Expression Tree Internal Assessment

Soumya Mahavadi

October 2020 — January 2021

Table of Contents

Criterion A: Planning

The client is my cousin, Ms. {Client}, who is also a fourth-grade teacher. Currently, {Client} is teaching her students about solving expressions with multiple operators. As a result, she has introduced the order of operations to her students to solve these expressions.

{Client}'s students have been inputting such expressions into their calculators to recognize how they are evaluated. While this is not an issue for simple expressions, like "7*2-3," as the expressions grow complex, they become difficult to enter. Intricate expressions often come with more sets of parentheses—parentheses that are extremely easy to misplace if not observed with a careful eye. {Client} has asserted that her students do encounter this problem, which results in them having to retype the entire expression again. This is extremely insufficient.

{Client} has consulted with me and asked if I would be able to design a computer program in order for her students to have an easier time inputting expressions. She specified that it would be straightforward for her fourth-graders to use, but also complex enough in order to evaluate expressions with basic operators as a normal calculator would.

After consulting with {Client}, I suggested that I code an expression tree, where students would input their expressions in prefix notation, and the program would convert it back to infix with the right parentheses and further evaluate it. This project will completely eliminate the problem of dealing with parentheses because none would have to be inputted at all—prefix notation is able to follow the order of operations *without* any sets of parentheses.

I chose to use C++ for this project due to my own familiarity as I have experience coding in the language for four years. C++ is also equipped with features like object-oriented programming and linked lists, which are necessary for this project. Moreover, C++ is compatible with multiple software, which increases the accessibility of the program for {Client} and her students.

Success Criteria

- Users will be presented with a menu option to manually enter an expression.
- The program will continue to run and present the user with the menu after each task until the user quits.
- The program can convert an expression from prefix to infix notation.
- The program can add, subtract, multiply, divide, and calculate exponents.
- The program can calculate multiple operators in one expression.
- The program can accurately evaluate the expression.
- The program can handle printing and evaluating errors if no expression is inputted.

Criterion B(1): Record of Tasks

Task Number	Planned Action	Planned Outcome	Time Estimate	Target Completion Date
1	Advisor discussion	Approval of initial ideas	½ hour	10/15/2020
2	Initial consultation with {Client}	{Client} explained problem and desired solution	1 hour	10/16/2020
3	Second consultation	Determine success criteria	½ hour	10/19/2020
4	Complete design of program	Create flowcharts of complex algorithms and design test plan	2 weeks	11/10/2020
5	Begin creation of menu function	Determine menu options	1 hour	11/12/2020
7	Complete creation of ENode class	Code the attributes, accessors, and mutators	1 week	11/20/2020
9	Complete create and print methods	Use recursion to construct expression tree and print in infix notation	1-2 weeks	12/01/2020
10	Complete evaluate and print answer methods	Ensure that program accurately evaluates expressions	1 week	12/08/2020
11	Finish menu function	Update switch statements by calling related methods	½ hour	12/09/2020
12	Third consultation	Review program functions; discuss improvements	1 hour	12/15/2020
13	Test the final product	Ensure program is accurate in prefix to infix conversions and evalutations	1 week	12/23/2020
14	Record demonstration video	Reveal product functionality	2 days	12/28/2020
15	Final consultation	Ensure that success criteria are satisfied and that product can be used by students	1 hour	01/04/2021

Criterion B(2): Design

Figure 1: Algorithm to Create Expression Tree

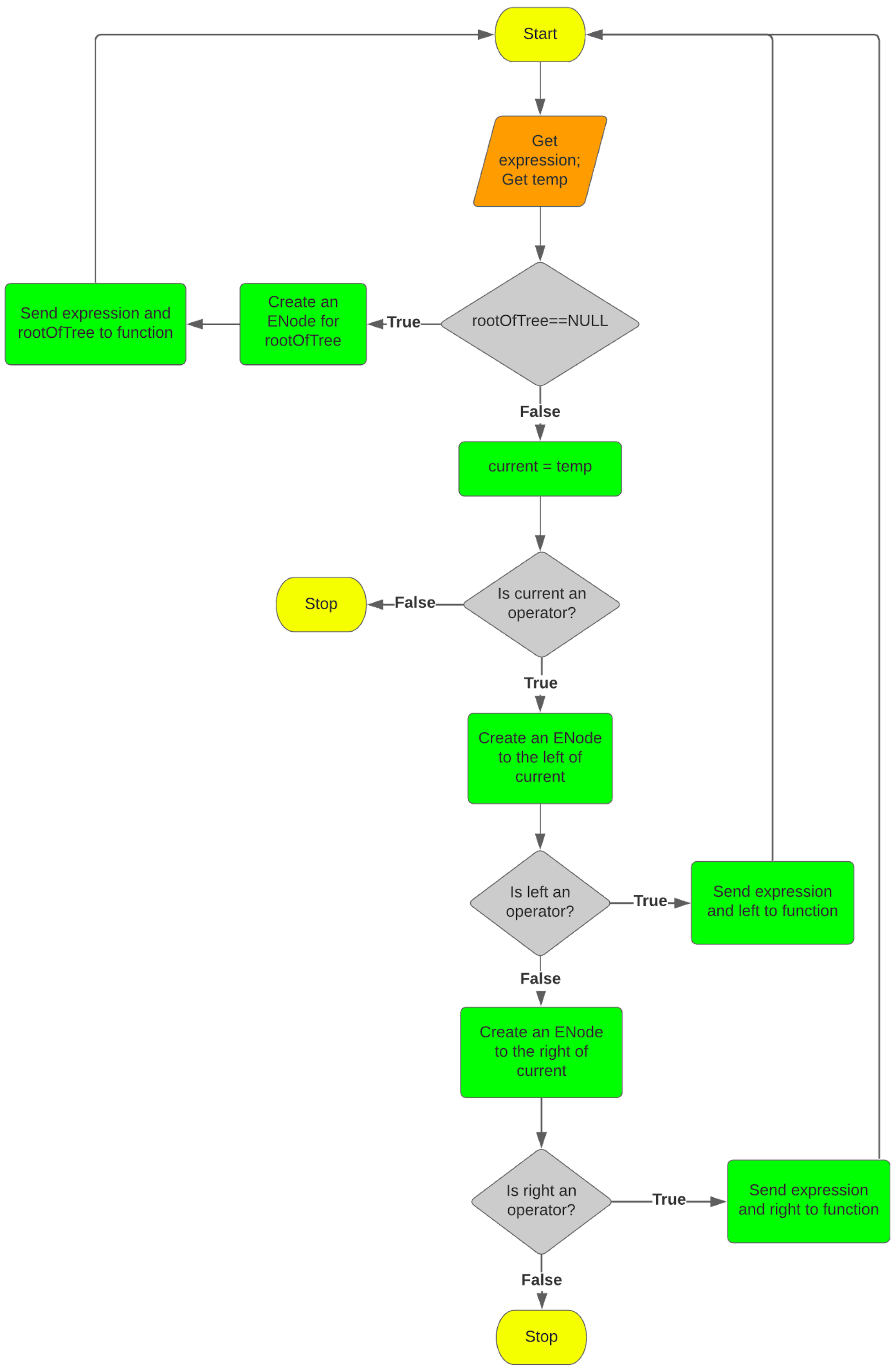

Start
Get expression; Get temp
rootOfTree==NULL
True
Create an ENode for rootOfTree
Send expression and rootOfTree to function
False
current = temp
Is current an operator?
False
Stop
True
Create an ENode to the left of current
Is left an operator?
True
Send expression and left to function
False
Create an ENode to the right of current
Is right an operator?
True
Send expression and right to function
False
Stop

Figure 2: Algorithm to Print

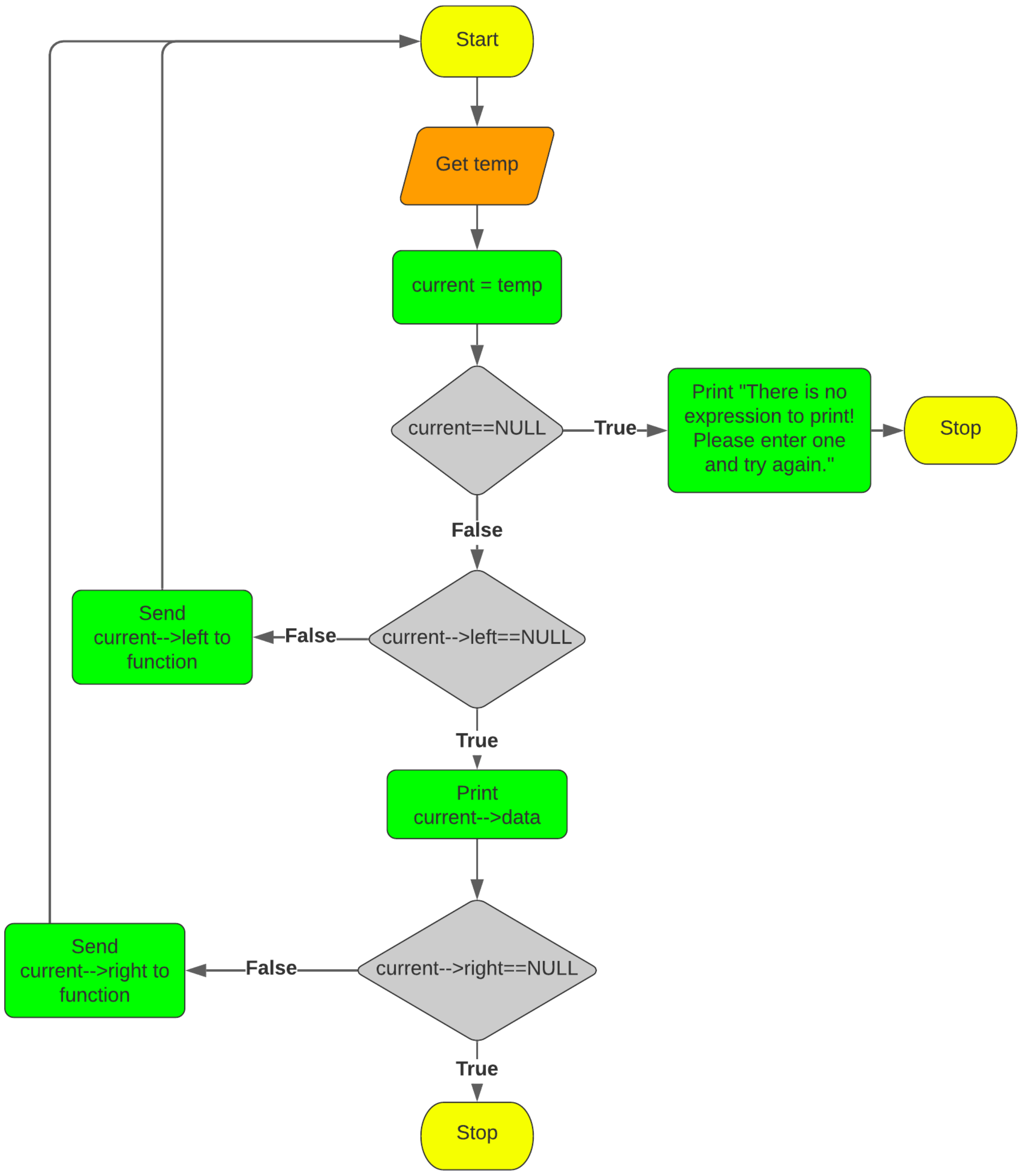
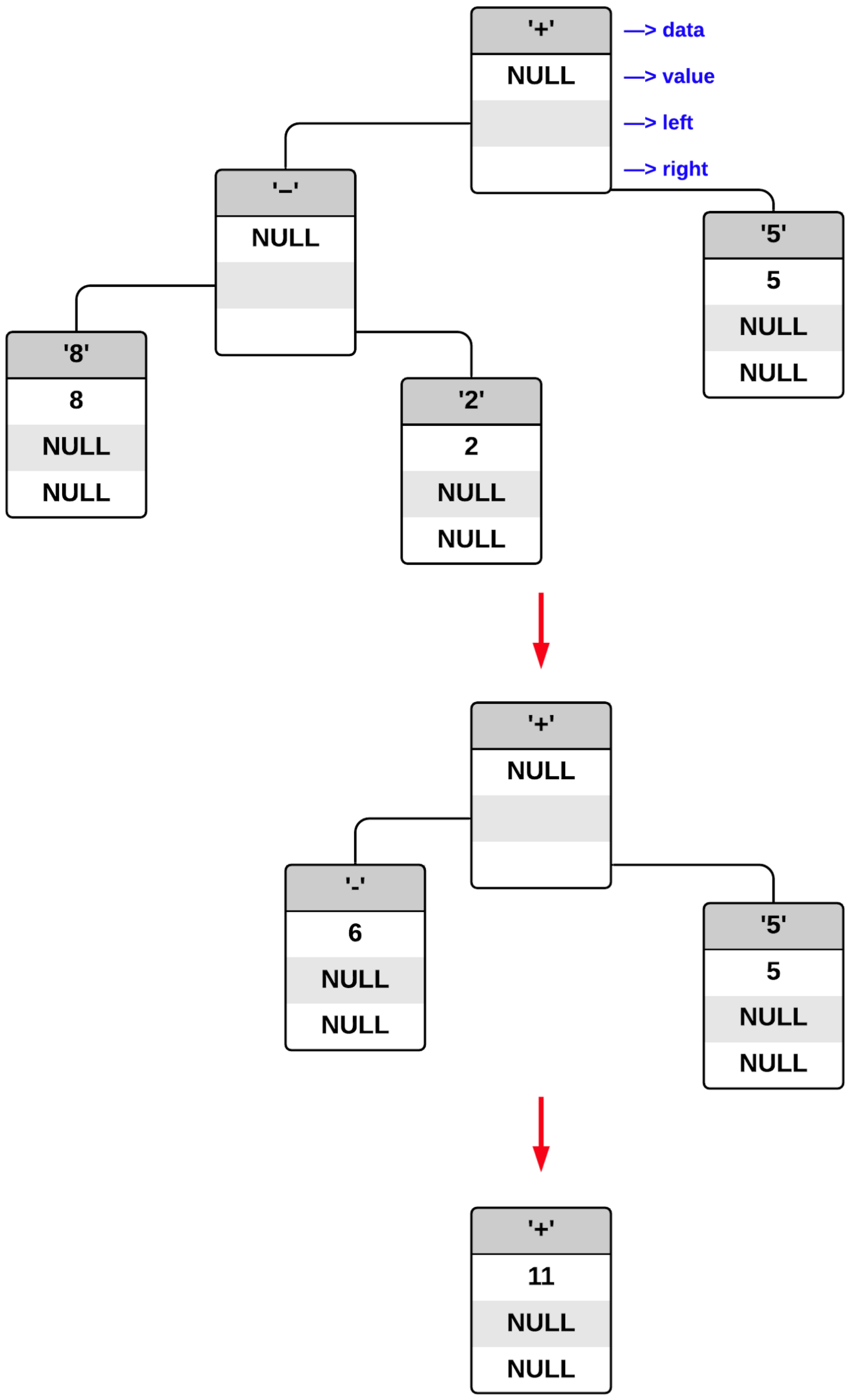


Figure 3: Visualization and Evaluation of Expression Tree

Example

Prefix: +-825

Infix: ((8 - 2) + 5)



<u>Summary of Methods</u>

#1 - Create a Menu

- Allows user to control functionality: insert, print, or evaluate expression

```
//PSEUDOCODE FOR MENU OPTIONS

If option 1
    request prefix expression
    get EXPRESSION
    ROOT = 0
    new ENode (EXPRESSION, ROOT)
Else If option 2
    print EXPRESSION in infix notation
Else If option 3
    print answer to EXPRESSION
Else If option 4
    quit
Else
    ask to choose a valid option
```

#2 - Create an Expression Tree

- Creates and links ENode objects to create the tree (constructs linked list)

```
//PSEUDOCODE FOR CONSTRUCTING LINKED LIST OF ENODES IN CREATE FUNCTION

If CURRENT is an operator
    declare and initialize an ENode called LEFT
    CURRENT->LEFT = LEFT

    If LEFT is an operator
        Create (EXPRESSION, LEFT)

    declare and initialize an ENode called RIGHT
    CURRENT->RIGHT = RIGHT

    If RIGHT is an operator
        Create (EXPRESSION, RIGHT)
```

(see Figure 1 for flowchart of full algorithm)

#3 - Print the Expression Tree

- Traverses through expression tree to print prefix expression in infix notation
    - Check left —> print —> check right
- Utilizes recursion

```
//PSEUDOCODE FOR PRINTING THE EXPRESSION TREE

Print (ENode)
    If CURRENT->LEFT is not NULL
        Print(CURRENT->LEFT)
    print CURRENT->DATA
    If CURRENT->RIGHT is not NULL
        Print(CURRENT->RIGHT)
```

(see Figure 2 for flowchart of full algorithm)

#4 - Evaluate the Expression Tree

- Traverses through tree to evaluate user-inputted expression
    - Check left —> check right —> evaluate
- Avoids evaluating expression if encounters divide by 0 error
- Utilizes recursion

```
//PSEUDOCODE FOR EVALUATING THE EXPRESSION TREE

Evaluate (ENode)
    If CURRENT->LEFT is an operator
        Evaluate (CURRENT->LEFT)
    If CURRENT->RIGHT is an operator
        Evaluate (CURRENT->RIGHT)

    If CURRENT == '*'
        CURRENT = CURRENT->LEFT * CURRENT->RIGHT
    Else If CURRENT == '+'
        CURRENT = CURRENT->LEFT + CURRENT->RIGHT
    Else If CURRENT == '-'
        CURRENT = CURRENT->LEFT - CURRENT->RIGHT
    Else If CURRENT == '/'
        If CURRENT->RIGHT == 0
            PLACE = 0
        Else
            CURRENT = CURRENT->LEFT / CURRENT->RIGHT
    Else If CURRENT == '^'
        CURRENT = CURRENT->LEFT ^ CURRENT->RIGHT
```

(see Figure 3 for visualization of expression tree and evaluation method)

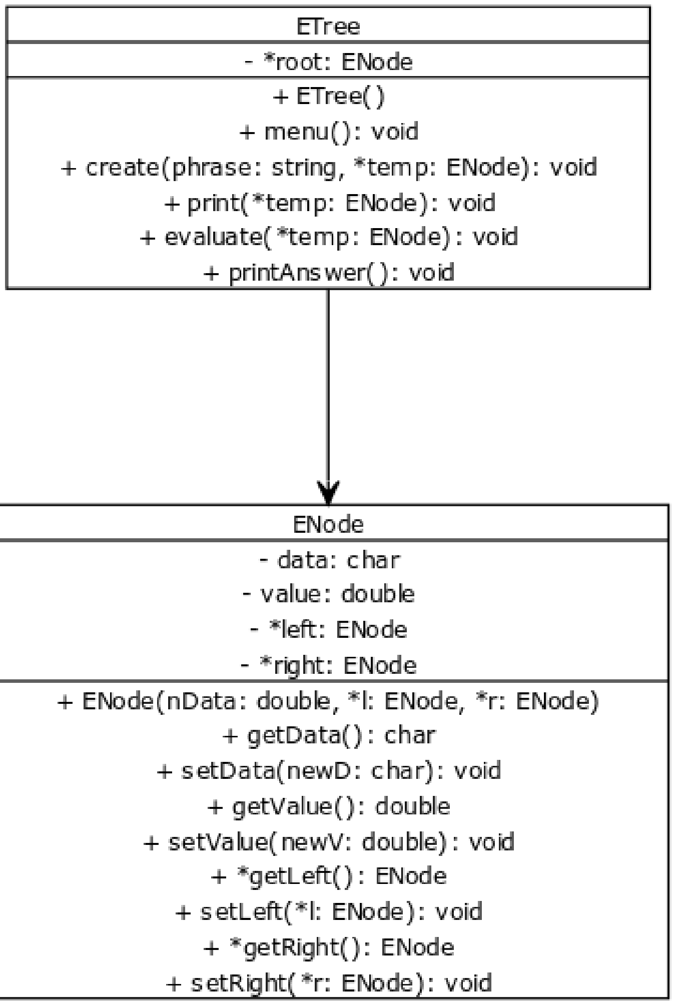
<u>Test Plan</u>

Test #	Test Type	Nature of Test	Example
1	When starting program, greeting should show up followed by menu	Check that menu is accurately displayed	*"Welcome…1) Insert a prefix expression..."*
2	Check that menu works	Menu should call code for option	User enters *"1"* *"Please enter an expression in prefix notation…"*
3	Check that program loops back to menu after completing task	Run option and check if program displays menu when done	*"Please enter an expression in prefix notation…"* User enters *"+-726"* *"Choose an option (enter a number):"*
4	Check that program quits if option '4' is selected	Input '4' as option	*"Choose an option (enter a number):"* User enters *"4"* *"[Process completed]"*
5	Check that program can handle errors if no entered expression	Upon entering program, choose option 2, then option 3	User enters *"2"* *"There is no expression to print. Please enter one and try again!"* User enters *"3"* *"No Expression Tree = No Answer. Please enter an expression first!"*
6	Check that program has ability to go back to menu if option '1' is inputted	Input "b" as prefix expression	*"Please enter an expression in prefix notation…"* User enters *"b"* *"Choose an option (enter a number):"*
7	Check that prefix to infix conversion works	Enter expression and print	User enters *"+-726"*

			“((7-2)+6)”
8	Check that program can calculate multiple operators and accurately evaluate expression	Enter complex expression and evaluate	User enters *“*/+231*+45–62”* *“The answer to the expression is: 180”*
9	Check that program can handle divide by 0 error	Enter undefined expression	User enters *“/70”* *“Sorry, you cannot divide by zero (undefined). Please enter a new (defined) expression.”*

Criterion C: Development

UML Diagram


ETree
- *root: ENode
+ ETree( )
+ menu(): void
+ create(phrase: string, *temp: ENode): void
+ print(*temp: ENode): void
+ evaluate(*temp: ENode) : void
+ printAnswer( ): void
ENode
- data: char
- value: double
- *left: ENode
- *right: ENode
+ ENode(nData: double, *l: ENode, *r: ENode)
+ getData( ) : char
+ setData(newD: char): void
+ getValue(): double
+ setValue(newV: double) : void
+ *getLeft( ) : ENode
+ setLeft(*l: ENode): void
+ *getRight(): ENode
+ setRight(*r: ENode): void

List of Techniques

A. Pointers

B. Console colored output

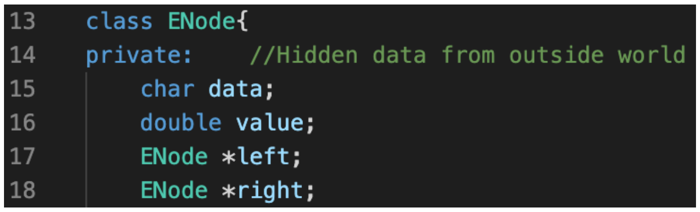
C. Switch case

D. Creating an ENode

E. Calling methods from other methods

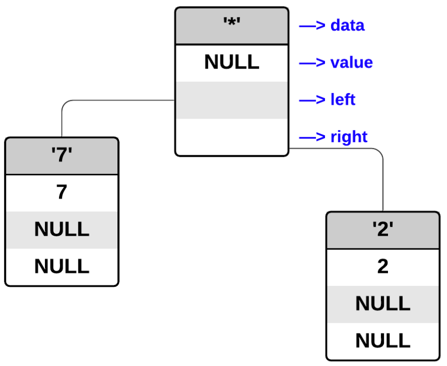
F. Try and catch errors

A. Pointers

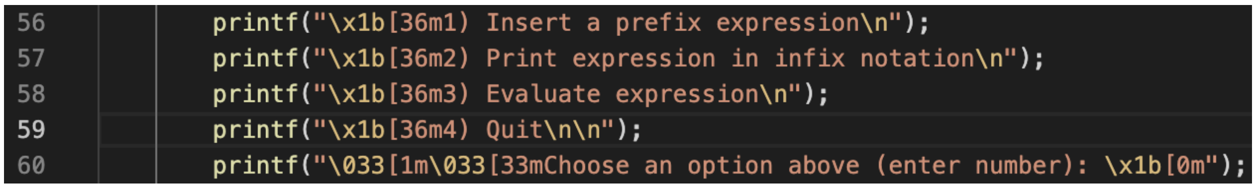


```
13  class ENode{
14  private:    //Hidden data from outside world
15      char data;
16      double value;
17      ENode *left;
18      ENode *right;
```

Lines 17 and 18 from the *ENode* class show the declaration of two ENode pointer variables called *left* and *right*. Pointers hold the memory addresses of such ENode objects rather than their data; this allows the memory of the nodes to be dynamically allocated and de-allocated. Pointer variables are especially useful for a linked list—a dynamic data structure—which is used to construct the expression tree. In particular, almost every ENode in the tree “points” to a **left** and **right** ENode. This concept can be represented diagrammatically: In this example, the ENode with data ‘*’ points to the ENode to its **left**, ‘7’, and to the ENode to its **right**, ‘2’. This “pointing” creates a link between the nodes, by means of their addresses, and further composes the linked list, or expression tree.

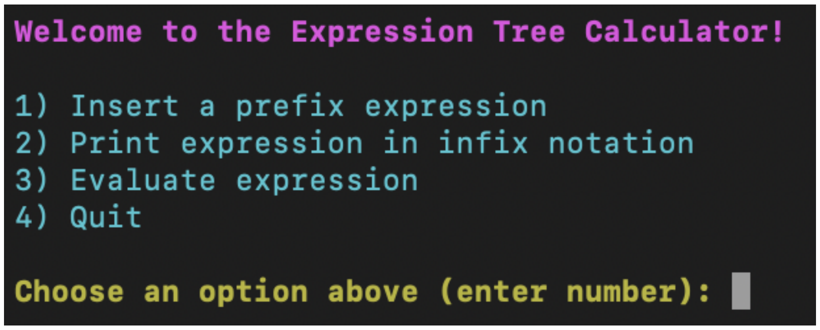


B. Console Colored Output



```
56          printf("\x1b[36m1) Insert a prefix expression\n");
57          printf("\x1b[36m2) Print expression in infix notation\n");
58          printf("\x1b[36m3) Evaluate expression\n");
59          printf("\x1b[36m4) Quit\n\n");
60          printf("\033[1m\033[33mChoose an option above (enter number): \x1b[0m");
```

Lines 56 to 60 are within the *menu* function, and they detail how the menu options are printed for the user to view. Initially, these options were displayed using “cout.” However, after running through the program several times, I realized that it may be hard for the users to decipher the program’s output since everything on the console was displayed in the color white. To eliminate this problem, the function “printf()” was utilized to format and print the menu options in color. As this program was developed on a MacBook, ANSI escape codes that the OS X software responds to were referenced.



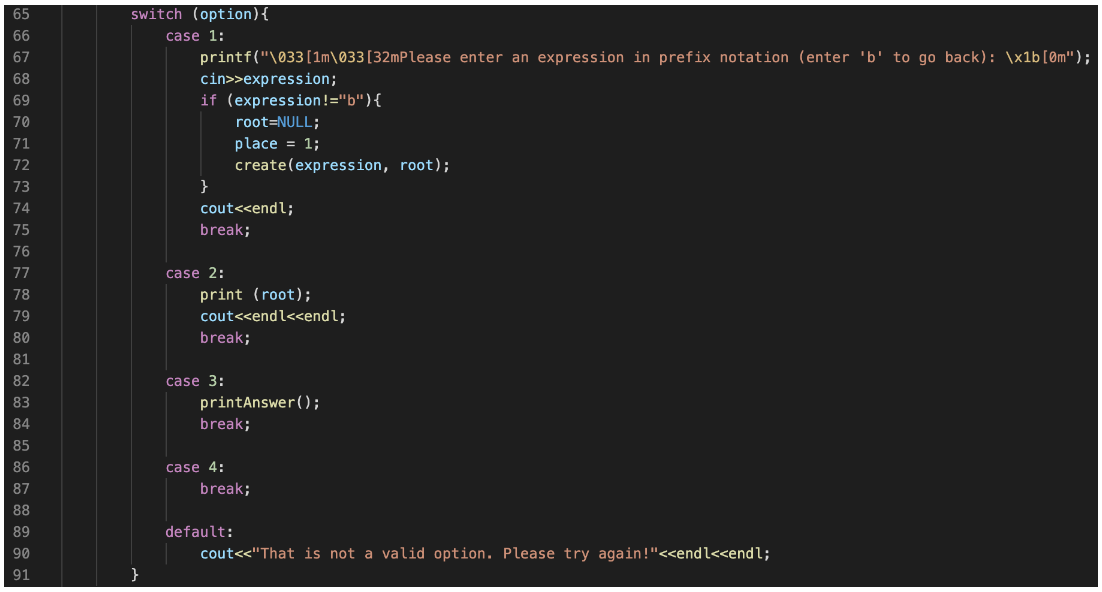
```
Welcome to the Expression Tree Calculator!

1) Insert a prefix expression
2) Print expression in infix notation
3) Evaluate expression
4) Quit

Choose an option above (enter number):
```

This is a screenshot of the console that was taken upon starting the program—the different colors are clearly visible. The use of color allows the text to be more easily read and further makes the product more attractive for the users.

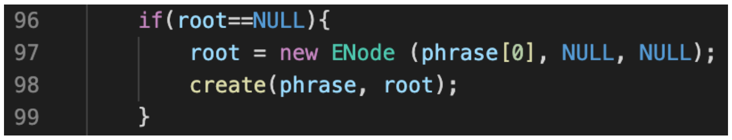
C. Switch Case



```
65          switch (option){
66              case 1:
67                  printf("\033[1m\033[32mPlease enter an expression in prefix notation (enter 'b' to go back): \x1b[0m");
68                  cin>>expression;
69                  if (expression!="b"){
70                      root=NULL;
71                      place = 1;
72                      create(expression, root);
73                  }
74                  cout<<endl;
75                  break;
76
77              case 2:
78                  print (root);
79                  cout<<endl<<endl;
80                  break;
81
82              case 3:
83                  printAnswer();
84                  break;
85
86              case 4:
87                  break;
88
89              default:
90                  cout<<"That is not a valid option. Please try again!"<<endl<<endl;
91          }
```

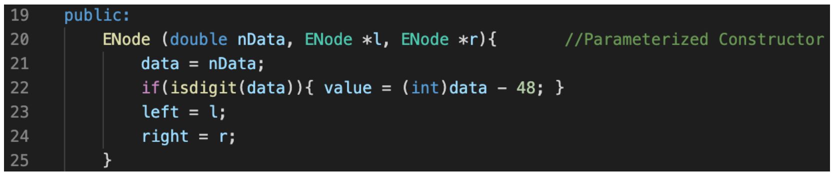
Lines 65 to 91 of the code reveal the application of the switch case technique. First, the switch statement takes in the option inputted by the user. If the option matches one of the four cases outlined from line 66 to 87, then the program will carry out the respective code and resume its function. However, if the option does not match any case, the program will redirect the user to enter a valid option, which handles a simple user error. This technique is useful because it is able to execute the menu functions, similar to if-else statements, whereas this technique also improves the clarity and further reduces the amount of repetitiveness of the code.

D. Creating an ENode



```
96      if(root==NULL){
97          root = new ENode (phrase[0], NULL, NULL);
98          create(phrase, root);
99      }
```

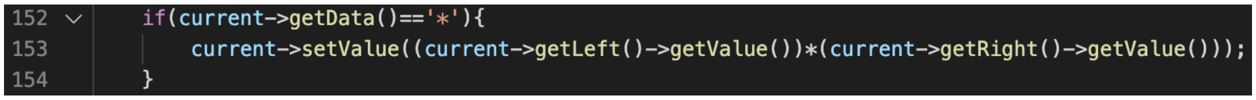
Line 97 of the code shows the initialization of *root* as an ENode object. This initialization is significant because it is the first node of the expression tree, thereby literally "initiating" its construction. After this root node is established, the code will continue to create ENodes for the rest of the operators and operands in the expression to complete the rest of the tree. To create an ENode, the code calls upon its constructor and passes in its respective parameters:



```
public:
    ENode (double nData, ENode *l, ENode *r){       //Parameterized Constructor
        data = nData;
        if(isdigit(data)){ value = (int)data - 48; }
        left = l;
        right = r;
    }
```

Given the parameters, this code initializes the ENode by assigning its attributes. In this case, the root node would not be assigned a value as it is an operator rather than an operand.

E. Calling methods from other methods

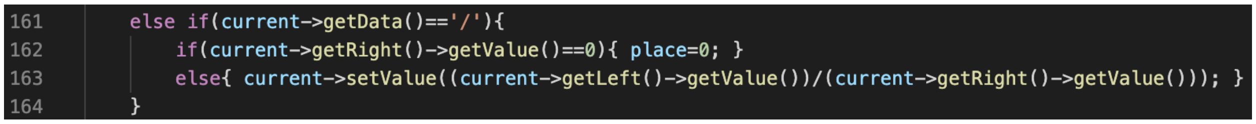


```
152      if(current->getData()=='*'){
153          current->setValue((current->getLeft()->getValue())*(current->getRight()->getValue()));
154      }
```

Lines 152 to 154 are found within the *evaluate* method, which is further a part of the *ETree* class. In particular, this code calls the methods in the ENode class from the evaluate method. This technique is useful because, since *current* is an ENode, its attributes (data, value, and left and right nodes) are encapsulated within its own class. *(Refer to lines 15 to 18 of the screenshot in technique A for the declaration of the private variables).* Therefore, in order to retrieve current's attributes while in the ETree class, its respective methods—known as mutators and accessors—from the ENode class are called. This allows the evaluation of the expression.

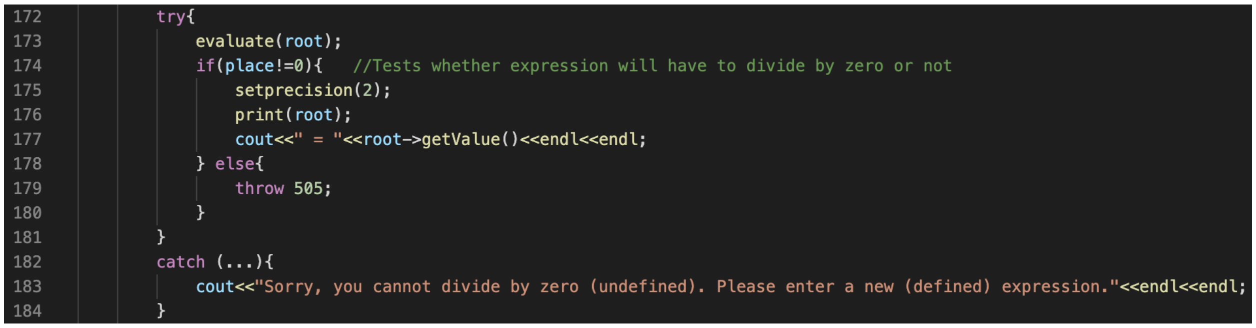
<u>F. Try and Catch Errors</u>

Perhaps the most significant error that a user could encounter in this program is the divide by 0 error. To avoid this error, a two-part algorithm is incorporated in the code. The first part can be found within the *evaluate* method:



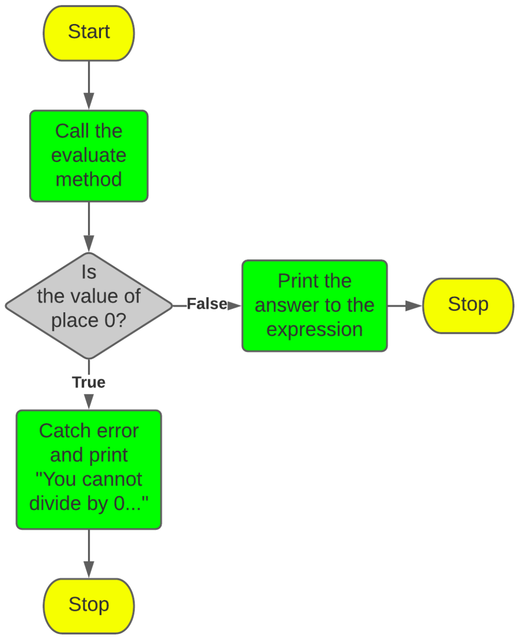
```
161        else if(current->getData()=='/'){
162            if(current->getRight()->getValue()==0){ place=0; }
163            else{ current->setValue((current->getLeft()->getValue())/(current->getRight()->getValue())); }
164        }
```

Line 162 of the screenshot mentions the use of the variable named *place*, which is declared and initialized as a global variable. Before evaluating the division operator, the algorithm will check if the divisor is 0. If so, then the program will set the place variable equal to 0. If not, the program will continue to evaluate the expression. The second part of the algorithm is as follows:



```
172            try{
173                evaluate(root);
174                if(place!=0){   //Tests whether expression will have to divide by zero or not
175                    setprecision(2);
176                    print(root);
177                    cout<<" = "<<root->getValue()<<endl<<endl;
178                } else{
179                    throw 505;
180                }
181            }
182            catch (...){
183                cout<<"Sorry, you cannot divide by zero (undefined). Please enter a new (defined) expression."<<endl<<endl;
184            }
```

Lines 172 to 184 depict try and catch statements, which are useful because they allow you to attempt to execute the code, but also handle any possible errors that may arise from it. This technique is ideal for this type of error.


Start
Call the evaluate method
Is the value of place 0?
False
Print the answer to the expression
Stop
True
Catch error and print "You cannot divide by 0..."
Stop

In particular, the try block will "try" to evaluate the expression; if it encounters the divide by 0 error (which is indicated by the value of place being 0), then the catch block will "catch" and handle the error. This technique can be represented diagrammatically

Criterion E: Evaluation

Success Criteria	**Evaluation**
Users will be presented with a menu option to manually enter an expression.	MET—Selection of the first menu option allows the user to enter an expression.
The program will continue to run and present the user with the menu after each task until the user quits.	MET—Menu appears after each task until option 4 is chosen by the user, quitting the program.
The program can convert an expression from prefix to infix notation.	MET—Program uses (check left —> print—> check right) recursive algorithm to print the expression.
The program can add, subtract, multiply, divide, and calculate exponents.	MET—Program can calculate basic operators with no errors.
The program can calculate multiple operators in one expression.	MET—Program can handle multiple operators if there are enough operands.
The program can accurately evaluate the expression.	MET—Expression answers are correct and can be checked against other calculators.
The program can handle printing and evaluating errors if no expression is inputted.	MET—Program redirects the user to enter an expression first.

Feedback

After reviewing the final project, {Client} suggested to combine options 2 and 3, as there was no reason to keep them separate. I agreed with this judgment, as the combination could make the program more efficient. Both {Client} and her students appreciated the incorporation of color in the expression tree calculator. Overall, {Client} asserted that she was pleased with the final product, and that it would be a straightforward, effective tool for her students to use to calculate expressions.

Improvements

Inclusion of decimals and multiple-digit numbers—This program can only calculate single-digit numbers. As expressions grow complex with more rational numbers, it would be ideal if the program could also evaluate them.

Extra operators—The program would be more advanced if it could calculate operators like factorials and square roots. However, this was not considered a problem since {Client}'s students have not learned such complex operators yet.

Appendix

<u>Notes from first meeting with {Client}:</u>

- Client brought up that she was teaching the concept of the order of operations to her students
- Client wants some type of tool that would make it easier for her students to enter expressions in a calculator to observe the order of operations
- Solution should be straightforward and simple for a fourth-grader to use

<u>Notes from second meeting with {Client}:</u>

- Expression tree calculator solution was suggested
- Client was initially puzzled about the prevalence of prefix notation in the solution, but after it was explained that prefix notation would eliminate the use of parentheses, she agreed that it was a suitable solution
- Client confirmed that the program can be written in C++
- Client helped suggest success criteria for the product and confirmed them as well

<u>Notes from third meeting with {Client}:</u>

- Client understood the different functions of the program
- Client was able to test out some of the functions herself and appreciated the accuracy of the code
- Client suggested that options 2 and 3 in the product (print and evaluate) be combined, but asserted that this change was not completely necessary and that the product would still be useful without it

<u>Notes from fourth and final meeting with {Client}:</u>

- Both client and her students tested out the final product
- Everyone seemed pleased with the product and found it easy to use
- Students (users) enjoyed the use of color in the product
- Client and I affirmed that the product achieved the outlined success criteria