COSE213 Data Structures
Programming Assignment #3:
Doubly Linked List, Stack, and Deque

Introduction

In this programming assignment, you will implement three fundamental data structures: Doubly Linked List, Stack, and Deque. These implementations will reinforce your understanding of linked data structures, array-based data structures, and the use of stacks for problem-solving. Additionally, you will implement custom error handling through exception classes.

The assignment aims to deepen your understanding of how to efficiently manage collections of data while ensuring robust error handling.

Part 1: Doubly Linked List

You will implement a Doubly Linked List class. This class should include two sentinel nodes:

1. header (points to the first element),
2. trailer (points to the last element).

These two nodes are pointers of Node objects that contains three variables:

1. prev (pointer to the previous node),
2. ele (the data element of type std::string),
3. next (pointer to the next node).

Required Methods:

1. Constructor: Initializes an empty list with header and trailer. “header” and “trailer” are pointing each other at initialization.
2. Destructor: Frees all allocated memory.
3. size(): Returns the number of elements in the list.
4. empty(): Checks if the list is empty.
5. front(): Returns the front element.

6. back(): Returns the back element.
7. add_front(const std::string& e): Adds an element to the front of the list.
8. add_back(const std::string& e): Adds an element to the back of the list.
9. remove_front(): Removes the front element.
10. remove_back(): Removes the back element.

Part 2: Stack Implementation

You will implement an array-based Stack that stores std::string elements. The stack should follow a Last In First Out (LIFO) principle.

Required Methods:

1. Constructor: Takes an integer parameter for the initial stack capacity.
2. Destructor: Frees the allocated memory.
3. size(): Returns the number of elements currently in the stack.
4. empty(): Checks if the stack is empty.
5. top(): Returns the top element of the stack.
6. push(const std::string& e): Adds an element to the top of the stack.
7. pop(): Removes the top element of the stack.

Part 3: Deque Implementation (Based on Doubly Linked List)

You will implement a Deque using the **Doubly Linked List** created in Part 1. A deque allows adding and removing elements from both the front and back.

Required Methods:

1. size(): Returns the number of elements in the deque.
2. empty(): Checks if the deque is empty.
3. push_front(const std::string& e): Adds an element to the front of the deque.
4. push_back(const std::string& e): Adds an element to the back of the deque.
5. pop_front(): Removes the front element from the deque.
6. pop_back(): Removes the back element from the deque.
7. front(): Returns the front element.
8. back(): Returns the back element.

Part 4: Deque reverse() Implementation (Based on Stack)

Implement a function reverse() for the Deque class that reverses the order of the deque's elements. You must use the **Stack** implemented in Part 2 to achieve this.

Part 5: Exception Handling

You are given custom exception classes for handling errors in the data structures defined in *container_exception.h*:

1. ContainerException: Base class for container-related exceptions.
2. ContainerOverflow: Derived class that handles stack overflow errors.
3. ContainerEmpty: Derived class that handles attempts to access or remove elements from an empty list, deque, or stack.

Exception Conditions:

1. Throw ContainerEmpty when trying to access or remove an element from an empty container.
2. Throw ContainerOverflow when trying to push an element onto a full stack.

Skeleton Code

We provide the following files for you to start with:

- main.cpp
- doubly_linked_list.h
- doubly_linked_list.cpp
- stack.h
- stack.cpp
- deque.h
- deque.cpp
- Makefile

Your job is to complete the implementation of doubly_linked_list.cpp, stack.cpp, and deque.cpp. **You cannot change the given header files (*.h)**. We provide an example **Makefile** for compilation. You can add your own files if necessary, but note that you should explain them in ReadMe.txt and address them in your Makefile. You can test your code with main.cpp we provide, but the code written in main.cpp does not cover all the test cases in grading.

Submission

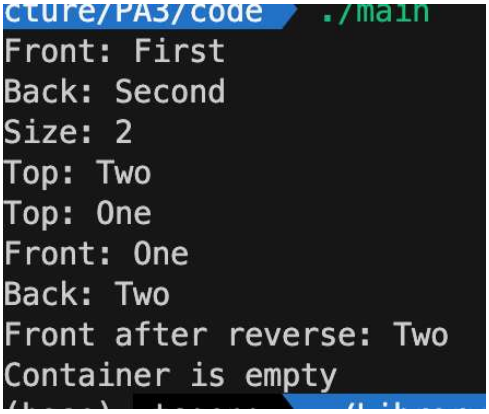
- **Submission Format**: Compress all the relevant files (doubly_linked_list.cpp, stack.cpp, deque.cpp, Makefile, ReadMe.txt, etc.) into a single **zip** file named **STUDENT_ID.zip** and submit it.
    a. **doubly_linked_list.cpp, stack.cpp,** and **deque.cpp**, and **optional files** that you personally added
    b. **Makefile**: Create your own Makefile to compile your project. You can use the Makefile we provided. This will be part of your grade, so make sure it compiles your project correctly when typing make in the terminal. See the "compilation" section in the grading criteria below for details.
    c. **ReadMe.txt**: Include a Readme file explaining any specific considerations for grading your project or any deviations from the instructions.
- **Deadline: 18 May 11:59PM**
    a. **Late Submission**: The late submission follows the policy that was explained in the course overview lecture.

Restrictions

- **Allowed Libraries**: You are permitted to use **only** the following C++ standard libraries: **<iostream>**, **<string>**, and **<exception>**.
- **C++ Version**: It is recommended that you use a recent version of C++ for this assignment. **Minimum Version: C++11**. **Recommended Version: C++17 or higher**.

Example Output

Below is the console output of running main.cpp with correct implementation of *.cpp files.



Grading Criteria

- **Submission format**
  - It is mandatory to ensure the integrity of the provided header files. Your submission should follow the instructions in the “submission” section.
- **Compilation**
  - The “**make**” command MUST work to create the “**main**” program (see the example Makefile provided).
  - While you only need to submit part of the source files, assume that the provided *.h files and main.cpp will be included during compilation, so you don’t need to worry about compilation errors due to missing those files.
  - If compilation fails, the score will be zero.
  - If a runtime exception halts the program (e.g., a crash due to a segmentation fault), the score will be based on progress up to the point of termination.
- **Main Parts**
  - Various test cases will be used to assess the functionality of each requirement described in this document. Scores will only be given to correctly passed test cases.