Data Structures II
Fall 2009
Red Black Trees

These notes are being created in conjunction with the teaching of CPSC 482 in the Fall 2009 term at the University of Northern British Columbia.

1 Red Black Trees

Definition A red-black tree is a Binary Search Tree (BST) where every node is coloured either red or black. The colouring obeys the following rules:

- every red node has a black parent (consequently, the root is black).
- the number of black nodes on any path from the root to a node with less than two children is constant over all such paths.

These two properties make red-black trees into a form of B-tree with branching factor of 2–4, if you regard red nodes as belonging to part of a super node containing their black parent.

1.1 Insertion

Insertion into a red-black tree is relatively easy to do in a recurive fashion where you first insert into a subtree recursively, then (possibly) rebalance the result.

Insertion into a tree — Algorithm Set the root equal to the tree obtained by inserting into the tree root recursively as below, and then re-colour the root black if necessary (this is the only way that the black height increases).

Insertion into a node — Recursive Algorithm There are preconditions and postconditions whose maintainance results in a much simpler algorithm.

1. call this algorithm with only black nodes (or null nodes). Deal with the red nodes internally.
2. ensure that the tree returned by this algorithm has the same black height as the tree it was called with.

Note that both of these invariants can be checked by your recursive insertion routine. Detection of invariant failure leads to quick debugging.

List of Cases. The algorithm now devolves into a list of cases. Here is one possible way of dividing into cases.

null tree When inserting into a null tree, simply return a tree consisting of one red node.

match There is always the possibility that the item that you are inserting matches the current node.[1] [2]

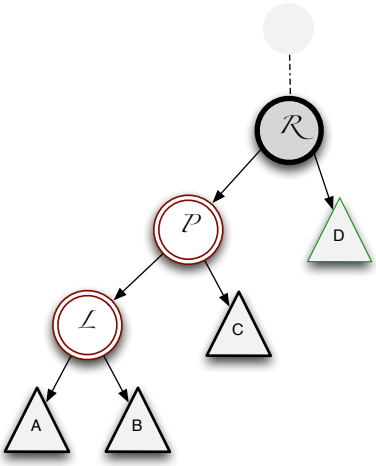
Otherwise there are symmetric left-right cases depending on how the value being inserted compares with the value in the current tree node. The following cases are when the item to be inserted is less than the value in the current node.

black child node When the child to be inserted into is black (or null), RECURSIVELY insert into the child, and you are done. Note that the post-condition of recursive insertion (that the black-height hasn't changed) meets black-height invariant of the tree, and becuase the node is black, it doesn't matter if the child into which you insert becomes red.

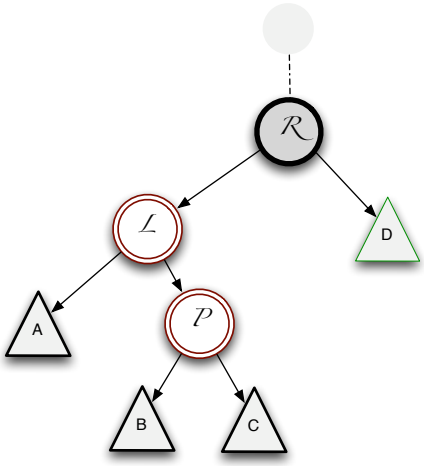
red child node Things start to become more complicated here. The most important complication is that we are not allowed to call ourselves

[1] Remember that `(x==y)` should imply that `(x.comparesTo(y)==0)`, but that the other implication may not hold, so some thought about insertion is important.

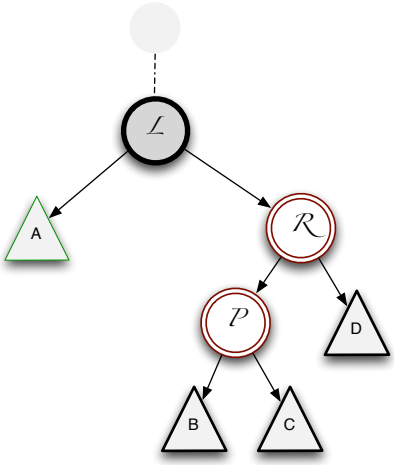
[2] If you are using lazy deletion you must check ensure that the lazy delete flag is set.



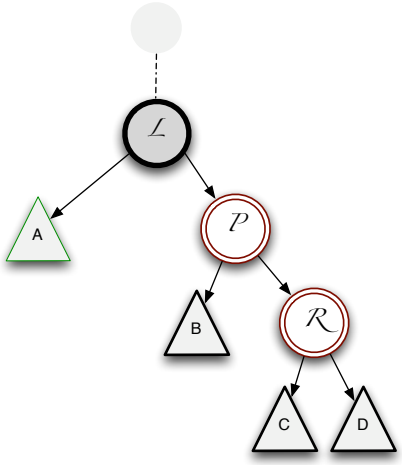
LL rotate



LR rotate



RL rotate



RR rotate

Figure 1: Red Red insert cases before rotate

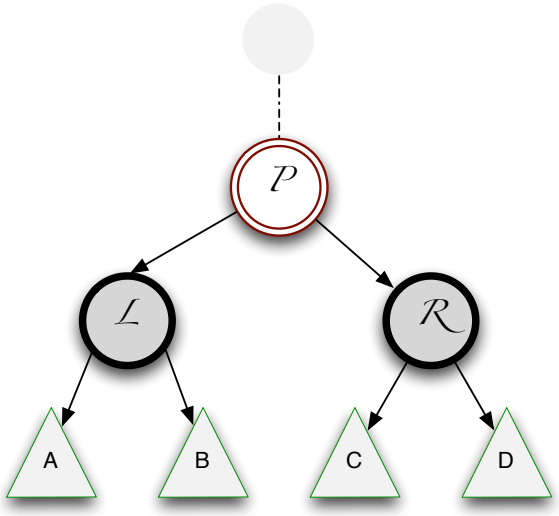


Figure 2: Red Red insert cases after rotate

recursively yet. We compare the value to be inserted against the value in the red child and have the following sub-cases.

If the item that you are inserting matches the red child, no tree-shape or colour changes are needed. Otherwise we pick a left or right grandchild into which you need to insert. Note that the grandchild must be black (or null). (Why??) RECURSIVELY insert into the grandchild.

If the grandchild remains black after the recursive insertion we are done, and we can return the original node.

Otherwise we have a red child and a (post-insert) red grandchild, and we need to rotate and re-color. This can be done in a uniform way. First name everything in sight, as shown in Figure 1. Next, relink and recolour as shown in Figure 2. Return the *new* parent node.

Note Carefully. The node returned from an insertion operation need not be the same as the original parent. Consequently *the caller* of the re-

cursive insert **must** perform an assignment with the result of the insertion call.

1.2 Deletion

Deletion from a red-black tree is moderately tricky. Again a recursive strategy works, where recursively one deletes from a subtree, then rebalances as needed.

When deleting, it is important to keep track of whether the tree currently being deleted from has lost black-height. Almost all of the complications of deletion are ensuring that black-height balance is restored after deletion. To simplify the coding, it pays to have a `private`, non-static `boolean needsHeightAdjust` that is set if there a black deficit.

It also pays to have several subroutines with different responsibilities. Here is one such way of dividing the labour.

`void delete(E);`. A top level routine that calls its recursive cousin, and ensures that the `root` is set to point to the result of deletion.

Most of the following functions could in principle be static. However, this fights with using an explicit `nullNode`.

`RedBlackNode<E> delete(RedBlackNode<E>,E);`. The recursive manager of deletion. This recursively calls itself on a child or calls `deleteThisNode` as appropriate. It then applies a rebalancing routine.

`RedBlackNode<E> deleteThisNode(RedBlackNode<E>);` This routine handles the case where we have found the node containing the data we wish to delete. It is important that this routine sets `needsHeightAdjust` appropriately before exiting.

As always when deleting from a BST, there are three cases depending on the number of children of the node to be deleted.

If there are no children, we are deleting a leaf, and we can simply return a null node. However, if the node being deleted is black, we must set `needsHeightAdjust`, otherwise we clear it. Note that deleting a black leaf has height consequences that we must fix later on.

If there is exactly one child, then we must be a black node, with one red leaf child. (Why??) In this case we can move the data from the red node into the black node, delete the red child, and clear the `needsHeightAdjust` variable.

If there are two children, we use the standard trick of deleting the leftmost child of the right subtree (using `deleteLeftmost`), and putting that value in the node that we actually want to delete. In this case, `needsHeightAdjust` gets set by `deleteLeftmost`.

`RBNode<E> deleteLeftmost(RBNode<E> tree, RBNode<E> w);`. This is a variant of the standard delete routine, used by `deleteThisNode` to handle the case where the node we want to delete has two non-trivial children. It is useful to pass down an extra variable saying in which node to store the data from the leftmost node being deleted.

This routine should be recursive, so that it can call "`rebalance`" after deleting from the left subtree.

Note that the leftmost node of a red-black tree is either a leaf or black node with one right red child (why???). If we delete a black leaf, we must set `needsHeightAdjust` to inform our callers.

`RBNode<E> rebalance(RBNode<E> node, int which);`. This function rebalances a node that has had a deletion from a child. The variable `which` indicates from which child the deletion occured. It uses and sets `needsHeightAdjust`. It returns a better balanced node.

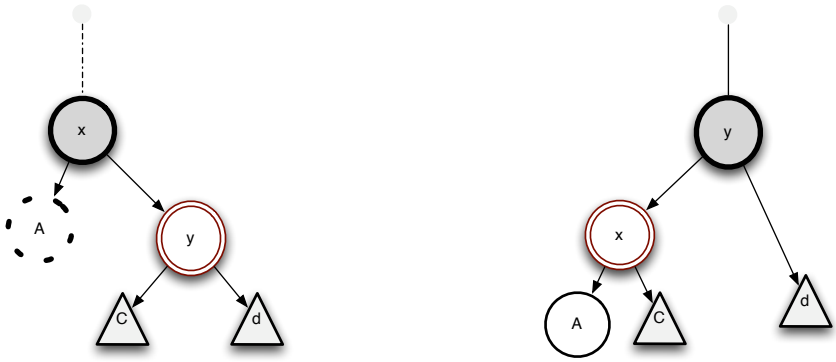
This function is the heart of the difficulty of red-black tree deletion and is described in more detail below.

1.3 Rebalancing After Deletion

Here we describe the `rebalance` algorithm in more detail. This function is called by `delete` and `deleteLeftmost` after deleting from a child node.

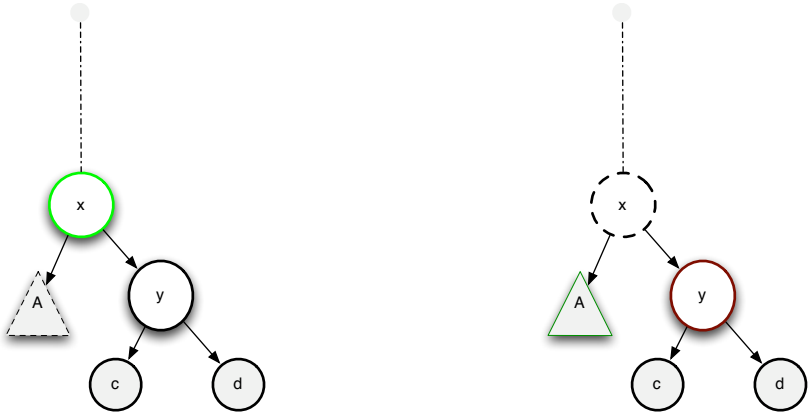
There is one very easy case. If `needsHeightAdjust` is false, we can simply return the node in question.

Otherwise, everything is concerned with the sibling of the child from which the deletion occured to cause the black deficit. There are four different cases.



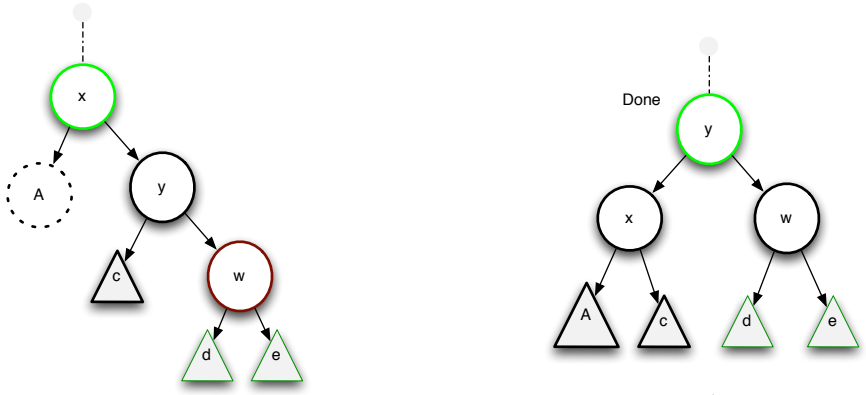
Before After

Figure 3: Deletion: Red sibling rebalance



Before After

Figure 4: Deletion rebalance: Lots of blacks



Before

After

Figure 5: Deletion: Black neice rebalance

Case I. The sibling is red, as shown in Figure 3. In this case, we perform the rotate shown in Figure 3, then recursively rebalance first `x`, then `y`. This looks dangerously like it might lead to infinite recursion, but we have improved the situation because after the rotation `A`'s sibling is black.

Case II. The sibling is black, as are both its children, as shown in Figure 4. This case may seem very special, but it is worth checking for because it doesn't require any rotations, just recolouring. We recolour the sibling red, which makes the sibling's tree's black count match that of the child which had the deletion.

The only possible conflict occurs when the original node (`x` in Figure 3) is red. However, this is a blessing in disguise. If the original node is red, we recolor it black and set `needsHeightAdjust` to `false`. Otherwise, we leave it black and let the black deficit trickle up.

The remaining cases are ones where the sibling is black, but at least one of its children is red.

Case III. If the sibling is black, and its "closest" child is black as shown in Figure 5, then we can perform a single rotate as shown in that figure. The result of this rotation is fix the black deficit, so we can set `needsHeightAdjust` to false.

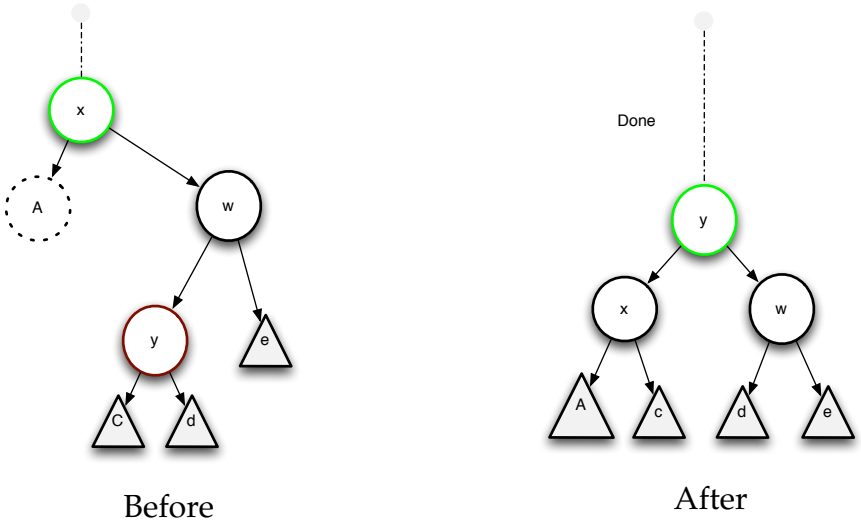


Figure 6: Deletion: Red neice rebalance

In this case the color of the original root is unknown, but we need to be careful to transfer it to the new root. Also note that the color of the "far nephew" changes from red to black.

Case IV. If the sibling is black, and its "closest" child is red as shown in Figure 6, then we can perform a double rotate as shown in that figure. The result of this rotation is fix the black deficit, so we can set `needsHeightAdjust` to false.

In this case the color of the original root is unknown, but we need to be careful to transfer it to the new root. Also note that both children become black.

2 Implementation Hints

It pays to write code to check invariants.

It pays to create code to print or otherwise display red-black tree structures very early on. These can be used to print output when invariants fail.

It pays to have drawings of before and after pictures for surgery.

A general strategy for surgery is to name everything in sight with local variables first (consulting your drawings). Then perform the surgery. Then check the "after" drawing and ensure that there are assignments or reasons that every link and color is as shown in the drawing.