2 Binary Search Trees

Lab Objective: *A tree is link-based data structure where each node may refer to more than one other node. This structure makes trees more useful and efficient than regular linked lists in many applications. Many trees are constructed recursively, so we begin with an overview of recursion. We then implement a recursively structured doubly linked binary search tree (BST). Finally, we compare the standard linked list, our BST, and an AVL tree to illustrate the relative strengths and weaknesses of each data structure.*

Recursion

A *recursive* function is one that calls itself. When the function is executed, it continues calling itself until reaching a *base case* where the value of the function is known. The function then exits without calling itself again, and each previous function call is resolved. The idea is to solve large problems by first solving smaller problems, then combining their results.

As a simple example, consider the function $f : \mathbb{N} \to \mathbb{N}$ that sums all positive integers from 1 to some integer n.

$$f(n) = \sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + f(n-1)$$

Since $f(n-1)$ appears in the formula for $f(n)$, f can be implemented recursively. Calculating $f(n)$ requires the value of $f(n-1)$, which requires $f(n-2)$, and so on. The base case is $f(1) = 1$, at which point the recursion halts and unwinds. For example, $f(4)$ is calculated as follows.

$$\begin{aligned}
f(4) &= 4 + f(3) \\
&= 4 + (3 + f(2)) \\
&= 4 + (3 + (2 + f(1))) \\
&= 4 + (3 + (2 + 1)) \\
&= 4 + (3 + 3) \\
&= 4 + 6 \\
&= 10
\end{aligned}$$

The implementation accounts separately for the base case and the recursive case.

```python
def recursive_sum(n):
    """Calculate the sum of all positive integers in [1, n] recursively."""
    if n <= 1:          # Base case: f(1) = 1.
        return 1
    else:               # Recursive case: f(n) = n + f(n-1).
        return n + recursive_sum(n-1)
```

Many problems that can be solved iteratively can also be solved with a recursive approach. Consider the function $g : \mathbb{N} \to \mathbb{N}$ that calculates the nth Fibonacci number.

$$g(n) = g(n-1) + g(n-2), \quad g(0) = 0, \quad g(1) = 1.$$

This function is doubly recursive since $g(n)$ calls itself twice, and there are two different base cases to deal with. On the other hand, $g(n)$ could be computed iteratively by calculating $g(0), g(1), \ldots, g(n)$ in that order. Compare the iterative and recursive implementations for g given below.

```python
def iterative_fib(n):
    """Calculate the nth Fibonacci number iteratively."""
    if n <= 0:                  # Special case: g(0) = 0.
        return 0
    g0, g1 = 0, 1               # Initialize g(0) and g(1).
    for i in range(1, n):       # Calculate g(2), g(3), ..., g(n).
        g0, g1 = g1, g0 + g1
    return g1

def recursive_fib(n):
    """Calculate the nth Fibonacci number recursively."""
    if n <= 0:                  # Base case 1: g(0) = 0.
        return 0
    elif n == 1:                # Base case 2: g(1) = 1.
        return 1
    else:                       # Recursive case: g(n) = g(n-1) + g(n-2).
        return recursive_fib(n-1) + recursive_fib(n-2)
```

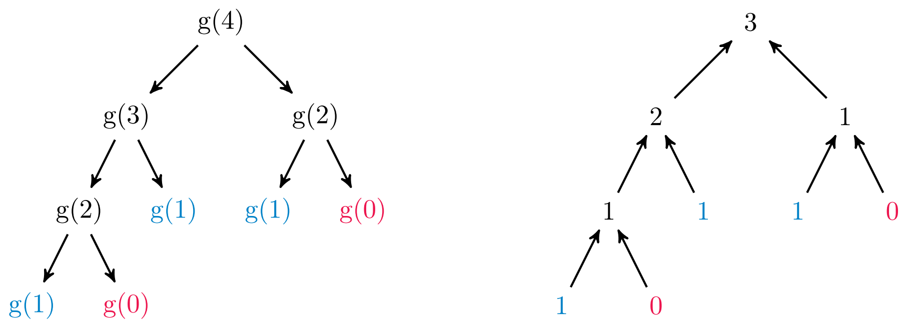

Figure 2.1: To calculate $g(n)$ recursively, call $g(n-1)$ and $g(n-2)$, down to the base cases $g(0)$ and $g(1)$. As the recursion unwinds, the values from the base cases are passed up to previous calls and combined, eventually giving the value for $g(n)$.

Problem 1. Consider the following class for singly linked lists.

```python
class SinglyLinkedListNode:
    """A node with a value and a reference to the next node."""
    def __init__(self, data):
        self.value, self.next = data, None

class SinglyLinkedList:
    """A singly linked list with a head and a tail."""
    def __init__(self):
        self.head, self.tail = None, None

    def append(self, data):
        """Add a node containing the data to the end of the list."""
        n = SinglyLinkedListNode(data)
        if self.head is None:
            self.head, self.tail = n, n
        else:
            self.tail.next = n
            self.tail = n

    def iterative_find(self, data):
        """Search iteratively for a node containing the data."""
        current = self.head
        while current is not None:
            if current.value == data:
                return current
            current = current.next
        raise ValueError(str(data) + " is not in the list")
```

Write a method that does the same task as `iterative_find()`, but with the following recursive approach. Define a function within the method that checks a single node for the data. There are two base cases: if the node is `None`, meaning the data could not be found, raise a `ValueError`; if the node contains the data, return the node. Otherwise, call the function on the next node in the list. Start the recursion by calling this inner function on the head node.
(Hint: see `BST.find()` in the next section for a similar idea.)

Achtung!

It is usually **not** better to rewrite an iterative method recursively, partly because recursion results in an increased number of function calls. Each call requires a small amount of memory so the program remembers where to return to in the program. By default, Python raises a `RuntimeError` after 1000 calls to prevent a stack overflow. On the other hand, recursion lends itself well to some problems; in this lab, we use a recursive approach to construct a few data structures, but it is possible to implement the same structures with iterative strategies.

Binary Search Trees

Mathematically, a *tree* is a directed graph with no cycles. Trees can be implemented with link-based data structures that are similar to a linked list. The first node in a tree is called the *root*, like the `head` of a linked list. The root node points to other nodes, which are called its children. A node with no children is called a *leaf node*.

A *binary search tree* (BST) is a tree that allows each node to have up to two children, usually called `left` and `right`. The left child of a node contains a value that is less than its parent node's value; the right child's value is greater than its parent's value. This specific structure makes it easy to search a BST: while the computational complexity of finding a value in a linked list is $O(n)$ where n is the number of nodes, a well-built tree finds values in $O(\log n)$ time.

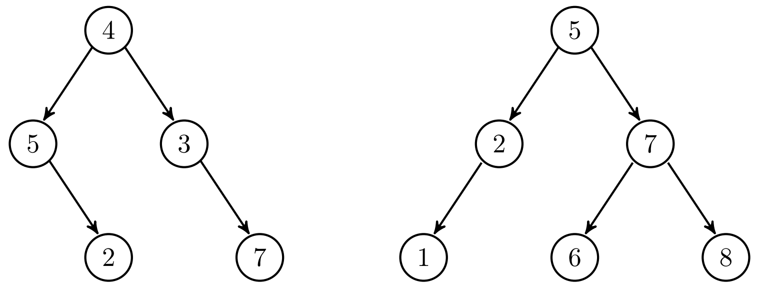

Figure 2.2: Both of these graphs are trees, but the tree on the left is not a binary search tree because 5 is to the left of 4. Swapping 5 and 3 in the graph on the left would result in a BST.

Binary search tree nodes have attributes that keep track of their value, their children, and (in doubly linked trees) their parent. The actual binary search tree has an attribute to keep track of its root node.

```python
class BSTNode:
    """A node class for binary search trees. Contains a value, a
    reference to the parent node, and references to two child nodes.
    """
    def __init__(self, data):
        """Construct a new node and set the value attribute. The other
        attributes will be set when the node is added to a tree.
        """
        self.value = data
        self.prev = None        # A reference to this node's parent node.
        self.left = None        # self.left.value < self.value
        self.right = None       # self.value < self.right.value

class BST:
    """Binary search tree data structure class.
    The root attribute references the first node in the tree.
    """
    def __init__(self):
        """Initialize the root attribute."""
        self.root = None
```

Note

Conceptually, each node of a BST partitions the data of its subtree into two halves: the data that is less than the parent, and the data that is greater. We will extend this concept to higher dimensions in the next lab.

Locating Nodes

Finding a node in a binary search tree can be done recursively. Starting at the root, check if the target data matches the current node. If it does not, then if the data is less than the current node's value, search again on the left child; if the data is greater, search on the right child. Continue the process until the data is found or until hitting a dead end. This method illustrates the advantage of the binary structure—if a value is in a tree, then we know where it ought to be based on the other values in the tree.

```python
class BST:
    # ...
    def find(self, data):
        """Return the node containing the data. If there is no such node
        in the tree, including if the tree is empty, raise a ValueError.
        """

        # Define a recursive function to traverse the tree.
        def _step(current):
            """Recursively step through the tree until the node containing
            the data 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 data == current.value:               # Base case 2: data found!
                return current
            if data < current.value:                # Recursively search left.
                return _step(current.left)
            else:                                   # Recursively search right.
                return _step(current.right)

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

Insertion

New elements are always added to a BST as leaf nodes. To insert a new value, recursively step through the tree as if searching for the value until locating an empty slot. The node with the empty child slot becomes the parent of the new node; connect it to the new node by modifying the parent's `left` or `right` attribute (depending on which side the child should be on) and the child's `prev` attribute.

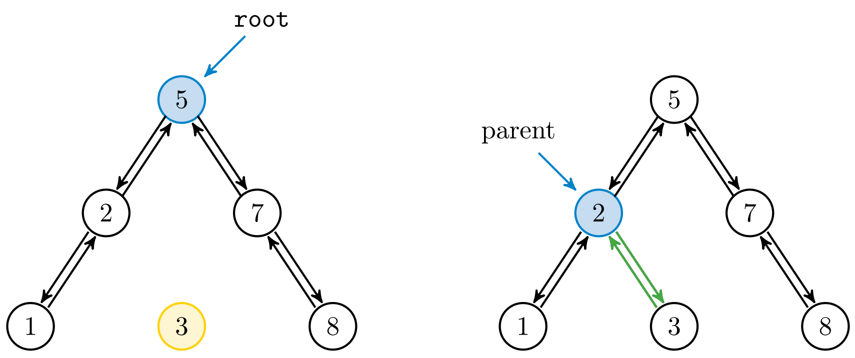

Figure 2.3: To insert 3 to the BST on the left, start at the root and recurse down the tree as if searching for 3: since $3 < 5$, step left to 2; since $2 < 3$, step right. However, 2 has no right child, so 2 becomes the parent of a new node containing 3.

Problem 2. Write an `insert()` method for the `BST` class that accepts some data.

1. If the tree is empty, assign the `root` attribute to a new `BSTNode` containing the data.
2. If the tree is nonempty, create a new `BSTNode` containing the data and find the existing node that should become its parent. Determine whether the new node will be the parent's `left` or `right` child, then double link the parent to the new node accordingly. (Hint: write a recursive function like `_step()` to find and link the parent).
3. Do not allow duplicates in the tree: if there is already a node in the tree containing the insertion data, raise a `ValueError`.

To test your method, use the `__str__()` and `draw()` methods provided in the Additional Materials section. Try constructing the binary search trees in Figures 2.2 and 2.3.

Removal

Node removal is much more delicate than node insertion. While insertion always creates a new leaf node, a remove command may target the root node, a leaf node, or anything in between. There are three main requirements for a successful removal.

1. The target node is no longer in the tree.
2. The former children of the removed node are still accessible from the root. In other words, if the target node has children, those children must be adopted by other nodes in the tree.
3. The tree still has an ordered binary structure.

When removing a node from a linked list, there are three possible cases that must each be accounted for separately: the target node is the head, the target node is the tail, or the target node is in the middle of the list. For BST node removal, we must similarly account separately for the removal of a leaf node, a node with one child, a node with two children, and the root node.

Removing a Leaf Node

Recall that Python's garbage collector automatically deletes objects that cannot be accessed by the user. If the node to be removed—called the *target node*—is a leaf node, then the only way to access it is via the target's parent. Locate the target with `find()`, get a reference to the parent node (using the `prev` attribute of the target), and set the parent's `right` or `left` attribute to `None`.

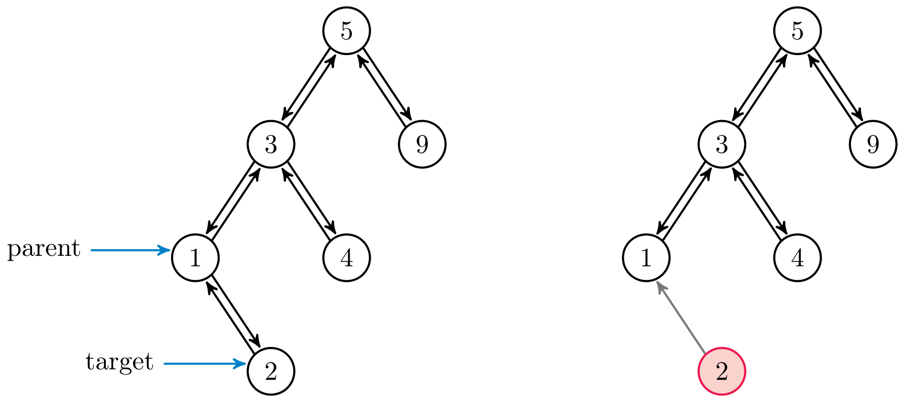

Figure 2.4: To remove 2, get a reference to its parent. Then set the parent's `right` attribute to `None`. Even though 2 still points to 1, 2 is deleted since nothing in the tree points to it.

Removing a Node with One Child

If the target node has one child, the child must be adopted by the target's parent in order to remain in the tree. That is, the parent's `left` or `right` attribute should be set to the child, and the child's `prev` attribute should be set to the parent. This requires checking which side of the target the child is on and which side of the parent the target is on.

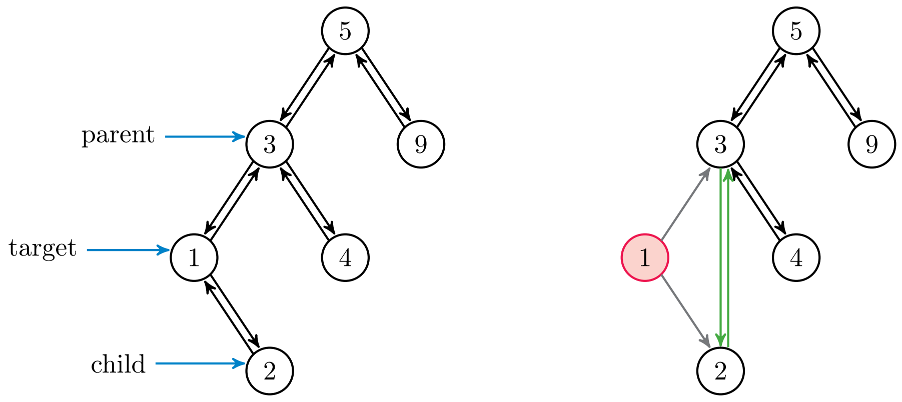

Figure 2.5: To remove 1, locate its parent (3) and its child (2). Set the parent's `left` attribute to the child and the child's `prev` attribute to the parent. Even though 1 still points to other nodes, it is deleted since nothing in the tree points to it.

Removing a Node with Two Children

Removing a node with two children requires a slightly different approach in order to preserve the ordering in the tree. The *immediate predecessor* of a node with value x is the node in the tree with the largest value that is still smaller than x. Replacing a target node with its immediate predecessor preserves the order of the tree because the predecessor's value is greater than the values in the target's left branch, but less than the values in the target's right branch. Note that because of how the predecessor is chosen, any immediate predecessor can only have at most one child.

To remove a target with two children, find its immediate predecessor by stepping to the left 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 value). Remove the predecessor, recording its value. Then overwrite the value of the target with the predecessor's value.

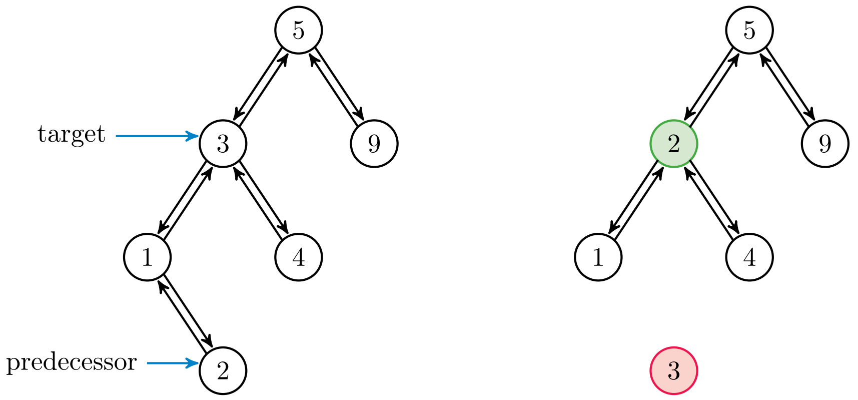

Figure 2.6: To remove 3, locate its immediate predecessor 2 by stepping left to 1, then right as far as possible. Since it is a leaf node, the predecessor can be deleted using the process in Figure 2.4. Delete the predecessor, and replace the value of the target with the predecessor's value. If the predecessor has a left child, it can be deleted with the procedure from Figure 2.5.

Removing the Root Node

If the target is the root node, the `root` attribute may need to be reassigned after the target is removed. This adds two extra cases to consider:

1. If the root has no children, meaning it is the only node in the tree, set the root to `None`.
2. If the root has one child, that child becomes the new root of the tree. The new root's `prev` attribute should be set to `None` so the garbage collector deletes the target.

When the targeted root has two children, the node stays where it is (only its value is changed), so `root` does not need to be reassigned.

Problem 3. Write a `remove()` method for the `BST` class that accepts some data. If the tree is empty, or if there is no node in the tree containing the data, raise a `ValueError`. Otherwise, remove the node containing the specified data using the strategies described in Figures 2.4–2.6. Test your solutions thoroughly.
(Hint: **Before coding anything**, outline the entire method with comments and `if`-`else` blocks. Consider using the following control flow to account for all possible cases.)

1. The target is a leaf node.
    (a) The target is the root.
    (b) The target is to the left of its parent.
    (c) The target is to the right of its parent.
2. The target has two children.
   (Hint: use `remove()` on the predecessor's value).
3. The target has one child.
   (Hint: start by getting a reference to the child.)
    (a) The target is the root.
    (b) The target is to the left of its parent.
    (c) The target is to the right of its parent.

AVL Trees

The advantage of a BST is that it organizes its data so that values can be located, inserted, or removed in $O(\log n)$ time. However, this efficiency is dependent on the *balance* of the tree. In a well-balanced tree, the number of descendants in the left and right subtrees of each node is about the same. An unbalanced tree has some branches with many more nodes than others. Finding a node at the end of a long branch is closer to $O(n)$ than $O(\log n)$. This is a common problem; inserting ordered data, for example, results in a "linear" tree, since new nodes always become the right child of the previously inserted node (see Figure 2.7). The resulting structure is essentially a linked list without a `tail` attribute.

An *Adelson-Velsky Landis tree* (AVL) is a BST that prevents any one branch from getting longer than the others by recursively "balancing" the branches as nodes are added or removed. Insertion and removal thus become more expensive, but the tree is guaranteed to retain its $O(\log n)$ search efficiency. The AVL's balancing algorithm is beyond the scope of this lab, but the Volume 2 text includes details and exercises on the algorithm.

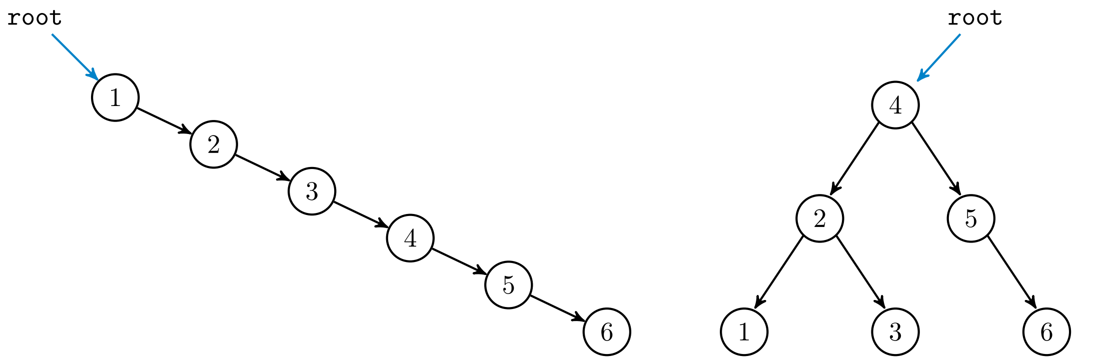

Figure 2.7: On the left, the unbalanced BST resulting from inserting 1, 2, 3, 4, 5, and 6, in that order. On the left, the balanced AVL tree that results from the same insertion. After each insertion, the AVL tree rebalances if necessary.

Problem 4. Write a function to compare the build and search times of the `SinglyLinkedList` from Problem 1, the `BST` from Problems 2 and 3, and the `AVL` provided in the Additional Materials section. Begin by reading the file `english.txt`, storing the contents of each line in a list. For $n = 2^3, 2^4, \ldots, 2^{10}$, repeat the following experiment.

1. Get a subset of n **random** items from the data set.
   (Hint: use a function from the `random` or `np.random` modules.)
2. Time (separately) how long it takes to load a new `SinglyLinkedList`, a `BST`, and an `AVL` with the n items.
3. Choose 5 **random** items from the subset, and time how long it takes to find all 5 items in each data structure. Use the `find()` method for the trees, but to avoid exceeding the maximum recursion depth, use the provided `iterative_find()` method from Problem 1 to search the `SinglyLinkedList`.

Report your findings in a single figure with two subplots: one for build times, and one for search times. Use log scales where appropriate.

Additional Material

Possible Improvements to the BST Class

The following are a few ideas for expanding the functionality of the `BST` class.

1. Add a keyword argument to the constructor so that if an iterable is provided, each element of the iterable is immediately added to the tree. This makes it possible to cast other iterables as a `BST` the same way that an iterable can be cast as one of Python's standard data structures.
2. Add an attribute that keeps track of the number of items in the tree. Use this attribute to implement the `__len__()` magic method.
3. Add a method for translating the `BST` into a sorted Python list.
   (Hint: examine the provided `__str__()` method carefully.)
4. Add methods `min()` and `max()` that return the smallest or largest value in the tree, respectively. Consider adding `head` and `tail` attributes that point to the minimal and maximal elements; this would make inserting new minima and maxima $O(1)$.

Other Kinds of Binary Trees

In addition to the AVL tree, there are many other variations on the binary search tree, each with its own advantages and disadvantages. Consider writing classes for the following structures.

1. A *B-tree* is a tree whose nodes can contain more than one piece of data and point to more than one other node. See the Volume 2 text for details.
2. The nodes of a *red-black tree* are labeled either red or black. The tree satisfies the following rules to maintain a balanced structure.
   (a) Every leaf node is black.
   (b) Red nodes only have black children.
   (c) Every (directed) path from a node to any of its descendent leaf nodes contains the same number of black nodes.

   When a node is added that violates one of these constraints, the tree is rebalanced and recolored.
3. A *Splay Tree* includes an additional operation, called splaying, that makes a specified node the root of the tree. Splaying several nodes of interest makes them easier to access because they are placed close to the root.
4. A *heap* is similar to a BST but uses a different binary sorting rule: the value of every parent node is greater than each of the values of its children. This data structure is particularly useful for sorting algorithms; see the Volume 2 text for more details.

Additional Code: Tree Visualization

The following methods may be helpful for visualizing instances of the `BST` and `AVL` classes. Note that the `draw()` method uses NetworkX's `graphviz_layout`, which requires the `pygraphviz` module (install it with `pip install pygraphviz`).

```python
import networkx as nx
from matplotlib import pyplot as plt
from networkx.drawing.nx_agraph import graphviz_layout

class BST:
    # ...
    def __str__(self):
        """String representation: a hierarchical view of the BST.

        Example:  (3)
                  / \     '[3]          The nodes of the BST are printed
                (2) (5)    [2, 5]       by depth levels. Edges and empty
                /   / \    [1, 4, 6]'   nodes are not printed.
              (1) (4) (6)
        """
        if self.root is None:
            return "[]"
        out, current_level = [], [self.root]
        while current_level:
            next_level, values = [], []
            for node in current_level:
                values.append(node.value)
                for child in [node.left, node.right]:
                    if child is not None:
                        next_level.append(child)
            out.append(values)
            current_level = next_level
        return "\n".join([str(x) for x in out])

    def draw(self):
        """Use NetworkX and Matplotlib to visualize the tree."""
        if self.root is None:
            return
        # Build the directed graph.
        G = nx.DiGraph()
        G.add_node(self.root.value)
        nodes = [self.root]
        while nodes:
            current = nodes.pop(0)
            for child in [current.left, current.right]:
                if child is not None:
                    G.add_edge(current.value, child.value)
                    nodes.append(child)
        # Plot the graph. This requires graphviz_layout (pygraphviz).
        nx.draw(G, pos=graphviz_layout(G, prog="dot"), arrows=True,
                with_labels=True, node_color="C1", font_size=8)
        plt.show()
```

Additional Code: AVL Tree

Use the following class for Problem 4. Note that it inherits from the `BST` class, so its functionality is dependent on the `insert()` method from Problem 2. Note that the `remove()` method is disabled, though it is possible for an AVL tree to rebalance itself after removing a node.

```python
class AVL(BST):
    """Adelson-Velsky Landis binary search tree data structure class.
    Rebalances after insertion when needed.
    """
    def insert(self, data):
        """Insert a node containing the data into the tree, then rebalance."""
        BST.insert(self, data)      # Insert the data like usual.
        n = self.find(data)
        while n:                    # Rebalance from the bottom up.
            n = self._rebalance(n).prev

    def remove(*args, **kwargs):
        """Disable remove() to keep the tree in balance."""
        raise NotImplementedError("remove() is disabled for this class")

    def _rebalance(self,n):
        """Rebalance the subtree starting at the specified node."""
        balance = AVL._balance_factor(n)
        if balance == -2:                                   # Left heavy
            if AVL._height(n.left.left) > AVL._height(n.left.right):
                n = self._rotate_left_left(n)                   # Left Left
            else:
                n = self._rotate_left_right(n)                  # Left Right
        elif balance == 2:                                  # Right heavy
            if AVL._height(n.right.right) > AVL._height(n.right.left):
                n = self._rotate_right_right(n)                 # Right Right
            else:
                n = self._rotate_right_left(n)                  # Right Left
        return n

    @staticmethod
    def _height(current):
        """Calculate the height of a given node by descending recursively until
        there are no further child nodes. Return the number of children in the
        longest chain down.
        """
        if current is None:     # Base case: the end of a branch.
            return -1           # Otherwise, descend down both branches.
        return 1 + max(AVL._height(current.right), AVL._height(current.left))

    @staticmethod
    def _balance_factor(n):
        return AVL._height(n.right) - AVL._height(n.left)
    def _rotate_left_left(self, n):
        temp = n.left
        n.left = temp.right
        if temp.right:
            temp.right.prev = n
        temp.right = n
        temp.prev = n.prev
        n.prev = temp
        if temp.prev:
            if temp.prev.value > temp.value:
                temp.prev.left = temp
            else:
                temp.prev.right = temp
        if n is self.root:
            self.root = temp
        return temp

    def _rotate_right_right(self, n):
        temp = n.right
        n.right = temp.left
        if temp.left:
            temp.left.prev = n
        temp.left = n
        temp.prev = n.prev
        n.prev = temp
        if temp.prev:
            if temp.prev.value > temp.value:
                temp.prev.left = temp
            else:
                temp.prev.right = temp
        if n is self.root:
            self.root = temp
        return temp

    def _rotate_left_right(self, n):
        temp1 = n.left
        temp2 = temp1.right
        temp1.right = temp2.left
        if temp2.left:
            temp2.left.prev = temp1
        temp2.prev = n
        temp2.left = temp1
        temp1.prev = temp2
        n.left = temp2
        return self._rotate_left_left(n)

    def _rotate_right_left(self, n):
        temp1 = n.right
        temp2 = temp1.left
        temp1.left = temp2.right
        if temp2.right:
            temp2.right.prev = temp1
        temp2.prev = n
        temp2.right = temp1
        temp1.prev = temp2
        n.right = temp2
        return self._rotate_right_right(n)
```