Assignment 3 Solutions, COMP 251B

Dipinder S. Rekhi

March 24, 2005

1 Exercise 12.1-5

Binary search tree can be thought of as a sorted structure that is created using comparisons only. This weakly implies that the lower-bound for creating such a tree in the worst-case is $\Omega(nlog\,n)$, because if it was asymptotically lower than that then it would mean that we'll have a comparison-based sorting algorithm with a lower-bound for the worst case asympotically smaller than nlog n. However, it is not the case, as proved in the book. For a more rigorous proof, we would use decision trees. Consider a decision tree for a comparison-based algorithm that creates a binary search tree from a given sequence on n numbers. The inner-nodes of this tree refer to the comparisons, and the edges emanating from those nodes refer to different paths the algorithm takes based on the comparison made. The leaf nodes refer to the BSTs formed after following one particular path in the tree. Since there are n! ways of arranging n input numbers, it means that there are n! possible BSTs (assuming that no two numbers are the same). In other words, the decision tree would have n! nodes. The decision tree would be a binary tree because a comparison of two non-identical numbers at each inner-node will result in either the first one being greater than the other or vice-versa. Let the length of the longest path from root to the leaf be h. 'h' would also be the height of the decision tree. Any binary tree of height h would have at the most 2^h leaves. So,

$$n! \leq 2^h$$

$$log\,n! \leq h$$

$$h = \Omega(nlog\,n)$$

Hence, in the worst case constructing a BST would take $\Omega(nlog\,n)$ time.

2 Join Operation on Red Black Trees

a) To update bh[T] in $O(log\, n)$ time within RB-INSERT and RB-DELETE functions, we make use of the property that number of black nodes from root to any leaf is the same. After RB-INSERT and RB-DELETE do their usual work, we travel from root to one of the leafs by following the left most path (we can follow any path). Along the way we count the number of black nodes encountered. The root is not counted. The number of black nodes encountered becomes the new value of bh[T]. Since the height of RB-tree is $O(log\, n)$, therefore this operation takes $O(log\, n)$ time. Hence, it dosent increase the asymptotic running time of RB-INSERT or RB-DELETE.

Calculating the black height of the nodes while descending down the tree is simple. Given the black height of the parent, the childs black height depends on its color. If it is black then the black height is one less than its parents, whereas, if it is red then its black height is the same as its parents. Therefore, it takes $O(1)$ to do this.

b) Given $bh[T_1] \geq bh[T_2]$. Consider the right most path is the tree T_1. Right most path means that when descending down the tree starting from the root, we always visit the right child. When we go down any path in a RB-Tree, the black height of the nodes decrease (see part a to convince yourself). It decreases from $bh[T_1]$ down to zero, such that all the values in this range belong to atleast one particular node in the descending path. The point of following the right most path is that we'll encounter the largest values. So while descending the right most path, when we encounter a black-node of black height h, it means that it is the largest black node in the tree with a black height h. Using the above reasoning, we descend down the right most path of tree T_1, until we encounter the black node of height $bh[T_2]$. Since the height of RB-tree is $O(log\, n)$, therefore this operation takes $O(log\, n)$ time.

c and d) These parts follow part (b) of the question and fit in the whole scheme of performing a join operation. Consequently, we can safely assume that y is a black node and tree subrooted at y has the same black height as tree T_2. Put node x in y's position. Make node y the left child and root of T_2 the right child of x. Color node x red so that properties 1,3 and 5 are maintained. To enforce properties 2 and 4 assign $z = x$, $T = T_1$ and execute RB-INSERT-FIXUP(T,z). To get a better understanding of how this works,

please refer to the book. This operation runs in $O(log\, n)$ time.

e and f) Part e is obvious. When $bh[T_1] < bh[T_2]$, the method is exactly the same except that we follow the left most path in tree T_2 to find the smallest black node that has the same black height as tree T_1. After that a similar procedure follows as in the above parts. Join operation is actually a sequence of the above steps. Since these steps are performed in time $O(log\, n)$, therefore Join operation can be performed in time $O(log\, n)$.

3 AVL Trees

a) Let $n(h)$ be the number of nodes in the AVL tree of height h. $n(0) \geq 0$ and $n(1) \geq 1$ are easy to imagine. Now consider a AVL tree of height $h > 1$. The root will have two subtrees where 1) either both the subtrees have a height h-1 or 2) one has a height h-1 and the other has the height h-2. If both the subtrees are of height h-2 then the given tree cannot be of the height h. The above reasoning follows the AVL tree property. So the number of nodes in the above tree would be at least $n(h-1) + n(h-2)$. Or

$$n(h) \geq n(h-1) + n(h-2)$$

The above recurrence is a fibonacci recurrence (see problem 4-5, page 87 of the book). $n(h-1) + n(h-2)$ is the hth fibonacci number F_h, where

$$F_h = \frac{(\frac{1+\sqrt{5}}{2})^h}{\sqrt{5}}$$

or

$$n(h) \geq \frac{(\frac{1+\sqrt{5}}{2})^h}{\sqrt{5}}$$

Taking log on both the sides (let $n(h) = n$)

$$log\, n \geq h * log(\frac{1+\sqrt{5}}{2}) - log(\sqrt{5})$$

or $h = O(log\, n)$

b) Given, a subtree rooted at x whose right and left subtrees are height

balanced, but x itself is not height balanced. Lets consider the case where left subtree is of the height h and right subtree is of the height h+2. Similar method should be followed for the opposite case with orientations changed from left to right and vice-versa. Let L be the left child and R be the right child of x. Let α and β be left and right children of R. If $height(\alpha) \leq height(\beta)$ then $LEFT - ROTATE(T, x)$ else if $height(\alpha) > height(\beta)$ then first $RIGHT - ROTATE(T, R)$ and then $LEFT - ROTATE(T, x)$ where T is the tree immediately after the insertion. Set the heights accordingly.

c) Following is the INSERT code

```
INSERT(x, z)
-       if(x = NULL)                    (Root)
-               root = Z
-               return
-       if(key[z] ≥ key[x])
-               if(RIGHT[x] = NULL)
-                       RIGHT[x] = z
-                       h[x] = newheightofx
-                       return
-               else
-                       INSERT(RIGHT[x], z)
-       else
-               if(LEFT[x] = NULL)
-                       LEFT[x] = z
-                       h[x] = newheightofx
-                       return
-               else
-                       INSERT(LEFT[x], z)
-       h[x] = max(left[x], right[x]) + 1
-       if(NOT - BALANCED(x))
-               BALANCE(x)
```

d) Consider the function INSERT (see above) running on an AVL tree. Since we go down from root to find a node whose child the new node z would become, it takes $O(log\, n)$ time. This is because height of the AVL tree is $O(log\, n)$. Now consider the path from root to the newly inserted node z. Let the first node (going up the path from z) that is imbalanced be x. If there

are other nodes up the path that are imba;anced then it would be because subtree at x is imbalanced (as a node got added to it). So if we fix this imbalance at x then we would not need to perform any more rotations. The new height of x after the insertion is h+1 where h was the height before insertion was done (we are considering the case where addition of a new node icreases the height of some nodes. otherwise we would not have to worry about imbalances). Now, after we perform a rotation at x, the height of the node that will be in x's position would go back to 'h'. Hence, the balance is achieved. If you look carefully at how balance works you would see that the height of the node in x's position would not stay the same, but decrease by one. This is because by just adding a single node we dont get into the situation where α and β have the same height.

4 Planning a Company Party

Let C_T be the total cost of the tree rooted at T. In this tree, if T is invited to the party then none of its children is invited. However, if T is not invited then some of its children may and some may not be invited. Let C_{Ti} be the cost of the tree rooted at T when T is invited. Let C_{Tn} be the cost of the tree rooted at T when T is not invited. It is obvious that

$$C_T = max(C_{Tn}, C_{Ti})$$

In turn,

$$C_{Ti} = \sum_{x \in Children\, of\, T} C_{xn}$$

Also,

$$C_{Tn} = \sum_{x \in Children\, of\, T} C_x$$

For the node x, of the final optimal tree, that is a leaf, $C_{xi} = C(x)$ where $C(x)$ is the conviviality rating of x and $C_{xn} = 0$. Using this and the above equations we can build an optimal tree in a bottom-up fashion.

5 Moving on a Checker Board

Let your current position on the board be x. If x belongs to the topmost row then the game is over and print the total. Otherwise, let Y be the set of positions that you can go from x. Let C_x be the maximum number of dollars that one can earn following a certain optimal path from x. Then following recusion holds,

$$C_x = max_{z \in Y}(p(x, z) + C_z)$$

Calculating C_x this way and finding the optimal path can be time consuming due to redundancy in recusion. Many problems have the same subproblems that are recalculated. To avoid this we start from the next to top most row. Calculate the C_x values for all the nodes x in this row. C_x values for all the nodes in the topmost row is zero. Go down one row and do similar calculations. Carry on like this until you get to the bottom most row. If the board is of the dimensions NxN, then these calculations can all be done in time $O(N^2)$.