Chapter 9: Red-Black Trees

Table 1.1: Characteristics of Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced).	Deletion algorithm is complex.
Red-black tree	Quick search, insertion, deletion. Tree always balanced.	Complex.
2-3-4 tree	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage.	Complex.
Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion,	Slow access to other items, access to largest item.
Graph	Models real-world situations.	Some algorithms are slow and complex.

Overview

- Ordinary binary search trees suffer from a troublesome problem:
  - They work well if the data is inserted into the tree in random order.
  - Become much slower if data is inserted in already sorted order (17, 21, 28, 36,…) or inversely sorted order (36, 28, 21, 17,…).
    - When the values to be inserted are already ordered, a binary tree becomes unbalanced.
      - With an unbalanced tree, the capability to quickly find (or insert or delete) a given element is lost.
- The red-black tree is, in most cases, the most efficient balanced tree, at least when data is stored in memory as opposed to external files.

Our Approach to the Discussion

- Red-black trees are not trivial to understand.
- Also, because of a multiplicity of symmetrical cases (for left or right children, and inside or outside grandchildren),
  - the actual code is more lengthy and complex.
- Hard to learn about the algorithm by examining code.

Our Approach to the Discussion

Conceptual

- Concentrate on conceptual understanding rather than coding details.
  - Aided by the `RBTree` Workshop applet.
  - How you can work in partnership with the applet to insert new nodes into a tree.
    - Including a human in the insertion routine certainly slows it down,
    - but it also makes it easier for the human to understand how the process works.
- Searching works the same way in a red-black tree as it does in an ordinary binary tree.
- Insertion and deletion, while based on the algorithms in an ordinary tree, are extensively modified.

Our Approach to the Discussion

Top-Down Insertion: (Will discuss next)

- This means that some structural changes may be made to the tree as the search routine descends the tree looking for the place to insert the node.

Bottom-Up Insertion:

- This involves finding the place to insert the node and then working back up through the tree making structural changes.
- Less efficient because two passes must be made through the tree.

How Trees Become Unbalanced

- The Tree Workshop applet from Chapter 8, “Binary Trees”
  - Use the Fill button to create a tree with only one node.
  - Then insert a series of nodes whose keys are in either ascending or descending order.
  - The result will be something like that in Figure 9.1.

Figure 9.1: Items inserted in ascending order

- The nodes arrange themselves in a line with no branches.
- Because each node is larger than the previously inserted one, every node is a right child, so all the nodes are on one side of the root.
- The tree is maximally unbalanced.
- If you inserted items in descending order, every node would be the left child of its parent; the tree would be unbalanced on the other side.

Degenerates to O(N)

- When there are no branches, the tree becomes, in effect, a linked list.
- The arrangement of data is one-dimensional instead of two-dimensional.
- As with a linked list, you must now search through (on the average) half the items to find the one you're looking for.
  - In this situation the speed of searching is reduced to O(N), instead of O(logN) as it is for a balanced tree.
  - Searching through 10,000 items in such an unbalanced tree would require an average of 5,000 comparisons, whereas for a balanced tree with random insertions it requires only 14.
  - For presorted data you might just as well use a linked list in the first place.
- Data that's only partly sorted will generate trees that are only partly unbalanced.

Degenerates to O(N)

- Use the Tree Workshop applet from Chapter 8 to attempt to generate trees with 31 nodes, you'll see that some of them are more unbalanced than others, as shown in Figure 9.2.
  - Although not as bad as a maximally unbalanced tree, this situation is not optimal for searching times.
- In the Tree Workshop applet, trees can become partially unbalanced, even with randomly generated data,
  - because the amount of data is so small that even a short run of ordered numbers will have a big effect on the tree.
  - Also, a very small or very large key value can cause an unbalanced tree by not allowing the insertion of many nodes on one side or the other.
    - A root of 3, for example, allows only two more nodes to be inserted to its left.
- With a realistic amount of random data it's not likely a tree would become seriously unbalanced.
  - However, there may be runs of sorted data that will partially unbalance a tree.
- Searching partially unbalanced trees will take time somewhere between O(N) and O(logN), depending on how badly the tree is unbalanced.

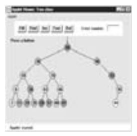


Figure 9.2:
A partially unbalanced tree

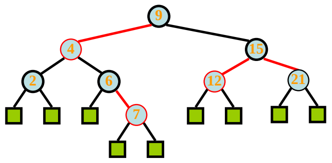
<u>**Balance**</u> to the Rescue!

To guarantee the quick O(log N) search times a tree is capable of

- We need to ensure that our tree is always balanced (or at least almost balanced).
- This means that each node in a tree needs to have roughly the same number of descendents on its left side as it has on its right.

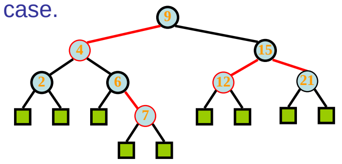
Red-Black Tree Characteristics

- The nodes are colored.
- During insertion and deletion, rules are followed that preserve various arrangements of these colors.



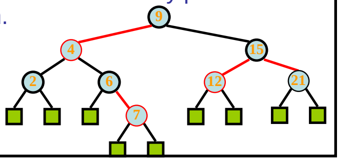
Red-Black Tree

- A Binary Search Tree.
- Every node in this tree is colored in either Red or Black.
- A historically popular alternative to the AVL tree. (Later this Chapter)
- Operation on red-black trees take O(log n) time in the worst case.



Red-Black Tree

- A red-black tree is a binary search tree where:
  - Every node is either red or black.
  - Each NULL pointer (leaf) is considered to be a black node
  - If a node is red, then both of its children are black.
  - Every path from a node to a leaf contains the same number of black nodes.
- The *black-height* of a node, n, in a red-black tree is the number of black nodes on any path to a leaf, not counting n.



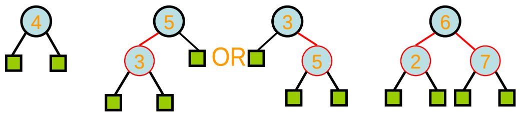
Red-Black Tree

A red-black tree must satisfy these properties:

1. The root is black.
2. All leaves are black.
3. Red nodes can only have black children. (although the converse isn't necessarily true)
4. All paths from a node to its leaves contain the same number of black nodes.

Red-Black Tree

- Root Property: The root is black
- External Property: Every external node is black
- Internal Property: The children of a red node are black
- Depth Property: All the external nodes have the same black depth



Red-Black Rules

When inserting (or deleting) a new node, certain rules, which we call the *red-black rules*, must be followed. If they're followed, the tree will be balanced:

1. Every node is either red or black.
2. The root is always black.
3. If a node is red, its children must be black (although the converse isn't necessarily true).
4. Every path from the root to a leaf, or to a null child, must contain the same number of black nodes.

Red-Black Rules

- The "null child" referred to in Rule 4 is a place where a child could be attached to a non-leaf node.
  - In other words, it's the potential left child of a node with a right child, or the potential right child of a node with a left child.
- The number of black nodes on a path from root to leaf is called the *black height*.
- Another way to state Rule 4 is that the *black height* must be the same for all paths from the root to a leaf.
- These rules probably seem completely mysterious.
  - It's not obvious how they will lead to a balanced tree, but they do;
  - Some very clever people invented them.
  - Copy them onto a sticky note, and keep it on your computer.
    - You'll need to refer to them often in the course of this chapter.

Duplicate Keys

- What happens if there's more than one data item with the same key?
  - This presents a slight problem in red-black trees.
  - It's important that nodes with the same key are distributed on both sides of other nodes with the same key.
  - That is, if keys arrive in the order 50, 50, 50,
    - you want the second 50 to go to the right of the first one, and the third 50 to go to the left of the first one.
    - Otherwise, the tree becomes unbalanced.
- This could be handled by some kind of randomizing process in the insertion algorithm.
  - However, the search process then becomes more complicated if all items with the same key must be found.
- It's simpler to outlaw items with the same key.
  - In this discussion we'll assume duplicates aren't allowed.

The Actions

- Suppose you see (or are told by the applet) that the color rules are violated. How can you fix things so your tree is in compliance? There are two, and only two, possible actions you can take:
  - You can change the colors of nodes.
  - You can perform rotations.
- Changing the color of a node means changing its red-black border color (not the center color).
- A rotation is a rearrangement of the nodes that hopefully leaves the tree more balanced.

Using the `RBTree` Workshop Applet

- **Clicking on a Node**
- **The Start Button**
- **The Ins Button**
- **The Del Button**
- **The Flip Button**
- **The RoL Button**
- **The RoR Button**
- **The R/B Button**
- **Text Messages**
- **Where's the Find Button?**

Using the `RBTree` Workshop Applet

- **Clicking on a Node**
  - The red arrow points to the currently selected node.
    - It's this node whose color is changed or which is the top node in a rotation.
  - You select a node by single clicking it with the mouse.
    - This moves the red arrow to the node.

Using the `RBTree` Workshop Applet

- **The Start Button**
  - When you first start the Workshop applet, and also when you press the Start button, you'll see that a tree is created that contains only one node.
  - Because an understanding of red-black trees focuses on using the red-black rules during the insertion process, it's more convenient to begin with the root and build up the tree by inserting additional nodes.
  - To simplify future operations, the initial root node is always given a value of 50. (You select your own numbers for subsequent insertions.)

Using the `RBTree` Workshop Applet

- **The Ins Button**
  - The Ins button causes a new node to be created, with the value that was typed into the Number box, and then inserted into the tree.
    - (At least this is what happens if no color flips are necessary. See the section on the Flip button for more on this possibility.)
  - Notice that the Ins button does a complete insertion operation with one push;
    - multiple pushes are not required as they were with the Tree Workshop applet in the last chapter.
  - The focus in the `RBTree` applet is not on the process of finding the place to insert the node, which is similar to that in ordinary binary search trees, but on keeping the tree balanced, so the applet doesn't show the individual steps in the insertion.

Using the `RBTree` Workshop Applet

- **The Del Button**
  - Pushing the Del button causes the node with the key value typed into the Number box to be deleted.
  - As with the Ins button, this takes place immediately after the first push; multiple pushes are not required.
  - The Del button and the Ins button use the basic insertion algorithms; the same as those in the Tree Workshop applet.
  - This is how the work is divided between the applet and the user: the applet does the insertion, but it's (mostly) up to the user to make the appropriate changes to the tree to ensure the red-black rules are followed and the tree thereby becomes balanced.

Using the `RBTree` Workshop Applet

- **The Flip Button**
  - If there is a black parent with two red children, and you place the red arrow on the parent by clicking on the node with the mouse, then when you press the Flip button the parent will become red and the children will become black.
    - That is, the colors are flipped between the parent and children.
  - You'll learn later why this is a desirable thing to do.
  - If you try to flip the root, it will remain black, so as not to violate Rule 2, but its children will change from red to black.

Using the `RBTree` Workshop Applet

- **The RoL Button**
  - This button carries out a left rotation.
  - To rotate a group of nodes, first single click the mouse to position the arrow at the topmost node of the group to be rotated.
  - For a left rotation, the top node must have a right child. Then click the button.
  - We'll examine rotations in detail later.

Using the `RBTree` Workshop Applet

- **The RoR Button**
  - This button performs a right rotation.
  - Position the arrow on the top node to be rotated, making sure it has a left child; then click the button.

Using the `RBTree` Workshop Applet

- **The R/B Button**
  - The R/B button changes a red node to black, or a black node to red.
  - Single click the mouse to position the red arrow on the node, and then push the button.
    - (This button changes the color of a single node; don't confuse it with the Flip button, which changes three nodes at once.)

Using the `RBTree` Workshop Applet

- **Text Messages**
  - Messages in the text box below the buttons tell you whether the tree is *red-black correct*.
  - The tree is red black correct
    - if it adheres to rules 1 to 4 listed previously.
    - If it's not correct, you'll see messages advising which rule is being violated.
  - In some cases the red arrow will point to where the violation occurred.

Using the `RBTree` Workshop Applet

- **Where's the Find Button?**
  - In red-black trees, a search routine operates exactly as it did in the ordinary binary search trees described in the last chapter.
    - It starts at the root, and, at each node it encounters (the current node), it decides whether to go to the left or right child by comparing the key of the current node with the search key.
  - We don't include a Find button in the `RBTree` applet
    - because you already understand this process and our attention will be on manipulating the red-black aspects of the tree.

Experimenting

Experiment 1

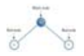


Figure 9.4: A balanced tree

- Press Start to clear any extra nodes.
- You'll be left with the root node, which always has the value 50.
- Insert a new node with a value smaller than the root, say 25, by typing the number into the Number box and pressing the Ins button.
  - This doesn't cause any rule violations, so the message continues to say Tree is red-black correct.
- Insert a second node that's larger than the root, say 75.
  - The tree is still red-black correct.
  - It's also balanced; there are the same number of nodes on the right of the only non-leaf node (the root) as there are on its left. The result is shown in Figure 9.4.

Experiment 1

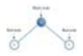


Figure 9.4: A balanced tree

- Notice that newly inserted nodes are always colored red (except for the root).
- This is not an accident.
- It's less likely that inserting a red node will violate the red-black rules than inserting a black one.
- This is because if the new red node is attached to a black one, no rule is broken.
- It doesn't create a situation in which there are two red nodes together (Rule 3), and it doesn't change the black height in any of the paths (Rule 4).
- Of course, if you attach a new red node to a red node, Rule 3 will be violated.
- However, with any luck this will only happen half the time.
- Whereas, if it were possible to add a new black node, it would always change the black height for its path, violating Rule 4.
- Also, it's easier to fix violations of Rule 3 (parent and child are both red) than Rule 4 (black heights differ), as we'll see later.

Experiment 2

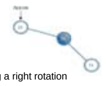


Figure 9.5: Following a right rotation

- Let's try some rotations.
- Start with the three nodes as shown in Figure 9.4.
- Position the red arrow on the root (50) by clicking it with the mouse.
- This node will be the *top node* in the rotation.
- Now perform a right rotation by pressing the RoR button.
  - The nodes all shift to new positions, as shown in Figure 9.5.

Experiment 2

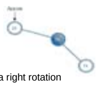


Figure 9.5: Following a right rotation

- In this right rotation,
  - the parent or top node moves into the place of its right child,
  - the left child moves up and takes the place of the parent,
  - and the right child moves down to become the grandchild of the new top node.
- Notice that the tree is now unbalanced;
  - there are more nodes to the right of the root than to the left.
  - Also, the message indicates that the red-black rules are violated, specifically Rule 2 (the root is always black).
  - Don't worry about this yet.
- Instead, rotate the other way.
  - Position the red arrow on 25, which is now the root (the arrow should already point to 25 after the previous rotation).
  - Click the RoL button to rotate left.
    - The nodes will return to the position of Figure 9.4.

Experiment 3

- Start with the position of Figure 9.4, with nodes 25 and 75 inserted in addition to 50 in the root position.
  - Note that the parent (the root) is black and both its children are red.
- Now try to insert another node.
  - No matter what value you use, you'll see the message `Can't Insert: Needs color flip`.
- As we mentioned, a color flip is necessary whenever, during the insertion process, a black node with two red children is encountered.
- The red arrow should already be positioned on the black parent (the root node), so click the Flip button.
  - The root's two children change from red to black.
  - Ordinarily the parent would change from black to red, but this is a special case because it's the root: it remains black to avoid violating Rule 2.
- Now all three nodes are black.
- The tree is still red-black correct.

Experiment 3

- Now click the Ins button again to insert the new node.
  - Figure 9.6 shows the result if the newly inserted node has the key value 12.
- The tree is still red-black correct. The root is black, there's no situation in which a parent and child are both red, and all the paths have the same number of black nodes (2). Adding the new red node didn't change the red-black correctness.

Experiment 4

Figure 9.6: Colors flipped, new node inserted

- Now let's see what happens when you try to do something that leads to an unbalanced tree. In Figure 9.6 one path has one more node than the other. This isn't very unbalanced, and no red-black rules are violated, so neither we nor the red-black algorithms need to worry about it. However, suppose that one path differs from another by two or more levels (where level is the same as the number of nodes along the path). In this case the red-black rules will always be violated, and we'll need to rebalance the tree.

Experiment 4

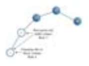


Figure 9.7: Parent and child are both red

- Insert a 6 into the tree of Figure 9.6. You'll see the message `Error: parent and child are both red.` Rule 3 has been violated, as shown in Figure 9.7.
- How can we fix things so Rule 3 isn't violated? An obvious approach is to change one of the offending nodes to black. Let's try changing the child node, 6. Position the red arrow on it and press the R/B button. The node becomes black.
- The good news is we fixed the problem of both parent and child being red. The bad news is that now the message says `Error: Black heights differ`. The path from the root to node 6 has three black nodes in it, while the path from the root to node 75 has only two. Thus Rule 4 is violated. It seems we can't win.
- This problem can be fixed with a rotation and some color changes. How to do this will be the topic of later sections.

More Experiments

- Experiment with the `RBTree` Workshop applet on your own. Insert more nodes and see what happens. See if you can use rotations and color changes to achieve a balanced tree. Does keeping the tree red-black correct seem to guarantee an (almost) balanced tree?
- Try inserting ascending keys (50, 60, 70, 80, 90) and then restart with the Start button and try descending keys (50, 40, 30, 20, 10). Ignore the messages; we'll see what they mean later. These are the situations that get the ordinary binary search tree into trouble. Can you still balance the tree?

The Red-Black Rules and Balanced Trees

- Try to create a tree that is unbalanced by two or more levels but is red-black correct. As it turns out, this is impossible. That's why the red-black rules keep the tree balanced. If one path is more than one node longer than another, then it must either have more black nodes, violating Rule 4, or it must have two adjacent red nodes, violating Rule 3. Convince yourself that this is true by experimenting with the applet.

Null Children

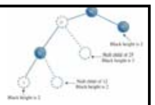


Figure 9.8: Path to a null child

- Remember that Rule 4 specifies all paths that go from the root to any leaf or to *any null children* must have the same number of black nodes. A null child is a child that a non-leaf node might have, but doesn't. Thus in Figure 9.8 the path from 50 to 25 to the right child of 25 (its null child) has only one black node, which is not the same as the paths to 6 and 75, which have 2. This arrangement violates Rule 4, although both paths to leaf nodes have the same number of black nodes.
- The term *black height* is used to describe the number of black nodes from between a given node and the root. In Figure 9.8 the black height of 50 is 1, of 25 is still 1, of 12 is 2, and so on.

Rotations

- To balance a tree, it's necessary to physically rearrange the nodes.
- If all the nodes are on the left of the root, for example, you need to move some of them over to the right side.
  - This is done using *rotations*.
- What rotations are and how to execute them.
  - Rotations are ways to rearrange nodes. They were designed to do the following two things:
    - Raise some nodes and lower others to help balance the tree.
    - Ensure that the characteristics of a binary search tree are not violated.

Rotations

- Recall that
  - in a binary search tree
    - the left children of any node have key values less than the node, while its right children have key values greater or equal to the node.
    - If the rotation didn't maintain a valid binary search tree it wouldn't be of much use,
      - because the search algorithm, as we saw in the last chapter, relies on the search-tree arrangement.
- Note that color rules and node color changes are used only to help decide when to perform a rotation;
  - fiddling with the colors doesn't accomplish anything by itself;
  - it's the rotation that's the heavy hitter.
  - Color rules are like rules of thumb for building a house (such as "exterior doors open inward"), while rotations are like the hammering and sawing needed to actually build it.

Simple Rotations

- In Experiment 2 we tried rotations to the left and right.
  - These rotations were easy to visualize
    - because they involved only three nodes.
- Let's clarify some aspects of this process.

What's Rotating?

- The term *rotation* can be a little misleading.
- The nodes themselves aren't rotated, the relationship between them changes.
- One node is chosen as the "top" of the rotation.
  - If we're doing a right rotation, this "top" node will move down and to the right, into the position of its right child.
    - Its left child will move up to take its place.
- Remember that the top node isn't the "center" of the rotation. If we talk about a car tire, the top node doesn't correspond to the axle or the hubcap, it's more like the topmost part of the tire tread.
- The rotation we described in Experiment 2 was performed with the root as the top node, but of course any node can be the top node in a rotation, provided it has the appropriate child.

Mind the Children

- You must be sure that,
  - if you're doing a right rotation,
    - the top node has a left child.
    - Otherwise
      - there's nothing to rotate into the top spot.
  - Similarly, if you're doing a left rotation, the top node must have a right child.

The Weird Crossover Node

- Rotations can be more complicated than the three-node example we've discussed so far.
- Click Start, and then, with 50 already at the root, insert nodes with following values, in this order: 25, 75, 12, 37.
- When you try to insert the 12, you'll see the `Can't insert: needs color flip` message.
  - Just click the Flip button.
    - The parent and children change color.
  - Then press Ins again to complete the insertion of the 12.
- Finally insert the 37. The resulting arrangement is shown in Figure 9.9a.

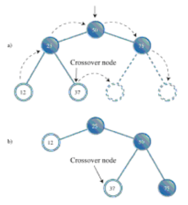


FIGURE 9.9: Rotation with crossover node

The Weird Crossover Node

- Now we'll try a rotation.
  - Place the arrow on the root (don't forget this!) and press the RoR button.
    - All the nodes move.
    - The 12 follows the 25 up,
    - and the 50 follows the 75 down.
- But what's this?
  - The 37 has detached itself from the 25, whose right child it was, and become instead the left child of 50.
  - Some nodes go up, some nodes go down, but the 37 moves *across*.
  - The result is shown in Figure 9.9b.
  - The rotation has caused a violation of Rule 4; we'll see how to fix this later.
- In the original position of Figure 9.9a,
  - the 37 is called an *inside grandchild* of the top node, 50. (The 12 is an *outside grandchild*.)
    - The inside grandchild, if it's the child of the node that's going up (which is the left child of the top node in a right rotation) is always disconnected from its parent and reconnected to its former grandparent.
    - It's like becoming your own uncle (although it's best not to dwell too long on this analogy).

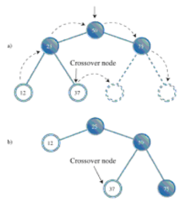


FIGURE 9.9: Rotation with crossover node

Subtrees on the Move

- We've shown individual nodes changing position during a rotation, but entire subtrees can move as well.
  - To see this,
    - click Start to put 50 at the root, and then insert the following sequence of nodes in order: 25, 75, 12, 37, 62, 87, 6, 18, 31, 43.
    - Click Flip whenever you can't complete an insertion because of the `Can't insert: needs color flip` message.
    - The resulting arrangement is shown in Figure 9.10a.

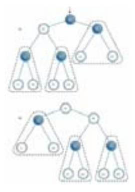


Figure 9.10: Subtree motion during rotation

Subtrees on the Move

- Position the arrow on the root, 50.
  - Now press RoR.
  - WoW! A lot of nodes have changed position.
  - The result is shown in Figure 9.10b. Here's what happens:
    - The top node (50) goes to its right child.
    - The top node's left child (25) goes to the top.
    - The entire subtree of which 12 is the root moves up.
    - The entire subtree of which 37 is the root moves across to become the left child of 50.
    - The entire subtree of which 75 is the root moves down.
- You'll see the `Error: root must be black` message but you can ignore it for the time being.
- You can flip back and forth by alternately pressing RoR and RoL with the arrow on the top node.
  - Do this and watch what happens to the subtrees, especially the one with 37 as its root.

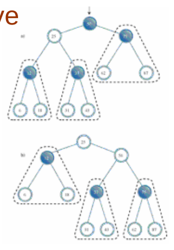


Figure 9.10: Subtree motion during rotation

Subtrees on the Move

- The figures show the subtrees encircled by dotted triangles.
- Note that the relations of the nodes within each subtree are unaffected by the rotation.
- The entire subtree moves as a unit.
- The subtrees can be larger (have more descendants) than the three nodes we show in this example.
- No matter how many nodes there are in a subtree, they will all move together during a rotation.

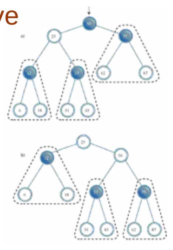


Figure 9.10: Subtree motion during rotation

Human Beings Versus Computers

- This is pretty much all you need to know about what a rotation does.
- To cause a rotation, you position the arrow on the top node, then press RoR or RoL.
- Of course, in a real red-black tree insertion algorithm, rotations happen under program control, without human intervention.
- Notice however that, in your capacity as a human being, you could probably balance any tree just by looking at it and performing appropriate rotations.
- Whenever a node has a lot of left descendants and not too many right ones, you rotate it right, and vice versa.
- Unfortunately, computers aren't very good at "just looking" at a pattern.
  - They work better if they can follow a few simple rules.
  - That's what the red-black scheme provides, in the form of color coding and the four color rules.

Inserting a New Node - Preview

- Don't worry if things aren't completely clear in the preview; we'll discuss things in more detail in a moment.
- In the discussion that follows we'll use X, P, and G to designate a pattern of related nodes.
  - X is a node that has caused a rule violation. (Sometimes X refers to a newly inserted node, and sometimes to the child node when a parent and child have a red-red conflict.)
    - X is a particular node.
    - P is the parent of X.
    - G is the grandparent of X (the parent of P).

Inserting a New Node - Preview

- On the way down the tree to find the insertion point,
  - you perform a color flip whenever you find a black node with two red children (a violation of Rule 2).
  - Sometimes the flip causes a red-red conflict (a violation of Rule 3). Call the red child X and the red parent P.
  - The conflict can be fixed with a single rotation or a double rotation, depending on whether X is an outside or inside grandchild of G.
  - Following color flips and rotations, you continue down to the insertion point and insert the new node.
- After you've inserted the new node X,
  - if P is black you simply attach the new red node.
  - If P is red, there are two possibilities:
    - X can be an outside or inside grandchild of G.
      - You perform two color changes (we'll see what they are in a moment).
      - If X is an outside grandchild, you perform one rotation,
      - and if it's an inside grandchild you perform two.
      - This restores the tree to a balanced state.

Inserting a New Node - Preview

- We'll divide the discussion into three parts, arranged in order of complexity:

  1. Color flips on the way down
  2. Rotations once the node is inserted
  3. Rotations on the way down

- If we were discussing these three parts in strict chronological order,
  - we'd examine part 3 before part 2.
  - However, it's easier to talk about rotations at the bottom of the tree than in the middle, and operations 1 and 2 are encountered more frequently than operation 3, so we'll discuss 2 before 3.

Inserting a New Node- Color Flips on the Way Down

- The insertion starts off doing essentially the same thing it does in an ordinary binary search tree:
  - It follows a path from the root to the place where the node should be inserted, going left or right at each node depending on the relative size of the node's key and the search key.
- However, in a red-black tree, getting to the insertion point is complicated by color flips and rotations.
  - We introduced color flips in Experiment 3; now we'll look at them in more detail.

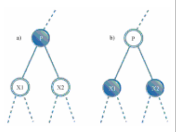


Figure 9.11: Color flip

Inserting a New Node- Color Flips on the Way Down

- Imagine the insertion routine proceeding down the tree, going left or right at each node, searching for the place to insert a new node.
- To make sure the color rules aren't broken, it needs to perform color flips when necessary.
  - Rule: Every time the insertion routine encounters a black node that has two red children, it must change the children to black and the parent to red (unless the parent is the root, which always remains black).
- How does a color flip affect the red-black rules?
  - For convenience,
    - let's call the node at the top of the triangle, the one that's red before the flip, P for parent.
    - We'll call P's left and right children X1 and X2. This is shown in Figure 9.11a.

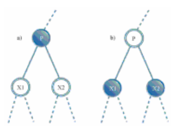


Figure 9.11: Color flip

Inserting a New Node- Color Flips on the Way Down

Black Heights Unchanged

- Figure 9.11b shows the nodes after the color flip.
  - The flip leaves unchanged the number of black nodes on the path from the root on down through P to the leaf or null nodes.
  - All such paths go through P, and then through either X1 or X2.
  - Before the flip, only P is black, so the triangle (consisting of P, X1, and X2) adds one black node to each of these paths.
- After the flip, P is no longer black, but both L and R are, so again the triangle contributes one black node to every path that passes through it.
  - So a color flip can't cause Rule 4 to be violated.
- Color flips are helpful
  - because they make red leaf nodes into black leaf nodes.
  - This makes it easier to attach new red nodes without violating Rule 3.

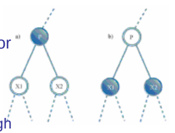


Figure 9.11: Color flip

Inserting a New Node- Color Flips on the Way Down

Could Be Two Reds

- Although Rule 4 is not violated by a color flip, Rule 3 (a node and its parent can't both be red) may be.
  - If the parent of P is black, there's no problem when P is changed from black to red.
  - However, if the parent of P is red, then, after the color change, we'll have two reds in a row.
- This needs to be fixed before we continue down the path to insert the new node.
  - We can correct the situation with a rotation, as we'll soon see.

Inserting a New Node- Color Flips on the Way Down

The Root Situation

- What about the root?
- Remember that a color flip of the root and its two children leaves the root, as well as its children, black.
  - This avoids violating Rule 2.
  - Does this affect the other red-black rules?
    - Clearly there are no red-to-red conflicts, because we've made more nodes black and none red.
    - Thus, Rule 3 isn't violated.
    - Also, because the root and one or the other of its two children are in every path, the black height of every path is increased the same amount; that is, by 1.
    - Thus, Rule 4 isn't violated either.

Inserting a New Node- Color Flips on the Way Down

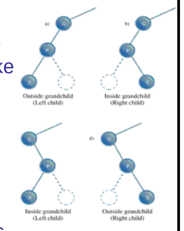
Finally, Just Insert It

- Once you've worked your way down to the appropriate place in the tree, performing color flips (and rotations) if necessary on the way down, you can then insert the new node as described in the last chapter for an ordinary binary search tree.
- However, that's not the end of the story…

Inserting a New Node - Rotations Once the Node is Inserted

Figure 9.12: Handed variations of node being inserted

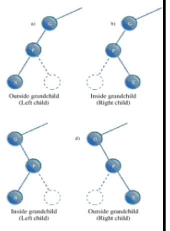
- The insertion of the new node may cause the red-black rules to be violated.
  - Therefore, following the insertion, we must check for rule violations and take appropriate steps.
- Remember that, as described earlier,
  - the newly inserted node, which we'll call X, is always red.
  - X may be located in various positions relative to P and G, as shown in Figure 9.12.



Inserting a New Node - Rotations Once the Node is Inserted

Figure 9.12: Handed variations of node being inserted

- Remember that
  - a node X is an outside grandchild if it's on the same side of its parent P that P is of its parent G.
    - That is, X is an outside grandchild if either it's a left child of P and P is a left child of G, or it's a right child of P and P is a right child of G.
    - Conversely, X is an inside grandchild if it's on the opposite side of its parent P that P is of its parent G.
- If X is an outside grandchild, it may be either the left or right child of P, depending on whether P is the left or right child of G.
- Two similar possibilities exist if X is an inside grandchild.
- It's these four situations that are shown in Figure 9.12.
  - This multiplicity of what we might call "handed" (left or right) variations is one reason the red-black insertion routine is challenging to program.



Inserting a New Node - Rotations Once the Node is Inserted

- The action we take to restore the red-black rules is determined by the colors and configuration of X and its relatives.
- Perhaps surprisingly, there are only three major ways in which nodes can be arranged (not counting the handed variations already mentioned).
  - Each possibility must be dealt with in a different way to preserve red-black correctness and thereby lead to a balanced tree.
  - Figure 9.13 shows what they look like. Remember that X is always red.

1. P is black.
2. P is red and X is an outside grandchild of G.
3. P is red and X is an inside grandchild of G.

- It might seem that this list doesn't cover all the possibilities. We'll return to this question after we've explored these three.

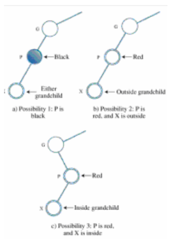

G
P
Black
Either grandchild
a) Possibility 1: P is black
G
P
Red
X
Outside grandchild
b) Possibility 2: P is red, and X is outside
G
P
Red
X
Inside grandchild
c) Possibility 3: P is red, and X is inside

Figure 9.13: Three post-insertion possibilities

Inserting a New Node - Rotations Once the Node is Inserted

Possibility 1: P Is Black

- If P is black, we get a free ride.
- The node we've just inserted is always red.
- If its parent is black, there's no red-to-red conflict (Rule 3), and no addition to the number of black nodes (Rule 4).
  - Thus no color rules are violated.
- We don't need to do anything else. The insertion is complete.

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 2: P Is Red, X Is Outside

- If P is red and X is an outside grandchild,
  - we need a single rotation and some color changes.
  - Let's set this up with the Workshop applet:
    - Start with the usual 50 at the root, and insert 25, 75, and 12.
    - You'll need to do a color flip before you insert the 12.
    - Now insert 6, which is X, the new node.
      - Figure 9.14a shows how this looks.
      - The message on the Workshop applet says `Error: parent and child both red`, so we know we need to take some action.

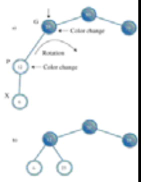


Figure 9.14: P is red, X is an outside grandchi

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 2: P Is Red, X Is Outside

- In this situation, we can take three steps to restore red black correctness and thereby balance the tree. Here are the steps:
  1. Switch the color of X's grandparent G (25 in this example).
  2. Switch the color of X's parent P (12).
  3. Rotate with X's grandparent G (25) at the top, in the direction that raises X (6). This is a right rotation in the example.

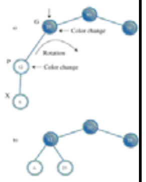


Figure 9.14: P is red, X is an outside grandchild

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 2: P Is Red, X Is Outside

- To switch colors, put the arrow on the node and press the R/B button.
- To rotate right, put the arrow on the top node and press RoR.
- When you've completed the three steps, the Workshop applet will inform you that the Tree is red/black correct. It's also more balanced than it was, as shown in Figure 9.14b.
- In this example, X was an outside grandchild and a left child.
  - There's a symmetrical situation when the X is an outside grandchild but a right child.
  - Try this by creating the tree 50, 25, 75, 87, 93 (with color flips when necessary).
    - Fix it by changing the colors of 75 and 87, and rotating left with 75 at the top. Again the tree is balanced.

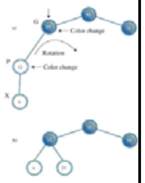


Figure 9.14: P is red, X is an outside grandchild

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 3: P Is Red and X Is Inside

- We need two rotations and some color changes.
  - To see this one in action, use the Workshop applet
    - create the tree 50, 25, 75, 12, 18. (Again you'll need a color flip before you insert the 12.)
    - The result is shown in Figure 9.15a.
- Note that the 18 node is an inside grandchild.
  - It and its parent are both red, so again you see the error message `Error: parent and child both red.`

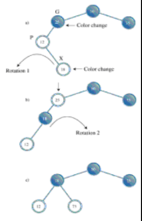


Figure 9.15: Possibility 3: P is red and X is an inside grandchild

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 3: P Is Red and X Is Inside

- Fixing this arrangement is slightly more complicated.
  - If we try to rotate right with the grandparent node G (25) at the top, as we did in Possibility 2, the inside grandchild X (18) moves across rather than up, so the tree is no more balanced than before.
    - (Try this, then rotate back, with 12 at the top, to restore it.)
    - A different solution is needed.
- The trick when X is an inside grandchild is to perform *two* rotations rather than one.
  - The first changes X from an inside grandchild to an outside grandchild, as shown in Figure 9.15b.
  - Now the situation is similar to Possibility 1, and we can apply the same rotation, with the grandparent at the top, as we did before.
    - The result is shown in Figure 9.15c.

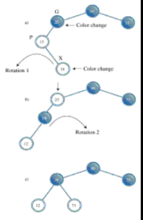


Figure 9.15: Possibility 3: P is red and X is an inside grandchild

Inserting a New Node
- Rotations Once the Node is Inserted

Possibility 3: P Is Red and X Is Inside

- We must also recolor the nodes. We do this before doing any rotations.
  - (This order doesn't really matter, but if we wait until after the rotations to recolor the nodes, it's hard to know what to call them.)
  - The steps are
    1. Switch the color of X's grandparent (25 in this example).
    2. Switch the color of X (*not* its parent; X is 18 here).
    3. Rotate with X's parent P at the top (*not* the grandparent; the parent is 12), in the direction that raises X (a left rotation in this example).
    4. Rotate again with X's grandparent (25) at the top, in the direction that raises X (a right rotation).
  - This restores the tree to red-black correctness and also balances it (as much as possible)
    - As with Possibility 2, there is an analogous case in which P is the right child of G rather than the left.

Inserting a New Node
- Rotations Once the Node is Inserted

What About Other Possibilities?

- Do the three Post-Insertion Possibilities we just discussed really cover all situations?
- Suppose, for example, that X has a sibling S; the other child of P.
  - This might complicate the rotations necessary to insert X.
  - But if P is black, there's no problem inserting X (that's Possibility 1).
  - If P is red, then both its children must be black (to avoid violating Rule 3).
    - It can't have a single child S that's black, because the black heights would be different for S and the null child.
    - However, we know X is red, so we conclude that it's impossible for X to have a sibling unless P is red.
- Another possibility is that G, the grandparent of P, has a child U, the sibling of P and the uncle of X.
  - Again, this would complicate any necessary rotations.
  - However, if P is black, there's no need for rotations when inserting X, as we've seen.
  - So let's assume P is red.
    - Then U must also be red, otherwise the black height going from G to P would be different from that going from G to U.
    - But a black parent with two red children is flipped on the way down, so this situation can't exist either.
- Thus the three possibilities discussed above are the only ones that can exist
  - (except that, in Possibilities 2 and 3, X can be a right or left child and G can be a right or left child).

Inserting a New Node
- Rotations Once the Node is Inserted

What the Color Flips Accomplished

- Suppose that performing a rotation and appropriate color changes caused other violations of the red-black rules to appear further up the tree.
  - One can imagine situations in which you would need to work your way all the way back up the tree, performing rotations and color switches, to remove rule violations.
- Fortunately, this situation can't arise.
  - Using color flips on the way down has eliminated the situations in which a rotation could introduce any rule violations further up the tree.
  - It ensures that one or two rotations will restore red-black correctness in the entire tree.
  - Actually proving this is beyond the scope of this book, but such a proof is possible.
- It's the color flips on the way down that make insertion in red-black trees more efficient than in other kinds of balanced trees, such as AVL trees.
  - They ensure that you need to pass through the tree only once, on the way down.

Inserting a New Node
- Rotations on the Way Down

- The last of the three operations involved in inserting a node: making rotations on the way down to the insertion point.
- As we noted, although we're discussing this last, it actually takes place before the node is inserted.
  - We've waited until now to discuss it only because it was easier to explain rotations for a just-installed node than for nodes in the middle of the tree.
- During the discussion of color flips during the insertion process,
  - We noted that it's possible for a color flip to cause a violation of Rule 3 (a parent and child can't both be red).
  - We also noted that a rotation can fix this violation.
- There are two possibilities, corresponding to Possibility 2 and Possibility 3 during the insertion phase described above.
  - The *offending node* can be an outside grandchild or it can be an inside grandchild.
- (In the situation corresponding to Possibility 1, no action is required.)

Inserting a New Node
- Rotations on the Way Down

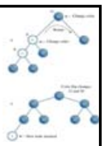


Figure 9.16: Outside grandchild on the way down

Outside Grandchild

- By "offending node" we mean the child in the parent-child pair that caused the red-red conflict.
  - Start a new tree with the 50 node,
  - and insert the following nodes: 25, 75, 12, 37, 6, and 18
  - You'll need to do color flips when inserting 12 and 6.

Inserting a New Node
- Rotations on the Way Down

Figure 9.16: Outside grandchild on the way down

Outside Grandchild

- Now try to insert a node with the value 3.
  - You'll be told you must flip 12 and its children 6 and 18.
    - You push the Flip button.
    - The flip is carried out, but now the message says `Error: parent and child are both red`, referring to 25 and its child 12.
    - The resulting tree is shown in Figure 9.16a.
- The procedure used to fix this is similar to the post-insertion operation with an outside grandchild, described earlier.
  - We must perform two color switches and one rotation.
  - So we can discuss this in the same terms we did when inserting a node,
    - we'll call the node at the top of the triangle that was flipped (which is 12 in this case) X.
    - This looks a little odd, because we're used to thinking of X as the node being inserted, and here it's not even a leaf node.
    - However, these on-the-way-down rotations can take place anywhere within the tree.

Inserting a New Node
- Rotations on the Way Down

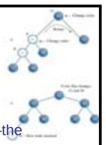


Figure 9.16: Outside grandchild on the way down

Outside Grandchild

- The parent of X is P (25 in this case), and the grandparent of X—the parent of P—is G (50 in this case).
- We follow the same set of rules we did under Possibility 2, discussed above.
  1. Switch the color of X's grandparent G (50 in this example). Ignore the message that the root must be black.
  2. Switch the color of X's parent P (25).
  3. Rotate with X's grandparent (50) at the top, in the direction that raises X (here a right rotation).
- Suddenly, the tree is balanced!
- It has also become pleasantly symmetrical.
- It appears to be a bit of a miracle, but it's only a result of following the color rules.
- Now the node with value 3 can be inserted in the usual way. Because the node it connects to, 6, is black, there's no complexity about the insertion.
- One color flip (at 50) is necessary. Figure 9.16b shows the tree after 3 is inserted.

Inserting a New Node
- Rotations on the Way Down

Inside Grandchild

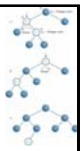


Figure 9.17: Inside grandchild on the way down

- If X is an inside grandchild when a red-red conflict occurs on the way down,
  - two rotations are required to set it right.
  - This situation is similar to the inside grandchild in the postinsertion phase, which we called Possibility 3.
- Click Start in the `RBTree` Workshop applet
  - begin with 50, and insert 25, 75, 12, 37, 31, and 43.
  - You'll need color flips before 12 and 31.
  - Now try to insert a new node with the value 28.
    - You'll be told it needs a color flip (at 37).
      - But when you perform the flip, 37 and 25 are both red, and you get the `Error: parent and child are both red` message.
        - » Don't press Ins again.
      - In this situation G is 50, P is 25, and X is 37, as shown in Figure 9.17a

Inserting a New Node
- Rotations on the Way Down

Inside Grandchild

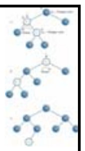


Figure 9.17: Inside grandchild on the way down

- To cure the red-red conflict, you must do the same two color changes and two rotations as in Possibility 3.
  1. Change the color of G (it's 50; ignore the message that the root must be black).
  2. Change the color of X (37).
  3. Rotate with P (25) as the top, in the direction that raises X (left in this example).
     - The result is shown in Figure 9.17b.
  4. Rotate with G as the top, in the direction that raises X (right in this example).
- Now you can insert the 28.
  - A color flip changes 25 and 50 to black as you insert it.
  - The result is shown in Figure 9.17c.

This concludes the description of how a tree is kept red-black correct, and therefore balanced, during the insertion process.

Deletion

- Recall,
  - coding for deletion in an ordinary binary search tree is considerably harder than for insertion.
  - The same is true in red-black trees, but in addition, the deletion process is, as you might expect, complicated by the need to restore red-black correctness after the node is removed.
- In fact, the deletion process is so complicated that many programmers sidestep it in various ways.
- One approach, as with ordinary binary trees, is to mark a node as deleted without actually deleting it.
  - A search routine that finds the node then knows not to tell anyone about it.
  - This works in many situations, especially if deletions are not a common occurrence.
- In any case, we're going to forgo a discussion of the deletion process.
  - Refer to Appendix B, "Further Reading," if you want to pursue it.

The Efficiency of Red-Black Trees

- Like ordinary binary search trees, a red black tree allows for searching, insertion, and deletion in O($\log_2$N) time.
  - Search times should be almost the same in the red-black tree as in the ordinary tree because the red-black characteristics of the tree aren't used during searches.
    - The only penalty is that the storage required for each node is increased slightly to accommodate the red-black color (a `boolean` variable).
  - More specifically, according to Sedgewick (see Appendix B),
    - in practice a search in a red-black tree takes about $\log_2$N comparisons, and it can be shown that it cannot require more than 2*$\log_2$N comparisons.

The Efficiency of Red-Black Trees

- The times for insertion and deletion are increased by a constant factor because of having to perform color flips and rotations on the way down and at the insertion point.
  - On the average, an insertion requires about one rotation.
  - Therefore, insertion still takes O($\log_2$N) time, but is slower than insertion in the ordinary binary tree.
- Because in most applications there will be more searches than insertions and deletions,
  - there is probably not much overall time penalty for using a red-black tree instead of an ordinary tree.
  - Of course, the advantage is that in a red-black tree sorted data doesn't lead to slow O(N) performance.

Implementation

- If you're writing an insertion routine for red-black trees, all you need to do (irony intended) is to write code to carry out the operations described above.
  - As we noted, showing and describing such code is beyond the scope of this book. However, here's what you'll need to think about.
    - You'll need to add a red-black field (which can be type boolean) to the Node class.
    - You can adapt the insertion routine from the `tree.java` program in Chapter 8.
    - On the way down to the insertion point, check whether the current node is black and its two children are both red.
      - If so, change the color of all three (unless the parent is the root, which must be kept black).
      - After a color flip, check that there are no violations of Rule 3.
        - » If so, perform the appropriate rotations: one for an outside grandchild, two for an inside grandchild.

Implementation

- When you reach a leaf node, insert the new node as in `tree.java`, making sure the node is red.
- Check again for red-red conflicts, and perform any necessary rotations.
- Perhaps surprisingly, your software need not keep track of the black height of different parts of the tree (although you might want to check this during debugging).
- You only need to check for violations of Rule 3, a red parent with a red child, which can be done locally (unlike checks of black heights, Rule 4, which would require more complex bookkeeping).
- If you perform the color flips, color changes, and rotations described earlier, the black heights of the nodes should take care of themselves and the tree should remain balanced.
  - The `RBTree` Workshop applet reports black-height errors only because the user is not forced to carry out insertion algorithm correctly.

Other Balanced Trees

- The *AVL tree* is the earliest kind of balanced tree.
  - Named after its inventors: Adelson- Velskii and Landis.
  - Each node stores an additional piece of data: the difference between the heights of its left and right subtrees.
    - This difference may not be larger than 1.
      - That is, the height of a node's left subtree may be no more than one level different from the height of its right subtree.
  - Following insertion, the root of the lowest subtree into which the new node was inserted is checked.
    - If the height of its children differs by more than 1, a single or double rotation is performed to equalize their heights.
    - The algorithm then moves up and checks the node above, equalizing heights if necessary.
    - This continues all the way back up to the root.

Other Balanced Trees

- Search times in an AVL tree are O(logN) because the tree is guaranteed to be balanced.
- However, because two passes through the tree are necessary to insert (or delete) a node, one down to find the insertion point and one up to rebalance the tree, AVL trees are not as efficient as red-black trees and are not used as often.

Other Balanced Trees

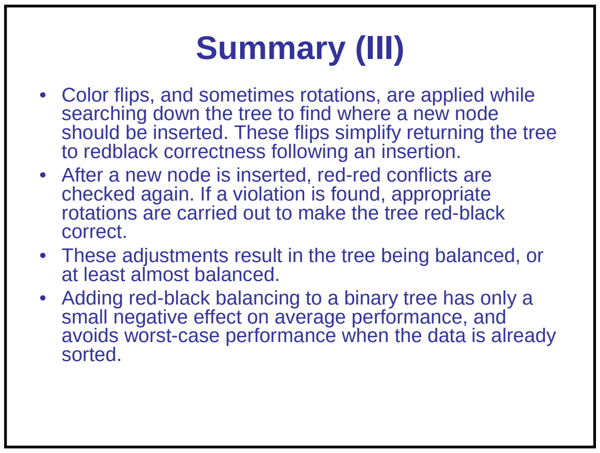
- The other important kind of balanced tree is the *multiway tree*,
  - in which each node can have more than two children.
  - One version of multiway trees, the 2-3-4 tree (next chapter).
  - One problem with multiway trees is that each node must be larger than for a binary tree, because it needs a reference to every one of its children.

Summary (I)

- It's important to keep a binary search tree balanced to ensure that the time necessary to find a given node is kept as short as possible.
- Inserting data that has already been sorted can create a maximally unbalanced tree, which will have search times of O(N).
- In the red-black balancing scheme, each node is given a new characteristic: a color that can be either red or black

Summary (II)

- A set of rules, called red-black rules, specifies permissible ways that nodes of different colors can be arranged.
- These rules are applied while inserting (or deleting) a node.
- A color flip changes a black node with two red children to a red node with two black children.
- In a rotation, one node is designated the top node.
- A right rotation moves the top node into the position of its right child, and the top node's left child into its position.
- A left rotation moves the top node into the position of its left child, and the top node's right child into its position.



Summary (III)

- Color flips, and sometimes rotations, are applied while searching down the tree to find where a new node should be inserted. These flips simplify returning the tree to redblack correctness following an insertion.
- After a new node is inserted, red-red conflicts are checked again. If a violation is found, appropriate rotations are carried out to make the tree red-black correct.
- These adjustments result in the tree being balanced, or at least almost balanced.
- Adding red-black balancing to a binary tree has only a small negative effect on average performance, and avoids worst-case performance when the data is already sorted.