Project 3 - Lists, Stacks, and Queues

CS 3358: Data Structures and Algorithms
Due 10/27/2021 11:59 pm CST

Project Instructions

Write code for each of the below questions. Starter code for the project will be made available the project folder in Canvas. Do not change the function signatures in the starter code. You may edit your code in whatever IDEs or text editors you prefer.

You must submit your final project code via Canvas for grading.

Responses will be graded on three axes: correctness, pseudocode and code logic, and code style. A majority of your grade is determined by how many test cases your code passes. The points awarded for each question break down as follows:

- Test Cases: 60%
  - We Will run your code against predefined test cases to check for correctness.
  - Any solutions that try to game the test cases will receive zero credit. For example, writing a linear search algorithm when we ask for binary search will result in zero credit for this portion of the grade, even if it passes the test cases.
- Pseudocode and Code Logic: 20%
  - For each question, you will be asked to write your pseudocode for your code in the comments. Writing pseudocode benefits you on two fronts: (1) it helps you plan your algorithm before you write your code, and (2) it allows us to give you credit in this category for your logic, even if the final code doesn't pass the "Test Cases". You may also write comments to explain your logic to the grader for any questions on which you're having trouble.
  - Again, any solutions that try to game the test cases or solutions that don't meet the requirements of the question will not receive credit.
- Code Style: 20%
  - We recommend referencing the Google C++ Style Guide, but you do not need to follow it exactly as long as your code style is consistent and readable.

I recommend getting started on the project early so you have time to ask questions on Slack or in office hours in case you get stuck. You may submit your code as many times as you like, up until the deadline. We will only grade your most recent submission.

Project Overview

This project will give you experience implementing a templated container class (the double-ended, doubly-linked list) and using it to implement two applications: a Reverse-Polish Notation (RPN) calculator and a call center simulation. Additionally, you will gain practice using a container of pointers which point to objects on the heap.

Starter Code

The starter code for this project contains nine files, three of which you should modify before submitting:

Part 1:

- **`dlist.h`**
  Header file for your `Dlist` class, which contains your double-ended, doubly-linked list implementation (Part 1). `Dlist` is a class template, which allows us to create a `Dlist` of doubles in the RPN calculator and a `Dlist` of pointers to `reqNode` structs in the call center simulation. Since `Dlist` is a class template, all of the function implementations need to be included in the header (.h) file instead of in a separate implementation (.cpp) file.
- **`list_test1.cpp`**
  Simple `Dlist` test case. You do not need to modify this file, but you can if you want to try out additional test cases.
- **`list_test1.out`**
  Expected output from running the simple Dlist test case. You do not need to modify this file.

Part 2:

- **`calc.cpp`**
  Your Reverse-Polish Notation calculator implementation (Part 2).
- **`calc_test1.in`**
  Sample RPN calculator input. You do not need to modify this file.
- **`calc_test1.out`**
  Expected output from running `calc.cpp` with the sample RPN calculator input. You do not need to modify this file.

Part 3:

- **`call.cpp`**
  Your call center simulation implementation (Part 3).
- **`call_test1.in`**
  Sample call center simulation input. You do not need to modify this file.

- **`call_test1.out`**
  Expected output from running `call.cpp` with the sample call center simulation input. You do not need to modify this file.

General Requirements

You must fully implement the `Dlist` class. You must use your `Dlist` class to implement both your stack in the RPN calculator and your queues in the call simulator. Note that the most common implementations of the calculator and simulator may not use all of a `Dlist`'s functionality.

Your three files will be tested independently. Be sure that they can be compiled separately, with no cross-file dependencies. In particular, do not assume that your `Dlist` will be used with your applications all the time. You must independently test each of your `Dlist` methods to ensure they compile correctly.

You may `#include <iostream>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other system header files or libraries may be included. For example, note that you may not use STL data structure libraries such as `<list>`, `<stack>`, or `<queue>`.

Input and/or output should only be done where it is specified. The starter code handles the input format for the RPN calculator.

You may not use `goto` or global variables.

Acknowledgements

This assignment is modified and reprinted with faculty permission from the University of Michigan.

Part 1 - Implementing DList (40 points)

Overview

The double-ended, doubly-linked list, or `Dlist`, is a templated container. The complete interface of the `Dlist` class is provided in `dlist.h`.

Program Requirements

Do not add any member functions or member variables to the `Dlist` class. Do not modify the class declaration.

The `Dlist` class defines the usual four maintenance methods: the default constructor, the copy constructor, the assignment operator, and the destructor. These maintenance methods are implemented for you in the starter code, and you should not modify them.

The `Dlist` class also defines the following five operational methods:

```
// EFFECTS: returns true if list is empty, false otherwise
bool IsEmpty() const;

// MODIFIES: this
// EFFECTS: inserts o at the front of the list
void InsertFront(const T &o);

// MODIFIES: this
// EFFECTS: inserts o at the back of the list
void InsertBack(const T &o);

// MODIFIES: this
// EFFECTS: removes and returns last object from non-empty list
//          throws an instance of emptyList if empty
T RemoveFront();

// MODIFIES: this
// EFFECTS: removes and returns last object from non-empty list
//          throws an instance of emptyList if empty
T RemoveBack();
```

Finally, the `Dlist` class defines three private utility methods in which to place code common to two or more of the maintenance methods:

```
// EFFECTS: called by constructors to establish empty list invariant
void MakeEmpty();

// EFFECTS: called by copy constructor/operator= to copy nodes
//          from a source instance l to this instance
void CopyAll(const Dlist &l);

// EFFECTS: called by destructor and operator= to remove and destroy
//          all list nodes
void RemoveAll();
```

Implement the five operational methods and three private utility methods in the provided file called `dlist.h`. You may not `#include` any other files in `dlist.h`, nor may you invoke any `using` directives, including `using namespace std`. We will test your `Dlist` implementation separately from the other components of this project, so it must work independently of the two applications described in Parts 2 and 3.

Implementation and Usage

To compile a program that uses a `Dlist`, you need only `#include "dlist.h"`, and you do not need to type `dlist.h` on the compiler command line. A simple test program called `list_test1.cpp` and the corresponding expected output `list_test1.out` are included in the starter code. You may compile and run the test program in Repl.it using the following command:

```
clang++-7 -pthread -std=c++17 -o list_test1 list_test1.cpp;
./list_test1
```

Part 2 - RPN Calculator (30 points)

Overview

The first application you must write is a Reverse-Polish Notation calculator. An RPN calculator is one in which the operators appear after their respective operands, rather than in between them. So, instead of computing the following:

```
(2 + 3) * 5
```

an RPN calculator would compute this equivalent expression:

```
2 3 + 5 *
```

RPN notation is convenient for several reasons. First, no parentheses are necessary since the computation is always unambiguous. Second, such a calculator is easy to implement given a stack. This is particularly useful, because it is possible to use the `Dlist` as a stack.

Program Requirements

The calculator is invoked with no arguments, and starts out with an empty stack. It takes its input from the standard input stream (`std::cin`), and writes its output to the standard output stream (`std::cout`). Here are the commands your calculator must respond to and what you must do for each:

<some number>	Number: A valid number has the following form: one or more digits [0 – 9] optionally followed by a decimal point and one or more digits. For example, 3 4.56 and 0.12 are all valid numbers, but -2, 1., and abc are not. A number, when entered, is pushed on the stack. This input is always valid. This operand is already implemented for you in the starter code.
+	Add: Pop the top two numbers off the stack, add them together, and push the result onto the top of the stack. This requires a stack with at least two operands.
-	Subtract: Pop the top two numbers off the stack, subtract the first number from the second, and push the result onto the top of the stack. This requires a stack with at least two operands.
*****	Multiply: Pop the top two numbers off the stack, multiply them together, and push the result onto the top of the stack. This

	requires a stack with at least two operands.
/	Divide: Pop the top two numbers off the stack, divide the second popped number by the first, and push the result onto the top of the stack. This requires a stack with at least two operands.
n	Negate: pop the top item off the stack, multiply it by -1, and push the result onto the top of the stack. This requires a stack with at least one operand.
d	Duplicate: Pop the top item off the stack and push two copies of the number onto the top of the stack. This requires a stack with at least one operand.
r	Reverse: Pop the top two items off the stack, push the first popped item onto the top of the stack and then push the second item onto the top of the stack (this just reverses the order of the top two items on the stack). This requires a stack with at least two operands.
p	Print: Print the top item on the stack to the standard output (`std::cout`), followed by a newline. This requires a stack with at least one operand and leaves the stack unchanged.
c	Clear: Pop all items from the stack. This input is always valid.
a	Print All: Print all items on the stack in one line, from top-most to bottom-most, each separated by a single space. The end of the output must be followed by a newline. This input is always valid and leaves the stack unchanged.
q	Quit: Exit the calculator. This input is always valid. This operand is already implemented for you in the starter code.

Note that the phrase "leave the stack unchanged" is not to be taken literally. It is okay to pop the top two operands off the stack for testing and, if there are any problems, push them back onto the stack (in the proper order) before reading the next command.

Each command is separated by whitespace. You may not assume that user input is always correct. There are three error messages to report:

1. If a user enters something other than one of the commands above, leave the stack unchanged, advance to the next whitespace character, and print the following message:

```
std::cout << "Bad input\n";
```

2. If a user enters a command that requires more operands than are present, leave the stack unchanged, advance to the next whitespace character, and print the following message:

```
std::cout << "Not enough operands\n";
```

3. If a user enters the divide command with a zero on the top of the stack, leave the stack unchanged, advance to the next whitespace character, and print the following message:

```
std::cout << "Divide by zero\n";
```

These error messages are already included in `main` in the starter code. All you need to do is throw the appropriate exception class (`emptyList`, `divZero`, or `badInput`) where required. Example usage:

```
if (stack.IsEmpty()) {
  throw emptyList();
}
```

Implementation and Usage

Implement your RPN calculator in the provided file called `calc.cpp`. In your implementation, you will use a `Dlist` of `doubles` (initialized in `main` in the starter code) as a stack. Your code must work correctly with any valid implementation of `Dlist`, not just your specific implementation.

A sample input for your RPN calculator program called `calc_test1.cpp` and the corresponding expected output `calc_test1.out` are included in the starter code. You may compile and run the calculator with the sample input in Repl.it using the following command:

```
clang++-7 -pthread -std=c++17 -o calc calc.cpp; ./calc < calc_test1.in
```

Here is a short example of the RPN calculator's usage:

```
2
3
4
+
*
```

```
p
14
+
Not enough operands
d
+
p
28
2
-
p
26
q
```

Part 3 - Call Center Simulation (30 points)

Overview

The second application you must write is a discrete-event simulator, modeling the behavior of a single reservation agent at Delta Airlines. When a customer calls Delta, they are asked to enter their SkyMiles number. Calls are then answered in priority order: customers who are Platinum Elite (those having flown 75,000 miles or more in the current or previous calendar year) have their calls answered first, followed by Gold Elite (50,000), Silver Elite (25,000), and finally "regular" customers.

We call this a discrete-event simulator because it considers time as a discrete sequence of points, with zero or more events happening at each point in time. In our simulator, time starts at "time 0", and progresses in increments of one. Each increment is referred to as a "tick". A discrete-event simulator is usually driven by a script of "independent events" plus a set of "causal rules".

Program Requirements

In our simulator, the independent events are the set of customers that place calls to the call center. These events are in a file. The first line of the file has a single entry which is the number of events (N) contained in the next N lines. Each of those N lines has the following format:

```
<timestamp> <Name> <status> <duration>
```

Each field is delimited by one or more whitespace characters. You may assume that the lines are sorted in timestamp-order, from lowest to highest. Timestamps need not be unique.

<timestamp>	An integer, zero or greater, that denotes the tick at which this call comes in.
<Name>	The name of the customer placing the call.
<status>	One of the following four strings: "none" – no special status "silver" – silver elite "gold" – gold elite "platinum" – platinum elite
<duration>	A positive integer, denoting the number of ticks required to service this call.

You may assume that the input file is semantically and syntactically correct. Your simulator must obtain this input file from the standard input stream `std::cin`.

Your simulator will maintain four queues, one for each status level. The simulation proceeds as follows (these are the causal rules):

- At the beginning of a "tick", announce it like this.

  ```
  Starting tick #<tick>
  ```

- Any callers with timestamps equal to that tick number are inserted into their appropriate queues. When a caller is inserted, you should print a message that exactly matches the following example (with the correct name and status):

  ```
  Call from Jeff a silver member
  ```

  Note: if two (or more) calls have the same timestamp, they should be printed in input file-order, not in priority-order.

- After any new calls are inserted into the call queues, the (single) agent is allowed to act using the following rules:
  - If the agent is not busy, the agent checks each queue, in priority order from Platinum to None. If the agent finds a call, the agent answers the call, printing a message that exactly matches the following example (with the correct name):

    ```
    Answering call from Jeff
    ```

    This will keep the agent busy for <duration> ticks.

  - If the agent was already busy at the beginning of this tick, the agent continues servicing the current client until the appropriate number of ticks have expired.

  - If the agent is not busy, and there are no current calls, the agent does nothing, and the clock advances. The program terminates only when all listed calls have been placed, answered, and completed.

Implementation and Usage

Implement your call center simulator in the provided file called `call.cpp`. In your implementation, you will use a `Dlist` of pointers to `reqNode` structs (initialized in `main`

in the starter code) as a queue. Your code must work correctly with any valid implementation of `Dlist`, not just your specific implementation.

A sample input for your RPN calculator program called `call_test1.cpp` and the corresponding expected output `call_test1.out` are included in the starter code. You may compile and run the calculator with the sample input in Repl.it using the following command:

```
clang++-7 -pthread -std=c++17 -o call call.cpp; ./call < call_test1.in
```

Here is the sample input file:

```
3
0 Andrew gold 2
0 Chris none 1
1 Brian silver 1
```

And here is the expected output produced by running the simulator using that input file:

```
Starting tick #0
Call from Andrew a gold member
Call from Chris a regular member
Answering call from Andrew
Starting tick #1
Call from Brian a silver member
Starting tick #2
Answering call from Brian
Starting tick #3
Answering call from Chris
Starting tick #4
```