Red-Black Trees

Lars Arge and Michail Lagoudakis

October 6, 1999

1 Binary Search Trees

We are interested in maintaining an ordered set S under the following operations:

- $Search(S, e)$: Return a pointer to the element e in S, if $e \in S$.
- $Insert(S, e)$: Insert the element e in S, if $e \notin S$.
- $Delete(S, e)$: Delete the element e from S, if $e \in S$.
- $Successor(S, e)$: Return the minimal element in S larger than e.
- $Predecessor(S, e)$: Return the maximal element in S smaller than e.

A *binary search tree* T is defined recursively; T consists of a node containing a single element $x \in S$ and two (possibly empty) subtrees T_l and T_r. The node containing x is called the *root* of T and T_l (T_r) the left (right) subtree. The root of the left (right) subtree is called the left (right) child of x. The elements in the tree satisfy the *search tree property*; all elements in T_l are smaller than x and all elements in T_r are larger than x—refer to Figure 1. A binary search tree on a set S of size n uses $\Theta(n)$ space.

Searching for an element e in a search tree is done using a simple recursive procedure; e is compared to the element stored in the root of the tree. If e is smaller, the search is recursively continued in the left subtree and otherwise the search is continued in the right subtree. As $O(1)$ time is spent in each node the total time used is $O(h)$, where h is the height of the tree. The **successor** or **predecessor** of e can be found in a similar way.

Insertion of an element e consists of searching for e and creating a new node at the place where the search path terminates—refer to Figure 2. Thus an insertion takes $O(h)$ time.

Deleting e is a little more involved. First a search is made for the node v holding e. If v has no children, it can simply be deleted. If v has one child, v is deleted and its single child is attached to v's parent—refer to Figure 3. In the case where v has two children, the successor of e is first found. This corresponds to finding the minimum element in the right subtree of v. Thus the node w containing

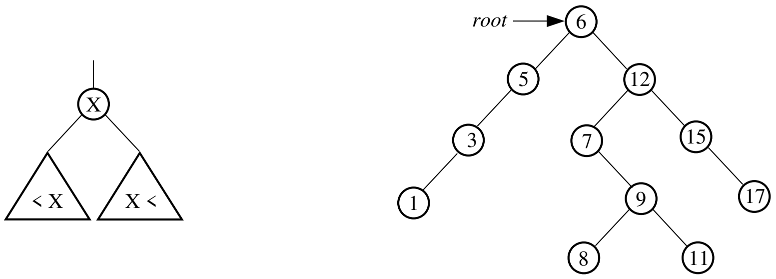

Figure 1: The search tree property (left) and an example of a binary search tree (right).

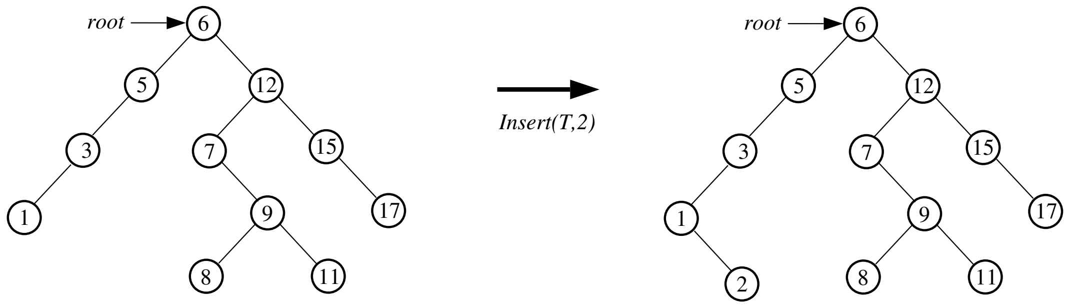

Figure 2: Inserting a node.

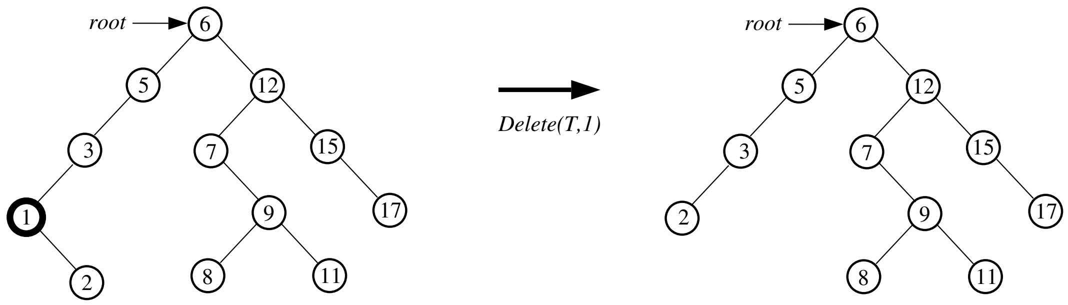

Figure 3: Deleting a node with one child.

the successor can be found simply by following left branches as long as possible, starting in the right subtree of v. Then the element in w is copied into v (the binary search tree property is preserved) and w is deleted. As w can have at most one child it can be deleted as discussed above—refer to Figure 4. Deletion takes $O(h)$ time.

All operations on a binary search tree take $O(h)$ time; h can be anywhere between $\log n$ (balanced tree) and n (unbalanced tree). In the following we discuss how to maintain the tree (relatively) balanced during updates.

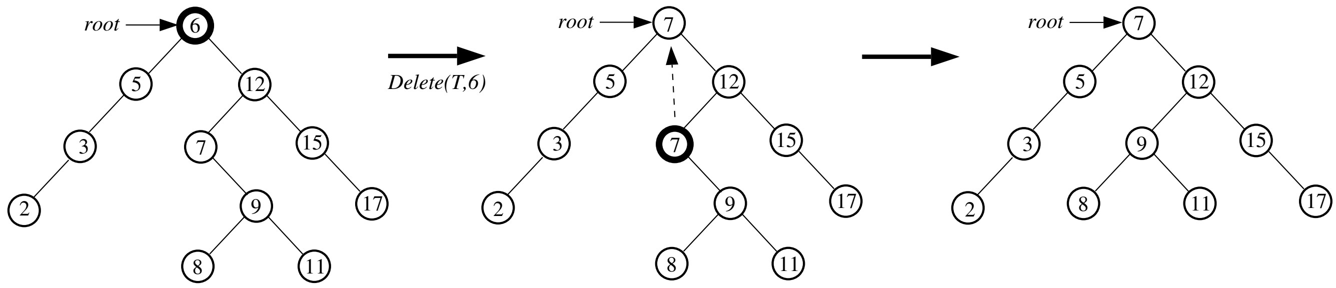

Figure 4: Deleting a node with two children.

2 Red-Black Trees

A *Red-Black tree* is a binary search tree where each node is colored either Black or Red. The colors are used to keep the tree balanced during update operations such that the height of the tree is $\Theta(\log n)$.

The following invariant must be satisfied at all times.

Red-Black Tree Color Invariant

1. The root is colored Black.
2. A Red node can only have Black children.
3. Every path from the root to a leaf contains the same number of Black nodes.

Here, "leaf" means a node with less than two children.

Lemma 1 *A red-black tree with n nodes has height $\Theta(\log n)$.*

Proof. All root-leaf paths must have the same number of Black nodes but we can have a Red node between every pair of Black nodes. This means that $h_{\max} \leq 2h_{\min}$, where $h_{\max}$ and $h_{\min}$ are the lengths of the longest and the shortest path in the tree, respectively. Then, using the fact that a complete binary tree with height h has $2^{h+1}-1$ nodes, we have

$$2^{h_{\min}+1} - 1 \leq n \leq 2^{h_{\max}+1} - 1 \implies h_{\min} \leq \log(n+1) - 1 \leq h_{\max} \leq 2h_{\min} \implies$$

$$(1/2)(\log(n+1)-1) \leq h_{\min} \leq \log(n+1) - 1 \implies h_{\min} = \Theta(\log(n+1)-1) = \Theta(\log n)$$

Since $h_{\min} \leq h_{\max} \leq 2h_{\min}$, it is also true that $h_{\max} = \Theta(\log n)$. ∎

2.1 Insertion in Red-Black Trees

An insertion in a red-black tree is initially the same as an insertion in a binary search tree; the new element is inserted in a leaf that is created at the appropriate place in the tree. The question is what color we should give to the new node. If it has a Red parent we cannot color it Red because of the second part of the invariant. We cannot color it Black either, as the path from the root to the new node would then get one more Black node than other root-leaf paths, violating the invariant. The new node is what we call "problematic".

It turns out that we can always either resolve the problem with some local rearrangement and recoloring of nodes, or push the problem up in the tree (two levels at a time). In the worst case it will be pushed all the way to the root, where it can be trivially handled. Thus the insertion procedure takes $O(\log n)$ time.

To present the recoloring and rebalancing process, we imagine coloring the problematic node Green to indicate that it is yet to be colored. During the process, we will maintain the *insertion invariant* that the problematic node has only Black children (if any). This is true just after the insertion as the new node has no children. We now have the following cases (the symbols we use are summarized in Figure 5):

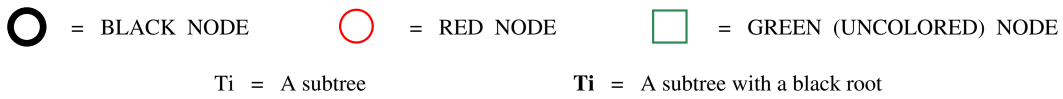

Figure 5: Symbols used in the figures.

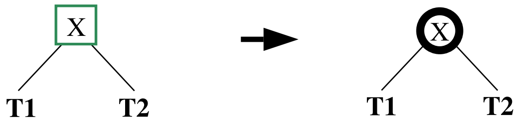

Figure 6: The problematic node is the root (case 1).

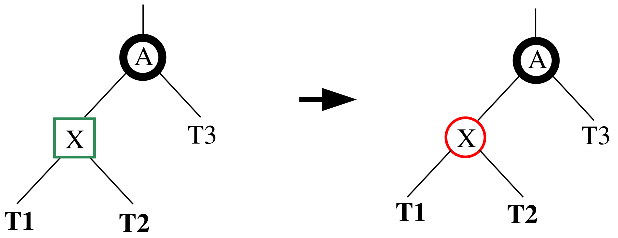

Figure 7: The parent of the problematic node is Black (case 2).

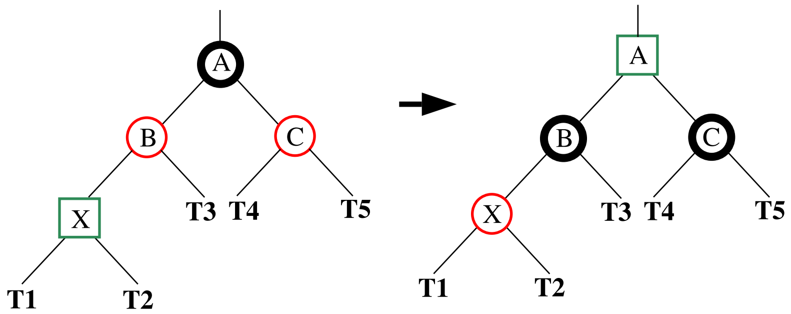

Figure 8: The parent and the uncle are both Red (case 3 a)).

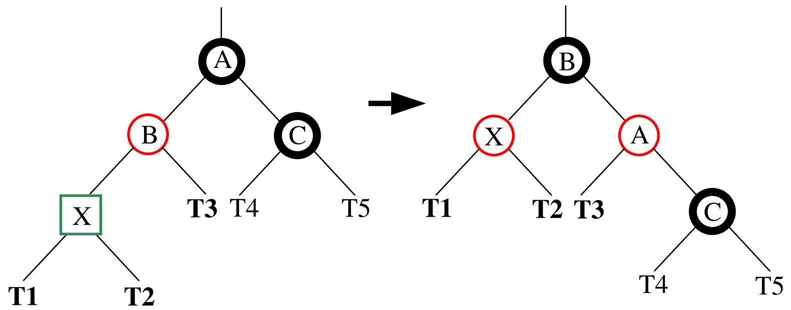

Figure 9: The parent is Red and the uncle is Black (case 3 b) 1).

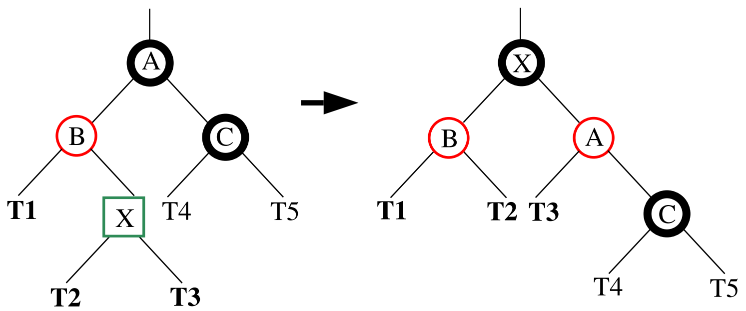

Figure 10: The parent is Red and the uncle is Black (case 3 b) 2).

1. ***Parent does not exist (problematic node is the root)***

   We color the problematic node X BLACK, as illustrated in Figure 6. The invariant is maintained as *all* root-leaf paths get one more BLACK node.

2. ***Parent is*** BLACK

   This case, illustrated in Figure 7, is also simple. We can safely color the problematic node X RED as it is not the root and as the subtrees **T1** and **T2** have BLACK roots.

3. ***Parent is*** RED

   This case is more involved and we have to distinguish between several cases depending on the color of X's uncle C. Note that X's parent B cannot be the root of the tree (it is RED).

   (a) ***Uncle exists and is*** RED

   We recolor the nodes as illustrated in Figure 8 (we only show one of several symmetric cases); We color X RED. To do so, we have to make sure that its parent is BLACK. We make it BLACK by "pushing" the BLACK node from A one level down, "splitting" it in two. After the rearrangement the problematic node has moved two levels up the tree to A. Note that we maintain the insertion invariant as the problematic node has BLACK children.

   The red-black color invariant is maintained as X has BLACK children (so RED is a valid color for it), and the number of BLACK nodes on each path from A to Ti remains unchanged (=1).

   (b) ***Uncle is*** BLACK ***or non-existing***

   We restructure and recolor the tree as shown in Figures 9 and 10 (again symmetric cases are not shown).

   The binary search tree property is maintained as the order of the elements in the tree is the same before and after the transformation ($T1, X, T2, B, T3, A, T4, C, T5$ in the first case and $T1, B, T2, X, T3, A, T4, C, T5$ in the second case). The red-black invariant is fulfilled after the transformation/recoloring; the RED nodes can only have BLACK children (because the subtrees **T1**, **T2** and **T3** have BLACK roots), and the number of BLACK nodes on paths to subtrees **T1**, **T2**, **T3**, T4, and T5 is the same before and after the transformation.

Remarks:

As discussed, every transformation either resolves the problem or propagates it up in the tree (Figure 8). As each transformation involves a constant number of pointer and color changes they can all be performed in $O(1)$ time. Thus an insertion takes $O(\log n)$ time.

Even though the restructurings in Figures 9 and 10 seem complicated they are actually obtained using one fundamental operation; a *rotation*. A rotation is a local restructuring operation that modifies the tree locally while maintaining the search tree property. Figure 11 a) illustrates a (right) rotation. The transformation on Figure 9 corresponds to such a rotation. The transformation on Figure 10 can be obtained using two rotations, or a *double rotation* as illustrated in Figure 11 b).

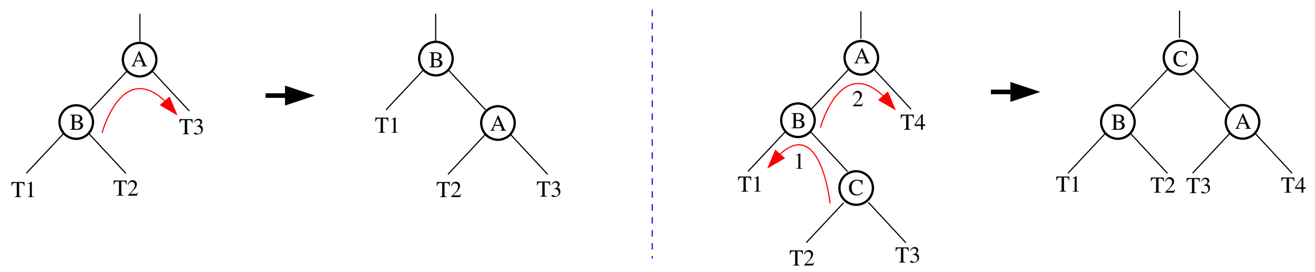

Figure 11: a) Single rotation. b) Double rotation.

2.2 Deletion from Red-Black Trees

Like an insertion, a deletion from a red-black tree initially proceeds as a deletion in a normal binary search tree. Only once the only step left is the deletion of a node X with less than two children we need to make sure the red-black tree coloring invariant is maintained. If X is RED, we can just remove it. Similarly, if X is BLACK and its single child is RED, we can safely remove X and color the child BLACK.

So, we are left with the (problematic) case where X is BLACK with a BLACK child or no children at all. As in the insertion case, it turns out that we can always resolve the problem either by some local rearrangements and recolorings, or by pushing the problematic BLACK node up in the tree (two levels at a time). We need to consider several cases. As previously, we depict the problematic node X as a square, but unlike in the insertion case, X is BLACK. We also maintain the *deletion invariant* that the problematic node has a BLACK son (if any).

1. ***Parent does not exist (the problematic node is the root)***

   We remove X as illustrated in Figure 12. The invariant is maintained; the new root is BLACK and all root-leaf paths have one less BLACK node than before the transformation.

2. ***Parent is*** RED

   Since the parent of X is RED, X must have a sibling (otherwise a root-leaf path through X would have at least one more BLACK node than the path terminating at the parent). Moreover, the sibling has to be BLACK. There are two cases, depending on the color of the nephew of X.

   (a) ***Nephew is*** BLACK ***(or non-existing)***
   The problematic node is removed as illustrated in Figure 13. The transformation maintains the invariants; the rotation maintains the search tree property and the number of BLACK nodes on all paths is maintained.

   (b) ***Nephew is*** RED
   The problematic node is removed with a double rotation as illustrated in Figure 14. The invariant is again maintained; the search tree property is maintained, C cannot be the root—since A is not the root—so the color of the root is not affected, and the number of BLACK nodes on all paths is maintained.

3. ***Parent is*** BLACK

   As in the previous case, X must have a sibling. However, unlike previously, X's sibling can be of any color:

   (a) ***Sibling is*** RED
   We resolve the problematic situation as illustrated in Figure 15; We perform a single rotation that, interestingly, moves the problematic node down! Luckily, the resulting situation is one where the parent of the problematic node is RED and the sibling is BLACK. This situation can be fully resolved locally as described above. As previously, it is easy to see that the invariant is maintained by the transformation.

   (b) ***Sibling is*** BLACK
   There are several cases, depending on the coloring of the nephews of X.

      i. ***Both Nephews are*** BLACK
      We push the problem up the tree as illustrated in Figure 16. In the new configuration the problematic node has a BLACK son as required. The invariant is also maintained since the recoloring of B means that the number of BLACK nodes on all root-leaf paths is maintained.

ii. ***One or both Nephews are*** RED
Depending on which of the nephews is RED we resolve the problem using either a single or a double rotation as illustrated in Figure 17 (if both nephews are RED any of the two can be used). As previously, it is easy to check that the invariant is maintained after the transformations.

Remark: An important feature of the red-black tree is that an update only results in $O(1)$ node rotations. The number of recolorings, however, can be up to $\Theta(\log n)$.

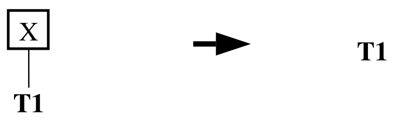

Figure 12: The problematic node is the root.

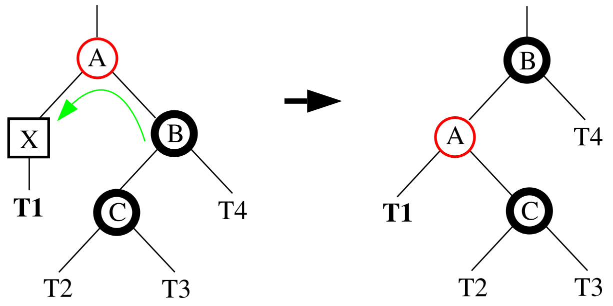

Figure 13: The parent is RED (sibling is BLACK) and nephew is BLACK.

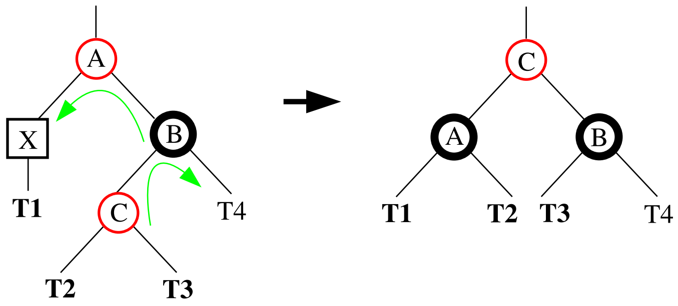

Figure 14: The parent is RED (sibling is BLACK) and nephew is RED.

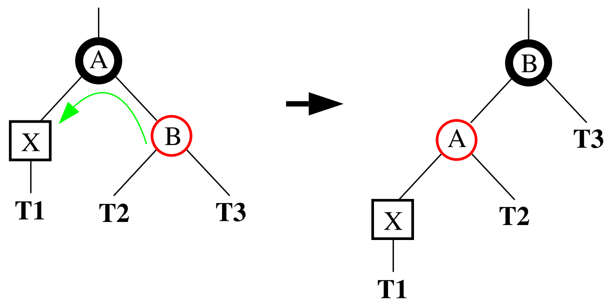

Figure 15: The parent is BLACK and the sibling is RED.

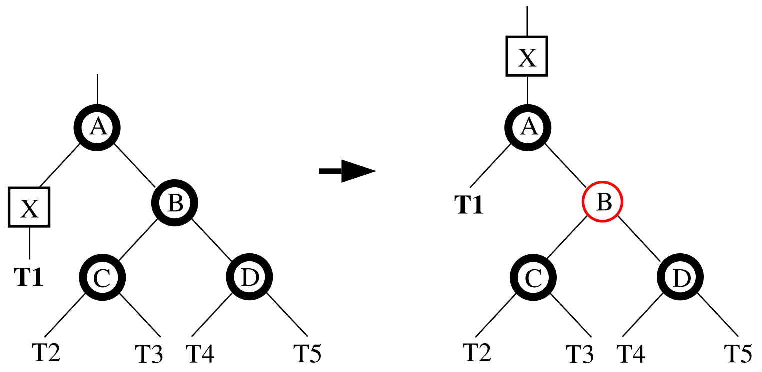

Figure 16: The parent is BLACK, the sibling is BLACK, and both nephews are BLACK.

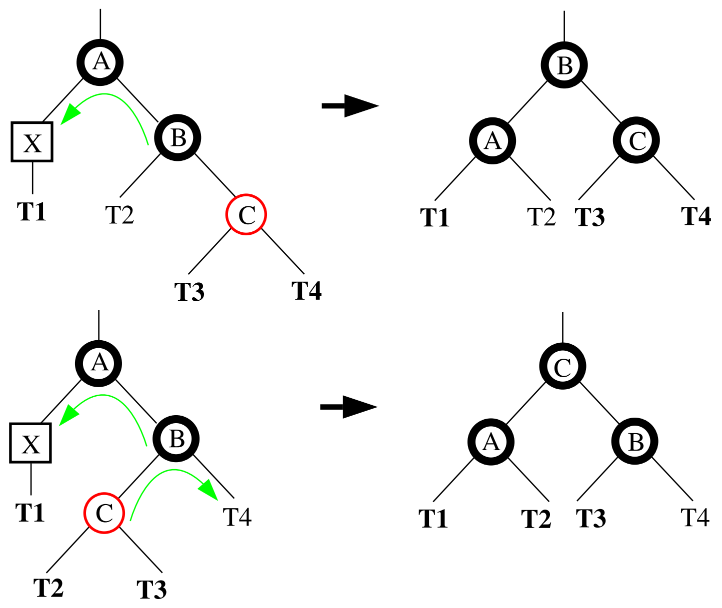

Figure 17: The parent is BLACK, the sibling is BLACK, and a nephew is RED.