Advanced Data Structures COL702
Programming Exercise #1

S.N.Maheshwari

Anupam Sobti

2015ANZ8497

Contents

Problem 1

Explain the insertion/deletion and access operations in 2-3 Trees.

Data is maintained on leaf nodes and the discriminant values are maintained on the internal nodes of the tree. An internal node has two values, known as discriminant values. The first value represents the largest value in the left subtree and the second value represents the largest value in the middle subtree if it exists.
Search Operation: $Tree.search(x)$
The function returns two values. The first value indicates whether or not the element was found (True/-False). The second value is the position of the leaf containing x or the leaf containing the value just greater than x. If x is the greatest value in the tree, it returns the node whose child has the largest value in the already present tree. The algorithm states that the left child is traversed if $x < discrValue1$, else the middle child is traversed if $x < discrValue2$ if the discrValue2 exists. Otherwise, the right child is traversed. During the traversal if the node being traversed is a leaf, x is compared to the data in leaf. If found equal, the search returns True,leafAddress else it returns False,leafAddress.

Insert Operation : $Tree.insert(x)$
Base Case:
The first node is maintained as a leaf node. When the second value is inserted, one internal node is made the root which contains the smaller of the two leaves as the discriminant value.
Insertion : Case 1
Search for the node to be inserted. If the node is already present in the tree, the search returns (True,location of the node). In this case, there is no need for insertion since the element is already present.
Insertion : Case 2
The search returns the leaf node which is just greater than the element to be inserted, say y. Case 2 represents the case where the parent of the leaf node returned contains two children. In this case, the third child can be accomodated by the parent of y itself. Say the parent of y is p. Let's say the order of children of p is $c1 < x < y$. The middle child is therefore updated to x and the second discriminant value is updated to x. Hence, p now has three children and the insertion is complete.
Insertion : Case 3
The search returns the leaf y whose parent p has three children. Say the children of p are in the order $x_1 < x_2 < x_3 < x$. A new node is instantiated, say q. The nodes x_3 and x are made the children of q. The second discriminant value of p is removed and x_3 is made the first discriminant value for q. q is then recursively inserted into parent of p.
The data for non-leaf nodes is calculated as the largest value in the subtree rooted at the node. This ensures the discriminant values being maintained when recursively inserting nodes at non-leaf levels.

Deletion : $Tree.delete(x)$
Base Case :
The base case is when there are only two leaves rooted at a node and one of the leaves is deleted. In this case, the remaining leaf is made the root and the pointer to the root is returned as the pointer of the new tree.
Deletion : Case 1
Case 1 is the case where the search operation returns the node x whose parent has three children. In this case, one of the children can be deleted from the parent of x while mainintaining the rest of the structure. If x is the leftmost child, the middle child is made the left child and the discriminant value 1 is made equal to discriminant value 2. The discriminant value 2 is updated to None. If it is the middle child, the node is deleted and discriminant value 2 is updated to None. If it is the right child, the middle child is made the right child and the discriminant value 2 is made None.

Problem 1 continued on next page. . .

Deletion : Case 2
Case 2 is the case where the node whose child has to be deleted has only 2 children. In this case, the node either needs to merge with it's sibling (case 2.1) or borrow an excess leaf from one of it's sibling (case 2.2). In case 2.1, the parent of the node being deleted (x) has a sibling with only 2 children. Therefore, it cannot lend another child to the parent of x. Therefore, keeping the children of x's parent and it's siblings in order, the two sibling are merged to form a node with 3 children. The discriminant values are recalculated from the left and middle subtree (in this case, leaves). Since, the number of nodes has reduced, this has to be recursively done for the next level as well. In case 2.2, a sibling with three children dispenses one of it's children and the node from which x is deleted comes back to 2 children. The discriminant values have to be recalculated for both the nodes.

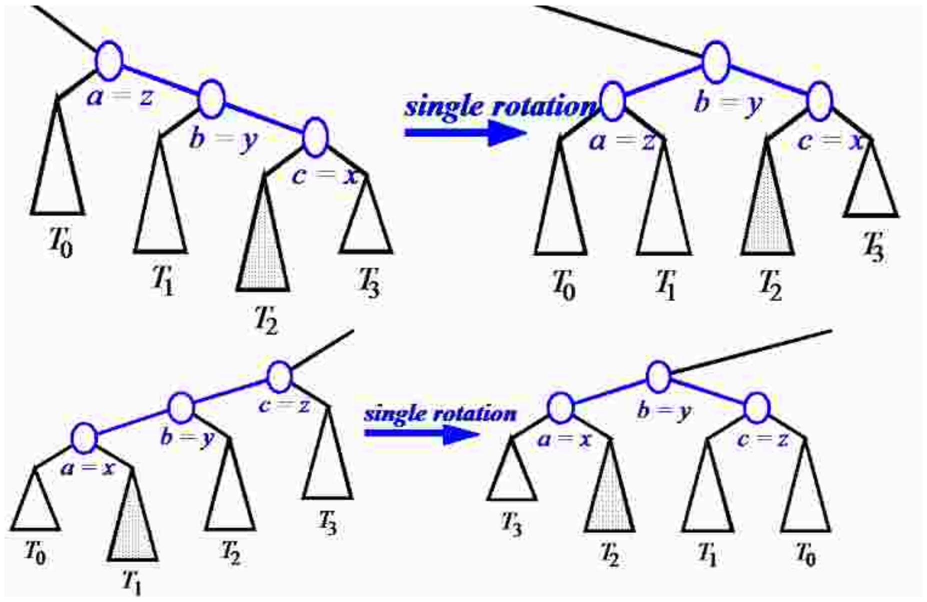
Problem 2

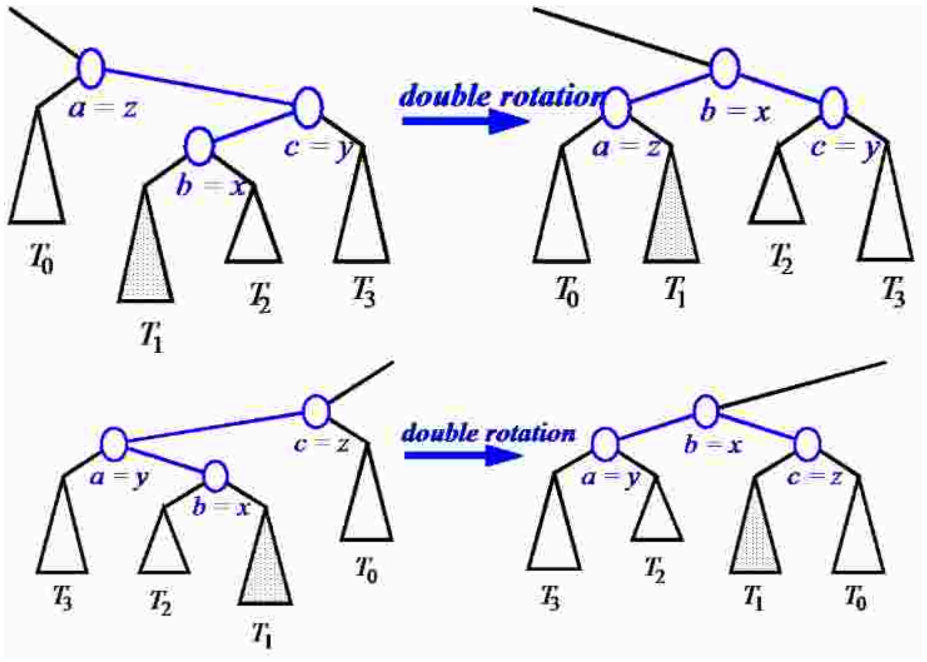
Explain the insert/delete/access operations in AVL Trees.

AVL Trees are said to be balanced if, for all the nodes in the tree, the difference in the heights of it's children is atmost 1. The heights of all the nodes are maintained on the nodes themselves as an attribute.

Search Operation : $Tree.search(x)$
The search operation is similar to a binary search tree. Till the element is not found or the node is null, make node it's left child if $x < value\ in\ node$ else make the node it's right child. Return true and the node reference if $x = value\ in\ node$ else return false and the node reference.

Insert Operation : $Tree.insert(x)$
The insertion in AVL Trees is done by simply following the search path and appending the node when a null is encountered. Thereafter, any imbalance in the nodes is rectified with the help of rotations. From the point of insertion, nodes are traversed upwards till the magnitude of difference in heights of the children of the node is at most 1. Let's say the difference is defined as the height of left child - the height of the right child. If the difference is positive, a right rotation has to be done (Rotations are discussed later) else if the difference is negative, a left rotation has to be done. The rotations are performed using three nodes, all of which exist on the path of traversal from the inserted node to the root. There are two types of rotations : Zig-zig and Zig-zag (both left and right). The rotations are illustrated in the figure below:





The heights of the nodes have to be updated after the rotation and the balance is restored.

Delete Operation : *Tree.delete(x)*