Illinois Institute of Technology
Department of Computer Science

Lecture 8: February 8, 2016

CS 430 Introduction to Algorithms
Spring Semester, 2016

1 Red-black trees

As we saw in the previous lecture, binary search trees provide efficient average-time access, but without some way to guarantee a balanced tree, they behave poorly in the worst case. The idea behind a *balanced tree* is precisely that: to guarantee some sort of balance in the tree structure, in particular to guarantee logarithmic height. There are many sorts of balanced trees, including weight-balanced trees, height-balanced (AVL) trees, B-trees, and red-black trees. We examine the red-black trees in detail here.

1.1 Properties of red-black trees

Every red-black tree satisfies five properties:

- Each node is either red or black.
- Every leaf is black.
- The root is black.
- No two red nodes are adjacent. That is, no red node has a red parent or a red child.
- Paths from the root to any leaf all pass through the same number of black nodes (the *black-height* of the tree).

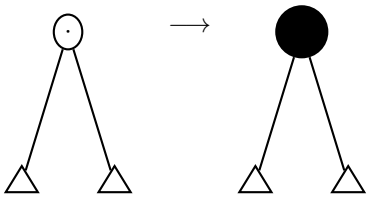
As well, we can insist that the root be black without altering the status of a tree: changing the root node from red to black will not violate any of the properties, though it increases the black-height by 1.

1.2 A similar structure: 2-3-4 trees

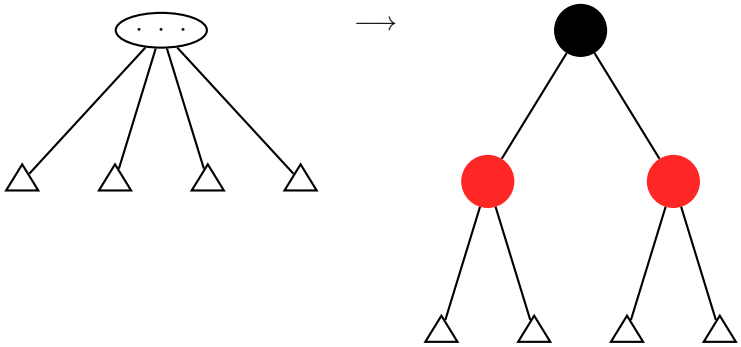
Although the primary structure we are studying is red-black trees, they are in fact equivalent to (and were originally presented as) 2-3-4 trees. Each node in a 2-3-4 tree contains either one data object and two children, two data objects and three children, or three data objects and four children. The other important property of 2-3-4 trees is that all its leaves have the same depth.

The tallest 2-3-4 tree of $n + 1$ leaves consists entirely of 2-nodes and has height $\lg(n + 1)$. The shortest 2-3-4 tree consists entirely of 4-nodes and has height $\log_4(n + 1) = \frac{1}{2}\lg(n + 1)$. Since 2-3-4 trees must be of logarithmic height, if we can find transformations between 2-3-4 trees and red-black trees which change the tree's height by at most a constant factor, then red-black trees must also have logarithmic height.

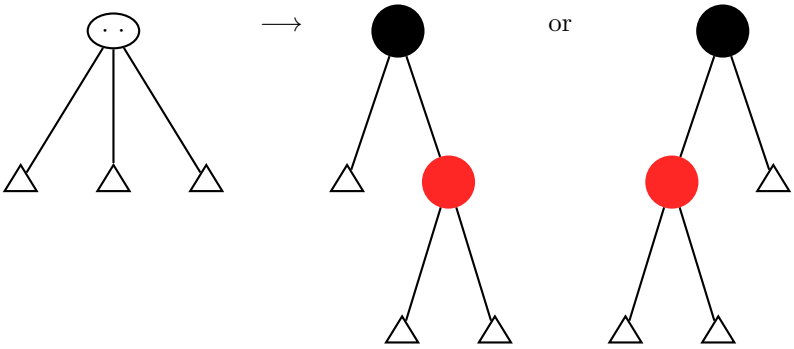
Such a transformation, fortunately, exists. A 2-node can be transformed to a black node with both children intact:



A 4-node can be transformed to a black node with two red children:



A 3-node can be transformed to a black node with two children, one a child of the 3-node, the other a red node parenting the other two children:



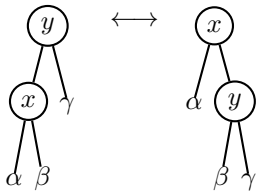
When the root is black, the same transformation can be applied in reverse to transform a red-black tree to a 2-3-4 tree.

This transformation clearly maintains the three local properties of red-black trees. Also, since each node in a 2-3-4 tree is mapped to a single black node (and perhaps additional red nodes) in a red-black tree, and all leaves in a 2-3-4 tree are at the same level, all leaves in a red-black tree share the same black-height.

Since the height of a 2-3-4 tree is between $\frac{1}{2}\lg(n+1)$ and $\lg(n+1)$, the black-height of a red-black tree is at most $\lg(n+1)$. And since the red-height of a tree can be no more than the black-height, the total height of a red-black tree is at most $2\lg(n+1)$.

1.3 Rotation

As the structure of a red-black tree is modified, it may come to violate the red-black properties. To restore them, we can perform an operation called *rotation*. We can rotate to the left or to the right:



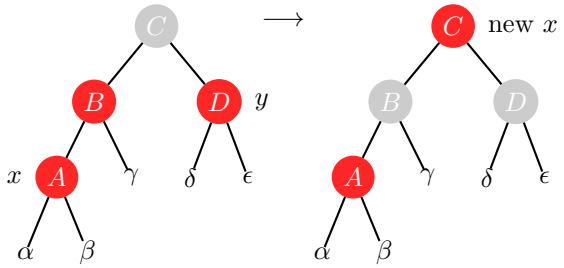
(A right rotation moves from the left side of the diagram to the right and a left rotation moves from the right side to the left.) Notice that rotation maintains the lexicographic nature of the tree.

1.4 Tree operations

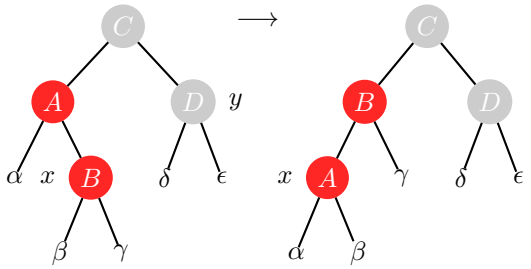
1.4.1 Insertion

To insert an element x into a red-black tree, we first insert it as if the tree were a simple binary search tree. Color the new node red. If the resulting tree is valid, then we're done. Otherwise we recolor nodes and perform rotations as necessary to restore the red-black properties to the tree. There are the following 3 cases to consider:[1]

1. x has a red uncle y. In this case recoloring is sufficient to restore the red-black properties. Make x's parent and uncle black and x's grandparent red.

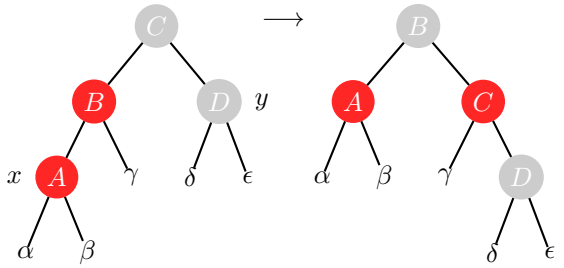


2. x has black uncle y and x is the right child. In this case, rotate the edge between x and its parent. This makes x's parent its left child and results in a Case 3.



3. x has black uncle y and x is the left child. In this case rotate the edge between x's parent and grandparent. Also color x's parent black and its old grandparent red. This restores the red-black properties.

[1] The details of this procedure are described in section 13.3 of CLRS.

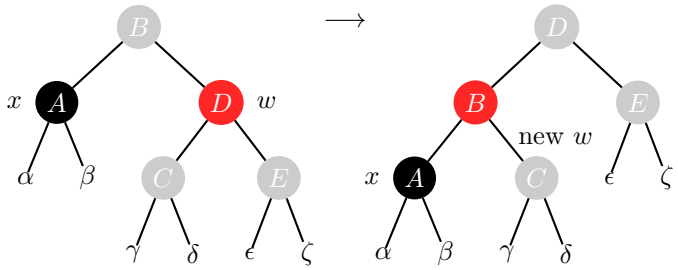


Since only local changes are made in each of these cases, insertion can be done in $O(\log n)$ time.

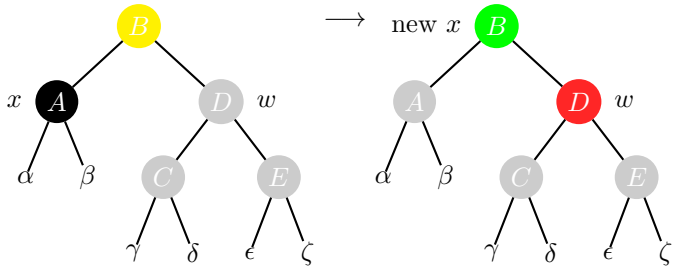
1.4.2 Deletion

Deletion, similarly, deletes a node as if it were in a simple binary search tree. If the deleted node was black, then any of its descendants have lost a point of "black depth" and repairs are needed to restore the red-black properties. In the interim, the properties can be patched by making a node "double black". Then rotations and recolorings are done to get rid of the double black node. Once again, several cases need to be dealt with:[2]

1. Double-black x has a red sibling w. In this case, rotate the edge between x and its parent so that this sibling becomes x's grandparent. Also recolor so that x's parent is red and its old sibling is black. This converts a case 1 situation into one of the other cases.

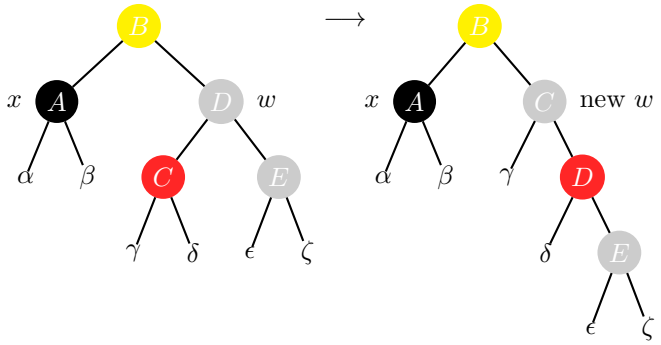


2. Double-black x has a black sibling w and black nephews. In this case, make x single black and the sibling red. To compensate for this, make x's parent black if it was red and double black if it was already black. This either solves the problem or moves the double black node up toward the root.

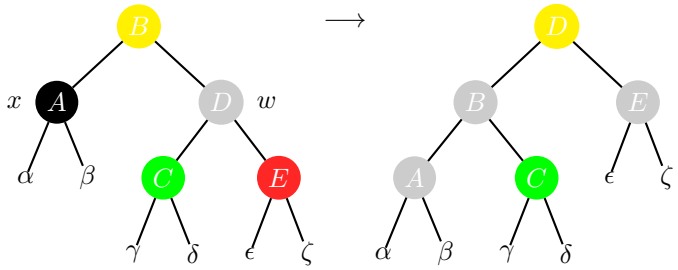


3. Double-black x has a black sibling w, a red left nephew, and a black right nephew. In this case, rotate the edge between the red nephew and the sibling so that the nephew becomes x's sibling. Also swap the colors of the nephew and sibling. This gives a situation handled by case 4.

[2] The details are described in section 13.4 of CLRS.



4. Double-black x has a black sibling w and red right nephew. In this case, rotate the edge between x's sibling and x's parent so that the former sibling becomes x's new grandparent. Then recolor so that the former right nephew becomes black, x becomes single black, x's parent becomes black, and x's former sibling gets the previous color of x's parent. This restores the red-black properties.



Again, since only local changes are made at each step and the double black node moves upward, deletion is accomplished in $O(\log n)$ time.