```
#include <iostream>
#include <cstdlib>
using namespace std;

/*
 * To go to a singly linked list I basically
 * removed all references to previous and tail.
 * I no longer support operations to the end of the list
 * since they are so inefficient.
 * And now all functions that expect an iterator
 * assume that the pointer is to the node in front of the node
 * of interest
 *
 * So that brings up how to recognize an empty list.
 * If an empty list is identified by the next field being NULL
 * then how do we implement the iterator? One way to do this is
 * to create an empty_list constant, which is like [] in Haskell.
 * All lists "terminate" with the empty list.
 * That should really be a "class" constant, so there is really
 * ONLY one, but I just made it an instance constant.
 * Then an emptyList has a pointer to itself and is never modified,
 * but indicates the end of the list.
 *
 */

template <typename Object>
class List {
   private:
        struct Node {
        Object  data;
        Node   *next;

        Node( const Object & d = Object( ), Node *n = NULL )
          : data( d ), next( n ) {}
         };

   public:
        class const_iterator {
      public:
        const_iterator( ) : current( NULL )
          { }

        const Object & operator* ( ) const
          { return this->retrieve( ); }

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

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

  bool operator== ( const const_iterator & rhs ) const
    { return current->next == rhs.current->next ; }
  bool operator!= ( const const_iterator & rhs ) const
    { return !( *this == rhs ); }

protected:
  Node *current;

  Object & retrieve( ) const
    { /* assume that current has the node 'ahead' of
                      * of the node of interest */
                     return current->next->data; }

  const_iterator( Node *p ) : current( p )
    { }

  friend class List<Object>;
   };

   class iterator : public const_iterator {

public:
  iterator( )
    { }

  Object & operator* ( )
    { return this->retrieve( ); }

  const Object & operator* ( ) const
    { return const_iterator::operator*( ); }

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

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

```
        }

      protected:
        iterator( Node *p ) : const_iterator( p )
          {}

        friend class List<Object>;
    };

   public:
        List() { init( ); }

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

    List( const List & rhs )
    {
        init( );
        *this = rhs;
    }

    const List & operator= ( const List & rhs )
    {
                 /* Since I got rid of pushing onto the back, I had to
rewrite
                  * operator=, I used a private utility copy
                  */
        if( this == &rhs )
            return *this;
        clear( );
                 copy(rhs.head);
        return *this;
    }

/*
 * init initializes the head to a new Node, as in the
 * doubly linked list, and makes the next field point to the
 * emptList constant.
 */

    void init( )
    {
        head = new Node;
                 emptyList = new Node;
        head->next = emptyList;
        emptyList->next = emptyList;
         }
```

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

        const_iterator begin() const {
                return const_iterator( head);
        }

        iterator end() {
                return iterator(emptyList);
        }

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

        bool empty() const {
                return head->next == emptyList;
        }

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

    Object & front( )
      { return *begin( ); }
    const Object & front( ) const
      { return *begin( ); }
    void push_front( const Object & x )
      { insert( begin( ), x ); }
    void pop_front( )
      { erase( begin( ) ); }

    iterator insert( iterator itr, const Object & x )
    {
        Node *p = itr.current;
        return iterator( p->next = new Node(x, p->next));
    }

/*
 * Since the itr is pointing to the node ahead of what we
 * erase, we need to change the next field of that node
 * to skip the erased node.
 */

    iterator erase( iterator itr )
    {
                Node *p = itr.current;
```

```
                Node *del = p->next;
                p->next = p->next->next;
        iterator retVal( p );
        delete del;
        return retVal;
    }

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

        return to;
    }

/*
 * Lab 2 methods
 */

        int size() {
                Node *p = head;
                int sz=0;
                while (p->next != emptyList->next) {
                        sz++;
                        p=p->next;
                }
                return sz;
        }

        bool member(const Object & x) {
                iterator from(head);
                iterator to(emptyList);
                for(iterator il = from; il != to; il++) {
                        if (x == *il)
                                return true;
                }
                return false;
        }

        void add_not_member(const Object & x) {
                if (!member(x))
                        push_front(x);
        }

        void printList() {
                iterator from(head);
                iterator to(emptyList);
                for(iterator il = from; il != to; il++)
                        cout << *il << ", ";
                cout << endl;
```

```
        }

        void remove(const Object & x) {
                Node *q = head;
                Node *p = q->next;
                while (p != emptyList) {
                        if (x == p->data) {
                                q->next = p->next;
                                delete p;
                                return;
                        }
                        q=p;
                        p = p->next;
                }
                return;
        }


  private:
    Node *head;
        Node *emptyList;
/*
 * copy is private. The argument is the header node of the list
 * to be copied. Set the head, and then traverse the list,
 * allocating new nodes for each node in the argument.
 * Then set the emptyList indicator at the end;
 */
        void copy(Node * p) {
                Node *q = head;
                p = p->next;
                while (p != p->next) {   // emptyList indicator
                        q->next = new Node(p->data, NULL);
                        q = q->next;
                        p = p->next;
                }
                q->next = emptyList;
                return;
        }


};
```