**Assignment 3: SEARCHING WITH TREES**

Department of Computer Engineering

Dr Mélanie Bouroche

**Submitted By:**

Aniket Agarwal

Student ID: 17317437

**Theory:**

A binary search tree is a [rooted](https://en.wikipedia.org/wiki/Rooted_tree) [binary tree](https://en.wikipedia.org/wiki/Binary_tree), whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the [binary search](https://en.wikipedia.org/wiki/Binary_search) property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.

The major advantage of binary search trees over other data structures is that the related [sorting algorithms](https://en.wikipedia.org/wiki/Sorting_algorithm) and [search algorithms](https://en.wikipedia.org/wiki/Search_algorithm) such as [in-order traversal](https://en.wikipedia.org/wiki/In-order_traversal) can be very efficient; they are also easy to code.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** |  | **Average** | **Worst case** |
| Space |  | O(n) | O(n) |
| Search |  | O(log n) | O(n) |
| Insert |  | O(log n) | O(n) |
| Delete |  | O(log n) | O(n) |

**Linked List:**

Item(1) -> Item(2) -> Item(3) -> Item(4) -> Item(5) -> Item(6) -> Item(7)

**Binary tree:**

Node (1)

/

Node(2)

/ \

/ Node(3)

RootNode(4)

\ Node(5)

\ /

Node (6)

\

Node (7)

In a linked list, the items are linked together through a single next pointer. In a binary tree, each node can have 0, 1 or 2 sub nodes, where (in case of a binary search tree) the key of the left node is lesser than the key of the node and the key of the right node is more than the node. As long as the tree is balanced, the search path to each item is a lot shorter than that in a linked list.

**Search paths:**

------ ------ ------

Key List Tree

------ ------ ------

1 1 3

2 2 2

3 3 3

4 4 1

5 5 3

6 6 2

7 7 3

------ ------ ------

Avg 4 2.43

------ ------ ------

By larger structures the average search path becomes significant smaller:

------ ------ ------

Items List Tree

------ ------ ------

1 1 1

3 2 1.67

7 4 2.43

15 8 3.29

31 16 4.16

63 32 5.09

------ ------ ------

**TASK 1:**

My approach to the problem:

* Create a recursive function for all the kinds of functions that the task needs like Insert, Delete, and Print etc.
* Insert all the values according to the task
* Print them in sorted form using inorder output method
* Generate to random values using rand() function and store as data1 and data2
* And delete all the nodes of the tree.

**TASK 2:**

My approach to the problem:

* Create a recursive function for all the kinds of functions that the task needs like Insert, counting nodes, checking tree balance, checking if it’s a BST or not, Delete, Print etc.
* Use these functions in the main or the functions as given by default (in task 2) and completed the tasks.
* In the first Bst\_init function I’m creating an array of unique random values up to 109000.
* For the stat function I am doing couple of thing like checking whether it’s a BST or not with unique doc\_ids or not, Checking whether the tree is balanced or not, Getting the height of the tree, getting the total no. of nodes, checking if the no. of nodes inserted equals the total no of nodes in a tree and also the average of total comparisons per searches.
* Function used for insertion(Used AVL tree as it’s a self-balancing tree so keeping the tree balanced got simpler using this method):

struct Bstnode \*rightRotate(struct Bstnode \*y)

{

struct Bstnode \*x = y->left;

struct Bstnode \*T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right))+1;

x->height = max(height(x->left), height(x->right))+1;

return x;

}

struct Bstnode \*leftRotate(struct Bstnode \*x)

{

struct Bstnode \*y = x->right;

struct Bstnode \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right))+1;

y->height = max(height(y->left), height(y->right))+1;

return y;

}

struct Bstnode\* insert(struct Bstnode\* node, char\* name, int doc\_id, int word\_count)

{

if (node == NULL)

{

struct Bstnode \*temp = (struct Bstnode \*)malloc(sizeof(struct Bstnode));

temp->doc\_id = doc\_id;

temp->name = name;

temp->word\_count = word\_count;

temp->height = 1;

temp->left = temp->right = NULL;inserts++;

return temp;

}

else

{

if (doc\_id < node->doc\_id)

node->left = insert(node->left, name,doc\_id,word\_count);

else if(doc\_id > node->doc\_id)

node->right = insert(node->right, name,doc\_id,word\_count);

else return node;

}

node->height = 1 + max(height(node->left),height(node->right));

int balance = getBalance(node);

// Left Left Case

if (balance > 1 && doc\_id < node->left->doc\_id)

return rightRotate(node);

// Right Right Case

if (balance < -1 && doc\_id > node->right->doc\_id)

return leftRotate(node);

// Left Right Case

if (balance > 1 && doc\_id > node->left->doc\_id)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && doc\_id < node->right->doc\_id)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

* Function used for Searching:

int bstdb\_get\_word\_count ( int doc\_id )

{

struct Bstnode \*runner;

runner = root;

while (1)

{

cmp++;

if (runner == NULL)

{ cmp++;return -1;}

else if (doc\_id==runner->doc\_id)

{searches++;cmp++;return runner->word\_count;}

else if (doc\_id<runner->doc\_id)

{ runner = runner->left;}

else

runner = runner->right;

}

}

Searching for the book name is also done similarly.

* Function for Deleting:

void tree\_delete(struct Bstnode\* root)

{

if (root == NULL) return;

tree\_delete(root->left);

tree\_delete(root->right);

free(root);

}

* Function for Printing:

void inorder(struct Bstnode \*root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d %s %d %d\n", root->word\_count,root->name,root->doc\_id,++i);

inorder(root->right);

}

}

* STAT FUNCTION:

void bstdb\_stat ( void ) {

// inorder(root);

printf("\nSTAT:-\n~~~~~~~~~~~~~\n");

printf("\nAvg comparisons per search ->%f",(cmp/searches));

printf("\nList size matches expected? ->");

if(inserts==countnodes(root)) printf("Y\n");

else printf("N\n");

printf("\nCheck if it is a BST or not:-\n");

if(checkBST(root,NULL,NULL)==0) printf("It is a Bst.\nAnd all the values are unique.\n");

else printf("It is not a Bst\n");

printf("\nHeight of the tree: %d\n",height(root));

printf("\nPrinting the no. of nodes: %d\n",countnodes(root));

printf("\nCheck if BST is a balanced or not:-\n");

if(Balanced(root)) printf("It is balanced\n");

else printf("It is not a balanced\n");

printf("\n\n");

}

* Sample Output for Task2(only showed for Binary search tree):

Profiling bstdb

-------------------------------------------

Total Inserts : 106775

Num Insert Errors : 0

Avg Insert Time : 0.000001 s

Var Insert Time : 0.000000 s

Total Insert Time : 0.105597 s

Total Title Searches : 10677

Num Title Search Errors : 0

Avg Title Search Time : 0.000000 s

Var Title Search Time : 0.000000 s

Total Title Search Time : 0.005382 s

Total Word Count Searches : 10677

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 : 0.004391 s

STAT:-

~~~~~~~~~~~~~

Avg comparisons per search ->15.795355

List size matches expected? ->Y

Check if it is a BST or not:-

It is a Bst.

And all the values are unique.

Height of the tree: 20

Printing the no. of nodes: 106775

Check if BST is a balanced or not:-

It is balanced