Expression Tree ADT

objectives

In this laboratory you will:

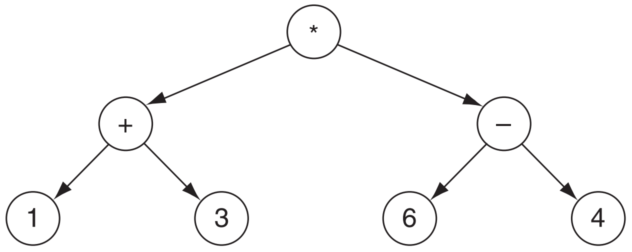
- Create an implementation of the Expression Tree ADT using a linked tree structure
- Develop an implementation of the Logic Expression Tree ADT and use your implementation to model a simple logic circuit
- Create an expression tree copy constructor
- Analyze how preorder, inorder, and postorder tree traversals are used in your implementation of the Expression Tree ADT

Overview

Although you ordinarily write arithmetic expressions in linear form, you treat them as hierarchical entities when you evaluate them. When evaluating the following arithmetic expression, for example,

```
(1+3)*(6-4)
```

you first add 1 and 3, then you subtract 4 from 6. Finally, you multiply these intermediate results together to produce the value of the expression. In performing these calculations, you have implicitly formed a hierarchy in which the multiply operator is built on a foundation consisting of the addition and subtraction operators. You can represent this hierarchy explicitly using the following binary tree. Trees such as this one are referred to as **expression trees**.



Expression Tree ADT

Data Items

Each node in an expression tree contains either an arithmetic operator or a numeric value.

Structure

The nodes form a tree in which each node containing an arithmetic operator has a pair of children. Each child is the root node of a subtree that represents one of the operator's operands. Nodes containing numeric values have no children.

Operations

`ExprTree ()`

Requirements:
None

Results:
Constructor. Creates an empty expression tree.

```
~ExprTree ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store an expression tree.

```
void build () throw ( bad_alloc )
```

Requirements:
None

Results:
Reads an arithmetic expression in prefix form from the keyboard and builds the corresponding expression tree.

```
void expression () const
```

Requirements:
None

Results:
Outputs the corresponding arithmetic expression in fully parenthesized infix form.

```
float evaluate () const throw ( logic_error )
```

Requirements:
Expression tree is not empty.

Results:
Returns the value of the corresponding arithmetic expression.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in an expression tree.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs an expression tree with its branches oriented from left (root) to right (leaves)—that is, the tree is output rotated counterclockwise 90 degrees from its conventional orientation. If the tree is empty, outputs “Empty tree”. Note that this operation is intended for testing/debugging purposes only. It assumes that arithmetic expressions contain only single-digit, nonnegative integers and the arithmetic operators for addition, subtraction, multiplication, and division.

We commonly write arithmetic expressions in **infix form**—that is, with each operator placed between its operands, as in the following expression:

```
( 1 + 3 ) * ( 6 – 4 )
```

In this laboratory, you construct an expression tree from the **prefix form** of an arithmetic expression. In prefix form, each operator is placed immediately before its operands. The expression above is written in prefix form as

```
* + 1 3 – 6 4
```

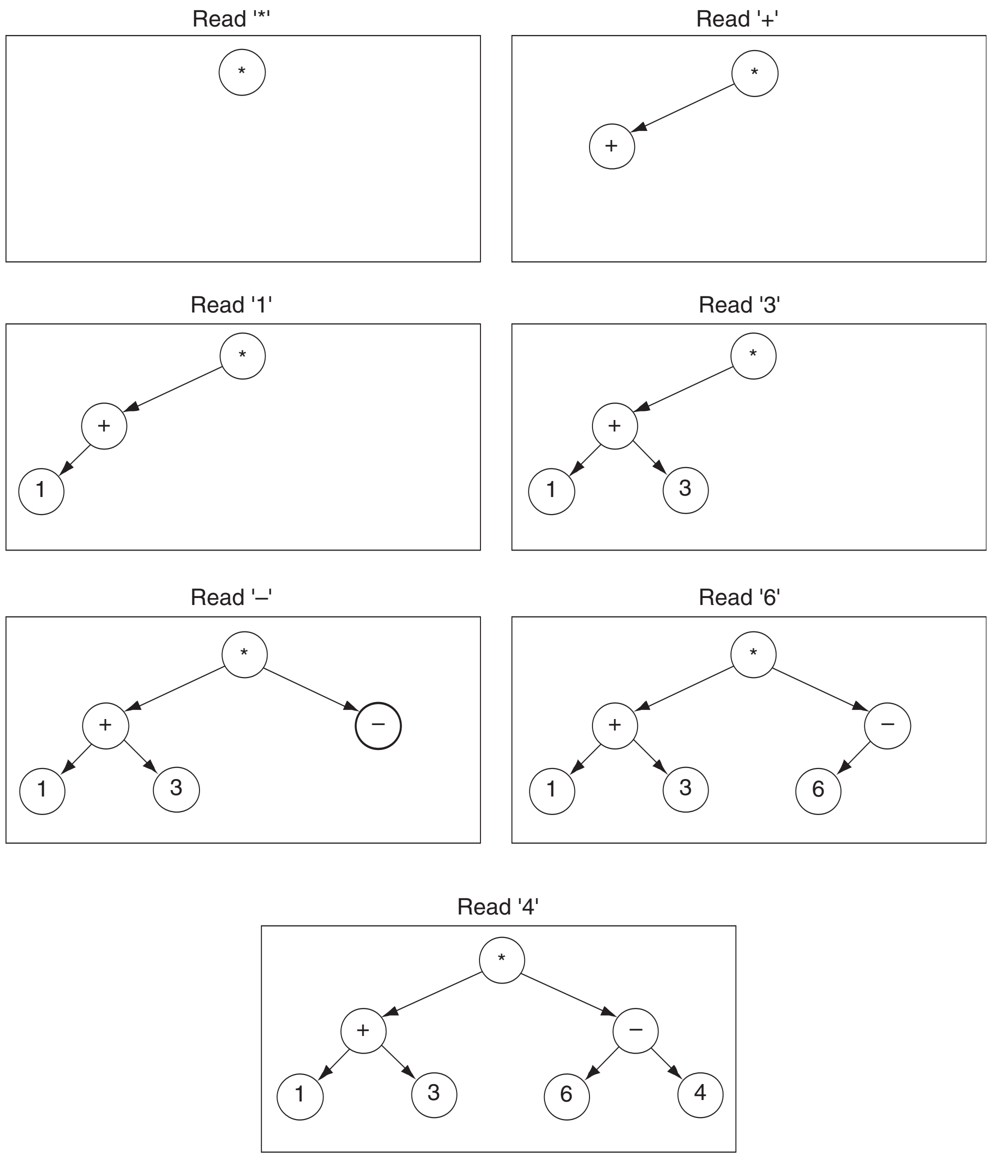
When processing the prefix form of an arithmetic expression from left to right, you will, by definition, encounter each operator followed by its operands. If you know in advance the number of operands an operator has, you can use the following recursive process to construct the corresponding expression tree.

```
Read the next arithmetic operator or numeric value.
Create a node containing the operator or numeric value.
if the node contains an operator
   then Recursively build the subtrees that correspond to the
      operator's operands.
   else The node is a leaf node.
```

If you apply this process to the arithmetic expression

```
* + 1 3 – 6 4
```

then construction of the corresponding expression tree proceeds as follows:



Note that in processing this arithmetic expression we have assumed that all numeric values are single-digit, nonnegative integers, and thus, that all numeric values can be represented as a single character. If we were to generalize this process to include multidigit numbers, we would have to include delimiters in the expression to separate numbers.

Laboratory 12: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 12: Prelab Exercise

Name ______________________________	Date ________________
Section ____________________________	

In the Overview you saw how the construction of an expression tree can be described using recursion. In this exercise you will use recursive functions to implement the operations in the Expression Tree ADT.

Step 1: Implement the operations in Expression Tree ADT using a linked tree structure. Assume that an arithmetic expression consists of single-digit, nonnegative integers (‘0’..‘9’) and the four basic arithmetic operators (‘+’, ‘-’, ‘*’, and ‘/’). Further assume that each arithmetic expression is input in prefix form from the keyboard with all of the characters on one line.

As with the linear linked structures you developed in prior laboratories, your implementation of the linked tree structure uses a pair of classes: one for the nodes in the tree (ExprTreeNode) and one for the overall tree structure (ExprTree). Each node in the tree should contain a character (`dataItem`) and a pair of pointers to the node's children (`left` and `right`). Your implementation also should maintain a pointer to the tree's root node (`root`). Base your implementation on the following declarations from the file *exprtree.hs*. An implementation of the `showStructure` operation is given in the file *show12.cpp*.

```
class ExprTree;            // Forward declaration of the ExprTree class

class ExprTreeNode         // Facilitator class for the ExprTree class
{
  private:

    // Constructor
    ExprTreeNode ( char elem,
                   ExprTreeNode *leftPtr, ExprTreeNode *rightPtr );

    // Data members
    char dataItem;           // Expression tree data item
    ExprTreeNode *left,      // Pointer to the left child
                 *right;     // Pointer to the right child

  friend class ExprTree;
};
```

```
//--------------------------------------------------------------------

class ExprTree
{
  public:

    // Constructor
    ExprTree ();

    // Destructor
    ~ExprTree ();

    // Expression tree manipulation operations
    void build ()               // Build tree from prefix expression
        throw ( bad_alloc );
    void expression () const;   // Output expression in infix form
    float evaluate () const     // Evaluate expression
        throw ( logic_error );
    void clear ();              // Clear tree

    // Output the tree structure -- used in testing/debugging
    void showStructure () const;

  private:

    // Recursive partners of the public member functions -- insert
    // prototypes of these functions here.
    void showSub ( ExprTreeNode *p, int level ) const;

    // Data member
    ExprTreeNode *root;   // Pointer to the root node
};
```

Step 2: The declaration of the ExprTree class in the file *exprtree.hs* does not include prototypes for the recursive private member functions needed by your implementation of the Expression Tree ADT. Add these prototypes and save the resulting class declarations in the file *exprtree.h*.

Step 3: Save your implementation of the Expression Tree ADT in the file *exprtree.cpp*. Be sure to document your code.

Laboratory 12: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

Test your implementation of the Expression Tree ADT using the test program in the file *test12.cpp*.

Step 1: Compile your implementation of the Expression Tree ADT in the file *exprtree.cpp*.

Step 2: Compile the test program in the file *test12.cpp*.

Step 3: Link the object files produced by Steps 1 and 2.

Step 4: Complete the following test plan by filling in the expected result for each arithmetic expression. You may wish to add arithmetic expressions to the test plan.

Step 5: Execute this test plan. If you discover mistakes in your implementation of the Expression Tree ADT, correct them and execute the test plan again.

Test Plan for the Operations in the Expression Tree ADT

Test Case	*Arithmetic Expression*	*Expected Result*	*Checked*
One operator	`+34`		
Nested operators	`*+34/52`		
All operators at start	`-/*9321`		
Uneven nesting	`*4+6-75`		
Zero dividend	`/02`		
Single-digit number	`7`		

Laboratory 12: In-lab Exercise 1

Name ______________________ Date ______________

Section ______________________

Computers are composed of logic circuits that take a set of Boolean input values and produce a Boolean output. You can represent this mapping from inputs to output with a logic expression consisting of the Boolean logic operators AND, OR, and NOT (defined below) and the Boolean values True (1) and False (0).

	(NOT)			(AND)	(OR)
A	-A	A	B	A*B	A+B
0	1	0	0	0	0
1	0	0	1	0	1
		1	0	0	1
		1	1	1	1

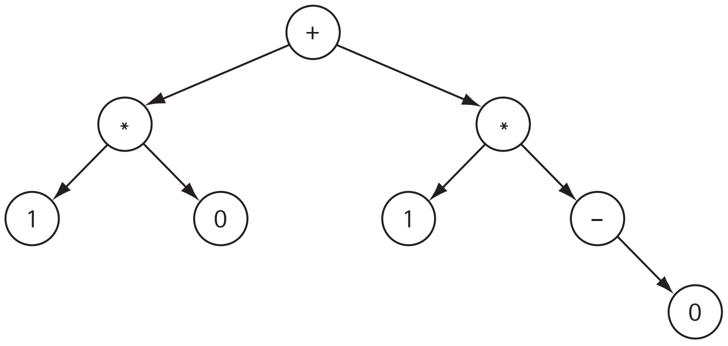
Just as you can construct an arithmetic expression tree from an arithmetic expression, you can construct a logic expression tree from a logic expression. For example, the following logic expression:

```
(1*0)+(1*-0)
```

can be expressed in prefix form as

```
+*10*1-0
```

Applying the expression tree construction process described in the overview to this expression produces the following logic expression tree.



Evaluating this tree yields the Boolean value True (1).

Construction of this tree requires processing a unary operator, the Boolean operator NOT (‘`-`’). When building a logic expression tree, we will choose to set the right child of any node containing the NOT operator to point to the operand and set the left child to null. Note that you must be careful when performing the remaining operations to avoid traversing these null left children.

Step 1: Modify the prototype of the `evaluate()` function in the file *exprtree.h* so that this function yields an integer value rather than a floating-point number. You may also need to modify the prototype of a related recursive private member function. Save the resulting class declarations in the file *logitree.h*.

Step 2: Create an implementation of the Expression Tree ADT that supports logic expressions consisting of the Boolean values True and False (‘`1`’ and ‘`0`’) and the Boolean operators AND, OR, and NOT (‘`*`’, ‘`+`’, and ‘`-`’). Base your implementation on the declarations in the file *logitree.h*. Save your implementation of the Logic Expression Tree ADT in the file *logitree.cpp*.

Step 3: Modify the test program in the file *test12.cpp* so that the header file for the Logic Expression Tree ADT (*logitree.h*) is included in place of the header file for the (arithmetic) Expression Tree ADT.

Step 4: Compile and link your implementation of the Logic Expression Tree ADT and the modified test program.

Step 5: Complete the following test plan by filling in the expected result for each logic expression. You may wish to include additional logic expressions in this test plan.

Step 6: Execute this test plan. If you discover mistakes in your implementation of the Logic Expression Tree ADT, correct them and execute the test plan again.

Test Plan for the Operations in the Logic Expression Tree ADT

Test Case	*Logic Expression*	*Expected Result*	*Checked*
One operator	`+10`		
Nested operators	`*+10+01`		
NOT (Boolean value)	`+*10*1-0`		
NOT (subexpression)	`+-1-*11`		
NOT (nested expression)	`-*+110`		
Double negation	`--1`		
Boolean value	`1`		

Having produced a tool that constructs and evaluates logic expression trees, you can use this tool to investigate the use of logic circuits to perform binary arithmetic. Suppose you have two one-bit binary numbers (X and Y). You can add these numbers together to produce a one-bit sum (S) and a one-bit carry (C). The results of one-bit binary addition for all combinations of X and Y are tabulated below.

```
    X
+   Y
-----
C   S
```

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

A brief analysis of this table reveals that you can compute the values of outputs S and C from inputs X and Y using the following pair of (prefix) logic expressions.

```
C = *XY       S = +*X-Y*-XY
```

Step 7: Using your implementation of the Logic Expression Tree ADT and the modified test program, confirm that these expressions are correct by completing the following table.

X	Y	C = *XY	S = +*X-Y*-XY
0	0	*00 =	+*0-0*-00 =
0	1	*01 =	+*0-1*-01 =
1	0	*10 =	+*1-0*-10 =
1	1	*11 =	+*1-1*-11 =