California State University, Sacramento
College of Engineering and Computer Science

Computer Science 130: Data Structures and Algorithm Analysis

Spring 2026 – Project #3 – Binary Search Tree

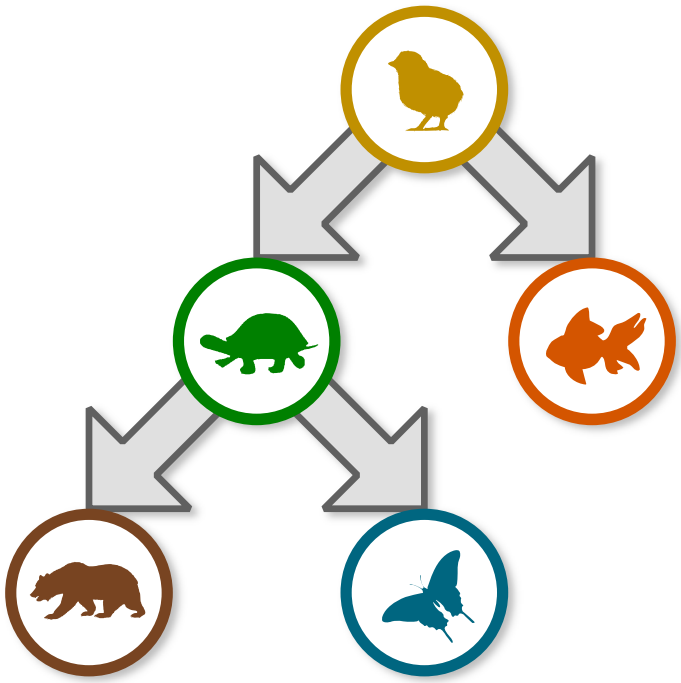
Overview

Now that we have a wonderful Zoo Class. We can make it more advanced.

For this assignment, you are going to create a very basic Binary Search Tree (BST). Once loaded, your BST can print the tree as an ASCII-art tree.

You don't have to balance the tree. In fact, don't even try it yet. Though, to be honest, a red-black tree (as long as you do the left-child rotate trick when adding), isn't all that hard.

Note: You are not going to throw away your Linked List Class. The Zoo Class will use each!



Part 1: Double-Checking your Work

For this first assignment, you are going to use the code you developed in Project #2. Before you continue, it is a great idea to double-check your code.

Zoo Constructor

Make sure that your Constructor for the Zoo Class contains **no** parameters.

Some students created a version that had a linked list as a parameter. The goal of an Abstract Data Type is to hide the data structure from the client. As you will see in this assignment, this is a good programming practice.

Part 2: Tree Node Class

It was important on your previous assignments that the Linked List Node Class was hidden (private) inside the Linked List Class. This is because the node was used internally, and, outside the class, the class should not be visible.

You are going to create a second node class. Yes! There will be two different node classes, but there won't be a problem if you declare them properly. The Linked List Node was declared – as private – in the Linked List. The Tree Node will be declared– as private – inside the Binary Search Tree Class (see below).

In recursively defined structures, like trees, **all** the coding (and complexity) is found in the recursive structure itself. So, for this assignment, all the logic will be found in the Tree Node Class. The BinarySearchTree Class will merely start recursion on the root.

`private class Node`		
	`Node(Animal)`	Constructor.
`Node`	`left`	
`Node`	`right`	
`Animal`	`value`	
`void`	`add(Animal)`	Please see below.
`String`	`toFormattedText(int indent)`	Please see below.
`void`	`toLinkedList(LinkedList)`	Please see below
`void`	`toLinkedList(LinkedList,Catagory)`	Please see below.

Pseudocode

Your Node class should be as follows:

```
class Node
    public Animal value
    public Node left
    public Node right

    ← All your methods go here
end class
```

Method `add(Animal)`

Adds the Animal to the correct position in the BST **using the ZooID as the key value**. If the key already exists, do nothing.

To add an Animal, you will write a recursive method that will either recurse to the left or right – depending on the key (Zoo ID). Whenever it can no longer recurse left or right, a new node is simply added.

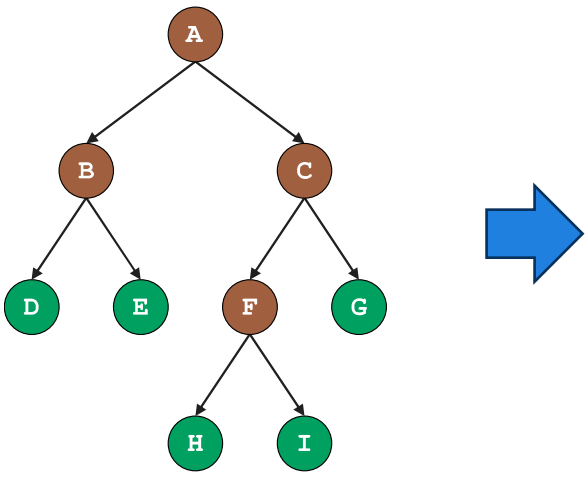
```
method add(Animal insert)
    if insert.ZooID < this node's Zoo ID then
        if left is null
            create a new left child for this node.
        else
            left.add(insert)                          ...Recursion
        end if
    end if

    if insert.ZooID > this node's Zoo ID then
        if right is null
            create a right child for this node.
        else
            right.add(insert)                         ...Recursion
        end if
    end if
end method
```

Method `toFormattedText(int indent)`

The method will recursively generate a string that represents the structure of the tree. One node will be displayed per line. Notice that the current node "this" is concatenated before the left and right recursive calls. This is an example of a **preorder** depth-first traversal.

```
method toFormattedText(int indent) returns a string
    Declare String result

    result += spaces for indent (2 or 3 times the indent)
    result += "+---- "
    result += this.value.ZooID, this.value.nickname(), and a newline

    if this.left isn't null
        result += this.left.toFormattedText(indent + 1)       ...Recursion
    end if

    if this.right isn't null
        result += this.right.toFormattedText(indent + 1)      ...Recursion
    end if

    return result
end method
```

For example, the following tree will be represented in ASCII. Note this is how folders are typically displayed on your computer.




```
+---- A
    +---- B
        +---- D
        +---- E
    +---- C
        +---- F
            +---- H
            +---- I
        +---- G
```

For example, let's say we add the following animals in the specified order. I've listed the ZooID and the nickname.

```
1. Herky the Hornet, 1947
2. Ninji the Cat, 2011
3. Anchor the Chicken, 1964
4. Sutter the Eagle, 1839
5. Rian the Mosquito, 2017
6. Scruffy the Ringtail, 50
7. Monarch the Bear, 1911
8. Sammy the Sturgeon, 61
9. Cuffy the Bear, 46
```

The following tree would be produced. **Your version is will look different. Don't just copy my example. Also, you were welcome to change the format of each line.**

```
+---- 1947: Herky the Hornet
    +---- 1839: Sutter the Eagle
        +---- 50: Scruffy the Ringtail
            +---- 46: Cuffy the Bear
            +---- 61: Sammy the Sturgeon
        +---- 1911: Monarch the Bear
    +---- 2011: Ninji the Cat
        +---- 1964: Anchor the Chicken
        +---- 2017: Rian the Mosquito
```

Binary Search Trees are extremely sensitive to the order that data is fed into them. In fact, once node is added, its position in the tree will never change. In the example below, I've added the animals, but I have switched the position of the first two.

```
1. Ninji the Cat, 2011
2. Herky the Hornet, 1947
3. Anchor the Chicken, 1964
4. Sutter the Eagle, 1839
5. Rian the Mosquito, 2017
6. Scruffy the Ringtail, 50
7. Monarch the Bear, 1911
8. Sammy the Sturgeon, 61
9. Cuffy the Bear, 46
```

The following tree would be produced. **Your version is will look different. Don't just copy my example. Also, you were welcome to change the format of each line.**

```
+---- 2011: Ninji the Cat
    +---- 1947: Herky the Hornet
        +---- 1839: Sutter the Eagle
            +---- 50: Scruffy the Ringtail
                +---- 46: Cuffy the Bear
                +---- 61: Sammy the Sturgeon
            +---- 1911: Monarch the Bear
        +---- 1964: Anchor the Chicken
    +---- 2017: Rian the Mosquito
```

Method `toLinkedList(LinkedList)`

To convert the tree to a Linked List instance, we will use an interesting bit of code. Notice that the list, we are creating, is passed to each nodes using recursion.

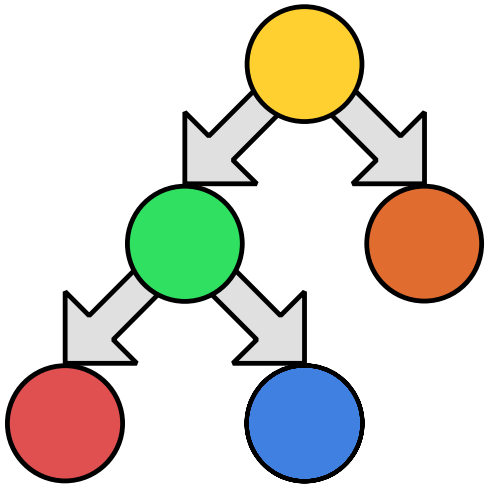
The current node adds itself after it first recurses left and right. This is a good example of post-order traversal. However, where we "visit" the node can be changed. You might want to experiment with moving it

```
method toLinkedList(LinkedList list)
    if this.left isn't null
        this.left.toLinkedList(list)       ...Recursion
    end if

    if this.right isn't null
        this.right.toLinkedList(list)      ...Recursion
    end if

    list.addTail(this.value)               ...Add the current node's Animal.
end method
```

Note: We *could* have implemented this logic by having each node return a linked list. Then, these could have been combining together. Whlie, this approach would have worked, it is far, far, less efficient.

Method `toLinkedList(LinkedList, Category)`

The method is nearly identical to the example above. The only difference is there is an if-statement that determines if we add the current node's animal. This if-statement doesn't affect the recursion code.



Part 3: BinarySearchTree Class

For this project, you are also to create a wrapper BinarySearchTree Class. In reality, this class doesn't do that much. The class simply starts recursion of the root itself. Naturally, there is some logic needed to handle a null root, but that is just a few basic if-statements.

public class BinarySearchTree		
	BinarySearchTree()	Constructor.
Node	**root**	Private
void	**add(Animal)**	This will start recursion from the root node if it exists. If not, create a new root node.
String	**toFormattedText()**	Returns a string of the tree's structure. This will start recursion from the root node using an indent of **0**.
LinkedList	**toLinkedList()**	Calls the root node with an instance of a LinkedList (declared in this method). This will start recursion from the root node. This linked list is returned.
LinkedList	**toLinkedList(Category)**	Calls the root with an instance of a LinkedList (declared in this method). This will start recursion from the root node. This linked list is returned.

Method `add(Animal)`

```
public method add(Animal insert)
    if this.root is null
        this.root = new Node
    else
        this.root.add(insert)      ...Start recursion
    end if
end method
```

Method `toLinkedList()`

```
method toLinkedList() returns a LinkedList
    Declare new LinkedList list

    if this.root isn't null
        this.root.toLinkedList(list)          ...Starts Recursion
    end if

    return list
end method
```

Part 4: Modifying the Zoo Class

It's time to take advantage of your Binary Search Tree class to enhance the Zoo class. We are going to replace the Animals linked list with a tree.

This is one of the great advantages of a properly written abstract data type. We are changing the internal data structure, and those, using the class, will not know. Their programs will continue to work properly.

We aren't going to modify the feeding queue. This is a linked list – and that makes a very natural queue. That will allow the feeding logic to continue to work as before. This is why the BinarySearchTree Class can convert itself into a linked list.

Zoo Class PRIVATE Members		
BinarySearchTree	**animals**	Update this to a BinarySearchTree
LinkedList	**feedingQueue**	This is assigned a to the result of the **`toLinkedList()`** method.

Updated Methods

Updated Zoo Class Methods		
`String`	`toList()`	Returns `toFormattedText()` off the animals Binary Search Tree.
`void`	`add(Animal)`	This will now call the `add()` method on the BinarySearchTree. In Project #2, you called `AddTail()` on the linked list
`void`	`startFeeding()`	Sets the Feeding Queue to the result of the Animals `toLinkedList()` method
`void`	`startFeeding(Category)`	Sets the Feeding Queue to the result of the Animals `toLinkedList(Category)` method

Part 5: Testing

Good Test Data

You need to test the logic of your BinarySearchTree class. In the `main()` function of your Tester Class (or whatever you call it – it doesn't matter to me), create at least **twenty (20)** instances of the Animal class. Yes, that is an increase from last time, but we need enough to test the tree properly.

Note: In the previous two projects, the ZooID value wasn't really all that important. However, in this lab, you need to create some nice random numbers. Otherwise, your tree will be, in form, a linked list.

Testing the BinarySearchTree

Before you attempt to change the Zoo Class, it's a good idea to test the BinarySearchTree separately. Create an instance and add all twenty animals. Then try to see if you can convert print its formatted text and if you can convert it to a linked list.

Testing the Zoo Class.

Once you are are confident that the BinarySearchTree class is working, it is time to test the Zoo Class. Because we didn't modify the interface (public methods) of the Zoo Class, you can actually reuse the feeding logic before Project #2. In other words, create a bunch of animals. Add them to the Zoo, print the tree, and then feed them like before.

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 and Project #2.
- You may use any of the programming languages listed above.
- Create some excellent testing for your class.
- Proper style (see below).
- 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 **April 17, 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++;
}
}
```