Assignment 2: Postfix Calculator using Stacks, Arrays and Doubly Linked Lists

COL106: Data Structures and Algorithms, Semester-I 2023–2024

Submission Deadline: 5:00 PM 20 August 2023

Honor Code

Following is the Honor Code for this assignment.

- Copying programming code in whole or in part from another or allowing your own code (in whole or in part) to be used by another in an assignment meant to be done individually is a serious offence.
- The use of publicly available codes on the internet is strictly not allowed for this assignment.
- The use of ChatGPT and other AI tools is strictly forbidden for this assignment. If any student has been found guilty, it will result into disciplinary action.
- Collaboration with any other person or team would be construed as academic misconduct.
- Outsourcing the assignment to another person or "service" is a very serious academic offence and could result in disciplinary action.
- Sharing of passwords and login ids is explicitly disallowed in this Institute and any instance of it is academic misconduct. Such sharing only compromises your own privacy and the security of our Dept./Institute network.
- Please ensure that your assignment directories and files are well-protected against copying by others. Deliberate or inadvertent copying of code will result in penalty for all parties who have "highly similar" code. Note that all the files of an assignment will be screened manually or by a program for similarity before being evaluated. In case such similarities are found, the origin as well as destination of the code will be held equally responsible (as the software cannot distinguish between the two).

Introduction

In this assignment you will implement a reverse Polish notation calculator, also called a post-fix notation calculator. For more details about the post-fix notation visit this link.

In order to implement reverse Polish notation calculator, you need to first implement a stack. There are three parts of this assignment. In part (a) of the assignment you will implement stacks using fixed size arrays. In part (b) you will use dynamically sized arrays. In part (c) you will implement them using doubly linked lists.

You have to implement a class `Stack` which has member functions for standard mathematical operations, as well as some non-standard operators and additional commands. You will need to provide definitions for these member functions. We will judge the correctness of your implementation by creating objects of the class `Stack` and calling these member functions. For evaluation, some test cases will be made available to you to help you debug your program but all the "evaluation test cases" will not be made available to you. You need to ensure that your program works correctly **on all the inputs**, not just on the test case inputs.

1 Getting Started

In all three parts, you will be given a header file `stack_{a,b,c}.h` which contains the specification of the methods of the class `Stack_{A,B,C}` which you need to implement in `stack_{a,b,c}.cpp`. For Part (c), you will also be given header files for `Node` and `List` which you will use to implement Linked Lists using `node.cpp` and `list.cpp`. Note that you may **not** modify the header files. The starter code is available here.

The table 2 gives a detailed description of the methods of the `Stack` class which you need to provide definitions for.

You will also need to handle errors. For this purpose, you must use the `stdexcept` library and throw appropriate run-time errors if erroneous inputs are given to your program. There can be two types of errors:

1. Popping from an empty stack: Throw a `std::runtime_error` exception with the message `"Empty Stack"`.
2. Trying to use arithmetic operations on a stack without enough arguments: Throw a `std::runtime_error` exception with the message `"Not Enough Arguments"`.
3. Pushing to a full stack: In Part A, if the number of elements in the stack when pushing is already 1024, throw a `std::runtime_error` exception with the message `"Stack Full"`.
4. Division by zero: Throw a `std::runtime_error` exception with the message `"Divide by Zero Error"`.
5. Fetching element out of stack using get_element_from_top(...) or get_element_from_bottom(...): Throw a `std::runtime_error` exception with the message `"Index out of range"`.
6. Failure of New: In Part B and C, if the `new` operator fails, throw a `std::runtime_error` exception with the message `"Out of Memory"`.

You are **not** allowed to use any STL data structures for this assignment. If we find you using them, you will get a zero in the entire assignment.

2 Post-fix notation calculator using stacks on fixed-sized arrays (20 marks)

Implement a stack of ints using an array of size 1024 elements. Using this stack, implement an extended post-fix notation calculator.

3 Post-fix notation calculator using stacks on dynamically-sized arrays (40 marks)

Implement a stack of ints using a dynamically sized array. A dynamically sized array is one whose size grows or shinks according to the number of elements it holds. The stack should be able to grow indefinitely (up to the memory available on the computer), and shrink down to 1024 elements if the number of elements is small. The specifications of the implementation are as follows:

- The minimum size of the array is 1024 elements.
- If the number of elements of the stack is equal to the size of the array and there is a push, double the size of the array.
- Similarly, shrink the array if the number of elements of the stack drop beyond a threshold (to be decided by you).

Your task is to implement a stack using the dynamically sized arrays you use *efficiently*. In particular, the typical stack operations should be $O(1)$ on average, as required. Note that we will rigorously test this part with adversarial cases to ensure your implement is efficient in *all* cases and not just some cases.

Commands	
Function	Description
`void push(int data)`	Push the argument on the stack
`int pop()`	Pop and return the top element of the stack
`int get_element_from_top(int idx)`	Return the element at index idx from the top
`int get_element_from_bottom(int idx)`	Return the element at index idx from the bottom
`void print_stack(bool top)`	Print the elements of the stack, one number per line. If top = 1, start printing from the top, else bottom.
`int add()`	Pop and add the top two elements of the stack and push the result back onto the stack
`int subtract()`	Pop the top two elements from the stack. Subtract the top element from the second element and push the results back onto the stack
`int multiply()`	Pop two elements from the stack and multiply them and push the result back onto the stack
`int divide()`	Pop two elements from the stack. Divide the second from the top element by the top element of the stack. Push the *floor* of the result, i.e. $\lfloor res \rfloor$ back onto the stack.
`int* get_stack()`	Return a pointer to the array used for the stack (for Part 1 and 2)
`List* get_stack()`	Return a pointer to the Linked List used for the stack (for Part 3)
`int get_size()`	Return the size of the stack

Bonus (5 Marks)

If you are confident your implementation has an amortized $O(1)$ cost for typical stack operations (push, pop), try to prove it! Submit a 1-2 page typed PDF along with your submission (see instructions 5), providing a formal proof that your solution is indeed efficient.

4 Post-fix notation calculator using stacks made using doubly linked lists (40 marks)

Implement a stack of ints using a doubly linked list. Your stack should be able to grow indefinitely (up to the memory available on the computer) as more elements are pushed onto it. Using this stack, implement an extended post-fix notation calculator according to the specifications given in Table 2.

In this part, you not only need to provide implementations of the stack functions in `stack.cpp`, but also of the linked list in `list.cpp` and of the node class in `node.cpp`. To test your program, you will need to compile and *link* all of these files. Assuming that we provide you a tester code in a `main.cpp` file, you can compile the program as:

```
$ g++ main.cpp node.cpp list.cpp stack_a.cpp stack_b.cpp stack_c.cpp -o test.out
```

Your program should dynamically re-allocate and free memory so that the stack can grow to large sizes up to the memory available to your program. The submission will be stress-tested with limited memory sizes and tested for correctness using inputs of different sizes.

5 Submission Instructions

Please read these instructions **very carefully**. You are required to submit the following 5 files in total on **Gradescope**:

- For Part A, `stack_a.cpp`
- For Part B, `stack_b.cpp`
- For Part C, `stack_c.cpp`, `node_c.cpp`, and `list.cpp`

If you have attempted the bonus part, you may also submit a PDF file named `proof.pdf`. **Carefully adhere to the submission instructions, failure to comply will result in a 10 % penalty.**

We don't want you to stay up late at night as it is detrimental to your health, so please note the submission deadline is **5 PM**, and not midnight.

6 Points to Note

1. For each part, read the instructions carefully and use only the data structure specified. You will receive a zero in the assignment if you try to use STL data structures and functionalities, or try to get around the assignment (eg. not using linked lists in Part 3)
2. Follow good memory management practices. We will stress test your submissions. There should be no memory leak, otherwise your code may fail on the evaluation test cases.