CS2308 - Foundations of Computer Science II
Program Assignment #3
(100 points = Extra credit for up to 10 points towards your final grade)

Due 11:59 pm on May 7, 2020

In this project, you will implement an ItemList (with integers as iterms) using classes. The purpose of this project is to:

- Learn how to separate the class specification and implementation
- Learn how to compile and link multiple files using a makefile
- Learn how to translate a UML diagram to a class description
- Learn how to write a test program to evaluate the correctness of your class implementation.

The Itemlist
The concept of a "list of items" corresponds to ordinary occurrences of to-do lists, grocery lists, membership lists, and so forth. To help distinguish the list from the pure array that implements it, "typical" array-subscript notation is avoided, replaced with pointer operations. Terminology is "neutralized": instead of "first" and "last" "elements" of an "array" there are "top" and "bottom" "items" of a "list".

Operations
Some operations on a list are shared with those we associate with arrays. Items are stored sequentially, making lists "linear". Items in your list are all the same type. The items can be searched and sorted. Lists are "traversed" (inspected item-by-item) in a "direction" from one point to another (usually, top to bottom or bottom to top); this is usually done to either search or output (print, display) the list.

Some operations on a list are not the typical set we associate with arrays. Elements of an array are typically assigned and changed, while items in a list are typically inserted or removed, often rearranging the list in the process. Elements of a list move from an item to the “next” or “previous” item using operators with those names, while elements in an array move from one index to another by computation (incrementing or decrementing).

Attributes
For the list itself, we have three attributes: **type**, **length**, and **size**. Size is not the number of items on the list; it is the maximum number of items we are able to keep on the list. Length is the number of items on the list at the moment. Type is the kind of items kept in the list: in a grocery list, items are the names of food items ("strings"); in this project, items are simply counting numbers (integers).

For each item on a list, we have two attributes: **value** and **position**. Value is simply the data of the item's type stored in the item. Position is somewhat more complicated in a computer implementation. For a grocery list, we can think of items as "first", "fourth", or "last" on the list. In our implementation, using pointer notation and arithmetic, items are located by their addresses. The name of the list is a pointer to the first element of the list. The "top" of the list (in other systems, the "head") is also the address of the first item; however, it is set to NULL when the list is empty. The "next available" item is the address of the item after the "bottom" item (which, in other systems, is referred to as the “end” or “tail”). It is useful to keep this address and compute the address of the "bottom" item; the “next available” item is equal to the name of the list when the list is empty. We also keep track of a “current” item on the list; it is the address of the list item currently being inspected, for whatever reason.

Public Interface

The public interface for class **ItemList** ***is completely centered on the current item***. Calling programs are given direct access to ONLY the current item. *All* searches move the current address, unless they fail. *Only* the value at the current address may be set or retrieved. All inserts are *before or after the item at the current address* and the inserted item becomes the current item. *Only* the current item may be removed. The calling program must position itself at the item (it may use class **ItemList** *movement methods* to do so) before it can then delete it; this includes searching for a value to remove. After an item is removed, the item *before* the removed item becomes the current item (unless it was the **top** item, in which case the new top item becomes the current item; if it was the only item, all pointers are reset). **Top** and **Bottom** are *not* identified as being "larger" or "smaller" than one another. Movement toward the **Bottom** is "moving to the **next** item"; movement toward the **Top** is "moving to the **previous** item". This means that a calling program is totally unaware of whether an array or linked list is being used to implement the item list, and whether it "builds" from left to right or from right to left. [*This becomes important with building stacks and queues with an item list.*]

The following information may be retrieved about the list:

1) the length and maximum size of the list,
2) whether the list is full or empty,
3) whether the current item is at the **Top** or **Bottom**, and
4) the value of the current item.

The UML description of class ItemList is show below:

ItemList
- listItem : int* - topItem : int* - nextAvailable : int* - currentItem : int* - maxSize : int
+ ItemList(arraySize : int) + ~ItemList() + setCurrentValue(value : int) : bool + insertBeforeCurrentItem(value : int) : bool + appendToList(value : int) : bool + removeCurrentItem() : bool + clearList() : void + search(value : int) : bool + moveToNextItem() : bool + moveToPreviousItem() : bool + moveToTopItem() : bool + moveToBottomItem() : bool + getListLength() const : int + getMaxSize() const : int + isEmpty() : bool + atTop() : bool + atBottom() : bool + getCurrentValue() const : int + displayAll() const : void

Tasks:

1. Add a method **removeValue(int);** . In order to remove the first item with a particular value, a calling program must first search for the value (making that item the current item) and then use **removeCurrentItem( )**. The new method will perform the search and then remove the item, adjusting all pointers accordingly. There is a function in the test program [**testRemoveVal( )**] that does the search externally, so you have a pattern to use. The removal process is pretty much the same as in **removeCurrentItem( )**. Modify the test function **testRemoveVal( )** to call **removeValue( )** instead of searching and calling **removeCurrentItem( )**.
2. Add a method **readFile( )** which will read integer values from a file named as *numbers.txt*. The method will open, check and close *numbers.txt*. The integers are put in the list using the **appendToList( )** method. Read the file one integer at a time; if there are too many integers for the size of the array, simply ignore the extras.
3. Then fill in test function **testReadFile( )** (its prototype is in the test program header, and definition "shell" is in the test program source file) and test the new method.
4. Add a public method **selectionSort( )**. This method implements the Selection Sort algorithm (which should be included in the header block comments for this function).
5. Add a private method **bubbleSort( )**. This method implements a Bubble Sort according to the algorithm below (which should be included in the header block comments for this function). It is not publicly available and is called by **binarySearch(int)** to sort the list prior to the Binary Search.
6. Then add a test function **testSortFile( )** to the source file **ItemListTests.cpp** to test the new methods (you will have to add a prototype, and also a **#define** statement, to **ItemListTests.h** in order to activate it in the same way other tests are activated).
7. Add a public method **binarySearch(int)** which implements the Binary Search algorithm (this algorithm should be included in the header block comments for the function). Notice that the first step in the algorithm is to sort the list. You will use the private bubbleSort() method for this particular sort.
8. Write a **makefile** to compile and execute the test program, creating a class object in the process.
9. Add the new functions to the UML diagram in ItemListClass.h.

Makefile

The makefile should have a rule for making each object file separately, from its associated source file. There should be a rule that makes the executable file **testItemList** from the object files, and a final rule for "testRun" (i.e., you make it by typing *make testRun*) that executes **testItemList**. The dependencies for each step are such that, if a needed object or executable file is not present, another rule exists to create it. If your program does not compile using the submitted makefile in the Linux server, *it does not considered as compile successfully*, whether or not it might compile in another system, or by using **g++** on the Linux command line.

The following are some hints on how to implement the **selectionSort, bubbleSort** and **binarySearch** algorithms in your itemlist class:

Binary Search Algorithm (a boolean function for an Item List)

[Do not change the pointer to the current item unless the search item is found.]
Sort the list in ascending order from top to bottom
Set the "first item" (a pointer) to be the top of the list.
Set the "last item" (a pointer) to be the "bottom item" ("next available item" – 1)
Set a boolean flag, e.g. "found", to false.
While "found" is false and the "first item" is less than or equal to the "last item"
    Calculate midpoint of the search area ("last item" – "first item") / 2
    ["integer division" -- truncate if result is not an integer]
    If the mid-array element ("first item" + midpoint) equals the desired value
        Set "found" to be true.
        Set the current item to be "first item" + midpoint.
    Otherwise, "narrow" the search area
        If the midpoint item is greater than the search value
            set the "last item" equal to the address of the midpoint item - 1
        Otherwise
            set the "first item" equal to the address of the midpoint item + 1
        End If
    End If
End While.
[At this point, if the loop ended because the first and last indices "crossed", the item was not found; otherwise it was found and the item with the value is now the current item.]
Return "found"

Bubble Sort Algorithm (a boolean function for an Item List)

[Traverse the list repeatedly, swapping values on each traversal, until a traversal where no swaps occur]
Do
    Set a boolean flag, e.g. "swap", to false.
    Move to the top of the list (sets the current item to the top)
    While not at the item above the bottom (i.e., the "next available item" – 2)
        If the current item value is greater than the next item value
            [Swap the two items]
            set a temporary value to be the current item value
            set the current item value to be the next item value
            set the next item value to be the temporary value
            set "swap" to be true
        Move to the next item
    End While
     [Now: the largest item value has been moved to the bottom of the list while smaller values are "bubbling" toward the top of the list. In repeated passes, the current item will reach the lower, sorted, portion of the list and its value will not be greater than any of those item values, so fewer and fewer swaps will take place]
While "swap" is true
[The current item should be the bottom item at the end of the last, swapless pass. Test that it is so, and return the result of the test (true or false)]

Selection Sort Algorithm (a boolean function for an Item List)

Move to the top of the list (sets the current item to the top)
[Traverse the remaining list]
While the current item is not the bottom item,
Set the minimum value (a variable) to be the value of the current item
Set the "minimum item" (a pointer) to be the current item
[Traverse the remaining list looking for the location of the lowest value]
Set the "scan item" (a pointer) to be the item after the current item
While the "scan item" is less than the "next available item"
If the value at the "scan item" is less than the minimum value
set the minimum value to be the "scan item" value
set the "minimum item" to be the "scan item"
Increment the "scan item"
End While
[Now: the current item marks the first value of the remaining list and the minimum item marks the item
with the smallest value in that remaining list.
Swap the values and move the current item to the next item -- this creates a new,
smaller search area to traverse.
All the values previous to the current item have been sorted
in ascending order as the current item moves on shrinking the list]

Set the value of the "minimum item" to the value of the current item
Set the value of the current item to be the minimum value
Move to the next item
End While
[The current item should be the bottom item at the end of the last, swapless pass.
Test that it is so, and return the result of the test (true or false)]

Comments and Suggestions:

DO NOT DELAY. Start the project as early as possible. Do not try and write the entire program all at one time. Work on the program in small sections.

Notes:

- **Your code will be graded in Linux system, make sure to test your code on the Linux server.**
- **A program that does not compile in Linux, using the makefile, is graded as not compiling.**

Program Submission

Please submit only one zip file (firstname_lastname_prog3.zip) to TRACS. Your zip file should contain the following files:

- ItemListClass.h
- ItemListMethods.cpp
- ItemListTests.h
- ItemListTests.cpp
- makefile
- numbers.txt

You will not get extra credit if you fail to submit your project to TRACS before the deadline.

Your program will be graded as follows, please make sure to check each item before you submit.

Program and Run Time Output: _____ (94 Points)

_____ (10) Task 1

_____ (5) Correctly implement the **removeValue** function

_____ (5) Correctly implement the **testRemoveVal** function

_____ (10) Task 2

_____ (3) Correctly add the **readFile** function

_____ (7) Correctly implement the **readFile** function

_____ (10) Task 3

_____ (3) Correctly add the **testReadFile** function

_____ (7) Correctly implement the **testReadFile** function

_____ (10) Task 4

_____ (3) Correctly add the **selectionSort** function

_____ (7) Correctly implement the **selectionSort** function

_____ (10) Task 5

_____ (3) Correctly add the **bubbleSort** function

_____ (7) Correctly implement the **bubbleSort** function

_____ (10) Task 6

_____ (3) Correctly add the **testSortFile** function

_____ (7) Correctly implement the **testSortFile** function

_____ (20) Task 7

_____ (5) Correctly add the **binarySearch** function

_____ (5) Correctly call the prive **bubbleSort** function before searching

_____ (10) Correctly implement the **binarySearch** function

_____ (10) Task 8

_____ (10) Correctly write the **makefile**

_____ (4) Task 9

_____ (4) Correctly add the new functions to the UML diagram in ItemListClass.h

Coding Standards: _____ (6 Points)

_____ (3) Documentation (program and function headers)

_____ (3) Comments Indentation Scheme / Use of { }

If your code cannot be compiled or executed, your final score will only be 50% of your accumulated score of each item above.

Final Score: _____ (100 Points)