Printout for cs320-09-A10-ListL.hpp

Name:
Date:
Assignment:
-1

```
// File: ListL/ListL.hpp

#ifndef LISTL_HPP_
#define LISTL_HPP_

#include <iostream> // ostream.
using namespace std;

template<class T> class LNode; // Forward declaration.
template<class T> class ListLIterator; // Forward declaration.

template<class T>
class ListL {
    friend class ListLIterator<T>;
private:
    LNode<T> *_head;

    ListL(ListL<T> const &rhs);
    // Copy constructor disabled.

public:
    ListL();  // Constructor
    // Post: This list is initialized to be empty.

    ~ListL();  // Destructor
    // Post: This list is deallocated.

    void append(T const &data);
    // Post: data is appended to this list.

    void clear();
    // Post: This list is cleared to the empty list.

    void concat(ListL<T> &suffix);
    // Post: suffix is appended to this list.
    // suffix is empty (cut concatenate, as opposed to copy concatenate).

    bool contains(T const &data) const;
    // Post: true is returned if data is contained in this list;
    // Otherwise, false is returned.

private:
    LNode<T> *copyHead(ListL<T> const &rhs);
    // Post: A deep copy of the head of rhs is returned.

public:
    bool equals(ListL<T> const &rhs) const;
    // Post: true is returned if this list equals list rhs;
    // Otherwise, false is returned.
    // Two lists are equal if they contain the same number
    // of equal elements in the same order.

    T const &first() const;
    // Pre: This list is not empty.
    // Post: The first element of this list is returned.

    bool isEmpty() const;
    // Post: true is returned if this list is empty;
    // Otherwise, false is returned.

    int length() const;
    int length2() const;
    // Post: The length of this list is returned.
```

```
    T const &max() const;
    T const &max3() const;
    // Pre: This list is not empty.
    // Post: The maximum element of this list is returned.

    ListL<T> &operator=(ListL<T> const &rhs);
    // Post: This list is a deep copy of rhs.

    void prepend(T const &data);
    // Post: data is prepended to this list.

    T remFirst();
    // Pre: This list is not empty.
    // Post: The first element is removed from this list and returned.

    T remLast();
    // Pre: This list is not empty.
    // Post: The last element is removed from this list and returned.

    void remove(T const &data);
    // Post: If data is in this list, it is removed;
    // Otherwise this list is unchanged.

    void reverse();
    // Post: This list is reversed.

    void setFirst(T const &data);
    // Pre: This list is not empty.
    // Post: The first element of this list is changed to data.

    void toStream(ostream &os) const;
    void toStream4(ostream &os) const;
    // Post: A string representation of this list is returned.

    ListL<T> *unZip();
    // Post: This list is every other element of this list
    // starting with the first.
    // A pointer to a list with every other element of this list
    // starting with the second is returned.

    void zip(ListL<T> &other);
    // Post: This list is the same perfect shuffle of this list and other,
    // starting with the first element of this.
    // other is the empty list (cut zip, as opposed to copy zip).
};

template<class T>
class LNode {
    friend class ListL<T>;
    friend class ListLIterator<T>;
private:
    T _data;
    LNode *_next;

private:
    LNode(T data);
};

template<class T>
class ListLIterator {
private:
    ListL<T> const *_listL;
    LNode<T> *_current;
public:
    void setIterListL(ListL<T> const *listL) { _listL = listL; }
```

```
    // Post: Positions the iterator to the first element.
    void first() { _current = _listL->_head; }

    // Post: Advances the current element.
    void next() { _current = _current->_next; }

    // Post: Checks whether there is a next element.
    bool hasNext() const { return _current->_next != nullptr; }

    // Post: Checks whether at end of the list.
    bool isDone() const { return _current == nullptr; }

    // Pre: The current element exists.
    // Post: The current element of this list is returned.
    T const &currentItem() const {
        if (_current == nullptr) {
            cerr << "currentItem precondition violated: "
                 << "Current element does not exist." << endl;
            throw -1;
        }
        return _current->_data;
    }
};

// ========= Constructors =========
template<class T>
ListL<T>::ListL():
    _head(nullptr) {
}

template<class T>
LNode<T>::LNode(T data):
    _data(data),
    _next(nullptr) {
    }

// ========= Destructor =========
template<class T>
ListL<T>::~ListL() {
    clear();
}

// ========= append =========
template<class T>
void ListL<T>::append(T const &data) {
    if (_head == nullptr) {
        prepend(data);
    } else {
        LNode<T> *p = _head;
        while (p -> _next != nullptr) {
            p = p -> _next;
        }
        LNode<T> *temp = new LNode<T>(data);
        p -> _next = temp;
    }
}

// ========= clear =========
template<class T>
void ListL<T>::clear() {
    LNode<T> *p;
    while (_head != nullptr) {
        p = _head;
        _head = _head -> _next;
        p -> _next = nullptr;
        delete p;
```

```
    }
}

// ========= concat =========
template<class T>
void ListL<T>::concat(ListL<T> &suffix) {
    cerr << "ListL<T>::concat: Exercise for the student." << endl;
    throw -1;
}

// ========= contains =========
template<class T>
bool ListL<T>::contains(T const &data) const {
    LNode<T> *p, *q;
    q = new LNode<T>(data);
    p = _head;
    while (p != nullptr) {
        if (p -> _data == q -> _data) {
            return true;
        } else {
            p = p -> _next;
        }
    }
    return false;
}

// ========= copyHead =========
template<class T>
LNode<T> *ListL<T>::copyHead(ListL<T> const &rhs) {
    if (rhs.isEmpty()) {
        return nullptr;
    } else {
        LNode<T> *p, *q, *result;
        p = rhs._head;
        result = new LNode<T>(p -> _data);
        q = result;
        while(p -> _next != nullptr){
            q -> _next = new LNode<T>(p -> _next -> _data);
            q = q -> _next;
            p = p -> _next;
        }
        return result;
    }
}
// ========= equals =========
template<class T>
bool ListL<T>::equals(ListL<T> const &rhs) const {
    LNode<T> *p  = _head;
    LNode<T> *q = rhs._head;
    while (p != nullptr && q != nullptr  && p -> _data == q -> _data){
        p = p -> _next;
        q = q -> _next;
    }
return (p == nullptr && q == nullptr);
}

// ========= first =========
template<class T>
T const &ListL<T>::first() const {
    if (_head == nullptr){
    cerr << "first precondition violated: "
         << "List cannot have a first element if it is empty." << endl;
    throw -1;
    }
    return _head -> _data;
}
```

```
// ========= isEmpty =========
template<class T>
bool ListL<T>::isEmpty() const {
    return _head == nullptr;
}

// ========= length =========
template<class T>
int ListL<T>::length() const {
    LNode<T> *p;
    p = _head;
    T count = 0;
    while (p != nullptr) {
        count++;
        p = p -> _next;
    }
    return count;
}

// ========= length2 =========
template<class T>
int ListL<T>::length2() const {
    ListLIterator<T> iter;
    iter.setIterListL(this);
    T count = 0;
    for (iter.first(); !iter.isDone(); iter.next()) {
        count++;
    }
    return count;
}

// ========= max =========
template<class T>
T const &ListL<T>::max() const {
    if (_head == nullptr) {
        cerr << "max precondition violated: An empty list has no maximum." << endl;
        throw -1;
    }
    T const *result = &_head -> _data;
    cerr << "ListL<T>::max: Exercise for the student." << endl;
    throw -1;
}

// ========= max3 =========
template<class T>
T const &ListL<T>::max3() const {
    cerr << "ListL<T>::max3: Exercise for the student." << endl;
    throw -1;
}

// ========= operator= =========
template<class T>
ListL<T> &ListL<T>::operator=(ListL<T> const &rhs) {
    if (this != &rhs) { // In case someone writes myList = myList;
        clear();
        _head = copyHead(rhs);
    }
    return *this;
}

// ========= operator== =========
template<class T>
bool operator==(ListL<T> const &lhs, ListL<T> const &rhs) {
    return lhs.equals(rhs);
}
```

```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, ListL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= prepend =========
template<class T>
void ListL<T>::prepend(T const &data) {
    LNode<T> *temp = _head;
    _head = new LNode<T>(data);
    _head -> _next = temp;
}

// ========= remFirst =========
template<class T>
T ListL<T>::remFirst() {
    if (_head == nullptr) {
        cerr << "remFirst precondition violated: Cannot remove an element from an empty
list." << endl;
        throw -1;
    }
    T result;
    LNode<T> *p = _head;
    result = p -> _data;
    _head = _head -> _next;
    delete p;
    return result;
    }

// ========= remLast =========
template<class T>
T ListL<T>::remLast() {
     if (_head == nullptr) {
        cerr << "remLast precondition violated: Cannot remove an element from an empty
list." << endl;
        throw -1;
    }
     T val;
     LNode<T> *p = _head;
     LNode<T> *q = nullptr;
     while(p -> _next != nullptr){
         q = p;
         p = p -> _next;
     }
     val = p -> _data;
     delete p;
     if(q == nullptr){
         _head = nullptr;
     } else{
         q -> _next = nullptr;
     }
     return val;
}

// ========= remove =========
template<class T>
void ListL<T>::remove(T const &data) {
    if (_head != nullptr) {
        LNode<T> *p = _head;
        LNode<T> *q;
        if (p -> _data == data) {
            _head = p -> _next;
            q = nullptr;
```

```
            delete p;
            delete q;
        } else {
            q = p;
            p = p -> _next;
            while (p != nullptr) {
                if (p -> _data == data) {
                    q -> _next = p -> _next;
                    p -> _next = nullptr;
                    delete p;
                    p = nullptr;
                    q = nullptr;
                    delete q;
                } else {
                    q = p;
                    p = p -> _next;
                }
            }
        }
    }
}

// ========= reverse =========
template<class T>
void ListL<T>::reverse() {
    LNode<T> *pReverse, *pRest, *temp;
    pReverse = nullptr;
    pRest = _head;
    if (_head != nullptr) {
        pReverse = _head;
        pRest = pRest -> _next;
        pReverse -> _next = NULL;
        while (pRest != NULL) {
            temp = pReverse;
            pReverse = pRest;
            pRest = pRest -> _next;
            pReverse -> _next = temp;
        }
        _head = pReverse;
        }
    }

// ========= setFirst =========
template<class T>
void ListL<T>::setFirst(T const &data) {
    if (_head == nullptr) {
        cerr << "setFirst precondition violated: Cannot set first on an empty list." <<
endl;
        throw -1;
    }
    _head -> _data = data;
}

// ========= toStream =========
template<class T>
void ListL<T>::toStream(ostream &os) const {
    os << "(";
    for (LNode<T> *p = _head; p!= nullptr; p = p -> _next) {
        if (p->_next != nullptr) {
            os << p -> _data << ", ";
        }
        else {
            os << p -> _data;
        }
    }
    os << ")";
```