7.1*a* Consider the implementation of a priority queue as an array of queues. With the implementation given in class, it is always necessary to begin the search for an empty queue starting at the queue corresponding to the highest priority.

Suppose we introduced a member variable `int current_top` which stores the index of highest priority non-empty queue. The push and top operations can be updated to use this member variaible. Update the `pop()` member function to correctly update this member variable after a pop operation.

```
template <typename Type, int M>
void Multiqueue<Type>::push( Type const &obj, int pri ) {
    if ( pri < 0 || pri >= M ) {
        throw illegal_argument();
    }

    queues[pri].push( obj );
    current_top = std::min( current_top, pri );
    ++count;
}

template <typename Type, int M>
Type Multiqueue<Type>::top() const {
    if empty() {
        throw underflow();
    }

    return queues[current_top].front();
}

template <typename Type, int M>
Type Multiqueue<Type>::pop() {
    if empty() {
        throw underflow();
    }









}
```

7.1*b* Suppose we use an AVL tree to implement a priority queue. Is the run time of the operations `push(…)`, `pop()`, and `top()` always $\Theta(\ln(n))$, or may some of these run in $O(\ln(n))$ time? Why?

7.1*c* Explain what the following class tries to accomplish.

```
template <typename Type>
class Lex_pair {
    private:
        int count_priority;
        static int count;

    public:
        Type priority;


    Lex_pair( Type const & );
    bool operator==( Lex_pair const &lp ) const;
    bool operator!=( Lex_pair const &lp ) const;
    bool operator< ( Lex_pair const &lp ) const;
    bool operator<=( Lex_pair const &lp ) const;
    bool operator> ( Lex_pair const &lp ) const;
    bool operator>=( Lex_pair const &lp ) const;
};

template <typename Type>
int Lex_pair<Type>::count = 0;

template <typename Type>
Lex_pair<Type>::Lex_pair( Type const &pri ):
priority( pri ),
count_priority( count++ ) {
    // only initializes variables
}

template <typename Type>
bool Lex_pair<Type>::operator==( Lex_pair const &lp ) const {
    return ( priority == lp.priority ) && ( count_priority == lp.count_priority );
}

template <typename Type>
bool Lex_pair<Type>::operator!=( Lex_pair const &lp ) const {
    return !( *this == lp );
}

template <typename Type>
bool Lex_pair<Type>::operator<( Lex_pair const &lp ) const {
    return ( priority < lp.priority ) ? true :
                ( priority == lp.priority ) && ( count_priority < lp.count_priority );
}

template <typename Type>
bool Lex_pair<Type>::operator<=( Lex_pair const &lp ) const {
    return (*this < lp) || (*this == lp);
}

template <typename Type>
bool Lex_pair<Type>::operator>( Lex_pair const &lp ) const {
    return lp < *this;
}

template <typename Type>
bool Lex_pair<Type>::operator>=( Lex_pair const &lp ) const {
    return lp <= *this;
}
```

7.1*d* Four of the member functions in the previous implementation simply call others. Which programming style is better, what is presented above or the following four variations? Note, the instructor does not have the *correct* answer for this question.

```
template <typename Type>
bool Lex_pair<Type>::operator!=( Lex_pair const &lp ) const {
    return !operator==( lp );
}

template <typename Type>
bool Lex_pair<Type>::operator<=( Lex_pair const &lp ) const {
    return operator<( lp ) || operator==( lp );
}

template <typename Type>
bool Lex_pair<Type>::operator>( Lex_pair const &lp ) const {
    return lp.operator<( *this );
}

template <typename Type>
bool Lex_pair<Type>::operator>=( Lex_pair const &lp ) const {
    return lp.operator<=( *this );
}
```