Data Structures and Algorithms I
Spring 2021
Programming Assignment #1

You are going to write a program that manipulates stacks and queues.

The program should ask the user for the name of an input text file and an output text file. The input file will contain a list of commands, one per line. Each command will direct the program to create a stack or a queue, to push a value onto a stack or a queue, or to pop a value from a stack or a queue. (Most modern sources use the terms "enqueue" and "dequeue" to indicate insertions into and deletions from queues, but we will use "push" and "pop" for both stacks and queues.)

The input file must contain one command per line. To be valid, a command must follow a very specific format. The command must consist of two or three words, depending on the type of command, separated by single spaces. The line must have no leading whitespace before the first word or trailing whitespace after the last word. For the purposes of this assignment, a "word" is defined to be a sequence of letters and digits, except for words representing values of integers or doubles, which might also contain a single negative sign, and/or in the case of doubles, a single decimal point. All commands (i.e., all lines in the text file) will end with a single Unix-style newline character ('\n').

The first word of each command must be "create", "push", or "pop" (all lowercase letters). The second word must be a valid name for a stack or a queue. The first character of every name must be one of 'i', 'd', or 's' (all lowercase), standing for integer, double, or string; this represents the data type that is stored in the particular stack or queue. The rest of the name must consist of only letters and digits. Both uppercase and lowercase letters are allowed, and the program should be case sensitive.

If the first word is "create", there must be a third word that will be either "stack" or "queue" (all lowercase letters). This represents the type of list being created. No two lists may have the same name. However, two lists storing different data types (e.g., one list storing integers and another storing strings) may have the same name other than the first characters (in this case, 'i' or 's'). There cannot be a stack and a queue of the same data type that share the same name.

If the first word is "push", there must be a third word representing a value to be pushed onto the stack or queue. This value must match the appropriate type of the specified stack or queue, and it must fit into a single variable (e.g., an integer will fit in a 32-bit signed int); if the value is a string, it must be a single word, as defined above (i.e., containing only letters and digits).

If the first word is "pop", there must not be a third word.

For the purposes of this assignment, you may assume that all commands in the input file will be valid! In other words, your program does not have to check that the lines in the text file represent valid commands; you may assume that this will be the case.

Your program should read and process the commands in the text file. After each command is read, your program should output the string "`PROCESSING COMMAND: `" followed by the text of the command and then a single Unix-style newline character. There should be exactly one space in between the ':' and text of the command. All output should be written to the specified output file.

If the command is a "create" command, and the name of the stack or queue that is specified has already been created (whether it was created as a stack or a queue), the program should output the string "`ERROR: This name already exists!`" using the same casing as is displayed here. If the name is new, the stack or queue should be created, and no additional output should be written.

If the command is a "push" command, and the specified name does not exist, the program should output the string "`ERROR: This name does not exist!`" using the same casing as is displayed here. If the stack or queue does exist, the push operation should be applied, and no additional output should be written.

If the command is a "pop" command, and the specified name does not exist, the program should output the string "`ERROR: This name does not exist!`" using the same casing as is displayed here. If the stack or queue does exist, but it is empty, the program should output the string "`ERROR: This list is empty!`" using the same casing as is displayed here. If the stack or queue does exist and is not empty, the pop operation should be applied, and the program should output "`Value popped: `", using the same casing as is displayed here, followed by the value that is popped from the stack or queue. There should be exactly one space in between the ':' and the value. For this assignment, a "pop" is assumed to both remove and return the appropriate value from the stack or queue.

You must follow these instructions exactly, so read them carefully! I will be comparing your output to my own using the "diff" command, available on Linux systems and Cygwin. *Your program's output should match mine exactly for all test cases.* If there are any differences, you will lose points.

Assume that the file `commands1.txt` exists in the current directory and contains the following text:

```
create i1 queue
create i1 queue
create i1 stack
create i2 stack
create s99 stack
push i1 50
push i1 100
push i2 -50
push i2 100
push s99 Hello
push s99 World
pop i2
pop s99
push s99 planet
pop i2
push i2 150
pop s99
pop s99
create d99 stack
push d99 0.123
push d99 -0.456
pop d99
pop s99
push dHelloWorld 0.5
pop dHelloWorld
push i2 200
pop i2
pop i1
push i1 150
push i1 200
pop i1
create dHelloWorld stack
create dHelloPlanet queue
push dHelloWorld 3.14
pop i2
pop i2
push dHelloWorld 3.1415
push dHelloPlanet -60.5
push dHelloWorld -1
pop dHelloWorld
pop dHelloPlanet
pop sR2D2
create sR2D2 queue
pop sR2D2
push sR2D2 123abcDEF
push sR2D2 G4H5I6j7k8l9
pop sR2D2
pop sR2D2
pop sR2D2
pop dHelloWorld
pop dHelloWorld
pop dHelloWorld
```

Then a sample run of your program might look like this:

```
Enter name of input file: commands1.txt
Enter name of output file: output1.txt
```

After this run, the output file `output1.txt` should look *exactly* like this:

```
PROCESSING COMMAND: create i1 queue
PROCESSING COMMAND: create i1 queue
ERROR: This name already exists!
PROCESSING COMMAND: create i1 stack
ERROR: This name already exists!
PROCESSING COMMAND: create i2 stack
PROCESSING COMMAND: create s99 stack
PROCESSING COMMAND: push i1 50
PROCESSING COMMAND: push i1 100
PROCESSING COMMAND: push i2 -50
PROCESSING COMMAND: push i2 100
PROCESSING COMMAND: push s99 Hello
PROCESSING COMMAND: push s99 World
PROCESSING COMMAND: pop i2
Value popped: 100
PROCESSING COMMAND: pop s99
Value popped: World
PROCESSING COMMAND: push s99 planet
PROCESSING COMMAND: pop i2
Value popped: -50
PROCESSING COMMAND: push i2 150
PROCESSING COMMAND: pop s99
Value popped: planet
PROCESSING COMMAND: pop s99
Value popped: Hello
PROCESSING COMMAND: create d99 stack
PROCESSING COMMAND: push d99 0.123
PROCESSING COMMAND: push d99 -0.456
PROCESSING COMMAND: pop d99
Value popped: -0.456
PROCESSING COMMAND: pop s99
ERROR: This list is empty!
PROCESSING COMMAND: push dHelloWorld 0.5
ERROR: This name does not exist!
PROCESSING COMMAND: pop dHelloWorld
ERROR: This name does not exist!
PROCESSING COMMAND: push i2 200
PROCESSING COMMAND: pop i2
Value popped: 200
PROCESSING COMMAND: pop i1
Value popped: 50
PROCESSING COMMAND: push i1 150
PROCESSING COMMAND: push i1 200
PROCESSING COMMAND: pop i1
Value popped: 100
PROCESSING COMMAND: create dHelloWorld stack
PROCESSING COMMAND: create dHelloPlanet queue
PROCESSING COMMAND: push dHelloWorld 3.14
PROCESSING COMMAND: pop i2
Value popped: 150
PROCESSING COMMAND: pop i2
ERROR: This list is empty!
PROCESSING COMMAND: push dHelloWorld 3.1415
PROCESSING COMMAND: push dHelloPlanet -60.5
PROCESSING COMMAND: push dHelloWorld -1
PROCESSING COMMAND: pop dHelloWorld
Value popped: -1
PROCESSING COMMAND: pop dHelloPlanet
Value popped: -60.5
PROCESSING COMMAND: pop sR2D2
ERROR: This name does not exist!
PROCESSING COMMAND: create sR2D2 queue
PROCESSING COMMAND: pop sR2D2
ERROR: This list is empty!
PROCESSING COMMAND: push sR2D2 123abcDEF
PROCESSING COMMAND: push sR2D2 G4H5I6j7k8l9
PROCESSING COMMAND: pop sR2D2
Value popped: 123abcDEF
PROCESSING COMMAND: pop sR2D2
Value popped: G4H5I6j7k8l9
PROCESSING COMMAND: pop sR2D2
ERROR: This list is empty!
PROCESSING COMMAND: pop dHelloWorld
Value popped: 3.1415
PROCESSING COMMAND: pop dHelloWorld
Value popped: 3.14
PROCESSING COMMAND: pop dHelloWorld
ERROR: This list is empty!
```

I will provide links to this example's input file and the output file from the course website. When I test your programs, however, I will use a few different test cases that I will not provide, including at least one that will be much longer (the longest test case will contain at least one hundred thousand pseudo-randomly generated commands).

Typically, if you were to implement a program like this in C++, you would likely use the provided C++ *list* class for everything. In other words, you would use it for stacks, for queues, and possibly for storing lists of stacks and queues. (You could use other provided classes to store collections of stacks and queues that would allow you to access them more efficiently, but we haven't covered those data structures yet.) Assuming you used the provided C++ *list* class for everything, you would probably want to have three separate high-level lists for the three different data types that you will be dealing with; e.g., one list would store all stacks and queues holding integers (or pointers to all stacks and queues holding integers).

While this would be completely reasonable, I am going to require you to create your own data structures for stacks and queues. The purpose of this assignment is not just to make sure that you understand how to *use* these data structures, but to also make sure that you understand how to *implement* these data structures, and also to give you experience with several advanced aspects of C++ including templates, inheritance, abstract classes, and polymorphism. In fact, you will need to mix these concepts together to really implement this well, and it can get confusing!

I suggest you create an abstract base class called *SimpleList* that provides some aspects of singly linked list functionality. The base class should include protected member functions that provide the ability to insert a node at the start of the list, to insert a node at the end of the list, and to remove a node from the start of the list. You should probably include a private nested class called *Node*, which contains one field for data and another to point to the next node. See the textbook's implementation of their *List* class as an example of something similar. The base class should maintain pointers to the start and end of the singly linked list; you can decide if you want to include header and trailer nodes (a.k.a. dummy nodes or sentinel nodes) at the start and end of each list. The base class should also include a private data member to store the name of the list and a public member function to retrieve it. The base class should also include two public pure virtual member functions for push and pop. The implementations should be in derived classes, and each can be implemented as a simple, one-line member function that calls the appropriate member function of the base class (to insert at the start or end of the list, or to remove from the start of the list).

You should implement two derived classes named *Stack* and *Queue*. You should use templates so that you only need to code the base class and the derived classes once. Of course, I am leaving out a lot of details that you will have to figure out; e.g., even the constructors, which in my own implementation accept the name of the stack or queue, can be syntactically confusing.

In terms of the program functionality, other than class member functions, I personally used two functions to open the input and output files, a few functions for syntax checking (not required for your programs, so don't bother with this), one template function to search for a *SimpleList* with a specified name, and then one large function to parse the input file and process all the commands. Of course, this large function makes many calls to my other functions, including various member functions of my *Stack* and *Queue* classes. The large function utilizes three lists; one contains pointers to all stacks and queues of integers, another contains pointers to all stacks and queues of doubles, and the third contains pointers to all stacks and queues of strings. For these lists, you may, as I did, use the provided C++ *list* class. I'll even show you my declarations:

```
list<SimpleList<int> *> listSLi; // all integer stacks and queues
list<SimpleList<double> *> listSLd; // all double stacks and queues
list<SimpleList<string> *> listSLs; // all string stacks and queues
```

A new stack of integers can be created and added to the first list with lines like these:

```
SimpleList<int> *pSLi;
pSLi = new Stack<int>(listName);
listSLi.push_front(pSLi);
```

Remember that your program does not have to check if commands are valid; you can assume that they will be. My program performs the checks, and it required about 150 lines of code to get it right. The total length of my program is a bit over 500 lines (including blank lines and comments, which account for about half of the lines). So, my program probably contains about (500 – 150) / 2 = 175 lines of code that implement the required functionality of this assignment.

You are welcome to deviate from my suggestions if you wish, but you will lose points for anything in your code that I think is not elegant.

The time it takes to do a push onto or a pop from a stack or queue should be worst-case O(M), where M is the total number of stacks and queues with the same data type as the current stack or queue. This is because you will need to do a linear search through the general list of the appropriate data type to find the stack or queue with the specified name, or to determine that it does not exist. (It is possible to make this more efficient using hash tables or balanced binary search trees, which we will learn about later in the course, but don't worry about that.) To search through one of the general lists for a stack or queue with some specified name, you will probably need to rely on iterators. I suggest creating a template function so that you only need to write a single function to search through the appropriate list. Once the appropriate stack or queue is found, *the push or pop itself should be a worst-case constant time operation*. This would not be the case if you use a vector or other resizable array to store nodes in your list class. (You would be able to achieve average constant time operations, but not worst-case constant time operations.) I am requiring that you use singly linked lists, which ensure worst-case constant time for push and pop if implemented correctly.

Your program must be written in C++, and it must compile and run correctly using "g++" with either Cygwin (freely available for Windows) or Ubuntu (a popular distribution of Linux). I will test your program under one of these two environments. You may use C++11 features if you wish; if you do, depending on your compiler, you might have to specify the "-std=c++11" flag when compiling. You may use C++14 or C++17 features as well, as long as the default compiler that comes with either Cygwin of Ubuntu supports them.

Note: If you find yourself getting strange compiler errors involving templates, of course you may come to me with questions, but you might want to first check out this website, which I consider very helpful: *https://isocpp.org/wiki/faq/templates*

Your grade will depend not only on correctness, but also style, elegance, comments, formatting, adherence to proper C++ conventions, and anything else that I think constitutes good programming. You should include one comment at the top of your code with your name and a brief description of what the program does. You should also include comments above functions, class definitions, member functions (either by their declarations or their implementations), and anywhere else where you think the code is doing something that is not obvious. Comments should be brief and should not state something that is obvious. *This is an individual programming assignment.* It is fine to ask each other questions, but you should not share code.

Submit your program by e-mail sent to `carl.sable@cooper.edu`. Send your source code (but *no executables or object files*) as an attachment. I will compile and run your program and test it on my test cases. The assignment is due before midnight on the night of Tuesday, April 27. This is a tough assignment, so get started early!