CS 201 Data Structures Library Phase 1 Due 2/4

DRAFT

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 order, or it is unordered.

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 and return a reference to value of type elmtype stored in the class for this purpose~~. If the ordered flag is true, verify that any change in the array leaves it still ordered. If necessary set the ordered flag to false.~~ Removed this test.	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 1.	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 1.	O(1) amortized
int Length();	returns the size of the array.	O(1)
int Capacity();	returns the capacity of the array.	O(1)
int Clear();	Frees any space currently used and starts over with an array of capacity 1 and size 0.	O(1)

bool Ordered();	Returns the status of the ordered flag.	O(1)
int SetOrdered();	Check to see if the array is in order. Set the order flag appropriately. Return 1 if the array was ordered and -1 otherwise.	O(size)
Elmtype Select(int k);	returns the k^{th} smallest element in the array. If ordered is true then return the item at index k-1. 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.	O(size*size) worst case
void QuickSort();	Sorts the values in the array using the quick sort algorithm. This should pick the partition value using the median of three technique. Set ordered to true. Should also make use of insertion sort to improve performance.	O(size * size) worst case O(size lg size) expected
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.	O(size * m)
int Search(elmtype e)	If ordered is true, 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.