AVL TREE

INSERTION AND DELETION OPERATIONS

A Design and Analysis of algorithm project by
Vikas Kalagi PES1201701654 and
Ganesha K S PES1201701731
From Class 4B

TREEs in Algorithms

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary:
The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.

```
   tree
   ----
    j    <-- root
  /   \
 f     k
/  \     \
a   h     z   <-- leaves
```

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess)

There are a lot many trees in data structures. They are:

Binary Trees

Binary Search Trees

AVL trees

B Trees (or general m-way search trees)

B+ Trees

Since our project deals with AVL trees we will speak about only AVL trees further,

AVL TREE

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.

WHY AVL TREE?

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(Log(n)) after every insertion and deletion, then we can guarantee an upper bound of O(Log(n)) for all these operations.

The height of an AVL tree is always O(Log(n)) where n is the number of nodes in the tree

OPERATIONS ON AN AVL TREE

1.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

Implementation

The following 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 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.

The following function does the insertion of a node in the AVL tree

NODE * insert (NODE * node, int key, char c [100], int (*fun_pointer) (NODE * node, int key, char c [100]));

Time 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(Log(n)). So, time complexity of AVL insert is O(Log(n)).

2. DELETION

Implementation

The following implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in 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 deleted node.
1) Perform the normal BST deletion.
2) The current node must be one of the ancestors of the deleted 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 Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
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 Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

The following function does the Deletion of a node in the AVL tree,

```
NODE * deleteNode (NODE * root, int key, char c [100], int
(*fun_pointer) (NODE * node, int key, char c [100]));
```

Time Complexity:

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

AVL TREE Structure:

There is a structure which is used for both integer and string matching. It is as shown below:

```
typedef struct Node

{       char a [100];
```

```
        int key;

        struct Node *left;

        struct Node *right;

        int height;

    } NODE;
```

Each node in our AVL tree contains

1) The variable a used for storing a string in the AVL tree
2) The variable key used for storing an integer value in the AVL tree
3) The left pointer and right pointer are used to move towards the left or right child of the current node in the AVL tree.
4) The height gives the height of the current node in the AVL tree.

The comparison of the input node with the integer or the string based on user's input is done by the following two functions,

```
int int_match (NODE * node, int key, char c [100]);

int str_match (NODE * node, int key, char c [100]);
```

We use these functions as callback in insertion, deletion and search functions.

The following function does the search and compare the input node:

```
void search (NODE * root, int key, char c [100], int (*fun_pointer)
(NODE * node, int key, char c [100]));
```