Machine Problem 2 Documentation

Machine Problem Title: AVL Tree Visualizer

Course & Section: CS102-1L-CIS201

Programming Languages Used: Python

Submitted By: Lasay, Lean & Olayvar, Hannali S.,

Date Submitted: 15/11/2025

1. The Problem Description

The AVL Tree Visualiser is a graphical application made in python with the usage of the tkinter library. It was made to show how AVLs work visually through dynamic visual representation. An AVL tree is a type of BST where heights of the left and right subtrees of every node differ by at most 1. When this is violatees during an attempt to insert a node, the tree will perform rotations to restore the Balance.

This project shows a real time visual representation of AVL Tree operations,

2. References and Project Resources

This section provides access to the project's supporting materials, including the repository containing the source code and raw flowchart file.

- Github Repository - source code
- Simple Flowchart PDF
- Flowchart Figma Ver

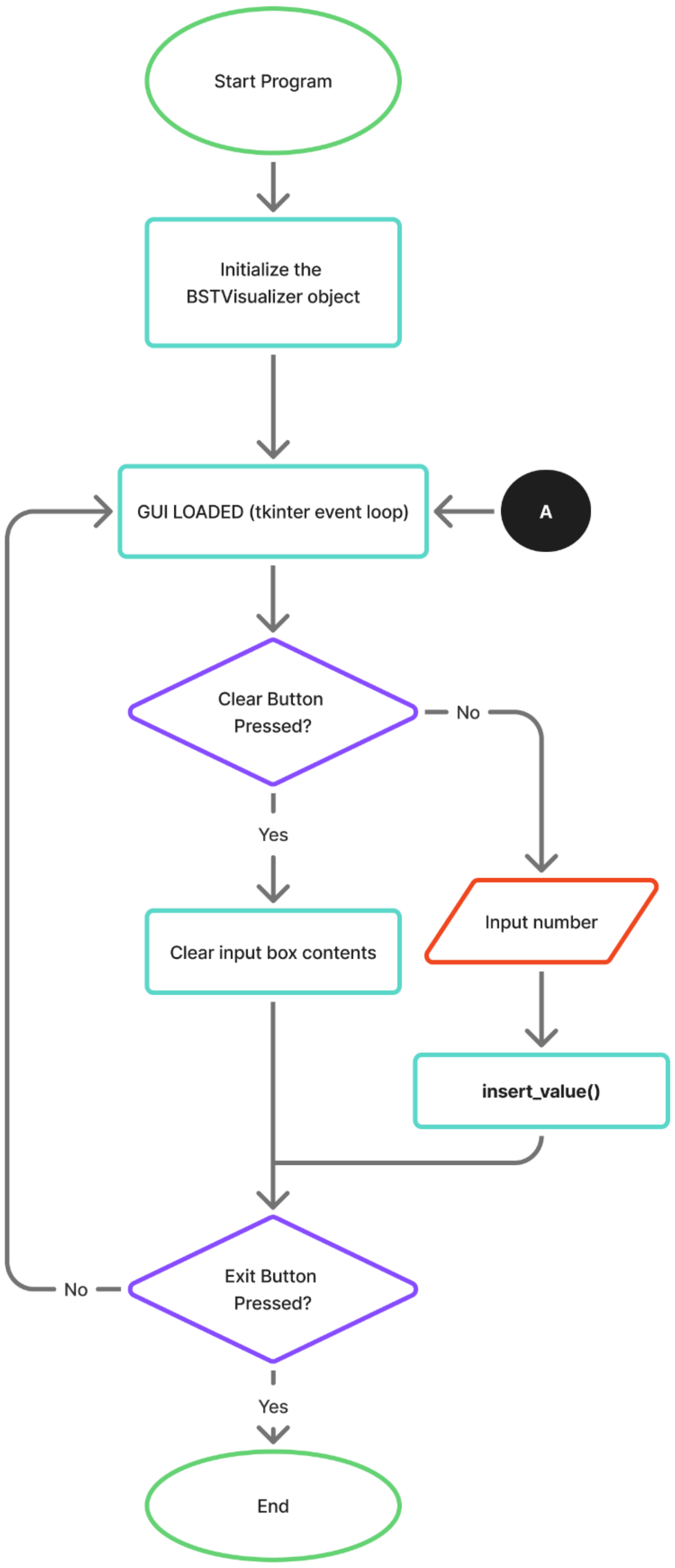
3. Program Design and Structure

Main Flowchart

This flowchart describes the dynamics of the main script. The simple flowchart shows the basic flow of the AVL Tree Visualizer.

The following occurs in the flowchart

- ➢ Set up and initialize the graphical user interface (GUI).
- ➢ Handle user actions in the main window, including the events triggered when specific buttons are pressed.

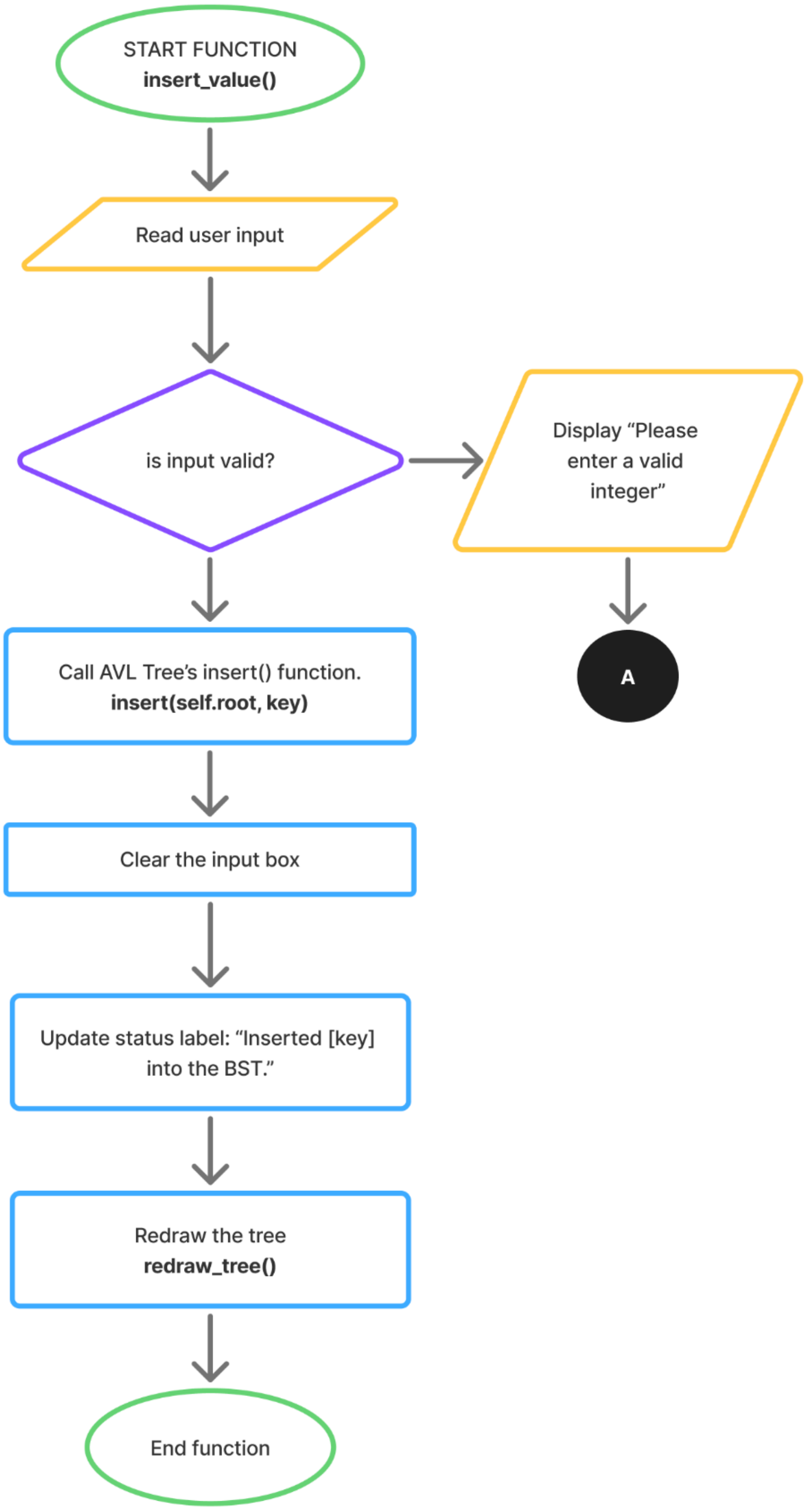


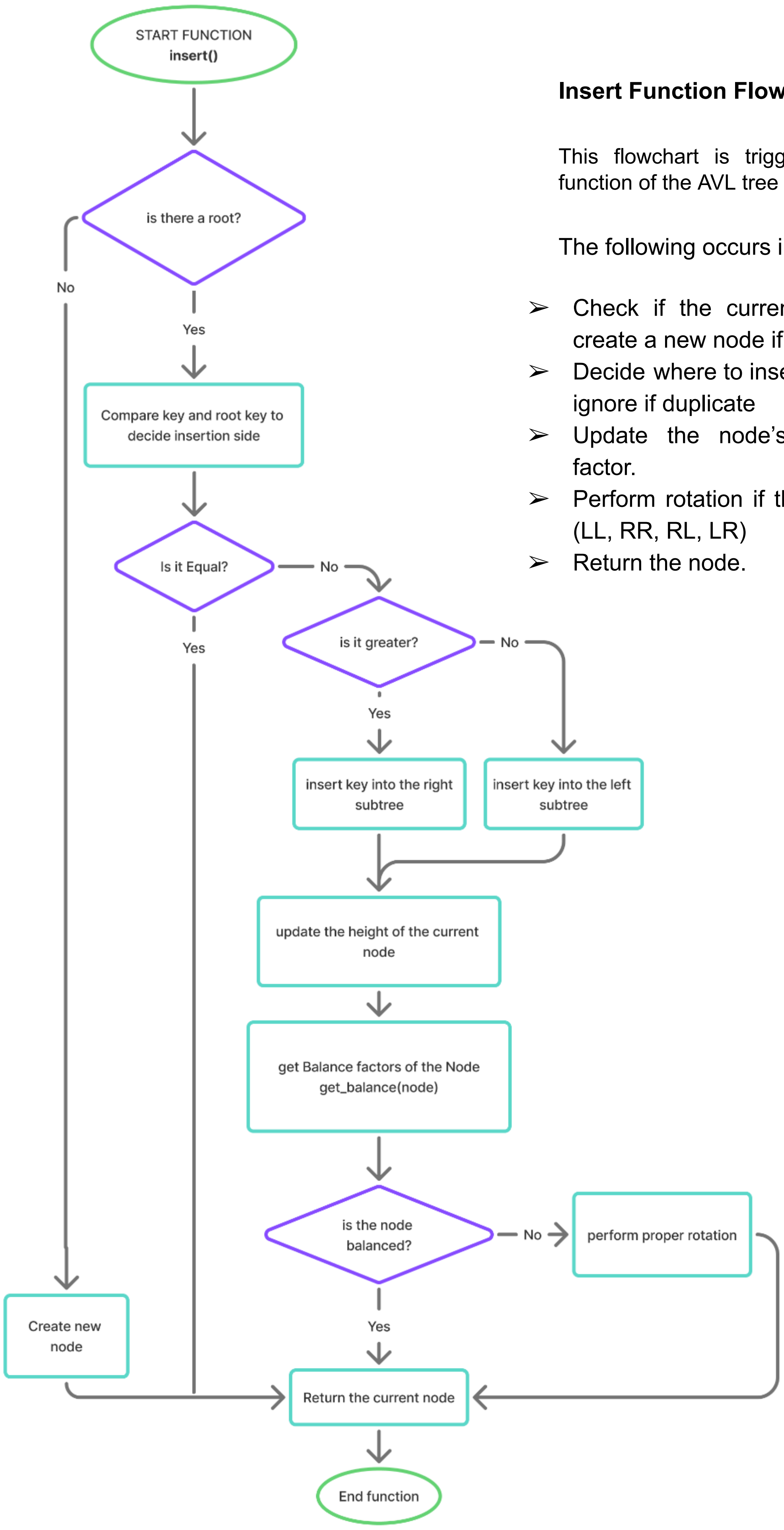
Insert_value Function Flowchart

This flowchart is triggered when the insert_value() function is called in the Main flowchart above.

The following occurs in the flowchart

- ➢ Read the user input from the entry box.
- ➢ Check if the input is a valid integer.
- ➢ If the input is invalid, display the message: “Please enter a valid integer.”
- ➢ If the input is valid, insert the value into the AVL tree using the insert() function.
- ➢ Clear the input box.
- ➢ Update the status label to show: “Inserted [key] into the BST.”
- ➢ Redraw the tree on the canvas to reflect the updated structure.




START FUNCTION
insert()
is there a root?
No
Yes
Compare key and root key to decide insertion side
Is it Equal?
No
Yes
is it greater?
No
Yes
insert key into the right subtree
insert key into the left subtree
update the height of the current node
get Balance factors of the Node get_balance(node)
is the node balanced?
No
perform proper rotation
Yes
Create new node
Return the current node
End function

Insert Function Flowchart

This flowchart is triggered when the **insert()** function of the AVL tree is called.

The following occurs in the flowchart

- ➢ Check if the current node is empty and create a new node if needed.
- ➢ Decide where to insert the key: left, right, or ignore if duplicate
- ➢ Update the node's height and balance factor.
- ➢ Perform rotation if the node is unbalanced (LL, RR, RL, LR)
- ➢ Return the node.

Classes

- Class **AVLTree** in **bst.py**

```python
class AVLTree:

    def insert(self, root, key):
        if root is None:
            return Node(key)

        #inserts
        if key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        else:
            return root

        root.height = 1 +
max(self.get_height(root.left),
self.get_height(root.right))

        #Get balance
        balance = self.get_balance(root)


        # LL Case
        if balance > 1 and key < root.left.key:
            return self.ll_case(root)

        # RR Case
        if balance < -1 and key > root.right.key:
            return self.rr_case(root)

        # LR Case
        if balance > 1 and key > root.left.key:
            root.left = self.rr_case(root.left)
            return self.ll_case(root)

        # RL Case
        if balance < -1 and key < root.right.key:
            root.right = self.ll_case(root.right)
            return self.rr_case(root)

        return root

    def get_height(self,root):
        if not root:
            return 0
        return root.height

    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) -
self.get_height(root.right)

    #rotations
    def ll_case(self, _1):
        _2 = _1.left
        child = _2.right
        _2.right = _1
        _1.left = child

        _2.height = 1+max(self.get_height(_2.left),
self.get_height(_2.right))
        _1.height = 1+max(self.get_height(_1.left),
self.get_height(_1.right))

        return _2

    def rr_case(self, _1):
        _2 = _1.right
        child = _2.left
        _2.left = _1
        _1.right = child

        _2.height = 1+max(self.get_height(_2.left),
self.get_height(_2.right))
        _1.height = 1+max(self.get_height(_1.left),
self.get_height(_1.right))
        return _2
```

Method	Description
insert(root, key)	Inserts a new node into the AVL tree. Updates and check the balance factors of nodes to determin what rotation is needed. Returns root after insertion
get_height(root)	Returns the height of the node. Used for computing the balance factor and updating the height after insertion.
get_balance(root)	Calculates the balance factor of a node. Subtracts the height of the right subtree to left subtree.
ll_case(node)	Performs a right rotation around _1, the unbalanced ancestor node (_2 represents its left child). Makes _2 the new root of that subtree and reassigns the right child of _2 as the left child of _2. After rotation, both nodes' heights are updated.
rr_case(node)	Performs a left rotation around _1, the unbalanced ancestor node (_2 represent its right child). Makes _2 the new root of that subtree and reassigns the right child of _2 as the left child of _2. After rotation, both nodes' heights are updated.

Description: This class is the core logic behind the dynamics of each node in the AVL Tree. It executes the insertion and self-balancing mechanism of the tree. Making sure that after every insertion, the balance factors does not exceed one. If it does, appropriate rotations (LL,RR, LR, RL) are performed to restore balance.

- Class **Node** in **node.py**

```
eclass Node:
   def __init__(self,key):
self.key = key
self.left = None
self.right = None
self.height = 1
```

Attribute	Description
key	Stores the data in the data field
left	Points to the left child node in the tree
right	Points to the right child node in the tree
height	Stores the height of the node, updated every insertion

Description: Each element in the tree is considered as a node. It contains a key (data) and two pointers, the left and right, that refers to its child nodes / subtrees. The height is used in calculating the balance factor.

- Main **script** in **main.py**

```
class BSTVisualizer:
    def __init__(self, window):
       #create the window

    def insert_value(self):
       # Insert user input into
AVL tree and redraw.


    def draw_tree(self, node, x,
y, x_offset):
         # Recursively draw tree
nodes and connecting lines.

    def redraw_tree(self):
        #"Clear and redraw
entire tree.

    def clear_tree(self):
        #Reset tree to empty


#MAIN PROGRAM
if __name__ == "__main__":
    window = tk.Tk()
    app = BSTVisualizer(window)
    window.mainloop()
```

Method	Description
main()	The main function that starts the program. Created the GUI window, initializes the visualizer, and starts Tkinter.
__init__(window)	Initializes Tkinter window, sets up elements and canvas for the visualization.
insert_value()	Gets input from the entry field, validates it, and inserts the value into the avl tree.
draw_tree(node, x, y, x_offset)	Recursively draws the tree in the GUI
redraw_tree()	Clears the canvas and redraws the entire tree based on the current AVL tree structure
clear_tree()	Removes all nodes from the canvas and resets tree.

Description: This class implements the visualization of the AVL Tree. Through tkinter, it generates a graphical user interface for the tree, It handles the user input, drawing of the tree structure on a canvas, and updates the display dynamically as nodes are inserted or cleared.

Grading rubric

	Excellent	Good	Fair	Poor
Problem description	15 Clearly and thoroughly explains the Towers of Hanoi problem; includes objectives, rules, and stack-based solution overview; well written and technically accurate.	10 Mostly clear and correct; minor omissions or unclear phrasing.	5 Basic explanation; missing some objectives or problem rules.	1 Incomplete, vague, or missing explanation of the problem.
Program design and structure	30 Provides clear and accurate flowchart; includes complete class and method descriptions with correct relationships; structure is well organized and easy to follow.	20 Flowchart and design are mostly correct; minor logical or structural inconsistencies; class/method. explanations mostly clear.	10 Flowchart incomplete or poorly labeled; class/method details are lacking.	2 Incorrect flowchart, classes, and structure

Total = 45 points