Binary Search Trees and Balanced Binary Search Trees

Chapter 12: Up to (including) section 12.3; section 12.4 is out.

Chapter 13.

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Search trees are data structures that support the following **dynamic-set** operations.

(1) Search
(2) Minimum
(3) Maximum
(4) Predecessor
(5) Successor
(6) Insert
(7) Delete

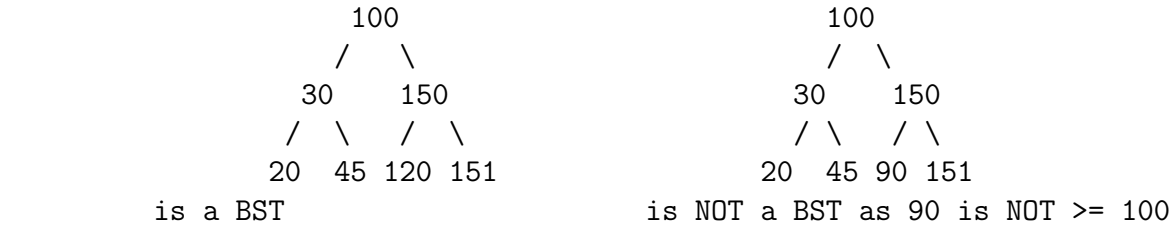
Such operations could run in time proportional to the height h of the search tree i.e. the running time is $O(h)$. If the search tree is a binary tree with every node having two children and all leaves at the same level (sometims called a complete binary tree or full binary tree), whose height is $\Theta(\lg n)$, then the running time becomes $O(h) = O(\lg n)$, where n is the size of the search tree (i.e. number of nodes), before the operation is performed on the tree. In general, however, there is no guarantee that a search tree is such that its height is $O(\lg n)$. If the tree, for example, is a linear chain, the same operations take $\Theta(n)$ worst-case running time.

Since a search tree supports all these seven operations, it can also be used to implement a **priority queue** or a **dictionary**.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A binary search tree T is a search tree that is organized as a binary tree. Such a tree is represented by a linked data structure every node x of which is an object with a **key** field, **left** and **right** pointers pointing to the left/right children of x (or NULL if a child does not exist), and a pointer to the **parent** of x. For $r = root(T)$, $parent(r)$ =NULL. Each node of **a binary search** tree T has the **binary-search-tree** property.

Binary Search Tree property. *Let x be any node of a binary search tree. For every node y of the left subtree of x we have key(y) $\leq$ key(x). For every node y of right subtree of x we have key(y) $\geq$ key(x).*



Given a binary search tree we can sort its elements in $O(n)$ time. The way we achieve this is by listing the elements of the tree by an **inorder tree walk/traversal** where, the left subtree of the root is recursively visited first, then the root, and then the right subtree of the root. One can define similarly **preorder** and **postorder** tree walks.

```
Inorder(x)  //To print all elements of T call Inorder(root(T))
1. if x != NULL
2.   then   Inorder (left(x))
3.          print(key(x))
4.          Inorder (right(x))
```

Running time is $O(n)$ as number of calls is linear in the number of nodes and each call with parameter x prints x.

Exercise. Since we can inorder traverse a BST T in $O(n)$ and thus list its keys in sorted order, does this mean that sorting can be realized in $O(n)$ time if we do the following: (a) build a binary search tree on the n input keys (b) perforam an inorder traversal in $O(n)$ time that lists the keys in sorted (i.e. ascending order).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A. Implementing a **BSTSearch** operation in such a tree.

```
 BSTSearch(x,value)
1. if (x==NULL) or (value == key(x))
2.   return(x);
3. if value < key(x)
4.   return   BSTSearch(left(x),value);
5. else
6.   return   BSTSearch(right(x),value);
```

B. Implementing a **BSTMax** (similarly for BSTMin) operation in such a tree that returns the element with Max/Min key.

```
 BSTMin(x) /* returns an element not a key */
1. while (left(x) != NULL)
2.  x = left(x);
3. return(x);
```

A BSTMax operation traverses T through right(x) pointers.

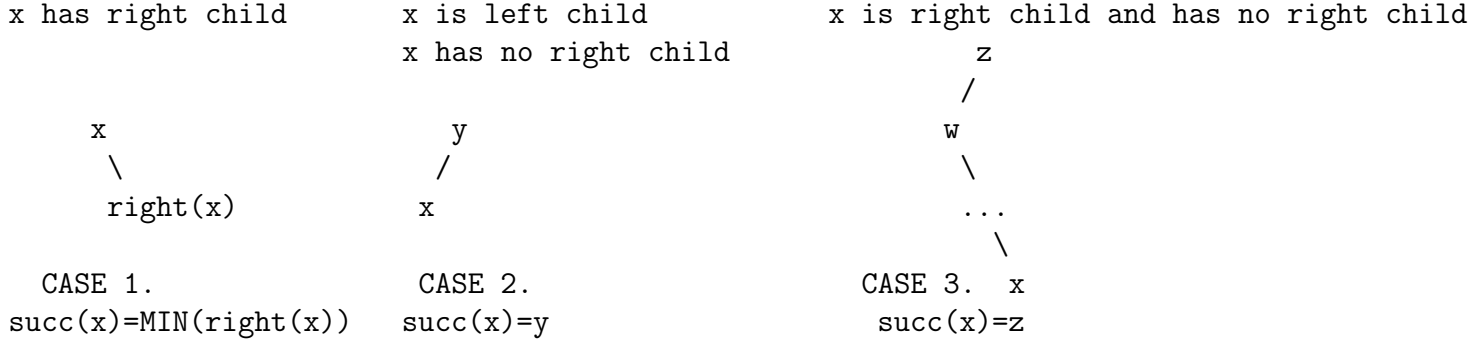
Running time for all operations is $O(h)$. In terms of n (number of nodes in the subtree rooted at x) this is $O(n)$ in the worst case.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Successor Operation

C. **BSTSuccessor** (similarly for BSTPredecessor) operation returns the element which is the successor of x (i.e. returns the element whose value is the smallest one among the keys whose values are greater than key(x)).

```
 BSTSuccessor(x)
1. if (right[x] != NULL) return(BSTMin(right(x)));
2. y=parent(x);
3. while (y !=NULL) and (x==right(y)) {
4.    x=y
5.    y=parent(y); }
6. return y;
```



(1) **x has a right child**, then successor(x) is in the tree rooted at right(x) (i.e. it is the element with minimum key of that subtree and is found in line 1 with a call to BSTMin). (2) **Otherwise x has no right child. Let x be the left child of its parent y**. Then the successor of x is its parent y (loop 3-5 is not executed and y is returned). (3) **Otherwise x has no right child and x is the right child of its parent y**. We move upwards until we find an ancestor w of x that is the left child of his parent. If such a node exists then succ(x) = parent(w)=z; if not node x has no successor (i.e. it is the maximum).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

D. **BSTInsert** inserts a value v into a binary search tree T rooted at $r = root(T)$. In fact we insert a node z such that z.key $= v$, z.left=NULL, z.right=NULL. BSTInsert works similarly to BSTSearch by traversing a path of T starting at the root until it reaches the node where it is to be inserted.

```
  BSTInsert(T,z)
1. y=NULL; x=root(T)
2. while x!=NULL {
3.   y=x;
4.   if key(z) < key(x)
5.     x=left(x);
6.   else
7.     x=right(x);
8. }
9. parent(z)=y;
10 if y==NULL
11.  root(T)=z;
12.else {
13.  if key(z) < key(y)
14.     left(y) = z;
15.  else
16.     right(y)=z;
   }
```

Running time is $O(h)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

E. **BSTDelete** deletes a node z from BST T and takes as an argument a tree T and a pointer to the node to be deleted z (i.e. we do not need to search for z). Three cases related to a deletion operation are considered.

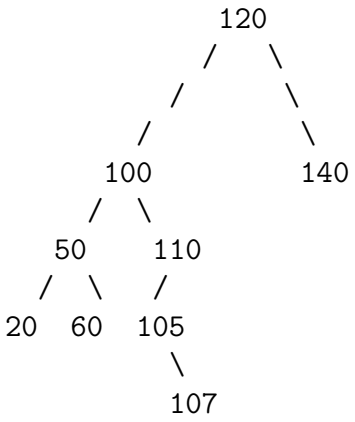
(1) z has no children. Splice out z so that left(parent(z)) or right(parent(z)) point to NULL and not to z.

(2) z has a single child. Then we spliceout/remove z by making its parent(z) point to child(z) rather than z.

(3) z has two children. Splice out successor(z) and replace the contents of z (z.key) with those of successor(z) (successor(z).key). [The predecessor(z) can ALSO BE USED in deletion.]

```
  BSTDelete(T,z)
1. if (left(z)==NULL || right(z) ==NULL)
2.      y=z;
3. else y=successor(z); // y is node to splice out
4. if (left(y)!= NULL)
5.      x=left(y);
6. else x=right(y); // node to be attached to parent(y)
7. if (x !=NULL)
8.   parent(x)=parent(y);  //splice out y in lines 7-13
9. if (parent(y)==NULL)
10.     root(T)=x;
11.else if (y==left(parent(y))
12.          left(parent(y))=x;
13.     else right(parent(y))=x;
14.if (y !=z)  // we work on the successor of y
15.  key(z)=key(y) //copy successor(z).key into z.key
16.return y
```

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.



Conclusion All tree operations have worst-case running time that is proportional to the height h of tree T. For a tree T of height h with n nodes this is $T(h) = O(h)$, if we expreess running time in terms of h. If we express running time as a function of n, as we usually do, $T(n) = O(h) = O(n)$ as a tree with n nodes has maximum height $h = n - 1$.

Case 1. Delete($z = 60$). z has no children. Remove z by making right(parent(z))=NULL deletion is completed.

Case 2. Delete($z = 50$) After 60 is deleted, 50 has only one child. Case 2 applies. Splice out 50, by making left(100)=left(50).

Case 3. Delete($z = 100$). Node 100 has two children 20 (after the two previous deletions) and 110. Find succ(100) in its right subtree, which is 105, replace key of node 100 with the 105 key, and splice out node 105 (effectively make 107 the left child of 110).

NOTE. In case 3, we can use equally well the predecessor of z instead of the successor z.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Although BSTrees look interesting, their worst case time performance is no better than that of an array or a doubly linked list. However, BSTs perform better on the average, as on the average case with random insertions, the height of the formed BST is $O(\lg n)$.

In the following sections we are going to introduce classes of binary search trees whose worst case performance is optimal, i.e. the worst case running time of any of the seven introduced operations is $O(\lg n)$ independently of any assumptions on the distribution of the keys. A BST on n nodes whose height is $O(\lg n)$ is called **balanced**.

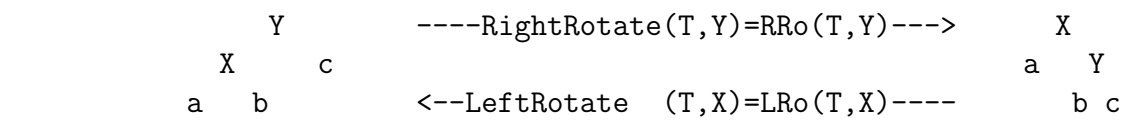
For such trees query or probe operations such as Sucessor, Predecessor, Search, Minimum, Maximum work the same way as the operations we have just described, i.e. no modification is required. For Insert and Delete, however, since these operations modify the structure of the tree the original Insert and Delete may not work properly as they may leave the tree imbalanced at the completion of the operation. Additional steps are required to guarantee that the tree is balanced after the operation Insert or Delete is completed.

A red-black tree is a balanced Binary Search Tree. In a red-black tree, each node is colored red or black with certain restrictions on how these colorings are performed. In a red black tree one bit of extra information needs to be maintained (0 for red, 1 for black are values for this bit), the color of a node. The C++ Standard Template Library implements for example a Red-Black Tree data structure.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Introduction

The Insert and Delete operations on red-black trees require that a certain rotation be performed on a subtree of the tree. Left and Right rotations are defined in picture format as follows.

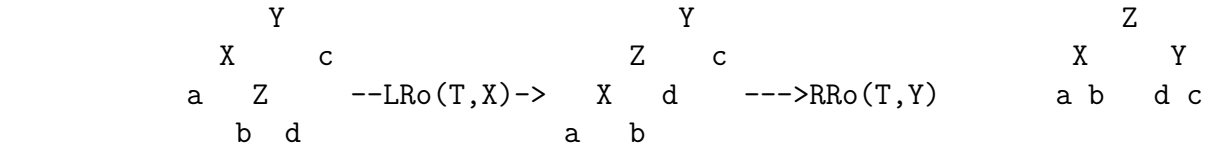


a,b,c, are subtrees (or NULL)

1. Single (left and right) rotations.

Rotation operations take constant time $O(1)$ to perform and can easily be described through pointer assignments of a $O(1)$ number of nodes.

A more complex operation is a double rotation shown below in terms of two rotations.



2. A Double (Right) Rotation.

A class of balanced BSTs is the red-black tree. A Red-Black Tree, remains balanced during Insert and Delete operations by using one extra bit per node. The extra bit signifies the color of the node which can be either Red or Black. Therefore the fields of a red-black tree node are **c, l, r, p, key**, where c is the color field l, r are the left and right child fields, p the parent field, and *key* is the key field of the search tree. If a left or right child does not exist the corresponding field contains NULL which is considered as an external node of the tree: in such a case a NULL node has r and l pointers with some distinct values (e.g. -2). The following property(ies) is(are) maintained as an invariant in a red-black tree. It(They) is(are) known as the **red-black tree property(ies)**.

RB0 Every node is colored either red or black.

RB1 The root is colored black and every leaf (i.e. NULL node) is represented in an rb tree and is also colored black.

RB2 If a node is red then both its children are black, i.e. there are no consecutive r nodes.

RB3 For each node, all node-to-descendant-leaf paths contain the same number of black nodes, or equivalently, all root-to-descendant-leaf paths contain the same number of black nodes.

A coloring of the vertices implies also a coloring of the edges of an r-b tree. Edge (u, v) is colored with the color of v.

Height vs Black-Height

Black-height bh(x) of a node x, is the number of black nodes on ANY PATH from x to a descendant leaf (but not including the color of x).

By property RB3 the black height of a node is well defined.

Note the difference between height and black-height. The height of a node x is $h(x)$. The black-height of a node x is $bh(x)$.

The height of a tree $h(T)$ is the height $h(r)$ of its root $r = root(T)$. What is the relationship between $h(r) = h(root(T))$ and $bh(r) = bh(root(T))$?

Theorem RB0. For any two paths P and Q from root r to a descendant leaf we have that $length(P) \leq 2length(Q)$.

Proof of RB0. Let $bh(r) = r(r) = A$. Then in the path from the root to a leaf (which are NULL nodes colored black), there are A black nodes, i.e. the minimum length of such a path is A. However, there may be red nodes as well. But each one red node must be followed by a black one due to RB2. Therefore there may be in a path as many as A red nodes as well. Therefore the maximum length of such a path from r to a leaf is $2A$. This proves the Theorem.

Theorem RB1. $bh(r) \geq h(r)/2$, where r is the root of a red-black tree T.

Proof of Theorem RB1. We know that $h(r) \leq 2bh(r)$ from Theorem (not Property) RB0. This shows that $bh(r) \geq h(r)/2$.

(c) Copyright A. Gerbessiotis. All rights reserved.

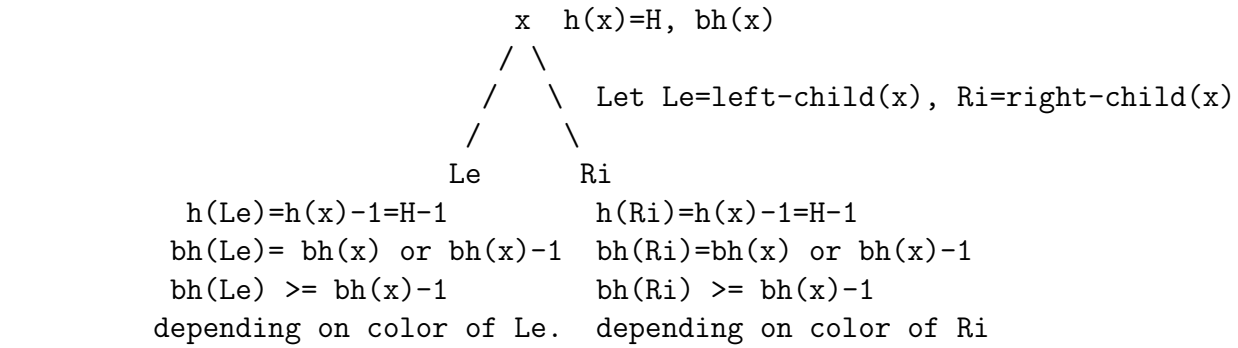
Lemma RB 2. Any subtree rooted at node x of a red-black tree has at least $2^{bh(x)} - 1$ internal nodes (null leaves don't count!).

Proof of Lemma RB3. (BY INDUCTION on the height H of x, $h(x) = H$).

Base Case. If height of x is 0, i.e. $H = 0$, then x is leaf in the red-black tree (i.e. it is a NULL). Then, $bh(x) = 0$ and since x is a leaf, the subtree at x has no internal nodes. However $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ and the basis of the induction is true.

Inductive Hypothesis. If the statement of the Lemma is true for all nodes x of height up to $H - 1$, then it is true for any node of height H.

Inductive Step. Consider node x of height $h(x) = H$ and black-height $bh(x)$.



By the inductive hypothesis the left subtree of x rooted at Le has at least $2^{bh(Le)} - 1 \geq 2^{bh(x)-1} - 1$ internal nodes and the right subtree rooted at Ri has at least $2^{bh(Ri)} - 1 \geq 2^{bh(x)-1} - 1$ internal nodes. Therefore the tree rooted at x has at least $1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) = 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ internal nodes. This proves the induction and the lemma.

Alternative proof of Theore RB2: If the root r has black height $bh(r)$, it means that there are no NULL leaves at distance $bh(r)-1$ or less from the root. This means that there are only internal nodes in level 0 (one node, the root) through level $bh(r)-1$, with 2^i nodes at level i, i.e. a total of $2^{bh(r)}-1$ internal nodes).

Theorem RBHEIGHT A red-black tree on n internal nodes has height $h(r) \leq 2\lg(n+1)$.

Proof of Theorem RBHEIGHT. Let $r = root(T)$. By Theorem RB1 we know that $bh(r) \geq h(r)/2$.

By Lemma RB2 we know that the tree rooted at r has at least $2^{bh(r)}-1$ nodes. Therefore $n \geq 2^{bh(r)}-1$. Since $bh(r) \geq h(r)/2$, we have that $n \geq 2^{h(r)/2}-1$. Therefore

$$\begin{aligned} n &\geq 2^{bh(r)}-1 \geq 2^{h(r)/2}-1 \\ n &\geq 2^{h(r)/2}-1 \\ n+1 &\geq 2^{h(r)/2} \\ \lg(n+1) &\geq h(r)/2 \\ 2\lg(n+1) &\geq h(r) \end{aligned}$$

An immediate consequence is that all probing operations requiring $O(h)$ time in a red-black tree, where h is the height of the tree, which translates into worst-case running time of $O(\lg n)$. We then prove the same for modifying operations such as Insert and Delete.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Insertion of a node z in an r-b tree is similar to insertion in a BST tree. When a new node is inserted in an r-b tree, we need to assign it a color. The choice of color that causes the fewest problems is RED, unless the inserted node becomes the root of the tree (i.e. tree was empty). Therefore,

Case 1. If node z is inserted into an empty tree, we color z BLACK, and make z the root of the tree.

Otherwise, the tree is not empty and,

Case 2. We perform the standard BST-Insert operations and color z red.

Note that coloring the z node BLACK would have definitely caused a violation of the RB3 property. Coloring z red may cause a violation of the RB2 property, if the parent of z in the tree is colored red. Therefore we need to fix the coloring problem of z in the case that both z and $p(z)$, the parent of z, are both colored RED. Towards this we need to apply a function FIX(z) to fix the RED color of z.

Note: If FIX(z) applies to the root, which was somehow, colored red in a previous call of FIX, then this application of FIX colors the root BLACK and exits.

The possible cases that may need to be examined when calling FIX(z) are classified based on z, its parent $pz = p(z)$, its grandparent $gz = p(p(z))$ and z's uncle uz which is the sibling of pz. We label these cases as XYc where,

(a) X is L or R depending on whether pz is left or right child of its gz,

(b) Y is L or R depending on whether z is the left or right child of its parent pz, and

(c) finally c is r or b depending on whether the uncle of z, i.e. uz is r or b.

Cases XYr cause recolorings. Cases XYb cause rotations. A change of a color may cause an RB2 violation higher in the tree (on the path to the root). In such a case a fixing on z reduces to fixing the color of gz (gz takes the place of z), i.e. of a node higher in the tree.

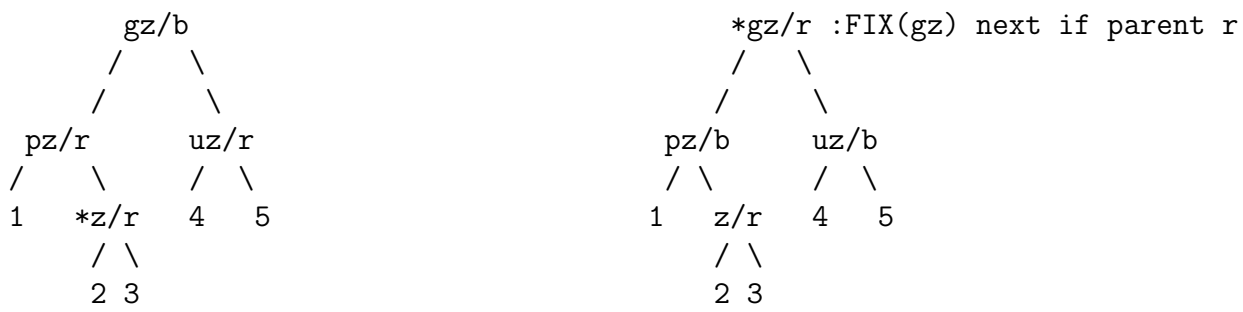
In all cases below z and pz are colored RED originally.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

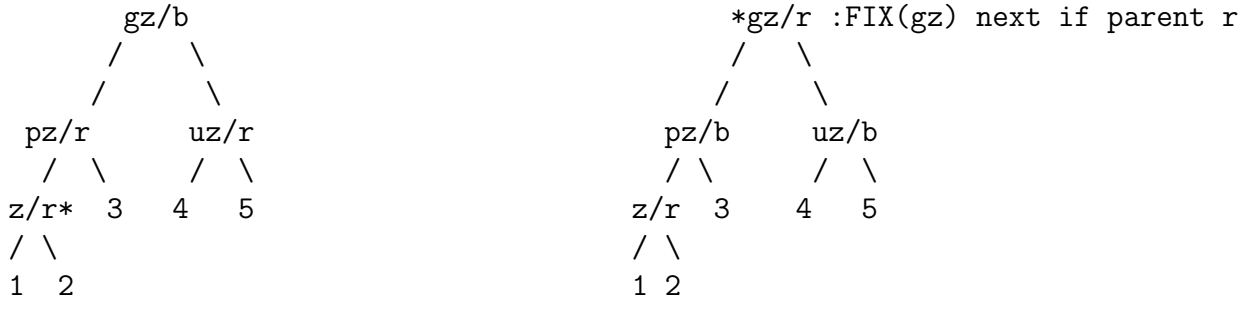
Case 1: LLr, LRr and RRr, RLr

Case 1. The first case involves the Insertion subcases **LRr and LLr** which are shown. Cases **RRr and RLr** are not shown but are symmetric. These cases require node recolorings only. Note that if gz is the root its color cannot change; this causes an increase to the blackheight of descendant nodes. FIX may cause a total of $O(\lg n)$ recursive calls higher in the tree.

```
// LRr, LLr shown (RRr, RLr  symmetric and not shown)
Case 1a: LRr ::gz b->r; gz and p(gz) may become red; Call Fix(gz) next.
```

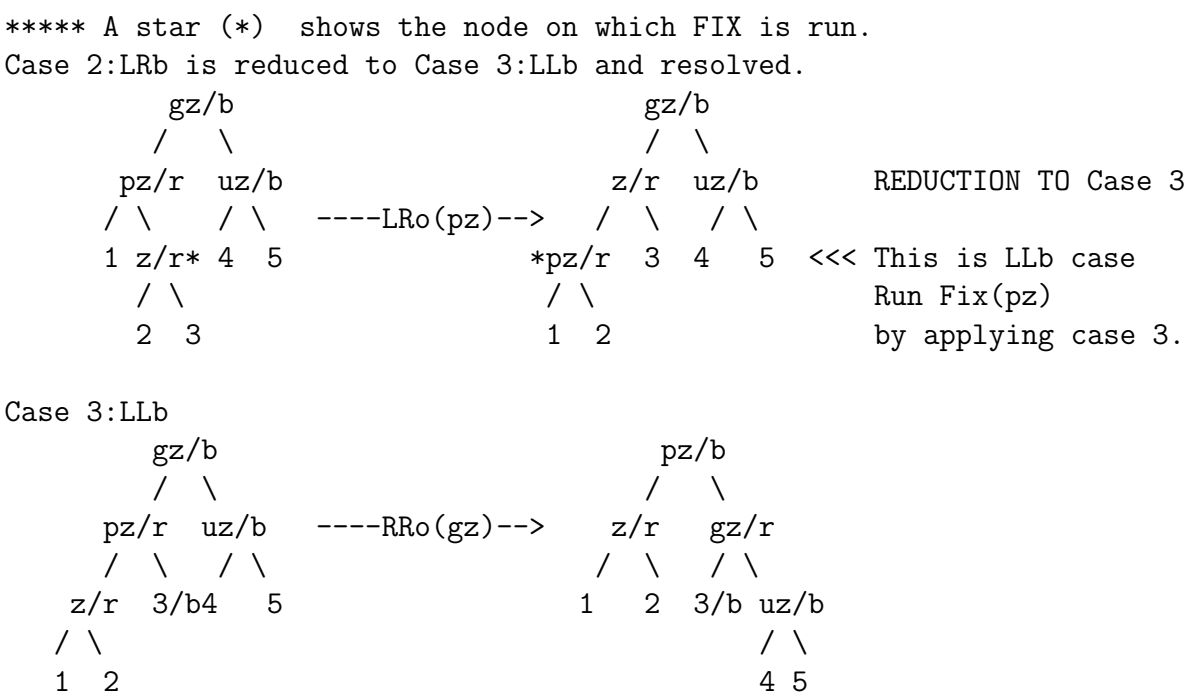



```
Case 1b: LLr :: Same as before (gz: b-->r)
```



Case 2: LRb, and Case 3: LLb

Case 2 covers the case of LRb, and Case 3 the case of LLb. Case 2 is ALWAYS FOLLOWED by Case 3. RLb and RRb are symmetric(not shown).



After the single rotation is performed there can be no way that there are two consecutive RED nodes in a path from the root to pz (root of the subtree in Case 3). Therefore after a single rotation we are done.

Conclusion. Insertion requires $O(\lg n)$ recolorings (Case 1) and $O(1)$ rotations (Cases 2 and 3).

Deletion in an r-b tree is identical to a Deletion in a BST tree. When we perform Delete(z), a node is spliced out; this node is called x. If z has no or one child, then x is z otherwise x is the successor (or the predecessor) of z. In any of these three cases we call y the only child of x, and $p(y)$ the new parent of y which was previously the parent of x.

1. If x is red, then y must be black and $p(y)$ must be black or otherwise x should have been black. Splicing out x causes no violations whatsoever.
2. If x is black, we have a violation of RB3. We distinguish the following subcases.
   3. If y is red, we recolor y black and the violation is resolved.
   4. If y is black and y becomes the root of the tree, no RB3 violation occurs, because all the paths from the root y will have black height one less.
   5. If y is black but not the root, we have a violation of RB3 that can not be resolved immediately. We "transfer" the BLACK color of x to y by coloring y DOUBLE-BLACK. We then need to fix y by calling FIXDELETE(y), i.e. a node calling FIXDELETE is a node "colored" BLACK twice.

Only case 5 needs resolution. Towards this, we consider y its parent $py = p(y)$ and and $v = sibling(y)$.
The cases that need to be considered are classified as Xcn, where X is L or R, c is r of b for Red and Black respectively, and n is 0, 1, or 2.

- If y is the right child of py imbalance is of type R otherwise it is of type L.
- Since all paths through y are one black node deficient, v can not be a NULL node, i.e. it has children. The color of v determines whether we have Lb or Rb if v is black or Lr and Rr if v is red.
- The number of a black v's red children determines whether we have $Rb0, Rb1, Rb2$ or $Lb0, Lb1, Lb2$ (we also have Lr0 and Rr0).

The $**$ below indicates the node y and $*$ its sibling for the particular case. As a result of a case, another case may apply, and then, the nodes playing the role of y and $sibling(y)$ are also indicated by $**$ and $*$ respectively.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Review of Cases 1, 2, 3,4

All four cases described below handle the cases LYn, where Y and n can vary. The RYn cases are treated similarly.

Case 1. This handles case Lr0. Since $v = sibling(y)$ is red it has black children, i.e. Lr1 and Lr2 cases are not possible since a red node can not have red children. Under this case py and v exchange colors following a left rotation and one of cases 2a, 3, 4 is subsequently entered and the operation is thus terminated after one (case 2a or case 4) or two steps (case 3 followed immediately by case 4), with the role of y played by y but the role of v played by the left child of v. Although y moves lower in the tree one level, since Case 2 follows Case 1, the process terminates.

Case 2. This handles the case Lb0. If Case 2 is entered through Case 1, py has color red, and then py and v switch colors (Subcase 2a). If however py has color black, py plays the role of y and y moves closer to the root. In summary, Case 2 is either a terminal case or the FIXDELETE process moves closer to the root.

Case 3. This handles an instance of Lb1, called Lb1rb where v has a red left child and black right child. This case is transformed through a rotation to an instance of Case 4 since Lb1rb is transformed into case Lb1br. Case 3 is followed by Case 4 that is terminal.

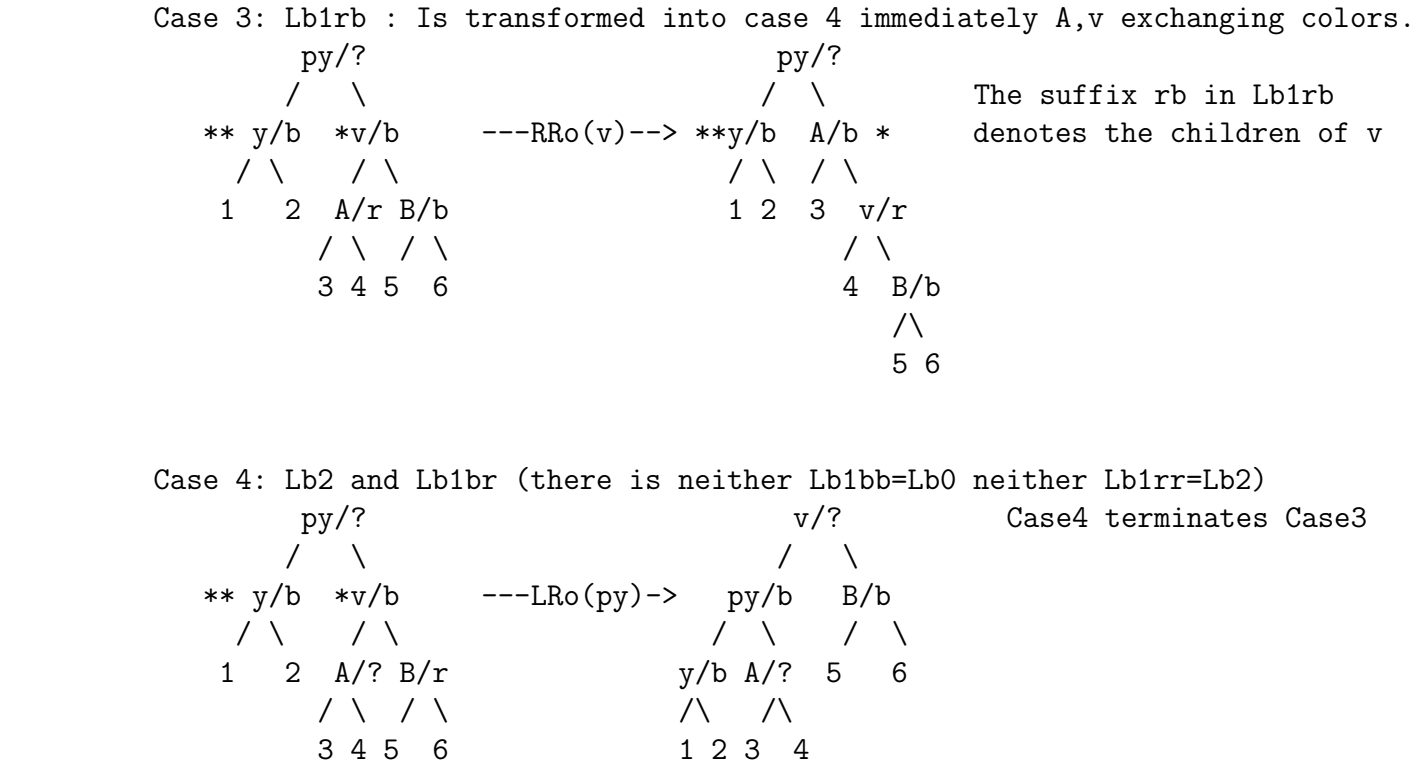
Case 4. This handles cases Lb1br and Lb2. Note that Lb1bb does not make sense as it is Lb0 (Case 2), and Lb1rr also does not make sense as it is Lb2 (examined in Case 4). A left rotation and color change gets rid of the extra black color of y. The case then terminates.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

```
Comment: Lr1 and Lr2 are not possible; a red node CANNOT have red children.
Case 1: Lr0 : A left rotation is performed and then Case 2a or 3 or 4 applies.
         py/b                              v/b      REMARK 1:
        /   \                             /   \       [ v and py switch colors]
   ** y/b  *v/r     ---LRo(py)-->    py/r   B/b
           / \                       / \    / \
          A/b B/b                **y/b A/b* 5 6
              5 6

Case 2.  Lb0
  Subcase 2a. If py is r (Case 1) color py with b and color v with r and stop
         py/r                              py/b
        /   \                             /   \
   ** y/b  *v/b                        y/b   v/r   and stop
           / \                               / \
          A/b B/b                           A/b B/b

  Subcase 2b, If py is b    FIXDELETE(py) py plays the role of y and py
         also carries the extra black inherited by y due to the x splice out.
         py/b                           ** py/b
        /   \                              /   \
   ** y/b  *v/b                        y/b   v/r
           / \                               / \
          A/b B/b                           A/b B/b
```



Running time is $O(\lg n)$ as well. Cases 1, 2a, 3, 4 terminate in $O(1)$ time, Case 2b advances (moves towards the top) one level every time it is executed, and the height of the RB tree is $O(\lg n)$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.