NAMAN ARORA
19323369
Computer Engineering

REPORT FILE
ASSIGNMENT 3 : SEARCHING WITH TREES

Task 1 - Binary Search Tree

- Structure I created :

```
struct Node
{
    char key;
    struct Node *left;
    struct Node *right;
};
```

- **INSERTION** : For insertion of each element of "FLOCCINAUCINIHILIPILIFICATTION"
  The function insert(struct Node* , char) is used.

  What is does is checks the value of the key to be inserted, with value at the root.
  - ❖ If value at root is NULL, then i create a new node there using the same key using the fucntion newNode(char).
  - ❖ If its less than or equal to the root, I recursively call insert to the left of the root and put left of the root as the root.

```
node->left  = insert(node->left, key);
```

- ❖ If its more than the root, I recursively call insert to the right of the root and put right of the root as the root.

```
node->right = insert(node->right, key);
```

- FOR PRINTING THE TREE I USED INORDER TRAVERSAL, as it gives out the sorted order of the elements.

- **SEARCHING :**
  - Used random function to generate 2 alphabets
  - These alphabets are now passed to be searched.
  - If we get root as the key, I return the root.
  - If key of root is smaller than the search item I call search towards the right of the root
  - If key of root is greater than the search item I call search towards the left of the root.
  - Recursive calling of search is used.

Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

- **DELETING :**
  - ❖ Post order traversal of the tree is used to delete the tree.
  - ❖ First delete the left subtree
  - ❖ The delete the right subtree
  - ❖ Now the delete the root
  - ❖ This is also a recursive function.

Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

OUTPUT -

A
A
C
C
C
C
F
F
H
I
I
I
I
I
I
I
I
I
L
L
L
N
N

N
O
O
P
T
T
U

N in Tree

W not in Tree

deleting node A
deleting node C
deleting node C
deleting node A
deleting node C
…..and so on.

Task 2 : A Practical Application

- IDs of the books are being created in the ascending order of the insertion of the books, like the first book gets the id=0, id=1, asd so on.

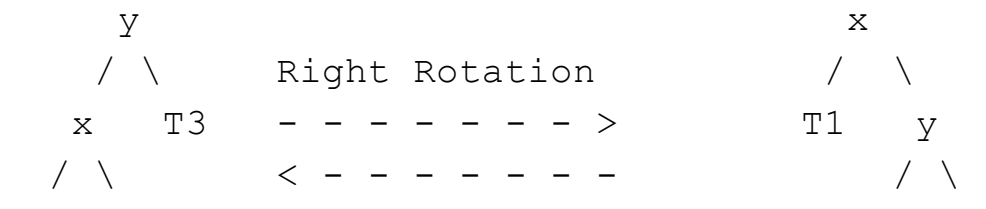
- **CONCEPT**
  - ❖ I have used the concept of AVL trees.
  - ❖ AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
  - ❖ Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree .

- **INSERTION :**

❖ To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- 1) Left Rotation
- 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on the left side) or x (on the right side)



Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

So BST property is not violated anywhere.

- **Steps for insertion**
- Let the newly inserted node be w
- **1)** Perform standard BST insert for w.
- **2)** Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- **3)** Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
- a) y is left child of z and x is left child of y (Left Left Case)
- b) y is left child of z and x is right child of y (Left Right Case)

- c) y is right child of z and x is right child of y (Right Right Case)
- d) y is right child of z and x is left child of y (Right Left Case)
- Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

(a) Left Left Case
(b) Left Right Case
(c) Right Right Case
(d) Right Left Case

- **IMPLEMENTATION :**

  The implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

  1) Perform the normal BST insertion.
  2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
  3) Get the balance factor (left subtree height – right subtree height) of the current node.
  4) If balance factor is greater than 1, then the current node is unbalanced and we are either in

Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in the left subtree root.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

- **COMPLEXITY :**
  The rotation operations (left and right rotate) take constant time as only a few pointers are being changed there. Updating the height and getting the balance factor also takes constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is the height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

OUTPUT-

```
Generating 105780 books... OK

Profiling listdb
-----------------------------------------

Total Inserts                 :    105780
Num Insert Errors             :         0
Avg Insert Time               :   0.000000 s
Var Insert Time               :   0.000000 s
Total Insert Time             :   0.018859 s
```

```
Total Title Searches          :         10578
Num Title Search Errors       :         0
Avg Title Search Time         :   0.000364 s
Var Title Search Time         :   0.000557 s
Total Title Search Time       :   3.854491 s

Total Word Count Searches     :         10578
Num Word Count Search Errors  :         0
Avg Word Count Search Time    :   0.000392 s
Var Word Count Search Time    :   0.000663 s
Total Word Count Search Time  :   4.148027 s

STAT
Avg comparisons per search  -> 52965.036349
List size matches expected? -> Y

Profiling bstdb
-----------------------------------------

Total Inserts                 :    105780
Num Insert Errors             :         0
Avg Insert Time               :   0.000000 s
Var Insert Time               :   0.000006 s
Total Insert Time             :   0.055523 s

Total Title Searches          :         10578
Num Title Search Errors       :         0
Avg Title Search Time         :   0.000001 s
Var Title Search Time         :   0.000000 s
Total Title Search Time       :   0.006194 s

Total Word Count Searches     :         10578
Num Word Count Search Errors  :         0
Avg Word Count Search Time    :   0.000001 s
Var Word Count Search Time    :   0.000000 s
Total Word Count Search Time  :   0.006890 s

STAT
```

```
Avg comparisons per search  -> 15.761297
List size matches expected? -> Y
Height of the tree  -> 17
Balanced Binary Tree  -> Y
Press Enter to quit...
```