1. Write pseudocode for:

a. The structure of a node (eg, define a struct node and its fields).

```
struct node {
    <T> value;// the value with type T at this node
    Node *pre;//pointers point to the previous node in list
    Node *next;//pointers point to the next node in list
    mutable std::mutex mu;// mutex for this node
}
```

b. The lookup() operation, which must take a key (of a templated type T) and return struct node* pointers to prev, cur, and next. Remember that prev is the node before the point where this key is or would be, cur is the node at or beyond the key in question, and if the key is found, next is the next node after the cur node. Remember that you need to grab the per- node locks during the lookup() operation.

```
template<class T>
Response<T> ConcurrentOrderedList<T>::lookup(const T key, bool
onlyLookup){
    head->mu.lock();

    Node<T> *ptr = head->next;
    Node<T> *parent = head;
    while (ptr) {
        if (ptr->pre != head) {
            ptr->pre->mu.lock();
        }
        ptr->mu.lock();

        if(ptr->value == key){ // Key found at *cur

            if (onlyLookup) {
                ptr->mu.unlock();
                ptr->pre->mu.unlock();
            }

            return Response<T>(ptr->pre, ptr, ptr->next, true);
        }
        else if(ptr->value > key){
            // Key not found. Should be between *pre and *cur

            if (onlyLookup) {
                ptr->mu.unlock();
                ptr->pre->mu.unlock();
            }
            return Response<T>(ptr->pre, ptr, ptr->next, false);
        }

        ptr->mu.unlock();
        ptr->pre->mu.unlock();
        // move to next node
        parent = ptr;
        ptr = ptr->next;
    }

    // Key not found. Should be between *pre and *cur
    if (onlyLookup) {
        parent->mu.unlock();
    }
    return Response<T>(parent, nullptr, nullptr, false);
}
```

c. The remove() and insert() operations, using the output of lookup(). Remember that these must unlock lookup()’s locks.

```
template<class T>
bool ConcurrentOrderedList<T>::insert(const T key){
    Response<T> response = lookup(key, false);
    if (response.found) {
        if (response.cur != nullptr) {
            response.cur->mu.unlock();
        }
        response.pre->mu.unlock();
        return false;
    }

    // Create a new node with key
    Node<T> *newNode = new Node<T>(key);
    newNode->pre = response.pre;
    newNode->next = response.cur;

    // Insert the new node into list
    response.pre->next = newNode;
    if (response.cur != nullptr) {
        response.cur->pre = newNode;
        response.cur->mu.unlock();
    }
    response.pre->mu.unlock();
    return true;
}

template<class T>
bool ConcurrentOrderedList<T>::remove(const T key){
    Response<T> response = lookup(key, false);
    if (!response.found) {
        if (response.cur != nullptr) {
            response.cur->mu.unlock();
        }
        response.pre->mu.unlock();
        return false;
    }

    // Remove the node with key from list
    response.pre->next = response.next;
    if (response.next != nullptr) {
        response.next->pre = response.pre;
    }
    delete response.cur;
    response.pre->mu.unlock();

    return true;
}
```

2. An easy start: write the C++ code corresponding to your struct node.

```
template<class E>
class Node{
public:
    Node(){
        pre = nullptr;
        next = nullptr;
    }
    Node(const E v){
        value = v;
        pre = nullptr;
        next = nullptr;
    }
    E value;
    Node *pre;
    Node *next;
    mutable std::mutex mu;
};
```

3. Write the C++ code for your lookup() operation.

```
template<class T>
Response<T> ConcurrentOrderedList<T>::lookup(const T key, bool
onlyLookup){
    /* Iterate through the list and find the key or the position for
inserting the key. Use a dummy head to present the head of the list,
and the first element in the list is stored from dummyHead->next */
    head->mu.lock();

    Node<T> *ptr = head->next;
    Node<T> *parent = head;
    while (ptr) {
        if (ptr->pre != head) {
            ptr->pre->mu.lock();
        }
        ptr->mu.lock();

        if(ptr->value == key){// Key found at *cur

            if (onlyLookup) {
                ptr->mu.unlock();
                ptr->pre->mu.unlock();
            }

            return Response<T>(ptr->pre, ptr, ptr->next, true);
        }
        else if(ptr->value > key){
            // Key not found. Should be between *pre and *cur

            if (onlyLookup) {
                ptr->mu.unlock();
                ptr->pre->mu.unlock();
            }

            return Response<T>(ptr->pre, ptr, ptr->next, false);
        }

        ptr->mu.unlock();
        ptr->pre->mu.unlock();
        // move to next node
```

```
            parent = ptr;
            ptr = ptr->next;
        }

        // Key not found. Should be between *pre and *cur
        if (onlyLookup) {
            parent->mu.unlock();
        }
        return Response<T>(parent, nullptr, nullptr, false);
    }
```

4. Write the C++ code for your remove() and insert() operations, using the output of lookup().

```
template<class T>
bool ConcurrentOrderedList<T>::insert(const T key){
    Response<T> response = lookup(key, false);
    if (response.found) {
        if (response.cur != nullptr) {
            response.cur->mu.unlock();
        }
        response.pre->mu.unlock();
        return false;
    }

    // Create a new node with key
    Node<T> *newNode = new Node<T>(key);
    newNode->pre = response.pre;
    newNode->next = response.cur;

    // Insert the new node into list
    response.pre->next = newNode;
    if (response.cur != nullptr) {
        response.cur->pre = newNode;
        response.cur->mu.unlock();
    }
    response.pre->mu.unlock();
    return true;
}

template<class T>
bool ConcurrentOrderedList<T>::remove(const T key){
    Response<T> response = lookup(key, false);
    if (!response.found) {
        if (response.cur != nullptr) {
            response.cur->mu.unlock();
        }
        response.pre->mu.unlock();
        return false;
    }

    // Remove the node with key from list
    response.pre->next = response.next;
    if (response.next != nullptr) {
        response.next->pre = response.pre;
    }
    delete response.cur;
    response.pre->mu.unlock();

    return true;
}
```

5. Explain how and why you handled unlocking lookup()'s locks, especially for a pure lookup() operation.

While traversing through the list, we always lock the previous node and the current node. The end point would be if the key is found at the current node or the value of the current node is larger than the key. Before reaching to the end point, we keep moving the current node pointer and the previous node pointer to the next (one step at a time), and release the locks before we move the pointers forward.

```
A ->  B  -> C
pre  cur
     pre   cur
```

For example,

1) lock(A), lock(B) : current points to B
2) lock(C), unlock(A) : current points to C

In the case of a pure lookup(), we unlock the locks of previous node and current node when we reach to the end point (before return). In the cases of lookup() for remove() and insert(), we keep the locks and unlocks them later in the remove()/insert() after the corresponding operation is done. And unlock them in the order of firstly current node and then previous node.

Bonus (up to 15% extra points on this homework): Actually compile this, including with a simple test harness to launch threads to insert and remove random elements. You may reuse any test harness you have written for labs in this class.

Please refer to the codes and makefile, readme under the directory of /homework3