Red-Black Trees

In our previous lecture we saw that many operations on binary search trees run in worst-case time $O(h)$, where h is the height of the tree. However, the height of the binary search tree is not necessarily nicely balanced. We showed the depending on how the elements are inserted, we may end up with a tree of height $O(n)$, meaning that our insert/delete operations would then run in time $O(n)$. Furthermore, although we showed that a randomly built binary search tree has height $O(\log n)$, this height is *not necessarily* maintained as we carry out inserts and deletes on the tree. This means that although we may start off with a nicely balanced tree, we can quickly disrupt its height as we carry out operations on the tree. Future operations then get slower and slower.

In this lecture we look at a variation of the binary search tree called the **red-black tree**. Such trees are **height-balanced** meaning that they **maintain a height** of $O(\log n)$ in the face of arbitrary insertions and deletions. The huge advantage of this balanced height is that the dynamic-set operations that run in time $O(h)$, now run in time $O(\log n)$ in a red-black tree .

1 The Red-Black tree

Red-Black trees are simply a *type of binary search tree*. However, they are defined in such a way that the height is guaranteed to be $O(\log n)$, and this height is maintained even after insert/delete operations. In order to accomplish this task, we simply add an additional attribute to the node object in the tree, that has exactly two possibilities:

$$x.color = red \qquad \text{or} \qquad x.color = black$$

Using the above attribute, we defined a red-black tree as a binary search tree that has the additional properties below:

The Red-Black tree properties:
The RB tree is a **binary search tree** with the following properties:

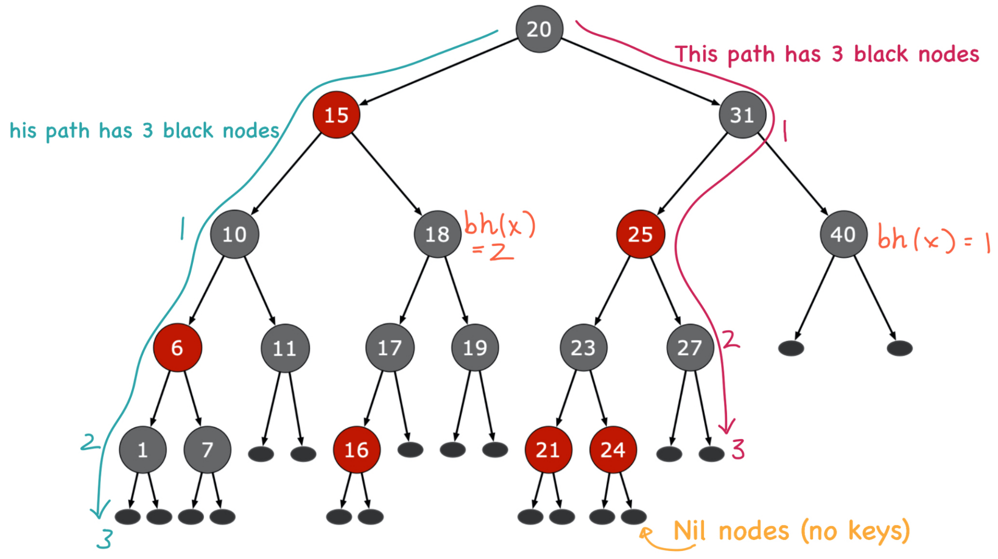
- Each node has a color: red or **black**
- The root is **black**
- A red node can only have **black** children
- Add additional **NIL black** nodes to the tree so that every real node has 2 children. These nodes do not contain keys, and are not considered part of the data set.
- For each node in the tree, all paths from that node to a NIL leaf have the *same number* of **black** nodes. Note that we *include* the NIL black nodes in the count.

One of the essential properties above states that *all* paths from a node to a leaf have the *same* number of black nodes. It is important to node that when we count the number of black nodes on a path to a leaf, then ensure you:

- Do *not* count the node itself
- Include the NIL leaf node

Below is an example of a red-black tree. One can check that all red nodes have only black children. Furthermore, the number of black nodes on any path from the root to a leaf is 3. If we count the number of black nodes from node 31 to a leaf, each path has exactly 2 black nodes. This fact must be true for all nodes in the tree, otherwise it is not a proper red-black tree.

The Nil nodes, are "additional" nodes, that have no keys, and are not consider part of the data set. However, they do "exist" in the sense they are not the same as the NIL value.



Since any node in a red-black tree has the same number of black nodes on any path down to a leaf, we give this number a special name called *black-height* of that node. Each node has its own black height. In the example above, node 18 has a black height of 2, since the number of black nodes on any path to a leaf is 2. Note that the root node has a black height of 3. We refer to the black-height of a *tree* as the black-height of its root node. The formal definition of black-heigh is given below:

Definition. *The **black height**, $bh(x)$ of node x is the number of black nodes on the path from x down to a NIL leaf (not including the node x itself).*

In the next section, we show how the specific properties of the red-black tree result in a tree that has height $\Theta(\log n)$.

2 Height of a Red-black tree

Since red-black trees are designed in such as way that the red nodes only have black children, then this restricts the number of red nodes that exist on any path. For example, paths of all red nodes do not exist. In fact, the most red nodes we can achieve on any path is by alternating between red and black nodes. This leads us the the following fact:

Minimum number of black nodes on a path:
The number of black-nodes on any path from the root to a leaf is at least half the actual number of nodes on the path.

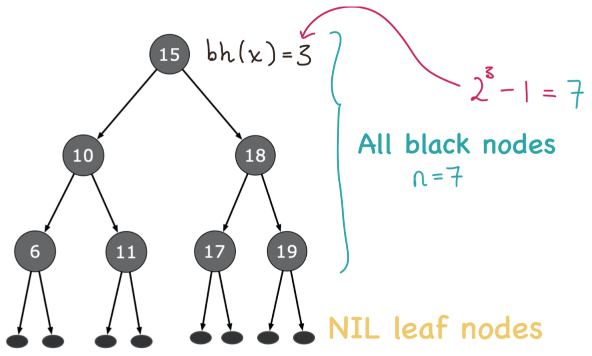
This property can be verified in the previous figure. Notice that the longest paths (of length 5) have 3 (real) black nodes. The paths of length 4 have in fact 3 black nodes. So the number of black nodes on any path it at least half of the total number of nodes on that path. It is impossible to have a path of length 5 with only 2 black nodes.

The reason we define black-height is so that we can use it to justify the overall balanced structure of the tree.

Theorem 1. *A red-black tree with n nodes (not counting the NIL nodes) has height $O(\log n)$*

Proof:. We won't show a formal proof of this fact here. Instead, we just sketch out the idea, and explain why the above theorem is true using the following facts.

- **Fact 1:** Imagine for a moment that *all* the nodes in the RB tree were black. The only way for such a RB tree to exist is if the tree were complete and full, as in the figure below:

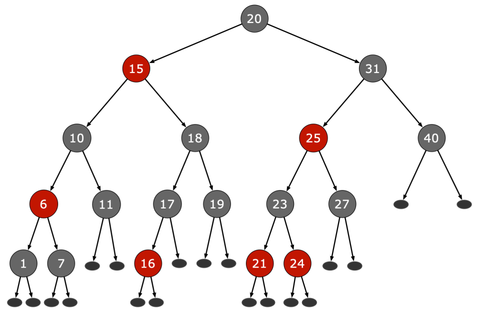


  In the above example, there are 7 actual nodes and the black-height of the tree is 3. Note that $7 = 2^3 - 1$. Similarly, a tree of all black nodes of black-height 5 has 31 actual nodes in the tree. Notice that $31 = 2^5 - 1$. In general, it looks like for an *all-black* red-black tree rooted at x, the number of nodes is given by:

$$\boxed{n = 2^{bh(x)} - 1}$$

- **Fact 2:** We can generalize this to red-black trees that have a mix of red and black nodes. The red-black tree below has a black-height of 3. It has $n = 18$ nodes. In this case, the number of nodes is *at least* $2^{bh(x)} - 1$. Thus a red-black tree with black-height given by $bh(x)$ has a total number of nodes bounded below by:

$$\boxed{n \geq 2^{bh(x)} - 1}$$



- **Fact 3:** If we isolate for n in the equation $n \geq 2^{bh(x)} - 1$ we get

$$bh(x) \leq \log(n + 1)$$

  Thus the black-height of a tree is $O(\log n)$. Now we just need to relate the black-height of a tree to the *actual* height of the tree. Recall from above that the number of black nodes on a path is at least half the total number of nodes. This means that the actual height of the tree is at most *double* the black height. Since the black-height is $O(\log n)$ then the actual height is also $O(\log n)$.

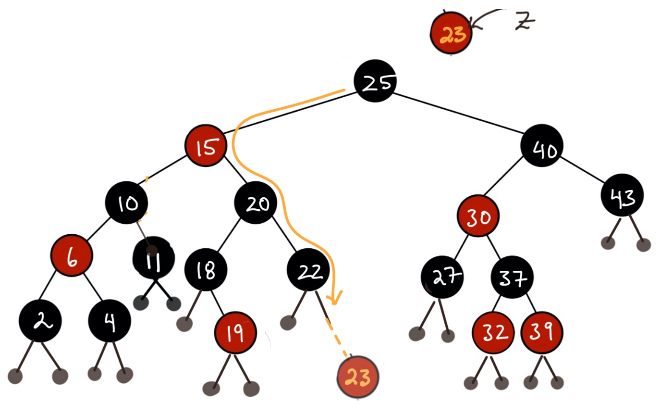
We have established that red-black trees have height $O(\log n)$. So as long as we can *maintain* the red-black properties during insertion/deletion, then the tree will remain balanced. This guarantees an $O(\log n)$ time for our dynamic-set operations. In the next section we look at how to vary the *insert* operation we saw in the regular BST so that it maintains the structure of the red-black tree.

3 Insertion

Insertion into a red-black tree is a two-step process. Recall that the tree is in fact a BST, so the insertion must obey the BST properties. However, simply inserting a new node with a certain color, may break the red-black tree properties. We fix this problem in the second step, with an algorithm called **RB-repair**.

Step 1: Insert node z into the BST

We insert a new node z using the insertion process for binary search trees. The insertion position takes the place of a NIL leaf. We temporarily insert the new node at this position, we color it red, and give it two NIL children of its own. Since the height of the red-black tree is $O(\log n)$, then the runtime of this step is $O(h) = O(\log n)$.



Step 2: RB-repair(z)

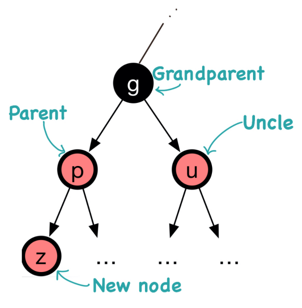
We now fix any red-black tree properties by applying an algorithm called **RB-repair** to node z. There are three cases that may arise after the insertion of the new node z.

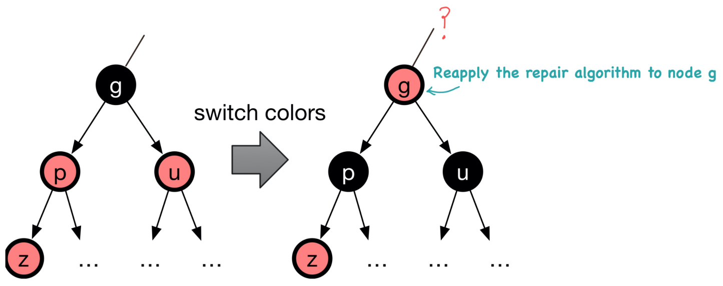
Case 0: When there is no extra work to do

- If the RB tree was initially empty, then the new node z becomes the root of the tree and is recolored black. RB-repair terminates.
- If the new node that was inserted had a **black** parent (as in the example above), then the properties of the RB tree are maintained and we have no new work to do. RB-repair terminates.

Case 1: What to do when the parent and uncle are red

The new node z is inserted as a red node in Step 1. If its parent is *also* red, we have violated a property of the RB trees. If the uncle is also red, as shown below, then we can simply recolor both the parent and the uncle to black, and recolor the grandparent to red:

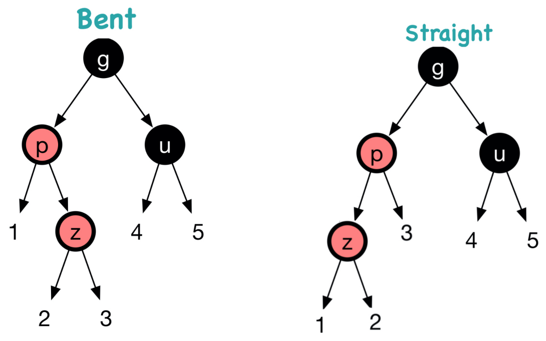




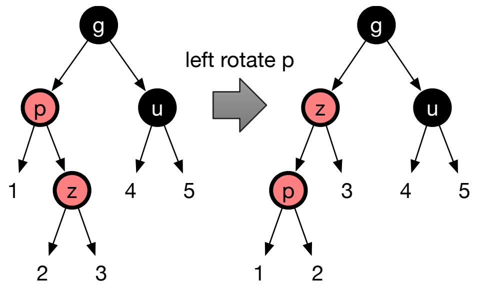
The result is that now z has a black parent. However, the recoloring process may have caused a problem with the parent of g. What if the parent of g was also red? Then we may have just caused a *new* violation further up the tree. To resolve this issue, we simply call our algorithm **recursively** for the node g: RB-repair(g). In doing so, the repair procedure moves up the tree repairing as it goes, until it reaches the root which is black.

Case 2: What to do when the parent is red and the uncle is black

Two different examples of this case are shown below. The one on the left is usually called the **bent** case and the one on the right is called the **straight** case.



If we come across a **bent** case, then in fact we *unbend* it using a rotation about the parent node p. A left rotation about p means that we reassign z as the left child of g, and p becomes the left child of z. We must also reassign the left child of z, labelled 2 in the diagram below. This child becomes the right child of p after the rotation. All of this reassigning only takes a constant amount of time.

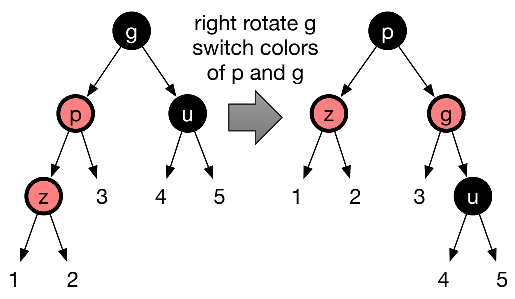


Now we move on to figuring out how to resolve the RB tree violation *assuming* that we are in case 2 and we have a straight case. In this case we can perform the following tasks:

- Perform a **rotation** about the grandparent of the new node. This shifts the node p up to the position of the grandparent, and the grandparent g rotates down to the other side. The subtrees get reassigned as in the figure. In the case of a right rotation, the right child of p becomes the left child of g.
- After the rotation, we switch the color of p to **black** and we switch the color of g to red

Once these operations are complete, there are no new possible violations in the rest of the tree.

.



Runtime of RB-repair: Case 1 and Case 2 together represent our RB-repair algorithm. In Case 1, we make a recursive call to the grandparent. In Case 2, there is no recursive call. Both Case 1 and Case 2 only take constant amount of time for the work they do. This means that in the worst case, the RB-repair may carry out a contant number of steps for all nodes on a path up to the root. Since the height of an RB tree is $O(\log n)$, the runtime of this algorithm is $O(\log n)$ in the worst case.

3.1 Runtime of Insert in RB tree:

The first step above involves inserting into the BST, which runs in time $O(\log n)$ in a RB tree because the height is $O(\log n)$. The second step, RB-repair, also runs in time $O(\log n)$. Therefore the overall runtime of insertion in an RB tree is $O(\log n)$.

A similar fix-it approach is possible for the delete operation. The repair algorithm for delete has several more cases. However, the involve a set of rotations and recolorings as in the insert algorithm.