CS 202
Assignment #12

Purpose: Learn to about linked lists
Due: Thursday (4/18)
Points: 125

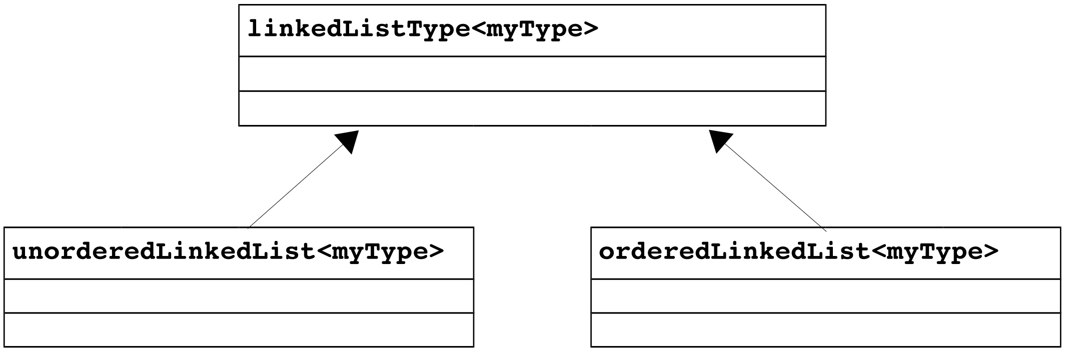
Assignment:

Design and implement three C++ classes;

- ***linkedListType***, that defines the basic properties of a linked list
- ***unorderedLinkedList***, derived from the class ***linkedListType*** class
- ***orderedLinkedList***, derived from the class ***linkedListType*** class

A main will be provided that uses the ***linkedListType*** class, the ***unorderedLinkedList*** class, and the ***unorderedLinkedList*** class.

The class hierarchy is shown as follows:



- Linked List Structure
  We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType info;
      nodeType<myType> *link;
};
```

- Linked List Class
  The linked list class will implement the functions.

linkedListType<myType>
#count: int
#*first: nodeType<myType>
#*last: nodeType<myType>
+linkedListType()
+linkedListType(const linkedListType<myType>&)
+~linkedListType()
+initializeList(): void
+isEmptyList() const: bool

`+print() const: void`
`+reversePrint() const: void`
`+length() const: int`
`+destroyList(): void`
`+front() const: myType`
`+back() const: myType`
`+firstPtr() const: nodeType<myType> *`
`+search(const myType&) const = 0: bool          (abstract)`
`+insert(const myType&) = 0: void                (abstract)`
`+insertLast(const myType&) = 0: void            (abstract)`
`+deleteNode(const myType&) = 0: void            (abstract)`
`-copyList(const linkedListType<myType>&): void`
`-recursiveReversePrint(nodeType<myType> *) const: void`

Linked List Type → Function Descriptions

- The *linkedListType()* constructor should initialize the list to an empty state (first = NULL, last = NULL, count = 0). The *initializeList()* function may be used. The *linkedListType(const linkedListType<myType>&)* copy constructor should create a new, deep copy from the passed list.

- The *~linkedListType()* destructor should linked list (releasing the allocated memory). The *destroyList()* function may be used.

- The *initializeList()* function should initialize the list to an empty state (first = NULL, last = NULL, count = 0). If the list is not empty, all current items should be deleted.

- The *isListEmpty()* function should determine whether the list is empty, returning ***true*** if the list is empty and ***false*** otherwise.

- The *print()* function should print all data elements of the linked list. Refer to the examples for formatting. The *reversePrint()* function should print all elements of the linked list in reverse order. The *reversePrint()* function must use the *recursiveReversePrint(nodeType<Type> *)* to provide a recursive implementation. Refer to the examples for formatting.

- The *length()* function should return the number of nodes in the list.

- The *destroyList()* function should delete all the nodes from the list (including ensuring that first = NULL, last = NULL, count = 0, when done).

- The *front()* function should return the first element of the list. The function must ensure that list exists. The *back()* function should return the last element of the list. The function must ensure that list exists. If the list is empty, the function should return NULL.

- The *firstPtr()* function should return a copy of the *first* pointer.

- The *search(const myType&)*, *insert(const myType&)*, *insertLast(const myType&)*, and *deleteNode(const myType&)* functions are abstract in the *linkedListType* class and no implemented is required in this class. However, an inheriting class must provide implementations.

- Unordered Linked List Class

  The unordered linked list class will implement the functions.

unorderedLinkedList<Type>
<no class variables>
+search(const myType&) const: bool
+insert(const myType&): void
+insertLast(const myType&): void
+deleteNode(const myType&): void

In the ***unorderedLinkedList*** class, we will need to access the protected class variables in the ***linkedListType*** class. The following statements should be included in the class definition.

```
protected:
      using linkedListType<myType>::count;
      using linkedListType<myType>::first;
      using linkedListType<myType>::last;
```

This is required in order for the derived class to recognize the inherited protected variables (from the base class) when a template is used.

Unordered Link List Type → Function Descriptions

- The *search(const myType&)* function determines whether specific item is in the list and returns ***true*** if the item found, and ***false*** otherwise. Since the list is unordered, the search must check every item in the linked list.

- The *insert(const myType&)* function should insert the item (passed) into the list at the beginning and update the count.

- The *insertLast(const myType&)* function should insert the item (passed) into the list at the end and update the count.

- The *deleteNode(const myType&)* function should remove the item (passed) from the list and update the count. If the item is not found, nothing should be changed.

- Ordered Linked List Class

  The unordered linked list class will implement the functions.

orderedLinkedList<myType>
<no class variables>
+search(const myType&) const: bool
+insert(const myType&): void

+insertLast(const myType&): void
+deleteNode(const myType&): void
+sortedMerge(const linkedListType<myType>&, const linkedListType<myType>&): void
-sMerge(nodeType<myType>*, nodeType<myType>*): nodeType<myType>*

In the ***orderedLinkedList*** class, we will need to access the protected class variables in the ***linkedListType*** class. The following statements should be included in the class definition.

```
protected:
     using linkedListType<myType>::count;
     using linkedListType<myType>::first;
     using linkedListType<myType>::last;
```

This is required in order for the derived class to recognize the inherited protected variables (from the base class) when a template is used.

Ordered Link List Type → Function Descriptions

- The *search(const myType&)* function determines whether specific item is in the list and returns ***true*** if the item found, and ***false*** otherwise. Since the list is ordered, the search should stop when it either finds the items or passes the possible location in the linked list.

- The *insert(const myType&)* function should insert the item (passed) into the list in its appropriate position (sorted). The *insertLast(const Type&)* function should use the *insert(const myType&)* function in order to maintain the sorted order.

- The *deleteNode(const Type&)* function should remove the item (passed) from the list and update the count. If the item is not found, nothing should be changed. Since the list is ordered, the search should stop when it either finds the items or passes the possible location in the linked list.

- The *sortedMerge (const orderedLinkedList<myType> &, const orderedLinkedList<myType> &)* function merges two sorted lists (which are passed in) into one sorted list. The *sortedMerge()* function will use the *firstPtr()* function to obtain the first pointers for each list and call the private *sMerge()* function, which will merge the lists recursively.

- The *sMerge(nodeType<Type> *, nodeType<Type> *)* should accept two nodes and return a pointer to a newly created new containing one of the passed nodes (the smallest one). The new node should be linked to the next smallest item (i.e., a recursive call to *sMerge*). The *sMerge* function must be implemented in a recursive manner.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 202 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Make File:

You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executable.

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

Example Execution:

Below is an example program execution for the main.

```
ed-vm% ./main

CS 202 - Assignment #12
Linked Lists

----------------------------------------------------------------------
List 1 - Integers

The item to be inserted is already in the list.
List 1 - Unordered:
55 43 11 56 125 88 32 14 3 90 89 2 19 4 56 34 22

List 1 Unordered, in reverse order:
22 34 56 4 19 2 89 90 3 14 32 88 125 56 11 43 55

List 1 Ordered:
2 3 4 11 14 19 22 32 34 43 55 56 88 89 90 125

List 1 Ordered, in reverse order:
125 90 89 88 56 55 43 34 32 22 19 14 11 4 3 2

----------------------------------------------------------------------
List 2 - Integers:

The item to be inserted is already in the list.
List 2 length: 17
List 2 Unordered:
22 34 56 4 19 2 89 90 3 14 32 88 125 56 11 43 55


List 2 Unordered New Length: 14
List 2 Unordered -> With nodes removed:  0  14  32
22 34 56 4 19 2 89 90 88 125 56 11 43 55

List 2 Unordered -> Search tests (for 14 and 90):
List 2: item 14 was not found.
List 2: item 90 was found.

----------------------------------------------------------------------
List 3 - Doubles:

List 3 Unordered:
22.1 34.5 56.6 4.3 19.2 2.5 89.4 90.8 3.14 14.3 32.9 88.1 125.3 56.7 11.5 43.8
55.2
```

```
List 3 Unordered (reverse order):
55.2 43.8 11.5 56.7 125.3 88.1 32.9 14.3 3.14 90.8 89.4 2.5 19.2 4.3 56.6 34.5
22.1

List 3 Unordered -> Search tests (for 11.5 and 19.2):
List 3: item 11.5 was found.
List 3: item 19.2 was not found.

List 3 Ordered:
2.5 3.14 4.3 11.5 14.3 19.2 22.1 32.9 34.5 43.8 55.2 56.6 56.7 88.1 89.4 90.8
125.3

List 3 Ordered (reverse order):
125.3 90.8 89.4 88.1 56.7 56.6 55.2 43.8 34.5 32.9 22.1 19.2 14.3 11.5 4.3 3.14
2.5

List 3 Ordered -> Search tests (for 11.5 and 19.2):
List 3: item 11.5 was found.
List 3: item 19.2 was not found.

----------------------------------------------------------------------
List 4 - Doubles:

List 4 Unordered (original list):
22.1 34.5 56.6 4.3 19.2 2.5 89.4 90.8 3.14 14.3 32.9 88.1 125.3 56.7 11.5 43.8
55.2

List 4 Unordered Delete Testing...
The item to be deleted is not in the list.

List 4 Unordered (modified):
56.6 4.3 19.2 2.5 89.4 90.8 14.3 32.9 88.1 125.3 56.7 11.5

List 4 Unordered - first item: 56.6
List 4 Unordered - last item: 11.5


List 4 Ordered (original list):
2.5 3.14 4.3 11.5 14.3 19.2 22.1 32.9 34.5 43.8 55.2 56.6 56.7 88.1 89.4 90.8
125.3

List 4 Ordered Delete Testing...
The item to be deleted is not in the list.

List 4 Ordered (modified):
2.5 4.3 11.5 14.3 19.2 32.9 56.6 56.7 88.1 89.4 90.8 125.3

List 4 Ordered - first item: 2.5
List 4 Ordered - last item: 125.3

----------------------------------------------------------------------
List 5 - Short's (0-4):

List is initially empty, adding 4, 3, 2, 1 and 0.

List 5 Unordered:
4 3 2 1 0

List 5 Unordered - first item: 4
List 5 Unordered - last item: 0
List 5 Ordered:
0 1 2 3 4

List 5 Ordered - first item: 0
List 5 Ordered - last item: 4
----------------------------------------------------------------------
Lists 6 and 7 - Even Short's (2-20)
```

```
Copy of list 5 with 5, 6, 7, 8, and 9 added.

List 6 Unordered (length=10) : 4 3 2 1 0 5 6 7 8 9
List 7 Unordered (length=10) : 4 3 2 1 0 5 6 7 8 9

Destroying List 6 Unordered...
List 6 Unordered (should be empty):

Initializing List 7 Unordered...
List 7 Unordered (should be empty):


List 6 Ordered (length=10) : 0 1 2 3 4 5 6 7 8 9
List 7 Ordered (length=10) : 0 1 2 3 4 5 6 7 8 9

Destroying List 6 Ordered...
List 6 Ordered (should be empty):

Initializing List 7 Ordered...
List 7 Ordered (should be empty):


----------------------------------------------------------------------
List 8 Ordered - Integers:

List A Ordered:
2 4 6 8 10 12 14
List B Ordered:
1 3 5 7 9 11 13 15
List C Ordered:
0
List D Ordered:
50

List 8 Ordered (sorted merge):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
List 8 Length: 15

List 9 Ordered (sorted merge, A only):
2 4 6 8 10 12 14
List 9 Length: 7

List 10 Ordered (sorted merge, B only):
1 3 5 7 9 11 13 15
List 10 Length: 8

List 11 Ordered (sorted merge, none):

List 11 Length: 0

List 12 Ordered (sorted merge, 0+A):
0 2 4 6 8 10 12 14
List 12 Length: 8

List 13 Ordered (sorted merge, A+50):
2 4 6 8 10 12 14 50
List 13 Length: 8

----------------------------------------------------------------------
String List:

String List is initially empty, adding words...

String List Unordered:
hills dog familiar big jumps a in green enters

String List Unordered - first item: hills
String List Unordered - last item: enters

String List Unordered is now empty.

String List Ordered:
a big dog enters familiar green hills in jumps

String List Ordered - first item: a
String List Ordered - last item: jumps

String List Ordered is now empty.

----------------------------------------------------------------------

Game Over, thank you for playing.

ed-vm%
```