Bangladesh University of Engineering and Technology

Department of Computer Science and Engineering

Assignment 2 |

CSE106 — DSA I

1 Queue ADT

Overview

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

We will start with an abstract class, which defines what features our queue should have. In our case we will define the following operations: `enqueue, dequeue, clear, size, front, back, empty` and `toString`. These methods present how any implementation of a queue should behave so that the user of the queue can use it without knowing the details of the implementation.

We already provided you with the abstract class `Queue` in the file `queue.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 queue implementations** for Task 1 and Task 2.
- For Task 3, you **must use** `std::stack` from the STL.
- No other built-in data structures (e.g. STL containers) allowed except where specified.

Method Specifications

The most fundamental operations first!

`void enqueue(int item)`

Description: Adds a new item to the back of the queue.

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

```
Queue* q = new ListQueue();
q->enqueue(5);
q->enqueue(15); // Queue: 5, 15 (15 is at back)
cout << q->toString() << endl;      // Output: <5, 15|
delete q;
```

`int dequeue()`

Description: Removes and returns the item at the front of the queue.

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

```
Queue* q = new ListQueue();
q->enqueue(8);
cout << q->toString() << endl;  // Output: <8|
int val = q->dequeue();         // val = 8
cout << q->toString() << endl;  // Output: <|
delete q;
```

`void clear()`

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

Parameters: None **Returns:** None

```
Queue* q = new ListQueue();
q->enqueue(10);
q->enqueue(20);
cout << q->toString() << endl;  // Output: <10, 20|
q->clear();                     // Queue is now empty
cout << q->toString() << endl;  // Output: <|
delete q;
```

`int size() const`

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

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

```
Queue* q = new ListQueue();
q->enqueue(3);
q->enqueue(7);
cout << q->toString() << endl; // Output: <3, 7|
int len = q->size();           // len = 2
delete q;
```

`int front() const`

Description: Returns the front item without removing it from the queue.

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

```
Queue* q = new ListQueue();
q->enqueue(42);
cout << q->toString() << endl;  // Output: <42|
int f = q->front();             // f = 42
cout << q->toString() << endl;  // Output: <42| (unchanged)
delete q;
```

`int back() const`

Description: Returns the back (rear) item without removing it from the queue.

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

```
Queue* q = new ListQueue();
q->enqueue(42);
q->enqueue(99);
cout << q->toString() << endl; // Output: <42, 99|
int b = q->back();             // b = 99
cout << q->toString() << endl; // Output: <42, 99| (unchanged)
delete q;
```

bool empty() const

Description: Checks if the queue is empty.

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

```cpp
Queue* q = new ListQueue();
cout << q->toString() << endl;  // Output: <|
bool isEmpty = q->empty();      // true
q->enqueue(10);
cout << q->toString() << endl;  // Output: <10|
isEmpty = q->empty();           // false
delete q;
```

string toString() const

Description: Converts queue to a string representation.

Parameters: None **Returns:** `String`

```cpp
Queue* q = new ListQueue();
q->enqueue(10);
q->enqueue(20);
q->enqueue(30);
// Example of toString() return format:
cout << q->toString() << endl; // Output: <10, 20, 30|
delete q;
```

2 Task 1: ArrayQueue Implementation

Overview

We will implement the Queue 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 queue becomes full (i.e., reaches its maximum capacity), we will resize it to double its capacity and copy all elements in order. When it is less than 25% of the current capacity, we will resize it to half its capacity (but not less than 2). This strategy balances memory usage and performance by minimizing resizing overhead while maintaining space efficiency.

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 front_idx` — *Index* of the current front item in the queue
- `int rear_idx` — *Index* of the current rear item in the queue

Method Specifications

You are required to implement an extra method in ArrayQueue.

`int getCapacity() const`

Description: Returns the number of elements the underlying array can currently hold without resizing.

Returns: Integer representing the current capacity of the dynamic array.

```
ArrayQueue* q = new ArrayQueue(20); // Initial capacity 20
q->enqueue(1);
q->enqueue(2);
cout << q->getCapacity() << endl; // Output: 20
delete q;
```

Constructor and Destructor

`ArrayQueue(int capacity = 10)`

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

Parameters: `capacity` - Initial capacity of the array (default: 2) **Returns:** None

```
ArrayQueue* queue = new ArrayQueue(20); // Creates queue with initial
    capacity of 20
ArrayQueue* defaultQueue = new ArrayQueue(); // Creates queue with
    default capacity (2)
```

~ArrayQueue()

Description: Destroys the ArrayQueue and releases all allocated memory.

Parameters: None **Returns:** None

```cpp
ArrayQueue* queue = new ArrayQueue();
delete queue; // Destructor called, memory freed
```

Implementation Requirements

ArrayQueue Requirements

- ✓ Implement the methods in `arrayqueue.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate the array when the queue is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Implement dynamic resizing:
  - When the queue becomes full, create a new array with double the current capacity and copy elements in logical order
  - When the queue's size drops to 25% or less of its capacity, shrink the array to half its current capacity (but not less than 2)
- ✓ Implement circular buffer behavior using modulo arithmetic for `front` and `rear` indices
- ✓ Gracefully handle edge cases (e.g., dequeue from empty queue)
- ✓ Time complexity requirements:
  - `enqueue, dequeue`: O(1) amortized time
  - `front, back, size, empty`: O(1) time

Testing Your Implementation

We have provided a file `array_queue_tester.cpp` to help you verify the correctness of your queue implementation. This tester creates two queue instances: one using your implemented queue class and another using the C++ Standard Library `std::queue`. The tester performs a sequence of randomly generated queue operations (such as `push`, `pop`, `front` etc.) on both queues. After each operation, the results of your implementation are compared against those of the `std::queue`. If any inconsistency is detected—such as mismatched return values, incorrect queue state, or unexpected behavior—the tester will report an error.

3 Task 2: ListQueue Implementation

Overview

We will now implement the Queue 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 queues 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 ListQueue maintains pointers to both the head (front of the queue) and tail (rear of the queue) nodes, and keeps track of the current size. When new elements are enqueued, they are added to the back (tail) of the list. When elements are dequeued, they are removed from the front (head). This allows efficient O(1) time for both enqueue and dequeue operations without needing to traverse the list.

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* front_node` — *Front pointer* marking the front of the queue
- `Node* rear_node` — *Rear pointer* marking the back of the queue
- `int current_size` — *Element counter* tracking the total items in the queue

Constructor and Destructor

`ListQueue()`

Description: Creates a new empty ListQueue.

Parameters: None **Returns:** None

```
ListQueue* queue = new ListQueue(); // Creates an empty linked list
    queue
```

~ListQueue()

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

Parameters: None **Returns:** None

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

Implementation Requirements

ListQueue Requirements

- ✓ Implement the methods in `listqueue.cpp` according to the specifications
- ✓ Implement proper memory management:
  - Deallocate all nodes when the queue is cleared or destructed
  - Ensure no memory leaks occur during any operation
- ✓ Gracefully handle edge cases
- ✓ Time complexity requirements:
  - `enqueue`, `dequeue`, `front`, `back`, `size`, `empty`: O(1) time
- ✓ New nodes should be inserted at the tail and removed from the head for efficient queue operations

Testing Your Implementation

We have provided a file `list_queue_tester.cpp` to help you verify the correctness of your queue implementation. This tester creates two queue instances: one using your implemented queue class and another using the C++ Standard Library `std::queue`. The tester performs a sequence of randomly generated queue operations (such as `push`, `pop`, `front` etc.) on both queues. After each operation, the results of your implementation are compared against those of the `std::queue`. If any inconsistency is detected—such as mismatched return values, incorrect queue state, or unexpected behavior—the tester will report an error.

4 Task 3: Syntax Checker using Stack

Overview

Implement a syntax checker that validates whether brackets in arithmetic expressions are properly matched using `std::stack`.

Have you ever written code with mismatched parentheses and gotten a syntax error? Compilers and interpreters use a similar technique to what you'll implement in this task. A **stack** is perfect for checking bracket matching because it follows a **Last In, First Out (LIFO)** principle—the most recently opened bracket should be the first one to close.

In this task, you will implement a function that checks whether brackets in an arithmetic expression are properly matched. The expression may contain three types of brackets:

- **Parentheses:** `( )`
- **Square brackets:** `[ ]`
- **Curly braces:** { }

An expression is considered **valid** if:

1. Every opening bracket has a corresponding closing bracket of the same type
2. Brackets are closed in the correct order (no interleaving)
3. There are no extra closing brackets without matching opening brackets

Examples

Valid Expressions

```
"(3 + 5) * 2"                     // Valid
"[(2 + 3) * {4 - 1}]"             // Valid
"{[()]}"                          // Valid
"((a + b) * (c - d))"             // Valid
""                                // Valid (empty string)
"2 + 3 * 4"                       // Valid (no brackets)
```

Invalid Expressions

```
"((3 + 5)"                 // Missing closing parenthesis
"(3 + 5]"                  // Mismatched brackets
"[(2 + 3)"                 // Missing closing square bracket
"{[(])}"                   // Incorrect order
")("                       // Extra closing bracket first
"((a + b) * (c - d)]"      // Extra closing bracket
```

Function Specification

bool isValidExpression(const string& expression)

Description: Checks whether brackets in an arithmetic expression are properly matched.

Parameters: `expression` — A string containing an arithmetic expression **Returns:** `bool`

Returns: `true` if all brackets are properly matched and balanced, `false` otherwise.

```cpp
#include <stack>
#include <string>
using namespace std;

bool isValidExpression(const string& expression) {
    // TODO: Implement this function
}

// Usage examples:
cout << isValidExpression("(3 + 5) * 2") << endl; // Output: 1 (true)
cout << isValidExpression("((3 + 5)") << endl;  // Output: 0 (false)
cout << isValidExpression("{[(])}") << endl;   // Output: 0 (false)
```

Implementation Requirements

Syntax Checker Requirements

✓ Implement the function in `syntax_checker.cpp`

✓ **Must use** `std::stack<char>` from the C++ Standard Library

✓ Include the header: `#include <stack>`

✓ Handle all three bracket types: `()`, `[]`, {}

✓ Ignore all non-bracket characters (numbers, operators, letters, spaces, etc.)

✓ Return `true` for empty strings or strings with no brackets

✓ Do not use any other STL containers besides `std::stack`

✓ Time complexity: O(n) where n is the length of the expression

✓ Space complexity: O(n) in worst case (all opening brackets)

Testing Your Implementation

Test Cases

```cpp
// Test your implementation with these cases:

// Basic valid cases
assert(isValidExpression("()") == true);
assert(isValidExpression("[]") == true);
assert(isValidExpression("{}") == true);
assert(isValidExpression("()[]{}") == true);

// Nested valid cases
assert(isValidExpression("{[()]}") == true);
assert(isValidExpression("((()))") == true);

// With expressions
assert(isValidExpression("(3 + 5) * 2") == true);
assert(isValidExpression("[(2 + 3) * {4 - 1}]") == true);

// Invalid cases
assert(isValidExpression("(") == false);
assert(isValidExpression(")") == false);
assert(isValidExpression("(]") == false);
assert(isValidExpression("([)]") == false);
assert(isValidExpression("((()") == false);

// Edge cases
assert(isValidExpression("") == true);
assert(isValidExpression("2 + 3 * 4") == true);
```

Helper Function (Optional)

You may find it helpful to implement a helper function:

`bool isMatchingPair(char opening, char closing)`

Description: Checks if an opening bracket matches a closing bracket.

```cpp
bool isMatchingPair(char opening, char closing) {
    return (opening == '(' && closing == ')') ||
           (opening == '[' && closing == ']') ||
           (opening == '{' && closing == '}');
}
```

5 Skeleton Code

You will receive a zip file containing the skeleton code. The folder structure is as follows:

```
|-- main.cpp
|-- listqueue.cpp
|-- arrayqueue.cpp
|-- queue.h
|-- syntax_checker.cpp
|-- listqueue_tester.cpp
|-- arrayqueue_tester.cpp
|-- syntax_checker_tester.cpp
```

You are required to edit only `arrayqueue.cpp`, `listqueue.cpp`, and `syntax_checker.cpp`.

6 Compiling and Running

Compiling Commands

```
# Test Array Queue Implementation
g++ -std=c++11 arrayqueue_tester.cpp arrayqueue.cpp -o main
./main 1000

# Test List Queue Implementation
g++ -std=c++11 listqueue_tester.cpp listqueue.cpp -o main
./main 1000

# Test Syntax Checker Implementation
g++ -std=c++11 syntax_checker_tester.cpp syntax_checker.cpp -o main
./main
```

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 main` flag specifies the output executable name
- For Task 3, `std::stack` is part of the STL, so no additional linking is needed

7 Submission

Submission Process

1. Create folder named: `<your_id>` (e.g., `2405xxx`)
2. Include all the files provided in the skeleton code.
3. Compress as `<your_id>.zip` and upload to Moodle

Deadline: Jan 24, 2026, 10:00 PM

WARNING

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

✗ Do not use ChatGPT/Copilot.

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

Evaluation Policy

Task 1	
ArrayQueue Functions	30%
ArrayQueue Resize	5%
Task 2	
ListQueue Functions	35%
Task 3	
Syntax Checker Implementation	25%
Proper Stack Usage	5%

Good luck with your implementation!