Review of linked list and resizing array

Linked list

A *linked list* is a data structure that consists of a sequence of elements, where each element points to the next element in the sequence. The first element is called the *head* of the list, and the last element points to `nullptr`. A linked list can be singly linked or doubly linked. In a singly linked list, each element points to the next element, whereas in a doubly linked list, each element points to both the next and the previous elements. It provided the following operations:

Operation	Description
`insertAtHead(x)`	Insert an item x at the head of the list
`insertAtTail(x)`	Insert an item x at the tail of the list
`deleteHead()`	Remove the head of the list
`get(i)`	Return the item at index i
`size()`	Return the number of items in the list
`empty()`	Return true if the list is empty

All operations take constant time, except `get(i)` which takes linear time in the worst case.

Resizing array

A *resizing array* is an array that automatically resizes itself when the number of elements exceeds its capacity. When the array is full, a new array of double the size is created, and all elements are copied to the new array. This operation takes n steps, where n is the number of elements in the array. Resizing arrays are useful when the number of elements is not known in advance. Similarly, when the number of elements in the array is less than a quarter of its capacity, a new array of half the size is created, and all elements are copied to the new array.
It provided the following operations:

Operation	Description
`push_back(x)`	Insert an item x at the end of the array
`pop_back()`	Remove the last item of the array
`get(i)`	Return the item at index i
`size()`	Return the number of items in the array
`empty()`	Return true if the array is empty

All operations take constant time in the worst case, except `push_back(x)` and `pop_back()` which take linear time in the worst case. However, the amortized time for `push_back(x)` and `pop_back()` is constant. By amortized, we mean that if we perform a sequence of m operations, the total time taken by all operations is $O(m)$, which means that the average time taken by each operation is constant.

Stack and Queue

Stack and *queue* are two important ADTs. A stack stores items in a last-in, first-out (LIFO) manner, whereas a queue stores items in a first-in, first-out (FIFO) manner. Following table summarizes the operations of a stack and a queue.

Stack	Queue	Description
`push(x)`	`enqueue(x)`	Insert an item x
`pop()`	`dequeue()`	Remove and return an item (most recently inserted item for stack and least recently inserted one for queue)
`empty()`	`empty()`	Return true if the stack/queue is empty
`size()`	`size()`	Return the number of items in the stack/queue

We have seen two implementations of stacks and queues in the previous semester: using linked list and using resizable array. Note that from the perspective of the user, the two implementations of stack (respectively queue) are indistinguishable as both provide the identical public interface.

Lab 1: Text Editor Buffer with Stacks (Array vs Linked List)

Learning Objectives

- Review linked list and resizing array data structures.
- Implement stack ADT on top of both data structures.
- Use stacks to build a text editor buffer.
- Compare time/space trade-offs between implementations.

Part A: Resizing Array

Implement a class `ResizingArray` that implements a dynamic array of integers. The class should provide the operations described in the previous section and use a resizing array to store the elements. It must handle edge cases, such as popping from an empty array or accessing an index out of bounds. Use the code provided in `lab-01.cpp` file as a starting point. You need to complete the implementation of the methods.

Part B: Linked List

Implement a class `LinkedList` that implements a singly linked list of integers. The class should provide the operations described in the previous section and use a linked list to store the elements. It must handle edge cases, such as deleting from an empty list or accessing an index out of bounds. Use the code provided in `lab-01.cpp` file as a starting point. You need to complete the implementation of the methods.

Part C: Stack ADT

Implement stack ADT twice using `ResizingArray` and `LinkedList` as the underlying data structure. The classes should provide the operations described in the previous section and use the respective data structure to store the elements. It must handle edge cases, such as popping from an empty stack. Use the code provided in `lab-01.cpp` file as a starting point. You need to complete the implementation of the methods.

Part D: Text Editor Buffer

You will now simulate a text editor buffer with two stacks (left of cursor, right of cursor). Implement TextBuffer twice: Using stacks based on `ResizableArray` and using stacks based on textttLinkedList.
The class should provide the following operations:

Operation	Description
`insert(c)`	Insert a character c at the current cursor position
`remove()`	Delete the character at the current cursor position
`moveLeft()`	Move the cursor one position to the left
`moveRight()`	Move the cursor one position to the right
`getText()`	Return the current text in the buffer as a string

The class should use two stacks to store the characters to the left and right of the cursor. Use the code provided in `lab-01.cpp` file as a starting point. You need to complete the implementation of the methods.

Part E: Testing and Comparison

Write a test program that creates two instances of `TextBuffer`: one using stacks based on `ResizingArray` and the other using stacks based on `LinkedList`. The program should perform a series of operations on both instances, such as inserting characters, deleting characters, and moving the cursor. Measure the time taken for each operation and the memory used by each instance. You can use the `stopwatch` class provided in the code to measure the time taken for sequence of operations.
Compare the performance of the two implementations and write a short report summarizing your findings.

Submission

Submit the following files:

- `lab-01.cpp`: Complete implementation of all classes and methods.
- `lab-01-report.txt`: A short report comparing the two approaches.