COIS 2020H-Data Structures & Algorithms

Winter 2024

Assignment 2 (12 %)

Overview:

Note: do not worry about this lengthy document; you may see that this assignment has many requirements. However, some of these requirements are trivial or variations on other methods.

In Assignment 1, you created an Array and a Linked List using built-in arrays and linked lists of Mobile Objects you defined yourself. In this assignment, you are to build your own ArrayList (this is basically a generic array) and a doubly Linked List.

Requirement:

1. Animal Class: it is basically the Animal class created in Assignment #1.
2. Animal subclasses:
    - Two types of these subclasses are already implemented in Assignment 1 (Cat and Snake). So, you can basically use these classes with the following modifications:
        - Generate their starting positions **randomly** in the range -25 to + 25 in x and y, while z should be initialized to (0).
        - Cats and Snakes need a new method, `Eat()`, which takes a parameter of type "Bird" (A type of animal) and deletes it. If a cat/snake gets close enough to a bird, it will eat it (details discussed later).
    - You are also going to make a new animal subclass: `Bird`. `Birds` can have the following names: {"Tweety", "Zazu", "Iago", "Hula", "Manu", "Couscous", "Roo", Tookie", "Plucky", "Jay"}. Generate their starting positions **randomly** in the range -100 to + 100 in X and Y, and between 0 and 10 in Z.
    - You do need to make the objects comparable - for now, make them comparable by Name (so they can be sorted Alphabetically by name). [we already discussed comparable. Refer to MSDN for more help https://docs.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-5.0)
- From their starting positions, birds will move randomly, at most +/-10 in x/y and +/- 2 in Z. (Clamped to +/- 100 in x, y, and 0 to 10 in Z).
- An explanation of what these do is in the discussion of `main()` below.

To complete this assignment, you will need two Collections, most of which you implement yourself. A List built using an array (essentially, this is the generic array), and a list built as a (Doubly) Linked List. Following are more details;

A. ArrayList (Of Generics), with the following requirements

1. A constructor: (`Public ArrayList (T)`). This can create an empty list.
2. `Private Grow()`: this should double the size of the array when it Grows. This method will get called by `AddFront` or `AddLast` if the array is out of space.
3. `AddFront`: this should add an item at the front (index 0) of the list (and move the rest of the list out of the way first)

4. `AddLast`: this should add an item after all current items (Hint: this is NOT Length -1 though). You'll probably need GetCount for this, or you may think of other solutions.
5. `GetCount`: this should return the number of items in the list (not the length of the array)
6. `InsertBefore`: this takes two object parameters, and it will insert the first object (first parameter) **before** the object in the second parameter.
7. `InPlaceSort`: This should sort the array list in place **without** making a new Array. For now, use built-in array sorting. (Make sure your `Animals` are comparable, and compare by *Name*)
8. `Swap(index1, index2):` should swap the two elements in the array list
9. `DeleteFirst`: this should delete the object at position 0, and shuffle all the elements back one to fill the hole
10. `DeleteLast`: This should delete the last **non-null** element (and update the count correctly).
11. `RotateLeft`: Should rotate all elements in the array left. An example of RotateLeft: Given a list {A, B, C, D} rotate all elements left one place becomes {B, C, D, A}
12. `RotateRight`: Reverse of RotateLeft {A, B, C, D} -> {D, A, B, C}
13. `Public Static ArrayList Merge`: this should take two ArrayLists, and return a third which is the first two merged in an unsorted order. (Remember to delete the first two if and only if you're done with them, which you most likely won't be)
14. `StringPrintAllForward`: This returns a string which is the whole structure printed from beginning to end (calling the object.ToString or similar)
15. `StringPrintAllReverse`: Same as above but in reverse order.
16. `Deleteall.` Delete all elements, not the list.

B. Doubly Linked List (of Generics):

This should have a head and a tail node, and nodes should have references to both the next and previous.

1. A constructor: (`Public DoublyLinkedList (T)`). This can create an empty list
2. `AddFront`: Adds a new node at the head
3. `AddLast`: Adds a new node at the tail
4. `GetCount`: Should be O(1) meaning you need to keep track of the count in other methods
5. `InsertAtRandomLocation`: keep track of the size of the list, generate a random number up to that size, insert at that position in the list
6. `Merge`, unlike ArrayList, this should be a method of a single list and should merge another list onto the calling one at the end (and update the count correctly). That is, the method will take one parameter.
7. `Public T FindClosest(Animal A)` [this could instead be public Object FindClosest or public Node FindClosest]: Searches the list to find the closest bird to the cat or snake position of the animal in the parameter A (alternatively you could just pass in the position of A). Use brute force search, meaning that you search the elements one by one. The equation for distance between two points is:
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$
8. `Public double FindDistance (position pos, object T)` this will return the distance from an object with position pos to the object.
9. `DeleteFirst`
10. `DeleteLast`
11. `GetEaten(object T target)` – this deletes "target" from the list. Make sure to test correctly the head/first and tail/last of the list as well as objects in the middle. This should print off which bird was eaten to the console (use `target.ToString()`)

12. `RotateLeft`
13. `RotateRight`
14. `StringPrintAllForward`: This returns a string, which is the whole structure printed from beginning to end (calling the `object`.ToString or similar)
15. `StringPrintAllReverse`: Same as above but in reverse order.
16. `DeleteAll.` Delete all elements, not the list.

C. Main method:

1. Create an `ArrayList` of Animals containing 3 Cats using `Addfront`.
2. Create an `ArrayList` of Animals containing 3 snakes using `Addlast`.
3. Merge the two lists.
4. Test `PrintallForward`, `PrintAllReverse` on the new (the merged) ArrayList.
5. Make a doubly linked list where the first 5 birds randomly choose from the names provided.
6. Make a second doubly linked list of the remaining 5 birds.
7. Merge the second list onto the first.

Your main program should have a while loop with the following procedure:

- Each iteration over the animals list the Cat or Snake will eat Birds that are in range (They should print off that they are eating something). Cats have a range of 8, snakes have a range of 3.
- If there is nothing in range to eat, the animal move towards the nearest bird. Cats move at a speed of 16 (i.e., the position will be changed by 16), snakes at a speed of 14.
- Then birds "that survive" will **randomly** move as per their rules above.
- This loop repeats **until all birds** have been eaten.
- After every fifth iteration, `PrintallForward`
- Report how many rounds it took.

Note on movement: Speed is a straight-line distance, cats and snakes only move in X and Y.

Test out each of the methods for your structures (basic test cases: Make sure it doesn't crash if the list is empty, make sure it works in a normal use case, don't worry about testing extremely large numbers of objects).

Quick Theory Notes: The two different approaches to merge (the one in arraylist and the one in linkedlist) are both valid in some sense. Which one you should prefer depends a bit on the appropriate design/responsibility of the relevant classes and possibly any memory constraints. The linkedlist methods related to distances and getting eaten are an example of where using your own data structure can give advantages over built in ones, since a built in one wouldn't really be able to work like that.

…..Check next page

Submission guidelines:

Use the provided submission template where you provide a copy of your testing and code pasted. Name it for your ***trentusername.docx***. Upload that, along with a copy of the same document saved as a PDF and a zip file (named for your trentusername.zip) containing your Visual Studio project directories.

Your source code MUST have comments explaining what each class and each method is for.