Homework 1

Due by: 10.7.2024, 23:59

General Guidelines

- See the general guidelines from HW(-1).
- Don't use any external libraries for this homework, except native C++11 libs.
- The homework assignment is 100% auto graded. You resubmit your solution to gradescope multiple times before the due date of this homework assignment.
- This homework assignment includes several .h files. Read ahead.
- Make sure to create new classes and files with the exact names defined ahead.

Exercise 1: parallel bounded queue – 30 points

Implement a bounded FIFO queue. All its memory should be allocated during queue creation.

1. The queue capacity should NOT grow at any point in time.
2. Implement a "*BoundedQueue"* C++ class which implements the bounded queue. It should inherit the abstract class in the *BoundedQueue.h* file (you are still allowed to extend your class with additional private member variables and functions).
3. Notice the semantics of pop and push, as defined in *BoundedQueue.h.*
4. Take good care to implement correct synchronization.

Upload *BoundedQueue_impl.h* file with your class implementation to gradescope.

Exercise 2: parallel bounded queue with one producer /one consumer – 20 points

Implement another version of a bounded FIFO queue which is known to be used by at most one producer (push) and one consumer (pop)

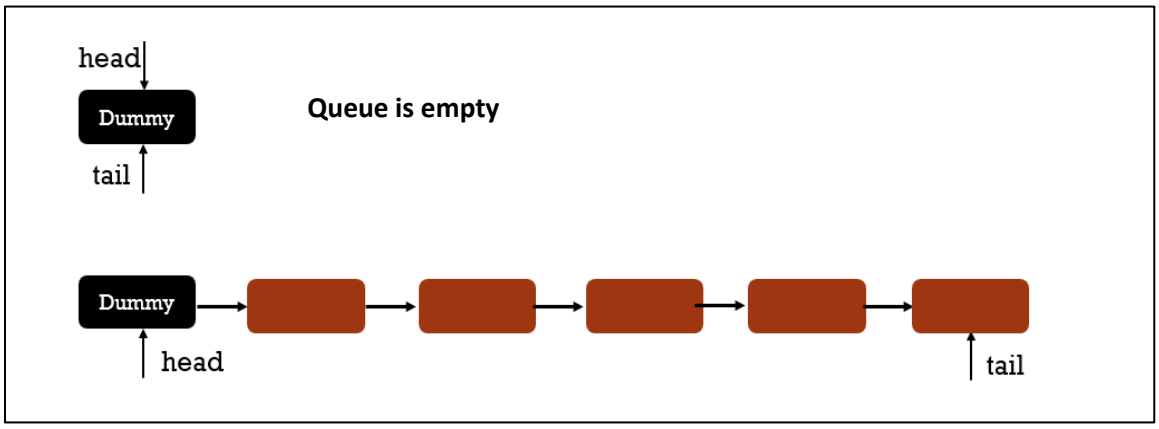
1. All its memory should be allocated during queue creation.
2. The queue capacity should NOT grow at any point in time.
3. Implement a "*BoundedQueue1p1c"* C++ class for this bounded queue. It should inherit the abstract class in the *BoundedQueue1p1c.h* file (you are still allowed to extend your class with additional private member variables and functions).
4. Notice the difference in behavior of *push* and *pop* compared with Exercise 1
   - With *pop*, if the queue is considered empty, return false. Otherwise, return true.
   - With *push*, if the queue is considered full, return false. Otherwise, return true.
5. Think hard how to leverage the assumption of one customer/producer to minimize synchronization.
   - You are NOT allowed to use any mechanism of blocked synchronization (e.g., mutexes) in your solution, whether directly (e.g., using C++ mutexes) or indirectly (i.e., implementing blocking locks by yourself)

Upload *BoundedQueue1p1c_impl.h* file with your class implementation to gradescope.

Exercise 3: parallel unbounded queue with one producer /one consumer – 40 points

Implement an unbounded FIFO queue with at most one producer and one customer.

1. Implement an "*UnboundedQueue1p1c*" C++ class for the unbounded queue. It should inherit the abstract class in the *UnboundedQueue1p1c.h* file (you are still allowed to extend your class with additional member variables and functions).
2. Notice the difference in behavior of *push* and *pop* compared with the other exercises:
   - With pop, if the queue is empty, return false. Otherwise, return true.
   - Push always succeeds.
3. Your implementation should be based on link lists in C (not C++). Any use of C++ STL containers or arrays is NOT allowed.
5. Think hard how to leverage the assumption of one customer/producer to minimize synchronization.
   - You are NOT allowed to use any mechanism of blocked synchronization, exactly as in exercise #2.
   - Hint: use a dummy node (see following figure)
     i. Upon queue initialization, it already contains one (dummy) element (no value inside). Moreover, during the queue lifetime, it may happen that other nodes will become dummy.
     ii. The queue never gets empty. Pop fails if the queue has only one element (dummy)
     iii. Tail points to latest element in the queue. Push is always done immediately after the tail element.
     iv. Pop is always done on head->next (NOT on head).
   - Convince yourself that the use of dummy elements helps you to reduce synchronization for empty queue (compare with tail==head==NULL)



Upload *UnboundedQueue1p1c_impl.h* file with your class implementation to gradescope.

Good luck!