CS 201 Data Structures Library Phase 1 Due 9/25

For phase 1 of the CS201 programming project, we will start with a circular dynamic array class and extend it to implement some of the algorithms discussed in class.

Your dynamic array class should be called CDA for circular dynamic array. The CDA class should manage the storage of an array that can grow and shrink. The class should be implemented using C++ templates. As items are added and removed from both the front and the end of the array, the items will always be referenced using indices 0…size-1. Your CDA class should include a flag indicating whether the array is in sorted in increasing or decreasing order, or it is unordered.

The only include files that should be necessary for the CDA class would be related to generating random values for Quickselect.

The public methods of your class should include the following (elmtype indicates the type from the template):

Function	Description	Runtime
CDA();	Default Constructor. The array should be of capacity 1, size 0 and ordered is false.	O(1)
CDA(int s);	For this constructor the array should be of **capacity** and **size** s with ordered = false.	O(1)
~CDA();	Destructor for the class.	O(1)
elmtype& operator[](int i);	Traditional [] operator. Should print a message if i is out of bounds (outside 0…size-1) and return a reference to value of type elmtype stored in the class for this purpose.	O(1)
void AddEnd(elmtype v);	increases the size of the array by 1 and stores v at the end of the array. Should double the capacity when the new element doesn't fit. If ordered is true, check to be sure that the array is still in order.	O(1) amortized
void AddFront(elmtype v);	increases the size of the array by 1 and stores v at the beginning of the array. Should double the capacity when the new element doesn't fit. The new element should be the item returned at index 0. If ordered is true, check to be sure that the array is still in order.	O(1) amortized
void DelEnd();	reduces the size of the array by 1 at the end. Should shrink the capacity when only 25% of the array is in use after the delete. The capacity should never go below 4.	O(1) amortized
void DelFront();	reduces the size of the array by 1 at the beginning of the array. Should shrink the capacity when only 25% of the array is in use after the delete. The capacity should never go below 4.	O(1) amortized
int Length();	returns the size of the array.	O(1)
int EmptySlots();	returns the capacity - size of the array.	O(1)
void Clear();	Frees any space currently used and starts over with an array of capacity 1 and size 0.	O(1)

int Ordered();	Returns the status of the ordered flag. 1 is increasing order, -1 is decreasing order, 0 is unordered.	O(1)
int SetOrdered();	Check to see if the array is in order. Set the order flag appropriately. Return 1 if the array was in increasing order, -1 if the array was in decreasing order and 0 otherwise.	O(size)
Elmtype Select(int k);	returns the k^{th} smallest element in the array. If the array is in increasing order, then return the item at index k-1. If the array is in decreasing order, then return the item at index size - k. Otherwise use the quickselect algorithm. Quickselect should choose a random partition element.	O(1) or O(size) expected
Void InsertionSort()	Performs insertion sort on the array. Sets ordered to true. Should return decreasing order.	O(size*size) worst case
void MergeSort();	Sorts the values in the array using the mergesort algorithm. Set ordered to true. Should return decreasing order.	O(size lg size) worst case
void CountingSort(int m);	Sorts the values in the array using counting sort, where the values in the array are in the range 0…m. Set ordered to true. You may assume that all values in the array are integers in the range 0…m. Should return decreasing order.	O(size * m)
int Search(elmtype e)	If the array is ordered in either increasing or decreasing order, perform a binary search of the array looking for the item e. Otherwise perform linear search. Returns the index of the item if found or -1 otherwise.	O(lg size) or O(size)

Your class should include proper memory management, including a destructor, a copy constructor, and a copy assignment operator.

For submission, all the class code should be in a file named CDA.cpp. Create a makefile for the project that compiles the file Phase1Main.cpp and creates an executable named Phase1. A sample makefile is available on Blackboard. Place both CDA.cpp and makefile into a zip file and upload the file to Blackboard.

- ☐ Create your CDA class
- ☐ Modify the makefile to work for your code (changing compiler flags is all that is necessary)
- ☐ Test your CDA class with the sample main provided on the cs-intro server
- ☐ Make sure your executable is named Phase1
- ☐ Develop additional test cases with different types, and larger arrays
- ☐ Create the zip file with CDA.cpp and makefile
- ☐ Upload your zip file to Blackboard

No late submissions will be accepted. There will be an opportunity to resubmit by 10/9 Resubmissions will have a 20 point penalty.