Assignment Stacks: Applications of Stacks

COSC 2336: Data Structures and Algorithms

Spring 2024

Objectives

- Implement Stack API functions.
- Practice using Stacks as data types and containers and their use for common applications.
- More practice writing linked list manipulations.
- More examples and practice writing recursive functions.
- Example of writing regular C++ functions as template functions and their uses.

Description

In this assignment, you will be using the `Stack` abstract data type we developed for this unit and discussed in our lectures, to implement 4 functions that use a stack data type to accomplish their algorithms. The functions range from relatively simple, straight forward use of a stack, to a bit more complex. But in all 4 cases, you should only need to use the abstract stack interface functions `push()`, `pop()`, `top()`, and `isEmpty()` in order to successfully use our `Stack` type for this assignment and the function you are asked to write. Though before we get to using stacks we have a bit more practice implementing API methods of the `Stack` abstraction, and more practice with manipulating linked lists.

As with the previous assignment, our textbooks also give examples of a `Stack` ADT Malik 6th edition pg. 1152, and Shaffer 3.2 C++ edition, pg. 121. The `Stack` ADT we give and complete in this assignment is similar in structure to the `List` ADT from the previous assignment. And array based (`AStack`) and linked list based (`LStack`) implementations of the `Stack` abstraction are again given and used for this assignment.

Recall from our materials this unit that a stack is a first-in-first-out (FIFO) data structure. You should be able to understand the the performance trade offs a bit now of using an array to implement the stack, or of using a linked list. For example, since we push values only on 1 end, if we are using an array, we don't want to be using index 0 of the array to insert new values on the stack for the `push()` and `pop()` operations. If we did that, we would have to always shift items up and down whenever we pushed and popped items to/from the stack, meaning both operations would be $\mathcal{O}(n)$. Instead we should be using the end of the array to push on items and pop them off from the stack. As long as we haven't filled up the capacity of our array memory, the operations are then constant time $\mathcal{O}(1)$ operations, as long as we don't have to grow the size of the static array. In which case, when that happens, the push operation does become $\mathcal{O}(n)$ since we have to copy all values from the old memory to the new memory. However, the pop operation would always be constant time if we are pushing and popping from the end of the array.

However, for a singly linked list, we could use either end for the `push()` operation and it would be constant time, as long as we keep pointers both to the `front` and `back` like we did with our `List` in a previous assignment. However, if the list is singly linked, the `pop()` operation is problematic if we try and use the end of the linked list. This is because, if you want to remove the last node of a singly linked list, you need a pointer to the node before the last one so you can make it the new `back` node after removal. This requires a search through the list, and thus `pop()` becomes $\mathcal{O}(n)$ if we use the back of the list. So for a linked list, we want to push and pop from the front of the list, because both operations can be done in $\mathcal{O}(1)$ constant time in that case.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/assg09-stack-functions-tests.cpp`	Unit tests of the functions using `Stack` data structures you will write for this assignment.
`src/assg09-AStack-tests.cpp`	Unit tests of the array based `AStack` implementation of the `Stack` API
`src/assg09-LStack-tests.cpp`	Unit tests of the link list based `LStack` implementation of the `Stack` API
`include/stack-functions.hpp`	Header file for the functions you will write that use stack for various applications.
`include/Stack.hpp`	Header file of the ADT base `Stack` class that defines the `Stack` interface / abstraction
`include/AStack.hpp`	Header file of the concrete array based implementation of the `Stack` abstract data type
`include/LStack.hpp`	Header file of the concrete linked list based implementation of the `Stack` abstract data type
`include/Node.hpp`	Header file of the Node structure used by the linked list implementation
`src/stack-functions.cpp`	Implementation file of the functions using stacks to implement the applications for this assignment.
`src/Stack.cpp`	Implementation file of common methods of the `Stack` base class
`src/AStack.cpp`	Implementation file for the `AStack` member functions that implement the concrete array based `Stack`
`src/LStack.cpp`	Implementation file for the `LStack` member functions that implement the concrete linked list based `Stack`

This week you will be adding in the `push()` and `pop()` member functions of the `LStack` class. This is kind of a warm up, but the intention is to get a bit more practice with manipulating linked lists. Since we need to add and remove values from the front of the linked list for an efficient stack implementation, these methods are pretty similar to the previous `insertFront()` and `deleteFront()` methods of the previous `List` assignment.

But the main work of the assignment will be writing several functions in the `stack-functions.[hpp|cpp]` files, where you will use `Stack` instances to solve various problems and applications.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment. Make sure to open the cloned folder and restart inside of the correct Dev Container.
3. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
4. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure that you link each issue you create with the `Feedback` pull request on your GitHub classroom.

Assignment Tasks

Task 1: Implement `LStack` missing `push()` and `pop()` Methods

As usual make sure that you create Task 1 issue on GitHub and have linked the issue to the `Feedback` pull request.

Also as usual, you should be practicing incremental development. Add the function prototypes for `push()` and `pop()` to the `LStack.hpp` header file, and add in stub implementations that do nothing (these are both void functions) in the `LStack.cpp` implementation file. You should be able to uncomment the first set of test cases in the `test-LStack.cpp` file and compile the project and run the tests before beginning to implement push and pop.

The `push()` function as we mentioned above is doing the same as the `insertFront()` function from a previous assignment. You are given a value of template type `T` to be pushed onto the stack. This should be a `const` reference parameter (see the `Stack.hpp` abstract base class definition for the signature of the function).

You need to simply put the given value on the front of the linked list of values being maintained by `LStack`. The pointer to the front of the list is called `topNode` instead of `front` in the `LStack` class, because this indeed is the top node of the stack of items that will be maintained. You should implement push as follows:

1. Create a new `Node` dynamically.
   - Initialize the nodes value to the value given to the `push()` function.
   - Initialize the `next` pointer to be `nullptr` for now.
2. Test if the stack is empty using the `isEmpty()` member method.
   - If the stack is empty, this new node becomes the only node in the stack, so point `topNode` to the node you just created.
   - If the stack is not empty, then point the new node to the current `topNode` of the stack, and repoint `topNode` to be this new node that you are pushing.
3. Don't forget to update the `size` of the stack, which has grown by 1 now.

Likewise `pop()` needs to be popping from the front / top of the list, like the `deleteFront()` function from a previous assignment. If you check the `Stack.hpp` abstract definition, you will see that `pop()` is a void function that takes no input parameters. Implement `pop()` by doing the following:

1. You should check if the stack is empty first, and if it is throw an exception of type `StackEmptyException` to indicate an illegal attempt to pop from an empty stack.
2. Otherwise, make a temporary pointer and point it to the current `topNode`
3. Move the `topNode` to point to the next node in the list.
4. Delete the old top node that is pointed to by your temporary pointer since it is no longer needed and we need to return its memory back to the heap.
5. Don't forget to update the `size` of the stack, which has now been reduced by 1.

As with the `LList` in a previous assignment, some of the constructors reuse the `push()` method to construct new stacks. So once you define your `push()` function, you also need to uncomment the calls in the array and copy constructor to `push()` so that your stack constructors will work.

Once you are satisfied with your work, and you are passing all of the tests now in the `test-LStack.cpp` and `test-AStack.cpp` test files, commit your changes and push them to the `Feedback` pull request of your GitHub repository.

Task 2: Implement `doParenthesisMatch()` Function

Perform the usual steps before starting task 2, and make sure that you have created Issue 2 in GitHub to work on this task. The first test case in the `test-stack-functions.cpp` file has a set of tests for this task 2 `doParenthsisMatch()` function.

In the second task, we will write a function that will check if a string of parenthesis is matching. This will be a regular C/C++ function (it is not a member of any of the `Stack` classes). You should put the function prototype in the `stack-functions.hpp` file, and the implementation of this function in `stack-functions.cpp`. This function takes a `string` as its input parameter, and it returns a `bool` boolean result of `true` or `false`. Strings will be given to the function of the format "(()((()))))", using only opening "(" and closing ")" parenthesis. Your function should be named `doParenthesisMatch()`.

The algorithm to check for matching parenthesis using a stack is fairly simple. A pseudo-code description might be

```
for each charcter c in expression
do
    if c is an open paren '('
        push it on stack

    if c is a close paren ')':
    then
        if stack is empty
            answer is false, because we can't match the current ')'

        else
            pop stack, because we just matched an open '(' with a close ')'
```

```
    endif
done
```

Your function will be given a C++ string class as input. It is relatively simple to parse each character of a C++ string. Here is an example for loop to do this:

```
s = "(())";
for (size_t index = 0; index < s.length(); index++)
{
    char c = s[index];

    // handle char c of the string expression s here
}
```

Once you are satisfied with your function and you are passing the tests in `test-stack-functions.cpp`, commit your work and push it to the `Feedback` pull request of your GitHub repository.

Task 3: Implement `decodeIDSequence()` Function

In task 3, we will also write a regular C/C++ function. This function will decode a string expression. Given a sequence consisting of 'I' and 'D' characters, where 'I' denotes increasing and 'D' denotes decreasing, decode the given sequence to construct a "minimum number" without repeated digits.

An example of some inputs and outputs will make it clear what is meant by a "minimal number".

Sequence		Output
IIII	→	12345
DDDD	→	54321
ID	→	132
IDIDII	→	1325467
IIDDIDID	→	125437698

If you are given 4 characters in the input sequence, the result will be a number with 5 characters, and further only the digits '12345' would be in the "minimal number" output. Each 'I' and 'D' in the input denotes that the next digit in the output should go up (increase) or go down (decrease) respectively. As you can see for the input sequence "IDI" you have to accommodate the sequence, thus the output goes from 1 to 3 for the initial increase, because in order to then decrease, and also only use the digits '123', we need 3 for the second digit so the third can decrease to 2.

Take a moment to think how you might write an algorithm to solve this problem? It may be difficult to think of any solution involving a simple iterative loop (though a recursive function is not too difficult).

However, the algorithm is relatively simple if we use a stack. Here is the pseudo-code:

```
for each index, character c in input sequence
do
    push character index+1 onto stack (given 0 based index in C)

    if we have processed all characters or c == 'I' (an increase)
    then
        pop each index from stack and append it to the end of result
    endif
done
```

Your function should be named `decodeIDSequence()`. It will take a string of input sequence, like "IDI" as input, and it will return a new string, the resulting minimal number. Notice we will be constructing a string to return here, so simply start with an empty string like `string result = ""` and append the digits to the end when you pop them from the stack as described.

When you have the function working and are satisfied, commit your changes and push them to the `Feedback` pull request of your repository for this assignment.

Task 4: Implement `insertItemOnSortedStack()` Template Function

In the fourth and fifth tasks, you will write two functions that will be able to sort a stack. First of all, you should write a simpler method that, given an already sorted stack as input, and an item of the same type as the stack type, the item should be inserted into the correct position on the sorted stack to keep it sorted. For example, the stacks will be sorted in ascending order, where the item at the bottom of the stack is the smallest value, and the item at the top is the largest, like this:

```
top:[ 8, 7, 5, 3, 1, ]:bottom
```

If we call the function to insert a 4 into this sorted stack, the result should be:

```
top:[ 8, 7, 5, 4, 3, 1 ]:bottom
```

Your function should be called `insertItemOnSortedStack()`. This function takes an item as its first parameter, and a reference to a Stack as its second parameter. You should create and use another temporary stack in your function in order to accomplish the task. The pseudo-code to accomplish this insertion is as follow:

```
given and inputStack
and you create a temporaryStack for this algorithm

while top of inputStack > item we want to insert
do
    pop topItem from inputStack
    push topItem onto the temporaryStack
done

at this point, items on inputStack are <= to the item we want to insert
so push item onto inputStack

now put items back from temporaryStack to original inputStack
while temporaryStack is not empty
do
    pop topItem from temporaryStack
    push topItem onto the inputStack
done
```

The `insertItemOnSortedStack()` is a `void` function, it doesn't return an explicit result. You instead need to pass in 2 parameters, an integer parameter, and a reference to a `Stack<T>&` which the function expects to be an already sorted stack. It is important you pass in the `Stack<T>` by reference, because you will be inserting the item onto this stack in the correct sorted location, and in order for your result to be returned back to the call, the stack must be passed in by reference. Notice we said you should pass in a `Stack<T>` as the parameter, not a more concrete `LStack<T>` or `AStack<T>`. This is an example of inheritance and polymorphism of our classes. Both of the concrete stacks are types of the abstract `Stack` class. So by saying you accept a `Stack<T>` of input, it means any concrete instance of the `Stack<T>` interface can be passed into your function.

As a final wrinkle, you will turn this function into a template function. Regular functions, like class member functions, can be templatized, as we discussed in our unit on C++ templates. We would like your function to be able to insert onto a a stack of strings `Stack<string>`, or a stack of characters, or of any time. So to pass all of the unit tests, some of which call your function with stacks of `int` values, and some with stacks of `string` values, you will have to turn it into a C++ template class. **Hint**: this means you need something like

```
template <class T>
```

before your function prototype and your function implementation.

NOTE: Also once you templatize this first function there is a bit of a kludge you need to do to get your template function(s) to compile correctly. At the bottom of the `stack-functions.cpp` file you will see the following code commented out:

```
/**
 * @brief Cause specific instance compilations
 *
```

```
 * This is a bit of a kludge, but we can use normal make dependencies
 * and separate compilation by declaring template class Stack<needed_type>
 * here of any types we are going to be instantiating with the
 * template.
 *
 * https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl
 * https://isocpp.org/wiki/faq/templates#separate-template-class-defn-from-decl
 */
template void insertItemOnSortedStack<int>(int item, Stack<int>& sortedStack);
template void insertItemOnSortedStack<string>(string item, Stack<string>& sortedStack);

template void sortStack<int>(Stack<int>& aStack);
template void sortStack<string>(Stack<string>& aStack);
```

You should uncomment the lines for the `insertItemOnSortedStack` definitions here. These will cause the compiler to include versions of your template function to work on integer and string stacks, so that everything will link correctly.

Once your work passes the tests in `test-stack-functions.cpp` for the `insertItemOnSortedStack()` function, commit your work and push it the `Feedback` pull request of your repository.

Task 5: Implement `sortStack()` Template Function

Once you have your `insertItemOnSortedStack()` template function working, it is even easier to use this function to create a `sortStack()` function. We could implement this function again using a temporary stack, but for this fourth and final function I want you instead to create a recursive function. A recursive function in this case is going to work in essentially the same way, but we will be using the OS/system function call stack implicitly to perform the algorithm, rather than explicitly creating and using our own temporary stack.

Create a function called `sortStack()`. This function should take a `Stack<int> &` (a reference to a Stack of `<int>` types) as its only parameters. You will later templatize this function as well, but all of the tests of sortStack() use stacks of strings, so get it working first for strings, then try and templatize the function. This is a `void` function, it doesn't return a result, but it implicitly causes the stack it is given to become sorted.

The function, as the name implies, will take an unsorted stack, and will sort them in the same order we used previously, e.g. in ascending order with the smallest item at the bottom of the stack, and the largest at the top. The pseudo-code to accomplish this using a recursive algorithm is as follows:

```
given inputStack as an input parameter

# the base case
if inputStack is empty, do nothing (return)

# the general case
take top item from inputStack and save it in a local variable

call sortStack(inputStack) recursively on this now smaller stack

# on return, the stack should be sorted, so
insertItemOnSortedStack(my item I popped, inputStack)
```

Once you have it working for type stacks, also templatize your `sortStack()` function, so that it will actually work to sort a `Stack` of any type. Also remember to uncomment the lines at the bottom of the `stack-functions.cpp` file so that your `sortStack()` will compile and link correctly.

Once you are satisfied with your work, commit it and push it to the `Feedback` pull request of your repository.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 40 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 10 points are awarded for completing each of the 6 tasks. However you should note that the autograder awards either all point for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U09-1 Implementing Stacks
- Lecture U09-2 Applications of Stacks
- Data Structures - Full Course Using C and C++ The videos 2:43:09 Introduction to stack through 4:14:00 Infix, Prefix and Postfix are all excellent introductions to the fundamentals and more advanced aspects of Stacks.
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates