**Assignment 3 Report**

**Aditya Kharbanda 23364313**

You will find the screenshots of the outputs for Task 2 Binary Search Tree Implementation at the last two pages. There are no errors when queried by the profiler.

To ensure that the BST is balanced, I have implemented the following new functions:

1. **Tree\_search**: Takes the root pointer and a value of doc\_id. Returns a pointer to a Node whose doc\_id value matches the given value. Or, returns NULL if no such node found.
2. **Height**: Takes a pointer to a node in the BST. Returns it’s height in the tree.
3. **Max**: Takes two integers and returns the maximum integer out of them.
4. **rightRotate**: Takes a pointer to a node and right rotates the tree about that node, while updating the height of node. The node becomes the right child of its left child. This is done when the balance factor of the node is greater than 1.
5. **leftRotate**: Takes a pointer to a node and left rotates the tree about that node, while updating the height of node. The node becomes the left child of its right child. This is done when the balance factor of the node is less than -1.
6. **getBalance**: Takes a pointer to a node in the BST as the argument and returns the Balance Factor of that node, i.e. returns the difference between the heights of the left and right subtree of that node.
7. **Createbst**: Takes the following arguments:
   1. Doc\_id
   2. Name
   3. Author
   4. Word Count
   5. Pointer to the root node.

Updates the total number of nodes. Creates a new node and populates it with the arguments. If the BST doesn’t exist, sets the root pointer to the new node and returns a pointer to the new node. If root isn’t NULL, the function first inserts the node in its right position, by finding the correct parent. Calculates the balance factor for the new node and then checks for the following cases:

1. Left Left Case: Right Rotate about the new node
2. Right Right Case: Left Rotate about the new node
3. Left Right Case: Left Rotate the left subtree of the new node, then right rotate the node.
4. Right Left Case: Right Rotate the right subtree of the new node then left rotate the node.

Returns a pointer to the root node.

1. **Tree\_delete**: Takes the root pointer as the argument and recursively frees all the dynamically allocated memory.

**Result**

After having implemented the above functions to maintain a self-balancing tree, here are some key results that made the BST implementation far superior over a Linked List implementation.

**Even Number Of Inputs:**

* Average Title Search Time reduced by 99.57%
* Total Title Search Time reduced by 99.4%
* Average Word Count Search Time reduced by 99.57%
* Total Word Count Search Time reduced by 99.38%
* Average Comparisons Per Search Reduced by 99.97%

**Odd Number Of Inputs:**

* Average Title Search Time reduced by 99.43%
* Total Title Search Time reduced by 99.14%
* Average Word Count Search Time reduced by 99.43%
* Total Word Count Search Time reduced by 99.19%
* Average Comparisons Per Search Reduced by 99.97%

**Extra Testing Parameters**

To further test the BST, the following parameters were evaluated:

1. Check whether the BST is balanced.
2. Check whether the number of books inserted is equal to the total number of nodes in the BST.
3. Compute the Average Comparisons Per Search.
4. Compute the height of the tree.

The functions to implement the above testing parameters are:

* **isBalanced**: Takes the root pointer as argument and recursively checks whether the difference in heights of left and right subtree to the root pointer is less than or equal to 1. Returns 1 if that’s the case, else returns 0.
* **totalNodes**: Takes the root pointer as argument and recursively calculates the total number of nodes in the BST.
* **G\_num\_comps** & **g\_num\_searches**: Two global variables that get updated in the tree\_search() function and the bstdb\_get\_word\_count() and bstdb\_get\_name() functions. Average Comparisons Per Search is calculated by dividing number of comparisons by number of searches.



