CADT IDT

W5-S3 PRACTICE

Stack & Queue

At the end of this practice, you should be able to…

Implement both Stack and Queue ADTs using a linked list.

Recognize the advantages of linked lists in stack and queue implementations, especially for dynamic data.

Compare the time and space complexity of stack and queue operations.

How to compile your code?

Assuming your file is named: `exercise.cpp`:

Open a **terminal** at your file location

Compile your Program using the following command

```
g++ -o exercise exercise.cpp
```

Run Your Program using the following command

```
./exercise
```

How to submit?

- ✓ Make a report PDF containing the screenshot of your program code and output for each exercise.
- ✓ **Submit your final source code report PDF to Microsoft team and turn it in.**

No internet during this work

No internet during this practice so you can **learn to solve problems on your own**.

*Specification of the **Stack & Queue** ADT*

A **Stack** follows the LIFO (Last-In-First-Out) principle, meaning that the most recently added element is the first to be removed. This structure operates by adding elements to the top (push) and removing elements from the top (pop).

Table 1: Stack ADT Operations Using Linked List

Operation	Description	Precondition	Usage Example

`push(value)`	Adds an element to the top of the stack.	None	`myStack <- []`
			`myStack.push(11)`
			`myStack.push(22)`
			`myStack.push(33)`
			
			`Result: myStack <- [33, 22, 11]`

`pop()`	Removes the top element from the stack.	Stack not empty	`myStack <- [33, 22, 11]`
			`myStack.pop()`
			
			`Result: myStack <- [22, 11]`

`peek()`	Returns the top element without removing it.	Stack not empty	`myStack <- [33, 22, 11]`
			`result = myStack.peek()`
			
			`Result: result = 33`

`isEmpty()`	Checks if the stack is empty.	None	`myStack <- [33, 22, 11]`
			`myStack.isEmpty()`
			
			`Result: false`

`String print()`	(for test purpose)
	
	Return a string representation of the Stack (Top -> bottom)
		None	`myStack <- [33, 22, 11]`
			`result = myStack.print()`
			
			`Result: result = 33 22 11`

A **Queue** follows the FIFO (First-In-First-Out) principle, where the element that is added first is the first to be removed. Elements are added to the back (enqueue) and removed from the front (dequeue).

Table 2:Queue ADT Operations Using Linked List

Operation	Description	Precondition	Usage Example

EXERCIS

`enqueue(value)`	Adds an element to the end of the queue.	None	`myQueue <- []`
			`myQueue.enqueue(11)`
			`myQueue.enqueue(22)`
			`myQueue.enqueue(33)`
			
			`Result: myQueue <- [11, 22, 33]`

`dequeue()`	Removes the front element from the queue.	Queue not empty	`myQueue <- [11, 22, 33]`
			`myQueue.dequeue()`
			
			`Result: myQueue <- [22, 33]`

`peek()`	Returns the front element without removing it.	Queue not empty	`myQueue <- [11, 22, 33]`
			`result = myStack.peek()`
			
			`Result: result = 11`

`isEmpty()`	Checks if the queue is empty.	None	`myQueue <- []`
			`myQueue.isEmpty()`
			
			`Result: true`

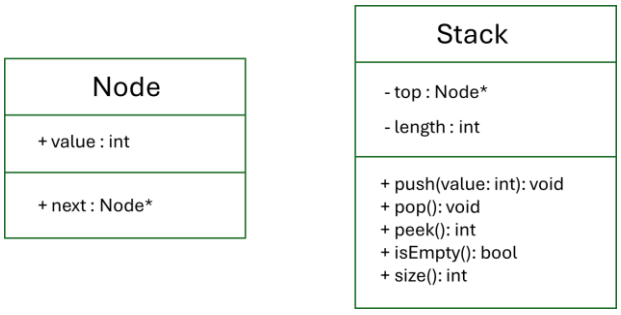
`String print()`	(for test purpose)
	
	Return a string representation of the Stack (Front -> Back)
		None	`myStack <- [11, 22, 33]`
			`result = myStack.print()`
			
			`Result: result = 11 22 33`

E 1: Implement the **Stack** ADT using a LinkedList

In this exercise, you will implement the Stack ADT using a singly linked list. The stack should support the standard operations such as push(), pop(), peek(), and isEmpty(). The linked list-based stack will dynamically allocate memory as elements are added and removed, ensuring that the stack can grow or shrink as needed without a fixed size.

SPECIFICATIONS

Your data structure should be a **class** as the following UML:



The **Stack** should **implement the 5 operations** required by the Stack ADT (see above)

Note:

- ***test-case.cpp***: This is your testing file where you will run the various test cases to validate the stack implementation.
- ***Stack.h***: This file contains the implementation of the Stack class with the methods push, pop, peek, isEmpty, and stackSize (to return the size of the stack). It also uses the Node class.

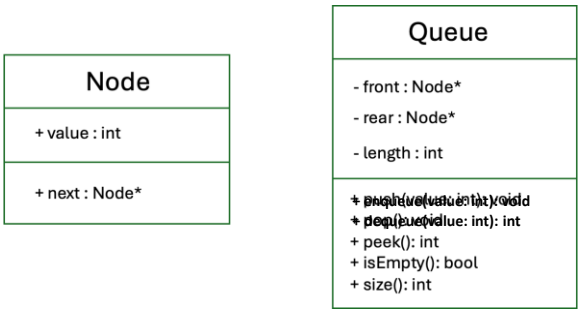
EXERCIS

*E 2: Implement the **Queue** ADT using a LinkedList*

In this exercise, you will implement the **Queue** ADT using a singly linked list. The queue should support standard operations like **enqueue** (adding an element to the end of the queue), **dequeue** (removing an element from the front), **peek** (retrieving the first element without removing it), and **isEmpty** (checking if the queue is empty). Implementing the queue using a linked list allows dynamic memory allocation, making it efficient for scenarios where the number of elements can change frequently.

SPECIFICATIONS

Your data structure should be a **class** as the following UML:



The **Stack** should **implement the 5 operations** required by the Stack ADT (see above) *Note:*

- ***test-case.cpp***: This is your testing file where you will run the various test cases to validate the stack implementation.
- ***Queue.h***: This file contains the implementation of the Queue class with the methods enqueue, dequeue, peek, isEmpty, and size. It also uses the Node class.

E 3: Identify Real-World Scenarios & Limitations

In this exercise, you'll analyze and understand the use cases and limitations of **stack** and **queue** data structures by applying them to real-world scenarios. The objective is to deepen your knowledge of where these data structures excel and where they may encounter performance or usability issues.

1. Real-World Scenarios:

For each data structure (stack and queue), identify **three** real-world scenarios where it can be effectively applied. Describe each scenario and explain why the LIFO (Last-In, First-Out) property of stacks or the FIFO (First-In, First-Out) property of queues makes them suitable for that use case.

2. Identify Limitation:

For each scenario you identified, analyze and discuss any limitations that might arise from using a stack or queue. Consider scenarios such as:

- Access to elements (e.g., in stacks, only the top element can be accessed).
- Performance issues (e.g., fixed-size stack overflow).
- Special cases where priority or order needs to be altered (e.g., queues unable to handle priority tasks).

3. Analysis Table

Complete the table below with the scenarios and limitations you have identified.

Data Structure	Real-world Scenario	Why this data structure works?	Limitaion
Ex. Stack	*Undo feature in text editors*	*LIFO order is ideal for undoing recent changes*	*Limited to recent actions; cannot undo specific steps*
Ex. Stack	Web Browser Back/Forward Navigation	LIFO order is ideal for the most recent action or data needs to be processed first.	Limitation no random access for all page only back and forth
Ex.stack	Backtracking in Maze Solving	LIFO order is ideal to explore a path and backtrack (pop) to the previous point when hitting a dead end.	Limitation memory usage for large mazes.
Ex.Queue	Printer Job Scheduling	FIFO order is ideal for ensuring print jobs are	Limitation: No Prioritization Urgent jobs must wait for

EXERCIS

		completed in the order they are received.	earlier ones to complete unless a priority system is added.
Ex.Queue	Customer Service Call Center	FIFO order is ideal for attended to in the order they called, maintaining fairness and orderliness.	Limitation: Wait Times: Customers at the back may face long delays if the queue is large.

Summary your findings here (if you want to further explain)

BONUS

EXERCISE 4: Implement a queue data structure for storing students' data

In this exercise, you are asked to design a program that can manipulate waiting queue of the students for registering to a competition program in public.

You can read data of students from File IO (txt or excel or CSV), or user input from keyboard. Let's assume that we have 10 students. Each student has ID, name, phone number, gender, and major.

Test your program by display all data in the queue. Then, find out who are the 3 first students in the queue.