Red-Black Trees

CPS 230 - Lecture Notes

Lars Arge and Michail Lagoudakis
Department of Computer Science
Duke University
Durham, NC 27708

1 Dynamic Ordered Sets

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

- $Search(S, x)$: Return a pointer to the element x, if $x \in S$, NIL otherwise.
- $Insert(S, x)$: Insert the element x in the set S, if $x \notin S$.
- $Delete(S, x)$: Delete the element x from the set S, if $x \in S$.
- $Successor(S, x)$: Return the minimal element in S which is greater than x, or NIL if x is the maximum element in S.
- $Predecessor(S, x)$: Return the maximal element in S which is less than x, or NIL if x is the minimum element in S.

Our goal is to design a data structure that implements *all* the above operations as *efficiently* as possible.

Note that a set contains distinct elements. Sometimes we are interested in a collection C of elements, whereby several copies of an element are allowed in the collection. The above operations can be extended for this case. We discuss sets here, unless otherwise noted.

In what follows, n indicates the number of elements in the set S at any time, i.e. $n = |S|$. Operation costs are expressed in terms of n.

2 Simple Implementations

2.1 Double Linked List

The elements are stored in nodes which are linked by pointers. Each node holds one element and points to two other nodes (one to the left and one to the right). All nodes together form a doubly linked linear chain. One end node is identified as the *head* of the list and the other as the *tail*. The list can be ordered or unordered. The space taken by this structure is $\Theta(n)$ (one node of constant size per element, plus a pointer to the head). This implementation results in the following time costs for the set operations.

- $Search(S, x) : O(n)$
  Start from the head and go through the list until x is found or the tail has been reached. In the case of an ordered list, search can terminate earlier.
- $Insert(S, x) : O(n)$
  First search for x in time $O(n)$. If found return, else insert x at the head and update the pointers locally ($O(1)$) for the unordered list. For the ordered list, identify the two nodes —between which x is to be inserted— in time $O(n)$ (this can be done during the search for x). Insert locally, updating the pointers ($O(1)$).

- *Delete*$(S, x) : O(n)$
  First search for x ($O(n)$). If not found, return. If found, delete x and update the pointers locally ($O(1)$).

- *Successor*(S, x), *Predecessor*$(S, x) : O(n)$
  Search for x ($O(n)$) and report the next (or the previous) element by following the appropriate pointer in the case of an ordered list. For an unordered list, go through the whole list again ($O(n)$) to find the closest-to-x element from above (or below). For the ordered and unordered case, if such an element does not exist, return NIL.

Notice that for a collection C of elements, insertion in an unordered list can be done in $O(1)$ time at the head of the list. All other costs remain the same.

2.2 Array

We store the elements of S in a 1-dimensional array one after the other in a sorted manner. The space taken by this structure is $\Theta(n)$ (one array cell per element). This implementation results in the following time costs for the set operations:

- *Search*$(S, k) : O(\lg n)$
  Search using *Binary Search*.

- *Insert*(S, x), *Delete*$(S, x) : O(n)$
  For Insert, first search for the element x as above ($O(\lg n)$). If found, then return, else make space for x by shifting the subsequent elements to the right by one cell ($O(n)$) and insert. For Delete, first search for x ($O(\lg n)$). If not found, then return, else delete x and shift the subsequent elements to the left by one cell ($O(n)$).

- *Successor*(S, x), *Predecessor*$(S, x) : O(\lg n)$
  Search for x ($O(\lg n)$) and report the next (or the previous) element ($O(1)$). If no such element exists, return NIL.

For a collection C of elements all costs remain the same.

Question : Can we do better (and without using indirect addressing, that is assuming a pointer machine model of computation)?
Answer : Yes! Read below.

3 Binary Search Trees

A *binary search tree* is an abstract data structure consisting of nodes, such that:

- Each node holds a single element.
- Each node has three pointers to three other (distinct) nodes : *left child, right child* and *parent.*
- Each node has 0, 1 or 2 children and exactly 1 parent (except one node). The single node with no parent is the *root* of the tree. Unused pointers are labeled with NIL.
- All nodes together form a (possibly incomplete) binary tree.
- All nodes satisfy the *search tree property*, that is, if a node p holds element x, then (see Figure 1)
  - All elements in the left subtree of p are less than or equal to x.

– All elements in the right subtree of p are greater than x.

The left (right) subtree of p is the binary tree rooted at the left (right) child of p.

The space taken by a binary search tree is $\Theta(n)$, since there is one node per element and each node contains a constant number of items (the element itself and three pointers). There is also a single pointer to the root of the tree.

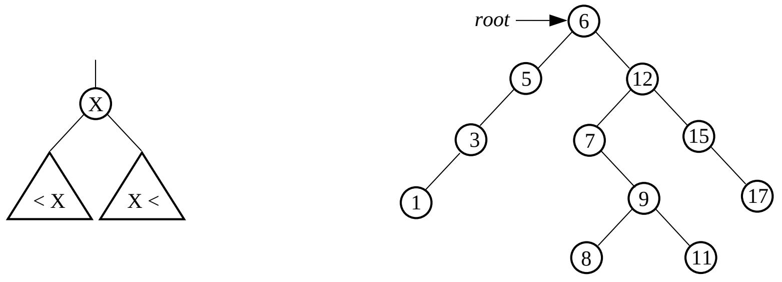

Figure 1: The search tree property (left) and an example of a binary search tree (right).

The elements of a set S stored in a binary search tree can be easily printed in sorted order using the following recursive procedure, called as InOrder($root$).

```
InOrder($v$)
  if ($v \neq$ NIL)
     InOrder($v.leftchild$)
     Print($v.element$)
     InOrder($v.rightchild$)
```

This actually implements an inorder traverse of the tree and it takes $\Theta(n)$ time since it is called twice for each node and printing takes $O(1)$ time.

The set operations on a binary search tree can be implemented in the same recursive manner. For example, to search for an element x in the tree, we call the following as Search($root, x$).

```
Search($v, x$)
  if ( ($v$ = NIL) or ($v.element = x$) )
     return $v$
  if ($v.element > x$)
     Search($v.leftchild, x$)
  else
     Search($v.rightchild, x$)
```

Notice that this corresponds to a descent from the root to some node, where $O(1)$ time is spent on each node. Since the search path cannot be longer than the height of the tree, the total time is $O(h)$, where h is the height of the tree.

Finding the successor of x requires a search for x and then finding the minimum element in the right subtree emanating from as deep as possible in the search path from the root to x. This can require at most an ascent through the path all the way to the root. The minimum element in a (sub)tree can be easily found by following left branches from the root of the (sub)tree, until there are no more. Similarly, we can find the predecessor, with the difference that we are looking for the subtree at the deepest left branch 'touching' the seach path, and the maximum in that subtree. In any case, this involves at most two descents and one ascent through the tree. So, the total time is linear in the length of the search path, or $O(h)$, where h is the height of the tree.

Insertion of x requires searching for x and creating a new node at exactly the place where the search path terminates. All we need to know is the parent of the new node in order to update (locally) the pointers as needed. Again this takes $O(h)$ time. Figure 2 shows an example.

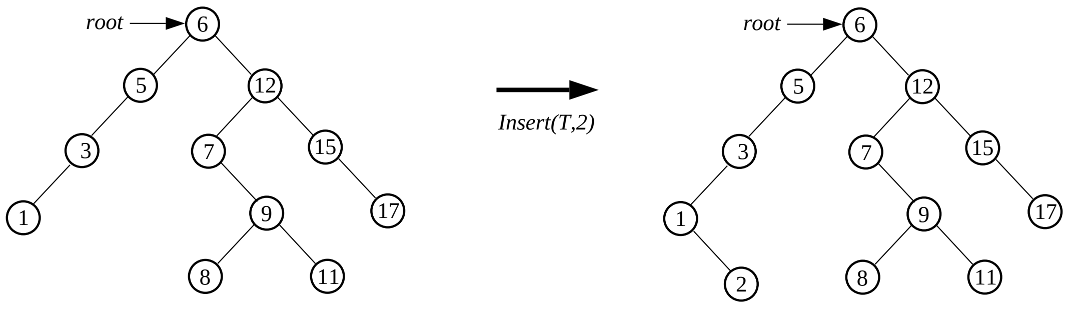

Figure 2: Insertion in a binary search tree.

Deletion is a little more involved. First we search for the node v that holds x. If v has no children, then we just delete v. If v has just one node, we delete v and its single child is attached to v's parent (at the place where v used to be). In both cases, all we need to do is some local updating of the pointers. Figure 3 shows such an example.

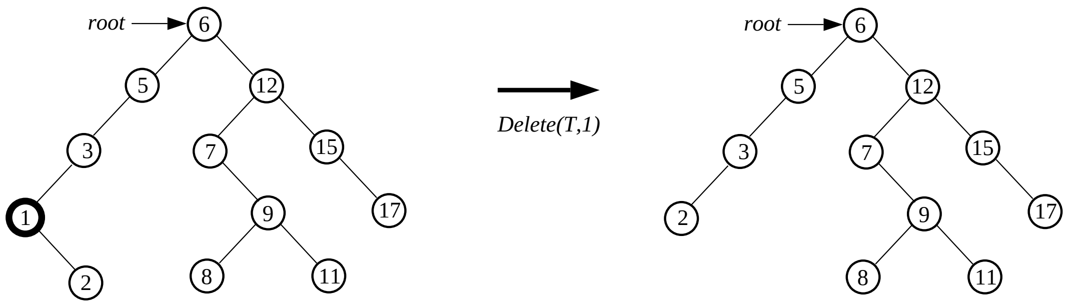

Figure 3: Deletion in a binary search tree.

In the case where v has two children, then we look for the successor of x, as described above. Alternatively, we could find the predecessor of x. Call this node w. Finding w takes at most $O(h)$. In either case, we copy the element of w into v (the binary search tree property is preserved) and we delete w. Notice that w can have at most one child, so it is easy to delete w with no extra effort. Figure 4 illustrates this case. In any case, the total time for deletion is $O(h)$.

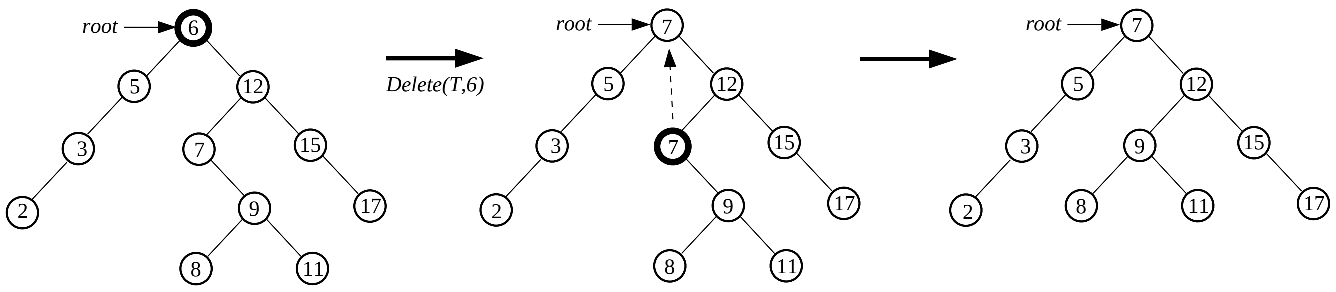

Figure 4: Deletion in a binary search tree.

All operations on a binary search tree are linear in the height of the tree, i.e. $O(h)$. Recall that the *height* of a tree is the length of the longest path from the root to any of the leaves. For a binary tree with n nodes the height can be anywhere between $\lg n$ (balanced) and n (unbalanced). Unfortunately, insertion and deletion operations can change the height of the tree arbitrarily. So, the key question is:

Question : *Is it possible to allow any sequence of insertions and/or deletions and guarantee a balanced tree of height $O(\lg n)$ at all times?*

The answer is *yes*. As we will see in the next section, *Red-Black Trees* solve this problem.

3.1 Single and Double Rotations

A *rotation* in a binary search tree is a local restructuring operation that modifies the structure of the tree locally without destroying the search tree property. Rotations are useful in the sense that whenever many nodes are accumulated on a single path (causing unbalancing of the tree), we can employ rotations to distribute nodes to the neighboring branches, thus restoring balance, without violating the search tree property.

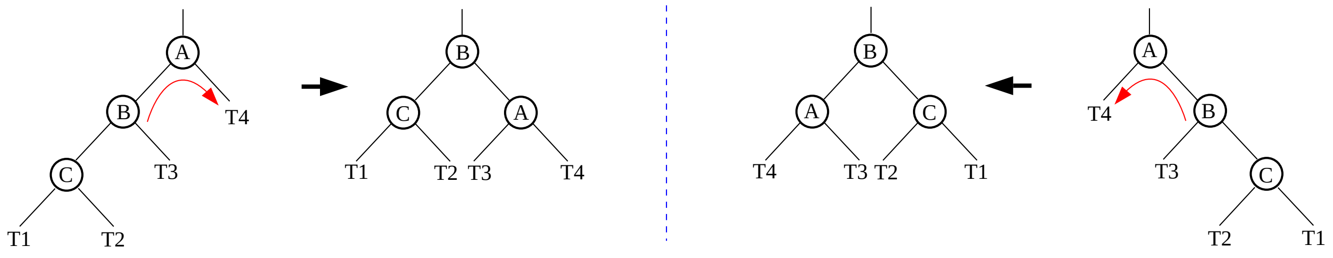

Figure 5: Single rotation (symmetric cases).

Figure 5 shows the *single* rotation. A single rotation basically moves one node from one (sub)path ($A - B - C - ...$) to its companion (sub)path ($A - T4$). From Figure 5 it is easy to see that the binary search tree property is preserved. Notice also that the paths to T1 and T2 have one node less after rotation, whereas the path(s) to T4 has one more. Single rotation applies when both B and C are left (or right) children of their parents.

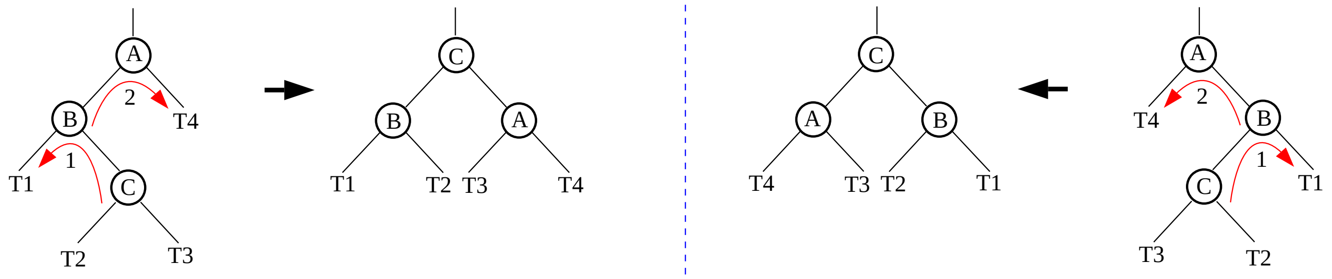

Figure 6: Double rotation (symmetric cases).

When one of B and C is a left child and the other is a right child, a *double* rotation (shown in Figure 6) is required in order to preserve the binary search tree property. A double rotation is realized as two single rotations, performed in the order shown in the figure. The net effect is the same: paths to T2 and T3 are discounted by one node which is added to the path(s) to T4.

In order to perform a single or a double rotation, only local information is required (the pointers to A, B, C and to A's parent), and takes only constant time.

4 Red-Black Trees

A *Red-Black tree* is a binary search tree, as defined in the previous section, with the additional requirement that each node is colored either Black or Red. That means that we need one more bit of information per node (e.g. 0=Red, 1=Black). This does not increase the total space taken by the structure, which is again $\Theta(n)$ for n elements.

We will show that using this extra bit of information, we can keep the tree balanced during any insertion/deletion operations (the other operations do not affect the tree). The height of a balanced tree is $O(\lg n)$, and so, all operations are guaranteed to take at most $O(\lg n)$ time.

The following invariant concerning the node coloring must be satisfied at all times.

Red-Black Tree Color Invariant

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

Here, 'leaf' means a node that has at least one NIL pointer as 'child'. NIL pointers are considered to be Black. So, leaves can be colored Red. All root-leaf paths must have the same number of Black nodes, but we can have a Red node between every two Black nodes. This implies that if there k black nodes on each root-leaf path, all root-leaf paths have length between k and $2k$. So, the longest path in the tree is at most twice as long as the shortest. With this observation, we can easily show that a red-black tree is indeed balanced.

Claim. *A red-black tree with n nodes has height $\Theta(\lg n)$.*

Proof. It is $h_{\max} \leq 2h_{\min}$, where $h_{\max}$ and $h_{\min}$ are the lengths of the longest and the shortest path in the tree, respectively. Then, using also the fact that a complete binary tree with height h has $2^{h+1} - 1$ nodes altogether, we have

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

since there are n nodes in the tree and n can be no less than the nodes of a complete binary tree with height $h_{\min}$ and no more than the nodes of a complete binary tree with height $h_{\max}$. From the inequalities above, it is

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

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

It should be obvious by now that all operations on a Red-Black tree take $O(\lg n)$ time. It remains to show that during an insertion (or a deletion) the color invariant can be maintained without increasing the cost of the operation. We consider insertion first and then the slightly more complicated case of deletion. The symbols and conventions used in the presentation are shown in Figure 7.

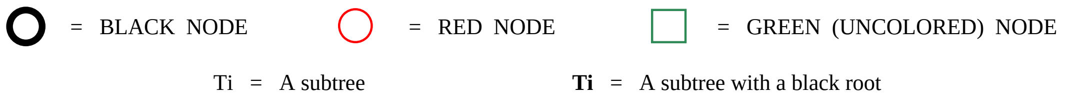

Figure 7: Symbols and conventions used in the figures.

In case this document is viewed/printed in color, nodes are colored with the appropriate color. Symmetric cases are paired in the same figure.

4.1 Insertion in Red-Black trees

Insertion in RB trees is initially the same as insertion in a binary search tree. Recall that the new element is inserted in a leaf that is created at the appropriate place in the tree. The question is: "what color should we give to that new node?" If it is colored RED and it has a RED parent, the RB tree color invariant is violated. If it is colored BLACK then the path to this node will have one extra BLACK node, which again violates the invariant. So, the new node is a problematic node with respect to its color.

It turns out that we can always resolve the problem locally with some local rearrangements and recolorings, or push the problem up in the tree (two levels at a time). So, the problem will be resolved somewhere within the tree, or in the worst case it will 'pop' all the way to the root, where it can be trivially handled. Since the height of the tree is bounded and, in fact, it is logarithmic to the number of elements, this recoloring and rebalancing process will take no more than $O(\lg n)$ steps. Therefore, the whole insertion procedure takes logarithmic time, $O(\lg n)$.

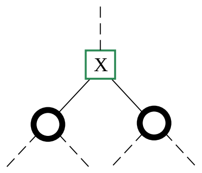

Figure 8: The problematic node (green square) and the context it appears.

To illustrate the recoloring and rebalancing process, let's color the problematic node GREEN which indicates that this node is yet to be colored. We claim that the problematic node wherever appears has only black (or no) children, as shown in Figure 8. Recall that NIL pointers are treated as BLACK, so this is indeed the case right after the insertion of the new node. We will see that whenever the problematic node is pushed up in the tree the same situation appears. So, it is sufficient to resolve this situation only.

We assign a color to the GREEN node depending on its parent color (if the parent exists). We distinguish among three cases:

1. ***Parent is*** BLACK

   This is the simplest case and it is illustrated in Figure 9 (the right half is the symmetric of the left half). In this case we can safely color the problematic node RED. Recall that the subtrees **T1** and **T2** have BLACK roots. It is easy to see that the invariant is not violated. X cannot be the root, so RED is a valid color. The RED-colored node X has only BLACK children. Also, no BLACK nodes have been added, so all paths have the same number of BLACK nodes. Notice that the problem is fully resolved.

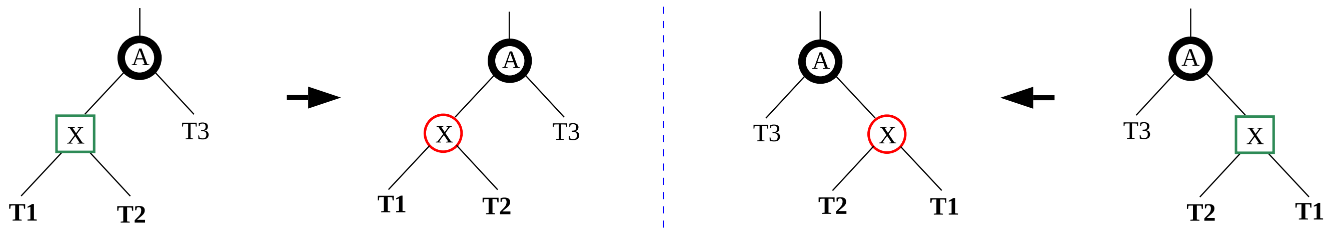

Figure 9: The parent of the problematic node is BLACK.

2. ***Parent is*** RED

   This case is little more involved, because we need to know the color of C, X's uncle, or B's sibling

(B is X's parent). B cannot be the root of the tree, because it is RED, so C exists. If not recall that NIL pointers are taken as BLACKs. Again, there are two subcases.

(a) ***Uncle is*** RED

This situation is illustrated in Figure 10 for the case where X is 'away' from the uncle and in Figure 11 for the case where X is 'close' to the uncle.

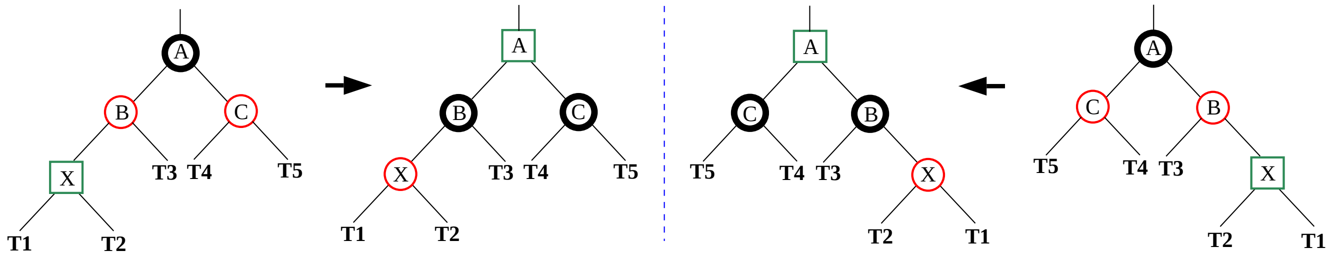

Figure 10: The parent and the uncle are both RED; X is 'away' from the uncle.

In this case, we color X RED. However, before doing so, we have to make sure that its parent has become BLACK. We can do so, by 'pushing' the BLACK node from A one level down, splitting it in two. Splitting is required to ensure that both paths get the same number of BLACK nodes. After all this, we are left with the problem of coloring A, but nevertheless the problem has moved two levels up in tree and we have the same problematic situation (uncolored node with BLACK children). The same procedure can be applied recursively to A, so we do not continue further.

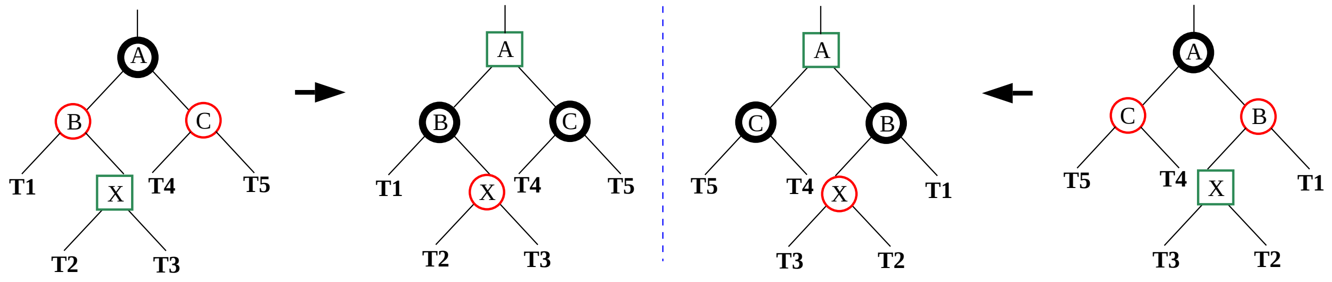

Figure 11: The parent and the uncle are both RED; X is 'close' to the uncle.

Let's examine if the invariant is maintained. X cannot be the root, and it has black children, so RED is a valid color for X and the first two parts are satisfied. Also, the number of BLACK nodes on each path ($A - ... - Ti$) remained unchanged (=1). So, the invariant is maintained and only a constant number of local operations is required.

(b) ***Uncle is*** BLACK

This situation is illustrated in Figure 12 for the case where X is 'away' from the uncle and in Figure 13 for the case where X is 'close' to the uncle.

This case implies that too many nodes have been accumulated on the path from the root to X, so the problem is resolved by 'donating' one node to the other path ($... - A - C - ...$). This is feasible since A and C are both BLACK and so there is 'space' for a RED node between them. However, this change has to be done gently to ensure that the binary search tree property is not destroyed. A single or a double rotation on ($A - B - X$) is exactly what is needed. The restructuring (rotations) and recoloring required in each case is shown in Figures 12 and 13. The binary search tree property holds after the transformation, since the transformation involves only rotations. Notice also that the problem is fully resolved in this case.

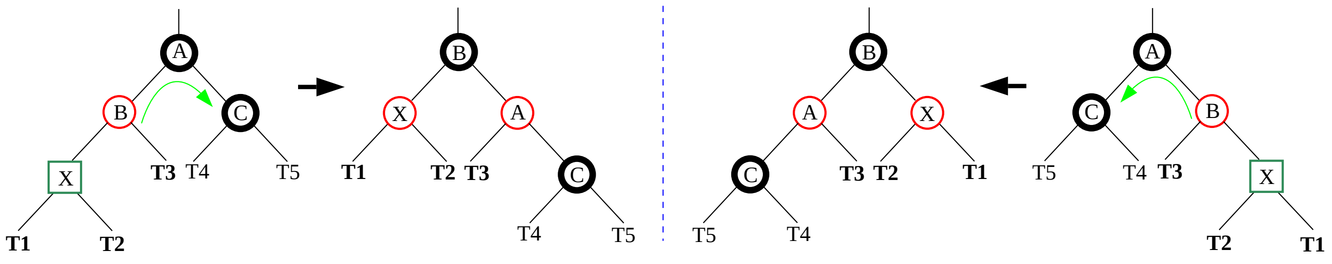

Figure 12: The parent is RED and the uncle is BLACK; X is 'away' from the uncle.

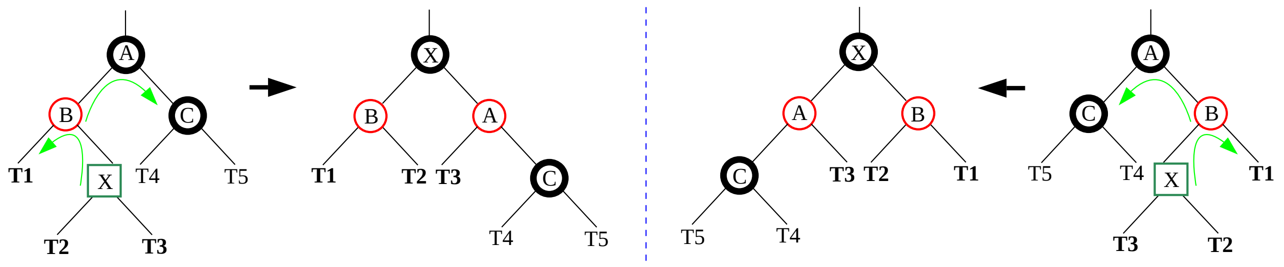

Figure 13: The parent is RED and the uncle is BLACK; X is 'close' to the uncle.

Let's examine if the invariant is maintained after recoloring. Firstly, the color of the root is unchanged (even if A was the root). The (afterwards) RED nodes A and B have only BLACK children, because C is black and the subtrees **T1**, **T2** and **T3** have black roots. Finally, the number of BLACK nodes on paths to subtrees **T1**, **T2** and **T3** is the same before and after the transformation (all take 1 BLACK node from this portion of the tree). The same for the paths to T4 and T5 (2 BLACK nodes in this portion of the tree).

3. ***Parent does not exist***

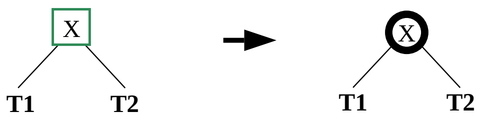

Figure 14: The problematic node is the root.

In this case, the problematic node is simply the root, either because X is the first element inserted in the tree or because the problem was pushed all the way up the tree. This is trivially resolved by coloring the node BLACK, as illustrated in Figure 14. This way the root becomes BLACK and *all* paths from the root get one more BLACK, i.e. they still have the same number of BLACK nodes. RED nodes are not affected. Thus, the invariant is maintained and the problem is fully resolved.

4.2 Deletion in RB trees

Deletion in RB trees initially proceeds exactly as the deletion in a binary search tree. The switching of the element to be deleted with its successor (or predecessor), if happened, does not affect the colors and the invariant. So, everything boils down to a deletion of a node X with just one or no children, which is the last step of the deletion in a binary search tree. If X is RED, then we just remove it. The invariant is not affected, since X cannot be the root, and its removal does not change the number of BLACK nodes in any path. If X is BLACK and its single child is RED, then we can safely remove X and color the child

BLACK. Again the invariant is maintained. This is true, even if X is the root, since the new root (X's child) will be BLACK. These two trivial cases are illustrated in Figure 15.

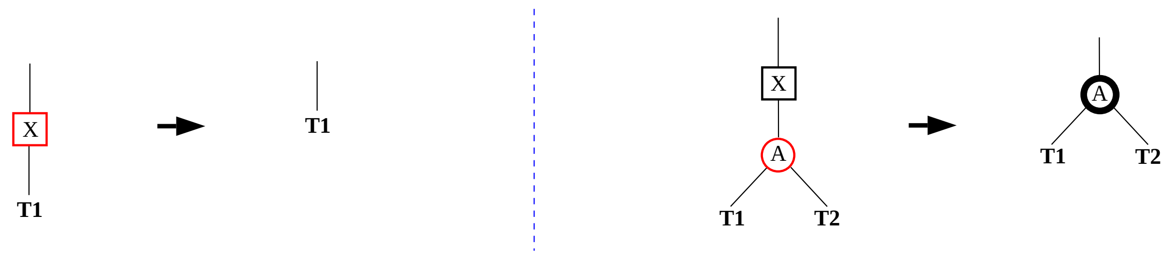

Figure 15: Deletion: two trivial cases.

So, the only case that is really the problem is when the node to be deleted is BLACK and it has a BLACK child or no children at all (remember that NIL pointers are treated as BLACK). This problematic situation is shown in Figure 16. We depict the problematic node with a black square.

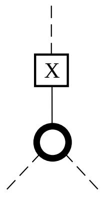

Figure 16: The problematic node (black square) and the context it appears.

Again, it turns out that we can always resolve the problem either locally by some local rearrangements and recolorings, or by pushing the problematic BLACK node up in the tree (two levels at a time). So, the problem will be either resolved somewhere within the tree, or in the worst case it will 'pop' all the way to the root, where it can be trivially handled. Since the height of the tree is bounded and, in fact, it is logarithmic to the number of elements, the recoloring and rebalancing process will take no more than $O(\lg n)$ steps. Therefore, the whole deletion procedure takes logarithmic time, $O(\lg n)$.

To resolve the problem, we need to consider several cases, depending on the colors of the parent of X, the sibling of X and the nephews of X (its siblings' children).

1. ***Parent is*** RED (and ***Sibling is*** BLACK)

   If the parent of X is RED, then X must have a sibling, otherwise the invariant would be violated (the path through X would have at least one more BLACK node, that is X, than the path that would terminate at the NIL pointer of X's parent). Moreover, the sibling has to be BLACK, given that it is a son of a RED node.

   There are two separate cases, depending on the color of the 'close nephew' of X, that is the child of X's sibling that lies between X and X's sibling in the total order of all elements in the tree. Graphically, the close nephew lies on the side closer to X in the tree. Note that the case where the close nephew does not exist (i.e. it is a NIL pointer) is equivalent to that of the close nephew being BLACK.

   (a) ***Close Nephew is*** BLACK

      This case and its resolution is illustrated in Figure 17. The idea is to pull the BLACK sibling B to the top to compensate for the loss of the BLACK X on the path $(A - X - T1)$. This is done by a single rotation.

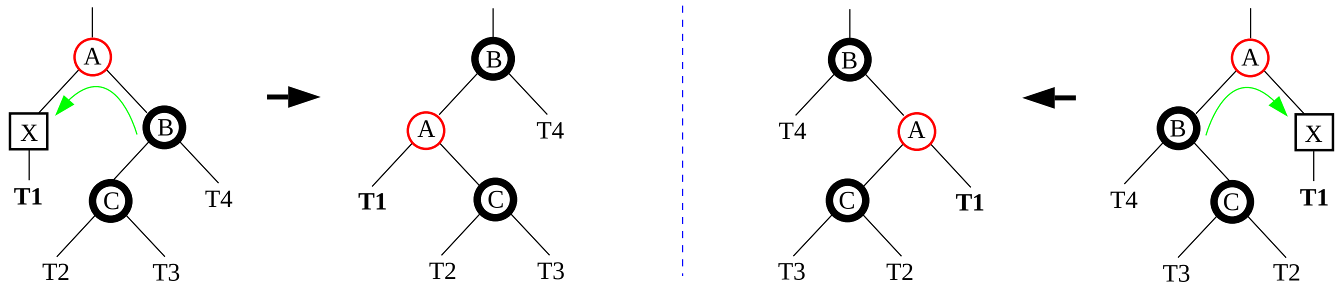

Figure 17: The parent is RED, sibling is BLACK, and the close nephew is BLACK.

The invariant is indeed maintained. Firstly, the root is not affected. A, which is colored RED, receives two BLACK children, C and the root of **T1** which is known to be BLACK. Finally, the paths to **T1** and T4 'receive' 1 BLACK node, and the paths to T2 and T3 'receive' 2 BLACK nodes from this portion of the tree, as before. So, the resolution is valid and complete; the problematic node has been eliminated and the invariant is satisfied.

(b) ***Close Nephew is*** RED

This case and its resolution is illustrated in Figure 18. The problem is fully restored by a double rotation and some recoloring.

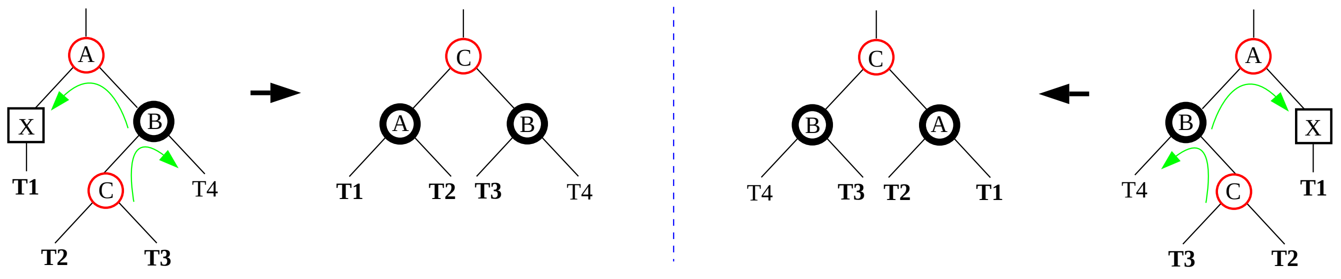

Figure 18: The parent is RED, sibling is BLACK, and the close nephew is RED.

The invariant is again maintained. C cannot be the root, since A is not the root, otherwise it wouldn't be RED. So, the color of the root is not affected. C which is colored RED receives two BLACK children, A and B. One BLACK node is contributed to each subtree from this portion of the tree, as before, so the number of BLACKs on each path is unchanged.

2. ***Parent is*** BLACK

X must have a sibling, as in the previous case, otherwise the invariant would be violated (the path through X would have at least one more BLACK node, that is X, than the path that would terminate at the NIL pointer of X's parent). However, we cannot determine the color of X's sibling given this situation. So, we distinguish among two cases:

(a) ***Sibling is*** RED

This situation is illustrated in Figure 19. The problem is attacked by a single rotation that, interestingly, pushes the problem down! However, notice that the resulting situation is the one where the parent is RED and the sibling is BLACK, which we just described and can be fully resolved locally. So, it doesn't really hurt to push the problem down once, because we are going to get rid of it afterwards.

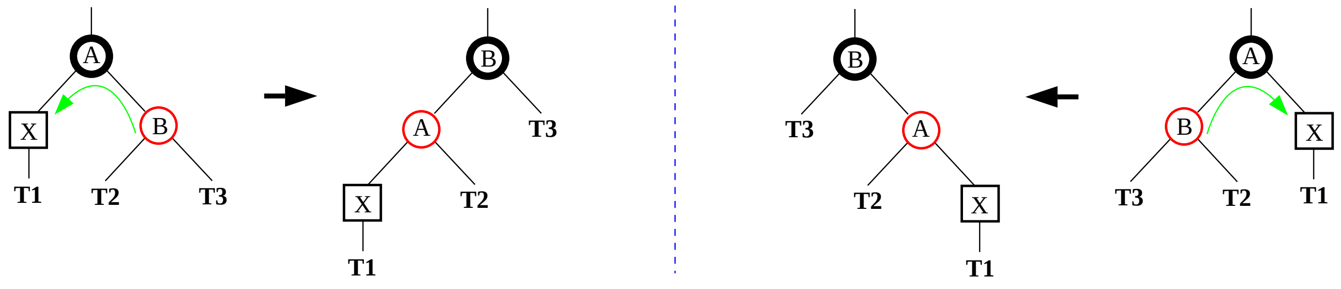

Figure 19: The parent is BLACK and the sibling is RED.

For the invariant, the root remains BLACK, even if A was the root. X and the root of **T2** are known to be BLACK, so A is well colored. It's easy to see, also, that the number of BLACKs on each path is preserved. So, the invariant is maintained.

(b) ***Sibling is*** BLACK

There are several cases, depending on the coloring of the nephews of X. Notice that the nephews do not necessarily exist. However, the case where some nephew does not exist is equivalent to that where this nephew is colored BLACK. So, nonexistent nephews are covered by the cases below. We use the term Close Nephew for the child of X's sibling that lies between X and X's sibling in the total order of all elements in the tree. Far Nephew is the one that is not Close. Graphically, the close nephew lies on the side closer to X in the tree, and the far nephew lies on the side away from X.

i. ***Far Nephew is*** RED

In this case, the problem is resolved with a single rotation (Figure 20). The idea is to take a BLACK node (A) from the companion branch to compensate for the loss of X. The loss of A is compensated for by coloring C BLACK. The problem is completely eliminated.

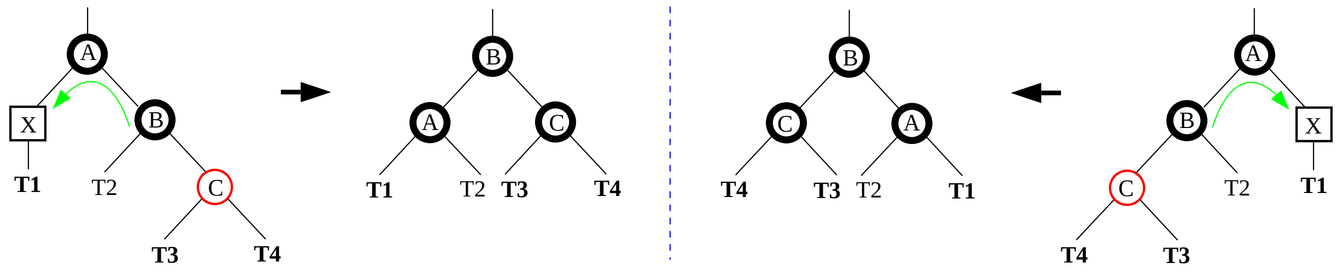

Figure 20: The parent is RED, the sibling is BLACK, and the far nephew is RED.

The invariant is again maintained. The root remains BLACK even if it was A. We do not color any node RED in tnis case. The contribution of BLACK nodes in the paths is the same as before.

ii. ***Far Nephew is*** BLACK

This case requires checking the color of the other nephew. Depending on its color there are two cases:

A. ***Close Nephew is*** RED

The fact that the close nephew is RED allows us to compensate, via BLACK-coloring, for the loss of the BLACK node moved on X's path to compensate for X's loss. This is achieved properly by a double rotation, as shown in Figure 21. The problem is completely eliminated, once more.

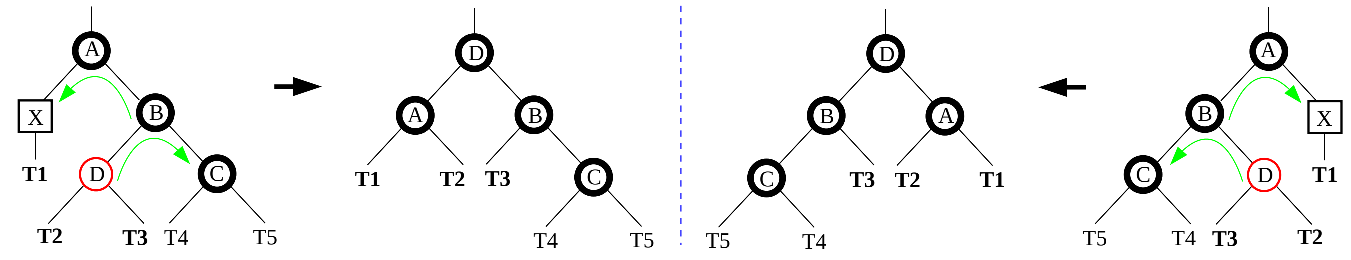

Figure 21: The parent is RED, the sibling is BLACK, the far nephew is RED, and the close nephew is RED.

This is similar to the previous case and the invariant is maintained for the same reasons.

B. ***Close Nephew is*** BLACK

This last case is tricky, because the existence of only BLACK nodes in the neighborhood, does not provide for removal of X and local recoloring and/or restructuring. The only thing we can do in case, is to push the problem up in the tree, as shown in Figure 22. Notice that the new problematic situation is the same as the one we started with, i.e. X is BLACK with a BLACK child. The same procedure can be applied recursively. The RED node (B) is (safely) introduced to compensate for the placement of the BLACK X on the common path.

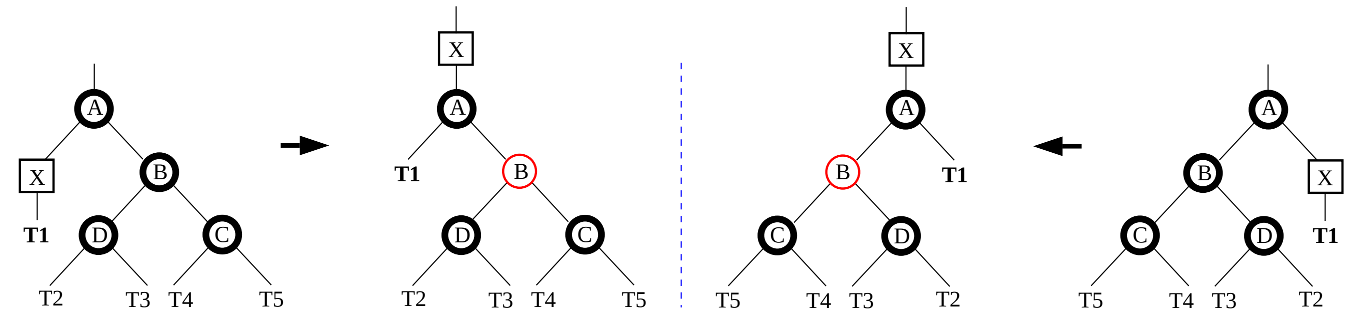

Figure 22: The parent is RED, the sibling is BLACK, the far nephew is RED, and the close nephew is BLACK.

The root is not affected unless A is the root, in which case X becomes the new root (a case handled below) and it is BLACK as required. The RED coloring of B is safe, because C and D are known to be BLACK (or NIL). The number of BLACK nodes on each path is preserved, as it can be easily seen from the figure.

3. ***Parent does not exist***

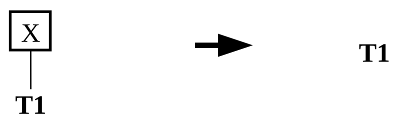

Figure 23: The problematic node is the root.

In this case, the problematic node is simply the root, either because it is the root itself that we want to delete or because X was pushed all the way up the tree. This is trivially resolved by removing X and setting the root to **T1**, whose root is known to be BLACK. This is illustrated in Figure 23.

Doing so, *all* paths from the root loose one Black node, but they still have the same number of Black nodes. Red nodes are not affected. Thus, the invariant is maintained and the problem is fully resolved.

4.3 Remarks on Red-Black Trees

- The root is the *only* place in the tree from where the number of Black nodes along each path changes (increases or decreases).
-
-
- more ...

Note : The trees we presented here, store elements in the nodes. These are the so-called *node-oriented* trees. Another approach is to store the elements only in the leaves of the tree. Then the internal nodes hold 'rating' elements that guide us through the tree (usually it is —recursively— the minimum of the elements in the two children). Note that the binary search tree property is again satisfied, but now each internal node is required to have exactly 2 children. Also, although there are more nodes in this variation, the overall size of the tree is again $\Theta(n)$ for storing n elements. These are the so-called *leaf-oriented* trees. In some cases, they are preferable. We concetrated on node-oriented trees, but the algorithms apply to leaf-oriented trees as well.