1. Red-Black Trees

By John Morris

A *red-black tree* is a binary search tree with one extra attribute for each node: the *colour*, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left,
                     *right,
                     *parent;
    }
```

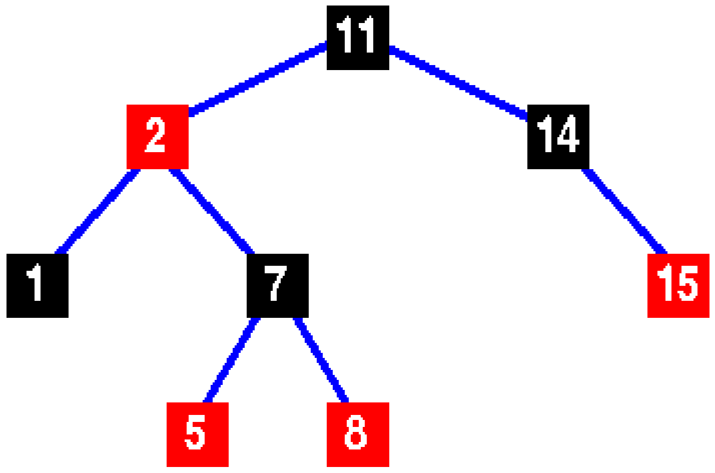
For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are coloured black.

Definition of a red-black tree

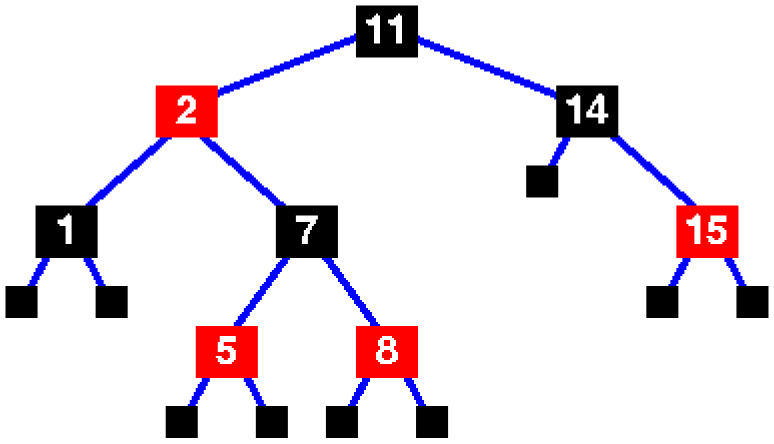
A red-black tree is a binary search tree which has the following *red-black properties*:

1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

3. implies that on any path from the root to a leaf, red nodes must not be adjacent.
However, any number of black nodes may appear in a sequence.



A basic red-black tree



Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

They are the NULL black nodes of property 2.

The number of black nodes on any path from, but not including, a node ***x*** to a leaf is called the *black-height* of a node, denoted **bh(x)**. We can prove the following lemma:

Lemma

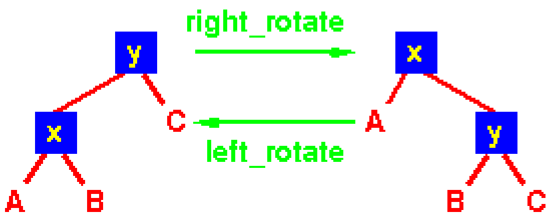
A red-black tree with ***n*** internal nodes has height at most 2**log(*n*+1)**.
(For a proof, see Cormen, p 264)

This demonstrates why the red-black tree is a good search tree: it can always be searched in **O(log n)** time.

As with heaps, additions and deletions from red-black trees destroy the red-black property, so we need to restore it. To do this we need to look at some operations on red-black trees.

Rotations

A rotation is a local operation in a search tree that preserves *in-order* traversal key ordering.



Note that in both trees, an in-order traversal yields:

```
A x B y C
```

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
    /* Set the parent to point to y instead of x */
    /* First see whether we're at the root */
    if ( x->parent == NULL ) T->root = y;
    else
        if ( x == (x->parent)->left )
            /* x was on the left of its parent */
            x->parent->left = y;
        else
            /* x must have been on the right */
            x->parent->right = y;
    /* Finally, put x on y's left */
    y->left = x;
    x->parent = y;
    }
```

Insertion

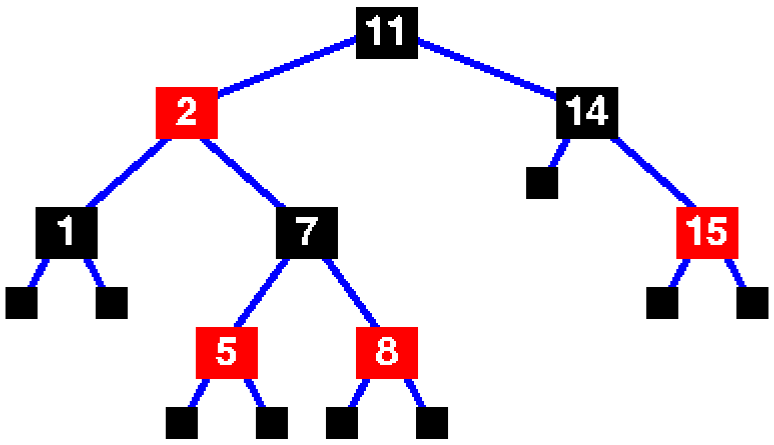
Insertion is somewhat complex and involves a number of cases. Note that we start by inserting the new node, `x`, in the tree just as we would for any other binary tree, using the `tree_insert` function. This new node is labelled red, and possibly destroys the red-black property. The main loop moves up the tree, restoring the red-black property.

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) ) {
       if ( x->parent == x->parent->parent->left ) {
           /* If x's parent is a left, y is x's right 'uncle' */
           y = x->parent->parent->right;
           if ( y->colour == red ) {
               /* case 1 - change the colours */
               x->parent->colour = black;
               y->colour = black;
               x->parent->parent->colour = red;
               /* Move x up the tree */
               x = x->parent->parent;
               }
           else {
               /* y is a black node */
               if ( x == x->parent->right ) {
                   /* and x is to the right */
                   /* case 2 - move x up and rotate */
                   x = x->parent;
                   left_rotate( T, x );
                   }
               /* case 3 */
               x->parent->colour = black;
               x->parent->parent->colour = red;
               right_rotate( T, x->parent->parent );
               }
           }
       else {
           /* repeat the "if" part with right and left
```

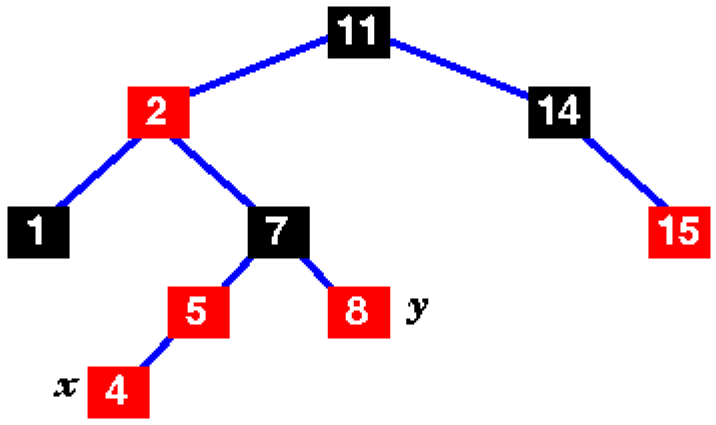
```
                  exchanged */
              }
          }
      /* Colour the root black */
      T->root->colour = black;
      }
```

Examination of the code reveals only one loop. In that loop, the node at the root of the sub-tree whose red-black property we are trying to restore, `x`, may be moved up the tree *at least one level* in each iteration of the loop. Since the tree originally has **O(log n)** height, there are **O(log n)** iterations. The `tree_insert` routine also has **O(log n)** complexity, so overall the `rb_insert` routine also has **O(log n)** complexity.

Here's an example of insertion into a red-black tree (taken from Cormen, p269).



Here's the original tree .. Note that in the following diagrams, the black sentinel nodes have been omitted to keep the diagrams simple.

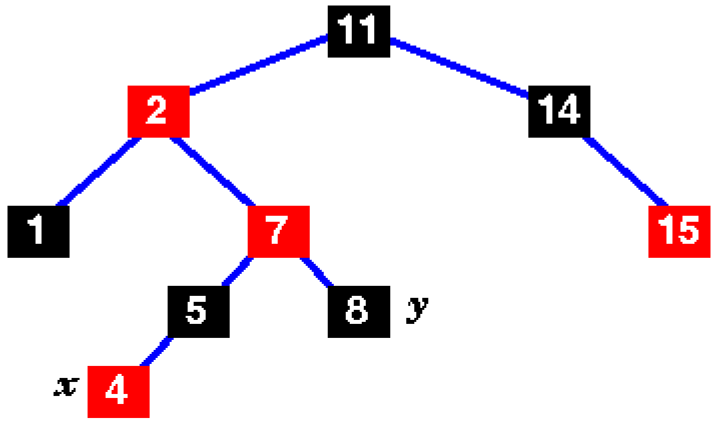


The tree insert routine has just been called to insert node "4" into the tree.

This is no longer a red-black tree - there are two successive red nodes on the path 11 - 2 - 7 - 5 - 4

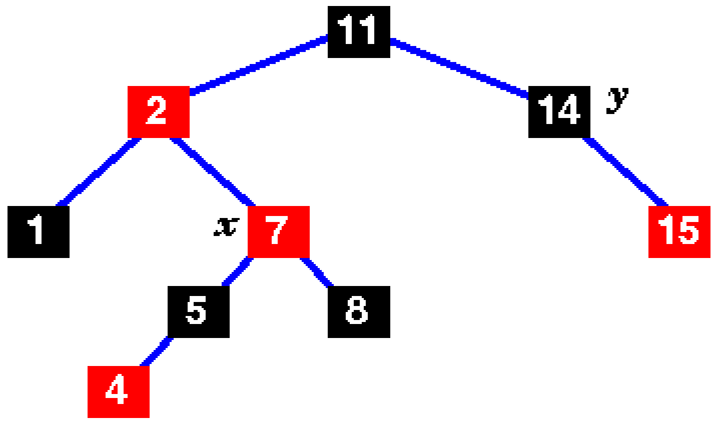
Mark the new node, x, and it's **uncle**, y.

y is red, so we have case 1 ...



Change the colours of nodes 5, 7 and 8.

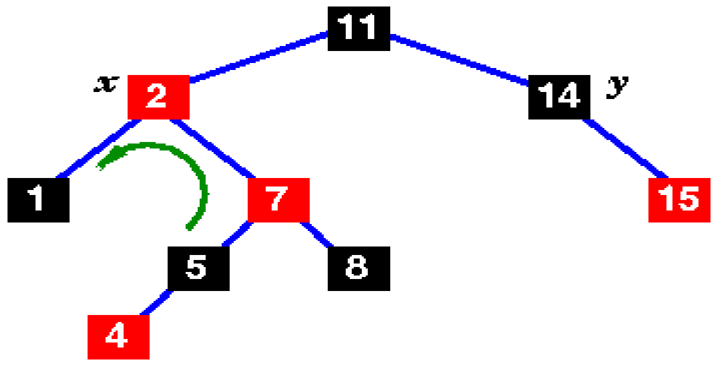
Move x up to its grandparent, 7.



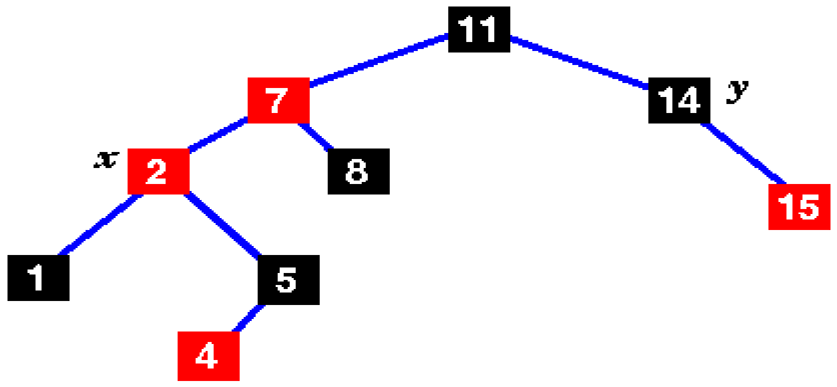
x's parent (2) is still red, so this isn't a red-black tree yet.

Mark the uncle, y.

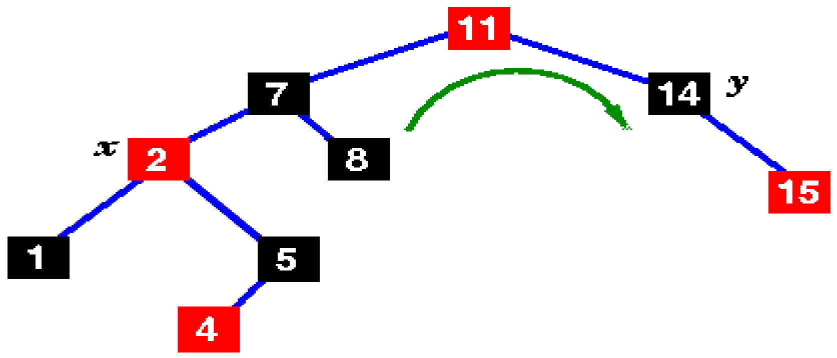
In this case, the uncle is black, so we have case 2 ...



Move x up and rotate left.



Still not a red-black tree .. the uncle is black, but x's parent is to the left ..



Change the colours of 7 and 11 and rotate right ..

B-tree

From Wikipedia, the free encyclopedia.

B-trees are tree data structures that are most commonly found in databases and filesystem implementations. B-trees keep data sorted and allow amortized logarithmic time insertions and deletions. Conceptually speaking B-trees grow from the bottom up as elements are inserted, whereas most binary trees generally grow down.

The idea behind B-trees is that inner nodes can have a variable number of child nodes within some pre-defined range. In consequence, B-trees do not need re-balancing as frequently as other self-balancing binary search trees. The lower and upper bounds on the number of child nodes are fixed for a particular implementation. For example, in a 2-3 B-tree (often simply **2-3 tree**), each internal node may have only 2 or 3 child nodes. A node is considered to be in an illegal state if it has an invalid number of child nodes.

B-trees have substantial advantages when the time needed to access another node may be significantly greater than the time needed to access values within a node. This is the case when an arbitrary node may reside on secondary storage until read into main memory, hence the use of B-trees in databases. Arranging that the node can have a relatively large number of child nodes increases the advantage.

There is some debate as to what *B* stands for. The most common belief is that *B* may stand for *balanced*, as all the leaf nodes appear at the same level in the tree (this is described as a *balanced* tree state). *B* may also stand for Rudolf Bayer, the designer of this data structure.

Inner node structures

Generally speaking, the "separation values" can simply be the values of the tree.

Each inner node has separation values which divide its sub-trees. For example, if an inner node has three child nodes (or sub-trees) then it must have two separation values a_1 and a_2. All values less than a_1 will be in the leftmost sub-tree, values between a_1 and a_2 will be in the middle sub-tree, and values greater than a_2 will be in the rightmost sub-tree.

Steps for deletion

1. If after removing the desired node, no inner node is in an illegal state then the process is finished.
2. If some inner node is in an illegal state then there are two possible cases:
    1) Its sibling node (a child of the same parent node) can transfer one of its child nodes to the current node and return it to a legal state. If so, after

updating the separation values in the parent and the two siblings the operation ends.

2) Its sibling does not have an extra child because it is on the lower bound too. In that case both these nodes are merged into a single node and the action is transferred to the parent node, since it has had a child node removed.

The process continues until the parent node remains in a legal state or until the root node is reached.

Steps for insertion

1. If after inserting the node into the appropriate position, no inner node is in an illegal state then the process is finished.
2. If some node has more than the maximum amount of child nodes then it is split into two nodes, each with the minimum amount of child nodes. This process continues action recursively in the parent node.

The action stops when either the node is in a legal state or the root is split into two nodes

Searching

Searching is performed very similar to a binary tree search, simply by following the separation values until the value is found or the end of the tree is reached.

Notes

Suppose *L* is the least number of children a node is allowed to have, while *U* is the most number. Then each node will always have between *L* and *U* children, inclusively, with one exception: the root node may have anywhere from *2* to U children inclusively, or in other words, it is exempt from the lower bound restriction, instead having a lower bound of its own (2). This allows the tree to hold small numbers of elements. The root having one child makes no sense, since the subtree attached to that child could simply be attached to the root. Giving the root no children is also unnecessary, since a tree with no elements is typically represented as having no root node.

Robert Tarjan proved that the amortized number of splits/merges is 2.

References

Original papers:

Rudolf Bayer, Binary B-Trees for Virtual Memory, ACM-SIGFIDET Workshop 1971, San Diego, California, Session 5B, p. 219-235.

Rudolf Bayer and McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, 173-189, 1972.

Summary:

Donald E. Knuth, "The Art of Computer Programming", second edition, volume 3, section 6.2.4, 1997.

External links

http://www.bluerwhite.org/btree

*B-Tree animation (Java Applet) (*http://slady.net/java/bt/*)*

*NIST's Dictionary of Algorithms and Data Structures: B-tree (*http://www.nist.gov/dads/HTML/btree.html*)*

*B-tree algorithms (*http://www.semaphorecorp.com/btp/algo.html*)*

*B-tree variations (*http://www.semaphorecorp.com/btp/var.html*)*

*Source code for a balanced tree (B-tree) (Windows required for test timings) (*http://hyperbolique.net/btree.html*)*