CI583: Data Structures and Operating Systems
Balanced Trees

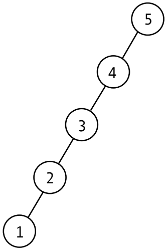
Outline

Unbalanced trees

Last time we saw how powerful and flexible a data structure is the tree. We saw that binary search trees can provide $O(\log n)$ retrieval, insertion and removal.

However, this is only true so long as the tree remains fairly well balanced. If we insert sequential data to a tree then the nodes arrange themselves just like a linked list. **Performance degrades to linear time.**

Unbalanced trees



Say we have a tree made up of 10,000 nodes. If the tree is maximally unbalanced, then the worst-case scenario of searching for an item is that it takes **10,000** steps. If the tree is completely balanced (or complete, or full), the worst-case scenario is **14**.

Unbalanced trees

Most of the time trees may not be maximally unbalanced but inputting a run of sequential data may cause it to be partially unbalanced, or it may have begun with a very small or large root.

In a tree of natural numbers, for instance, if the root is labelled 3 there can be at most two nodes in the left hand sub-tree. Operations on a tree like this will be somewhere between $O(n)$ and $O(\log n)$.

Self-balancing trees

Self-balancing trees are the solution to this problem. The first of these were AVL Trees, invented by Adelson-Velskii and Landis in 1962.

The idea is that self-balancing tree maintain the invariant that no path from root to leaf is more than twice as long as any other. To achieve this, the tree must re-balance itself after insertion and deletion.

Self-balancing trees

Variations on this idea are used in file system design, relational databases, and whenever fast access to a large amount of data is required.

For instance, relational databases store indices in memory in a self-balancing tree structure called a BTree or B+ Tree, providing logarithmic access time with little or no IO.

Linux file systems such as `ext3` store directory listings in a HTree, which uses a hash function to create a two-level balanced tree of files. HTree indexing improved the scalability from a practical limit of a few thousand files, into the range of tens of millions of files per directory.

Red-black trees

The type of self-balancing tree we will consider in detail is a BST called the red-black tree. Like the heap we saw in the last lecture, we define a series of invariants for RB-trees and make sure that they will all still hold after each operation.

Red-black trees were invented in 1972 by Bayer.

Red-black trees

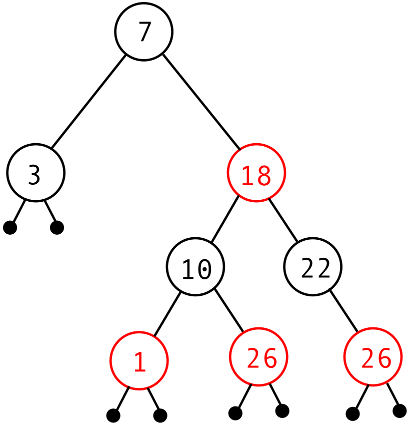
The invariants on RB-trees:

1. Each node is either red or black (think of this “colour” as an extra bit – we could use 1 or 0 or any other choice).
2. The root and leaves are black.
3. If a node is red, its children must be black.
4. For each node, x, every path from x to a leaf contains the same number of black nodes.

The motivation for these conditions is probably mysterious to you, but we will see that maintaining them results in a balanced tree and gives us the logarithmic performance we want.

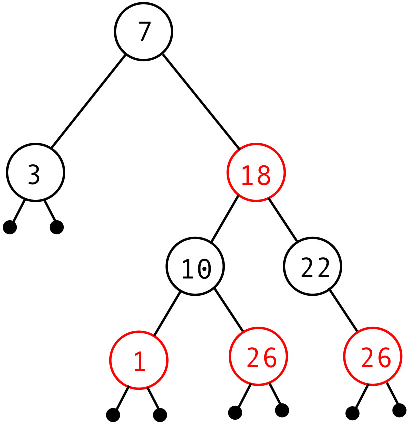
Red-black trees

An example. The small filled black circles represent null pointers in the leaves (not normally depicted). These are always black. We won't normally show them but this is what we mean when we say the "leaves" are black.



Red-black trees

The black-height of a node x is the number of black nodes on a path from x to a leaf, not including x. So we can state property 4 in terms of black-height.



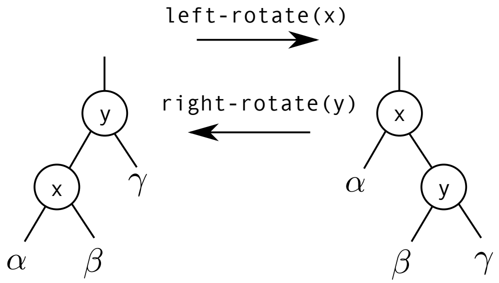
Red-black trees

Because we include the null pointers a red-black tree is a branching BST – every node has 2 or 0 children. The properties force a red-black tree with n nodes to have $O(\log n)$ height. Actually, the height will be $2(\log n + 1)$ – see (Cormen 2009, p309) for a proof.

Queries (e.g. search, find the minimum or maximum element etc) will require a visit to every level at worst, giving us $O(h)$ or $O(\log n)$ time. Updates (insertion and deletion) are more tricky.

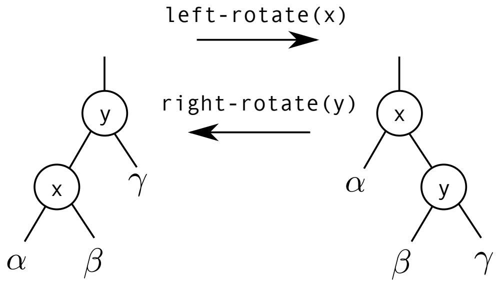
Rotations

Before we can describe how to update a red-black tree, we need to understand rotations. A rotation is a local change to the structure of the tree that preserves the RB properties.



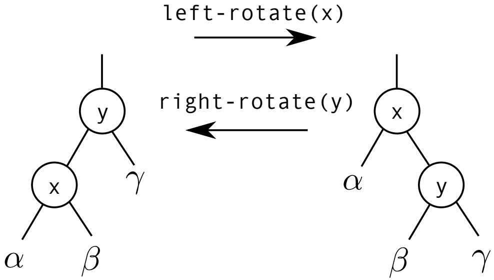
Rotations

The left-rotation pivots around the link from x to y. When we rotate in either direction we assume that x and y are not nil (i.e. they are real, internal nodes). The α, β and γ components might be nil or might be actual subtrees. Either way, they are properly balanced RB trees.



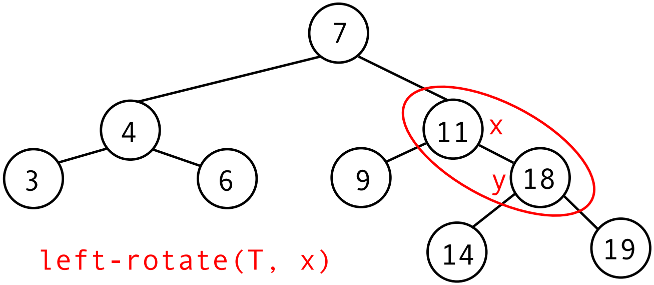
Rotations

We can easily see that rotations preserve the BST property: The keys in α are less than the key of x, which is less than the key of y, and so on.



Rotations

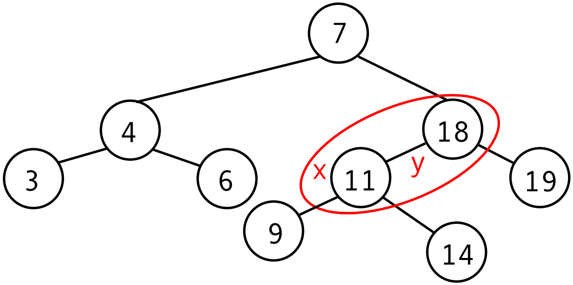
An example within a BST.



`left-rotate(T, x)`

Rotations

An example within a BST.



Rotations

Rotations take constant time since they only involve switching some pointers around. Recolouring is, of course, also done in constant time.

We will see that these two techniques are all we need to maintain the properties in a red-black tree.

Inserting to a red-black tree

We insert an element, x, to a red-black tree, T, as follows:

1. Insert x as if T were an ordinary BST – see last lecture. This step may break the RB properties.
2. Colour x red.
3. Restore the RB properties by recolouring and rotating.

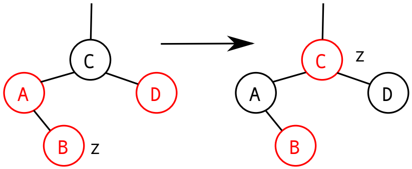
After restoring the RB properties we know that the new tree, T', is balanced (by the proof in Cormen mentioned before).

Inserting to a red-black tree

Demo

Case 1: Recolouring

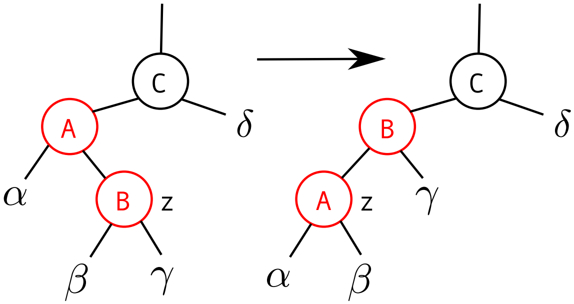
We can recolour a node whenever doing so does not change the black-heights of the tree. This occurs when the parent and uncle (other child of the grandparent) of the node are both red.



Recolouring moves the problem up the tree. *A* is shown with only one child because it doesn't matter if *B* is the right or left child.

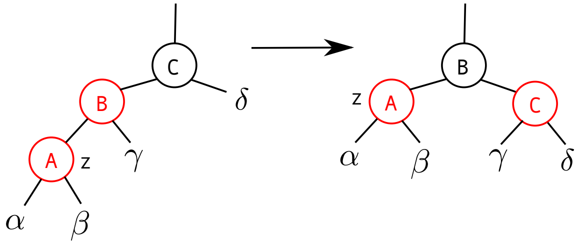
Case 2: left rotation

If we can't recolour any more, we use rotations. The first case is where z, the violating node, is the right child of its parent. We use a left rotation to achieve the situation where z is the left child.



Case 3: right rotation and recolouring

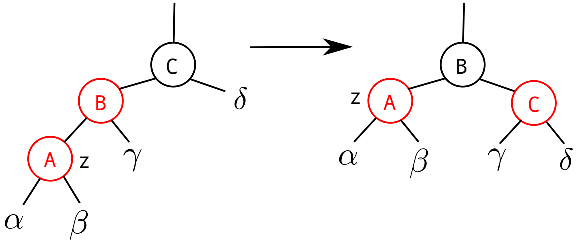
Case 2 is followed immediately by case 3, in which we use a right rotation and recolouring.



Note that case 2 falls through into case 3, but case 3 is a case of its own – i.e. if case 2 is not applicable we may still be able to apply case 3.

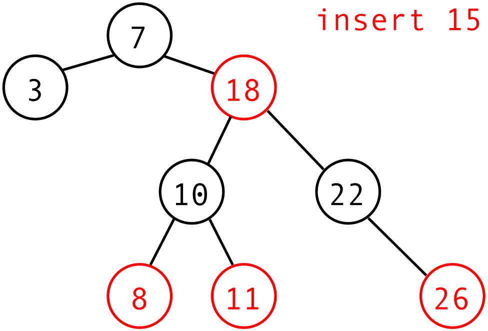
Case 3: right rotation and recolouring

Note that recolouring C is not a problem (will not produce two reds in a row) because we know that the root of the subtree δ is black – otherwise we would be in case 1. When there are no longer two red nodes in a row, the algorithm terminates.



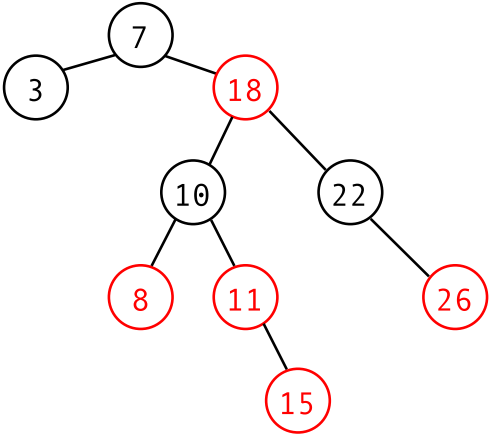
Red-black insertion
A complete example

Preparing to insert a value to a red-black tree.



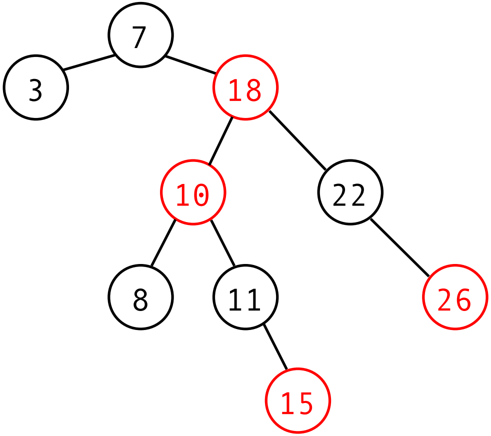
Red-black insertion
A complete example

After inserting the new node and colouring it red, we have broken property 3. Looking at the grandparent of the new node, we have a candidate for recolouring.



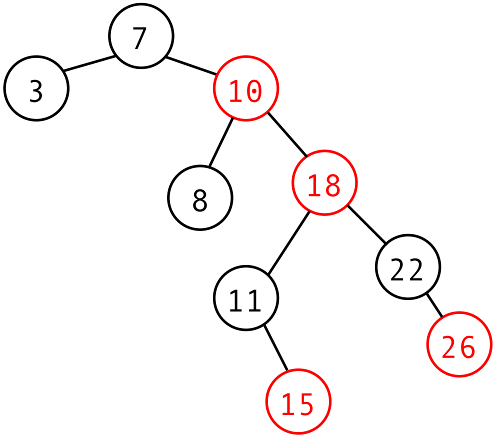
Red-black insertion
A complete example

Now the violation has moved further up the tree and we can't do any more recolouring. The violating node is the left child of its parent, so use right rotation.



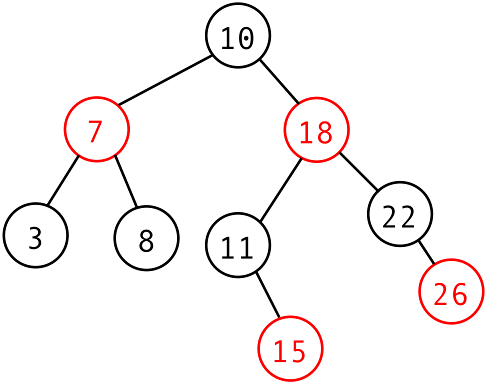
Red-black insertion
A complete example

We have straightened out the dog-leg. now the violating node is the right child of its parent. Rotate the left.



Red-black insertion
A complete example

Recolour the root and we are done.



Red-black insertion

The pseudocode for insertion to a red-black tree is quite easy to follow but a bit too long to go on a slide, simply because there are a lot of cases to consider. Again, see Cormen for an example.

Red-black deletion

Similarly to insertion, we delete from a red-black tree just as we would from a BST, then call a “fixup” routine to repair the RB properties that might have been broken in the previous step. Again, properties are fixed by recolouring and rotation.

Deleting a red node cannot violate the RB properties so we only call the “fixup” routine when the node we removed was black.

Red-black deletion

Let y be a black node removed from a red-black tree and x be the node that takes its place. What might have been broken by the removal of y?

1. If y was the root of the tree and x is red, we have violated property 2.
2. If x and $x.p$ are both red, we have violated property 3.
3. The removal of y means that there is one less black node in any path through x, so we have definitely violated property 4.

To get around the last problem we start by saying that x is either *doubly black* (if it was black to start with) or *red-black*. In this way, x contributes 2 or 1 to the black-height of any path passing through it, rather than 1 or 0.

Red-black deletion

We don't actually change the colour value of x (the node that took y's place). We keep track of it just by the fact that x is pointing to it. The goal of the deletion fixup algorithm is to move this extra blackness up the tree until:

1. x points to a red-black node, in which case we colour it back.
2. x points to the root, in which case we are done.
3. We apply recolouring and rotations until the properties are fixed.

Red-black deletion

Thus, whilst x is a non-root doubly black node, we have changes that need to be made. The cases are as follows:

1. **Case 1**: x's sibling, w, is red. Rotate and recolour.
2. **Case 2**: w is black and both of w's children are black. Recolour and move the problem further up the tree.
3. **Case 3**: w is black, $w.\mathit{left}$ is red and $w.\mathit{right}$ is black. Recolour and rotate.
4. **Case 4**: w is black, $w.\mathit{left}$ is red. Recolour and rotate.

Note that these cases aren't mutually exclusive.

Inserting to a red-black tree

Demo

Note to self: try [10, 34, 48, 79, 83], delete 10.

Next week

An overview of some algorithmic strategies: divide-and-conquer, greedy algorithms, backtracking, dynamic programming.