CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 6 – Stacks and Queues

(Week 8, starting 3 October 2016)

1. *Stack and Queue Operations*

In this exercise, we will use the STL Container **adapter** classes stack<T> and queue<T> to create stack and queue instances, and execute their operations, in the program below. Draw diagrams representing the contents of stack s1, stack s2 and queue q after each operation:

```
#include <stack>
#include <queue>

using namespace std;

int main () {
    queue<int> q;
    stack<int> s1, s2;

    s1.push(3);
    s1.push(2);
    s1.push(1);

    while (!s1.empty()) {
        s2.push(s1.top());
        s1.pop();
        if (!s1.empty())
            s2.push(s1.top());
        q.push(s2.top());
    }

    s1.push(q.front());
    q.pop();
}
```

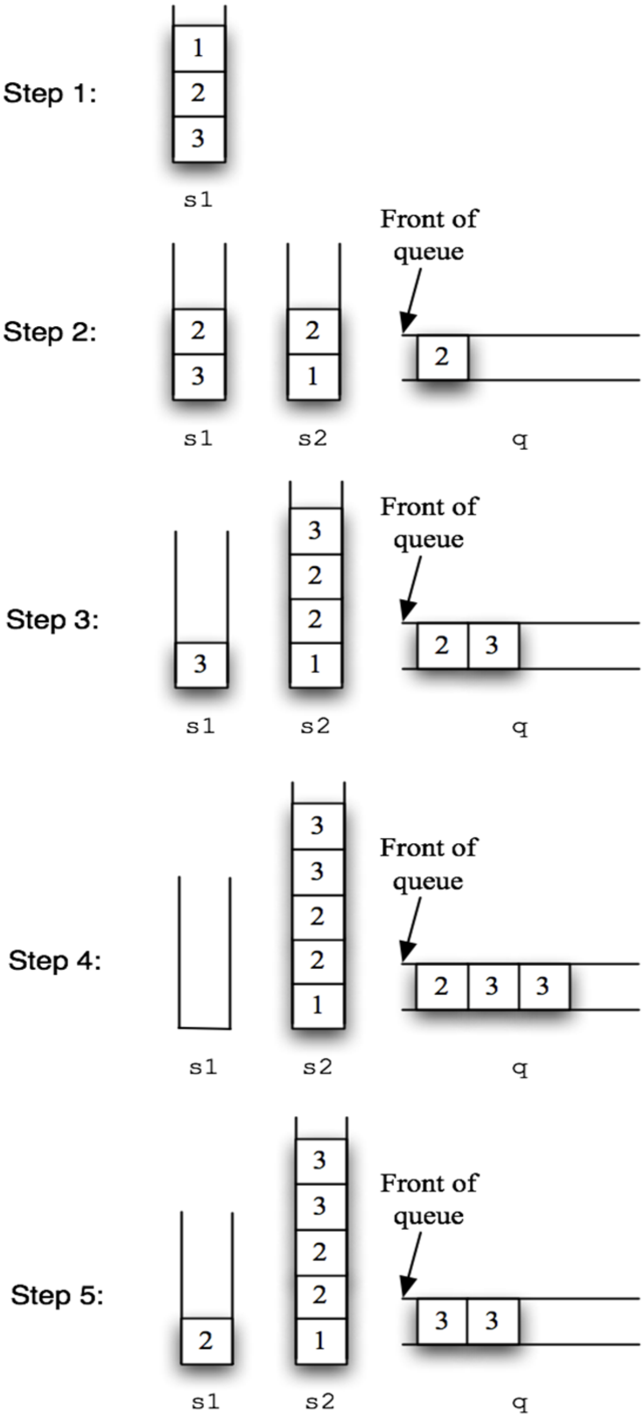
What is the **default underlying data structure** below a stack<T>? queue<T>?
Which data structures can you **choose to use** as the underlying data structure for each of the two adapters?

Answer

Both stack<T> and queue<T> use deque<T> as the default underlying data structure. You can also choose to use, for

stack<T> -	list<T>	vector<T>
queue<T> -	list<T>	

Think why it is reasonable that forward_list<T> is not an option to be used for either adapter, and why vector<T> is not an option to be used as a queue<T>.


Step 1:
1
2
3
s1
Step 2:
2
3
s1
2
1
s2
Front of queue
2
q
Step 3:
3
s1
3
2
2
1
s2
Front of queue
2
3
q
Step 4:
s1
3
3
2
2
1
s2
Front of queue
2
3
3
q
Step 5:
2
s1
3
3
2
2
1
s2
Front of queue
3
3
q

2. Stack and Queue Applications – Expression Evaluation

In the Lisp programming language, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expressions are enclosed in parentheses. There is only one operator in a pair of parentheses. The operators behave as follows:

- `( + a b c )` returns the sum of all the operands, and `( + )` returns 0.
- `( - a b c )` returns a - b - c - … and `( - a )` returns 0 - a.
  The minus operator must have at least one operand.
- `( * a b c )` returns the product of all the operands, and `( * )` returns 1.
- `( / a b c )` returns a / b / c / … and `( / a )` returns 1 / a, using **double division**
  The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:

```
( + ( - 6 ) ( * 2 3 4 ) )
```

The expression is evaluated successively as follows:

```
( + -6.0 ( * 2.0 3.0 4.0 ) )
( + -6.0 24.0 )
18.0
```

Design and implement an algorithm that uses up to 2 stacks to evaluate a legal Lisp expression composed of the four basic operators, integer operands, and parentheses. The expression is well formed (i.e. no syntax error), there will always be a space between 2 tokens, and we will not divide by zero.

Output the result, which will be one double value.

Answer

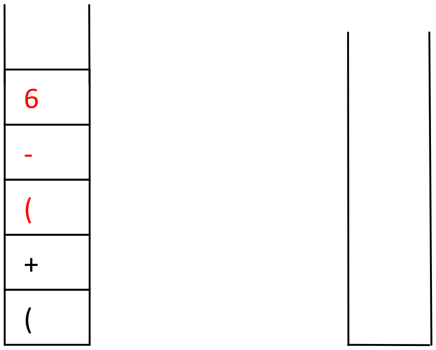
One algorithm uses two stacks. The first is used to store the tokens read from the expression one by one until the operator ")". The second stack is used to perform a simple operation on the operands in the innermost expression already in the first stack.

The tokens are pushed into the second stack in reverse order. Therefore, tokens from the second stack are popped in the order of input. The calculated result is then pushed back into the first stack.

An example is given over the next few pages:

Expression (+ (- 6) (* 2 3 4))

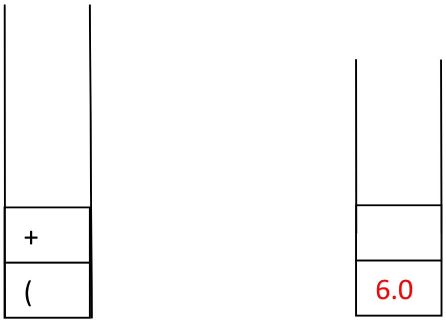
1. The main stack pushes the tokens one by one until it reads “)”



The main stack

The temporary stack for simple calculation

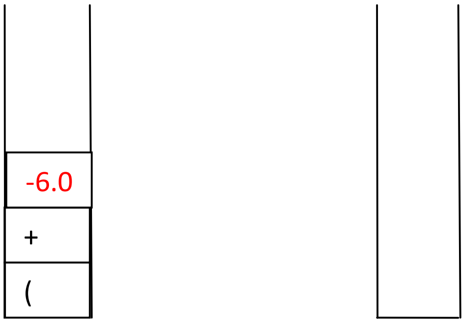
2. The main stack pops out the tokens and push them one by one to the temporary stack



The main stack

The temporary stack for simple calculation

3. The temporary stack pushes back the result after calculation

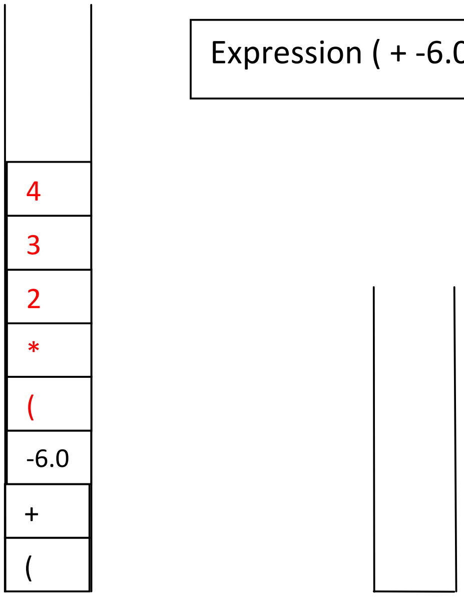


The main stack

The temporary stack for simple calculation

Expression (+ -6.0 (*2 3 4))

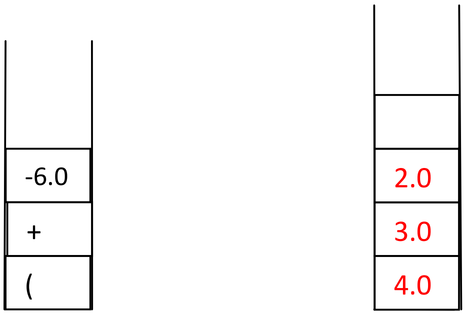
4. The main stack continues to push in the tokens from the expression until it reads “)”



The main stack

The temporary stack for simple calculation

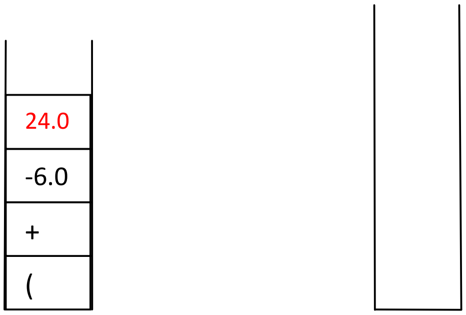
5. The main stack pops out the tokens and push them one by one to the temporary stack



The main stack

The temporary stack for simple calculation

6. The temporary stack pushes back the result after calculation.

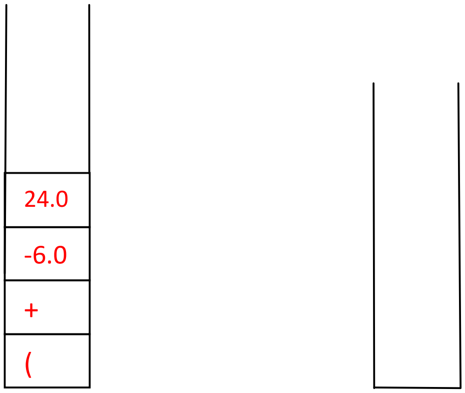


The main stack

The temporary stack for simple calculation

Expression (+ -6.0 24.0)

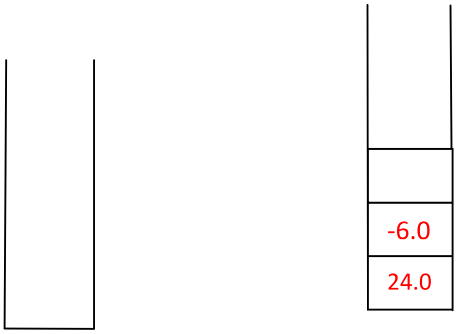
7. The main stack continues to push in the tokens from the expression until it reads ")"



The main stack

The temporary stack for simple calculation

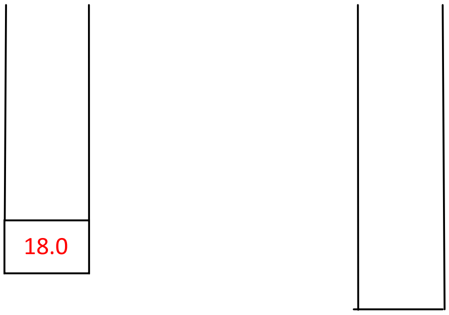
8. The main stack pops out the tokens and push them one by one to the temporary stack



The main stack

The temporary stack for simple calculation

9. The temporary stack pushes back the result after calculation. When it is at the end of expression, the final result is stored in the main stack.



The main stack

The temporary stack for simple calculation

```
double performOperation(stack<double>& oprnds, char oprtor) {
   double result = 0.0;
   switch (oprtor){
     case '+': // 0 + opr1 + opr2 ...
       result = 0.0;
       while (!oprnds.empty()) { result += oprnds.top(); oprnds.pop(); }
       return result;
     case '-':
       if (oprnds.size() == 1) return -oprnds.top(); // -opr1
       result = oprnds.top(); oprnds.pop();
       while (!oprnds.empty()) { result -= oprnds.top(); oprnds.pop(); }
       return result; // opr1 - opr2 - opr3 ...
     case '*':
       result = 1.0;
       while (!oprnds.empty()) { result *= oprnds.top(); oprnds.pop(); }
       return result;
     case '/':
       if (oprnds.size() == 1) return 1 / oprnds.top(); // 1/opr1
       result = oprnds.top(); oprnds.pop();
       while (!oprnds.empty()) { result /= oprnds.top(); oprnds.pop(); }
       return result; // opr1 / opr2 / opr3 ...
   } // switch-case: here returns; don't forget to break otherwise!
} // unspecified behaviour if operator invalid

int main() {
   stack<string> allTokens; // outer stack
   string currentToken;

   while (cin >> currentToken) {
      if (currentToken == ")") {
         stack<double> operands; // inner stack
         while (allTokens.top().size() > 1 || // while operand
            allTokens.top().find_first_of("+-*/")==string::npos) {
            operands.push(stod(allTokens.top()));
            allTokens.pop();
         }
         char oprtor = allTokens.top()[0];
         allTokens.pop(); allTokens.pop(); // remove oprtor, remove "("
         allTokens.push(to_string(performOperation(operands, oprtor)));
      } else {
         allTokens.push(currentToken);
      }
   }
   cout << allTokens.top() << endl;
   return 0;
}
```

Work hard preparing for midterms