Semester Project Part 1: Stacks and Queues

Data Structures and Analysis of Algorithms, akk5

Objectives

- To strengthen student's knowledge of C++ programming
- To give student experience reading and parsing strings of commands
- To give student experience in writing Data Structures for data types

Instructions

For this assignment you must write a program that implements and manages a collection of stacks and queues. For simplicity, each stack and queue will work with the say type of data, a node that stores entries comprised of a string representing a user id and an integer representing the user's ticket number. In addition, each stack and queue should store a string that represents its container name and an integer representing the number of entries currently in the stack or queue.

Your program should implement a command line (text-based interface) capable of handling the following commands:

exit – exits the program

load *<file>* - parses the contents of the file as if they were entered from the command line

display – displays a list of the saved stacks and queues. Should include the container type, the container name, and the current number of entries

create stack *<container>* - creates a new stack labeled container. Should inform the user on a failure.

create queue *<container>* - creates a new queue labeled container. Should inform the user on a failure.

find *<container>* - finds the specified container and displays its type and current number of entries. Should inform the user on a failure.

remove *<container>* - Removes the specified container. Should inform the user of a failure.

push *<uid> <ticket number> into <container>* - Inserts the specified uid, ticker number pair into the specified container. Should inform the user of a failure

peek *<container>* - displays the top most element of the container. Should inform the user of a failure.

pop *<container>* - removes the top most element of the container. Should inform the user of a failure.

mpop *<container> <n>* - performs the pop command n times; i.e. it removes the top n elements of the container. Should inform the user of how many elements were successfully removed from the container.

Guidance

Parsing text can be a frustrating part of any programming assignment and is probably more challenging than implementing the actual doubly linked list class and its methods. Although C++ supports multiple approaches to handling this challenge, I suggest the following approach.

First, forego the use of the >> operator in conjunction with any *istream* you might consider using (this would be *cin* and an input file stream for this program); instead, read the entirety of each line using a call to *getline*. We will break the data into its different parts afterwards using a process known as tokenization.

Once you have successfully read a command in a string, convert the string in a *stringstream* for further processing. The *stringstream* is probably new to most of you, but if you are comfortable working with streams it is easy enough to understand.

Stingstream is accessed by including sstream (*#include <sstream>*); since *stringstream* is in the std namespace, make certain you place a using clause in your code as well (using *std::stringstream*;).

You can convert a string to a *stringstream* as part of declaring the variable; as an example, the line of code below creates a *stringstream* labeled ss containing the contents of the string variable str:

```
stringstream ss(str);
```

At this point, any function, method, or operator that works with an *istream* will work with the *stringstream ss*. Of interest is a variation of the *getline* function. When tokenizing strings, it is necessary to break them apart based on a given delimiter character. The *getline* function supports a third argument that is often used to specify such a delimiter. As an example, the line of code below reads the first word of the *stringstream* labeled ss into the string variable cmd:

```
getline(ss,cmd, ‘ ‘);
```

For this assignment, it will also be necessary to convert a string into an integer. Since C++ 2011, this has been accomplished using the function *stoi*. The *stoi* function attempts to convert a string into an integer, returning the integer or throwing an exception if the conversion fails. The following block of code is an example of using *stoi* to convert the contents of the string str into an integer name sec:

```
try {
        sec = stoi(str);
} catch (…)
{
        // failed to convert the string to an integer, handle that failure here
}
```

You can also use the >> operation to tokenize the strings; however you will need to set the *ios::failbit* so you can detect when >> fails because of type mismatch or an unexpected end of line. To do this, use the following line of code:

```
ss.exceptions(ios::failbit);
```

Enabling the *ios::failbit* causes >> to throw an exception when it fails to extract information from the *istream* in question. Using this method will require you to place your entire input processing code block into a *try {} catch() {}* block. You will want to consider a code structure like

```
try
{

        ss >> x;

        if (x == “y”) {}

} catch (ios_base::failure)
{

        ss.clear();

}
```

Grading Breakdown

Point Breakdown	
Structure	12 pts
The program has a header comment with the required information.	3 pts
The overall readability of the program.	3 pts
Program uses separate files for main and class definitions	3 pts
Program includes meaningful comments	3 pts
Syntax	18 pts
Implements Class Node correctly	6 pts
Implements Class Stack correctly	6 pts
Implements Class Queue correctly	6 pts
Behavior	70 pts
Program handles all command inputs properly	
• Exit the program	7 pts
• Display list of stacks and queues correctly	7 pts
• Load a valid file	7 pts
• Create a new stack or queue	7 pts
• Find specified stack or queue	7 pts
• Remove specified stack or queue	7 pts
• Push a value into specified stack or queue	7 pts
• Peek from specified stack or queue	7 pts
• Pop from specified stack or queue	7 pts
• mPop from specified stack or queue	7 pts
Total Possible Points	**100pts**
Penalties	
Program does NOT compile	-100
Late up to 72 hrs	-10 per day
Late more than 72 hrs	-100

Header Comment

At the top of each program, type in the following comment:

/*

Student Name: <student name>

Student NetID: <student NetID>

Compiler Used: <Visual Studio, GCC, etc.>

Program Description:

<Write a short description of the program.>

*/

Example:

/*

Student Name: John Smith

Student NetID: jjjs123

Compiler Used: Eclipse using MinGW

Program Description:

This program prints lots and lots of strings!!

*/

Assignment Information

Due Date: 1/26/2020

Files Expected:

1. Main.cpp – File containing function main
2. Node.h - File containing the Node class definitions.
3. Stack.h – File containing the Stack class definitions.
4. Queue.h – File containing the Queue class definitions.
5. Node.cpp - File containing the code for the Node methods.
6. Stack.cpp - File containing the code for the Stack methods.
7. Queue.cpp - File containing the code for the Queue methods.