```
// based of code by Weiss, DSAAC++
#ifndef LIST_COMPLETE_H
#define LIST_COMPLETE_H

#include <algorithm>
using namespace std;

template <typename T>
class List
{
  private:
    // The basic doubly linked list node.
    // Nested inside of List, can be public
    // because the Node is itself private
    struct Node
    {
        T  data;
        Node   *prev;
        Node   *next;

        Node( const T & d = T{ }, Node * p = nullptr, Node * n = nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( T && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };

  public:
    class const_iterator
    {
      public:

        // Public constructor for const_iterator.
        const_iterator( ) : current{ nullptr }
          { }

        // Return the T stored at the current position.
        // For const_iterator, this is an accessor with a
        // const reference return type.
        const T & operator* ( ) const
          { return retrieve( ); }

        const_iterator & operator++ ( )
        {
            current = current->next;
            return *this;
        }

        const_iterator operator++ ( int )
        {
            const_iterator old = *this;
            ++( *this );
            return old;
        }

        const_iterator & operator-- ( )
        {
            current = current->prev;
            return *this;
        }

        const_iterator operator-- ( int )
```

```
        {
            const_iterator old = *this;
            --( *this );
            return old;
        }

        bool operator== ( const const_iterator & rhs ) const
          { return current == rhs.current; }

        bool operator!= ( const const_iterator & rhs ) const
          { return !( *this == rhs ); }

      protected:
        Node *current;

        // Protected helper in const_iterator that returns the T
        // stored at the current position. Can be called by all
        // three versions of operator* without any type conversions.
        T & retrieve( ) const
          { return current->data; }

        // Protected constructor for const_iterator.
        // Expects a pointer that represents the current position.
        const_iterator( Node *p ) :  current{ p }
          { }

        friend class List<T>;
    };

    class iterator : public const_iterator
    {
      public:

        // Public constructor for iterator.
        // Calls the base-class constructor.
        // Must be provided because the private constructor
        // is written; otherwise zero-parameter constructor
        // would be disabled.
        iterator( )
          { }

        T & operator* ( )
          { return const_iterator::retrieve( ); }

        // Return the T stored at the current position.
        // For iterator, there is an accessor with a
        // const reference return type and a mutator with
        // a reference return type. The accessor is shown first.
        const T & operator* ( ) const
          { return const_iterator::operator*( ); }

        iterator & operator++ ( )
        {
            this->current = this->current->next;
            return *this;
        }

        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
```

```
        }

        iterator & operator-- ( )
        {
            this->current = this->current->prev;
            return *this;
        }

        iterator operator-- ( int )
        {
            iterator old = *this;
            --( *this );
            return old;
        }


      protected:
        // Protected constructor for iterator.
        // Expects the current position.
        iterator( Node *p ) : const_iterator{ p }
          { }

        friend class List<T>;
    };

  public:
    List( )
      { init( ); }

    ~List( )
    {
        clear( );
        delete head;
        delete tail;
    }

    List( const List & rhs )
    {
        init( );
        for( auto & x : rhs )
            push_back( x );
    }

    List & operator= ( const List & rhs )
    {
        List copy = rhs;
        std::swap( *this, copy );
        return *this;
    }


    List( List && rhs )
      : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
    {
        rhs.theSize = 0;
        rhs.head = nullptr;
        rhs.tail = nullptr;
    }

    List & operator= ( List && rhs )
    {
        std::swap( theSize, rhs.theSize );
```

```
        std::swap( head, rhs.head );
        std::swap( tail, rhs.tail );

        return *this;
    }

    // Return iterator representing beginning of list.
    // Mutator version is first, then accessor version.
    iterator begin( )
      { return iterator( head->next ); }

    const_iterator begin( ) const
      { return const_iterator( head->next ); }

    // Return iterator representing endmarker of list.
    // Mutator version is first, then accessor version.
    iterator end( )
      { return iterator( tail ); }

    const_iterator end( ) const
      { return const_iterator( tail ); }

    // Return number of elements currently in the list.
    int size( ) const
      { return theSize; }

    // Return true if the list is empty, false otherwise.
    bool empty( ) const
      { return size( ) == 0; }

    void clear( )
    {
        while( !empty( ) )
            pop_front( );
    }

    // front, back, push_front, push_back, pop_front, and pop_back
    // are the basic double-ended queue operations.
    T & front( )
      { return *begin( ); }

    const T & front( ) const
      { return *begin( ); }

    T & back( )
      { return *--end( ); }

    const T & back( ) const
      { return *--end( ); }

    void push_front( const T & x )
      { insert( begin( ), x ); }

    void push_back( const T & x )
      { insert( end( ), x ); }

    void push_front( T && x )
      { insert( begin( ), std::move( x ) ); }

    void push_back( T && x )
      { insert( end( ), std::move( x ) ); }
```

```
    void pop_front( )
      { erase( begin( ) ); }

    void pop_back( )
      { erase( --end( ) ); }

    // Insert x before itr.
    iterator insert( iterator itr, const T & x )
    {
        Node *p = itr.current;
        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ x, p->prev, p } );
    }

    // Insert x before itr.
    iterator insert( iterator itr, T && x )
    {
        Node *p = itr.current;
        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ std::move( x ), p->prev,
p } );
    }

    // Erase item at itr.
    iterator erase( iterator itr )
    {
        Node *p = itr.current;
        iterator retVal( p->next );
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        --theSize;

        return retVal;
    }

    iterator erase( iterator from, iterator to )
    {
        for( iterator itr = from; itr != to; )
            itr = erase( itr );

        return to;
    }

        // Added for CSE 330 Lab4

        iterator find(T x)
        {
                iterator itr = begin();
                while (itr != end())
                {
                        if (*itr == x)
                                return itr;
                        ++itr;
                }
                return end();
        }

        bool selforg_find(const T  x)
        {
                iterator found = find(x);
                if (found == end())
```

```
                        return false;
                erase(found);
                push_front(x);
                return true;
        }

        iterator& circulate(iterator start, int steps)
        {
                if (steps == 0)
                        return start;

                iterator itr = start;
                for (int i = 1; i <= steps; i++)
                {
                        cout << *itr << "->";
                        if (itr == end())
                                itr = begin();
                        else
                                ++itr;
                }
                cout << *itr << endl;
                return itr;
        }

  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};

#endif
```