California State University, Sacramento
College of Engineering and Computer Science

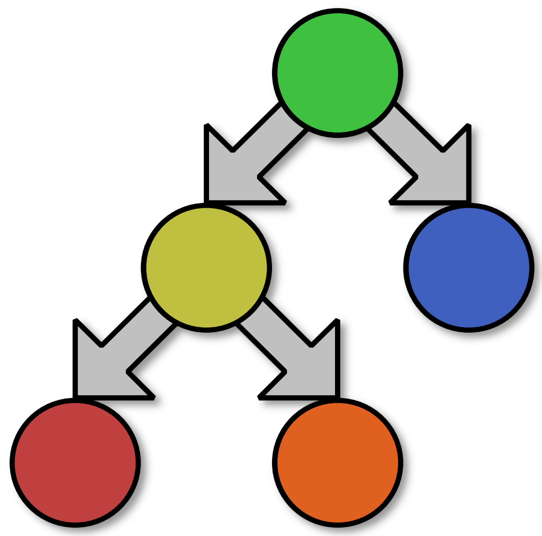
**Computer Science 130: Data Structures and Algorithm Analysis**

**Spring 2024 – Project 3 – Binary Search Tree**

## Overview

For this assignment, you are to create a very basic Binary Search Tree (BST). It will read data from a file – in fact, the same exact files that you used in Project #2. Once loaded, your BST can print the tree as both a sorted list and 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.



## Part 1: Node Class

### Interface

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 Node Class.

The Tree Class, which is defined in the handout below, merely starts recursion on the root.

| **public class Node** | | |
|---|---|---|
| | **Node(int key, string value)** | Constructor. |
| **Node** | **left** | |
| **Node** | **right** | |
| **int** | **key** | The key to find the value. |
| **string** | **value** | The value that the node contains. |
| **string** | **toTree(String label, int indent)** | Returns a string of the tree's structure. *Please see below.* |
| **String** | **toSortedList()** | Returns a string sorted by the key. *Please see below.* |
| **void** | **add(int key, string value)** | Adds the key to the correct position in the BST. If the key already exists, do nothing. |
| **string** | **find(int key)** | Finds a node with the key and returns its value. If the node is not found, you can return an empty string. |

## Pseudocode

Your Node class should be as follows:

```
class Node
    public int key
    public String value
    public Node left
    public Node right

    ... All your methods go here
end class
```

## Adding to the Tree

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

```
method add (int key, String value)
    if key < this node's key then
        if left is null
            create a new left child for this node.
        else
            left.add(key, value)
        end if
    end if

    if key > this node's key then
        if right is null
            create a right child for this node.
        else
            right.add(key, value)
        end if
    end if
end method
```

## Creating the ASCII Tree

The method will recursively generate a string for the structure of the tree. One node will be displayed per line. In the examples below, I'm using spaces followed by label that identifies each as either "L" for left and "R" for right. You can use spaces, dashes, etc… You can use any size of indentation you like (2, 3, etc…).

Notice that the current node "this" is concatenated before the left and right recursive calls. This is an example of an **preorder** depth-first traversal.

```
Function toTree(String label, int indent) returns a string
    Declare String result

    result += spaces for indent (2 or 3 times the indent)
    result += label + ": "
    result += this.key, this.value, and a newline

    if left isn't null
        result += left.toTree("L", indent + 1)
    end if

    if right isn't null
        result += right.toTree("R", indent + 1)
    end if

    return result
End Function
```

The following could be produced by this algorithm. Your version is can look a bit different if you wish, but you must indent your lines and (somehow) indicate which is the left and right branch. Notice that this is the point of the label field.

```
-: (1947) Sacramento State
   L: (1869) Transcontinental Railroad
      L: (1848) Gold Rush Begins
         L: (1846) Bear Flag Revolt
         R: (1850) California joins the U.S.
      R: (1911) California Flag officially adopted
   R: (1976) Apple Founded
      L: (1968) Intel Founded
      R: (2003) Tesla Founded
```

## Creating the Sorting list

The method will recursively generate a string and return the list is sorted order. It is quite easy to do.

Notice that the current node "this" is concatenated after the left recursive call and before right recursive call. This is an example of an **inorder** depth-first traversal.

```
Function toSortedList() returns a string
    Declare String result

    if left isn't null
        result += left.toSortedList()
    end if

    result += this.key, this.value, and a comma

    if right isn't null
        result += right.toSortedList()
    end if

    return result
End Function
```

# Part 2: BinarySearchTree Class

## Interface

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 an null root, but that is just a few basic if-statements.

| **public class BinarySearchTree** | | |
|---|---|---|
| | **BinarySearchTree()** | Constructor. |
| **Node** | **root** | Private |
| **string** | **about()** | Returns text about you – the author of this class. |
| **string** | **toTree()** | Returns a string of the tree's structure.<br>It will start recursion from the root node with indent 0 and a label of "Root". |
| **String** | **toSortedList()** | Returns a string sorted by the key. Please see below.<br>It will start recursion from the root node. |
| **void** | **add(int key, String value)** | Adds the key to the correct position in the BST. If the key already exists, do nothing. |
| **string** | **find(int key)** | Finds a node with the key and returns the value. If the node is not found, you can return an empty string. |

## Pseudocode

```
class BinarySearchTree
    private Node root
    ... All your methods go here.
end class
```

# Part 3: Input File Format

A number of test files will be provided to you for testing your code. The format is designed to be easy to read in multiple programming languages. You need to use the classes, built in your programming language, to read the source files.

## File Format

The first line of the data contains the total digits in the key. You might want to save this value – I can be used to separate the key from the value (using the substring function found in most programming languages).

```
Key 1
Value 1
Key 2
Value 2
...
Key n
Value n
0
END
```

The following is one of the most basic test files on the website.

**File: california.txt**

```
1947
Sacramento State
1976
Apple Founded
1869
Transcontinental Railroad
2003
Tesla Founded
1848
Gold Rush Begins
1911
California Flag officially adopted
1850
California joins the U.S.
1968
Intel Founded
0
END
```

## Part 4: Testing

Once you have finished your code, you need to test it using some good test data. Now you can see why you wrote the ToTree method. It is vital to verifying if your methods are working correctly.

For example, if the following file is added to the Binary Search Tree.

| File: halloween calories.txt |
|---|
| 73<br>M&M's Fun size<br>60<br>Tootsie Pop<br>40<br>Starburst Fun Size<br>70<br>Kit Kat Snack Size<br>30<br>Laffy Taffy<br>80<br>Snickers Fun Size<br>50<br>Nerds Mini Box<br>77<br>Hershey's Milk Chocolate Fun Size<br>82<br>Almond Joy Snack Size<br>0<br>END |

It will result in the following tree.

```
-: (73) M&M's fun size
   L: (60) Tootsie Pop
      L: (40) Starburst Fun Size
         L: (30) Laffy Taffy
         R: (50) Nerds Mini Box
      R: (70) Kit Kat Snack Size
   R: (80) Snickers Fun Size
      L: (77) Hershey's Milk Chocolate Fun Size
      R: (82) Almond Joy Snack Size
```

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

**File: halloween calories 2.txt**

```
60
Tootsie Pop
73
M&M's Fun size
40
Starburst Fun Size
70
Kit Kat Snack Size
30
Laffy Taffy
80
Snickers Fun Size
50
Nerds Mini Box
77
Hershey's Milk Chocolate Fun Size
82
Almond Joy Snack Size
0
END
```

Observe that, this minor change of order, has had a profound impact on the structure of the tree. The first key added will always become the root. And it will remain the root.

```
-: (60) Tootsie Pop
   L: (40) Starburst Fun Size
      L: (30) Laffy Taffy
      R: (50) Nerds Mini Box
   R: (73) M&M's fun size
      L: (70) Kit Kat Snack Size
      R: (80) Snickers Fun Size
         L: (77) Hershey's Milk Chocolate Fun Size
         R: (82) Almond Joy Snack Size
```

## Assignment Rules

- 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.
- You **must** use recursion in the Node class. The BinarySearchTree only starts recursion on the root.
- You may use any programming language you are comfortable with. I strongly recommend not using C (C++, Java, C#, Visual Basic are all good choices).

## Requirements

| | |
|---|---|
| 1 | Correct use of recursion – it must happen in the Node class. |
| 2 | Correct interfaces |
| 3 | Correct toTree method. It must print something to denote left and right branches. |
| 4 | Correct toSortedList method. It must use recursion. |
| 5 | Correct add method. It must use recursion. |
| 6 | Correct find method. It must use recursion. |
| 7 | Proper Style |
| 8 | Reading from the test file. |

## Due Date

Due **April 14, 2024** by 11:59 pm.

Given you did a good job on the Tree Evaluator, then this shouldn't be a difficult assignment. **Do not send it to canvas.** E-Mail the following to dcook@csus.edu:

- The source code.
- The main program that runs the tests.

**The e-mail server will delete all attachments that have file extensions it deems dangerous. This includes .py, .exe, and many more.**

**So, please send a ZIP File containing all your files.**

# Proper Style

## Well-formatted code

Points will be deducted if your program doesn't adhere to basic programming style guidelines. The requirements are below:

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. Please see below:

5. You must adhere to one-way-in-one-way-out code. It is considered poor form to use return outside a final If-Statement or Switch. **Any project using a "dead return", as pictured below, will lose 50%.**

```
return;
```

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 poor formatted and documented.

```
void foo()
{
    int x;

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