Final Project

20%

A. Rules

Deadline

2021/1/10 (Sun) 23:59:59

Things you should know

Files to Edit and Submit: You need to implement functions by filling in portions of main.cpp, Expr.cpp, Expr.h, LargeNumber.cpp and LargeNumber.h in this project. You should submit these *five files only*. Pack these files and upload it to Ceiba. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's judgements -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will check your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the TA for help. Office hours and TA's email are there for your support; please use them. We want this project to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask. One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out.

Discussion: Please be careful not to post spoilers.

B. Helicopter Tour

In this project, we will be building a real-time (Python-like) calculator that does all sorts of arithmetic.

Below is what the finished project will look like, do go ahead and give it a quick glance.

```
ans = 0
>>>  + * 12 x y
ans = + * 12 x y
>>> + * ans z ans
ans = + * + * 12 x y z + * 12 x y
>>> subst subst ans 12 x x eval + 1 2
ans = + * + * 3 3 y z + * 3 3 y
>>> eval subst subst ans y 2 z 5
```

```
ans = 66
>>> quit
```

Roadmap for this project

There are five main parts in this final project:

1. Understanding prefix notation and expression tree (B.1 ~ B.2)
2. Implementation of the nodes for an expression tree (Part C)
3. Dynamic construction & modification of an expression tree from user input (Part D)
4. Making a handle class for the nodes (Part E)
5. Modifying the containers to support large number computing (Part F)

B.1 Infix / Prefix / Postfix

While we use infix expressions in our daily lives, computers have trouble understanding this format because they need to keep in mind rules of operator precedence and also brackets. Prefix and postfix expressions are easier for a computer to understand and evaluate.

Given two operands `a` and `b` and an operator `⊙`, the infix notation implies that `⊙` will be placed in between `a` and `b` i.e `a⊙b`. When the operator is placed after both operands i.e `ab⊙`, it is called a postfix notation. And when the operator is placed before the operands i.e `⊙ab`, it is called a prefix notation.

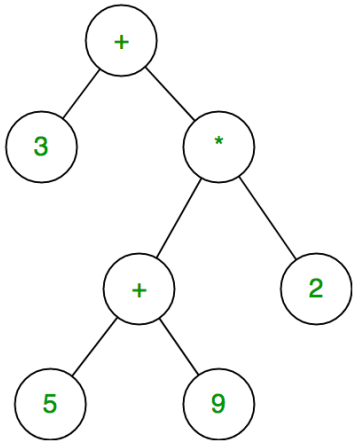
Examples of infix to prefix and post fix

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

B.2 WHAT is an expression tree?

An expression tree is a special kind of tree used for representing an expression.

The above shows the tree of an infix expression `(3+((5+9)*2))`.



C. Implementing nodes for an expression tree (7%)

C.1 Class `expr_node`

To create the nodes in an expression tree, first we need is an abstract class as a base class.

The code is given by:

```
class expr_node{
  public:
    expr_node(){}
    virtual ~expr_node() {}
    virtual int eval() const = 0;
    virtual void subst(const string&, const string&) = 0;
    virtual string getString() const = 0;
};
```

Now, we'll start by declaring 2 simple yet important nodes called `uni_node` and `binary_node`, they both inherit from the base class `expr_node`.

C.2 Class `uni_node` & `binary_node`

The uni-node represent a node of a number or a variable.
The binary node represent a node of a binary operator.

You are required to finish the definition of all the public functions in both classes.

The code is given by:

```
class uni_node: public expr_node{
  private:
    string value;
  public:
    uni_node(const string& _val);
    ~uni_node();
    int eval() const;
    void subst(const string& from, const string& to);
    string getString() const;
};

class binary_node: public expr_node{
  private:
    string op;
    expr_node* lchild;
```

```
    expr_node* rchild;
  public:
    binary_node(const string& _op, expr_node* lch, expr_node* rch);
    ~binary_node(); // should also delete its subtree recursively
    int eval() const;
    void subst(const string& from, const string& to); // should also propagate this operation to its child
    string getString() const;
};
```

1. Constructor

Construct the child nodes and set the value of the variables.

The name of a variable could be any combination of characters except operators(+-*/%).
e.g. `xy2`, `HaHa`, `tmp_2`.

2. Destructor

Destruct the expression tree with the given node as its root.

3. Evaluate

Calculate the value of the expression tree with the given node as its root.
Note that there are only 5 types of operators, + - * / %. For division, please implement the integer division.

For uni-node, you do not have to specially handle the case of evaluating a variable.

4. Substitute

If there exist any node with value `from` in the expression tree, substitute it with the value `to`. Your program should support substitution of

a. variable by integer

   e.g. `subst("x","2")` : substitute "x" with "2".

b. integers by variable

   e.g. `subst("2","x")` : substitute "2" with "x".

c. variable by variable

   e.g. `subst("y","x")` : substitute "y" with "x".

Note that we will ignore the case in which `from` or `to` is an operator.

5. getString

Return the prefix expression of the expression tree with the given node as its root. Please separate each operator and/or operand by a " ".

C.3 Results

If everything is done correctly, your program should be able to run this code segment:

```
// creates an uni-node with value 1 pointed by exp1
uni_node* exp1 = new uni_node("1");

// creates an uni-node with value x pointed by exp2
uni_node* exp2 = new uni_node("x");

// creates an binary node for addition with two uni-nodes as its left/right child
binary_node* exp3 = new binary_node("+", exp1, exp2);

// substitute x with 2 to get "1+2"
exp3->subst("x", "2");

// print out the prefix expression of the expression tree, "+ 1 2"
cout << exp3->getString() << endl;

// evaluate the tree that essentially calculates "1+2", prints 3
cout << exp3->eval() << endl;

// recycle the memory allocated by the whole tree
delete exp3;
```

Or even simpler :

```
expr_node* t = new binary_node("+", new uni_node("1"), new uni_node("x"));
t->subst("x","2");
cout << t->getString() << endl;
cout << t->eval() << endl;
delete t;
```

D. Making your expression trees more accessible (5%)

In this part, we will be creating an interactive interface for users to create/modify/evaluate a prefix expression.

Below shows the behavior of the completed program.

```
ans = 0
>>>  + * 12 x y
ans = + * 12 x y
>>> + * ans z ans
ans = + * + * 12 x y z + * 12 x y
>>> subst subst ans 12 x x eval + 1 2
ans = + * + * 3 3 y z + * 3 3 y
>>> eval subst subst ans y 2 z 5
ans = 66
>>> quit
```

Explanation

1. `ans` is initialized to 0. After running every command, the result should be assigned to `ans`.

2. You can input any prefix expression in every command. You can use `eval` and `subst` in every command too. Your program should print out `ans`, and it should also support `eval` and `subst`. `ans` and variables such as x, y or z can appear in every command.
3. `eval` means evaluation of the expression. For example, the result of `eval + 1 2` is 3.
4. `subst` means substitution of an operand. Your program should support substitution of variables by integers and substitution of integers by variables. For example, the result of `subst + * 12 x y x 5` is `subst + * 12 5 y`, and the result of `subst + * 12 x y 12 z` is `subst + * z x y`.
5. `quit` should terminate the execution of your program.

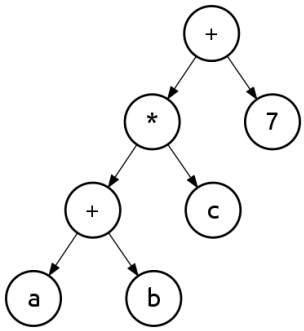
More about the input format

For this assignment, there are basically three components in an input command

1. a prefix expression

   meaning : build an expression tree representing the prefix expression

   e.g. `+ * + a b c 7` (every element is separated by a " ")



2. substitute operation (denoted by subst)

   structure : "subst `prefix expression` `from` `to`"

   meaning : substitute everything in `from` to `to` in the expression

   e.g. "subst `* + a 1 1` `a` `7`" means substituting every `a` in `* + a 1 1` to `7`

   which yields "`* + 7 1 1`" (a modified prefix expression)

3. evaluate operation (denoted by eval)

   structure : eval `prefix expression`

   meaning : evaluate the value of the prefix expression

   e.g. eval `* + 1 2 3` will return 3*(1+2) = 9

Before we proceed,

Always remember ***the substitute operation returns an expression tree***

Now you know what each component means, well and good.

However from this assignment, you'll see inputs like this form

"eval subst + - a 5 7 a 2"

Don't fret yet if you have 0 idea what it means

Let's break it down step by step

First, it is clear from the structure of subst operation, that the last two token `a` `2` are supposed to be `from` and `to`

Now, the input can be viewed as "eval (subst `+ - a 5 7` `a` `2`)"

But wait, what's "subst `+ - a 5 7` `a` `2`"?

It's the prefix expression "`+ - 2 5 7`"

So the original input is actually "eval `+ - 2 5 7`", which yields an integer 4

Now let's try another example :

"eval sbust subst * - 3 b a a 2 b eval - 2 3"

Give it a thought, see if you understand this.

The answer is 8.

Let's start from the beginning.

Observe the last four tokens, "eval `-` `2` `3`", doesn't that look familiar?

Of course it does, it's just a simple integer -1

Now, what's left is "eval subst subst * - 3 b a a 2 b -1"

See anything? No?

What about if I temporarily rewrite "subst * - 3 b a a 2" as "prefix expression"

Now the whole input becomes "eval subst `prefix expression` `b` `-1`"

You can see that all this does is substitute every `b` in `prefix expression` into `-1`, then evaluate the expression.

And what's `prefix expression`, it's "subst * - 3 b a a 2", which is the prefix expression "* - 3 b 2"

Quick summary

What does "eval `subst subst * - 3 b a a 2 b eval - 2 3`" do?

It "eval"s a `prefix expression` returned by "subst subst * - 3 b a a 2 b eval - 2 3"

And what does "subst `subst * - 3 b a a 2` `b` `eval - 2 3`" do?

It "sub"s all `b` into `eval - 2 3` from the expression returned by "subst * - 3 b a a 2"

And what does "subst `* - 3 b a` `a` `2`" do?

It returns the prefix expression "* - 3 b 2"

From all the examples above, we can see that it's easier for us to interpret the input backwards

And that's what we recommend doing in this assignment.

Now, in order to complete this task, you will need

1. to implement an aiding function tokenizer
2. thorough understanding of what the input does
3. basic understanding of Stack

D.1 Tokenizer

`bool tokenizer(const string& str, vector<string>& tokens)`

Process the input string `str` and split it into a vector `tokens` of strings by white-spaces.

If the input string is "quit", return false to end the program. Otherwise, return true.

Sample Input

```
str = "Programming is fun"
```

Result

```
tokens = "Programming", "is", "fun"
return true
```

Hint: If you need any help, please refer to `string_demo.cpp`. You will find most of your answer there.

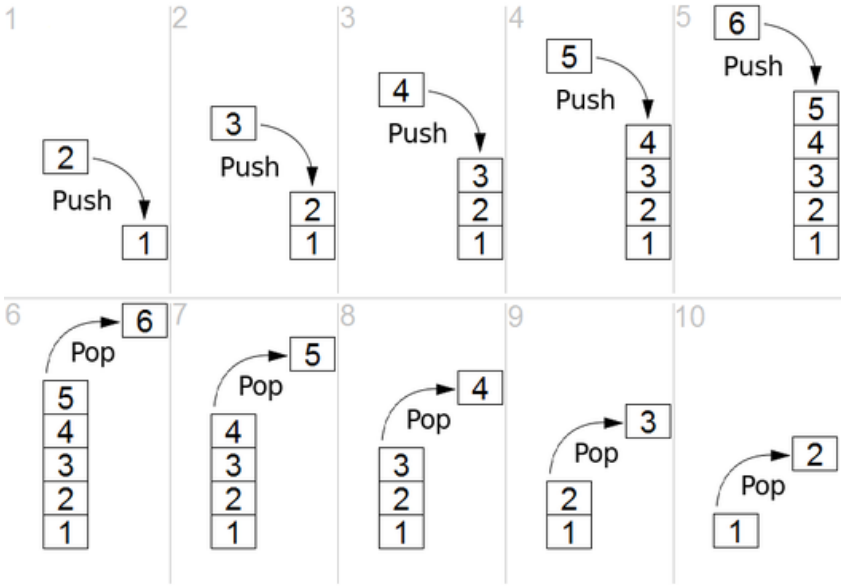
D.2 Stack

What is a stack?

In computer science, a **stack** is an abstract data type that serves as a collection of elements, with two main principal operations:

- **Push**, which adds an element to the collection, and
- **Pop**, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, **LIFO** (**last in, first out**). Additionally, a peek operation may give access to the top without modifying the stack. The name "stack" for this type of structure comes from the analogy to a set of physical items stacked on top of each other. This structure makes it easy to take an item off the top of the stack, while getting to an item deeper in the stack may require taking off multiple other items first.



How to use a stack?

```
#include <list>
```

Hint: If you need any help, please refer to `list_demo.cpp`. You will find most of your answer there.

D.3 Building an expression tree with a stack

We'll start by giving an example.

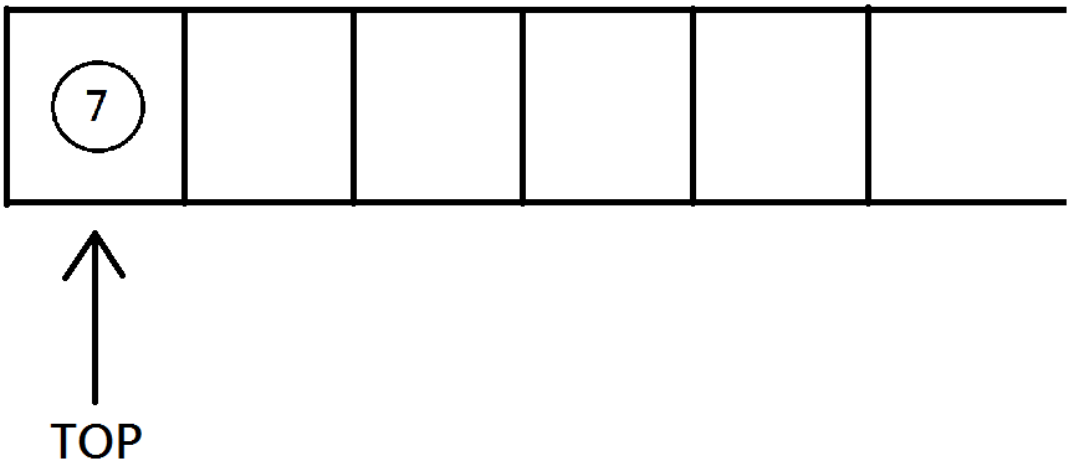
Consider the prefix expression "`+ * a b 7`"

First, we initialize an empty stack called S.

Now, starting from the end of the expression, we have a token 7.

Since 7 is an operand, we create an uni-node with the value 7 and push it into S.

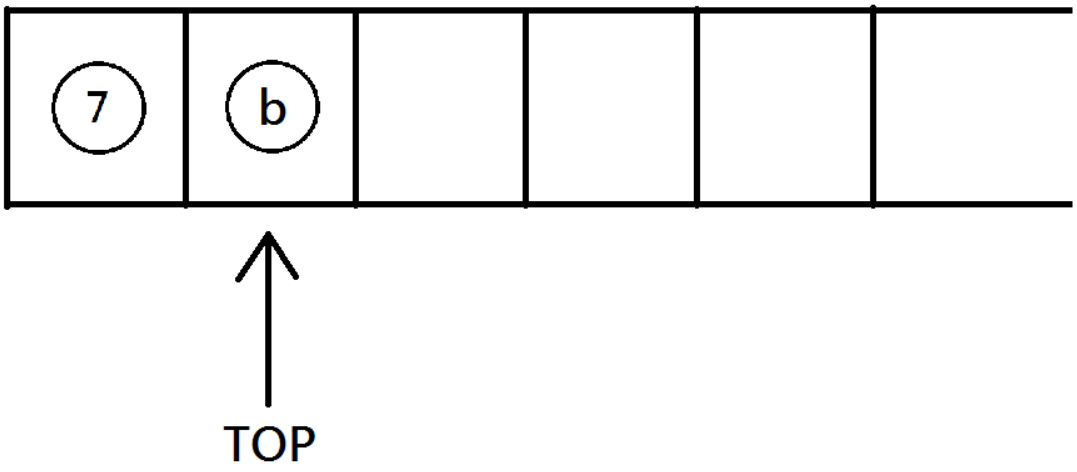
Now, S becomes



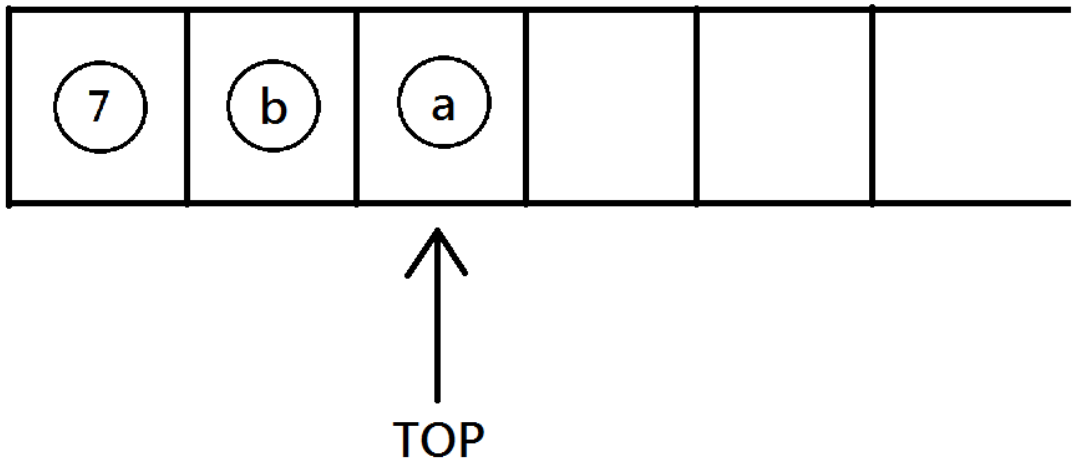
Next, we move on to the second last token, which is b

Following the same logic above, we initialize an uni-node with value b and push it into S.

Now, S becomes



Likewise, insert the following token a into S



Now comes the tricky part, we encounter this time an operator * rather than an operand.

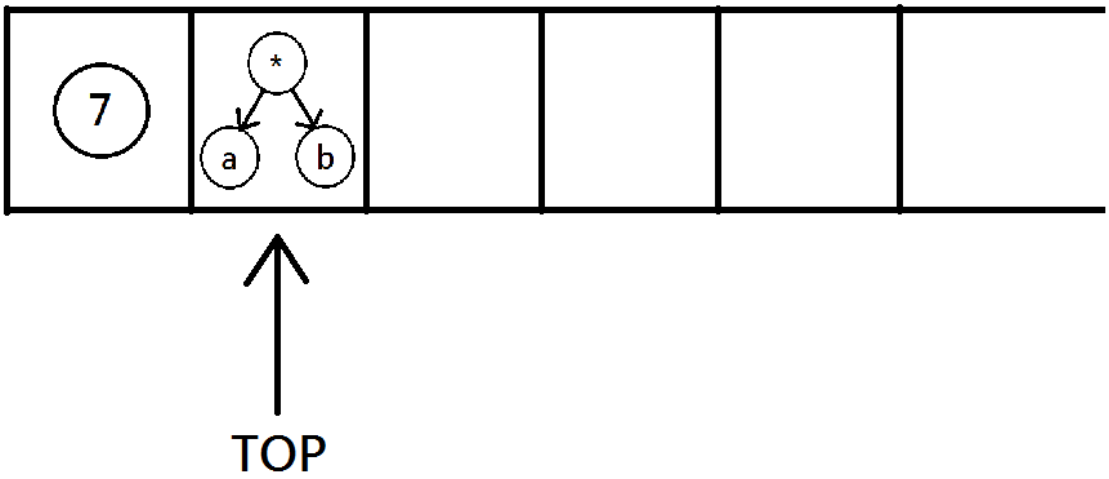
What we do is :

1. pop 2 nodes from the stack
2. construct a new binary-node with the 2 popped nodes
3. push the new binary-node back into the stack

So, in this case, we will

1. pop an uni-node with the value a, and another uni-node with the value b
2. construct a new binary-node with those two uni-nodes
3. push the new binary-node back into the stack

Thus, S becomes

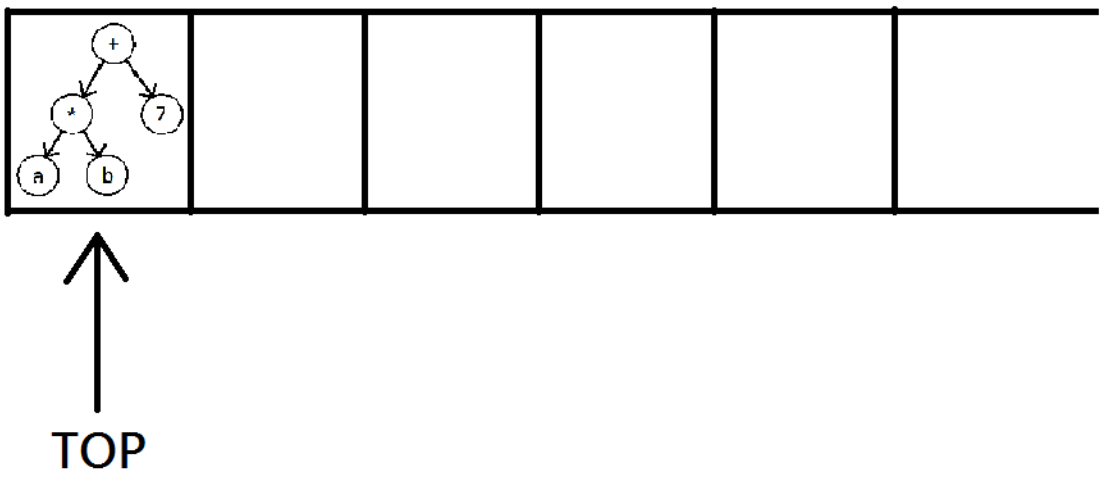


Moving on, again we encounter an operator, this time it's a +

Following the same steps illustrated above, we will

1. pop the binary-node we just created in the last step, and another uni-node with the value 7
2. construct a new binary-node with those two nodes
3. push the new binary-node back into the stack

Finally, S is now



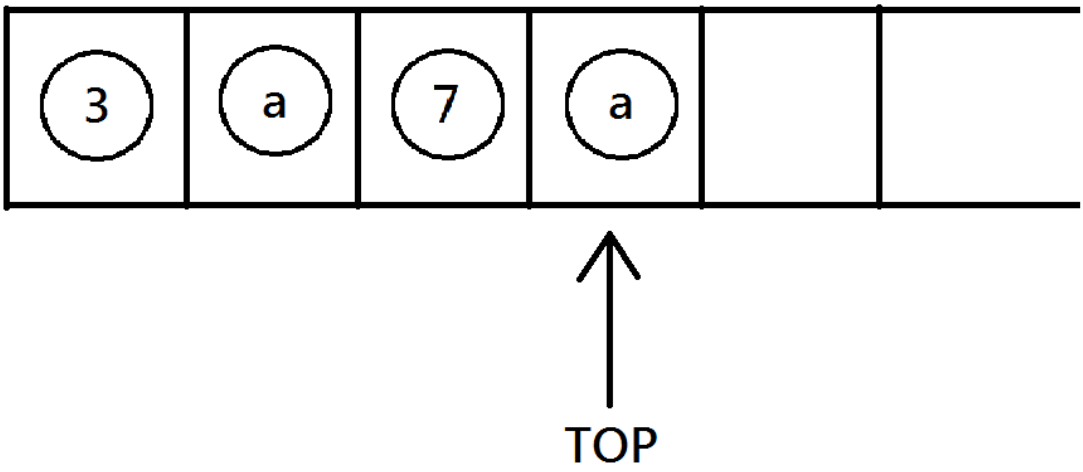
As you can see, the tree we want will be at the top of the stack.

D.4 Substituting an expression tree with a stack

Again we demonstrate with an example.

Consider the operation "subst + a 7 a 3"

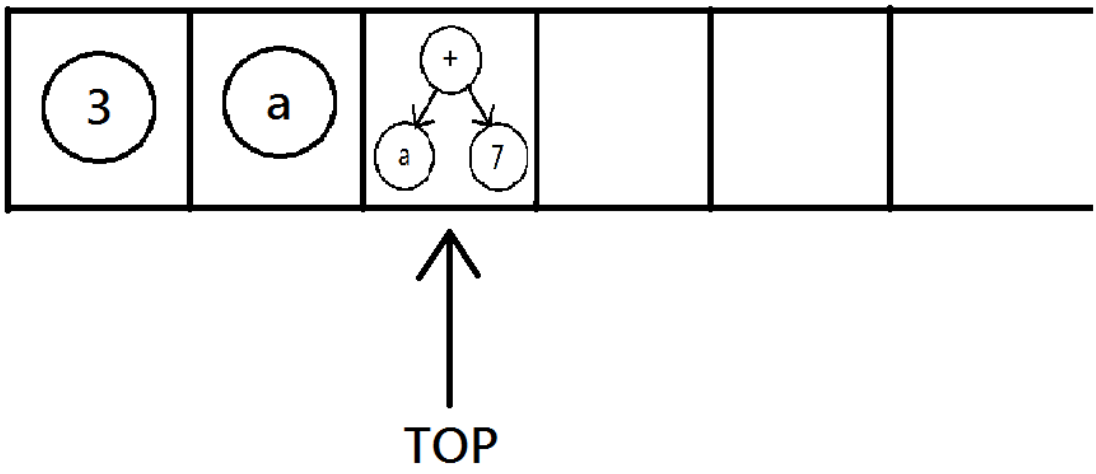
If we follow the same logic we did before, before we hit the operator + the stack S will be



Now, we process the operator + the same way as before

1. pop two nodes from the stack, in this case they are two uni-nodes containing a and 7
2. construct a new binary-node with those two uni-nodes
3. push the new binary-node back into the stack

As a result, S now becomes



Now, we encounter the token "subst", let's remind ourselves what the subst structure looks like

subst `prefix expression` `from` `to`

But because we decided to process the tokens from right to left, `to` will be pushed into the stack first, followed by `from` and lastly by `prefix expression`.

So the first node we pop will be the `prefix expression` for subst, and the second node will be `from`, and the third node will be `to`.

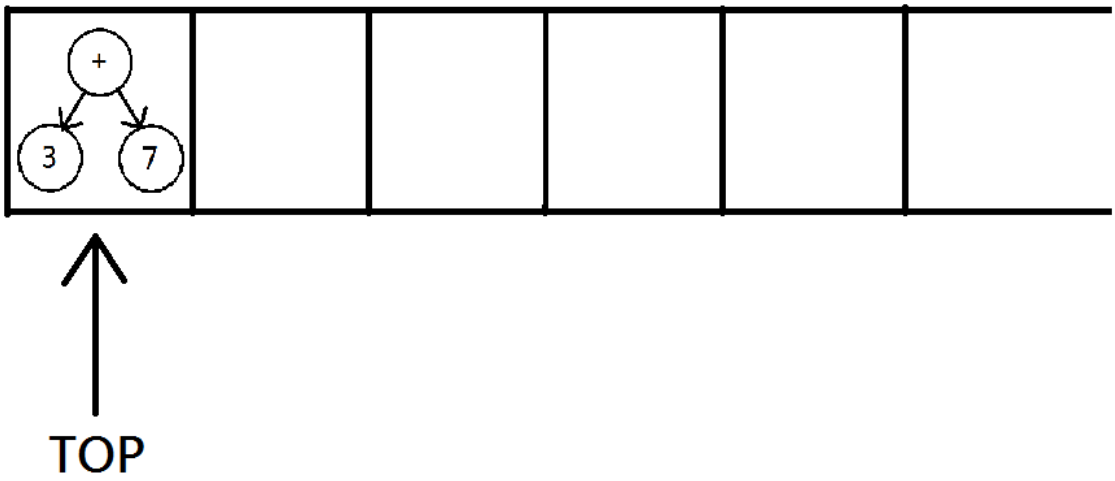
So, to summarize, what we do when we encounter subst is

1. pop three nodes from the stack, its order being `prefix expression`, `from`, then `to`
2. call the subst member function for `prefix expression` with `from` and `to` to obtain a "modified" `prefix expression`
3. push the "modified" `prefix expression` back into the stack

In this case, `prefix expression` is "+ a 7", `from` is a, and `to` is 3

After substituting `prefix expression`, the "modified" `prefix expression` is "+ 3 7"

Push the "modified" `prefix expression` back to the stack, thus the stack becomes



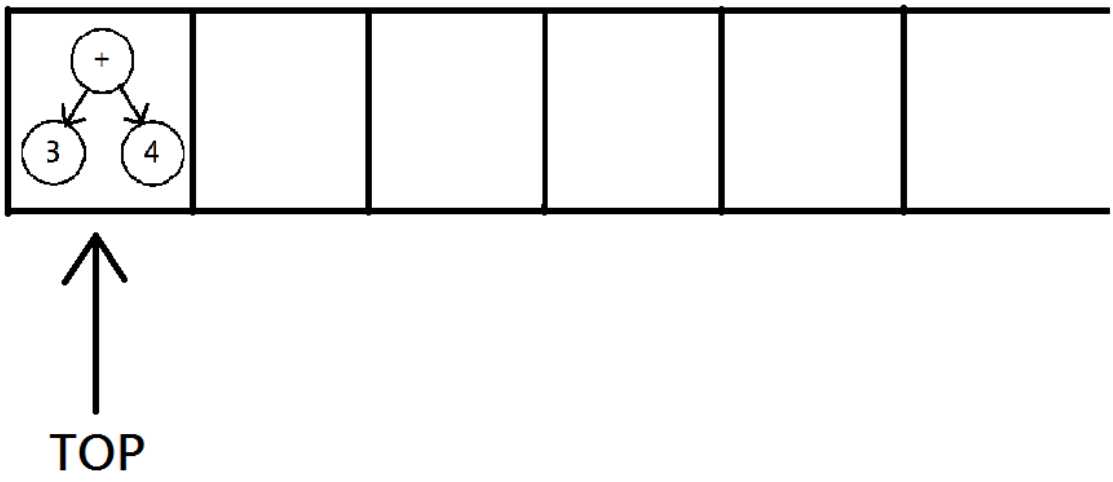
Again, the tree we want will be at the top of the stack.

D.5 Evaluating an expression tree with a stack

Consider the operation "eval + 3 4"

Now you should be familiar enough to handle everything before encountering the eval token

After processing tokens "4", "3", "+", the stack becomes



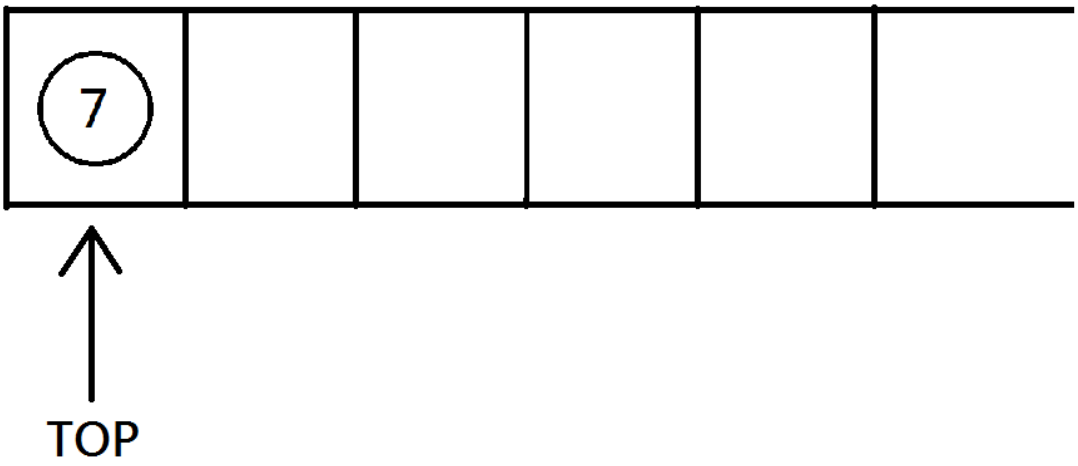
Now comes the token "eval", as the structure of eval suggests, an prefix expression should follow the eval token

eval `prefix expression`

Now, what we will do is pretty straightforward

1. pop a single node from the stack, which will be the `prefix expression`
2. call the eval member function for the node, obtain an int answer
3. create a new uni-node with answer, and push it back to the stack (note that the tree will NOT be pushed back)

So, in this case, `prefix expression` "+ 3 4" evaluates an integer 7, and an uni-node containing 7 will be pushed back into the stack



E. Pointers Handling (5%)

E.1 Expr (handle class)

There are some problems if we construct the tree using `expr_node`.

Take the following code as an example:

```
//(1+2)
binary_node *exp3 = new binary_node('+', new uni_node(1), new uni_node(2));
binary_node *exp4 = new binary_node('*', exp3, exp3);
cout << exp4->eval();
```

The problems are that

1. we have to remember to delete the nodes since the nodes are allocated dynamically. Also, we do not have pointers to the objects that created in the inner new calls(e.g. `new uni_node(1)`, `new uni_node(2)`).
2. deleting exp3 first will cause exp4 to crash since exp4 also points to the expression.

So we need a handle class `Expr` to manage the pointers and the destructor.

E.2 Modify class `expr_node`

In order to support class `Expr`, we need to modify class `expr_node` first.

1. Add member: use count

The use count records how many `Expr`s point at particular `expr_node` to avoid the problem 2 mentioned above.

Maintain a use count in class `expr_node` and set to one when an `expr_node` is constructed.

2. Replace `expr_node*` members with `Expr`

Since we want to use `Expr` to maintain pointers to `expr_node`, please modify the constructor and the data types of variables in class `binary_node`.

E.3 Class `Expr`

The declaration of the class is given below:

```
class Expr {
public:
    Expr(const Expr& expr);
    Expr(const string& operand);
    Expr(const string& op, Expr left, Expr right);
    ~Expr();
    Expr& operator= (const Expr& expr);
    int eval() const;
    string getString() const;
    void subst(const string& from, const string& to);

protected:
```

```
    expr_node *p;
};
```

There is a pointer to `expr_node` which is a protected variable of this class.

You are required to implement the public functions.

1. Constructors

The `Expr` constructors can represent all kind of `expr_node`s. Each constructors will create an object of an appropriate class.

E.g. `Expr("1")` create an `uni_node`.

Also, when a new `Expr` points to another `Expr_1`, increase the use count of the `expr_node` of `Expr_1`.

E.g. `Expr(Expr_1)` : the use count of the `expr_node` of `Expr_1` will increase by one.

2. Destructor

Decrease the use count of the `expr_node` and delete the `expr_node` when the use count equals zero.

3. Copy constructors

`Expr& operator= (const Expr& expr)`

The assignment operators has to increase and decrement the use counts of the right-hand and left-hand sides, respectively.

Delete the `expr_node` when the use count equals zero.

4. Evaluate

Calculate the value of the `expr_node` and return the value.

5. getString

Return the prefix expression of the `expr_node`.

4. Substitute

If there is any nodes with value `from` in the expression tree, substitute it with the value `to`.

F. Make your expression tree more powerful (3%)

Introduction

In week 11, we implemented arithmetics of large numbers using functions. In that scenario, data and the functions that work on that data are separate entities that are combined together to produce the desired result. Because of this separation, traditional programming often does not provide a very intuitive representation of reality. Object-oriented programming (OOP) provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

Class LargeNumber

Up to this point, we have build an expression tree which can operate (small) integers. In this section, we will build a class named `LargeNumber` which allows our expression tree to operate

arbitrary large integers. Thanks to the class structure, upgrading our previous code on expression trees is easy once we have implemented the `LargeNumber` class.

Problem Statement

Implement the following header file `LargeNumber.h` of a class called `LargeNumber` in the `LargeNumber.cpp`. There are total five arithmetic operations need to be implemented, namely addition, subtraction, multiplication, quotient, and remainder. Feel free to add more member functions and variables in the public, protected, and/or private sections if needed.

```
class LargeNumber
{
public:
  // null constructor
  LargeNumber();

  // LargeNumber a("123456")
  LargeNumber(const string &raw);

  // copy constructor
  LargeNumber(const LargeNumber &ref);

  // destructor
  ~LargeNumber();

  // addition
  LargeNumber operator+(const LargeNumber &num) const;

  // substraction
  LargeNumber operator-(const LargeNumber &num) const;

  // multiplication
  LargeNumber operator*(const LargeNumber &num) const;

  // quotient
  LargeNumber operator/(const LargeNumber &num) const;

  // remainder
  LargeNumber operator%(const LargeNumber &num) const;

  // copier
  LargeNumber operator=(const LargeNumber &num) const;
};
```

Remark

All numbers are integers and could be of any length. Here is some specification on the quotient operation and the remainder operation. Based on the C99 Specification: `a == (a / b) * b + a % b`. From this, we can write a function to calculate `(a % b) == a - (a / b) * b`, that is,

```
int remainder(int a, int b)
{
    return a - (a / b) * b;
}
```

Example A. `5/(-3)` is `-1`

$\Longrightarrow$ `(-1) * (-3)` + `5%(-3)` = `5`

This can only happen if `5%(-3)` is `2`.

Example B. `(-5)/3` is `-1`

$\Longrightarrow$ `(-1) * 3` **+** `(-5)%3` **=** `-5`

This can only happen if `(-5)%3` is `-2`

G. Some words from TAs

Congratulations! You have learned a lot in this semester, and we sincerely believe you have earned the critical ability to solve problem using computer programming. We hope you enjoy coding this semester. Get ready for the final exam, and have fun in winter vacation. You will become an extraordinary programmer if you keep on learning. Last but not the least, we wish you a merry Christmas and a happy new year. Happy winter vacation!