3D5A Lab8&9

Abstract
The objective of this experiment was to learn how to implement a tree, become familiar with Binary Search Trees and assess the performance of Binary Search Trees.

Results:

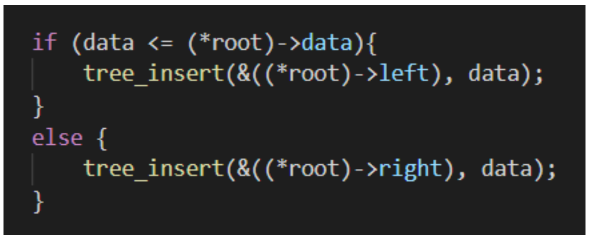
1. In task 1, I implemented a AVL Binary Search Tree (BST) using char as the data records. I began by first writing a suitable structure to represent the node. I then implemented the following four functions:

   - `void tree_insert ( Tree_Node** root, char data );`
   - `Tree_Node* tree_search ( Tree_Node* root, char data );`
   - `void tree_printed_sorted ( Tree_node* root );`
   - `void tree_delete ( Tree_Node* root );`

   I tested these functions by loading a sequence of characters into the tree and searching for random characters in that main function. My implementation of the tree_insert(Tree_Node** root, char data) function took the root and data record as parameters. It compared the data with the data record stored in the root. If the data was less than the data record than it would call itself recursively using root->left as the first parameter. However, if the data was equal to or greater than data record stored in the root it would call itself recursively using root->right as the first parameter; that is, until it reached an empty node. This is shown in Figure 1. This means that any data that is less than the root was stored to the left of it and any data that was equal to or greater than the root were stored to the right of it.

   It is important to note that if an already ordered sequence was loaded into a BST, the tree could become unbalanced and perform like a linked list. This is very inefficient because this means that worst-case scenario for a linked list is O(n). However, the worst-case scenario for a balanced BST is O(log n) which means it is very efficient. I resolved this problem by implementing AVL BST, a self-balancing BST. It checks the height differences after each node is added and performing rotations to if the difference in heights become too great to balance itself. This was all performed by my implementation of the tree_insert(Tree_Node** root, char data).

   My implementations of tree_search (Tree_Node* root, char data), tree_printed_sorted (Tree_node* root) and tree_delete (Tree_Node* root) were much simpler than this. tree_search (Tree_Node* root, char data) searched iteratively through the BST for the data record. It would return the node if it found a node with that data record or NULL if it couldn't find a node. tree_printed_sorted (Tree_node* root) printed all the data records from left to right by calling itself recursively and tree_delete (Tree_Node* root) deleted every node by calling itself recursively as well.



```c
if (data <= (*root)->data){
    tree_insert(&((*root)->left), data);
}
else {
    tree_insert(&((*root)->right), data);
}
```

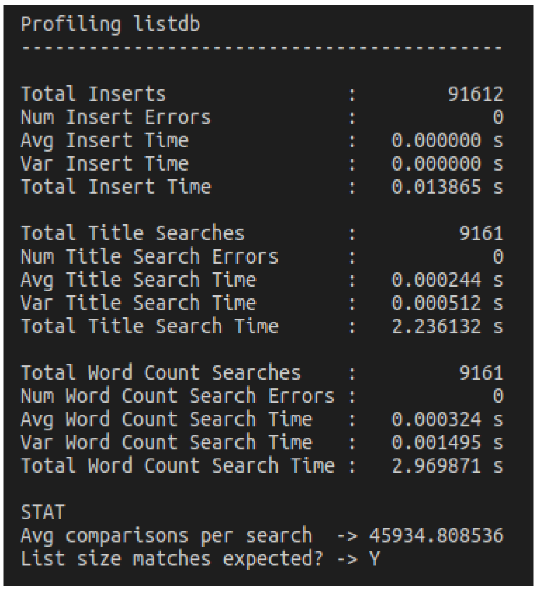
Figure 1. Recursive Call

2. In task2, we had to build a reasonably simple database which stores documents in memory. The database had to store the name and word count of each book. As well as this, it had to assign a unique identifier to each book as it is added to the database. The developer that was hired before me attempted to build the database using a linked list, but I decided to build the database using a AVL BST instead because it would perform drastically better than the linked list. I had to write six functions for this task.

   - `int bstdb_init ( void );`
   - `int bstdb_add ( char *name, int word_count);`
   - `int bstdb_get_word_count ( int doc_id );`
   - `char* bstdb_get_name ( int doc_id );`
   - `void bstdb_stat ( void );`
   - `void bstdb_quit ( void );`

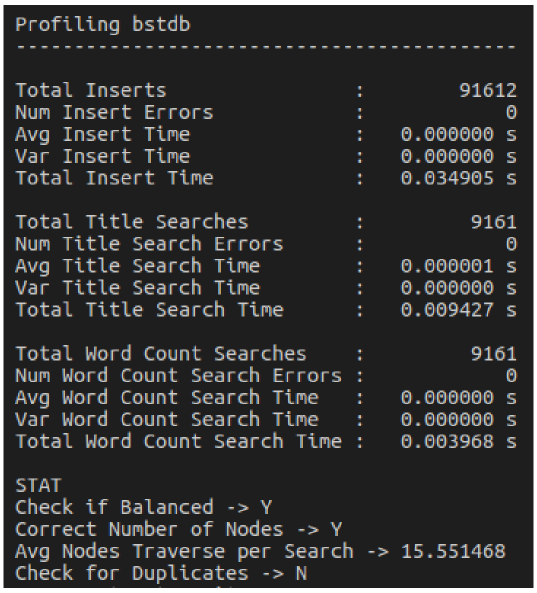
   My implementation of the bstdb_init (void) function initialised BST and all the global variables. My implementation bstdb_add (char *name, int word_count), similarly to tree_insert (Tree_Node** root, char data), created a new node in the binary tree, populated it with the name and word_count and stored it in the tree. It then checked the height difference between each node and performing rotations to if the difference in heights to balance the tree. My implementation of bstdb_get_word_count (int doc_id) and bstdb_get_name (int doc_id) searched for the doc_id, similarly to how Tree_Node* tree_search (Tree_Node* root, char data) searched for the data record, and my implementation of bstdb_quit (void) deleted the tree, similarly to how tree_delete (Tree_Node* root) deleted the tree in task 1. The only notable difference between my program for task1 and task 2 is my implementation of bstdb_stat (void). It checked if the tree was balanced, checked if number of nodes in the tree matched our expected result, calculate the average number of nodes visited per search and verified that there were no duplicate IDs in the tree.

   My implementation of the of bstdb_stat (void) was a great help for comparing the performance of a Linked List against a AVL BST. I observed that the total insertion time by a factor of 3 the total search time decreased by a factor of 237, which was evident for both search functions as shown in Figure 2 and 3. This was a result of the average comparison per second decreasing from 45934 to 15, a drastic improvement. This was because I built a AVL BST instead of a Linked List.



```
Profiling listdb
-----------------------------------------

Total Inserts                :      91612
Num Insert Errors            :          0
Avg Insert Time              :   0.000000 s
Var Insert Time              :   0.000000 s
Total Insert Time            :   0.013865 s

Total Title Searches         :       9161
Num Title Search Errors      :          0
Avg Title Search Time        :   0.000244 s
Var Title Search Time        :   0.000512 s
Total Title Search Time      :   2.236132 s

Total Word Count Searches    :       9161
Num Word Count Search Errors :          0
Avg Word Count Search Time   :   0.000324 s
Var Word Count Search Time   :   0.001495 s
Total Word Count Search Time :   2.969871 s

STAT
Avg comparisons per search  -> 45934.808536
List size matches expected? -> Y
```

Figure 2.



```
Profiling bstdb
-----------------------------------------

Total Inserts                :      91612
Num Insert Errors            :          0
Avg Insert Time              :   0.000000 s
Var Insert Time              :   0.000000 s
Total Insert Time            :   0.034905 s

Total Title Searches         :       9161
Num Title Search Errors      :          0
Avg Title Search Time        :   0.000001 s
Var Title Search Time        :   0.000000 s
Total Title Search Time      :   0.009427 s

Total Word Count Searches    :       9161
Num Word Count Search Errors :          0
Avg Word Count Search Time   :   0.000000 s
Var Word Count Search Time   :   0.000000 s
Total Word Count Search Time :   0.003968 s

STAT
Check if Balanced -> Y
Correct Number of Nodes -> Y
Avg Nodes Traverse per Search -> 15.551468
Check for Duplicates -> N
```

Figure3.

It is evident from these two figures that an AVL BST would perform drastically better than a Linked List for this database. I even verified that it was an AVL tree by checking if it was balanced using my STAT function.

References

1. https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
2. https://simpledevcode.wordpress.com/2014/09/16/avl-tree-in-c/
3. https://simpledevcode.files.wordpress.com/2014/09/5a863-800px-tree_rebalancing.gif?w=641&h=453&zoom=2