Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Assignment 2 | Stack

CSE106 — DSA I

Introduction

You've probably seen a **stack** today—maybe a pile of books, plates, or clothes. If you've ever placed an item on top of such a pile or removed the top one, congratulations— you've already implemented a stack. That's exactly how the *stack* data structure works in programming.

A **stack** is a simple yet powerful data structure. It operates on a **Last In, First Out (LIFO)** principle: the last item added is the first one to be removed. Think of it like stacking books—easy to add or remove from the top, but hard to reach the bottom without disturbing everything else.

In programming, the two primary operations of a stack are:

- **Push** – Add an item to the top of the stack.
- **Pop** – Remove the item from the top of the stack.

Now, you might wonder: why use a stack instead of an array or linked list? After all, those can do *more*. The answer lies in the **concept**. A stack is an **Abstract Data Type (ADT)**—it defines a *behavior* (push/pop), not how that behavior is implemented. Arrays and linked lists are **data structures**— they are concrete implementations that can support stacks, among other things.

Think of an ADT like a blueprint for a car: it describes what the car should do (e.g., go fast, be safe), not how it's built. The data structure is the engine and chassis—the actual design that fulfills the blueprint. Using stacks helps us solve real problems more cleanly and intuitively. For example, when folding laundry, you often create two stacks: one for folded clothes and one for unfolded ones. Many programming problems—like parenthesis matching, expression evaluation, and parsing—benefit from this same simple push/pop structure.

What You'll Do

In this assignment, you'll:

1. Define a **Stack ADT**.
2. Implement the stack using **dynamic arrays**.
3. Implement the stack using **linked lists**.
4. **Test** your implementations for correctness.

Let's get started.

1 Stack ADT

Overview

We define the Stack ADT as a C++ *abstract class*.

You are probably familiar with *abstract classes* and inheritance. Abstract classes fit perfectly for the idea of an ADT. An abstract class defines a set of operations (functions) that must be implemented by any concrete class that inherits from it. The abstract class defines the interface, while the concrete classes provide the implementation.

We will start with the abstract class, which defines what features our stack should have. In our case we will define the following operations: *push, pop, clear, size, top, empty* and *print*. These methods present how any implementation of a stack should behave so that the user of the stack can use it without knowing the details of the implementation.

We already provided you with the abstract class `Stack` in the file `stack.h`. It contains the required methods definitions. You don't need to modify this file. So, for this section you don't need to write any code.

Implmentation Notes

- ➤ You have to use C++ for this assignment.
- ➤ You have been given template code,
  YOU SHOULD STRICTLY EDIT TEMPLATE FILES ONLY.
- ➤ **DO NOT MODIFY UNLESS THERE IS A `TODO` COMMENT**.
- ➤ You must use **only your own stack implementations** for all the tasks.
- ➤ No built-in data structures (e.g. STL) allowed.

Method Specifications

The most fundamental operations first!

void push(int item)

Description: Pushes a new item to the top of the stack.

Parameters: `item` — element to push (`int`) **Returns:** None

```
Stack* s = new ListStack();
s->push(5);
s->push(15); // Stack: 5, 15 (top)
s->print();  // Output: |15, 5>
delete s;
```

int pop()

Description: Removes and returns the item at the top of the stack.

Parameters: None **Returns:** Top item (`int`)

```
Stack* s = new ListStack();
s->push(8);
s->print();      // Output: |8>
int val = s->pop(); // val = 8
s->print();      // Output: |>
delete s;
```

void clear()

Description: Empties the stack, removing all elements. After this operation, the stack becomes logically empty.

Parameters: None **Returns:** None

```
Stack* s = new ListStack();
s->push(10);
s->push(20);
s->print();  // Output: |20, 10>
s->clear(); // Stack is now empty
s->print();  // Output: |>
delete s;
```

`int size() const`

Description: Returns the number of elements currently in the stack.

Parameters: None **Returns:** Integer representing stack size

```
Stack* s = new ListStack();
s->push(3);
s->push(7);
s->print(); // Output: |7, 3>
int len = s->size(); // len = 2
delete s;
```

`int top() const`

Description: Returns the top item without removing it from the stack.

Parameters: None **Returns:** Top item (`int`)

```
Stack* s = new ListStack();
s->push(42);
s->print();        // Output: |42>
int top_val = s->top(); // top_val = 42
s->print();        // Output: |42> (stack unchanged)
delete s;
```

`bool empty() const`

Description: Checks if the stack is empty.

Parameters: None **Returns:** `true` if the stack is empty, `false` otherwise.

```
Stack* s = new ListStack();
s->print(); // Output: |>
bool isEmpty = s->empty(); // isEmpty = true
s->push(10);
s->print(); // Output: |10>
isEmpty = s->empty(); // isEmpty = false
delete s;
```

void print() const

Description: Prints all elements in the stack, from top to bottom. The exact format may depend on the implementation.

Parameters: None **Returns:** None

```
Stack* s = new ListStack();
s->push(10);
s->push(20);
s->push(30);
// Example of print() output format:
s->print(); // Output: |30, 20, 10>
delete s;
```

2 ArrayStack Implementation

Overview

We will implement the Stack ADT using a dynamically allocated array.

You must have used arrays at this point. Whenever we need to store more than one item of the same type, we use an array. Arrays are great, but they have some limitations. They are fixed in size, meaning you have to decide how many items you want to store before you create the array. However, there is a way to overcome this limitation by using a **dynamic array**; dynamically allocating an array of a size that you can change at runtime. And if we still need more space, we can create a new array and copy all the elements from the old array to the new one. This is called **resizing** the array.

In this assignment when the stack reaches 50% of its capacity, we'll resize it to double the capacity and copying all elements. When the stack becomes 25% full, we will resize it again to half its capacity. This approach provides a balance between memory usage and performance.

Data Member Specifications

Private Data Members

- `int* data` — *Dynamic array pointer* that stores elements in contiguous memory
- `int capacity` — *Maximum size* of the array; doubles when full and shrinks when sparsely populated
- `int current_size` — *Element count* tracking the number of items currently in the stack

Constructor and Destructor

ArrayStack(int capacity = 10)

Description: Creates a new ArrayStack with the specified initial capacity.

Parameters: `capacity` — Initial capacity of the array (default: 10) **Returns:** None

```
ArrayStack* stack = new ArrayStack(20); // Creates stack with initial
    capacity of 20
ArrayStack* defaultStack = new ArrayStack(); // Creates stack with
   default capacity (10)
```

~ArrayStack()

Description: Destroys the ArrayStack and releases all allocated memory.

Parameters: None **Returns:** None

```
ArrayStack* stack = new ArrayStack();
delete stack; // Destructor called, memory freed
```

Implementation Requirements

ArrayStack Requirements

- ✓ Implement the methods in `arraystack.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate the array when the stack is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Implement dynamic resizing:
  - When the stack reaches 50% of its capacity, create a new array with double the capacity.
  - The array should shrink to half its capacity (but not less than 10) when it becomes 25% full.
- ✓ Gracefully handle edge cases
- ✓ Time complexity requirements:
  - `push, pop`: O(1) amortized time
  - `top, size, empty`: O(1) time

3 ListStack Implementation

Overview

We will now implement the Stack ADT using a linked list.

You probably also know about **linked lists**. They are a data structure where each element (node) contains a value and a pointer to the next node. Linked list-based stacks offer several advantages: they can grow dynamically without requiring resizing operations, and they efficiently use memory as each element is allocated individually. You don't need to resize the entire structure when adding or removing elements.

The ListStack maintains a pointer to the head node (the top of the stack) and keeps track of the current size. When new elements are pushed, they are added to the front of the list. When elements are popped, they are removed from the front as well. If on the other hand, elements were pushed to the back of the list, we would have to traverse the entire list to pop an element, which would be inefficient.

Data Member Specifications

Private Data Members and Structures

- `struct Node` — *Internal structure* representing each element:
  - `int data` — *Value* stored in the node
  - `Node* next` — *Pointer* to the next node in the sequence
  - `Node(int value, Node* next)` — *Constructor* to initialize a new node
- `Node* head` — *Top node pointer* marking the stack's entry point
- `int current_size` — *Element counter* tracking the total items in the stack

Constructor and Destructor

`ListStack()`

Description: Creates a new empty ListStack.

Parameters: None **Returns:** None

```
ListStack* stack = new ListStack(); // Creates an empty linked list stack
```

~ListStack()

Description: Destroys the ListStack and releases all allocated memory for nodes.

Parameters: None **Returns:** None

```
ListStack* stack = new ListStack();
// ... use the stack
delete stack; // Destructor called, all nodes are deallocated
```

Implementation Requirements

ListStack Requirements

- ✓ Implement the methods in `liststack.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate all nodes when the stack is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Gracefully handle edge cases
- ✓ Time complexity requirements:
  - `push, pop, top, size, empty`: O(1) time
- ✓ New nodes should be inserted at the head (front) of the list for efficient stack operations

4 Testing

Overview

We will run some tests on your stack implementations to ensure they work correctly.

Eventually one day you will be a software engineer or at least you will build software in one form or another. You will need to use all these data structures and algorithms in your career but the end goal is to build software that works. No one likes software that crashes or has bugs. But how do you know if your software is working?

The quick answer is: **testing**. Testing is the process of running your code with various inputs to ensure it behaves as expected. We often run our code with test case files that cover a wide range of scenarios, including edge cases and typical usage.

In this assignment, we will provide you with a testing framework that will automate this process for you. Instead of testing with input files, we will run a series of test functions that check the behavior of your stack implementations. This idea of testing the behavior of your code is called **unit testing**. You will learn more about unit testing in the future, but for now, we will use a simple framework to test your stack implementations.

The Testing Framework

The testing framework follows a simple process:

1. Test cases are defined as functions that accept a `Stack*` parameter and return a boolean result
2. If the test passes, it returns `true`; otherwise, it returns `false`
3. The `StackTester` class manages adding new test cases and running them
4. Dynamic casting is used to identify which implementation (`ArrayStack` or `ListStack`) is being tested
5. Each test is run on all stack implementations
6. Results are displayed with color-coded `PASS`/`FAIL` indicators

You do **NOT** need to modify the framework itself. Your task is to **implement the individual test cases** that verify your stack implementations work correctly.

Where to Edit

Edit These Parts in `test.cpp`

- Locate the test function declarations in `test.cpp` (starting around line 100)
- Three test cases are already implemented for you: `test_push`, `test_pop`, and `test_top`
- Implement at least 4 additional test functions that have `// TODO` comments (The remaining 3 are optional)
- Do not change the function signatures or the main function

Test Case Examples

Study these examples to understand how to implement your test cases:

Example: Push Test

```
// Test for push method
bool test_push(Stack* stack) {
    stack->clear();           // Start with a clean state
    stack->push(42);          // Perform the operation we're testing

    // Verify expected outcomes
    return stack->size() == 1 && stack->top() == 42;
}
```

Example: Pop Test

```
// Test for pop method
bool test_pop(Stack* stack) {
    stack->clear();           // Start with a clean state
    stack->push(10);
    stack->push(20);          // Setup test conditions

    int popped = stack->pop();  // Perform the operation we're
       testing

    // Verify multiple expected outcomes
    return popped == 20           // Correct value was popped
        && stack->size() == 1     // Size decreased correctly
        && stack->top() == 10;    // Top element is now correct
}
```

Test Cases to Implement

Complete these test functions in `test.cpp`:

Required Test Cases

- `test_size` — Test if size() correctly reports the number of elements
- `test_empty` — Test if empty() correctly detects when a stack is empty or not
- `test_clear` — Test if clear() properly removes all elements
- `test_multiple_push_pop` — Test LIFO behavior with multiple operations

Additional Test Cases (Optional)

- `test_empty_stack_operations` — Test behavior when operating on an empty stack
- `test_array_resizing` — Test if ArrayStack properly resizes when needed
- `test_stress` — Test with a large number of operations

Implementation Requirements

Testing Guidelines

- ✓ Implement the tests in `test.cpp` file
- ✓ Always start with a clean state using `stack->clear()`
- ✓ Test one specific behavior in each test function
- ✓ Include both typical usage and edge cases
- ✓ Check multiple aspects of the state after operations

5 Compiling and Running

Compiling Commands

```
# Compile the stack implementations and test program
g++ -std=c++11 test.cpp arraystack.cpp liststack.cpp -o stack_test

# Run the test program
./stack_test
```

Notes on Compilation

- Be sure to include all source files in the compilation command
- The `-std=c++11` flag ensures compatibility with the C++11 standard
- The `-o stack_test` flag specifies the output executable name

6 Submission

Submission Process

1. Create folder named: `<your_id>` (e.g., `2305xxx`)
2. Include the following files:
   - `stack.h` — Stack ADT header file
   - `arraystack.cpp` — Array-based stack implementation
   - `liststack.cpp` — Linked List-based stack implementation
   - `test.cpp` — Test cases for both implementations
   - Any shared code files needed
   - `readme.txt` — Brief description of your files
3. Compress as `<your_id>.zip` and upload to Moodle

Deadline: May 16, 2025, 10:00 PM

WARNING

✗ Do not copy your code from your peers or other sources;
Will result in **-100% penalty**.

✗ No submissions will be accepted after the deadline. Submit on time.

Evaluation Policy

Component	Points
Adherence to Template	20
Array-based Stack	25
List Stack	25
Writing Tests	10
Bonus Tests	5
Passing Tests	15
Submission format	5
Total	**105**

Good luck with your implementation!