ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Questions 2.4.
Assistances and comments will be acknowledged.

3.4.1*a* What is the state of the stack after the following sequence of pushes and pops:

```
Deque<int> d;
d.push_front( 3 );
d.push_back( 5 );
d.push_front ( 2 );
d.push_front ( 15 );
d.push_back ( 42 );
d.pop_front ();
d.pop_front ();
d.push_back ( 14 );
d.push_front ( 7 );
d.pop_back ();
d.push_front ( 9 );
d.pop_back ();
d.pop_front ();
d.push_front ( 51 );
d.pop_front ();
d.pop_back ();
```

3.4.1*b* A deque could be implemented using a doubly linked list. Can swapping the member variables for the head and tail achieve an $\Theta(1)$ algorithm that will reverse the order of the elements in the deque?

3.4.1*c* An implementation could potentially use any of the following data structures to implement the Queue ADT. Indicate the best-case run times if we use the listed data structures in as optimal a sense as possible. Choose the implementation that minimizes the run time of as many operations as possible. In the case of arrays, you may have to move entries if the array is not full—an $\Theta(n)$ operation.

	Singly linked list a head pointer	Singly linked list with head and tail pointers	Doubly linked list with head and tail pointers	One-ended array	Two-ended array	Circular array
`void push_front(…)`						
`Type pop_front()`						
`Type front() const`						
`void push_back(…)`						
`Type pop_back()`						
`Type back() const`						

3.4.4*a* The implementation of a deque in the standard template library is a linked list of arrays of a fixed size. If a `push_front` operation is performed and the front array is full, a new array is allocated which

links to the current array at the front. A similar modification is made if a `push_back` operation occurs when the back array is full. Suppose that the capacity of each array is n. Suppose that a new element placed in a new array at the front of the deque is in location $n - 1$ while a new element placed in a new array at the back of the deque is in location 0. Describe an algorithm of how you would access the k^{th} entry from the front of the deque, and describe an algorithm of how you would access the k^{th} entry from the back of the deque. We will assume that the first and last elements would be the 0^{th} element from their end of the deque.

3.4.5*a* The following is an implementation of an iterator for your `Single_list` class. The constructor is private, it uses a default copy constructor, and it exports the operations `++` (both pre-autoincrement and post-autoincrement) `*`, `==`, and `!=`. It must declare Single_list to be a friend, as `begin()` and `end()` must call the private constructor.

```
template <typename Type>
class Single_list {
    private:
        Single_node<Type> *list_head;
        Single_node<Type> *list_tail;
        int node_count;

    public:
        class iterator {
            private:
                Single_node<Type> *current;

                iterator( Single_node<Type> *ptr ):current( ptr ) {
                    // empty constructor
                }
            public:
                iterator operator++();
                iterator operator++( int );
                Type &operator*();
                bool operator==( iterator const & );
                bool operator!=( iterator const & );

                friend class Single_list;
        };

        // ...

        iterator begin();
        iterator end();
};

iterator begin() {
    return iterator( head() );
}

iterator end() {
    return iterator( nullptr );
}
```

The functionality of the iterator is as follows:

```
int main() {
        Single_list<int> s;

        s.push_front( 3 );
        s.push_front( 7 );
        s.push_front( 4 );
        s.push_front( 5 );

        for ( Single_list<int>::iterator itr = s.begin(); itr != s.end(); ++itr ) {
                std::cout << *itr << std::endl;
        }

        return 0;
}
```

To achieve this functionality:

1. Pre-autoincrement must update the value of `current` to point to the next node so long, unless `current` is already assigned nullptr, in which case it stays unchanged. The operator returns `*this`; that is, a copy of itself.
2. Post-autoincrement must first make a copy of `*this`, then update the value of `current` as in pre-autoincrement, and then it returns the copy.
3. Dereference returns a reference to the `element` stored in the node pointed to by `current`.
4. Two iterators are equal if both store the member variables `current` store the same address.
5. Two iterators are unequal if both the member variables `current` store different addresses.

Implement these member functions.

```
// pre-autoincrement
template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::iterator::operator++() {
    // your implementation
}

// post-autoincrement
template <typename Type>
typename Single_list<Type>::iterator Single_list<Type>::iterator::operator++( int ) {
    // your implementation
}

template <typename Type>
Type &Single_list<Type>::iterator::operator*() {
    // your implementation
}

template <typename Type>
bool Single_list<Type>::iterator::operator==( Single_list<Type>::iterator const &itr ) {
    // your implementation
}

template <typename Type>
bool Single_list<Type>::iterator::operator!=( Single_list<Type>::iterator const &itr ) {
    // your implementation
}
```