ESO211 (Data Structures and Algorithms) Lecture Notes Set 7

Shashank K Mehta

1 Binary Search

If a subset S of a universal set U is stored in an array, then how to find whether any given element x present in S or not? Clearly the search from one end of the array to the other will take $O(|S|)$ time. But can we improve the time using the fact that the elements of S are stored in an array in which any location can be accessed in constant time?

Interestingly, the search can be made very fast if U is a totally ordered set and elements of S are stored in the array in sorted order (increasing or decreasing). Suppose elements of S are stored in increasing order from $A[0]$ to $A[n-1]$. Also suppose we are searching an element x. Then we will first check if $A[\lfloor n/2 \rfloor] = x$? If yes, then we have found the desired element. Otherwise if $A[\lfloor n/2 \rfloor] < x$, then we know that x cannot be in $A[0 : \lfloor n/2 \rfloor]$. Similarly, if $A[\lfloor n/2 \rfloor] > x$, then we know that x cannot be in $A[\lfloor n/2 \rfloor : n-1]$. This method reduces the search space by half in just one comparison.

In general, if we are searching x in the range $A[i : j]$, then we must first check if $A[i+\lfloor (j-i)/2 \rfloor] = x$? The search must continue till the search space reduces to a single location. Algorithm 1 describes binary search method.

The time complexity of binary search is $O(\log\ n)$ because after each comparison we reduce the search space by half so after $\log\ n$ comparisons the search space will shrink to a zero or one elements. This is a tremendous gain over $O(n)$. But if the set S if dynamic, i.e., elements must be added / deleted to / from S, then it will be very expensive to maintain the elements in sorted order in contiguous locations in the array. While insertion and deletion are very cheap on a linked list (provided a pointer is available to the position where this operation to be performed) we cannot perform binary search on it.

So the challenge for us is to design a data structure in which binary search method can be used to seach and insertion / deletion operations be as convenient as in a linked list. Such a data structure is indeed possible.

2 Binary Search Trees (BST)

A Binary Search Tree (BST) is a data structure to store a set in which each element is 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 sub-tree of the node and it is smaller than the

```
Input: Array A containing n elements in non-decreasing order in contiguous locations from a
       to a + n − 1 and element x to be searched.
i := a;
j := a + n − 1;
while j > i do
    m := ⌊j − i⌋;
    if A[i + m] = x then
        return "At location i + m;
    else
        if A[i + m] < x then
            i := i + m + 1;
        else
            j := i + m − 1;
        end
    end
end
if j = i AND x = A[i] then
    return "At location i;
else
    Not found;
end
```

keys of all the elements in its right sub-tree. 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 in figure **??**.

i

2.1 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.

Observe that the search process traverses from the root node to a leaf node. Hence the worst case time complexity of searching in a BST is the $O(\textit{tree-depth})$ where the tree depth is the length of the longest path from the root to any leaf node (actually the cost is $1 + \textit{depth}$ since we check each node on the path, while the depth is the number of edges on the longest path.)

2.2 Insertion

Suppose an element is stored in a node pointed by z and a pointer *root* points to a BST. Then Algorithm 3 shows how to insert the node into this BST.

It is easy to see that essentially this procedure is same as the search procedure. Hence its cost is also $O(\textit{tree-depth})$.

2.3 Deletion

Suppose the root of a BST is pointed by a pointer *root* and a pointer x points to some node in this tree. Then Algorithm 4 shows how to delete the node pointed by x from the BST.

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

```
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
```

Algorithm 1: $Search(root, k)$

We started out to design a data structure which gives the advantage of binary search tree and in which insertion and deletion operation are efficient. We have found that all three operations in a BST cost $O(tree\text{-}depth)$. If the depth of the tree is itself $\Theta(n)$, then thus data structure would fail to serve its purpose. In the next section we will describe a way to keep tree depth to its minimum.

2.4 Balanced Binary Search Tree

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 $n \leq 2^{d+1} - 1$ or $d \geq \log_2(n+1) - 1$. Thus the depth of the tree cannot be less than $\log_2(n+1) - 1$. 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 it a balanced BST. In the following subsection we will discuss one such family of balanced BST called *Red-Black trees.*

3 Red-Black Trees

Definition 1 *A* red-black tree *is a binary search tree in which each node is assigned a data element. We think of* nil *as a* terminal *node or* leaf node *which will not have any data. Remaining nodes will be called* data-nodes. *In implementation we may not actually place a terminal node. The* depth *of a red-black tree is the length of the the longest root-to-leaf path. Note that this will be one more than actual depth because terminal node is not an actual node. In addition each node, including terminal nodes, is assigned a color subject to the following conditions.*

```
if root = nil then
    root := z;
    Return;
end
x := root;
found := false;
while x ≠ nil AND ¬found do
    if z.key = x.key then
        found := true;
        return "Element already present in tree";
    else
        y := x;
        if x.key < z.key then
            x := x.left;
        else
            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
```

Algorithm 2: $Insert(z, root)$

```
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
```

Algorithm 3: $Delete(x, root)$

1. *each node is colored Red or Black*

2. *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 sub-tree 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 sub-tree.*

Lemma 1 *If a red-black tree has n data-nodes, then its depth is at most* $2.log(n+1)$*. Hence it is a balanced tree. In addition the black height of the root is at most* $1 + \log_2(n+1)$*.*

Proof Suppose the black-height of the root is b. We will consider two cases (i) root node is black, and (ii) the root node is red. Let D denote the depth of the tree (the number of edges on the longest path from the root to any terminal node). Recall that this is one more than the actual depth because terminal node is not a real node.

Case (i): The minimum depth occurs when the tree has no red node and the maximum depth occurs when every alternative node is red on at least one path from the root to a leaf. So $b-1 \leq D \leq 2(b-1)$.

Let us now determine a relation between n and b. The number of data nodes will be minimum when the tree will have no red node. So each root-leaf path will have b nodes. So $2^{b-1}-1 \leq n$. The maximum number of internal nodes will be possible when every root-leaf path will have red vertices alternately. So $n \leq 2^{2b-2}-1$. Hence $b-1 \leq \log_2(n+1) \leq 2(b-1)$. Combining the two inequalities we get $(1/2).\log_2(n+1) \leq D \leq 2.\log_2(n+1)$.

Case (ii): In this case the minimum depth and minimum internal nodes will occur when the root is the only red node. On the other hand the maximum depth will occur when at least one root-leaf path has b red nodes and maximum internal nodes will occur if every root-leaf path has b red nodes. So in this case $b \leq D \leq 2b-1$.

The relation between n and b can be determined by using the fact that both child sub-trees of the root have black roots. Let data nodes in them be n_1 and n_2. So $n = n_1 + n_2 + 1$. The black depths of both child sub-trees will remain b. So using the results of Case (i), the lower bound for n is $1+2.(2^{b-1}-1) = 2^b-1$. The upperbound for n is $1+2(2^{2b-2}-1) = 2^{2b-1}-1$. Thus $b \leq \log_2(n+1) \leq 2b-1$. Combining the two inequalities gives $(1/2)(1+\log_2(n+1)) \leq D \leq 2.\log_2(n+1)-1$.

Then combining the two cases we get $(1/2).\log_2(n+1) \leq D \leq 2.\log_2(n+1)$. The upperbound establishes that a red-black tree is a balanced binary search tree.

For the second part of the lemma, combine the bounds for b in terms of n in the two cases. It combines to $(1/2)(\log_2(n+1)+1) \leq b \leq 1+\log_2(n+1)$. ■

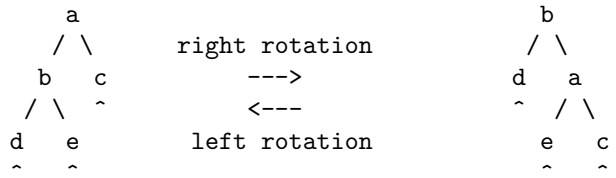
Corollary 1 *Search operation on a red-black tree has time complexity* $O(\log_2(n+1))$ *where n denotes the number of data nodes in it.*

3.1 Operations on a Red Black tree

If we perform insert or delete operations on a red-black tree as described in the last section, then the tree will no longer remain a red-black tree. Hence we have to devise an efficient methods to perform these operations.

We begin by describing a fundamental step which is used in every balanced binary search tree to re-balance an off-balance BST. It is called *rotation*.

Definition 3 *Left/Right rotation*



here symbol '∧' stands for a sub-tree.

Observe that if a pointer is provided to the root of the sub-tree, then it takes $O(1)$ time to perform left or right rotation at that sub-tree. Next we describe the algorithms for search, insertion, and deletion in an R-B tree.

(1) **Search**

The search process same as in Algorithm 2. So the complexity is $O(log_2(n+1))$ where n is the number of data nodes.

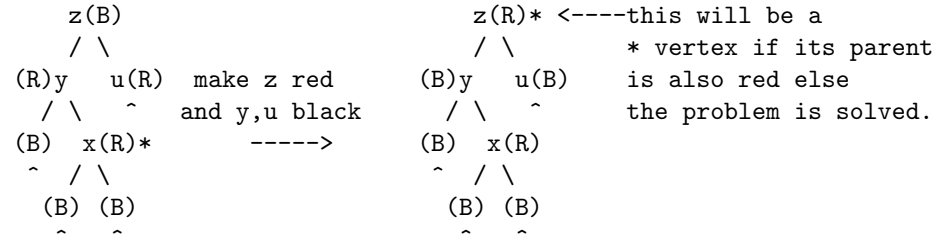
2) **Insertion**

Perform insertion using Algorithm 3. Color the new node by Red. But the new tree may have one problem: two consecutive Red-nodes defect, i.e., successive red-node problem at one location. Next we describe a method to fix this defect.

Suppose a distorted red-black tree has one defect. It has a node y which is coloured red and one of its child nodes is x and that too is red. Following step describes the fixing process. The defect of two consecutive red nodes is classified into 5 cases. For each case a fixing step is described below. These steps should applied till the tree becomes defect-free. We will show that this method will fix the tree in $O(\log(n+1))$ time. We have put a '*' (an asterisk) at the lower of the two consecutive red-nodes to highlight the defect. Symbol '∧' denotes a sub-tree.

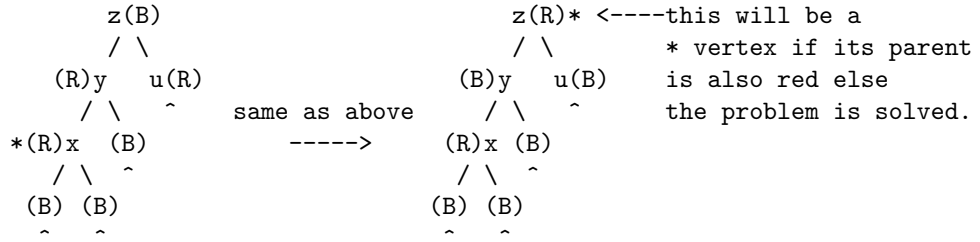
Case (a): If vertex y is the tree root then simply color it with Black. This will result into a healthy red-black tree.

Case (b): In this case both child nodes of z (parent of y) are red. Further y is left and x is right child or its mirror image (y is right and x is left child).



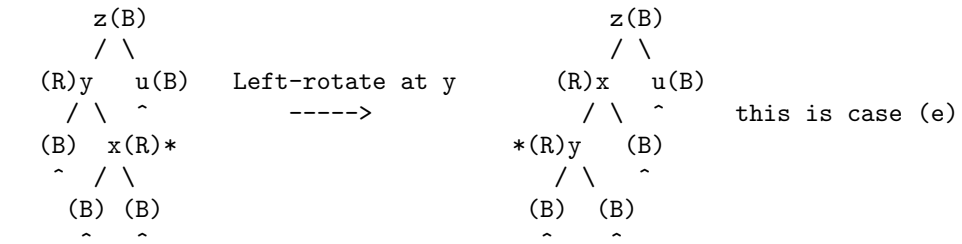
If z is the tree root or if its parent is black, then the resulting tree is healthy red-black tree. Otherwise it will have one defect of two consecutive red nodes. But in this case the depth of starred node reduces by 2.

Case (c): In this case both child nodes of z (parent of y) are red. Further y and x are left children or its mirror image (both are right children).

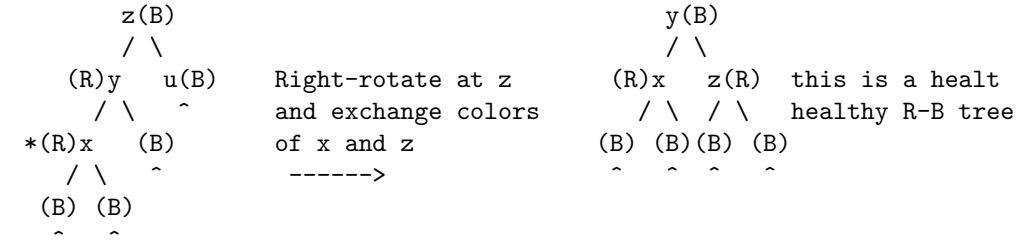


If z is the tree root or if its parent is black, then the resulting tree is healthy red-black tree. Otherwise it will have one defect of two consecutive red nodes. But in this case the depth of starred node reduces by 2.

Case (d): In this case z has one red and one black child. Further y is left and x is right child or its mirror image (y is right and x is left child).



Case (e): In this case z has one red and one black child. Further y and x both are left children or its mirror image (both are right children).



In each case either the defect is fixed in at most two steps or the depth of the starred vertex reduces by 2. Hence the defect will get fixed in $O(2 + depth/2) = O(depth) = O(\log_2(n+1))$ time. The time for insertion is also $O(\log_2(n+1))$. Hence the total time complexity of insertion is $O(log(n+1))$.

(3) **Deletion**

In this case also we delete an item as discussed earlier, i.e., as in Algorithm 4. Suppose the item to be deleted is in some node x. Recall that in this algorithm, as the first step, we overwrite the leftmost item of the right sub-tree of x on to x. Here we repeat this step without changing the colour of x. The second step is also followed exactly as in Algorithm 4.

Suppose the parent of the deleted node is g, its parent is a and its only child (right-child) is f. Also suppose the right child of a is c. After the second step f becomes the left child of a. Initially the colours of all the vertices are kept unchanged.

Observe that if the deleted node g is red, then the resulting tree is still a valid Red-Black tree because the black depth of each node remains unchanged.

The problem arises when the deleted node was Black. In this case the black height of f is one less than the black height of c. The defect is that the black height of f is one short. We place * at this node to hightlight this defect.

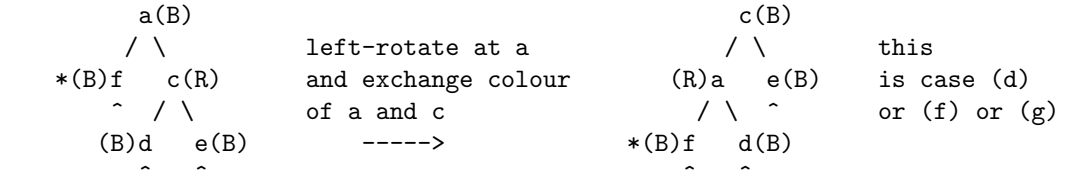
Problem: Given a tree with label * at a node such that it satisfies all conditions of RB tree except that the subtree at * has one black length shorter.

Case (a) If the starred node is the root of the tree, then the tree is free from any defect.

Case (b) If starred vertex is red, then re-color it to black. This will repair the tree.

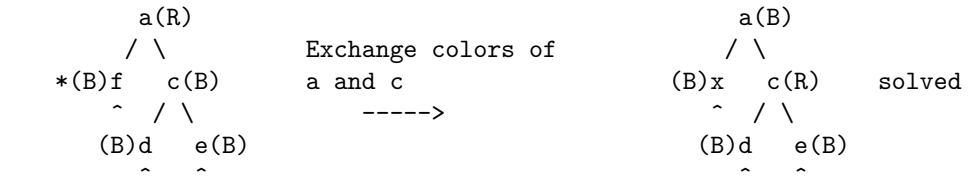
In all the remaining cases f is black. We split them into 5 cases. In the first case a is black and c is red. In the next case a is red and c is black. The remaining case is when both, a and c, are black. This case is split into 3 cases based on the colours of the children of c.

Case (c)

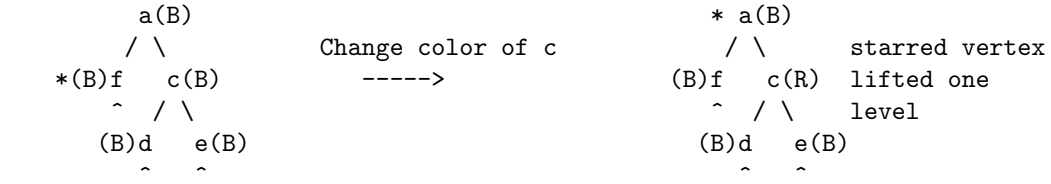


This step reduces to Case (d) or (f) or (g).

Case (d)

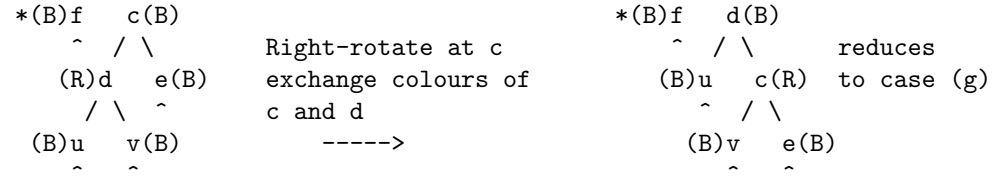


Case (e)

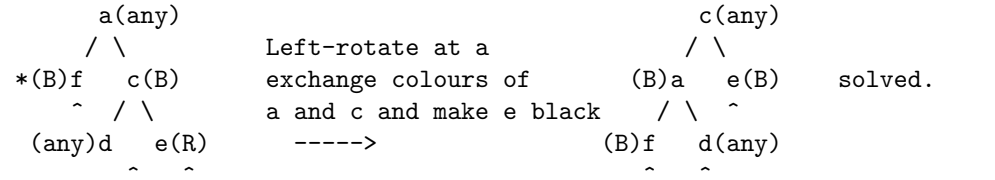


After this step the black height of the starred node increases by 1.

Case (f)



Case (g)



In this algorithm after at most 3 steps either the tree is fixed or the starred vertex rises by one level, i.e., its black height increases by 1. Since the black height of the root is at most $\log_2(n+1)$, the total number of steps will not be more than $3.\log_2(n+1)$. The time complexity of the deletion as well as that of repairing is $O(log(n+1))$. So the over all time complexity is $O(\log_2(n+1)$.