

CSC 263, Lecture 4

June 8, 2004

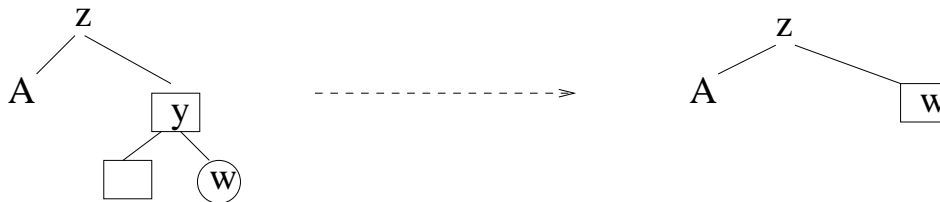
For another reference, see chapters 13 and 14 of the textbook (CLRS).

1 Red-Black Trees (continued)

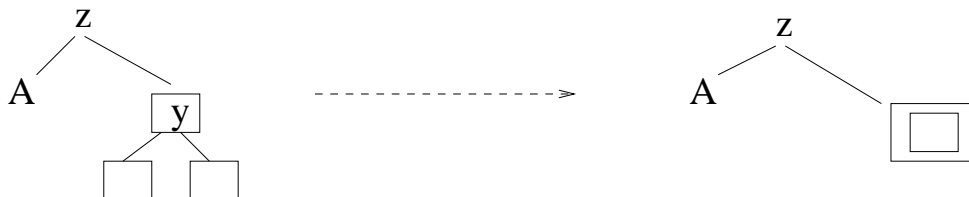
It remains to be seen how red-black trees can support deletion in time $O(\log n)$, where n is the number of nodes in the tree. Recall that **Red-BlackINSERT** was essentially the same as the **INSERT** operation for binary search trees except that it was followed by a “fix-up” process whenever the new node (which we colored red) had a red parent. If we perform **DELETE**(R, x) on a red-black tree R , then we remove a node y (which is either x or $\text{succ}(x)$). If y happens to be colored black, the black-height balance of the tree will almost certainly be upset and property 3 of red-black trees will be violated. So again, we will need a “fix-up” process.

Recall also that **DELETE** always removes a node that has at most one child (if x has two children, then we remove $\text{succ}(x)$, which never has two children). So, in short we have to worry about only those cases where y is black and y has at most one child.

- **Case A: y has one child:** y 's child, call it w , must be red since otherwise property 3 would be violated in the subtree rooted at y . So we can just remove y and make w black to preserve the black-height balance for any node above y .

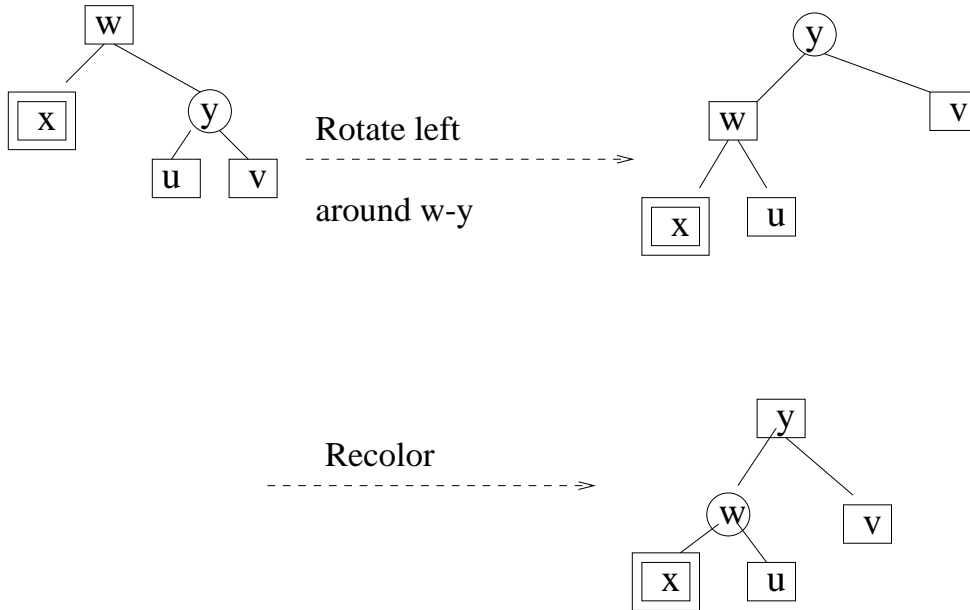


- **Case B: y has no children:** We can't apply the above trick if y has no children. Recall that the *NIL* values at the bottom of the tree are considered black leaves. To preserve the black-height for y 's ancestors, we'll remove y and replace it with a *NIL* node that is not just black, but “double-black” (denoted by a double rectangle). But while this upholds property 3 of red-black trees, it violates property 1 (that every node must be either red or black).



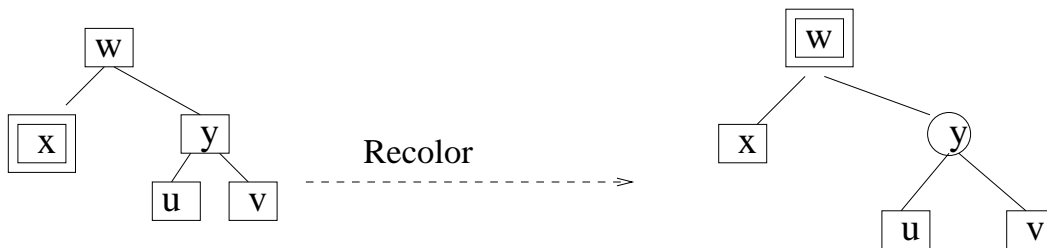
Now, we consider the problem of removing a double-black node from an arbitrary position in the tree. There are five cases for this; in all, x (which might be NIL) will be the double-black node that we want to remove. You should check that the transformations preserve properties 2 and 3.

- **Case 1: x 's sibling is red:** In this case, we modify the tree so that x 's neighbor is black and then apply one of the other cases. This is so that in general we can rely on the fact that x 's neighbor will be black:



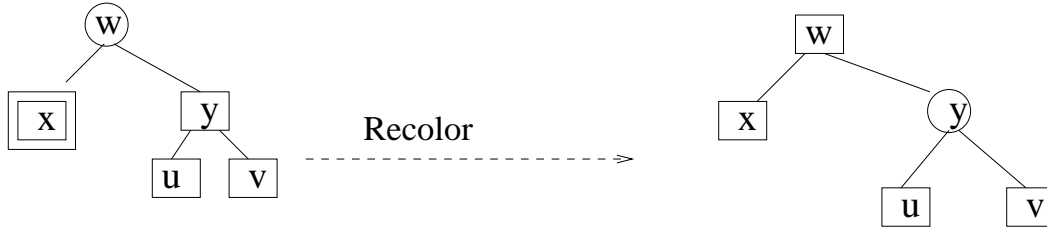
Notice that this transformation has moved the double-black node downwards in the tree.

- **Case 2: x 's parent, sibling and nephews are all black:**



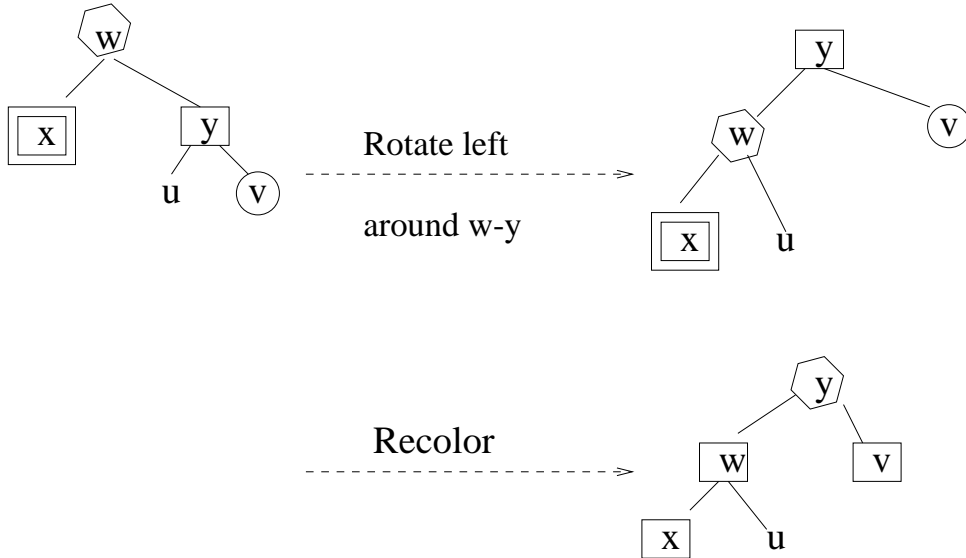
Notice that this transformation has moved the double-black node upwards in the tree. If this keeps happening, then eventually the root will become the double-black node and we can just change it to black without violating any properties. Otherwise we will be able to apply one of the other cases. Is it possible that Case 1 and Case 2 can conflict with each other by moving the double-black node downwards and then upwards in an infinite loop?

- **Case 3: x 's sibling and nephews are black, x 's parent is red:**



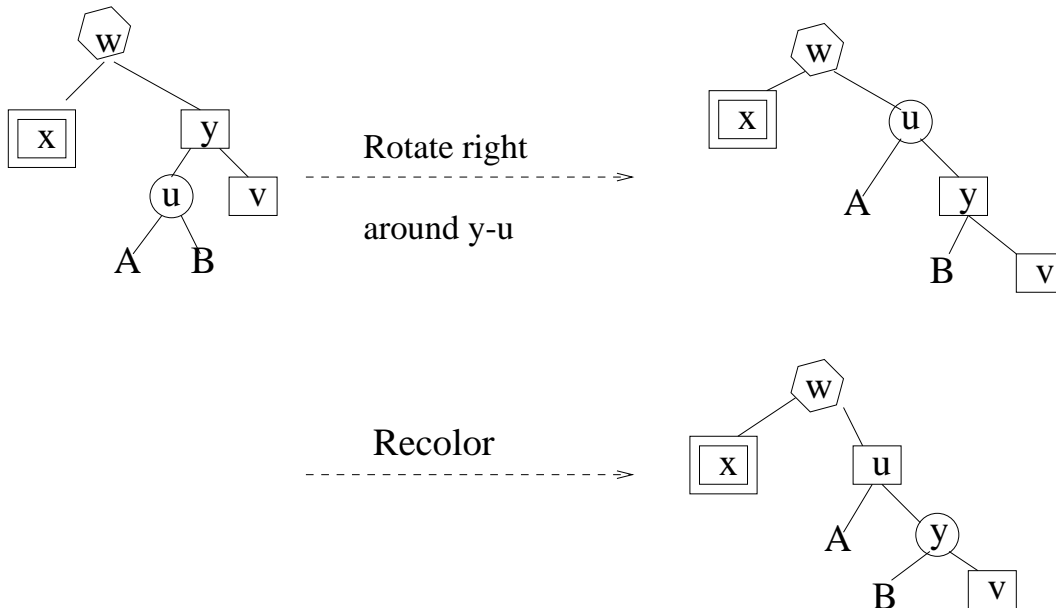
We can stop here because we have eliminated the double-black.

- **Case 4: x 's far nephew is red:** x 's parent w can start off as either color here (we'll denote this by a hexagon). After the transformation, y takes whatever color w had before.



Again, since there are no double-black nodes on the right, we can stop.

- **Case 5: x 's far nephew is black, x 's near nephew is red:** Here we're going to transform so that x 's far nephew becomes red. Then we can apply Case 4.



Similarly to **Red-BlackINSERT**, the fix-up process for **Red-BlackDELETE** in the worst case has to move the double-black node all the way from a leaf to the root. This requires $O(\text{height of the tree})$ operations, which for a red-black tree is $O(\log n)$.

2 Augmenting Red-Black Trees

A red-black tree by itself is not very useful. All you can do is search the tree for a node with a certain key-value. To support more useful queries we need more structure.

Example 1: Let's say we want to support the query **MIN(R)**, which returns the node with minimum key-value in red-black tree R .

One solution is to traverse the tree starting at the root and going left until there is no left-child. This node must have the minimum key-value. Since we might be traversing the height of the tree, this operation takes $O(\log n)$ time.

Alternatively, we can store a pointer to the minimum node as part of the data structure (at the root, for instance). Then, to do a query **MIN(R)**, all we have to do is return the pointer ($R.min$), which takes time $O(1)$. The problem is that we might have to update the pointer whenever we perform **INSERT** or **DELETE**.

- **INSERT(R, x):** Insert x as before, but if $key(x) < key(R.min)$, then update $R.min$ to point at x . This adds $O(1)$ to the running time, so its complexity remains $O(\log n)$.
- **DELETE(R, x):** Delete x as before, but if $x = R.min$, then update $R.min$: if x was the minimum node, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is *red* (otherwise property 3 would be violated). This right-child is $succ(x)$ and becomes the new minimum if it exists. If x had no children, then the new minimum is the parent of x . Again we add $O(1)$ to the running time of **DELETE** so it still takes $O(\log n)$ in total.

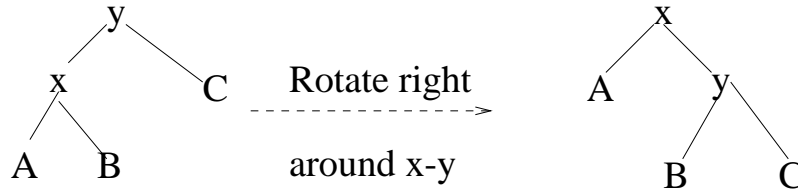
This is the best-case scenario. We support a new query in $O(1)$ -time without sacrificing the running times of the other operations.

Example 2: Now we want to know not just the minimum node in the tree, but, for any x in the tree, the minimum node in the subtree rooted at x . We'll call this query **SUBMIN(R, x)**.

To achieve this query in time $O(1)$, we'll store at each x a pointer $x.min$, to the minimum node in its subtree. Again, we'll have to modify **INSERT** and **DELETE** to maintain this information.

- **INSERT(R, x):** Insert x as before, but, for each y that is an ancestor of x , if $key(x) < key(y.min)$, then update $y.min$ to point at x . This adds $O(\log n)$ to the running time, so its complexity remains $O(\log n)$.
- **DELETE(R, x):** Delete x as before, but, for each y that is an ancestor of x , if $x = y.min$, then update $y.min$: if x was the minimum node in a subtree, then it had no left child. Since it is a red-black tree, its right-child, if it has one, is *red* (otherwise property 3 would be violated). This right-child is $succ(x)$ and becomes the new minimum if it exists. If x had no children, then the new minimum is the parent of x . Again we add $O(\log n)$ to the running time of **DELETE** so it still takes $O(\log n)$ in total.
- *Fix-up:* The rotations (but not the recolorings) in the fix-up processes for **INSERT** and **DELETE** might affect the submins of certain nodes. Let's say we are rotating right around $x-y$. The submin of x will be in the subtree A (or will be x itself if A is empty). This doesn't change

after the rotation. The submin of y , however, which used to be the same as x 's submin, is now in B (or y itself if B is empty). So, we set $\text{SUBMIN}(R, y)$ to y if B is empty, or to $\text{SUBMIN}(R, z)$, where z is the root of B . It takes just constant time to reset $\text{SUBMIN}(R, y)$, so rotations still take $O(1)$. The modification for a left rotation will be similar.



Example 3: We want to support the following queries:

- $\text{RANK}(R, k)$: Given a key k , what is its "rank", i.e., its position among the elements in the red-black tree?
- $\text{SELECT}(R, r)$: Given a rank r , what is the key with that rank?

For example, if R contains the key-values 3,15,27,30,56, then $\text{RANK}(R, 15) = 2$ and $\text{SELECT}(R, 4) = 30$.

Here are three possibilities for implementation:

1. Use red-black trees without modification:
 - Queries: Simply carry out an inorder traversal of the tree, keeping track of the number of nodes visited, until the desired rank or key is reached. This requires time $\Theta(n)$ in the worst case.
 - Updates: No additional information needs to be maintained.
 - Problem: Query time seems too long. We would expect to be able to carry out both types of queries in only $\Theta(\log n)$ time.
2. Augment red-black trees so that each node x has an additional field $\text{rank}(x)$ that stores its rank in the tree.
 - Queries: Similar to SEARCH , choosing path according to key or rank field (depending on the type of query). This requires time $\Theta(\log n)$, just like SEARCH .
 - Updates: Carry out normal update procedure, then update the rank field of all affected nodes. This can take time $\Theta(n)$ in the worst case, since any insertion or deletion affects the rank of every node with higher key-value.
 - Problem: We've achieved the $\Theta(\log n)$ query time we wanted, but at the expense of the update time, which has gone down from $\Theta(\log n)$ to $\Theta(n)$. We would like all operations to have time at worst $\Theta(\log n)$.
3. Augment red-black trees so that each node has an additional field $\text{size}(x)$ that stores the number of nodes in the subtree rooted at x (including x itself).

- Queries: We know that

$$\text{rank}(x) = 1 + \text{number of nodes that come before } x \text{ in the tree .}$$

RANK(R,k): Given key k , perform **SEARCH** on k keeping track of "current rank" r (which starts out as 0): when going left, r remains unchanged; when going right let $r := r + \text{size}(\text{left}(x)) + 1$. When x found such that $\text{key}(x) = k$, output $r + \text{size}(\text{left}(x)) + 1$. Note that we did not deal with degenerate cases (such as when k does not belong to the tree), but it is easy to modify the algorithm to treat those cases.

SELECT(R,r): Given rank r , start at $x = R$ and work down, looking for a node x such that $r = \text{size}(\text{left}(x)) + 1$ (return that node once it is found). If $r < \text{size}(\text{left}(x)) + 1$, then we know the node we are looking for is in the left subtree, so we go left without changing r . If $r > \text{size}(\text{left}(x)) + 1$, then we know the node we are looking for is in the right subtree, and that its relative rank in that tree is equal to $r - (\text{size}(\text{left}(x)) + 1)$, so we change r accordingly and go right. Once again, we did not deal with degenerate cases (such as when r is a rank that does not correspond to any node in the tree), but they are easily accomodated with small changes to the algorithm.

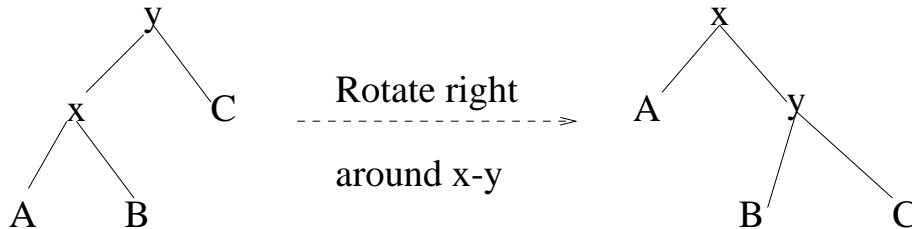
- Query time: $\Theta(\log n)$, as desired, since both algorithms are essentially like **SEARCH** (tracing a single path down from the root).
- Updates: **INSERT** and **DELETE** operations consist of two phases for red-black trees: the operation itself, followed by the fix-up process. We look at the operation phase first, and deal with the fix-up process afterwards.

INSERT(R,x): We can set $\text{size}(x) := 1$, and simply increment the size field for every ancestor of x .

DELETE(R,x): Consider the node y that is actually removed by the operation (so $y = x$ or $y = \text{succ}(x)$). We know the size of the subtree rooted at every node on the path from y to the root decreases by 1, so we simply traverse that path to decrement the size of each node.

We've shown how to modify the **INSERT** and **DELETE** operations themselves. If we show how to do rotations and keep the size fields correct, then we'll know how to do the whole fix-up process, since each case just consists of a rotation and/or a recoloring (recoloring does not affect the size field of any node).

Rotations: Consider right rotations (left rotations are similar).



$$\begin{aligned} \text{size}(y) &= \text{size}(A) + \text{size}(B) + \text{size}(C) + 2 \\ \text{size}(x) &= \text{size}(A) + \text{size}(B) + 1 \end{aligned}$$

$$\begin{aligned} \text{size}(x) &= \text{size}(A) + \text{size}(B) + \text{size}(C) + 2 \\ \text{size}(y) &= \text{size}(B) + \text{size}(C) + 1 \end{aligned}$$

The only size fields that change are those of nodes x and y , and the change is easily computed from the information available. So each rotation can be performed while maintaining the size information with only a constant amount of extra work.

- Update time: We have only added a constant amount of extra work during the first phase of each operation, and during each rotation, so the total time is still $\Theta(\log n)$.

Now, we have finally achieved what we wanted: each operation (old or new) takes time $\Theta(\log n)$ in the worst-case.

We've shown three examples of *augmenting* red-black trees to support new queries. The augmentation was very minor in the first example, but became quite significant in the last example. We can augment any data structure (not just red-black trees) according to the following scheme:

1. Pick a data structure to start with.
2. Determine additional information that needs to be maintained.
3. Check that the additional information can be maintained during each of the original operations (and at what additional cost, if any).
4. Implement the new operations.