Topic 8: Search Trees and Red-Black Trees

(CLRS 10, 12.0–12.3, 13 (read this note instead))

CPS 230, Fall 2001

1 Introduction

- Up until now we've looked primarily at *batched problems*, in which no preprocessing is done and the entire input file must be processed. Sorting is a classic example of a batched problem. Others we've seen include selection, finding the median, and matrix multiplication.
- In this lecture, we start to discuss data structures for *online problems*, in which computation is done in response to a continuous series of query operations. In particular, we'll start with data structures for maintaining ordered sets (or dictionaries).
- A common technique for online problems is to organize the data items via a hierarchical index, so that only a very small portion of the data needs to be examined in response to each query.
- The data being queried can be either *static*, which can be preprocessed for efficient query processing, or *dynamic*, where the queries are intermixed with updates such as insertions and deletions.
- Dynamic data structures play an important role in algorithms design.

2 Maintaining an ordered set dynamically

- We want to maintain an ordered set S under the following dynamic operations:
  - SEARCH(e): Return (pointer to) element e in S (if $e \in S$)
  - INSERT(e): Insert element e in S
  - DELETE(e): Delete element e from S
  - SUCCESSOR(e): Return (pointer to) minimal element in S larger than e
  - PREDECESSOR(e): Return (pointer to) maximal element in S smaller than e

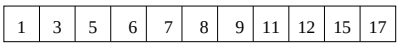
2.1 Models of computation

- Just as we saw with sorting, we can get different results depending upon whether we use a pointer machine model, in which case we operate using comparisons and linked structures, or if we use a more general RAM model that permits arithmetic operations and indirect addressing. The power of the RAM model allowed us to get linear-time sorting algorithms in certain situations.
- To get realistic models, we add some restrictions to the RAM model. One restriction is to allow computations on arbitrary-sized integers, but to charge $O(\log \ell)$ as the cost of an operation involving integers of ℓ bits.

- A more common model is to instead use *words* of $\log n$ bits, and to allow constant-time arithmetic operations on words. The motivation is that pointers must be $\Theta(\log n)$ bits, and it's reasonable to allow pointer manipulation in constant time.

2.2 Ordered array implementation

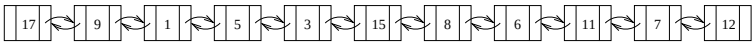
- The first implementation that comes to mind is the ordered (i.e., sorted) array:



  - SEARCH can be performed in $O(n)$ time by scanning through array or in $O(\log n)$ time using binary search
  - PREDECESSOR/SUCCESSOR can be performed in $O(\log n)$ time like searching
  - INSERT/DELETE takes $O(n)$ time since we need to expand/compress the array after finding the position of e

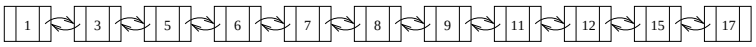
2.3 Double linked list implementation

- Unordered list



  - SEARCH takes $O(n)$ time since we have to scan the list
  - PREDECESSOR/SUCCESSOR takes $O(n)$ time
  - INSERT takes $O(1)$ time since we can just insert e at beginning of list
  - DELETE takes $O(n)$ time since we have to perform a search before spending $O(1)$ time on deletion

- Ordered list



  - SEARCH takes $O(n)$ time since we cannot perform binary search
  - PREDECESSOR/SUCCESSOR takes $O(n)$ time
  - INSERT/DELETE takes $O(n)$ time since we have to perform a search to locate the position of insertion/deletion

3 Binary Search Trees

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. If a subtree is empty, we say that it consists of an *external node* or *leaf*.

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 $O(1)$

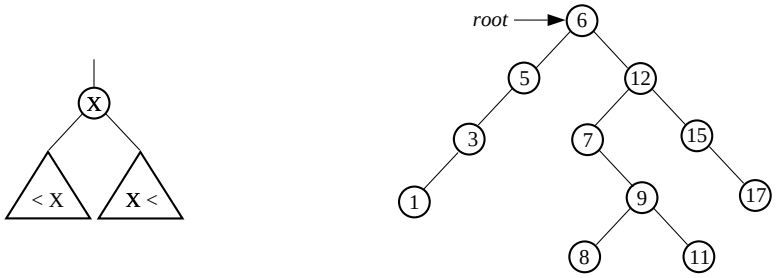


Figure 1: The search tree property (left) and an example of a binary search tree (right). The leaves (i.e., external nodes) are not pictured. A node with k children (for $0 \leq k \leq 2$) has $2 - k$ leaf nodes immediately below it. For example, since node 1 has no children, we say that there are two leaves immediately below node 1. Since node 3 has one child (namely, node 1), we say that there is one leaf immediately below node 3. Leaves can be reprsented by **nil** pointers.

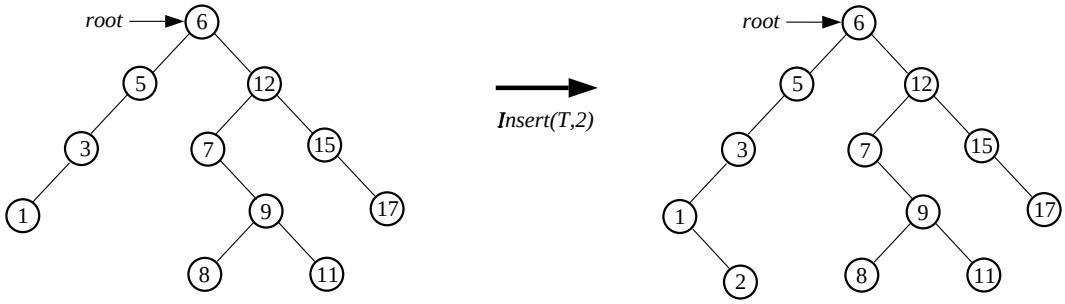


Figure 2: Inserting a node.

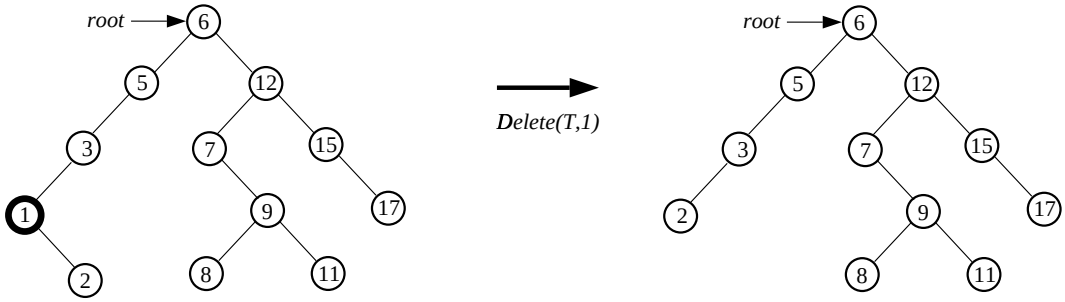


Figure 3: Deleting a node with one child.

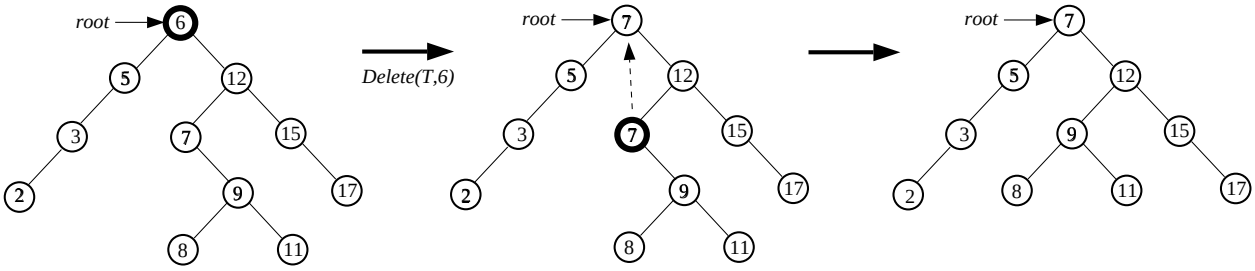


Figure 4: Deleting a node with two children.

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 leaf 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. (The node w containing 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 keep the tree (relatively) balanced during updates, so that all operations take $O(\log n)$ time.

4 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 an external node, as defined previously. That is, we can imagine any node with no children as having two leaf nodes below it, and any node with exactly one child has one leaf node below it, as explained in Figure 1.

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 root-leaf path in the tree, respectively. Then, using the fact that a complete binary tree with height h (i.e., root-to-leaf path length h) has $2^h - 1$ nodes, we have

$$\begin{aligned} 2^{h_{\min}} - 1 \leq n \leq 2^{h_{\max}} - 1 \quad &\Longrightarrow \quad h_{\min} \leq \log(n+1) \leq h_{\max} \leq 2h_{\min} \\ &\Longrightarrow \quad \frac{1}{2}\log(n+1) \leq h_{\min} \leq \log(n+1) \end{aligned}$$

Since $h_{\min} \leq h_{\max} \leq 2h_{\min}$, it is also true that $h_{\max} \leq 2\log(n+1)$. ■

4.1 Insertion into Red-Black Trees

An insertion into a red-black tree is initially the same as an insertion into a binary search tree; the new element is inserted in a leaf at the appropriate place in the tree (thus creating two new leaf nodes immediately below it). 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 paths from the root to the leaves below it 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):

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 (since B is RED), and therefore X has a grandparent (call it A).

   (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 into 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 nonexisting***

   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

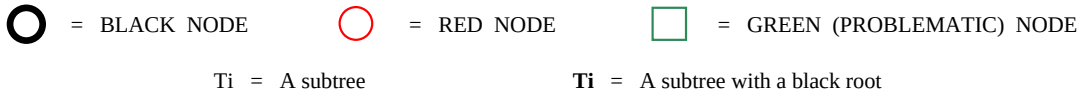


Figure 5: Symbols used in the figures.

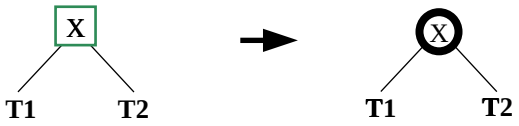


Figure 6: The problematic node is the root (case 1).

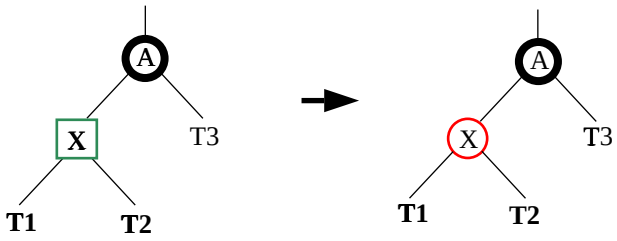


Figure 7: The parent of the problematic node is BLACK (case 2).

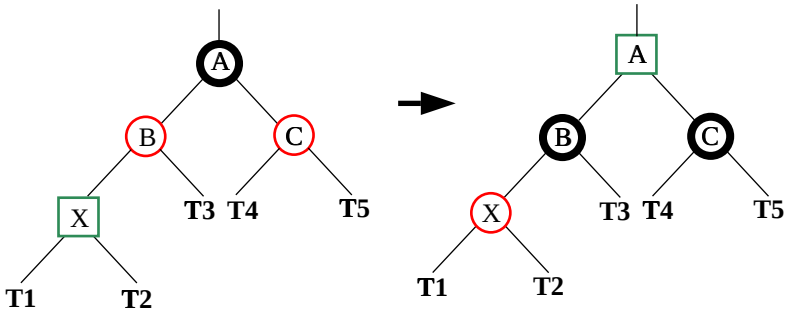


Figure 8: The parent and the uncle are both RED (case 3 a)).

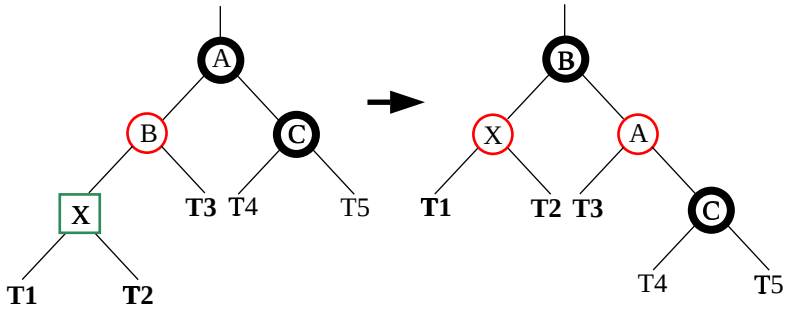


Figure 9: The parent is RED and the uncle is BLACK (case 3 b) 1).

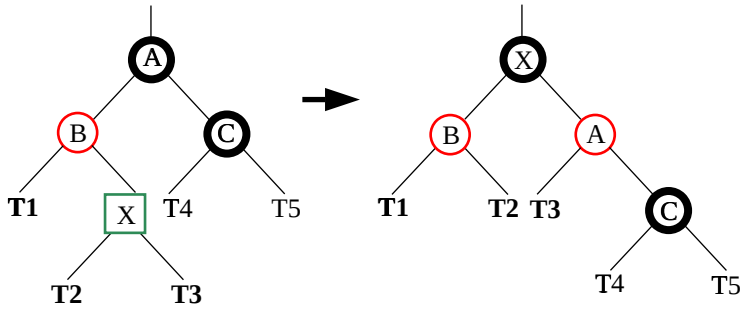


Figure 10: The parent is RED and the uncle is BLACK (case 3 b) 2).

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).

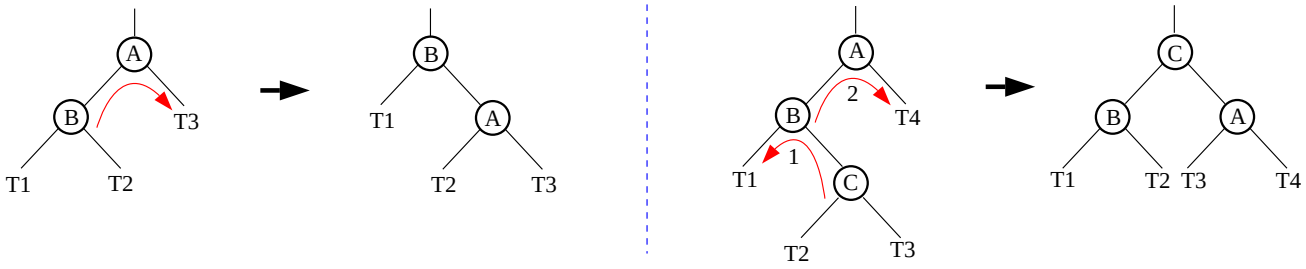


Figure 11: a) Single rotation. b) Double rotation.

4.2 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, as described in Section 3. When the only remaining step is the deletion of a node X with no left child, we need to make sure that the red-black tree coloring invariant is maintained. If X is RED, it must have no children at all. (If X had a right child, the right child couldn't also be RED and therefore would have to be BLACK, but then the root-to-leaf path to the leaf to X's left would have one less black node than the path that goes through X's right child, which violates the invariants.) Therefore, we can simply remove X. 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 no children. There's no easy fix. We'll handle the situation by a case-by-case analysis, as for insertion. We'll see below 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).

As previously, we depict the problematic node X as a square, but unlike in the insertion case, X's color is already determined (namely, X is BLACK). We also maintain the *deletion invariant* that the problematic BLACK node X has a BLACK son (if any at all). (Initially, X has no children, but during the course of pushing X up the tree, it may have one child, but the child will always be BLACK.)

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, X's parent is immediately above a leaf L (external node), and a root-leaf path through X would have at least one more BLACK node than the root-leaf path to L.) Moreover, the sibling has to be BLACK, since the parent is RED. There are two cases, depending on the color of the nephew of X.

(a) ***Nephew is*** BLACK ***(or nonexisting)***
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 root-leaf paths stays the same.

(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 root-leaf paths stays the same.

3. ***Parent is*** BLACK

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

(a) ***Sibling is*** RED
Since the sibling B is RED, it must have two children, or else there is a root-leaf path with fewer BLACK nodes than a root-leaf path through X. Since B is RED, it also follows that B's children are BLACK.
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 and nephew are each BLACK. This situation can be fully resolved locally as described in case 2(a) above. As before, 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 ***(or nonexisting)***
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 results at most three node rotations. The number of recolorings, however, can be up to $\Theta(\log n)$.

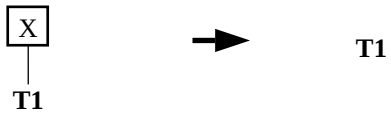


Figure 12: The problematic node is the root.

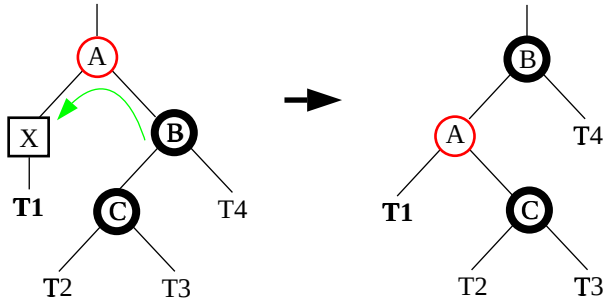


Figure 13: The parent is RED (sibling is BLACK) and nephew is BLACK.

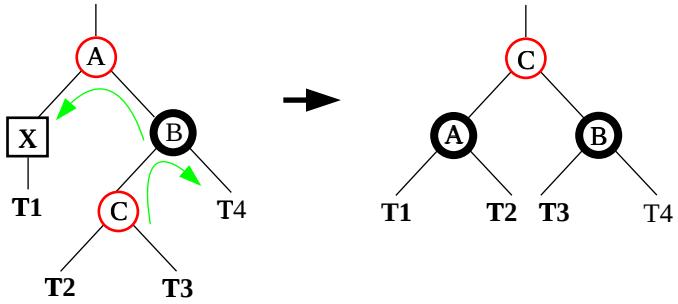


Figure 14: The parent is RED (sibling is BLACK) and nephew is RED.

5 Correspondence between Red-Black Trees and 2-3-4 Trees

There is a natural correspondence between red-black trees and 2-3-4 trees, which are multiway balanced trees in which the nodes having no children are all on the same level, and each internal node has between two and four children.

Here's how we can convert a 2-3-4 tree into a red-black tree: For each node in a 2-3-4 tree, starting from the root, make it a a BLACK node in a red-black tree. If the 2-3-4 node has two children, its two children are BLACK. If it has three children, we replace it by a node with two children: one BLACK child and the other child is a RED node that has two BLACK children of its own. The three new BLACK nodes represent the original three children. Similarly, if the 2-3-4 node has four children, we replace it by a BLACK node with two RED children that each have two BLACK children. The four new BLACK nodes represent the original four children.

On any path in the 2-3-4 tree starting at the root, each level adds one BLACK node to the corresponding red-black tree. Since childless nodes in the 2-3-4 tree are all on the same level, the number of BLACK nodes on any root-to-leaf path in the red-black tree must be the same. By construction, each RED node has BLACK children, so the invariants of red-black trees are satisfied.

The correspondence goes both ways, so that any red-black tree can be transformed into a corresponding 2-3-4 tree. 2-3-4 trees operate by splitting and merging nodes along a root-to-leaf path when the degrees get too large or small. The balancing operations that we studied in this lecture note are the corresponding operations on the corresponding red-black tree.

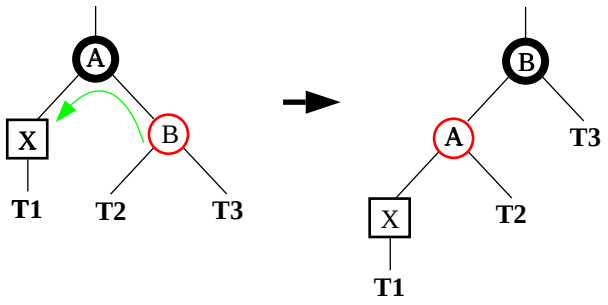


Figure 15: The parent is BLACK and the sibling is RED.

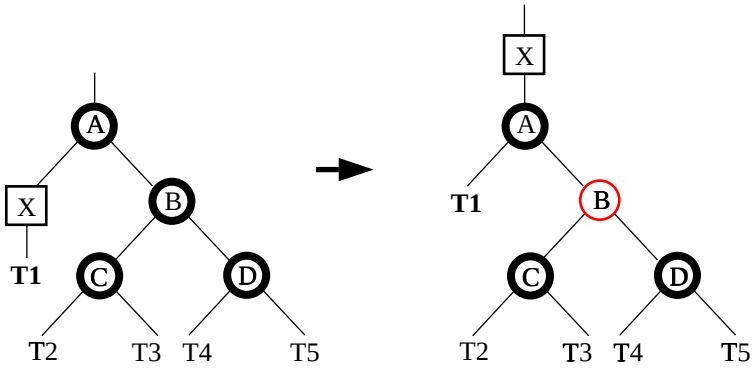


Figure 16: The parent is BLACK, the sibling is BLACK, and both nephews are BLACK or nonexisting. Note that the actual key value represented by X may need to be modified to maintain the search tree property, but since X is going to be deleted anyway, it doesn't matter.

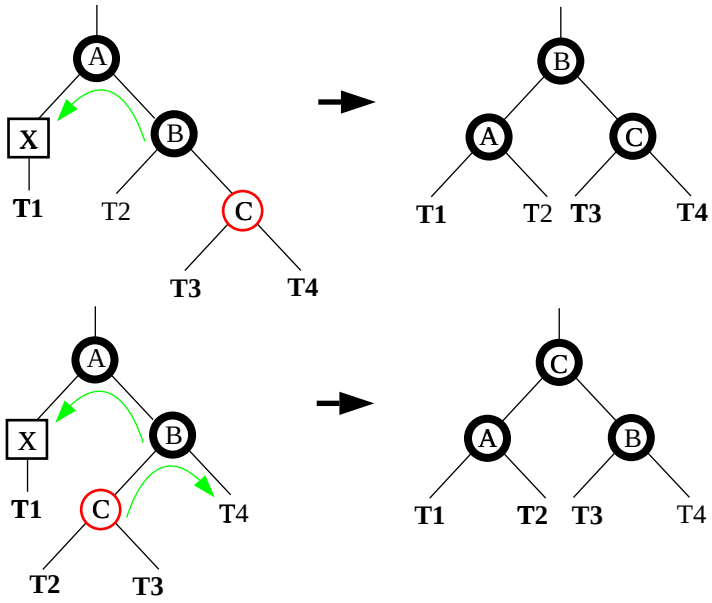


Figure 17: The parent is BLACK, the sibling is BLACK, and a nephew is RED.