makefile

```
#################################################################
# Program:
#     Week 07, LIST
#     Brother Ercanbrack, CS235
# Author:
#     Yurii Vasiuk
# Summary:
#     The implementation of list and its use in the application
# Time:
#     <how long did it take to complete this program>?
#################################################################

#################################################################
# The main rule
#################################################################
a.out: list.h week07.o fibonacci.o
       g++ -o a.out week07.o fibonacci.o
       tar -cf week07.tar *.h *.cpp makefile

#################################################################
# The individual components
#      week07.o       : the driver program
#      fibonacci.o    : the logic for the fibonacci-generating function
#      <anything else?>
#################################################################
week07.o: list.h week07.cpp
       g++ -c week07.cpp

fibonacci.o: fibonacci.h fibonacci.cpp list.h
       g++ -c fibonacci.cpp
```

Commented [ES1]: Update!!

bigNumber.h

```
/***********************************************************************
* Header:
*    Big Number
* Summary:
*    This class will be used for calculating big Fibonacci numbers.
*    I need to overload +, =, and << operators
* Author
*    Yura Vasiuk
************************************************************************/

#ifndef BIGNUMBER_H
#define BIGNUMBER_H

#include "list.h"
#include <iostream>
using namespace std;

/*********************************************
* BIGNUMBER
*********************************************/
class BigNumber
{
public:
   // default constructor
   BigNumber() {}

   // non-default constructor
   BigNumber(int num) { myNumber.push_front(num); }

   // operator = (assign rhs big number to the lhs big number)
   Node<int> * & operator=(BigNumber rhs)
   {
      this->myNumber.getHead() = rhs.myNumber.getHead();
      return this->myNumber.getHead();
```

```
   }

   // operator + (add two big numbers, return the new big number)
   BigNumber & operator+(BigNumber rhs);

   // operator << (display the big number accordingly to the Test Bed requirements)
   friend ostream & operator<<(ostream & out, BigNumber rhs);

private:
   List<int> myNumber;
};

/************************************************
* BIGNUMBER :: ADDITION OPERATOR
* Add one big number to another
************************************************/
BigNumber & BigNumber::operator+(BigNumber rhs)
{
   BigNumber theSum = BigNumber(0);
   int sum = 0;
   int keep = 0;
   int carry = 0;
   ListIterator<int> itTHIS = this->myNumber.rbegin();
   ListIterator<int> itRHS = rhs.myNumber.rbegin();

   while (itTHIS != NULL || itRHS != NULL)
   {
      keep = 0;
      sum = *itTHIS + *itRHS + carry;
      if (sum > 999)
      {
         keep %= 1000;
         carry = keep / 1000;
      }
      else
      {
         keep = sum;
         carry = 0;
      }
      theSum.myNumber.push_front(keep);
      // move the iterators
      --itTHIS;
      --itRHS;
   }
   // the last addition
   if (itTHIS == NULL && itRHS != NULL)
   {
      keep = *itRHS + carry;
      theSum.myNumber.push_front(keep);
      --itTHIS;
   }
   if (itRHS == NULL && itTHIS != NULL)
   {
      keep = *itTHIS + carry;
      theSum.myNumber.push_front(keep);
      --itRHS;
   }

   // keep going if there data to push
   while (itTHIS != NULL)
   {
      keep = *itTHIS;
      theSum.myNumber.push_front(keep);
      --itTHIS;
   }
   while (itRHS != NULL)
   {
      keep = *itRHS;
      theSum.myNumber.push_front(keep);
      --itRHS;
   }

   return theSum;
};

/************************************************
* BIGNUMBER :: OUTPUT OPERATOR
* Output the big number
************************************************/
```

Commented [ES2]: This is incorrect.
It should return a BigNumber not a Node<T>.

```
BigNumber & operator = (BigNumber rhs)
{
   myNumber = rhs;
   return *this;
}
```

```
ostream & operator<<(ostream & out, BigNumber rhs)
{
   ListIterator<int> it = rhs.myNumber.begin();

   while (it != NULL)
   {
      out << *it;
      --it;
      if (it != NULL)
         out << ", ";
   }

   return out;
};

#endif // BIGNUMBER_H
```

Commented [ES3]: You need to handle leading zeros.
Use setfill('0') and setW(3)

fibonacci.h

```
/*************************************************************************
 * Header:
 *    FIBONACCI
 * Summary:
 *    This will contain just the prototype for fibonacci(). You may
 *    want to put other class definitions here as well.
 * Author
 *    Yurii Vasiuk
 ************************************************************************/

#ifndef FIBONACCI_H
#define FIBONACCI_H

// the interactive fibonacci program
void fibonacci();

#endif // FIBONACCI_H
```

list.h

```
/*************************************************************************
* Header:
*    List
* Summary:
*    Custom made List analogous to the std::list
*    The class will use Node and ListIterator classes
* Author
*    Yura Vasiuk
************************************************************************/

#ifndef LIST_H
#define LIST_H

#include <iostream>
using namespace std;

template <typename T>
class ListIterator;

/*********************************************
* NODE
* A class to be used in LinkedList
*********************************************/
template <typename T>
class Node
{
public:
   T data;
   Node<T> * pNext;
   Node<T> * pPrev;

   Node() : pNext(NULL), pPrev(NULL) {}
   Node(T data) { this->data = data; this->pNext = NULL; this->pPrev = NULL; }
   Node(T data, Node<T> * pNext, Node<T> * pPrev)
   {
      this->data = data; this->pNext = pNext; this->pPrev = pPrev;
   }
};
```

```
/*********************************************
* LIST
* Custom made List, analogous to the std::list
*********************************************/
template <typename T>
class List
{
public:
   // default constructor
   List() : numItems(0), pHead(NULL) , pTail(NULL) {}     // done

   // copy constructor : copy it
   List(const List<T> & rhs) throw (const char *);        // done

   // destructor
   ~List() { clear(); }                                   // done

   // assignment operator
   List<T> & operator=(const List<T> & rhs) throw (const char *);  // done

   // check if empty
   bool empty() { return numItems == 0; }                 // done

   // what is the number of items in the list
   int size() { return numItems; }                        // done

   // empy the list of all the items
   void clear();                                          // done

   // add an item to the back of the list
   void push_back(T t) throw (const char *);              // done

   // add an item to the front of the list
   void push_front(T t) throw (const char *);             // done

   // returnt the element at the front of the list
   T & front() throw (const char *);                      // done

   // return the element at the back of the list
   T & back() throw (const char *);                       // done

   // return the interator to the front of the list
   ListIterator<T> begin() { return ListIterator<T>(pHead); }   // done

   // return the interator to the back of the list
   ListIterator<T> rbegin() { return ListIterator<T>(pTail); }  // done

   // return the iterator to the past-the-front of the list
   ListIterator<T> rend() { return NULL; }                // done

   // return the iterator to the past-the-back of the list
   ListIterator<T> end() { return NULL; }                 // done

   // insert the passed item before the passed pointer
   ListIterator<T> insert(ListIterator<T> pInsertBefore, T t) throw (const char *);

   // remove the item at the passed poiter
   void remove(ListIterator<T> pRemoveHere) throw (const char *);

   // I will need it for BigNumber operator=
   Node<T> * & getHead() { return pHead; }

private:
   int numItems;
   Node<T> * pHead;
   Node<T> * pTail;
};

/*********************************************
* LIST :: COPY CONSTRUCTOR
* Create a new List and copy the data into it
*********************************************/
template <typename T>
List<T> :: List(const List<T> & rhs) throw (const char *)
{
   pHead = NULL;
   pTail = NULL;
   numItems = 0;
```

```
   // nothing to do
   if (rhs.pHead == NULL)
      return ;

   // axiliary pointers
   Node<T> * pTraverseOld = NULL;
   Node<T> * pTraverseNew = NULL;
   Node<T> * pTemp = NULL;

   // make the first node
   pHead = new Node<T>;
   pTail = pHead;
   // assign pointers
   pHead->pNext = NULL;
   pHead->pPrev = NULL;
   // assign traverses
   pTraverseNew = pHead;
   pTraverseOld = rhs.pHead;

   while (pTraverseOld->pNext != NULL)
   {
      // fill the new node data
      pTraverseNew->data = pTraverseOld->data;
      // making a one more node, fill the new node address
      pTraverseNew->pNext = new Node<T>;
      // temp
      pTemp = pTraverseNew;
      // move the traverses
      pTraverseOld = pTraverseOld->pNext;
      pTraverseNew = pTraverseNew->pNext;
      // assign pPrev of the current last node and increase numItems
      pTraverseNew->pPrev = pTemp;
      numItems++;
   }
   // fill the last node data and increase numItems
   pTraverseNew->data = pTraverseOld->data;
   numItems++;
   // assign the pTail
   pTail = pTraverseNew;
}

/*************************************************
* LIST :: ASSIGNMENT OPERATOR
* Copy the data into the list
*************************************************/
template <typename T>
List<T> & List<T> :: operator=(const List<T> & rhs) throw (const char *)
{
   // clear the current list
   this->clear();

   // nothing to do
   if (rhs.pHead == NULL)
      return * this;

   // axiliary pointers
   Node<T> * pTraverseOld = NULL;
   Node<T> * pTraverseNew = NULL;
   Node<T> * pTemp = NULL;

   // make the first node
   this->pHead = new Node<T>;
   this->pTail = this->pHead;
   // assign pointers
   this->pHead->pNext = NULL;
   this->pHead->pPrev = NULL;
   // assign traverses
   pTraverseNew = this->pHead;
   pTraverseOld = rhs.pHead;

   while (pTraverseOld->pNext != NULL)
   {
      // fill the new node data
      pTraverseNew->data = pTraverseOld->data;
      // making a one more node, fill the new node address
      pTraverseNew->pNext = new Node<T>;
      // temp
      pTemp = pTraverseNew;
```

```
      // move the traverses
      pTraverseOld = pTraverseOld->pNext;
      pTraverseNew = pTraverseNew->pNext;
      // assign pPrev of the current last node and increase numItems
      pTraverseNew->pPrev = pTemp;
      numItems++;
   }
   // fill the last node data and increase numItems
   pTraverseNew->data = pTraverseOld->data;
   numItems++;
   // assign the pTail
   this->pTail = pTraverseNew;

   return * this;
}

/************************************************
* LIST :: CLEAR
* Empty the list of all the items
************************************************/
template <typename T>
void List<T> :: clear()
{
   // there is nothing to delete
   if (pHead == NULL)
      return;

   // axiliary pointers
   Node<T> * pDelete = pHead;
   Node<T> * pTraverse = pHead->pNext;

   while (pTraverse != NULL)
   {
      delete pDelete;
      pDelete = pTraverse;
      pTraverse = pTraverse->pNext;
   }
   // delete the last node and set the head and tail to NULL
   delete pDelete;
   pHead = NULL;
   pTail = NULL;
   // last thing to do
   numItems = 0;
}

/************************************************
* LIST :: PUSH_BACK
* Add an item to the back of the list
************************************************/
template <typename T>
void List<T> :: push_back(T t) throw (const char *)
{
   // attempt to allocate a new node
   try
   {
      if (empty())
      {
         pHead = new Node<T>();
         pTail = pHead;
      }
      else
         pTail->pNext = new Node<T>();
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // case 1) only one node in the list
   if (empty())
   {
      pHead->data = t;
      numItems++;
   }
   // case 2) reassign pTail, assign pPrev, fill with data, add numItems
   else
   {
      Node<T> * temp = pTail;
      pTail = pTail->pNext;
```

```
      pTail->pPrev = temp;
      pTail->data = t;
      numItems++;
   }
}


/*************************************************
* LIST :: PUSH_FRONT
* Add an item to the front of the list
*************************************************/
template <typename T>
void List<T> :: push_front(T t) throw (const char *)
{
   // temporary pointer
   Node<T> * temp;
   // attempt to allocate a new node, switch the pointers
   try
   {
      if (empty())
      {
         pHead = new Node<T>();
         pTail = pHead;
      }
      else
      {
         temp = pHead;
         pHead = new Node<T>();
         pHead->pNext = temp;           // the first node pointer
         pHead->pNext->pPrev = pHead;   // the second node pointer
      }
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // finally, fill the data and increment the numItems
   pHead->data = t;
   numItems++;
}

/*************************************************
* LIST :: FRONT
* Return the element at the front of the list
*************************************************/
template <typename T>
T & List<T> :: front() throw (const char *)
{
   if (empty())
      throw "ERROR: unable to access data from an empty list";
   else
      return pHead->data;
}

/*************************************************
* LIST :: BACK
* Return the element at the back of the list
*************************************************/
template <typename T>
T & List<T> ::back() throw (const char *)
{
   if (empty())
      throw "ERROR: unable to access data from an empty list";
   else
      return pTail->data;
}

/*************************************************
* LIST :: INSERT
* Insert the passed element before the passed pointer
* and return the pointer to the inserted element
*************************************************/
template <typename T>
ListIterator<T> List<T> :: insert(ListIterator<T> pInsertBefore, T t) throw (const char *)
{
   // convert the pointer from ListIterator to Node type
   Node<T> * pNodeInsertBefore = pInsertBefore.p;
```

```
   Node<T> * temp;
   // attempt to allocate a new node
   try
   {
      temp = new Node<T>(t);
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: unable to allocate a new node for a list";
   }

   // special cases
   // 1) insert after the list
   if (pNodeInsertBefore == NULL)
   {
      push_back(t);
      return NULL;
   }
   // 2) insert before the list
   if (pNodeInsertBefore->pPrev == NULL)
   {
      push_front(t);
      return NULL;
   }

   // usual cases
   // case 1) insert into empty list
   if (pHead == 0)
   {
      pHead = temp;
      pTail = temp;
   }
   // case 2) insert before the last node
   else if (pNodeInsertBefore == pTail)
   {
      pTail->pPrev->pNext = temp;
      temp->pPrev = pTail->pPrev;
      temp->pNext = pTail;
      pTail->pPrev = temp;
   }
   // case 3, 4) the rest: insert in the middle, insert before the first node
   else
   {
      temp->pNext = pNodeInsertBefore->pPrev->pNext;
      temp->pPrev = pNodeInsertBefore->pPrev;
      pNodeInsertBefore->pPrev = temp;
      // before the first, repoint the pHead
      if (pNodeInsertBefore == pHead)
         pHead = temp;
      else
         temp->pPrev->pNext = temp;
   }

   // the last thing to do
   numItems++;

   return NULL;
}

/*************************************************
* LIST :: REMOVE
* Remove the element at the passed pointer
*************************************************/
template <typename T>
void List<T> :: remove(ListIterator<T> pRemoveHere) throw (const char *)
{
   // nothing to remove
   if (pRemoveHere == end())
      throw "ERROR: unable to remove from an invalid location in a list";

   // convert the pointer from ListIterator to Node type
   Node<T> * pNodeRemoveHere = pRemoveHere.p;

   // reassign the pointers
   // case 1) remove the fist node
   if (pHead == pNodeRemoveHere)
   {
      pNodeRemoveHere->pNext->pPrev = NULL;
      pHead = pHead->pNext;
```

```
   }
   // case 2) remover the last node
   else if (pTail == pNodeRemoveHere)
   {
      pTail = pTail->pPrev;
      pTail->pNext = NULL;
   }
   // case 3) the rest, remove in the middle
   else
   {
      pNodeRemoveHere->pPrev->pNext = pNodeRemoveHere->pNext;
      pNodeRemoveHere->pNext->pPrev = pNodeRemoveHere->pPrev;
   }
   // delete the node
   delete pNodeRemoveHere;

   // last thing to do
   numItems--;
}

/*************************************************
* LIST ITERATOR
* An iterator through List
*************************************************/
template <typename T>
class ListIterator
{
public:
   // default constructor
   ListIterator() : p(0x00000000) {}

   // initialize to direct p to some item
   ListIterator(Node<T> * p) : p(p) {}

   // copy constructor
   ListIterator(const ListIterator<T> & rhs) { *this = rhs; }

   // assignment operator
   ListIterator<T> & operator = (const ListIterator<T> & rhs)
   {
      this->p = rhs.p;
      return *this;
   }

   // not equals operator
   bool operator != (const ListIterator<T> & rhs) const
   {
      return rhs.p != this->p;
   }

   // equals operator
   bool operator == (const ListIterator<T> & rhs) const
   {
      return rhs.p == this->p;
   }

   // dereference operator
   T & operator * ()
   {
      return p->data;
   }

   // prefix increment
   ListIterator<T> & operator++()
   {
      //p++;
      p = p->pNext;
      return *this;
   }

   // prefix decrement
   ListIterator<T> & operator--()
   {
      //p++;
      p = p->pPrev;
      return *this;
   }

   // these two functions will need the access to the iterator's private *p
```

```
   friend ListIterator<T> List<T>::insert(ListIterator<T> pInsertBefore, T t);
   friend void List<T>::remove(ListIterator<T> pRemoveHere);

private:
   Node<T> * p;
};

#endif // LIST_H
```

fibonacci.cpp

```
/***********************************************************************
 * Implementation:
 *    FIBONACCI
 * Summary:
 *    This will contain the implementation for fibonacci() as well as any
 *    other function or class implementations you may need
 * Author
 *    <your names here>
 ***********************************************************************/

#include <iostream>
#include "fibonacci.h"   // for fibonacci() prototype
#include "list.h"        // for LIST
#include "bigNumber.h"   // for big number class
using namespace std;

/*********************************************
* CALCULATEFIBONACCI
* Calculate and return the passed Fibonacci number
*********************************************/
int calculateFibonacci(int num)
{
   int fibonacci = 0;
   int temp1 = 0;
   int temp2 = 1;

   for (int i = 1; i <= num; i++)
   {
      if (i == 1)
         fibonacci = 1;
      else
      {
         fibonacci = temp1 + temp2;
         temp1 = temp2;
         temp2 = fibonacci;
      }
   }

   return fibonacci;
}

/*********************************************
 * FIBONACCI
 * The interactive function allowing the user to
 * display Fibonacci numbers
 *********************************************/
void fibonacci()
{
   // show the first serveral Fibonacci numbers
   int number;
   cout << "How many Fibonacci numbers would you like to see? ";
   cin  >> number;

   // your code to display the first <number> Fibonacci numbers
   for (int i = 1; i <= number; i++)
      cout << "\t" << calculateFibonacci(i) << endl;

   // prompt for a single large Fibonacci
   cout << "Which Fibonacci number would you like to display? ";
   cin  >> number;

   // your code to display the <number>th Fibonacci number
   cout << "\t" << calculateFibonacci(number) << endl;
}
```

Commented [ES4]: This is correct because you didn't use your linked list and this won't handle the big numbers.

week07.cpp

```
/**********************************************************************
* Program:
*    Week 07, LIST
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise the List class.  When you
*    submit your program, this should not be changed in any way.  That being
*    said, you may need to modify this once or twice to get it to work.
**********************************************************************/

#include <iostream>     // for CIN and COUT
#include <iomanip>      // for SETW
#include <string>       // for the String class
#include <cassert>      // for ASSERT
#include "list.h"       // your List class should be in list.h
#include "fibonacci.h"  // your fibonacci() function
using namespace std;


// prototypes for our four test functions
void testSimple();
void testPush();
void testIterate();
void testInsertRemove();

// To get your program to compile, you might need to comment out a few
// of these. The idea is to help you avoid too many compile errors at once.
// I suggest first commenting out all of these tests, then try to use only
// TEST1.  Then, when TEST1 works, try TEST2 and so on.
#define TEST1   // for testSimple()
#define TEST2   // for testPush()
#define TEST3   // for testIterate()
#define TEST4   // for testInsertRemove()

/**********************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 **********************************************************************/
int main()
{
   // menu
   cout << "Select the test you want to run:\n";
   cout << "\t1. Just create and destroy a List\n";
   cout << "\t2. The above plus push items onto the List\n";
   cout << "\t3. The above plus iterate through the List\n";
   cout << "\t4. The above plus insert and remove items from the list\n";
   cout << "\ta. Fibonacci\n";

   // select
   char choice;
   cout << "> ";
   cin  >> choice;
   switch (choice)
   {
      case 'a':
         fibonacci();
         break;
      case '1':
         testSimple();
         cout << "Test 1 complete\n";
         break;
      case '2':
         testPush();
         cout << "Test 2 complete\n";
         break;
      case '3':
         testIterate();
         cout << "Test 3 complete\n";
         break;
      case '4':
         testInsertRemove();
         cout << "Test 4 complete\n";
         break;
      default:
```

```
         cout << "Unrecognized command, exiting...\n";
   }

   return 0;
}

/*****************************************
 * TEST SIMPLE
 * Very simple test for a List: create and destroy
 *****************************************/
void testSimple()
{
#ifdef TEST1
   try
   {
      cout.setf(ios::fixed | ios::showpoint);
      cout.precision(5);

      // Test 1.a: a bool List with default constructor
      cout << "Create a bool List using the default constructor\n";
      List <bool> l1;
      cout << "\tSize:  " << l1.size()                  << endl;
      cout << "\tEmpty? " << (l1.empty() ? "Yes" : "No") << endl;

      // Test 1.b: double List and add one element
      cout << "Create a double List and add one element: 3.14159\n";
      List <double> l2;
      l2.push_back(3.14159);
      cout << "\tSize:  " << l2.size()                  << endl;
      cout << "\tEmpty? " << (l2.empty() ? "Yes" : "No") << endl;
      cout << "\tFront: " << l2.front()                 << endl;
      cout << "\tBack:  " << l2.back()                  << endl;

      {
         // Test 1.c: copy the double List
         cout << "Copy the double List using the copy-constructor\n";
         List <double> l3(l2);
         cout << "\tSize:  " << l3.size()                  << endl;
         cout << "\tEmpty? " << (l3.empty() ? "Yes" : "No") << endl;
         cout << "\tFront: " << l3.front()                 << endl;
         cout << "\tBack:  " << l3.back()                  << endl;
      }

      // test 1.d: Copy the List using the assignment operator
      cout << "Copy a double List using the assignment operator\n";
      List <double> l4;
      l4.push_back(1.0); // this node will get destroyed with the =
      l4 = l2;
      cout << "\tSize:  " << l4.size()                  << endl;
      cout << "\tEmpty? " << (l4.empty() ? "Yes" : "No") << endl;
      cout << "\tFront: " << l4.front()                 << endl;
      cout << "\tBack:  " << l4.back()                  << endl;
   }
   catch (const char * error)
   {
      cout << error << endl;
   }
#endif //TEST1
}

/*****************************************
 * TEST PUSH
 * Add a whole bunch of items to the List.  This will
 * test the push_back() and push_front() algorithm
 *****************************************/
void testPush()
{
#ifdef TEST2
   try
   {
      // create
      List <int> l1;
      int size1 = 0;

      // test push_back
      cout << "Enter integer values to put on the back, type 0 when done\n";
      int value;
      do
      {
```

```
         if (l1.empty())
            cout << "\t(   ...   ) > ";
         else
            cout << "\t( "
                 << l1.front()
                 << " ... "
                 << l1.back()
                 << " ) > ";
         cin  >> value;
         if (value)
         {
            l1.push_back(value);
            size1++;
         }
      }
      while (value);

      // test copy
      List <int> l2(l1);
      assert(l1.size() == l2.size());
      cout << "Copied l1 into l2\n";

      // modify the front and back of l1 and l2
      if (!l1.empty())
      {
         assert(l1.back()  == l2.back());
         assert(l1.front() == l2.front());
         l1.back()  =  42;
         l1.front() = -42;
      }
      cout << "Modified l1\n";

      // test empty
      l2.clear();
      int size2 = 0;
      cout << "Copied list l2 is "
           << (l2.empty() ? "empty." : "not empty.") << endl;

      // test push_front
      cout << "Enter integer values to put on the front, type 0 when done\n";
      do
      {
         if (l2.empty())
            cout << "\t(   ...   ) > ";
         else
            cout << "\t( "
                 << l2.front()
                 << " ... "
                 << l2.back()
                 << " ) > ";
         cin  >> value;
         if (value)
         {
            l2.push_front(value);
            size2++;
         }
      }
      while (value);

      // make sure the original list is not changed
      assert(l1.size() == size1);
      assert(l2.size() == size2);
      cout << "Sizes of l1 and l2 are correct\n";

      // make sure that l1 was not changed
      if (!l1.empty())
      {
         assert(l1.back()  ==  42);
         assert(l1.front() == -42);
      }
      cout << "The list l1 was unchanged\n";
   }
   catch (const char * error)
   {
      cout << error << endl;
   }
#endif // TEST2
}
```

```
/*************************************
 * LIST :: DISPLAY
 * Display the contents of the list forwards
 *************************************/
template <class T>
ostream & operator << (ostream & out, List <T> & rhs)
{
   out << '{';

#ifdef TEST3
   ListIterator <T> it;
   for (it = rhs.begin(); it != rhs.end(); ++it)
      out << "  " << *it;
#endif // TEST3

   out << "  }";

   return out;
}

/*************************************
 * TEST ITERATE
 * We will test the iterators. We will go through the
 * list forwards and backwards
 *************************************/
void testIterate()
{
#ifdef TEST3
   // create
   cout << "Create a string List with the default constructor\n";
   List <string> l;

   // instructions
   cout << "Instructions:\n"
        << "\t+ dog   pushes dog onto the front\n"
        << "\t- cat   pushes cat onto the back\n"
        << "\t#       displays the contents of the list backwards\n"
        << "\t*       clear the list\n"
        << "\t!       quit\n";

   char command;
   string text;
   do
   {
      cout << l << " > ";
      cin  >> command;

      try
      {
         switch (command)
         {
            case '+':
               cin >> text;
               l.push_front(text);
               break;
            case '-':
               cin >> text;
               l.push_back(text);
               break;
            case '#':
               cout << "\tBackwards: {";
               for (ListIterator <string> it = l.rbegin();
                    it != l.rend();
                    --it)
                  cout << "  " << *it;
               cout << "  }\n";
               break;
            case '*':
               l.clear();
               break;
            case '!':
               // do nothing, we will exit out of the loop
               break;
            default:
               cout << "Unknown command\n";
               cin.ignore(256, '\n');
         }
      }
      catch (const char * e)
```

```
      {
         cout << '\t' << e << endl;
      }
   }
   while (command != '!');
#endif // TEST3
}

/****************************************
 * TEST INSERT REMOVE
 * We will insert items in a list from the location
 * specified by the iterator, and remove items from
 * the list from the given iterator
 ****************************************/
void testInsertRemove()
{
#ifdef TEST4
   // first, fill the list
   List <char> l;
   for (char letter = 'a'; letter <= 'm'; letter++)
      l.push_back(letter);

   // instructions
   cout << "Instructions:\n"
        << "\t+ 3 A  put 'A' after the 3rd item in the list\n"
        << "\t- 4    remove the fourth item from the list\n"
        << "\t!       quit\n";

   char command;
   do
   {
      ListIterator <char> it;
      int index = 0;
      char letter;

      // display the list with indicies in the row above
      cout << ' ';
      for (it = l.begin(); it != l.end(); ++it)
         cout << setw(3) << index++;
      cout << endl;
      cout << l << endl;

      // prompt for the next command
      cout << "> ";
      cin  >> command;

      try
      {
         switch (command)
         {
            case '+':
               cin >> index >> letter;
               it = l.begin();
               while (index-- > 0)
                  ++it;
               l.insert(it, letter);
               break;
            case '-':
               cin >> index;
               it = l.begin();
               while (index-- > 0)
                  ++it;
               l.remove(it);
               break;
            case '!':
               break;
            default:
               cout << "Unknown command\n";
               break;
         }

         // error recovery: unexpected input
         if (cin.fail())
         {
            cin.clear();
            cin.ignore(256, '\n');
         }

      }
```

```
      // error recovery: thrown exception
      catch (const char * e)
      {
         cout << '\t' << e << endl;
      }
   }
   while (command != '!');
#endif // TEST4
}
```

Test Bed Results

a.out:

```
------------------------------------------------------------------
Starting Test 1

   > Select the test you want to run:
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   >    a. Fibonacci
   > > 1

This is little more than a test to see if the code can compile
   > Create a bool List using the default constructor
   >    Size:  0
   >    Empty? Yes

Using push_back(), we have a one element list
   > Create a double List and add one element: 3.14159
   >    Size:  1
   >    Empty? No
   >    Front: 3.14159
   >    Back:  3.14159

The copy constructor should create a new version of the list
   > Copy the double List using the copy-constructor
   >    Size:  1
   >    Empty? No
   >    Front: 3.14159
   >    Back:  3.14159

Create a list with one node. That list will get destroyed
with the following copy constructor
   > Copy a double List using the assignment operator
   >    Size:  1
   >    Empty? No
   >    Front: 3.14159
   >    Back:  3.14159
   > Test 1 complete

Test 1 passed.
------------------------------------------------------------------

------------------------------------------------------------------
Starting Test 2

   > Select the test you want to run:
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   >    a. Fibonacci
   > > 2

Test push_back() by adding items to the back of the list

Create an integer List with the default constructor
   > Enter integer values to put on the back, type 0 when done
   >    (   ...   ) > 2
   >    ( 2 ... 2 ) > 4
   >    ( 2 ... 4 ) > 6
   >    ( 2 ... 6 ) > 8
   >    ( 2 ... 8 ) > 0
Copy the list from l1 to l2 using the copy constructor.
```

```
   > Copied l1 into l2
Next change the end of l1 to -42 in the front and 42 in the back.
   > Modified l1
Finally, clear all the elements out of l2
   > Copied list l2 is empty.

Test push_front() by adding items to the front of the list\n

   > Enter integer values to put on the front, type 0 when done
   >    (   ...   ) > 1
   >    ( 1 ... 1 ) > 3
   >    ( 3 ... 1 ) > 5
   >    ( 5 ... 1 ) > 7
   >    ( 7 ... 1 ) > 0
Test the size() method
   > Sizes of l1 and l2 are correct
Make sure that l1 was unchanged from earlier
   > The list l1 was unchanged
   > Test 2 complete

Test 2 passed.
---------------------------------------------------------------

---------------------------------------------------------------
Starting Test 3

   > Select the test you want to run:
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   >    a. Fibonacci
   > > 3
   > Create a string List with the default constructor
   > Instructions:
   >    + dog   pushes dog onto the front
   >    - cat   pushes cat onto the back
   >    #       displays the contents of the list backwards
   >    *       clear the list
   >    !       quit

Test push_front() three times.
Everything should appear in the opposite order
Note that we will create an iterator on an empty list here
   > {  } > +three
   > {  three  } > +four
   > {  four  three  } > +five

Test push_back() three times.
Things are added in the correct order
   > {  five  four  three  } > -two
   > {  five  four  three  two  } > -one
   > {  five  four  three  two  one  } > -zero

Next we will display the list backwards.
This will exercise rbegin(), rend(), and the -- operator
   > {  five  four  three  two  one  zero  } > #
   >    Backwards: {  zero  one  two  three  four  five  }

Next test clear() which should remove everthing from the list
   > {  five  four  three  two  one  zero  } > *

Test rbegin() and rend() on an empty list
   > {  } > #
   >    Backwards: {  }

Finally we should be able to start the list over after clear()
the same as if we were starting from a fresh list
   > {  } > +front
   > {  front  } > -back
   > {  front  back  } > #
   >    Backwards: {  back  front  }
   > {  front  back  } > !
   > Test 3 complete

Test 3 passed.
---------------------------------------------------------------

---------------------------------------------------------------
```

```
Starting Test 4

   > Select the test you want to run:
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
   >    3. The above plus iterate through the List
   >    4. The above plus insert and remove items from the list
   >    a. Fibonacci
   > > 4
   > Instructions:
   >    + 3 A  put 'A' after the 3rd item in the list
   >    - 4    remove the fourth item from the list
   >    !       quit

Remove 'b', the element in slot 1
   >    0  1  2  3  4  5  6  7  8  9 10 11 12
   > {  a  b  c  d  e  f  g  h  i  j  k  l  m  }
   > > -1

Remove 'd', the element in slot 2
   >    0  1  2  3  4  5  6  7  8  9 10 11
   > {  a  c  d  e  f  g  h  i  j  k  l  m  }
   > > -2

Remove 'c', the element in slot 1
   >    0  1  2  3  4  5  6  7  8  9 10
   > {  a  c  e  f  g  h  i  j  k  l  m  }
   > > -1

Now we will put 'B' in the slot between 0 and 1
   >    0  1  2  3  4  5  6  7  8  9
   > {  a  e  f  g  h  i  j  k  l  m  }
   > > +1B

Next we will put 'D' in the slot between 1 and 2
   >    0  1  2  3  4  5  6  7  8  9 10
   > {  a  B  e  f  g  h  i  j  k  l  m  }
   > > +2D

Now we will pu tC in the slot between 1 and 2
   >    0  1  2  3  4  5  6  7  8  9 10 11
   > {  a  B  D  e  f  g  h  i  j  k  l  m  }
   > > +2C

Test removing off the end of the list. This is a special case
   >    0  1  2  3  4  5  6  7  8  9 10 11 12
   > {  a  B  C  D  e  f  g  h  i  j  k  l  m  }
   > > -12

Test removing off the beginning of the list, another special case
   >    0  1  2  3  4  5  6  7  8  9 10 11
   > {  a  B  C  D  e  f  g  h  i  j  k  l  }
   > > -0

Test adding onto the beginning of the list, another special case
   >    0  1  2  3  4  5  6  7  8  9 10
   > {  B  C  D  e  f  g  h  i  j  k  l  }
   > > +0A

Finally, test adding onto the end of the list, another special case
   >    0  1  2  3  4  5  6  7  8  9 10 11
   > {  A  B  C  D  e  f  g  h  i  j  k  l  }
   > > +12M

All done! The CAPITAL letters are the ones added
   >    0  1  2  3  4  5  6  7  8  9 10 11 12
   > {  A  B  C  D  e  f  g  h  i  j  k  l  M  }
   > > !
   > Test 4 complete

Test 4 passed.
---------------------------------------------------------------

---------------------------------------------------------------
Starting Test 5

   > Select the test you want to run:
   >    1. Just create and destroy a List
   >    2. The above plus push items onto the List
```

```
>    3. The above plus iterate through the List
>    4. The above plus insert and remove items from the list
>    a. Fibonacci
> > a
> How many Fibonacci numbers would you like to see? 10
>    1
>    1
>    2
>    3
>    5
>    8
>    13
>    21
>    34
>    55
> Which Fibonacci number would you like to display? 500
> \t315178285\n
Exp: \t139,423,224,561,697,880,139,724,382,870,407,283,950,070,256,587,697,307,264,108,962,948,325,571,622,863,290,
691,557,658,876,222,521,294,125\n

Test 5 failed.
------------------------------------------------------------


============================================================
Failed 1/5 tests.
============================================================
```

Grading Criteria

Criteria	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%	Weight	Score
List interface	The interfaces are perfectly specified with respect to const, pass-by-reference, etc.	week07.cpp compiles without modification	All of the methods in List match the problem definition	List has many of the same interfaces as the problem definition	The public methods in the List class do not resemble the problem definition	20	
List Implementation	Passes all four List testBed tests	Passes three testBed tests	Passes two testBed tests	Passes one testBed test	Program fails to compile or does not pass any testBed tests	20	
Whole Numbers	The WholeNumber class supports all the common operators perfectly	A WholeNumber class exists but does not implement any of the common operators	Able to perfectly handle large numbers without a WholeNumber class -or- a WholeNumber class exists but has one minor bug	An attempt was made to use the List class to represent large numbers	No attempt was made to handle large whole numbers	30	
Fibonacci	The most efficient solution was found	Passes the Fibonacci testBed test	The code essentially works but with minor defects	Elements of the solution are present	The Fibonacci problem was not attempted	10	
Code Quality	There is no obvious room for improvement	All the principles of encapsulation and modularization are honored	One function is written in a "backwards" way or could be improved	Two or more functions appears "thrown together."	The code appears to be written without any obvious forethought	10	
Style	Great variable names, no errors, great comments	No obvious style errors	A few minor style errors: non-standard spacing, poor variable names, missing comments, etc.	Overly generic variable names, misleading comments, or other gross style errors	No knowledge of the BYU-I code style guidelines were demonstrated	10	
Extra Credit	10% Implement a ListConstIterator	5% Extend the WholeNumber class to include subtraction	10% Extend the WholeNumber class to include the extraction operator	10% Extend the WholeNumber class to include multiplication		40	
Total	Fibonacci application doesn't use big number BigNumber class partly works but not called			-15			85/100

vas14001@byui.edu