an \texttt{TNode} class for node-of-an-expression-tree that uses the linked node implementation. This class has member functions for building a node tree from its top level operator or constant and possibly addresses of the roots of its subtrees. The \texttt{TNode} class should have member functions to traverse the expression tree whose root node is the object for which each member function is called. The project must demonstrate the use of these member functions.

It is your choice as to whether or not to use a class with constructors or other member functions, or just a plain struct. You must make use a struct or class that contains two pointer-to-tree-node fields or members, and some data.

The second objective is to gain deeper understanding between expression parsing using operand and operator stacks, and (1) building the expression tree, (2) traversing the expression tree, (3) evaluating the expression tree, (4) printing the expression tree by traversing it several ways. This objective is supposed to reveal the relationship between post-order traversal, postfix notation and evaluation.

**Getting Started, etc.**

Read the entire assignment and plan how you will do the project! The relevant chapters in Main and Savitch for this project are: 5 on dynamically allocated linked data structures, 7 on stacks and expressions, 9 on recursion and 10 (pages 453-459, 462-467, 479-485 and the self-test on page 491) on trees.

All requirements from previous projects are in force: Using RCS, submission of a directory including a build script, the RCS database files, and no object, linked executable or core files. Submit your work to project \texttt{proj5}.

Program organization, style, etc., plus the requirements for Part 1 will count for 50 points. The remaining 50 points count for the requirements for Part 2. At least Part 1 must compile and link for you to get any points at all.

A 40-50 score here indicates the minimum for passing below C. I think the C level will remain around 60% or so.
You may use the following declarations to get started on Part 1:

```c
struct TNode
{
    //enum {CONST, OPERATION} NType;  //optional, see page 678.
    char symbol;    //optional: maybe modify to support multi-char constants.
    //double value;  //optional: value==the constant’s value if NType==CONST
    //    //optional: otherwise, symbol==’+’, ’-’, ’*’, or ’/’,
    //    //optional: and NType==CONST.
    TNode *pleft;
    TNode *pright;
};
TNode * newTNode( char ch );  //Dynamically allocates one TNode and returns
                                //its address.
void InOrderTraverse( TNode *pnode, int depth )
{
    if(pnode->pleft) { InOrderTraverse( pnode->pleft, depth+1); }
    for(int i=0; i<depth; i++) {cout << ' ';} //depth spaces were printed.
    cout << pnode->symbol << endl ; //This node’s symbol was printed.
    //you finish it!!
}

//optional: class TNode
//{{
    // private: TNode *pleft; TNode *pright; char symbol;  //etc.
    // public:
    // TNode( char ch );
    // TNode( char ch, TNode *pleftparameter, TNode *prightparameter);
    // void InOrderTraverse( int depth ){
    //    //recursive calls within the body look like:
    //    //pleft->InOrderTraverse( depth + 1 );  // etc...
    // }
//}};
```

1 Part 1

Create declarations and functions described under the objectives to make a program that builds a binary tree so that each node contains one C/C++ char data item. The tree can be build by a sequence of function calls you write. You can code into those calls the particular letters, spaces and punctuation symbols required below. It would be a good idea to use some pointer-to-node temporary variables to hold the addresses of nodes right after they are build. Building the tree need not be done recursively.

Requirements: Build your tree so that an in-order left-to-right traversal prints out your full name; but, if your name has fewer than 15 characters including spaces, you must add something else (like “Sir”, “Madame”, “Albany, N. Y.”, etc.,) to make it at least 15 characters. The tree must
be built so that about half the characters are in the left subtree. (Something like this is coded on page 477.)

A more interesting option, for 5 points of extra credit, is to prompt for any string (which could be limited in length) and then demonstrate that your program builds the tree specified above automatically from a name supplied at run time.

After the tree is built, the program must demonstrate your recursive function to count the nodes and 3 functions to traverse the tree in left-to-right pre-order, in-order and post-order. The in-order output, read line by line, should spell out your name, etc., of course. Each character must be printed on its own line and be indented by the number of spaces equal to the recursion depth of the activation that prints it (see the sample code for InOrderTraverse()).

1.1 Credit

Submit a build script and sources so that when we execute the build script in its own directory, it builds a linked executable file named part1 which we can run to test it. (We will also examine program sources to grade for file structure, pre- and postconditions, style and check for cheating. A fake “solution” whose output makes it appear to meet the requirements but which doesn’t build and traverse a tree is a form of cheating.)

2 Part 2

Part 2 is a continuation of Project 4.

Add to your Project 4 infix expression evaluator code the functionality to build the expression tree while your infix expression evaluator is active. Up to the That evaluates to .. line it should input and print exactly what is specified for Project 4. Your program should also save the evaluated number in a variable so your program can automatically compare this number with the number obtained from recursive evaluation of the expression tree. See below.

The expression tree will be built “from the bottom up,” not using any recursive procedures, during a computation that is controlled by your stacks and the input expression.

2.1 Main Idea, Hint...

Remember the idea of putting in each number stack entry information about were the sub-expression whose value is that number, besides the number?

The idea is: Whenever a number is determined either from a constant digit, or by doing an operation, the expression tree node holding that constant or the operation symbol is allocated and filled in with data and pointers. The address of that tree node is (alongside the string index and the evaluated or converted number) stored in the number stack entry! Your code should use the node addresses retrieved from entries popped from the number stack to fill in the pleft and pright pointer fields of that new tree node. (Remember a way to convert the ASCII code ch for a digit 0, ... 9 to the corresponding integer? It is: (ch - ’0’).)

This idea will be dramatized in the lecture.
2.2 Continuing with Details..

After printing the value in “That evaluates to ..” your program should print:

Count of expression tree nodes is ...

where ... denotes how many nodes (for both constants and operations) there are in the expression tree your program built, as calculated by a recursive node counting traversal.

Next, print 4 more results, each preceded by the indicated label:

- **Pre-order Traversal:**
  Print each node’s operator or constant, one node per line, indented one space for each unit of recursion depth (see the “getting started” examples), as the tree is traversed in left-to-right pre-order.

- **In-order Traversal:**
  Analogous the above, except for the order.

- **Post-order Traversal and Evaluation:**
  Analogous again, except for the order, and, after each operator or constant, on the same line, print a space and then **value=...** to display the double precision floating point value obtained when that node was evaluated.

- The program should compare the number just printed with the number saved (see above) from your project 4 evaluation algorithm to check that they are equal. If they are equal, print “Values computed by the two algorithms match: Good.”

  Otherwise, print the copy of the stack algorithm’s answer again in a message that this program has a bug.

Finally, on a new line print the prompt

**Type a fully parenthesized arithmetic expression:**

Then accept and process new expressions, until a blank line is entered to terminate the run.

2.3 Credit

To get credit for Part 2, make your build script create the linked executable file named **part2** so we know you did it and can test it. (Submit to project **proj5**.)

We will not grade it on inputs with variables or multi-char constants. If you implemented those in Project 4, it might be easier to reuse your Project 4 work if you support some of these features in Project 5.

3 Test Yourself with Lab 6

Use your Part 2 program to evaluate some infix expressions and obtain their equivalent postfix expressions. Then, translate each postfix expression into PostScript and append the PostScript “pop-and-print” operator **==**. Verify that when you input each PostScript expression to the Ghostscript interpreter (**gs**), Ghostscript computes the same value as your Part 2 program did.