CMSC 341

List 2

List.H

```
template <class Object>
class List
{
public:
   List();
   List(const List &rhs);
   ~List();
   const List &operator=(const List &rhs);
   bool isEmpty( ) const;
   void makeEmpty( );
   void remove (const Object &x);
   void insert (const Object &x,
                const ListItr<Object> &p);
```

List.H (cont)

```
   ListItr<Object> first( )const;
   ListItr<Object> zeroth( )const;
   ListItr<Object> find(const Object &x) const;
   ListItr<Object>
       findPrevious(const Object &x) const;
private:
   ListNode<Object> *header;
};
```

ListNode.H

```
template <class Object>
class ListNode {
  ListNode(const Object &theElement=Object( ),
           ListNode *n = NULL)
      :element(theElement), next(n)
      { /* no code */ }

  Object element;
  ListNode *next;

  friend class List<Object>;
  friend class ListItr<Object>;
};
```

ListItr.H

```
template <class Object>
class ListItr {
public:
   ListItr():current(NULL) { /* no code */ }
   bool isPastEnd() const
        {return current == NULL;}
   void advance( ) {
        if (!isPastEnd()) current = current->next;
   }
   const Object &retrieve() const {
        if (isPastEnd()) throw BadIterator();
        return current->element;
   }
private:
   ListNode<Object> *current;
   ListItr(ListNode<Object> *theNode)
        :current(theNode){ /* no code */}
   friend class List<Object>;
};
```

Linked List Implementation

```
// default constructor
template <class Object>
List<Object>::List( ) {
   header = new ListNode<Object>;
}


template <class Object>
bool List<Object>::isEmpty() const {
   return !(header->next);
}
```

Linked List Implementation (cont)

```
// construct an interator “pointing to” header
template <class Object>
ListItr<Object> List<Object>::zeroth( ) const {
   return ListItr<Object>(header);
}

// construct an iterator “pointing to” first data
// element
template <class Object>
ListItr<Object> List<Object>::first( ) const {
   return  ListItr<Object>(header->next);
}
```

Linked List Implementation (cont)

```
// insert at position after the one pointed
// to by the iterator
template <class Object>
void List<Object>::
insert(const Object &x, const ListItr<Object> &p)
{
  if (!p.isPastEnd()) //text uses p.current!=NULL
      p.current->next =
          new ListNode<Object>(x,p.current->next);
}
```

Linked List Implementation (cont)

```
// return iterator pointing to item
// that was found
template <class Object>
ListItr<Object> List<Object>::
find(const Object &x) const
{
   ListNode<Object> *p = header->next;
   while (p!=NULL && p->element !=x)
       p = p->next;
   return ListItr<Object>(p);
}
```

Linked List Implementation (cont)

```
// return iterator to the item before
// the one passed in
template <class Object>
ListItr<Object> List<Object>::
findPrevious( const Object &x ) const
{
  ListNode<Object> *p = header;
  while (p->next!=NULL && p->next->element != x)
      p = p->next;
  return ListItr<Object>(p);
}
```

Linked List Implementation (cont)

```
// remove specified item
template <class Object>
void List<Object>::
remove(const Object &x)
{
   ListItr<Object> p = findPrevious(x);
   if (p.current->next != NULL){
       ListNode<Object> *oldnode
                    = p.current->next;
       p.current->next = p.current->next->next;
       delete oldnode;
   }
}
```

Linked List Implementation (cont)

```
// remove all the data, but don't destroy
// the list
template <class Object>
void List<Object>::makeEmpty( )
{
  while (!isEmpty())
      remove (first().retrieve());
}
```

Linked List Implementation (cont)

```
// List destructor
template <class Object>
List<Object>::~List()
{
   makeEmpty();  //deletes all nodes except header
   delete header;
}
```

Linked List Implementation (cont)

```
// assignment operator with deep copy
template <class Object>
const List<Object>& List<Object>::
operator= (const List<Object> &rhs)
{
   if (this != &rhs){
       makeEmpty( );
       ListItr<Object> ritr = rhs.first();
       ListItr<Object> itr = zeroth();
       while (!ritr.isPastEnd()) {
             insert(ritr.retrieve(), itr);
             ritr.advance();
             itr.advance();
       }
   }
   return *this;
}
```