Project-4: Derived Sorted Lists
F15 CSCI 232 - Data Structures and Algorithms

Phillip J. Curtiss, Assistant Professor
Computer Science Department, Montana Tech
Museum Building, Room 105

November 16, 2015

Project-4: Due 2015-11-30 by midnight

Purpose: The sorted list ADT can be viewed as a specialized form of the ListADT. Rewrite the ListADT to form an AS-IS relationship with a new class you are to write, SortedListADT. Implement an Interface for the SortedListADT based on the UML specification below. You are provided the full implementation of ListADT, including the overloading of the `operator==` and the `operator+` - you may need to override other operators, or even these operators, within the SortedListADT.

Objectives:

- Using UML Diagrams to Specify ADT
- Using existing C++ class methods
- Using Class Inheritence
- Understanidng Class Inheritence Relationships
- Create methods that implement the SortList ADT
- Manipulate Lists of Complex Data Types
- Manipulate Linked List Structures
- Work with Pointers in C++
- Overriding operators within Classes
- Pass objects by reference
- Use C++ source file standards and doxygen to generate documentation
- Modify and add features to the provided main program

Obtaining the Project Files: You will complete the project using your own user account on the department's linux system `katie.mtech.edu`. You should ssh into your account, and execute the command `mkdir -p csci232/proj4`. This will make the directory `proj4` inside the directory `csci232`, and also create the directory `csci232` if it does not yet exist - which it should from our previous lab and project sessions. You should then change the current working directory to proj4 by executing the command `cd csci232/proj4`. You can test that you are in the correct current working directory by executing the command `pwd` which should print something like `/home/students/<username>/csci232/proj4`, where `<username>` is replaced with the username associated with your account. Lastly, execute the command `tar -xzvf ~curtiss/csci232/proj4.tgz` and this will expand the project files from the archive into your current working directory. You may consult `man tar` to learn about the archive utility and its command line options.

Table 1: UML Specification for the sortedList ADT

```
+isEmpty(): boolean
+getLength(): integer
+insertSorted(newEntry: ItemType): void
+removeSorted(entry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+getPosition(entry: ItemType): integer
+contains(entry: ItemType): boolean
+loadList(string: filename): boolean
+displayList(): void
```

Project Function: Your classes will be used with a driver program we will provide for testing. This driver program will instantiate multiple instances of your SortedListADT class. It will then populate these instances with items of different fundamental types - such as ints, strings, etc. The program will then use the `operator<<` to display the output of the SortedListADT objects and will use the `operator==` to compare multiple instances of the SortedListADT to determine if the objects are identical. Lastly, it will use the `operator+` to create and/or expand objects of SortedListADT.

The driver program will use different input files of varying lengths to accomplish these tests. Your classes should make no assumptions about the fundamental ItemTypes used by the driver program, the number of instances of your SortedListADT created by the driver program, now the length of any given instance of your SortedListADT.

Building the Project: The project includes a `Makefile` you may use to generate the object files from source code files. If you wisht to provide a test driver for your project, make sure to edit the Makefile as needed. Consult the `Makefile` and understand the rules included and the dependencies and the rule sets that are used to generate the executable program. Use caution when updating the `Makefile` to ensure rule sets make sense.

Helpful Reminders: Study and pay close attention to the provided class(es) and methods. Understand their return types and use them in the code you author to provide robust code that can handle exceptions to inputs and boundary conditions. Look at all the code provided. Read the codes's comments and implement what is required where indicated. Make sure you are reusing code inside methods from inherited classes. Be cognizant of the *best practices we discussed in lecture and abide by good coding style - all of which will be factored into the assessment and grade for this project.* Be sure to review the UML diagram and the `Makefile` and understand how files are being generated and their dependencies.

Submission of Project: You have been provided a `Makefile` for this project that will help you not only build your project, but also submit the project in the correct format for assessment and grading. Toward the bottom of the provided `Makefile` you should see lines that look like:

```
# Rule to submit programming assignments to graders
# Make sure you modify the $(subj) $(msg) above and the list of attachment
# files in the following rule - each file needs to be preceeded with an
# -a flag as shown
subj    = "CSCI232 DSA - Proj4"
msg     = "Please review and grade my Project-3 Submission"
submit: listing-A1.cpp listing-A2.cpp
        $(tar) $(USER)-proj4.tgz $?
        echo $(msg) | $(mail) -s $(subj) -a $(USER)-proj4.tar.gz $(addr)
```

Make sure you update the dependencies on the `submit:` line to ensure all the required files (source files) are included in the archive that gets created and then attached to your email for submission. You do not need to print out any of your program files - submitting them via email will date and time stamp them and we shall know they come from your account. If you submit multiple versions, we will use the latest version up to when the project is due.

Questions: If you have any questions, please do not hesitate to get in contact with either Phil Curtiss (`pjcurtiss@mtech.edu`) or Ross Moon (`rmoon@mtech.edu`) at your convenicne, or stop by during office hours, and/or avail yourself of the time in the MUS lab when Ross is available.

Project File Manifest:

Sorted List Interface

```
/** Interface for the Sorted ADT list
 * @file SortedListInterface.h
 */
#ifndef _SORTEDLIST_INTERFACE
#define _SORTEDLIST_INTERFACE

template < class ItemType > class SortedListInterface
{
public:

        // Place your methods here that specify the SortedListInterface

};  // end SortedListInterface
#endif
```

Sorted List Header

```
/*
 * @file SortedListADT.h
 *
 */
#ifndef _SORTEDLISTADT
#define _SORTEDLISTADT

// SortedListInterface description
#include "SortedListInterface.h"

// Standard exception handling classes
#include <stdexcept>


/** @class SortedListADT SortedListADT.h "SortedListADT.h"
 * SortedListADT defines a class representing a sorted list of items
 * where the items are in an order represented by a sort index
 */
template<class ItemType>
class SortedListADT: public SortedListInterface<ItemType>
                     private ListADT<ItemType>
{
```

```
private:

                // Place your private data members and methods here

public:

                // Place your public methods here
};

#include "SortedListADT.cpp"
#endif
```

Sorted List Implementation

```
/*
 * @file SortedListADT.cpp
 *
 */
#ifndef _SORTEDLISTADTIMP
#define _SORTEDLISTADTIMP

// ListADT class description
#include "SortedListADT.h"

// ListADT class description
// Implemented as-a class relationship
#include "ListADT.h"

// Standard exception handling classes
#include <stdexcept>

template<class ItemType>
SortedListADT<ItemType>::SortedListADT()
{
        // Implement the default constructor
}// end default constructor

template<class ItemType>
SortedListADT<ItemType>::SortedListADT(const SortedListADT<ItemType>& aList)
{
        // Implement the copy constructor - if needed
} // end copy constructor - deep copy of SortedListADT param

template<class ItemType>
SortedListADT<ItemType>::~SortedListADT()
{
        // Implement the default destructor if needed

} // end default destructor

// Implement the remaining methods required by the SortedListADT

#endif
```

List Interface

```
/** Interface for the ADT list
 * @file ListInterface.h
 */
#ifndef _LIST_INTERFACE
#define _LIST_INTERFACE

using namespace std;
#include <string>
#include "ListNode.h"

template < class ItemType > class ListInterface
{
public:
/** Sees whether this list is empty.
@return True if the list is empty; otherwise returns false. */
  virtual bool isEmpty() const = 0;

/** Gets the current number of entries in this list.
@return The integer number of entries currently in the list. */
  virtual int getLength() const = 0;

/** Inserts an entry into this list at a given position.
@pre None.
@post If 1 <= position <= getLength() + 1 and the insertion is
successful, newEntry is at the given position in the list,
other entries are renumbered accordingly, and the returned
value is true.
@param newPosition The list position at which to insert newEntry.
@param newEntry The entry to insert into the list.
@return True if insertion is successful, or false if not. */
  virtual bool insert(int newPosition, const ItemType & newEntry) = 0;

/** Removes the entry at a given position from this list.
@pre None.
@post If 1 <= position <= getLength() and the removal is successful,
the entry at the given position in the list is removed, other
items are renumbered accordingly, and the returned value is true.
@param position The list position of the entry to remove.
@return True if removal is successful, or false if not. */
  virtual bool remove(int position) = 0;

/** Removes all entries from this list.
@post List contains no entries and the count of items is 0. */
  virtual void clear() = 0;

/** Gets the entry at the given position in this list.
@pre 1 <= position <= getLength().
@post The desired entry has been returned.
@param position The list position of the desired entry.
@return The entry at the given position. */
  virtual ItemType getEntry(int position) const = 0;
/** Replaces the entry at the given position in this list.
@pre 1 <= position <= getLength().
@post The entry at the given position is newEntry.
@param position The list position of the entry to replace.
@param newEntry The replacement entry. */
  virtual void setEntry(int position, const ItemType & newEntry) = 0;

/** Returns the position within the List of the entry provided or -1 if
    not found in the List
@pre 1 <= position <= getLength()
@post none
@param entry to search for in the List
@return the position within the List if entry is found, or -1 otherwise */
  virtual int getPosition(const ItemType & entry) = 0;

/** Returns a boolean value indicating whether the entry provided is in the List
@pre none
@post none
@param entry to search for in the List
@ return boolean value indicating whether the entry is in the List */
  virtual bool contains(const ItemType & entry) = 0;

};  // end ListInterface
#endif
```

List Main Driver

```
/*
 * @file ClientDriver.cpp
 *
 * Phillip J. Curtiss, Assistant Professor
 * Computer Science Department, Montana Tech
 * F15 CSCI232 -
 */

// Our SortedListInterface
#include "SortedListInterface.h"

// Requried to be able to write to output
#include <iostream>

// Search the std namespace when reference is not found
using namespace std;

/*
 * Main driver program - entry pointer
 */
int main()
{
        // Create two integer based SortedListADT lists
        SortedListInterface<int> l1 = SortedListADT<int>();
        SortedListInterface<int> l2 = SortedListADT<int>();
        // Create two string based SortedListADT Lists
        SortedListInterface<string> S1 = SortedListADT<string >();
        SortedListInterface<string> S2 = SortedListADT<string >();

// Rest of the ClinetDriver will be provided later for testing

        // Output end of report identifier
        cout << endl << " -- End of Output -- " << endl;

        // Terminate normally
        return(0);
}
```

ListNode Header

```
/*
 * @file ListNode.h
 *
 * Phillip J. Curtiss, Assistant Professor
 * Computer Science Department, Montana Tech
 * F15 CSCI232 - Lab 11/04 - Inheritance and Operator Overloading
 */
#ifndef _LISTNODE
#define _LISTNODE

/** @class ListNode ListNode.h "ListNode.h"
 * ListNode is a private class used to support other classes
 * such as ListADT. This establishes an as-a relationship
 * between the ListNode class and the ListADT class.
 */
template < class ItemType>
class ListNode
{
        // This entire class is private, therefore we must
        // explicity grant access to private members using
        // the friend construct. Since ListADT is a template
        // class, we provide a generic template here, different
        // from the template ListNode is using.
        template <class U>
        friend class ListADT;

        // Private constructors means that only classes specifically
        // granted access by the ListNode class can use instances of
        // ListNode
    private :
            ItemType item;                              // A data item
            ListNode<ItemType>* next;    // Pointer to next node

                /** Default constructor: Creates an instance of a ListNode
                 *  and inializes data members
                 * @pre None.
                 * @post A ListNode instance */
            ListNode();
```

```
                /** Constructor: Creates a ListNode with its item initialized
                 *  to the parameter provided
                 * @pre anItem is an ItemType previously created
                 * @post A ListNode instance with its item data member initialized
                 * @param anItem which represents an ItemType data type to initialize item
            ListNode( const ItemType& anItem );

                /** Constructor: Creates a ListNode with its item and next initialized
                 * @pre anItem is an ItemType previously created
                 * @pre nextNodePtr is a pointer to ListNode<ItemType> previously created
                 * @post A ListNode instance with its item and next data members initialize
                 * @param anItem which represents an ItemType data type to initialize item
                 * @param nextNodePtr which represents a pointer of ListNode<ItemType> to
            ListNode( const ItemType& anItem, ListNode<ItemType>* nextNodePtr);

}; // end ListNode
#include "ListNode.cpp"
#endif
```

ListNode Implementation

```
/*
 * @file ListNode.cpp
 *
 * Phillip J. Curtiss, Assistant Professor
 * Computer Science Department, Montana Tech
 * F15 CSCI232 - Lab 11/04 - Inheritance and Operator Overloading
 */
#ifndef _LISTNODEIMP
#define _LISTNODEIMP

// ListNode class description
#include "ListNode.h"

template<class ItemType>
ListNode<ItemType>::ListNode(): next(nullptr) {}
// end default constructor - sets next to nullptr

template<class ItemType>
ListNode<ItemType>::ListNode(const ItemType& anItem): item(anItem), next(nullptr) {}
// end constructor - initializes item from parameter and sets next to nullptr

template<class ItemType>
ListNode<ItemType>::ListNode(const ItemType& anItem, ListNode<ItemType>* nextNodePtr): item
// end constructor - initializes item from parameter and next from parameter

#endif
```

ListADT Header

```
/*
 * @file ListADT.h
 *
 * Phillip J. Curtiss, Assistant Professor
 * Computer Science Department, Montana Tech
 * F15 CSCI232 - Lab 11/04 - Inheritance and Operator Overloading
 */
#ifndef _LISTADT
#define _LISTADT

// ListInterface description
#include "ListInterface.h"

// As-is relationship to ListNode
#include "ListNode.h"

// Standard exception handling classes
#include <stdexcept>

// Required for overloading operator<< from ostream class
#include <ostream>

/** @class ListADT ListADT.h "ListADT.h"
 * ListADT defines a class representing a list of items
 * where the items are in a user defined position within
 * the list
 */
template<class ItemType>
class ListADT: public ListInterface<ItemType>
{
// Friend Declarations almost always come first
// before standard class access decorators (public, protected, private)
// This friend declarattion grants access to the ostream class for any
// private or protected members from the ListADT class. This is needed as the
// operator<< must have access to internal data members in order to produce the
// correct output - note that since ListADT and ostream are template classes we
// must chose a generic template that is not the same as that of ListADT
        template<class friendItemType>
        friend std::ostream& operator<<(std::ostream& outputStream, const ListADT<friendIte

private:
        int itemCount; // Used to hold the number of items in the list
        ListNode<ItemType> *headPtr; // Used as the head pointer of the linked list of iten

        /** Return a pointer to a node in the linked list from a given position provided
         *  as a parameter
         * @pre none
         * @post none
         * @param position and integer representing a position in the linked list
         * @return a pointer to a node in the linked list corresponding to the position pa
        ListNode<ItemType>* getNodeAt(int position) const;

        /** Return an item from a given node in the linked list
         * @pre a pointer to a node in the list must be valid and in the linked list
         * @post none
         * @param node is a pointer to a node in the linked list
         * @return an item of ItemType corresponding to the node pointer
         * @comment Accessor method for item in ListNode class */
        ItemType getItem(ListNode<ItemType> *node) const throw(std::invalid_argument);

        /** Return a node pointer to the next data member of a node provided as a
         *  parameter
         * @pre a pointer to a node in the list must be valid and in the linked list
         * @post none
         * @param a pointer to a node in the linked list
         * @return the next data member of the node pointer parameter
         * @comment Accessor method for next in ListNode class */
        ListNode<ItemType>* getNext(ListNode<ItemType> *node) const throw(std::invalid_argu

public:
        /** Default constructor: Creates an instance of a ListADT and initializes its
         *  itemCount and headPtr data members to default values
         * @pre none
         * @post a new instance of a ListADT with default data member values */
        ListADT();

        /** Constrcutor: Creates an instance of a ListADT with initial values based on
         *  a ListADT passed into the constructor as a parameter. The Constrctor makes
         *  a deep copy of the parameter ListADT. Data members are set to correspond with
         *  the ListADT parameter
         * @pre a ListADT object that is passed as a parameter to the constrctor
         * @post a new instance of a ListADT with a deep copy of the ListADT parameter
         * @param aList is a ListADT object instance */
        ListADT(const ListADT<ItemType>& aList);

        /** Default destructor: Destroys the ListADT object taking care to free memory
         *  allocated for the linked list structure.
         * @pre none
         * @post the freeing of allocated memory for the linkec list structure
         * @commend this is a virtual method and should be overridden if this class is
         * inherited by another class */
        virtual ~ListADT();

        /** Tests whether the ListADT is empty - i.e. has no nodes in its linked list
         * @pre none
         * @post none
         * @return boolean value indicating whether the linked list has no nodes or
         * at least one node - we test itemCount */
        bool isEmpty() const;

        /** Returns the length of the ListADT - i.e. the number of nodes in the linked
         *  list structure
         * @pre none
         * @post none
         * @return an integer representing the number of nodes in the linked list structur
        int getLength() const;

        /** Inserts a newEntry of ItemType into the list at a given position within the li
         * @pre the position must be valid in the list - i.e. 0 < position <= getLength()
         * @post new item in the linked list structure at the specified position
         * @param a position at which to insert the new item
```

```
     * @param newEntry the node to insert itno the linked list at the specified positio
     * @return boolean value as to the success of the entry of the newEnter into the li
     * list structure */
    bool insert(int newPosition, const ItemType& newEntry);

    /** Removes a node from the linked list structure at a given position
     * @pre the position must be valid in the list - i.e. 0 < position <= getLength()
     * @post the node at the specified position is removed from the list
     * @param position is an integer representing a valid position in the linked list
     * @return boolean value as to the success of the removal of the node at the
     * specified position */
    bool remove(int position);

    /** Removes all nodes from the linked list and sets the itemCount and headPtr acco
     * @pre none
     * @post the linked list of nodes of items in the list is freed and the nodes dele
    void clear();

    /** Returns the item portion of a node found at the specified position
     * @pre the position must be valid in the list - i.e. 0 < position <= getLength()
     * @post none
     * @return the item portion of the node specified by the provided position */
    ItemType getEntry(int position) const throw(std::out_of_range);

    /** Replaces the node in the linked list structure corresponding to the specified
     * @pre the position must be valid in the list - i.e. 0 < position <= getLength()
     * @post the node at the specified position is replaced with the newEntry paramete
     * @param position is an int representing a valid location within the linked list
     * @param newEntry is a node to place in the linked list at the specified position
    void setEntry(int position, const ItemType& newEntry);

    /** Return the position within the linked list correspnding to the node parameter
     * @pre the entry must be a node compared to the nodes in the linked list
     * @post none
     * @param entry is a node used to search through the linked list
     * @return the position within the linked list structure at which the node is foun
    int getPosition(const ItemType& entry);

    /** Returns a boolean value as to whether a given node is contained in the linked
     * @pre the entry must be a node to compare to the nodes in the linked list
     * @post none
     * @param entry is a node used to search through the linked list
     * @return a boolean value as to the success of finding the entry in the linked li
    bool contains(const ItemType& entry);

    /** Returns a boolean value as to the sameness comparison of two ListADT object in
     * @pre two ListADT object instances must exist (lhs == rhs)
     * @post none
     * @param the right hand side (rhs) of the == operator - used to compare to the in
     * @return boolean value representing the sameness of the two instance objects com
    bool operator==(const ListADT<ItemType>& rhs) const;

    /** Returns a concatenated instance of the ListADT object from two other instance
     * @pre two ListADT object instances must exist (lhs + rhs)
```

```
          * @post a new ListADT representing the concatenated ListADT objects as operands
          * @return the new ListADT formed from the concatenation of the two ListADT object
        ListADT<ItemType> operator+(const ListADT<ItemType>& rhs);
};

#include "ListADT.cpp"
#endif
```

ListADT Implementation

```
/*
 * @file ListADT.cpp
 *
 * Phillip J. Curtiss, Assistant Professor
 * Computer Science Department, Montana Tech
 * F15 CSCI232 - Lab 11/04 - Inheritance and Operator Overloading
 */
#ifndef _LISTADTIMP
#define _LISTADTIMP

// ListADT class description
#include "ListADT.h"

// ListNode class description
// Implemented as-a class relationship
#include "ListNode.h"

// Standard exception handling classes
#include <stdexcept>

// Used to override the operator<<
#include <ostream>

template<class ItemType>
ListADT<ItemType>::ListADT(): itemCount(0), headPtr(nullptr) {}
// end default constructor - sets itemCount and headPtr to default values

template<class ItemType>
ListADT<ItemType>::ListADT(const ListADT<ItemType>& aList)
{
        // Initialize our data members
        itemCount = 0;
        headPtr = nullptr;

        // Iterate through the nodes in the parameter linked list and create new
        // nodes initialized to the item of the parameter linked list node and
        // add to our linked list
        for (ListNode<ItemType> *aListPtr = aList.headPtr; aListPtr != nullptr; aListPtr =
                (void) ListADT<ItemType>::insert(itemCount + 1, aListPtr->item);

} // end copy constructor - deep copy of ListADT param

template<class ItemType>
ListADT<ItemType>::~ListADT()
```

```
{
        // Call clear to clear the linked list structure of this
        // object instance
        ListADT<ItemType>::clear();

} // end default destructor

template<class ItemType>
ListNode<ItemType>* ListADT<ItemType>::getNodeAt(int position) const
{
        // Initialize our current point to the nullptr to return
        // if position is out of bounds
        ListNode<ItemType> *curPtr = nullptr;

        // If position is in bounds of linked list
        if (position >= 1 && position <= itemCount)
        {
                // Set current pointer to head of linked list and iterate
                // through the list until the requested position is reached
                curPtr = headPtr;
                for (int posIdx = 1; posIdx < position; posIdx++)
                        curPtr = curPtr->next;
        }

        // return the pointer at the requested position or
        // nullptr if position is out of bounds
        return(curPtr);
} // end of getNodeAt method

template<class ItemType>
bool ListADT<ItemType>::isEmpty() const
{
        // return boolean value reflecting if there
        // are any items in the linked list.
        return(itemCount == 0);

} // end of isEmpty Method

template<class ItemType>
int ListADT<ItemType>::getLength() const
{
        // return the number of items in the linked
        // list structure
        return(itemCount);

}// end of getLength method

template<class ItemType>
bool ListADT<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
        // Check to see if specified position is within the bound of the
        // linked list + 1 to allow for new addition on the end
        if (newPosition >=1 && (newPosition <= itemCount + 1))
        {
                // Create a new ListNode initialized with the newEntry
                ListNode<ItemType> *newNodePtr = new ListNode<ItemType>(newEntry);

                // If the newEntry is to be placed at the head of the list...
                if (newPosition == 1)
                {
                        newNodePtr->next = headPtr; // set next for new node to headPtr
                        headPtr = newNodePtr; // set headPtr to the new node
                }
                else
                {
                        // Get the previous node from the specified position
                        ListNode<ItemType> *prevPtr = getNodeAt(newPosition - 1);

                        newNodePtr->next = prevPtr->next; // set next for new node to next
                        prevPtr->next = newNodePtr; // set next of prev node to the new no
                }

                // Increase the number of items in the linked list and return our
                // success at the insert
                itemCount++;
                return(true);
        }

        // Return our failure to insert the newEntry
        return(false);
} // end of insert method

template<class ItemType>
ItemType ListADT<ItemType>::getEntry(int position) const throw(std::out_of_range)
{
        // If the specified position is in range for the linked list,
        // then return the item associated with the node at the specified position
        if (position >= 1 && position <= itemCount)
                return(getNodeAt(position)->item);

        // If position is out of range, then throw exception
        throw std::out_of_range("Position␣out␣of␣bounds");
} // end of getEntry method

template<class ItemType>
void ListADT<ItemType>::setEntry(int position, const ItemType& newEntry)
{
        // Find the node a the position specified
        ListNode<ItemType> *curPtr = getNodeAt(position);

        // If the node was found, set its item property to that specified
        if (curPtr != nullptr)
                curPtr->item = newEntry;
} // end of setEntry method

template<class ItemType>
int ListADT<ItemType>::getPosition(const ItemType& entry)
{
        int position = 0;
        ListNode<ItemType> *curPtr = headPtr;

        // Iterate through the linked list comparing nodes until there is a match
        for (position = 0; curPtr != nullptr; curPtr = curPtr -> next, position++);

        // If match, then return position, otherwise -1
        if (curPtr != nullptr)
                return(position);
        else
                return(-1);
} // end of getPosition method

template<class ItemType>
bool ListADT<ItemType>::contains(const ItemType& entry)
{
        ListNode<ItemType> *curPtr = nullptr;
        bool retVal = false;

        // Iterate through the linked list looking for entry and comparing to item
        // of every node in the linked list; if found, set retVal to true and stop iteratio
        if (itemCount >= 1)
                for (curPtr = headPtr; curPtr != nullptr && retVal == false; curPtr = curP
                        if (curPtr -> item == entry)
                                retVal = true;

        // return result of search
        return(retVal);
} // end of contains method

template<class ItemType>
bool ListADT<ItemType>::remove(int position)
{
        // If the position is within the bounds of the
        // linked list
        if (position >= 1 && position <= itemCount)
        {
                ListNode<ItemType> *curPtr = nullptr;

                // If to remove the headPtr, then ...
                if (position == 1)
                {
                        curPtr = headPtr; // temp set curPtr to headPtr (node to be remove
                        headPtr = headPtr->next; // set headPtr now to next node in linked
                }
                else
                {
                        // Get previous node in the linked list
                        ListNode<ItemType> *prevPtr = getNodeAt(position - 1);

                        curPtr = prevPtr->next; // temp set curPtr to node to be removed
                        prevPtr->next = curPtr->next; // set prev node's next to curPtr no
                }
                // Remove node pointed to be curPtr and reduce itemCount
                curPtr->next = nullptr;
                delete curPtr;
                curPtr = nullptr;
                itemCount--;

                // return our success
                return(true);
        }

        // return our failure
        return(false);
} // end of remote method

template<class ItemType>
void ListADT<ItemType>::clear()
{
        ListNode<ItemType> *curPtr = nullptr;

        // Set itemCount to reflect empty linked list
        itemCount = 0;

        // Iterate through nodes in linked list and remove the node
        // at the first position
        for (curPtr = headPtr; curPtr != nullptr; curPtr = curPtr -> next)
                remove(1);
} // end of clear method

template<class ItemType>
ItemType ListADT<ItemType>::getItem(ListNode<ItemType> *node) const throw(std::invalid_argu
{
        // Return the item of the node pointer if not nullptr
        if (node != nullptr)
                return(node->item);

        // If nullptr as argument throw exception
        throw std::invalid_argument("Can't getItem of nullptr");
} // end of getItem method

template<class ItemType>
ListNode<ItemType>* ListADT<ItemType>::getNext(ListNode<ItemType> *node) const throw(std::
{
        // Return the next of the node pointer if not nullptr
        if (node != nullptr)
                return(node->next);

        // If nullptr as argument throw exception
        throw std::invalid_argument("Can't getNext of nullptr");
} // end of getNext method

template<class ItemType>
bool ListADT<ItemType>::operator==(const ListADT<ItemType>& rhs) const
{
        bool isEqual = true;
        // If the linked lists are of different lengths, then we set our
        // return value to false
        if (itemCount != rhs.getLength())
                isEqual = false;
        else
        {
                ListNode<ItemType>* lhsPtr = headPtr;
                ListNode<ItemType>* rhsPtr = rhs.headPtr;

                // Linked lists are the same length, so we now iterate through each
                // node from both lists and compare the item property for equality
                while ( (lhsPtr != nullptr) && (rhsPtr != nullptr) && isEqual)
                {
                        ItemType lhsItem = lhsPtr->item;
                        ItemType rhsItem = rhsPtr->item;
                        isEqual = (lhsItem == rhsItem);

                        lhsPtr = lhsPtr->next;
                        rhsPtr = rhsPtr->next;
                }
        }

        // Return whether the linked list has identical nodes in the same order
        return (isEqual);
} // end of operator==

template<class ItemType>
ListADT<ItemType> ListADT<ItemType>::operator+(const ListADT<ItemType>& rhs)
{
        // Create a new ListADT object initialized from the linked list nodes
        // found in lhs operand - this pointer refers to the current instance object,
        // which for C++ operators is always the lhs operand
        ListADT<ItemType> concatList = ListADT<ItemType>(*this);

        // iterate through the nodes in the rhs operant and add these to the new
        // ListADT at the end of the list (getLength() + 1)
        for (ListNode<ItemType> *curPtr = rhs.headPtr; curPtr != nullptr; curPtr = curPtr->
                concatList.insert(concatList.getLength() + 1, curPtr->item);

        // return the new ListADT that is concatenation of the lhs and rhs
        return(concatList);
} // end of operator++

template<class friendItemType>
std::ostream& operator<<(std::ostream& outStream, const ListADT<friendItemType>& outputLis
{
        // set curPtr to the headnode of the ListADT operand
        // create a variable curItem to hold the item property of a node
        ListNode<friendItemType> *curPtr = outputList.headPtr;
        friendItemType curItem;

        // Iterate through all the nodes in the linked list
        while (curPtr != nullptr)
        {
                curItem = outputList.getItem(curPtr); // Set the curItem to the item prope
                curPtr = outputList.getNext(curPtr); // Set the curPtr to the next property
                outStream << curItem; // send to the outStream the item property
                if (curPtr != nullptr) // if the next node is not null, then send a space
                        outStream << "␣";
        }

        // return our outStream with the written contents from the
        // ListADT linked list
        return (outStream);
} // end of ostream operator<<

#endif
```

Makefile

```
#
# Makefile for Generating C++ executables
#
# F15 CSCI 232 - Data Structures and Algorithms
# Phillip J. Curtiss, Associate Professor
# Computer Science Department, Montana Tech
# Museum Buildings, Room 105
#
# Project-4: SortedListADT
# Date Assigned: 2015-11-16
# Date Due: 2015-11-30 by Midnight

# Define Macros related to printing and submitting programs
a2ps    = a2ps -T 2
mail    = mail
addr    = pcurtiss@mtech.edu rmoon@mtech.edu
tar     = tar -cvzf

# Define Macros to help generate the program file required
DIA     = dia2code
C++     = g++ -std=c++11
CFLAGS  = -Wall -Werror
LD      = g++
LDFLAGS =
LIBS    =
OBJS    = SortedListADT.o ListADT.o ListNode.o
EXEC    = proj4

# Provide dependency lists here, one on each line - don't␣forget␣to␣make␣sure
# if you have source files depending (or generated by) UML diagrams to include them as wel
.SUFFIXES:                      .dia
all:                            $(EXEC)
${EXEC}:                        ClientDriver.o
ClientDriver.o:         ClientDriver.cpp SortedListInterface.h
SortedListADT.o:        SortedListADT.cpp SortedListADT.h SortedListInterface.h ListADTInt
ListADT.o:                      ListADT.cpp ListADT.h ListInterface.h ListNode.h
```

```
ListNode.o:                    ListNode.cpp ListNode.h

##############################################################################
# Rules Used to Generate executable from object - DO NOT EDIT
$(EXEC):          $(OBJS)
        $(LD) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

# Rule to generate object code from cpp source files - DO NOT EDIT
.cpp.o:
        $(C++) $(CFLAGS) -c $<

# Rule to generate header and source files from Dia UML Diagram - DO NOT EDIT
.dia.h:
        $(DIA) -t cpp $<

.dia.cpp:
        $(DIA) -t cpp $<
##############################################################################

# Rule to Clean up (i.e. delete) all of the object and execuable
# code files to force make to rebuild a clearn executable only
# from the source files
clean:
        rm -f $(OBJS) $(EXEC)

# Rule to submit programming assignments to graders
# Make sure you modify the $(subj) $(msg) above and the list of attachment
# files in the following rule - each file needs to be preceeded with an
# -a flag as shown
subj    = "CSCI232_DSA_-_Proj4"
msg     = "Please_review_and_grade_my_Project-3_Submission"
submit: _PLACE_YOUR_SOURCE_CODE_FILES_HERE_
        $(tar) $(USER)-proj4.tgz $?
        echo $(msg) | $(mail) -s $(subj) -a $(USER)-proj3.tgz $(addr)

print:  _PLACE_YOUR_SOURCE_CODE_FILES_HERE_
        $(a2ps) $?
```