Part I

Labs

Lab 1

Data Structures II

Lab Objective: *Implement tree data structures and understand their relative strengths and weaknesses.*

Recursion

Recursion is an important problem solving technique in computer programming. A recursive function is one that calls itself. When the function is executed, it continues calling itself until it reaches a specified base case. Then the function exits without calling itself again, and each previous function call is resolved. As a simple example, suppose we want to recursively sum all positive integers from 1 to some integer n.

```
def recursive_sum(n):
    """Calculate the sum of all positive integers in [1, n] recursively."""

    # Typically the base case comes first. There are no positive integers less
    # than 1, so if 'n' is 1 we stop the recursion and return 1 (since the sum of
    # all integers in [1, 1] is 1).
    if n == 1:
        return 1

    # If the base case hasn't been reached, the function recurses by calling
    # itself on the next smallest integer and adding 'n'.
    else:
        return n + recursive_sum(n-1)
```

The computer calculates `recursive_sum(5)` with a sequence of function calls.

```
# To find the recursive_sum(5), we need to calculate recursive_sum(4).
# But to find recursive_sum(4), we need to calculate recursive_sum(3).
# This continues until the base case is reached.

recursive_sum(5)        # return 5 + recursive_sum(4)
    recursive_sum(4)        # return 4 + recursive_sum(3)
        recursive_sum(3)        # return 3 + recursive_sum(2)
            recursive_sum(2)        # return 2 + recursive_sum(1)
                recursive_sum(1)        # Base case: return 1.
```

Now that we've reached a base case, we can unwind the recursion. Reading from bottom to top, we substitute the values that result from each function call.

```
recursive_sum(5)        # 5 + 10 = 15
    recursive_sum(4)        # 4 + 6 = 10
        recursive_sum(3)        # 3 + 3 = 6
            recursive_sum(2)        # 2 + 1 = 3
                recursive_sum(1)        # Base case: return 1.
```

So `recursive_sum(5)` returns 15 (which is correct, since $1 + 2 + 3 + 4 + 5 = 15$). Many problems that can be solved by iterative methods can also be solved (often more efficiently) with a recursive approach. Compare, for example, the following two methods for calculating the n^{th} Fibonacci number.

```
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    fibonacci = list()      # Initialize an empty list.
    fibonacci.append(0)     # append 0 (the 0th Fibonacci number).
    fibonacci.append(1)     # append 1 (the 1st Fibonacci number).
    for i in range(1, n):
        # Starting at the third entry, calculate the next number
        # by adding the last two entries in the list.
        fibonacci.append(fibonacci[-1] + fibonacci[-2])
    # When the entire list has been loaded, return the nth entry.
    return fibonacci[n]

def recursive_fib(n):
    """Calculate the nth Fibonacci number recursively."""
    # The base cases are the first two Fibonacci numbers.
    if n == 0:              # Base case 1: the 0th Fibonacci number is 0.
        return 0
    elif n == 1:            # Base case 2: the 1st Fibonacci number is 1.
        return 1
    # If we haven't reached a base case, the function recurses by calling
    # itself on the previous two Fibonacci numbers.
    else:
        return recursive_fib(n-1) + recursive_fib(n-2)
```

This time, the sequence of function calls is slightly more complicated because `recursive_fib` calls itself twice until a base case is reached.

```
recursive_fib(5)        # The original call makes two additional calls:
    recursive_fib(4)        # this one...
        recursive_fib(3)
            recursive_fib(2)
                recursive_fib(1)        # Base case 2: return 1
                recursive_fib(0)        # Base case 1: return 0
            recursive_fib(1)        # Base case 2: return 1
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
    recursive_fib(3)        # ...and this one.
        recursive_fib(2)
            recursive_fib(1)        # Base case 2: return 1
            recursive_fib(0)        # Base case 1: return 0
        recursive_fib(1)        # Base case 2: return 1
```

The sum of all of the base case results, from top to bottom, is $1+0+1+1+0+1+0+1=5$, so `recursive_fib(5)` returns 5 (correctly). The key to recursion is understanding the base cases correctly and making correct recursive calls.

Problem 1. Rewrite the following iterative function for finding data in a linked list using recursion. Use a `LinkedList` object from the previous lab to test the function.

```
# solutions.py

def iterative_search(linkedlist, data):
    current = linkedlist.head
    while current is not None:
        if current.data == data:
            return current
        current = current.next
    raise ValueError(str(data) + " is not in the list.")
```

(Hint: define an inner function to perform the actual recursion)

Warning

It is not always better to rewrite an iterative method recursively. In Python, a function may only call itself 999 times. On the 1000^{th} call, a `RuntimeError` is raised to prevent a stack overflow. Whether or not recursion is appropriate depends on the problem to be solved and the algorithm to solve it.

Trees

A *tree* data structure is a specialized linked list. Trees are more difficult to build than standard linked lists, but they are almost always more efficient. While the computational complexity of finding a node in a linked list is $O(n)$, a well-built, balanced tree will find a node with a complexity of $O(\log n)$. Some types of trees can be constructed quickly but take longer to retrieve data, while others take more time to build and less time to retrieve data.

The first node in a tree is called the *root*. The root node points to other nodes, called children. Each child node in turn points to its children. This continues on each branch until its end is reached. A node with no children is called a *leaf node*.

Mathematically, a tree is a directed graph with no cycles. Therefore a linked lists qualifies as a tree, albeit a boring one. The head node is the root node, and it has one child node. That child node also has one child node, which in turn has one child. This continues until the end of the list, with the last node as the only leaf node.

Other kinds of trees may be more complicated. See Figure **??**.

Figure 1.1: Examples of trees, some of which are binary search trees.

Binary Search Trees

A *binary search tree* (BST) data structure is a tree that allows each node to have up to two children, called `left` and `right`. The left child of a node contains data that is less than its parent node's data. The right child's data is greater. Conceptually, each level of a BST partitions the dataset into two halves. For any given node, each node in the left subtree contains data that is less than the data in the top node, and each node in the right subtree contains data that is greater than the original node's data.

The far right tree in Figure 5.1 is an example of a binary search tree. In practice, binary search tree nodes have attributes that keep track of their data, children, and (in doubly-linked trees) their parent.

```
# BST.py

class BSTNode(object):
    """A Node class for Binary Search Trees. Contains some data, a
    reference to the parent node, and references to two child nodes.
    """
    def __init__(self, data):
        """Construct a new node and set the data attribute. The other
        attributes will be set when the node is added to a tree.
        """
        self.data = data
        self.prev = None        # A reference to this node's parent node.
        self.left = None        # This node's data will be less than self.data
        self.right = None       # This node's data will be greater than self.data

    def __str__(self):
        """String representation: the data contained in the node."""
        return str(self.data)
```

The actual binary search tree class has an attribute pointing to its root.

```
# BST.py

class BST(object):
    """Binary Search Tree data structure class.
    The first node is referenced to by 'root'.
    """
    def __init__(self):
        """Initialize the root attribute."""
        self.root = None
```

Finding in a Binary Search Tree

Many tree algorithms are best understood and implemented using recursion. For instance, finding a node in a binary search tree can be done recursively. Starting at the root, we check if the data we are looking for matches the current node. If it does not, then if the data is less than the current node's data we search again

Figure 1.2: Examples of finding in a BST, including the conceptual partitioning of the number line.

Figure 1.3: Examples of inserting to a BST.

on the left child. If the data is greater, we search on the right child. This process continues until the data is found or, if the data is not in the tree, an empty child is searched. See Figure 5.2 for an example. Carefully read the following code; similar techniques will be used for subsequent methods.

```
# BST.py

def find(self, data):
    """Return the node containing 'data'. If there is no such node in the
    tree, raise a ValueError with error message "<data> is not in the tree."
    """
    # First, check to see if the tree is empty.
    if self.root is None:
        raise ValueError(str(data) + " is not in the tree.")

    # Define a recursive function to traverse the tree.
    def _step(current, item):
        """Recursively step through the tree until the node containing
        'item' is found. If there is no such node, raise a Value Error.
        """
        if current is None:                     # Base case 1: dead end.
            raise ValueError(str(data) + " is not in the tree.")
        if item == current.data:                # Base case 2: the data matches.
            return current
        if item < current.data:                 # Step to the left
            return _step(current.left,item)
        else:                                   # Step to the right
            return _step(current.right,item)

    # Start the recursion on the root of the tree.
    return _step(self.root, data)
```

Inserting to a Binary Search Tree

To insert new data into a binary search tree, a leaf node is added at the correct location. First, we find the node that should be the parent of the new node. We find the parent recursively, using a similar approach to the `find` method. Once the correct parent is found, the new node is added as the left or right child of the parent. See Figure 5.3 for an example.

Problem 2. Implement the `insert` method in the `BST` class. To accomplish this, write a recursive `_find_parent` method within `insert`. Find the correct parent, then determine whether the new node will be its left or right child. Then double-link the parent and the new child. Be sure to consider the

Figure 1.4: Examples of removing a leaf node.

Figure 1.5: Examples of removing a leaf node with one child. Include a bad example (losing a subtree) and a good example (proper deletion).

special case of inserting to an empty tree. To test your tree, use (but do not modify) the provided `BST.__str__` method.

Do not allow for duplicates in the tree: if the user executes `insert(x)` and there is already a node in the tree containing `x`, raise a ValueError.

Removing from a Binary Search Tree

Deleting nodes from a binary search tree is more difficult than searching and inserting. Insertion always creates a new leaf node, but removal may delete any kind of node. This leads to several different cases to consider.

Removing a leaf node

In Python, an object is automatically deleted if there are no references to it. Call the node to be removed the *target node*, and suppose it has no children. To remove the target, find the target's parent, then delete the parent's reference to the target. Then there are no references to the target, so the target node is deleted. Since the target is a leaf node, removing it does not affect the rest of the tree structure. See Figure 5.4.

Removing a node with one child

If the target node has one or more children, we must be careful not to delete the children when the target is removed. Simply removing the target as if it were a leaf node would delete the entire subtree originating from the target.

To avoid deleting all of the target's descendents, we point the target's parent to an appropriate successor. If the target has only one child, then that child is the successor. Connect the target's parent to the successor, and double-link by setting the successor's parent to be the target node's parent. Then, the target has no references pointing to it, so it is deleted. The target's successor, however, is pointed to by the target's parent, and so it remains in the tree. See Figure 5.5.

Removing a node with two children

Removal is more complicated if the target node has two children. To delete this kind of node, first we find it's immediate in-order predecessor. The predecessor is the node with the largest data that is smaller than the target's data. It may be found by moving to the left child of the target (so that it's value is less than the target's value), and then to the right for as long as possible (so that it has the largest such

Figure 1.6: Examples of removing a leaf node with two children.

value). Note that because of how the predecessor is chosen, any predecessor can only have at most one child.

Once the predecessor is found, the target and its predecessor must switch places in the graph, and then the target must be removed. This can be done by simply switching the data values for the target and its predecessor. Then the node with the target data has at most one child, and may be deleted accordingly. If the predecessor was chosen appropriately, then the binary search tree structure and ordering will be maintained once the deletion is finished.

The easiest way to implement this is to use recursion. First, because the predecessor has at most one child, we may recursivley remove the predecessor node by calling `remove` on the predecessor's data. Then set the data stored in the target node as the predecessor's data. See Figure 5.6.

Removing the root node

In each of the above cases, we must also consider the subcase where the target is the root node. If the root has no children, resetting the root or calling the constructor will do. If the root has one child, that child becomes the new root of the tree. If the root has two children, the successor becomes the new root of the tree.

Problem 3. Implement the `remove` method in the `BST` class. If the tree is empty, or if the target node is not in the tree, raise a `ValueError`.

Make sure to cover all possible cases:

1. The tree is empty (`ValueError`)
2. The target is not in the tree (`ValueError`)
3. The target is the root node:
   (a) the root is a leaf node, hence the only node in the tree
   (b) the root has one child
   (c) the root has two children
4. The target is in the tree but is not the root:
   (a) the target is a leaf node
   (b) the target has one child
   (c) the target has two children

Test your solution thoroughly with each case.

(Hint: use `find` wherever appropriate.)

Figure 1.7: Example of an AVL rebalancing.

AVL Trees

Binary search trees are a good way of organizing data so that it is quickly accessible. However, pathologies may arise when certain data sets are stored using a basic binary serch tree. This is best demonstrated by inserting ordered data into a binary search tree. Since the data is already ordered, each node will only have one child, and we essentially end up with a linked list.

```
# Adding ordered integers sequentially destroys the efficiency of a BST.
>>> unbalanced_tree = BST()
>>> for i in xrange(10):
...     unbalanced_tree.insert(i)
...
# The tree is perfectly flat, so it loses its search efficiency.
>>> print(unbalanced_tree)
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
```

Problems also arise when one branch of the tree becomes much longer than the others, leading to longer search times.

An AVL tree is a tree that prevents any one branch from getting longer than the others. It accomplishes this by periodically "balancing" the branches as nodes are added. See Figure 5.7 The AVL's balancing algorithm is beyond the scope of this text, but details and exercises on the algorithm can be found in Chapter 2 of the Volume II text.

```
>>> balanced_tree = AVL()
>>> for i in xrange(10):
...     balanced_tree.insert(i)
...
# The AVL tree is balanced, so it retains (and optimizes) its search efficiency.
>>> print(balanced_tree)
[3]
[1, 7]
[0, 2, 5, 8]
[4, 6, 9]
```

Problem 4. Compare the speed of building and searching the different data structures we have implemented so far. Visualize the results by creating a plot with two subplots: one for build times, and one for search times. Repeat

the following for n varying from 500 to 5000 at intervals of 500:

Use the `create_word_list` function in the provided `WordList` module to generate a list of n randomized words from the file `English.txt`. Time (separately) how long it takes to load a `LinkedList`, a `BST`, and an `AVL` with the data set. Use `add` to load the `LinkedList` and `insert` to load the trees. Then choose 5 random words from the data set, and time how long it takes to find each word in each object. Use the `iterative_search` function from problem 1 to search the `LinkedList` and `find` to search the trees. Calculate the average search time for each object.

In the first subplot, plot the number of words in each data set against the time it took to build each object. In the second subplot, plot the number of words in each data set against the average time it took to search each object. Your plot should look similar to Figure **??**.