MTE 140 – Assignment #2
Stack ADT and Queue ADT Implementation

Learning Objectives:

- After completing this assignment, students will be able to (1) implement Stack ADT that dynamically grows and shrinks in size, and (2) implement Queue ADT that is circular in nature.

Part1: Dynamic Stack Implementation [50 marks]

Let us implement the Stack ADT using a dynamically resizable array as an aptly called Dynamic Stack.

- The header file **`a2_dynamic_stack.hpp`**, which is included below, provides the structure of the **`DynamicStack`** class with declarations of member functions. Do not modify the signatures of any of these functions.

- Implement all of the member functions listed in **`a2_dynamic_stack.cpp`**.

- Both files can be downloaded from LEARN along with other A2-related files.

The details of the header file **`a2_dynamic_stack.hpp`** are as follows:

- **`StackItem`** defines the kind of data that the stack will contain. Being public, it can be accessed directly as **`DynamicStack::StackItem`**.

- **`StackItem EMPTY_STACK`** defines a constant that will be used to indicate an empty stack. Being public, it can be accessed directly as **`DynamicStack::EMPTY_STACK`**.

- Member variables:
  1. **`items_`**: An array of stack items.
  2. **`capacity_`**: Maximum number of elements allowed in the current stack.
  3. **`size_`**: Current number of elements in the stack.
  4. **`init_capacity_`**: Initial capacity of the array (i.e., the capacity used in the constructor). This is used by **`pop()`** to determine if we should decrease the capacity.

- Constructors and Destructor:
  1. **`DynamicStack()`**: Default constructor of the class **`DynamicStack`**. It uses 16 as the initial capacity of the array, and allocates the required memory space for the stack. The function appropriately initializes the fields of the created empty stack.
  2. **`DynamicStack(unsigned int capacity)`**: Parametric constructor of the class **`DynamicStack`**. It allocates the required memory space for the stack of

the given capacity. The function appropriately initializes the fields of the created empty stack.

3. **~DynamicStack()**: Destructor of the class **DynamicStack**. It deallocates the memory space allocated for the stack.

- Constant member functions:
    1. **int size() const:** Returns the number of items in the stack.
    2. **bool empty() const:** Returns **true** if the stack is empty, and **false** otherwise.
    3. **void print() const:** Prints the stack items sequentially and in order, from the top to the bottom of the stack.
    4. **StackItem peek() const:** Returns the value at the top of the stack without removing it. If the stack is empty, it returns the **EMPTY_STACK** constant instead.

- Non-constant member functions:
    1. **void push(StackItem value):** Takes as an argument a **StackItem** value. If the stack is not full, the value is pushed onto the stack. Otherwise, the capacity of the stack is doubled, and the item is then pushed onto the resized stack.
    2. **StackItem pop():** Removes and returns the top item from the stack as long as the stack is not empty. If the number of items remaining in the stack after popping is less than or equal to one quarter of the capacity of the array, then the array is halved. However, if the new halved capacity is less than the initial capacity, then no resizing takes place. Finally, If the stack is empty before the pop, the **EMPTY_STACK** constant is returned.

```
#ifndef A2_DYNAMIC_STACK_HPP
#define A2_DYNAMIC_STACK_HPP
class DynamicStack {
public:
      // Defines the kind of data that the stack will contain
      typedef int StackItem;
      // Defines a constant that will be used to indicate an empty stack
      static const StackItem EMPTY_STACK;

private:
      // Befriend so tests have access to variables.
      friend class DynamicStackTest;

      // MEMBER VARIABLES
      // An array of stack items.
      StackItem* items_;
      // Maximum number of elements allowed in the stack.
      int capacity_;
      // Current number of elements in the stack.
      int size_;
      // Initial capacity of the array (i.e., the capacity set in the constructor).
      // This is used by pop() to determine if we should decrease the capacity.
      int init_capacity_;

      // Copy constructor. Declared private so we don't use it incorrectly.
      DynamicStack(const DynamicStack& other) {}
      // Assignment operator. Declared private so we don't use it incorrectly.
      DynamicStack operator=(const DynamicStack& other) {}

public:
      // CONSTRUCTORS/DESTRUCTOR
      // Default constructor of the class DynamicStack. It uses 16 as the initial
      // capacity of the array, and allocates the required memory space for the
      // stack. The function appropriately initializes the fields of the created
      // empty stack.
      DynamicStack();

      // Parametric constructor of the class DynamicStack. It allocates the required
      // memory space for the stack of the given capacity. The function
      // appropriately initializes the fields of the created empty stack.
      DynamicStack(unsigned int capacity);

      // Destructor of the class DynamicStack. It deallocates the memory space
      // allocated for the stack.
      ~DynamicStack();

      // ACCESSORS
      // Returns the number of items in the stack.
      int size() const;

      // Returns true if the stack is empty and false otherwise.
      bool empty() const;

      // Prints the stack items sequentially and in order, from the top to the
      // bottom of the stack.
      void print() const;

      // Returns the value at the top of the stack without removing it. If the
      // stack is empty, it returns the EMPTY_STACK constant instead.
      StackItem peek() const;

      // MUTATORS
      // Takes as an argument a StackItem value. If the stack is not full, the value
      // is pushed onto the stack. Otherwise, the capacity of the stack is doubled,
      // and the item is then pushed onto the resized stack.
      void push(StackItem value);

      // Removes and returns the top item from the stack as long as the stack is
      // not empty. If the number of items remaining in the stack after popping
      // is less than or equal to one quarter of the capacity of the array, then
      // the array is halved. However, if the new halved capacity is less than
      // the initial capacity, then no resizing takes place. Finally, If the stack
      // is empty before the pop, the EMPTY STACK constant is returned.
      StackItem pop();
};
#endif
```

Part2: Circular Queue Implementation [50 marks]

Let us implement the Queue ADT using a circular array as an aptly called Circular Queue.

- The header file **`a2_circular_queue.hpp`**, which is included below, provides the structure of the **`CircularQueue`** class with declarations of member functions. Do not modify the signatures of any of these functions.

- Implement all of the member functions listed in **`a2_circular_queue.cpp`**.

- Both files can be downloaded from LEARN along with other A2-related files.

The details of the header file **`a2_circular_queue.hpp`** are as follows:

- **`QueueItem`** defines the kind of data that the queue will contain. Being public, it can be accessed directly as **`CircularQueue::QueueItem`**.

- **`QueueItem EMPTY_QUEUE`** defines a constant that will be used to indicate an empty queue. Being public, it can be accessed directly as **`CircularQueue::EMPTY_QUEUE`**.

- Member variables:
    1. **`items_`**: An array of queue items.
    2. **`head_:`** Index of the first element in the circular queue.
    3. **`tail_:`** Index of the element after the last item in the circular queue.
    4. **`capacity_`**: Maximum number of items in the queue.
    5. **`size_`**: Current number of items in the queue.

- Constructors and Destructor:
    1. **`CircularQueue():`** Default constructor of the class **`CircularQueue`**. It uses 16 as the initial capacity of the array, and allocates the required memory space for the queue. The function appropriately initializes the fields of the created empty queue.
    2. **`CircularQueue(unsigned int capacity)`**: Parametric constructor of the class **`CircularQueue`**. It allocates the required memory space for the queue of the given capacity. The function appropriately initializes the fields of the created empty queue.
    3. **`~CircularQueue()`**: Destructor of the class **`CircularQueue`**. It deallocates the memory space allocated for the queue.

- Constant member functions:
    1. **`int size() const:`** Returns the number of items in the queue.
    2. **`bool empty() const:`** Returns **`true`** if the queue is empty, and **`false`** otherwise.
    3. **`bool full() const:`** Returns true if the queue is full, and false otherwise.
    4. **`void print() const:`** Prints the queue items sequentially and in order, from the front to the rear of the queue.

5. **QueueItem peek() const:** Returns the value at the front of the queue without removing it from the queue. If the queue is empty, it returns the **EMPTY_QUEUE** constant instead.

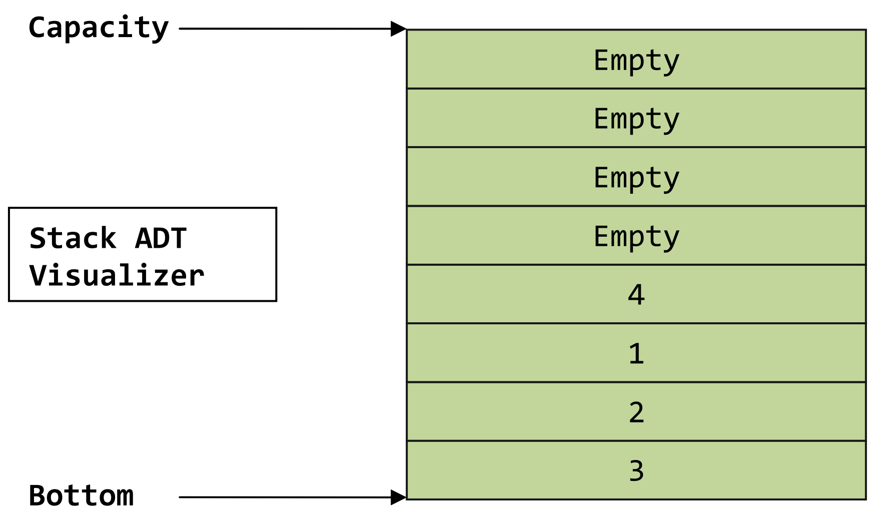
- Non-constant member functions:
  1. **bool enqueue(QueueItem value):** Takes as an argument a **QueueItem** value. If the queue is not at capacity, it inserts the value at the rear of the queue after the last item, and returns **true**. If the insertion fails due to lack of space, it returns **false**.
  2. **QueueItem dequeue():**Removes the item from the front of the queue and returns it. If the queue is empty, it returns the **EMPTY_QUEUE** constant instead.

```
#ifndef A2_CIRCULAR_QUEUE_HPP
#define A2_CIRCULAR_QUEUE_HPP

class CircularQueue {
public:
      // Defines the kind of data that the queue will contain.
      typedef int QueueItem;
      // Defines a constant that will be used to indicate an empty queue.
      static const QueueItem EMPTY_QUEUE;

private:
      friend class CircularQueueTest;

      // MEMBER VARIABLES
      // An array of queue items.
      QueueItem *items_;

      // Index of the first element in the circular queue.
      int head_;
      // Index of the element after the last item in the circular queue.
      int tail_;
      // Maximum number of items in the queue.
      int capacity_;
      // Current number of items in the queue.
      int size_;

      // Copy constructor. Declared private so we don't use it incorrectly.
      CircularQueue(const CircularQueue& other) {}
      // Assignment operator. Declared private so we don't use it incorrectly.
      CircularQueue operator=(const CircularQueue& other) {}

public:
      // CONSTRUCTORS/DESTRUCTOR
      // Default constructor of the class CircularQueue. It uses 16 as the
      // initial capacity of the array, and allocates the required memory
      // space for the queue. The function appropriately initializes the
      // fields of the created empty queue.
      CircularQueue();
      // Parametric constructor of the class CircularQueue. It allocates
      // the required memory space for the queue of the given capacity.
      // The function appropriately initializes the fields of the created
      // empty queue.
      CircularQueue(unsigned int capacity);
      // Destructor of the class CircularQueue. It deallocates the memory
      // space allocated for the queue.
      ~CircularQueue();

      // ACCESSORS
      // Returns the number of items in the queue.
      int size() const;

      // Returns true if the queue is empty and false otherwise.
      bool empty() const;

      // Returns true if the queue is full and false otherwise.
      bool full() const;

      // Prints the queue items sequentially and in order, from the front
      // to the rear of the queue.
      void print() const;

      // Returns the value at the front of the queue without removing it
      // from the queue. If the queue is empty, it returns the EMPTY_QUEUE
      // constant instead.
      QueueItem peek() const;

      // MUTATORS
      // Takes as an argument a QueueItem value. If the queue is not at capacity,
      // it inserts the value at the rear of the queue after the last item, and
      // returns true. If the insertion fails due to lack of space, it
      // returns false.
      bool enqueue(QueueItem value);

      // Removes the item from the front of the queue and returns it. If the queue
      // is empty, it returns the EMPTY_QUEUE constant instead.
      QueueItem dequeue();
};
#endif
```

Bonus: Stacks and Queues Visualizer [up to 10 marks]

Develop a data structure visualizer for your Part1 and Part2 implementation above. The visualizer needs to be called from each **`print()`** method to display data structure state. It needs to be able to handle visualization of all the test cases provided with the assignment, and needs to able to handle data structure state for the full duration of each test case.

More specifically, the visualizer should create a new window on the platform of your choice (e.g., Windows, Mac OS), and then draw the state of the data structure using corresponding graphical shapes. See example below for one such visualization.



You may not modify the implementation of your ADT methods, but you may include additional files to help implement the visualizer.

You may include animations and make the visualizer dynamic, where the state of the visualizer is updated after each mutator operation. Additional marks may be awarded for extra features.

On Windows, a simple API that can help you implement this visualizer is Windows API. Reference URL: http://zetcode.com/gui/winapi/

Alternatively, there are options available within specific IDEs that can help with the implementation, such as Visual Studio templates for Windows or Xcode templates for Mac OS.

Deliverables

- **Submit a single zipped archive named `studentid1_studentid2.zip` (e.g., 20000001_20000002.zip) with the files `a2_dynamic_stack.cpp` and `a2_circular_ queue.cpp` included inside.**

- **Do not modify `a2_dynamic_stack.hpp` for Part1 and `a2_circular_ queue.hpp` for Part2 of the assignment. If you do, we will not be able to test your code, and you will be awarded 0 for the missed test cases.**

- **Adequately document your code, especially mutator methods for Part1 and Part2.**

- **Ensure that all code is your own code and that it has not been copied from someone outside of your team. Copying code from other students without attribution or permission is a violation of academic integrity and Policy 71.**

- **If you choose to complete the bonus objective, then also include the corresponding .cpp and .hpp files in your zipped archive.**

- **Submit the zipped archive to the assignment dropbox on LEARN. Do *not* include any other files in the archive.**

- **Check – and double check – that your archive is not empty and that it contains all the required files. In the past, we had student accidentally submit empty archives, or archives that did not include all the required files.**

- **Before submitting your code, ensure that your code compiles with the gcc compiler used in the Engineering computer labs (i.e., v4.8.1), and that it runs without freezing or crashing against the Dev-C++ version used in the Engineering computer labs (i.e., v5.7.1); both the required compiler and the IDE can be found on the lab computers in CPH 1346.**

- **Write the names and IDs of both group members in the files that you submit as comments. Only one submission is required from each group.**

- **The assignment is due on Tue Jun 27th by 8:30am.**

- **No late submissions will be accepted.**