California State University, Sacramento
College of Engineering and Computer Science

Computer Science 130: Data Structures and Algorithm Analysis

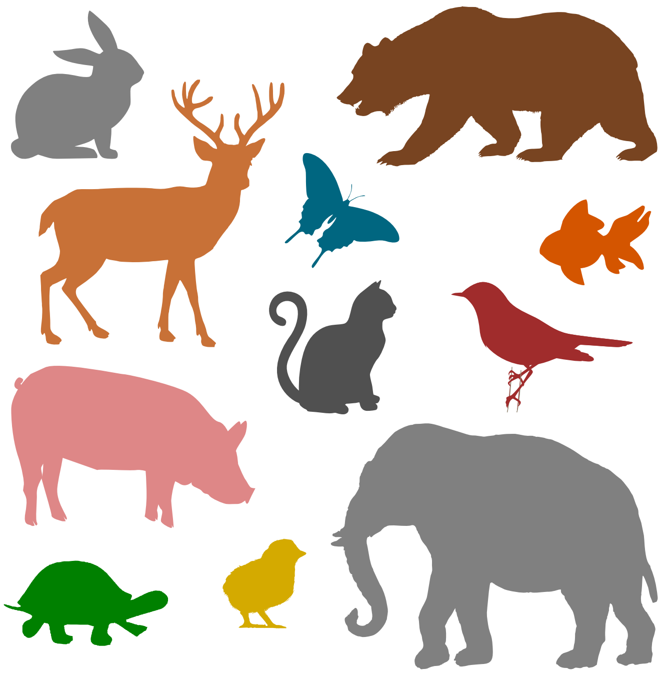
Spring 2026 – Project #2 – Abstract Data Type

Overview

For this semester, we are going to create a basic program that helps organize a zoo.

Zoos can be wonderful places where children and adults can learn to appreciate all the creatures of this world All animals are beautiful in their own ways.

Now that we have a working Animal Class and a nice Linked List Class data structure, we can finally create a useful abstract data type. In particular, you are going to create a new class called Zoo.



Part 1: Double-Checking your Work

For this first assignment, you are going to use the code you developed in Project #1. Before you continue, it is a great idea to double-check your code.

Enumerations

A common mistake in beginner programs, is to make a mistake on enumerations. For example, following is the source code for the Category Enumeration. Note: standard Java naming conventions state that each constant below should be capitalized.

```
public enum Category
{
    BUG,
    BIRD,
    AQUATIC,
    MAMMAL,
    ECTOTHERM,
    OTHER
}
```

If you wrote the constants in Pascalcase (aka Propercase) where you capitalize only the first letter, that's okay. I really don't mind if you violate this particular ad-hoc rule.

Check for Static

It is important to reiterate that you need to check your code for `static`. A static function is associated with the class, but not an instance. Normally, your program has a static function called **`main()`**. This should be the **only** static definition in your program.

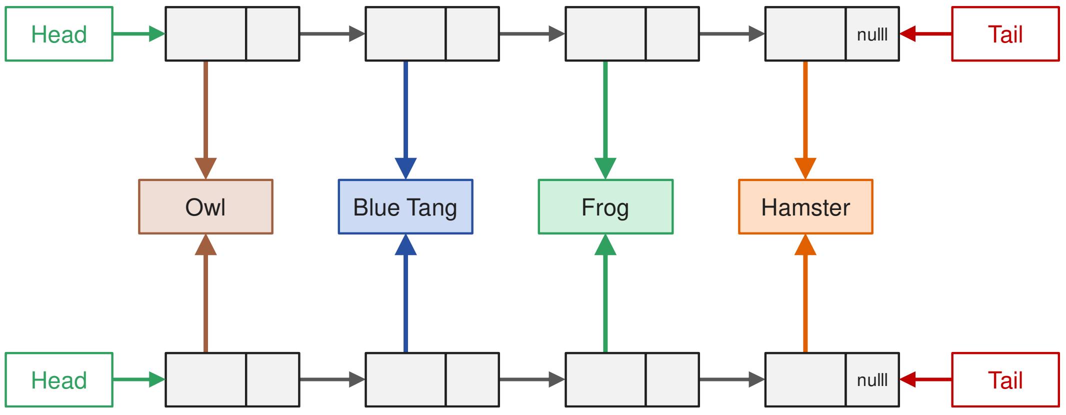
Part 2: Modify the Linked List Class

Before we continue with your new class, you need to make two changes to your linked list class. Don't worry, it's not a huge change, you are just going to add two methods.

You should definitely make sure you did the class correctly at this point, or this next method will not work.

`LinkedList Class – new methods`		
`LinkedList`	**`Duplicate()`**	Returns a new linked list containing the same objects stored in the current linked list.
`LinkedList`	**`Duplicate(Category)`**	Returns a new linked list containing the same objects stored in the current linked list. However, this only returns a linked list containing the specified Category of animals.

The new method will return an exact copy of the current linked list. Don't make copies of the Animal instances, you can simply add them to the new linked list. This is one of those excellent features that you can find in reference languages like Java. You can have multiple objects referencing in the same instance. For example, look at the picture below.



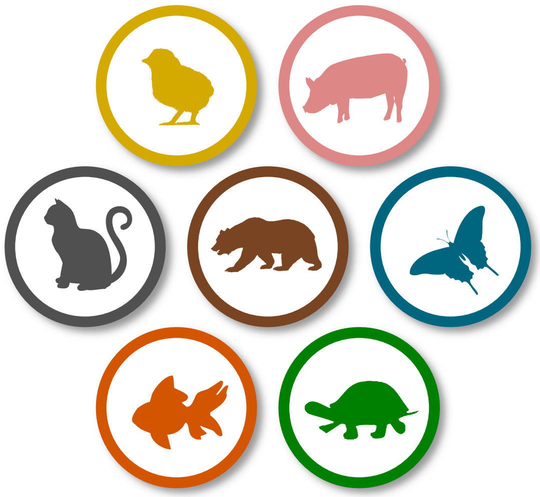
Here we have two linked lists. The nodes have references to the next one in the series. The value field, in each node, references an instance of the Animal Class. In this example, notice that the Owl, Blue Tang (Dora from Finding Nemo), Frog, and Hamster are being referenced by two different nodes.

You should also note that the last nodes, in each linked list, are referenced twice (once by the previous node and then by the Tail variable).

So, the only difference between the two new methods is simple. The method **`Duplicate()`** returns a linked list with **all** the animals. The method **`Duplicate(Category)`** returns a linked list contain only the animals of the specified category. For example, **`Duplicate(Category.AQUATIC)`** will return a list containing only the aquatic animals.

Part 3: Zoo Class

It is time to create our main Zoo Class. Like well-designed abstract data types, the data structure will be hidden from the user. This will allow us to change it later in the semester (hint, hint).



Internally, we are going to store two copies of the Linked List class. Why two copies? Well, one will contain all the animals in our zoo.

The second Linked List will be used as a queue. All the animals need to be fed, and we need to make sure we don't feed an animal twice (or forget to feed them altogether). This will be duplicated from the main list before we start feeding our friends. Why not use the main list? Well, that would make us lose our data! By duplicating the Linked List, we can have a working queue and preserve our original data.

`Zoo Class PRIVATE Members`		
`LinkedList`	**`animals`**	This linked list is used to maintain the main list of animals.
`LinkedList`	**`feedingQueue`**	This is assigned a duplicate of the "animals" list anything feeding starts.

Constructors

The Zoo Class will have only one constructor. You should set **`animals`** to a new instance of the Linked List class.

`Zoo Class Constructors`
`Zoo()`

Methods

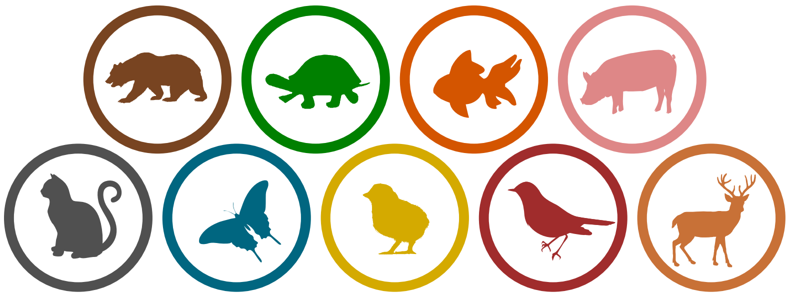
The Linked List data structure will be doing most of the work in our class. As a result, most of the methods below will contain very little code.

`Zoo Class Methods`		
`void`	**`add(Animal)`**	Adds an Animal instance to main list. This is called "**`animals`**" in the field list above.
`string`	**`ToList()`**	Returns the contents of the main animal list. Simply return the result of the linked list's **`ToList()`** method.
`void`	**`startFeeding()`**	Duplicates the main linked list using the **`duplicate()`** method. The result is stored into the **`feedingQueue`** field of this class.
`void`	**`startFeeding(Category)`**	Duplicates the main linked list using the **`duplicate(Category)`** method. The result is stored into the **`feedingQueue`** field of this class.
`Animal`	**`feedNext()`**	Removes the head from the **`feedingQueue`** and returns it to the caller.
`boolean`	**`doneFeeding()`**	Returns true of either the **`feedingQueue`** variable is null or (if it is not null), the result of **`IsEmpty()`**.

Part 4: Testing

You need to test the logic of your Linked List class. In the **`main()`** function of your Tester Class (or whatever you call it – it doesn't matter to me), create at least **ten (10)** instances of the Animal class. Add them to an instance of the Zoo Class.

You can try using **`ToList()`** method. Also, create a loop and feed the animals (outputting the nick name each time). Try it using both StartFeeding methods. Do not read values from a file or the keyboard.



Allowed Programming Languages

You may use any of the following programming languages.

- C#
- C++ (not recommended)
- Java

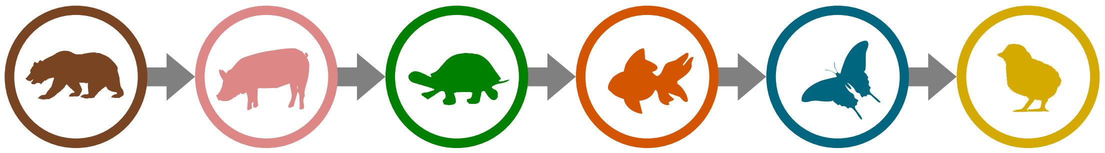
The following **cannot** be used:

- Groovy
- JavaScript
- Kotlin
- Lua
- Nim
- Python
- Ruby
- Scala
- Swift
- TypeScript
- Visual Basic .NET

Requirements

The following are the requirements for this assignment:

- The classes and methods defined above.
- You **must** use your code from Project #1.
- You may use any of the programming languages listed above.
- Create some excellent testing for your class.
- Proper style (see below).
- Do **not** put your `main()` method inside the Linked List class. This will cause major issues going forward.
- This **must** be completely all your code. If you share your solution with another student or re-use code from another class, you will receive a zero.
- Do **not** use any built-in collection classes such as ArrayList. Many programming languages have nice collection classes available. Do not use any of them. **If you use any of these, you will receive a zero. No exceptions. No resubmissions.**



Due Date & Submission

Due **March 20, 2026,** by **11:59 pm.**

E-Mail your source code files to **dcook@csus.edu.** Please zip your source code files into a single .zip or .tar file. Please take note of the following. If you send your files incorrectly, I either won't receive them or won't be able to grade them.

1. Send **ONLY** `.cpp`, `.java`, or `.cs` files. The e-mail server will delete all attachments that have file extensions it deems dangerous. This includes `.jar`, `.exe`, and `.class`.
2. Do **NOT** simply zip an entire folder. The server will delete it.
3. Do **NOT** send a cloud link. I cannot open or grade these. Send an attachment.
4. Do **NOT** send it to canvas. I will not read nor grade Canvas e-mails.

Proper Style

Points will be deducted if your program doesn't adhere to basic programming style guidelines.

1. If programming C++, Java, or C#, I don't care where you put the starting curly bracket. Just be consistent.
2. Indentation must be used.
3. Indentation must be consistent. Three or four spaces works. **Beware of the tab character!** It might not appear correctly on my computer (tabs are inconsistent in size).
4. Proper commenting. Not every line needs a comment, but sections that contain logic often do. Add a comment before every section of code – such as a loop or If Statement. Any complex idea, such as setting a link, must have a comment.

Well-formatted code

The following code is well formatted and commented.

```
void foo()
{
    int x;

    //Print off the list
    x = 0;
    while (x < this.count)
    {
       items.Add(this[x]);  //Add the item to the temporary list
       x++;
    }
}
```

Poorly-formatted code

The following code is poorly formatted and documented.

```
void foo()
{
    int x;

x = 0;
while (x < this.count)   {
     items.Add(this[x]);  //Call add on items.
       x++;
}
}
```