[CS302-Data Structures]

Homework 2: Stacks

Instructor: Erin Keith

Teaching Assistants: Mustafa Solmaz, Huan Nguyen

Spring 2020 Semester

Section 1. Stack ADT – Overview

Data Items

The data items in a stack are of generic DataType. This means use should use templating and your Node class.

Structure

The stack data items are linearly ordered from the most recently added (the top) to the least recently added (the bottom). This is a LIFO scheme. Data items are inserted onto (pushed) and removed from (popped) the top of the stack.

Operations

**Constructor**. Creates an empty stack.

**Copy constructor**. Initializes the stack to be equivalent to the other Stack object parameter.

**Overloaded assignment operator**. Sets the stack to be equivalent to the other Stack object parameter and returns a reference to the modified stack.

**Destructor**. Deallocates the memory used to store the stack.

**push (const DataType& newDataItem)**

Requirements: Stack is not full.

Results: Inserts newDataItem onto the top of the stack.

**DataType peek()**

Requirements: Stack is not empty

Results: returns a copy of the value of the most recently added (top) data item from the stack

**void pop()**

Requirements: Stack is not empty

Results: Removes the most recently added (top) data item from the stack and returns the value of the deleted item.

**void clear()**

Requirements: None

Results: Removes all the data items in the stack.

**bool isEmpty() const**

Requirements: None

Results: Returns true if the stack is empty. Otherwise, returns false.

**bool isFull () const**

Requirements: None

Results: Returns true if the stack is full. Otherwise, returns false.

Section 2. Implementation Approaches

As in the class, we consider both

* Array-based implementations
* Linked-List implementations

**For this assignment, create the link-based implementation!**

Section 3.

Programming Exercise 1

We commonly write arithmetic expressions in the so-called infix form. That is, with each

operator placed between its operand, as below

(5+6)\*(4/3)

Although we are comfortable writing expressions in this form, infix form has the

disadvantage that parentheses must be used to indicate the order in which operators

are to be evaluated. These parentheses, in turn, complicate the evaluation process.

Evaluation is much easier if we can simply evaluate operators from left to right.

Unfortunately, this evaluation will not work with the infix form of arithmetic expressions.

However, it will work if the expression is in postfix form. In the postfix form of an arithmetic

expression, each operator is placed immediately after its operands. The expression

above is written in postfix form as

56+43/\*

Note that both forms place the numbers in the same order (reading from left to right).

The order of the operators is different, however, because the operators in the postfix form

are positioned in the order that they are evaluated. The resulting postfix expression is hard

to read at first, but it is easy to evaluate programmatically. We will do so with stacks.

Suppose you have an arithmetic expression in postfix form that consists of a sequence of

single digit, nonnegative integers and the four basic arithmetic operators (+,-,\*,/). This

expression can be evaluated using the following algorithm in conjuction with a stack of

floating-point numbers.

Read the expression character-by-character. As each character is read in:

* If the character corresponds to a single digit number (characters ‘0’ to ‘9’), then

push the corresponding floating-point number onto the stack.

* If the character corresponds to one of the arithmetic operators (characters ‘+’, ‘-

‘, ‘\*’, ‘/’), then

* + Pop a number off of the stack. Call it operand1.
  + Pop a number off of the stack. Call it operand 2.
  + Combine these operands using the arithmetic operator, as follows
    - Result = operand2 operator operand1
  + Push result onto the stack.

When the end of the expression is reached, pop the remaining number off the stack. This

number is the value of the expression. Applying this algorithm to the arithmetic expression

34+52/\*

Results 17.5 as expected.

Exercise 1.

Assignment 1.

Create a program that reads the postfix form of an arithmetic expression, evaluates it, and outputs the result. Assume that the expression consists of **single-digit**, nonnegative integers (‘0’ to ‘9’) and the FIVE basic arithmetic operators (‘+’,’-‘,’\*’,’/’,’^’) (note to correctly handle ‘^’). Further assume that the arithmetic expression is input from the keyboard with all the characters separated by white space on one line. Save your program in a file called postfix.cpp