UNIVERSITY OF ARKANSAS

CSCE 4133/5133 Algorithms

Homework Assignment #3

Submission Deadline: Mid-Night (11:59PM) March 19, 2026

Instructions

- **Submissions:** Students can use either Google Docs or LaTeX for writing the report. The report template is provided in the Blackboard (Material/Report-template.zip). Your submission should be a zip file containing your report and the source code, including your implementation. Name the zip file as `lastname_studentID.zip`. Submissions should be made via BlackBoard.
- **Policy:** Students must start working on this assignment independently and review the late-day policy.
- If you have any questions please contact us via Teams or email: kimtran@uark.edu and maalexra@uark.edu.

I. Overview

In homework 2, we studied the breadth first search (BFS) and the depth first search (DFS) algorithms. According to the property of BFS, the algorithm always produces the shortest path if the graph is unweighted. In other words, BFS produces the path with least number of edges. However, in practice, each edge of a graph has a weight, e.g., the distance between two buildings. Also, we always want to go to the destination with the shortest distance. In this case, BFS cannot produce the shortest path. Interestingly, we can modify the BFS algorithm to find the shortest path. In particular, at each time, instead of popping out the node from head of the queue, we are going to pop out the node with the shortest path in the current queue. Performing this operator takes O(n) (n is the number of nodes in the queue) to search for the minimum node in the current queue. We can optimize this step using binary search tree and reduce the complexity to O(logn).

In this homework 3, we are going to implement Binary Search Tree to store data in order so that we can search or find the minimum value quickly. The map is represented as a graph structure where each node represents one building (or a landmark), and an undirected edge between two nodes represents the path between buildings/landmarks. A list of edges in a graph is represented by an adjacency list that is implemented by a linked list (each edge also includes a weight value). The modified BFS algorithm has been implemented.

In this assignment, you are **REQUIRED** to implement **Binary Search Tree**. If you think the homework is too long, don't worry, it is long because we provide you with the detail of the descriptions. There are some hints that may help to be success in this homework:

- You should start the homework early.
- You should read the descriptions of homework carefully before you start working on the homework.
- Before implementing the required functions, you should try to compile the source code first.Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forgot binary search tree, you are encouraged to revise the lecture slides.

The source code is organized as follows:

File or Folder	Required Implementation	Purpose
assets/	No	Contain the data of homework
linked_list.py	No	Implement the linked list structure
graph.py	No	Implement the graph structure and algorithms
bfs.py	No	Implement the modified BFS algorithm
bst.py	Yes	Implement the BST
avl.py	Yes	Implement the AVL
main.py	No	Implement the main program

II. Implementation

In this section, we are going to implement the basic version of BST. First, we will review the definition of the node in BST and class BST.

```python
class BSTNode:
    def __init__(self, key: T = None, height: int = 0, left: Optional['BSTNode'] = None,
    right: Optional['BSTNode'] = None):
        if key is not None:
            self.key = key
        self.height = height
        self.left = left if left is not None else None
        self.right = right if right is not None else None

    def __del__(self):
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root: Optional[BSTNode] = None

    def __del__(self):
        self.clear(self.root)
        self.root = None

    def clear(self, node: Optional[BSTNode]) -> None:
        # YOUR CODE HERE
        pass

    def _find(self, key: T, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _findMaximum(self, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _findMinimum(self, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _insert(self, key: T, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _remove(self, key: T, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def find(self, key: T) -> Optional[BSTNode]:
        return self._find(key, self.root)

    def findMaximum(self) -> Optional[BSTNode]:
        return self._findMaximum(self.root)

    def findMinimum(self) -> Optional[BSTNode]:
        return self._findMinimum(self.root)

    def insert(self, key: T) -> Optional[BSTNode]:
        self.root = self._insert(key, self.root)
        return self.root

    def remove(self, key: T) -> Optional[BSTNode]:
        self.root = self._remove(key, self.root)
        return self.root

    def getRoot(self) -> Optional[BSTNode]:
        return self.root
```

The **BSTNode** includes four attributes, i.e. the key of the node, the height of the node, the left and right children pointers of the node. For class **BST**, there is a private attribute which is a root of the BST. **There are six private methods that you need to implement in bst.py:**

- `def clear(self, node: Optional[BSTNode])`: Release (deallocate) the memory of all nodes in the subtree whose root is given by the parameter **node**. In Python, this can be done by assigning each node reference to `None`.
- `def _find(self, key: T, node: Optional[BSTNode])`: Find the given **key** in the subtree whose root is the parameter **node**. If the key exists in the subtree, return a pointer to the node containing the key; otherwise, return the NULL pointer.

- `def _findMaximum(self, node: Optional[BSTNode])`: Find the maximum key in the subtree whose root is the parameter **node**. Return a pointer to the node containing the maximum key. If the subtree is empty, return the NULL pointer. The `_findMinimum` function is similar.
- `def _insert(self, key: T, node: Optional[BSTNode])`: Insert the given **key** into the subtree whose root is the parameter **node**. Return the root pointer of the subtree after insertion. If the key already exists, return the root unchanged (duplicate keys are not inserted).
- `def _remove(self, key: T, node: Optional[BSTNode])`: Remove the given **key** from the subtree whose root is the parameter **node**. Return the root pointer of the subtree after deletion.

There is a sample test implemented in the main program (**main.py**) for the BST, as follows (the sample test for AVL is almost the same):

```python
def testBST():
    bst = BST()
    for i in range(10):
        print("Insert " + str(i) + " into BST")
        bst.insert(i)

        print("Find " + str(i) + ". ")
        if bst.find(i) is not None and bst.find(i).key == i:
            print("Found " + str(i) + " in BST")
        else:
            return False

        print("Find " + str(i + 1) + ". ")
        if bst.find(i + 1) is None:
            print("Not found " + str(i + 1) + " in BST")
        else:
            return False
        print()
    print("Maximum value in BST: ")
    if bst.findMaximum() is None or bst.findMaximum().key != 9:
        return False
    else:
        print(str(bst.findMaximum().key) + "\n")

    for i in range(9):
        print("Remove " + str(i) + " out of BST")
        bst.remove(i)

        print("Find " + str(i) + ". ")
        if bst.find(i) is None:
            print("Not found " + str(i) + " in BST")
        else:
            return False

        print("Find " + str(i + 1) + ". ")
        if bst.find(i + 1) is not None and bst.find(i + 1).key == i + 1:
            print("Found " + str(i + 1) + " in BST")
        else:
            return False
        print()

    return True
```

After your BST works correctly, you can proceed to implement the AVL tree, which is an improvement over the BST. The definition of an AVL tree is as follows:

```python
class AVL(BST):
    def getHeight(self, node: Optional[BSTNode]) -> int:
        return -1 if node is None else node.height

    def getBalance(self, node: Optional[BSTNode]) -> int:
        return 0 if node is None else self.getHeight(node.left) - self.getHeight(node.right
    )

    def rotateRight(self, y: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def rotateLeft(self, x: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _insert(self, key: T, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def _remove(self, key: T, node: Optional[BSTNode]) -> Optional[BSTNode]:
        # YOUR CODE HERE
        return None  # Placeholder

    def insert(self, key: T) -> Optional[BSTNode]:
        self.root = self._insert(key, self.root)
        return self.root

    def remove(self, key: T) -> Optional[BSTNode]:
        self.root = self._remove(key, self.root)
        return self.root
```

There are four methods that you need to implement in `avl.py`:

- `def rotateRight(self, y: Optional[BSTNode])`: Perform a right rotation on the subtree whose root is the parameter **y**. Return the new root of the subtree after rotation. This operation is used to rebalance the tree when a left-heavy imbalance occurs.

- `def rotateLeft(self, x: Optional[BSTNode])`: Perform a left rotation on the subtree whose root is the parameter **x**. Return the new root of the subtree after rotation. This operation is used to rebalance the tree when a right-heavy imbalance occurs.
- `def _insert(self, key: T, node: Optional[BSTNode])`: Insert the given **key** into the subtree whose root is the parameter **node**. After insertion, update the height of each affected node and rebalance the subtree if necessary using appropriate rotations. Return the root pointer of the subtree after insertion. Duplicate keys are not inserted.
- `def _remove(self, key: T, node: Optional[BSTNode])`: Remove the given **key** from the subtree whose root is the parameter **node**. After deletion, update the height of each affected node and rebalance the subtree if necessary using appropriate rotations. Return the root pointer of the subtree after deletion.

III. Compilation and Testing

In this homework 3, we use the same environment (Python3) used in the previous ones to compile the source code. If you implement all methods correctly, the unit test function should return "True", and you should see the following lines in your output:

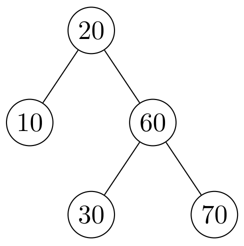
```
Your BST implementation is correct
Your AVL implementation is correct
Shortest path from JBHT to HAPG: JBHT -> HILL -> WJWH -> HAPG
Total Distance: 47354
```

IV. Practice Problems

Answer the following questions in the report:

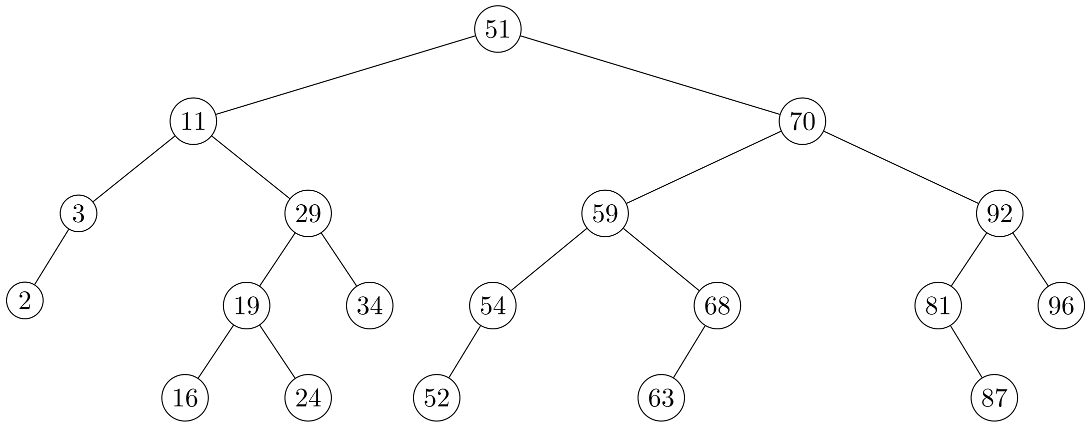
A. What is the maximum number of nodes in a binary tree of height h? (Assume the height of a single-node tree is 0).

B. Given a AVL Tree:



Insert 25 and maintain AVL tree. Your answer should be the AVL tree after insertion.

C. Given the follwoing Binary Tree, insert 47 into the binary Tree.



D. (After inserting 47) Delete 92 from the binary tree. Also make sure the Binary tree is balanced.

E. What is the worst-case time complexity of inserting a node into an AVL tree? Why?

F. What is the worst-case time complexity of deleting a node in an AVL tree? Why?

V. Submission

Write a report detailing the key ideas for implementing each required function and include the implemented functions themselves. You must also include a screenshot of the final output (from either command prompt or terminal) after running the program.

Write your full name, email address and student ID in the report. Your submission should be a zip file of your report and the source code, including your implementation. Name the zip file as 'lastname_studentID.zip'. Submission via BlackBoard.