ESO211 (Data Structures and Algorithms) Lectures 20 to 22

Shashank K Mehta

1 Binary Search Trees

A Binary Search Tree (BST) is a data structure to store a set in which each element is associated with a unique key and the set of all keys are totally ordered (i.e., all pairs of keys can be compared with each other). For example the set is all IITK students and the key is their roll numbers.

This data structure is a binary tree in which each node stores one element of the set. We will assume the following notations. Let x be a pointer to a node. The $x.value$ denotes the stored element, $x.key$ is the associated key, $x.parent$ is the pointer to the parent node, $x.left$ points to the left child-node and $x.right$ points to the right child-node.

The rule of storing the elements in a BST is that for any node pointed by x in the tree, $x.key$ is greater than the keys of all the elements in the left subtree and it is smaller than the keys of all the elements in the right subtree. Note that since each element has a unique key and a BST stores a set, all keys are distinct. See the example of a BST below. ¡figure¿

1.1 Search

The algorithm for searching an element with key k in a BST is as follows. Assume that *root* points to the root of the BST.

```
x := root;
found := false;
while x ≠ nil AND ¬found do
    if k = x.key then
        found := true;
    else
        if x.key < k then
            x := x.left;
        else
            x := x.right;
        end
    end
end
if found then
    return x.value
else
    return Not Found;
end
```

Observe that the process traverses from the root node to a leaf node at most. Hence the worst

case time complexity of searching in a BST is the $O(tree - depth)$ where the tree depth is the length of the longest path from the root to any leaf node (actually the cost is $1 + depth$ since we check each node on the path, while the depth is the number of edges on the longest path.)

1.2 Insertion

Suppose an element is stored in a node pointed by z which is to be inserted in a BST pointed by $root$. The the insertion process is given as follows.

```
if root = nil then
|  root := z;
|  Return;
end
x := root;
found := false;
while x ≠ nil AND ¬found do
|  if k = x.key then
|  |  found := true;
|  else
|  |  if x.key < k then
|  |  |  y := x;
|  |  |  x := x.left;
|  |  else
|  |  |  y := x;
|  |  |  x := x.right;
|  |  end
|  end
end
if ¬found then
|  if x = y.left then
|  |  y.left := z;
|  else
|  |  y.right := z;
|  end
|  z.parent := y;
end
```

It is easy to see that essentially this procedure is same as the search procedure. Hence its cost is also $O(depth)$.

1.3 Deletion

Suppose x points to a node in a BST. The entry of this node is to be deleted. Then the procedure for deletion is as follows.

Once again the time complexity of this procedure is $O(tree - depth)$.

```
if x.right ≠ nil then
|   z := x.right;
|   while z.left ≠ nil do
|   |   z := z.left
|   end
|   x.value := z.value;
|   x.key := z.key;
|   if z = z.parent.left then
|   |   z.parent.left := z.right
|   else
|   |   z.parent.right := z.right
|   end
|   z.right.parent := z.parent;
else
|   if x = root then
|   |   root := x.left
|   else
|   |   if x = x.parent.left then
|   |   |   x.parent.left := x.left
|   |   else
|   |   |   x.parent.right := x.left
|   |   end
|   |   x.left.parent := x.parent;
|   end
end
```

1.4 Balanced BST and Balancing

Let T be a binary tree of depth d (number of edges on the longest path from root to a leaf). Then it can contain at most $2^{d+1} - 1$ nodes. Hence $d \geq \log_2(n+1) - 1$. Thus the depth cannot be less than $\log_2 n$. We have seen above that all the operations in a BST have time complexity $O(tree - depth)$. Hence if the tree depth is minimum possible, then we will have time complexity $O(\log_2 n)$ for search, insertion, and deletion.

This motivates us to define the notion of balanced trees. A tree family is said to be balanced if the depth in these trees is at most $c.\log_2 n$ for some constant c. If a BST is balanced and if we restructure after each insertion and deletion so that it remains balanced, then we call them balanced BST. In the next section we will discuss one such family of balanced BST called Red-Black trees.

2 Red-Black Trees

Definition 1 *A* red-black tree *is a regular binary search tree in which each node is assigned a set-element. We think of* nil *as a* terminal node*. In implementation we may or may not actually place such a node. In addition each node, including terminal nodes, is also assigned a color subject to the following conditions.*

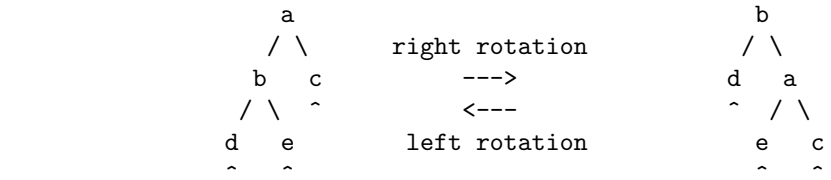
1. *each node is colored Red ot Black*
2. *Root node and all terminal nodes are colored black*
3. *both children of a red node are black*
4. *from each node all paths to terminal nodes in its subtree have same number of black nodes.*

Definition 2 *The* black height *of a node, a, in a red-black tree is the number of black nodes on any path from a to a terminal node in its subtree.*

Lemma 1 *If a red-black tree has n internal nodes (non-terminal), then its depth is at most* $2.log(n+1)$*. Hence it is a balanced tree.*

Proof Suppose the black-height of the tree is b. Then the longest path from the root to any terminal node has length at most $2(b-1)$ because the first and the last nodes are black and the path length is the number of edges on it. Therefore the depth of the tree is at most $2b-3$ since the last node is a terminal node. We know that a binary tree can have at most $2^{depth+1} - 1$ nodes. Since the tree has black depth b, its depth is at most $b-2$ because the terminal node does not count and the depth is the number of edges on the path. If we remove all the red nodes and shrink the tree by connecting each black node with the next black node above it, then we find all leaf nodes at depth $b-2$. Although there may be some node with more than 2 children in this shrunk tree, no node except the leaf node has fewer than 2 children. So the number of nodes in the tree cannot be less than $2^{b-1} - 1$. So $\log_2(n+1) \geq b-1$. So $depth \leq 2b-3 \leq 2\log_2(n+1) - 1$. ■

Definition 3 *Left/Right rotation*



here ^ *stands for a subtree.*

Next we describe the algorithms for search, insertion, and deletion in an R-B tree.

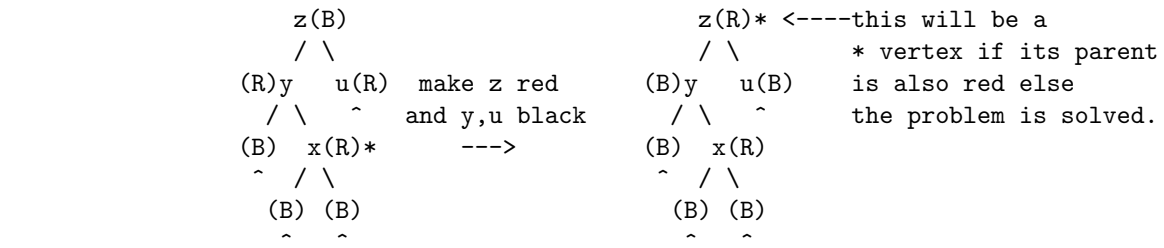
1) **Search** same as in any BST. So the complexity is $O(log(n+1))$ where n is the number of internal nodes.

2) **Insertion** Insert as usual. Color the new node by Red. The new tree may have one problem: two consecutive Red-nodes, i.e., successive Red-node problem. Label the lower of the two successive Red nodes by * (asterix).

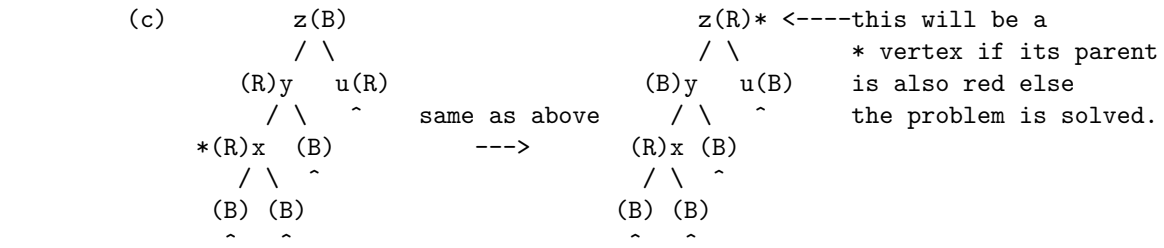
Solution of successive Red-node problem (y=upper Red node and x=lower Red node). We put asterix at the lower node at the problem site.

(a) If y=tree-root then simply color it with Black.

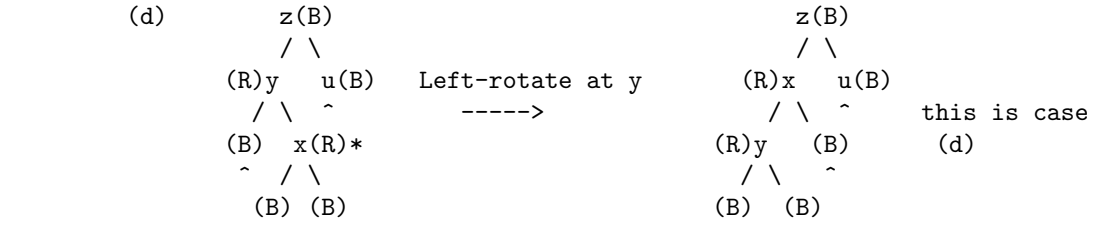
(b)

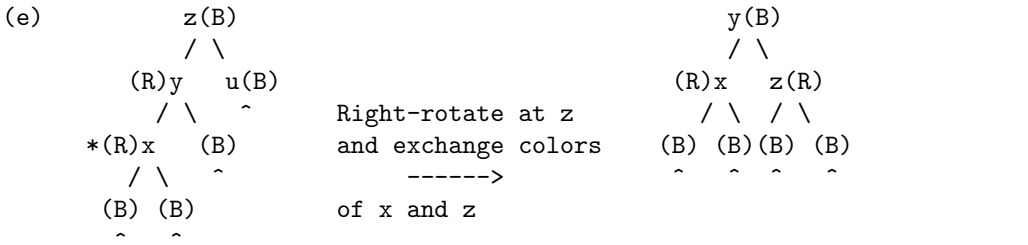


(c)



(d)





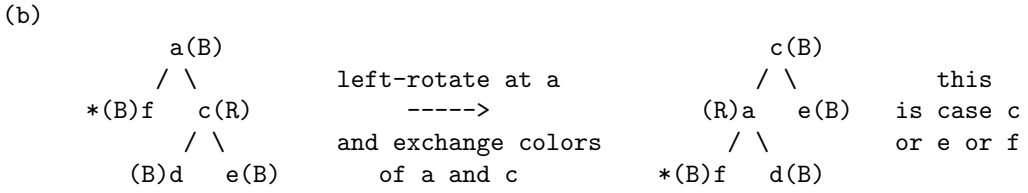
So the complexity is $O(log(n+1))$.

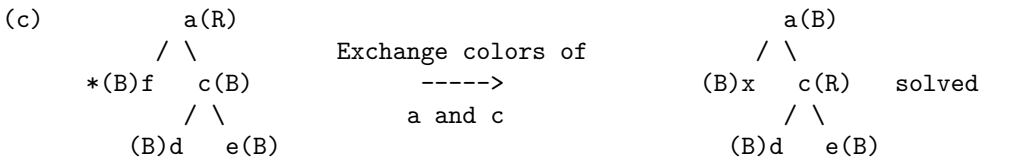
Deletion

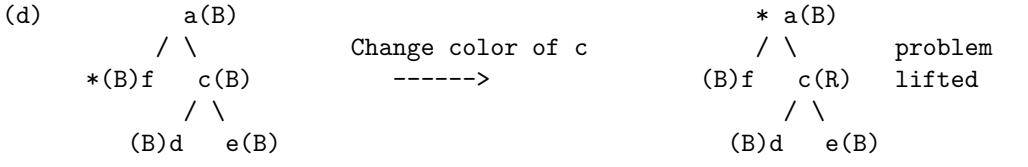
In this case also we delete the node as discussed for general binary tree. If the deleted node is Red, then observe that the resulting tree is still a valid Red-Black tree. The problem arises when the deleted node was Black. In the new tree at the location of the deleted node we have one black-depth less. We place * at this node.

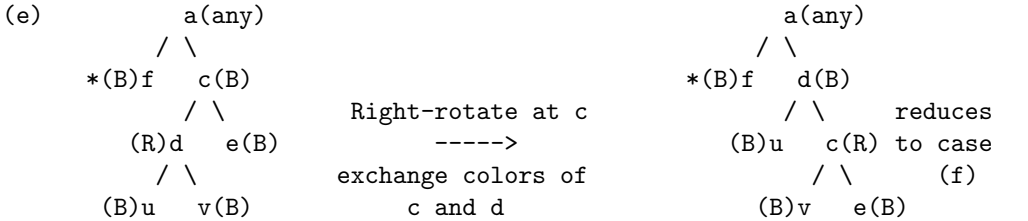
Problem: Given a tree with label * at a node such that it satisfies all conditions of RB tree except that the subtree at * has one black length shorter.

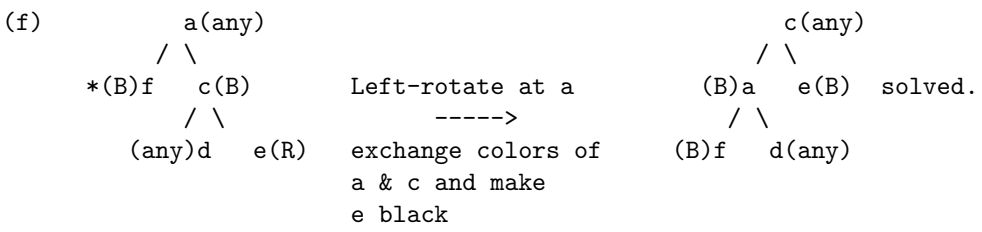
(a) If * is Red, then re-color it to Black.











See that it terminates and the complexity is $O(log(n+1))$.