9 Red-Black (search) trees

9.1 Definition of redblack trees

Bayer (1972), also Guibas and Sedgewick (1978). See Wikipedia for later improvements.

There are ways of keeping a binary search tree sufficiently well balanced so that the worst-case cost of search, insert, and delete, is $O(\log n)$ (in a tree with n nodes): AVL trees, 2/3-trees, for example, but we focus on red-black trees, which seem to be the best.

A red-black tree is a binary tree in which every node has a colour, red or black, with the following two properties.

No double red: every red node has a black parent.

Rank balancing: for every node v, for every leaf descendant x of v, the path from v to x in the tree contains the same number of *black* nodes. This number is called the *rank* of v.

Rank balancing can be introduced another way.

Definition of $\mathrm{rank}(v)$**:**

$$\begin{array}{c}\text{maximum rank of its children if } v \text{ is red}\\ 1 + \text{maximum rank of its children if } v \text{ is black}\end{array}$$

and

Balancing:

$$\text{both children of } v \text{ have the same rank.}$$

We write 'both children' on the understanding that if one or both children are missing then they are treated as if their ranks were zero.

9.2 Depth of a red-black tree

(9.1) Lemma *Let v be a node in a red-black tree. If v has rank k, then it has at least $2^k - 1$* black *descendants.*[1]

Proof. By an inductive argument. If u is a leaf, then either it is red with rank 0 and $2^0 - 1$ black descendants, or it is black with rank 1 and $2^1 - 1$ black descendants.

Induction: Suppose that v has one child u or two children u and w. If v has only one child,[2] u, say, then u is red and v is black with rank 1 and $2^1 - 1$ black descendants.

If v has two children u and w, then by rank balancing they have the same rank ℓ. If v is red then v has rank ℓ and it has at least $2(2^\ell - 1)$ black descendants (induction) which is at least $2^\ell - 1$ since the latter is nonnegative.

If v is black then it has rank $\ell + 1$ and each child has at least $2^\ell - 1$ black descendants; since v is also black, it has at least $2^{\ell+1} - 1$ black descendants, as required. ∎

(9.2) Lemma *Let T be a nonempty red-black tree whose root has rank k. Then T has height at most $2k$.*

[1] This corrects an error in the notes.

[2] Another error corrected.

Proof. Let u be any node in T, and let r be its rank. Claim that the height of u is at most $2r$.

Consider a *longest* path from u to a a leaf descendant x. This is a sequence of nodes; let R be the subsequence of red nodes and B the subsequence of black nodes. Let R' be the subsequence of red nodes *excluding* u if u is red. Then for every node v in R' its parent is in B, so $|R'| \leq |B| = r$. Therefore $|R| \leq r + 1$ and the path contains at most $2r + 1$ nodes. The number of nodes in the path, minus 1, is the height of u, and is at most $2r$. Or the tree has height at most $2k$. ∎

(9.3) Corollary *A red-black tree with n nodes has height $O(\log n)$.*

Proof. Let h be the height of T and k the rank of its root.[3] T has at least $2^k - 1$ black nodes,[4] so therefore

$$\begin{aligned} 2^k - 1 &\leq n \\ 2^k &\leq n + 1 \\ k <&\leq \log_2(n + 1) \\ h \leq 2k &\leq \log_2(n + 1) \end{aligned}$$ ∎

9.3 Operations on red-black search trees

A red-black search tree is a red-black tree whose nodes carry sortable keys in ascending order.

Operations search, insert, and delete follow the pattern for general binary search trees, but insertion will usually lead to a *double red* situation in which exactly one node has a red parent, or a *rank deficit* situation in which exactly one node has a sibling of different rank.

Searching is $O(\log n)$. Procedure to fix double red and rank deficit are illustrated below. They also are $O(\log n)$.

9.4 Rotation

In summary, a red-black tree supports search, insert, and delete, in time $O(\log n)$.

The fix double red and fix rank deficit procedures involve

- Promotion: change red nodes to black
- Demotion: change black nodes to red
- Rotation: as illustrated. The management of colours is a separate issue, but the main point is that all the actions except promotion or demotion involve one or two rotations, and *rotations preserve inorder.*

[3] Possibly T is empty, but that is trivial.

[4] Another correction,

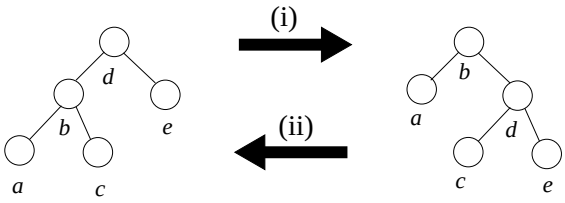


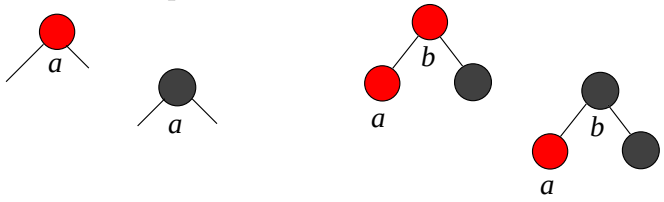
Figure 1: Rotation: (i) rotate left child b up; (ii) rotate right child d up.

9.5 Fix double red

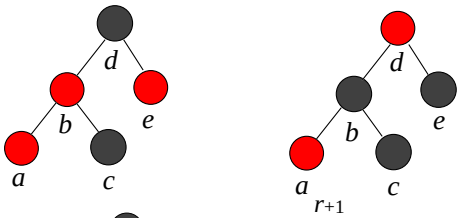
Generally, p will point to a red node, and it will have a red parent, though later on this may not be true.

In the simplest case p points to a red root with no parent.

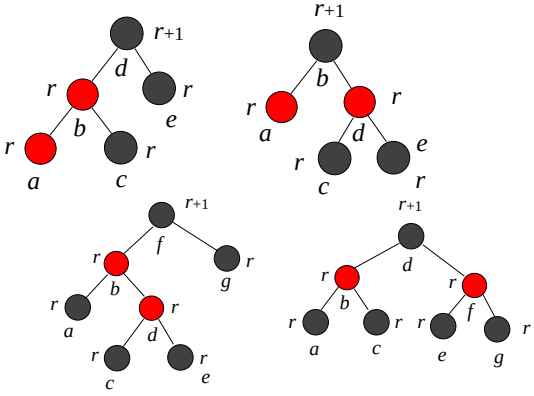
Case: root a is red.
Make it black and stop.
Case: red parent b is root.
Make it black and stop.



Case: p points to a red node a, red parent b, black grandparent d, red sibling e of b. 'Promote': Make b and e black and d red, let $p = d$, and continue.



Case: p points to a red node a whose rank is r, with various other nodes as shown. Rotate b upwards and change the colours as shown. The problem is fixed.

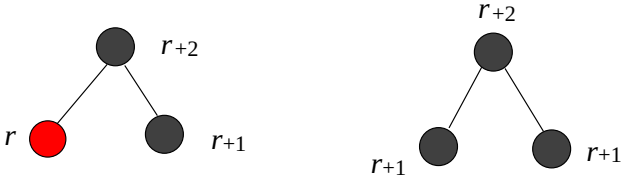


Case: p points to d as shown. Rotate d upwards, and rotate it upwards again to obtain the layout shown. Adjust the colours as shown. The problem is fixed.

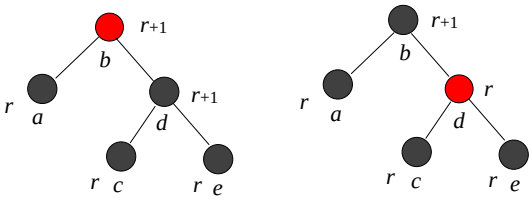
9.6 Fix rank deficit

Two siblings have different ranks, r and $r + 1$. In general we assume that p points to the deficient node, though initially the deficient node could be null.

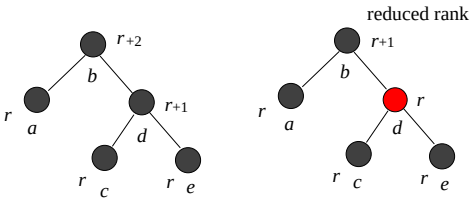
Case: The deficient node is red. Make it black and stop.



Case: the deficient node a is black, with red parent b and black sibling d both of whose children are null or black. Make d red and b black and stop.



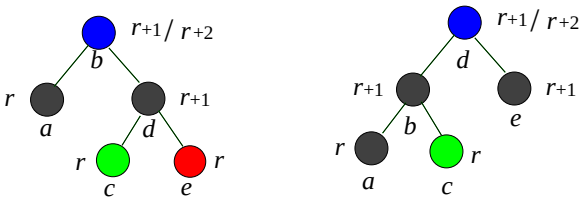
Case: p points to the deficient node a which is black, with black parent b and black sibling d both of whose children are null or black. **Demote:** make d red and continue with p pointing to b (whose rank has dropped by 1).



In the later figures, colours green and blue are used where both red and black are possibilities.

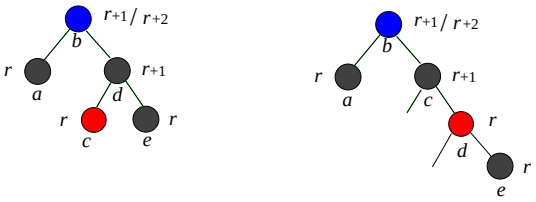
Case: p points to a black node a, its sibling d is also black, and the child e of d, which is more distant from a, is red. Rotate d upwards and adjust the colours as shown. Done.[a]

[a] An error in the figure, the new rank of b, is corrected.

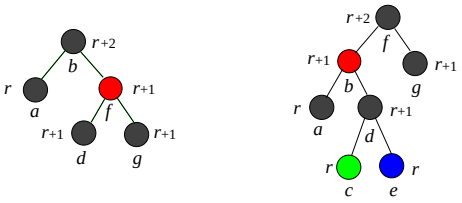


Case: p points to a black node a, its sibling d is also black, the child c of d, which closer to a, is red, and its sibling e is null or black. Rotate d upwards, and adjust the colours as shown. The left child of d after rotation was the right child of c before rotation and is black, so the no-double-red condition holds. There is still a rank deficit at a, but the altered tree fits the previous case and is corrected in one step.[a]

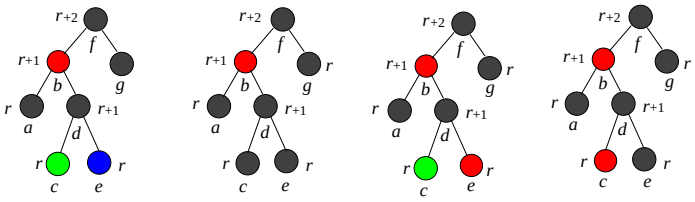
[a] This corrects a mistake in the notes.



Case: p points to a black node a, and its sibling f is red. Rotate f upwards and adjust colours as shown. The node a is still deficient, but the cases raised have already been dealt with; they depend on the colours of the children c and e of d.



Here are the three different possibilities depending on the colours of c and e:



9.7 Join and split

Join. Given red-black search trees T_1, T_2, and a key x, where the keys in T_1 are less than x and those in T_2 are greater than x, it is possible to join T_1, x, and T_2 into a single red-black search tree, in $O(\log n)$ operations.

Of course the trees T_1 and T_2 are lost.

This is the general idea.

- Create a new node N containing the key x. Let r_1 and r_2 be the ranks of T_1 and T_2.
- If $r_1 = r_2$, then N will be the root of the new tree, black, with left and right subtrees T_1 and T_2.
- If $r_1 < r_2$, go down the left branch of T_2 until a *black* node y of rank r_1 is found. Let z be the parent of y in T_2. Make N red, give it left subtree T_1 and right child y, and make N the left child of z. This can create a double red; call fix double red.
- If $r_1 > r_2$: similar.

Split. Given a red-black search tree T and a key x occurring in T, it is possible in $O(\log n)$ operations to break up T into two search-trees, one, T_1, containing all keys $< x$ and the other, T_2, containing all keys $> x$. (The key x is separated from these trees.)

Of course the structure of T is lost.

The idea is: suppose that N was the node containing x in the tree. Follow the path from N to the root, identifying subtrees to be added to T_1 and T_2. This is somewhat tricky.

T_1 and T_2 can be built by repeated joining.

This involves $O(\log n)$ join operations each costing $O(\log n)$.

However with care and ingenuity the split operation can be accomplished in $O(\log n)$ operations rather than $O((\log n)^2)$. It's somewhat complicated to accomplish and more complicated to analyse.