**Assignment: Sort Algorithm Within Three Classes**

**The Classes**

As you have seen so far, there are a number of different means by which arbitrary data payloads can be held by an object. As is the case with Abstract Data Types (ADTs), it does not matter so much to the user how the data is held, but more about its attributes and what can be done with it. In this exercise, you are going to be working with three of them, three implementations of the same class if you like, all capable of having their data accessed via an index value. Unlike the previous exercise’s ordered list, these have no implicit form of order except that driven by the user. The data repository classes you’ll be developing (all named the same if you like) are implemented with:

1. A Simple Array (growable via data copying),
2. A Linked List of Arrays (growable by adding additional array-based objects to the list, see chart 26), and
3. A Simple unordered linked list (growable by adding node objects to the list).

All of these will be implemented, perhaps as three implementations of the same class, perhaps as separate classes. (*So, yes, this is a more involved assignment, with more points and more time as a result.)*

Unless you desire otherwise, the only methods that must be implemented over all of these are:

**Constructor** …

* For the Simple Array, a default constructor will decide on the initial amount of empty array space to allocate.
* Similarly, for the Linked-List of Arrays, the default constructor will allocate a single array object of some size (e.g., 32, 100) linked by a single header object. Assume that the number of values to be later inserted into the array exceeds this. For example, you may set the number of entries per array per a constant – say 10 or 32, whatever is easy for you. But it must be less than the number of values that the test case will be inserting into the object. Each new linked array can be of equal size.
* For the Linked List, this could start off as a simple empty list.

Your object will also be keeping track of the number of integer values residing in the added to the object. So if you would prefer to preallocate storage for the Linked-List Array and simple linked list, that too would be acceptable. (The tests, though, ought to be such that the object will need to grow to allow the additional of all of the values added.

**Add** … For this exercise, the user should be able to assume that values inserted into the array are packed forward into this object. As each new integer value is added, it is inserted into what would correspond to the next available entry in the object. The input values added to the object will be a single unique positive integer value. Just as with the ordered list exercise, this integer value will be the only payload. The object should also keep track of the number of these integer values already in the object. Recall that the constructor will have specified an array with fewer entries than will be ultimately added; the array-based implementation(s) will need to determine whether more storage is required and take appropriate action on some Add operations.

**Remove** … There will be no Remove method required in this exercise. As a result, there will also be no need for compressing the contents of the object. You are, though, welcome to support – and test – a remove method.

**Find Indexed** … Because the values added are guaranteed to be compressed forward in the objects, the user can refer to the contents of the objects via an input index value. The Find Index value will return the integer value at the indexed location if one exists or will throw an Out\_Of\_Bounds exception if the input integer value is negative or exceeds the number of values then in the object. You can choose whether the input index value is 0-based or 1-based, documenting such appropriately, and adjusting same when determining whether an Out\_Of\_Bounds exception is to be thrown

**Sort** … Notice that, no matter the implementation, the methods so far are unaware of the implementation. That continues to be the case with this method, the **Sort**. The user will have added arbitrary integer values into your object, with no guarantee that they are sorted in some manner. (They may well have been sorted per your implementation, but the user remains unaware of this.) As such, your class also provides a **Sort** method, allowing the user to force the contents of the object – at least temporarily – into some sorted order. Although there are many forms of “sort” possible, we’ll assume here that the Sort implies an ascending value sort; the low indexed location contains a lower value. There are also many means by which this sorting can be accomplished. This assignment leaves the choice of which up to you. D2L-published charts outline a few of these and your text covers these and others. Code for the array-based implementations can be found in each; you will be responsible for altering same to work with the other two classes. [I recommend completing these methods for the array-based class first for this reason.]

*[This list of methods represents the user’s only means of accessing the data registered in these objects. As such, even though the object may be implemented via an array, the user is incapable of using normal Java array syntax to access into what you happen to know is an array.]*

**The Test Case(s)**

1. Construct the object.
2. Add an arbitrary number of unique positive integer values, a number of adds at least four times the size of the pre-allocated size in the default constructor.
3. Invoke the sort method.
4. Ensure that the object’s contents is sorted via the Find Indexed method (at least).
5. Add an equal number of similarly positive integer values to the now sorted object.
6. Invoke the sort method.
7. Again, ensure that the object’s contents is sorted via the Find Indexed method (at least).