Lab 06: Stack and Deque

Overview

This lab will practice stacks and deques. More specifically, we implement the stack using a dynamic array, and the deque using a doubly linked list.

Stack

Stacks are data structures that follow the LIFO (last in, first out) rule to control its push and pop behavior.

On Lab 04, you implemented a queue on top of a dynamic array. It was able to efficiently insert at the back of the queue, but when dequeuing, the entire list had to be shifted. For today's lab, you will build on the dynamic array queue from Lab 04 to create a stack on top of a dynamic array. This time, popping (equivalent to dequeuing) can be done efficiently by tracking the top of the stack which works similar to the back of the queue from last week.

Hints: You are given a functioning dynamic array queue, the solution to Lab 04, in Mimir's **starter code**. You can modify its existent behavior to get an efficient stack while focusing mostly on the pop (dequeue) operation.

Deque

Deques (double-ended queues) are queues for which elements can be added to or removed from either the front or back.

For labs 04 and 05, you were assigned the task of building queues. For those labs, we used a dynamic array as their underlying data structure. Today, you will use a doubly linked list to store the required data. Similarly to lab 05, using a doubly linked list provides constant time operations for both enqueuing and dequeuing. However, this time the deque will allow you to add or remove elements from both its front and back.

Submission

You will need to implement seven standard functions of the Deque data structure:

1. `bool empty() const;`
2. `void push_back(`*`type`*` elem); void push_front(`*`type`*` elem);`
3. `void pop_back(); void pop_front();`
4. *`const Node<type>*`*` front() const; `*`const Node<type>*`*` back() const;`

The `empty` function returns `true` if the deque is empty or `false` otherwise. `Push_back` takes an element as input parameter and adds such an element to the back of the deque. `Push_front` adds an element to the front of the deque. `Pop_front` removes the element at the front of the deque. `Pop_back` removes an element at the back of the deque. The `front` function returns a pointer to the element at the front of the deque. `Back` returns a pointer to the element at the back of the deque.

You will also need to implement four standard functions of the `Stack` data structure:

```
1. bool empty() const;
2. void push(type elem);
3. void pop();
4. const T* top() const;
```

The `empty` function returns `true` if the stack is empty or `false` otherwise. `Push` takes an element as input parameter and adds such an element to the top of the stack. Pop removes the element at the top of the stack. The `top` function returns a pointer to the element at the top of the stack.

For the stack, you should implement public getter methods to expose some of the private variables you may use for internal tracking. This is used to help our automatic test scripts to see the underlying logic of your code. The following methods should be available:

```
5. const T* getData() const;
6. unsigned getCapacity() const;
7. unsigned getSize() const;
8. unsigned getTopIndex() const;
```

The `getData` function returns a constant pointer to the underlying dynamic array used to carry data stored in the queue. `getCapacity` should return the allocated size of the underlying dynamic array. `getSize` exposes the actual size used at a time. `getTopIndex` should return the index of the top element of the stack. Note that for empty stacks the top index is not tested, since it can vary depending on implementation choices.

Also for testing purposes, **you should reallocate your internal array by 1 every time it's needed**. Having a constant allocation offset helps automated test scripts to verify your solution more quickly. Remember to properly deallocate unused dynamic arrays (or nodes) to avoid memory leaks. Memory leak is one of the required tests on Mimir for both `Stack` and Deque.

Hints: You need to reallocate the underlying array when enqueueing new values if the current capacity is not enough to hold all the elements. You should also be mindful of edge cases such as not allowing dequeue to throw an exception with an empty deque.

Complete both the "Lab 06a. Stack" and the "Lab 06b. Deque" activities on Mimir. You should include both the header (Stack.hpp or Deque.hpp) and the source file (Stack.cpp or Deque.cpp) for your implementation**.**

You will implement seven standard functions for the Deque data structure, four standard functions for the `Stack` data structure, and four getter functions. A battery of automated tests will be run to ensure that the errors have been fixed and the results can be properly computed.

A grade of "complete" on this lab work requires a score of 100%.