In computer science, a **red–black tree** is a self-balancing binary search tree data structure noted for fast storage and retrieval of ordered information. The nodes in a red-black tree hold an extra "color" bit, often drawn as red and black, which help ensure that the tree is always approximately balanced.[1]

When the tree is modified, the new tree is rearranged and "repainted" to restore the coloring properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that this rearranging and recoloring can be performed efficiently.

The (re-)balancing is not perfect, but guarantees searching in $O(\log n)$ time, where n is the number of entries in the tree. The insert and delete operations, along with tree rearrangement and recoloring, also execute in $O(\log n)$ time.[2][3]

Tracking the color of each node requires only one bit of information per node because there are only two colors (due to memory alignment present in some programming languages, the real memory consumption may differ). The tree does not contain any other data specific to it being a red–black tree, so its memory footprint is almost identical to that of a classic (uncolored) binary search tree. In some cases, the added bit of information can be stored at no added memory cost.

History

In 1972, Rudolf Bayer[4] invented a data structure that was a special order-4 case of a B-tree. These trees maintained all paths from root to leaf with the same number of nodes, creating perfectly balanced trees. However, they were not *binary* search trees. Bayer called them a "symmetric binary B-tree" in his paper and later they became popular as 2–3–4 trees or even 2–3 trees.[5]

In a 1978 paper, "A Dichromatic Framework for Balanced Trees",[6] Leonidas J. Guibas and Robert Sedgewick derived the red–black tree from the symmetric binary B-tree.[7] The color "red" was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.[8] Another response from Guibas states that it was because of the red and black pens available to them to draw the trees.[9]

In 1993, Arne Andersson introduced the idea of a right leaning tree to simplify insert and delete operations.[10]

In 1999, Chris Okasaki showed how to make the insert operation purely functional. Its balance function needed to take care of only 4 unbalanced cases and one default balanced case.[11]

The original algorithm used 8 unbalanced cases, but Cormen et al. (2001) reduced that to 6 unbalanced cases.[1] Sedgewick showed that the insert operation can be implemented in just 46 lines of Java.[12][13] In 2008, Sedgewick proposed the left-leaning red–black tree, leveraging Andersson's idea that simplified the insert and delete operations. Sedgewick originally allowed nodes whose two children are red, making his trees more like 2–3–4 trees, but later this restriction was added, making new trees more like 2–3 trees. Sedgewick implemented the insert algorithm in just 33 lines, significantly shortening his original 46 lines of code.[14][15]

Terminology

The **black depth** of a node is defined as the number of black nodes from the root to that node (i.e. the number of black ancestors). The **black height** of a red–black tree is the number of black nodes in any path from the root to the leaves, which, by requirement 4, is constant (alternatively, it could be defined as the black depth of any leaf node).[16]:154–165 The black height of a node is the black height of the subtree rooted by it. In this article, the black height of a null node shall be set to 0, because its subtree is empty as suggested by the example figure, and its tree height is also 0.

Properties

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:[17]

1. Every node is either red or black.
2. All null nodes are considered black.
3. A red node does not have a red child.
4. Every path from a given node to any of its leaf nodes (that is, to any descendant null node) goes through the same number of black nodes.
5. (Conclusion) If a node **N** has exactly one child, the child must be red. If the child were black, its leaves would sit at a different black depth than **N**'s null node (which is considered black by rule 2), violating requirement 4.

Some authors, e.g. Cormen & al.,[17] claim "the root is black" as fifth requirement; but not Mehlhorn & Sanders[16] or Sedgewick & Wayne.[15]:432–447 Since the root can always be changed from red to black, this rule has little effect on analysis. This article also omits it, because it slightly disturbs the recursive algorithms and proofs.

As an example, every perfect binary tree that consists only of black nodes is a red–black tree.

The read-only operations, such as search or tree traversal, do not affect any of the requirements. In contrast, the modifying operations insert and delete easily maintain requirements 1 and 2, but with respect to the other requirements some extra effort must be made, to avoid introducing a violation of requirement 3, called a **red-violation**, or of requirement 4, called a **black-violation**.

The requirements enforce a critical property of red–black trees: *the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf.* The result is that the tree is height-balanced. Since operations such as inserting, deleting, and finding values require worst-case time proportional to the height h of the tree, this upper bound on the height allows red–black trees to be efficient in the worst case, namely logarithmic in the number n of entries, i.e. $h \in O(\log n)$ (a property which is shared by all self-balancing trees, e.g., AVL tree or B-tree, but not the ordinary binary search trees). For a mathematical proof see section Proof of bounds.

Red–black trees, like all binary search trees, allow quite efficient sequential access (e.g. in-order traversal, that is: in the order Left–Root–Right) of their elements. But they support also asymptotically optimal direct access via a traversal from root to leaf, resulting in $O(\log n)$ search time.

Analogy to 2–3–4 trees

Red–black trees are similar in structure to 2–3–4 trees, which are B-trees of order 4.[18] In 2–3–4 trees, each node can contain between 1 and 3 values and have between 2 and 4 children. These 2–3–4 nodes correspond to black node – red children groups in red-black trees, as shown in figure 1. It is not a 1-to-1 correspondence, because 3-nodes have two equivalent representations: the red child may lie either to the left or right. The left-leaning red-black tree variant makes this relationship exactly 1-to-1, by only allowing the left child representation. Since every 2–3–4 node has a corresponding black node, invariant 4 of red-black trees is equivalent to saying that the leaves of a 2–3–4 tree all lie at the same level.

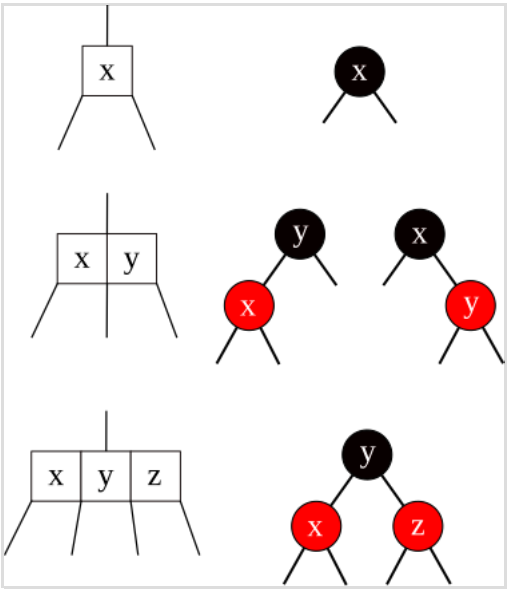


Figure 1: Similarities between 2–3–4 trees and red-black trees

Despite structural similarities, operations on red–black trees are more economical than B-trees. B-trees require management of vectors of variable length, whereas red-black trees are simply binary trees.[19]

Applications and related data structures

Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time. Not only does this make them valuable in time-sensitive applications such as real-time applications, but it makes them valuable building blocks in other data structures that provide worst-case guarantees. For example, many data structures used in computational geometry are based on red–black trees, and the Completely Fair Scheduler and epoll system call of the Linux kernel use red–black trees.[20][21] The AVL tree is another structure supporting $O(\log n)$ search, insertion, and removal. AVL trees can be colored red–black, and thus are a subset of red-black trees. The worst-case height of AVL is 0.720 times the worst-case height of red-black trees, so AVL trees are more rigidly balanced. The performance measurements of Ben Pfaff with realistic test cases in 79 runs find AVL to RB ratios between 0.677 and 1.077, median at 0.947, and geometric mean 0.910.[22] The performance of WAVL trees lie in between AVL trees and red-black trees.

Red–black trees are also particularly valuable in functional programming, where they are one of the most common persistent data structures, used to construct associative arrays and sets that can retain previous versions after mutations. The persistent version of red–black trees requires $O(\log n)$ space for each insertion or deletion, in addition to time.

For every 2–3–4 tree, there are corresponding red–black trees with data elements in the same order. The insertion and deletion operations on 2–3–4 trees are also equivalent to color-flipping and rotations in red–black trees. This makes 2–3–4 trees an important tool for understanding the logic behind red–black trees, and this is why many introductory algorithm texts introduce 2–3–4 trees just before red–black trees, even though 2–3–4 trees are not often used in practice.

In 2008, Sedgewick introduced a simpler version of the red–black tree called the left-leaning red–black

tree[23] by eliminating a previously unspecified degree of freedom in the implementation. The LLRB maintains an additional invariant that all red links must lean left except during inserts and deletes. Red–black trees can be made isometric to either 2–3 trees,[24] or 2–3–4 trees,[23] for any sequence of operations. The 2–3–4 tree isometry was described in 1978 by Sedgewick.[6] With 2–3–4 trees, the isometry is resolved by a "color flip," corresponding to a split, in which the red color of two children nodes leaves the children and moves to the parent node.

The original description of the tango tree, a type of tree optimised for fast searches, specifically uses red–black trees as part of its data structure.[25]

As of Java 8, the HashMap has been modified such that instead of using a LinkedList to store different elements with colliding hashcodes, a red–black tree is used. This results in the improvement of time complexity of searching such an element from $O(m)$ to $O(\log m)$ where m is the number of elements with colliding hashes.[26]

Implementation

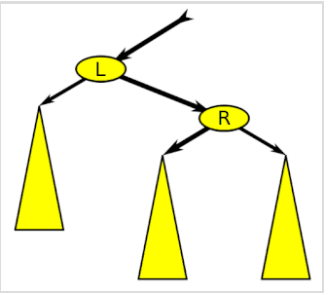
The read-only operations, such as search or tree traversal, on a red–black tree require no modification from those used for binary search trees, because every red–black tree is a special case of a simple binary search tree. However, the immediate result of an insertion or removal may violate the properties of a red–black tree, the restoration of which is called *rebalancing* so that red–black trees become self-balancing. Rebalancing (i.e. color changes) has a worst-case time complexity of $O(\log n)$ and average of $O(1)$,[27]:310[16]:158 though these are very quick in practice. Additionally, rebalancing takes no more than three tree rotations[28] (two for insertion).

This is an example implementation of insert and remove in C. Below are the data structures and the `rotate_subtree` helper function used in the insert and remove examples.

```c
typedef enum Color: char {
    BLACK,
    RED
} Color;

typedef enum Direction: char {
    LEFT,
    RIGHT
} Direction;

// red-black tree node
typedef struct Node {
    struct Node* parent; // null for the root node
    union {
        // Union so we can use ->left/->right or ->child[0]/->child[1]
        struct {
            struct Node* left;
            struct Node* right;
        };
        struct Node* child[2];
    };
    Color color;
    int key;
} Node;

typedef struct {
    struct Node* root;
} Tree;

static Direction direction(const Node* N) {
```

```
    return N == N->parent->right ? RIGHT : LEFT
};

Node* rotate_subtree(Tree* tree, Node* sub, Direction dir) {
    Node* sub_parent = sub->parent;
    Node* new_root = sub->child[1 - dir]; // 1 - dir is the opposite direction
    Node* new_child = new_root->child[dir];

    sub->child[1 - dir] = new_child;

    if (new_child) {
        new_child->parent = sub;
    }

    new_root->child[dir] = sub;

    new_root->parent = sub_parent;
    sub->parent = new_root;
    if (sub_parent) {
        sub_parent->child[sub == sub_parent->right] = new_root;
    } else {
        tree->root = new_root;
    }

    return new_root;
}
```

Notes to the sample code and diagrams of insertion and removal



Left rotation and right rotation, animated.

The proposal breaks down both insertion and removal (not mentioning some very simple cases) into six constellations of nodes, edges, and colors, which are called cases. The proposal contains, for both insertion and removal, exactly one case that advances one black level closer to the root and loops, the other five cases rebalance the tree of their own. The more complicated cases are pictured in a diagram.

- 🔴 symbolises a red node and ⚫ a (non-NULL) black node (of black height ≥ 1), 🌗 symbolises the color red or black of a non-NULL node, but the same color throughout the same diagram. NULL nodes are not represented in the diagrams.
- The variable **N** denotes the current node, which is labeled N or N in the diagrams.
- A diagram contains three columns and two to four actions. The left column shows the first iteration, the right column the higher iterations, the middle column shows the segmentation of a case into its different actions.[29]

1. The action "entry" shows the constellation of nodes with their colors which defines a case and mostly violates some of the requirements.
   A blue border rings the current node **N** and the other nodes are labeled according to their relation to **N**.
2. If a rotation is considered useful, this is pictured in the next action, which is labeled "rotation".
3. If some recoloring is considered useful, this is pictured in the next action, which is labeled "color".[30]
4. If there is still some need to repair, the cases make use of code of other cases and this

after a reassignment of the current node **N**, which then again carries a blue ring and relative to which other nodes may have to be reassigned also. This action is labeled "reassign".
For both, insert and delete, there is (exactly) one case which iterates one black level closer to the root; then the reassigned constellation satisfies the respective loop invariant.

- A possibly numbered triangle with a black circle atop △ represents a red–black subtree (connected to its parent according to requirement 3) with a black height equal to the iteration level minus one, i.e. zero in the first iteration. Its root may be red or black.
  A possibly numbered triangle △ represents a red–black subtree with a black height one less, i.e. its parent has black height zero in the second iteration.

Remark

For simplicity, the sample code uses the disjunction:

```
U == NULL || U->color == BLACK // considered black
```

and the conjunction:

```
U != NULL && U->color == RED   // not considered black
```

Thereby, it must be kept in mind that both statements are *not* evaluated in total, if `U == NULL`. Then in both cases `U->color` is not touched (see Short-circuit evaluation). (The comment `considered black` is in accordance with requirement 2.)
The related `if`-statements have to occur far less frequently if the proposal[29] is realised.

Insertion

Insertion begins by placing the new (non-NULL) node, say **N**, at the position in the binary search tree of a NULL node whose in-order predecessor's key compares less than the new node's key, which in turn compares less than the key of its in-order successor. (Frequently, this positioning is the result of a search within the tree immediately preceding the insert operation and consists of a node `P` together with a direction `dir` with `P->child[dir] == NULL`.) The newly inserted node is temporarily colored red so that all paths contain the same number of black nodes as before. But if its parent, say **P**, is also red then this action introduces a red-violation.

```
// parent is optional
void insert(Tree* tree, Node* node, Node* parent, Direction dir) {
    node->color = RED;
    node->parent = parent;

    if (!parent) {
        tree->root = node;
        return;
    }

    parent->child[dir] = node;

    // rebalance the tree
    do {
        // Case #1
        if (parent->color == BLACK) {
            return;
        }

        Node* grandparent = parent->parent;
```

```
        if (!grandparent) {
            // Case #4
            parent->color = BLACK;
            return;
        }

        dir = direction(parent);
        Node* uncle = grandparent->child[1 - dir];
        if (!uncle || uncle->color == BLACK) {
            if (node == parent->child[1 - dir]) {
                // Case #5
                rotate_subtree(tree, parent, dir);
                node = parent;
                parent = grandparent->child[dir];
            }

            // Case #6
            rotate_subtree(tree, grandparent, 1 - dir);
            parent->color = BLACK;
            grandparent->color = RED;
            return;
        }

        // Case #2
        parent->color = BLACK;
        uncle->color = BLACK;
        grandparent->color = RED;
        node = grandparent;

    } while (parent = node->parent);

    // Case #3
    return;

}
```

The rebalancing loop of the insert operation has the following invariants:

- **Node** is the current node, initially the insertion node.
- **Node** is red at the beginning of each iteration.
- Requirement 3 is satisfied for all pairs node←parent with the possible exception **node←parent** when **parent** is also red (a red-violation at **node**).
- All other properties (including requirement 4) are satisfied throughout the tree.

Notes to the insert diagrams

- In the diagrams, **P** is used for **N**'s parent, **G** for its grandparent, and **U** for its uncle. In the table, "—" indicates the root.
- The diagrams show the parent node **P** as the left child of its parent **G** even though it is possible for **P** to be on either side. The sample code covers both possibilities by means of the side variable `dir`.
- The diagrams show the cases where **P** is red also, the red-violation.
- The column *x* indicates the change in child direction, i.e. o (for "outer") means that **P** and **N** are both left or both right children, whereas i (for "inner") means that the child direction changes from **P**'s to **N**'s.
- The column group *before* defines the case, whose name is given in the column *case*. Thereby possible values in cells left empty are ignored. So in case **I2** the sample code covers both possibilities of child directions of **N**, although the corresponding diagram

shows only one.

- The rows in the synopsis are ordered such that the coverage of all possible RB cases is easily comprehensible.
- The column *rotation* indicates whether a rotation contributes to the rebalancing.
- The column *assignment* shows an assignment of **N** before entering a subsequent step. This possibly induces a reassignment of the other nodes **P**, **G**, **U** also.

before				*case*	*rotation*	*assignment*	*after*				*next*	*Δh*
P	**G**	**U**	*x*				**P**	**G**	**U**	*x*		
●				**I1**							✓	
●	●	●		**I2**		**N:=G**	?				?	2
—				**I3**							✓	
●	—			**I4**			●				✓	
●	●	●	i	**I5**	**P↶N**	**N:=P**	●	●	●	o	**I6**	0
●	●	●	o	**I6**	**P↷G**		●	●	●		✓	

Insertion

This synopsis shows in its *before* columns, that all possible cases[31] of constellations are covered.

- If something has been changed by the case, this is shown in the column group *after*.
- A ✓ sign in column *next* signifies that the rebalancing is complete with this step. If the column *after* determines exactly one case, this case is given as the subsequent one, otherwise there are question marks.
- In case **2** the problem of rebalancing is escalated $\Delta h = 2$ tree levels or 1 black level higher in the tree, in that the grandfather **G** becomes the new current node **N**. So it takes maximally $\frac{h}{2}$ steps of iteration to repair the tree (where h is the height of the tree). Because the probability of escalation decreases exponentially with each step the total rebalancing cost is constant on average, indeed amortized constant.
- Rotations occur in cases **I6** and **I5 + I6** – outside the loop. Therefore, at most two rotations occur in total.

Insert case 1

The current node's parent **P** is black, so requirement 3 holds. Requirement 4 holds also according to the loop invariant.

Insert case 2

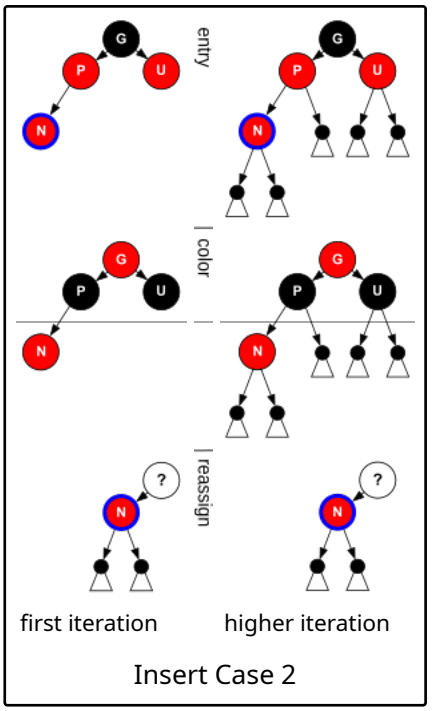
If both the parent **P** and the uncle **U** are red, then both of them can be repainted black and the grandparent **G** becomes red for maintaining requirement 4. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed. However, the grandparent **G** may now violate requirement 3, if it has a red parent. After relabeling **G** to **N** the loop invariant is fulfilled so that the rebalancing can be iterated on one black level (= 2 tree levels) higher.

Insert case 3

Insert case 2 has been executed for $\frac{h-1}{2}$ times and the total height of the tree has increased by 1, now being h. The current node **N** is the (red) root of the tree, and all RB-properties are satisfied.

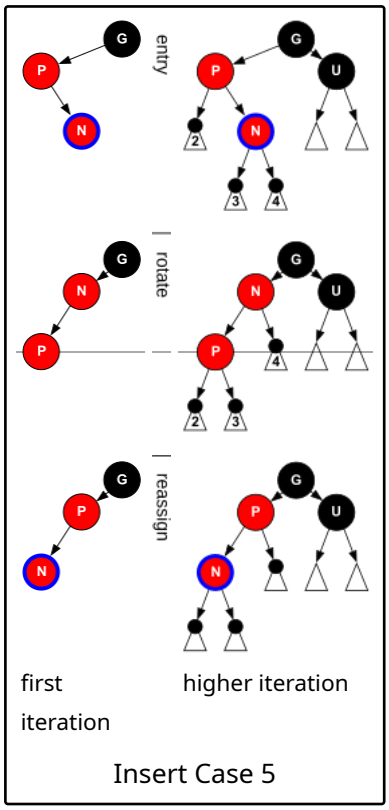
Insert case 4

The parent **P** is red and the root. Because **N** is also red, requirement 3 is violated. But after switching **P**'s color the tree is in RB-shape. The black height of the tree increases by 1.



Insert Case 2

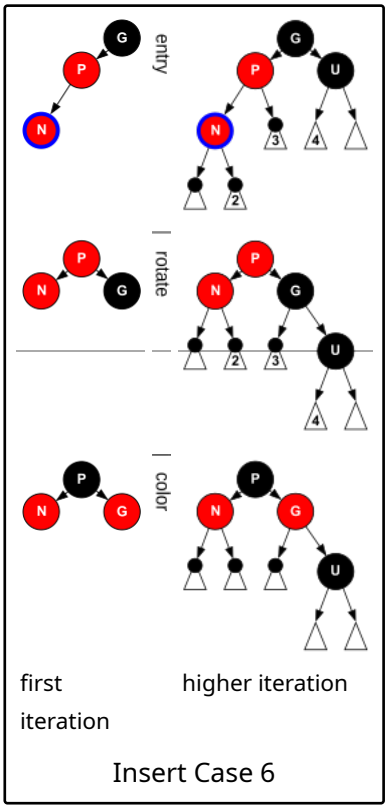
Insert case 5



Insert Case 5

The parent **P** is red but the uncle **U** is black. The ultimate goal is to rotate the parent node **P** to the grandparent position, but this will not work if **N** is an "inner" grandchild of **G** (i.e., if **N** is the left child of the right child of **G** or the right child of the left child of **G**). A `dir`-rotation at **P** switches the roles of the current node **N** and its parent **P**. The rotation adds paths through **N** (those in the subtree labeled **2**, see diagram) and removes paths through **P** (those in the subtree labeled **4**). But both **P** and **N** are red, so requirement 4 is preserved. Requirement 3 is restored in case 6.

Insert case 6



Insert Case 6

The current node **N** is now certain to be an "outer" grandchild of **G** (left of left child or right of right child). Now `(1-dir)`-rotate at **G**, putting **P** in place of **G** and making **P** the parent of **N** and **G**. **G** is black and its former child **P** is red, since requirement 3 was violated. After switching the colors of **P** and **G** the resulting tree satisfies requirement 3. Requirement 4 also remains satisfied, since all paths that went through the black **G** now go through the black **P**.

Because the algorithm transforms the input without using an auxiliary data structure and using only a small amount of extra storage space for auxiliary variables it is in-place.

Removal

Simple cases

- When the deleted node has **2 children** (non-NULL), then we can swap its value with its in-order successor (the leftmost child of the right subtree), and then delete the successor instead. Since the successor is leftmost, it can only have a right child (non-NULL) or no child at all.
- When the deleted node has **only 1 child** (non-NULL). In this case, just replace the node with its child, and color it black.

    - The single child (non-NULL) must be red according to conclusion 5, and the deleted node must be black according to requirement 3.
- When the deleted node **has no children** (both NULL) and is the root, replace it with NULL. The tree is empty.
- When the deleted node **has no children** (both NULL), and **is red**, simply remove the leaf node.
- When the deleted node **has no children** (both NULL), and **is black**, deleting it will create an imbalance, and requires a rebalance, as covered in the next section.

Removal of a black non-root leaf

The complex case is when **N** is not the root, colored black and has no proper child (⇔ only NULL children). In the first iteration, **N** is replaced by NULL.

```
void remove(Tree* tree, Node* node) {
    Node* parent = node->parent;

    Node* sibling;
    Node* close_nephew;
    Node* distant_nephew;

    Direction dir = direction(node);

    parent->child[dir] = NULL;
    goto start_balance;

    do {
        dir = direction(node);
start_balance:
        sibling = parent->child[1 - dir];
        distant_nephew = sibling->child[1 - dir];
        close_nephew = sibling->child[dir];
        if (sibling->color == RED) {
            // Case #3
            rotate_subtree(tree, parent, dir);
            parent->color = RED;
            sibling->color = BLACK;
            sibling = close_nephew;

            distant_nephew = sibling->child[1 - dir];
            if (distant_nephew && distant_nephew->color == RED) {
                goto case_6;
            }
            close_nephew = sibling->child[dir];
            if (close_nephew && close_nephew->color == RED) {
                goto case_5;
            }

            // Case #4
            sibling->color = RED;
            parent->color = BLACK;
            return;
        }

        if (distant_nephew && distant_nephew->color == RED) {
            goto case_6;
        }

        if (close_nephew && close_nephew->color == RED) {
            goto case_5;
        }

        if (!parent) {
            // Case #1
            return;
        }
```

```
        if (parent->color == RED) {
            // Case #4
            sibling->color = RED;
            parent->color = BLACK;
            return;
        }

        // Case #2
        sibling->color = RED;
        node = parent;

    } while (parent = node->parent);

case_5:

    rotate_subtree(tree, sibling, 1 - dir);
    sibling->color = RED;
    close_nephew->color = BLACK;
    distant_nephew = sibling;
    sibling = close_nephew;

case_6:

    rotate_subtree(tree, parent, dir);
    sibling->color = parent->color;
    parent->color = BLACK;
    distant_nephew->color = BLACK;
    return;
}
```

The rebalancing loop of the delete operation has the following invariant:

- At the beginning of each iteration the black height of **N** equals the iteration number minus one, which means that in the first iteration it is zero and that **N** is a true black node ● in higher iterations.
- The number of black nodes on the paths through **N** is one less than before the deletion, whereas it is unchanged on all other paths, so that there is a black-violation at **P** if other paths exist.
- All other properties (including requirement 3) are satisfied throughout the tree.

Notes to the delete diagrams

- In the diagrams below, **P** is used for **N**'s parent, **S** for the sibling of **N**, **C** (meaning *close* nephew) for **S**'s child in the same direction as **N**, and **D** (meaning *distant* nephew) for **S**'s other child (**S** cannot be a NULL node in the first iteration, because it must have black height one, which was the black height of **N** before its deletion, but **C** and **D** may be NULL nodes).
- The diagrams show the current node **N** as the left child of its parent **P** even though it is possible for **N** to be on either side. The code samples cover both possibilities by means of the side variable `dir`.
- At the beginning (in the first iteration) of removal, **N** is the NULL node replacing the node to be deleted. Because its location in parent's node is the only thing of importance, it is symbolised by ↙ (meaning: the current node **N** is a NULL node and left child) in the left column of the delete diagrams. As the operation proceeds also proper nodes (of black height ≥ 1) may become current (see e.g. case **2**).

- By counting the black bullets (● and 🯅) in a delete diagram it can be observed that the paths through **N** have one bullet less than the other paths. This means a black-violation at **P**—if it exists.
- The color constellation in column group *before* defines the case, whose name is given in the column *case*. Thereby possible values in cells left empty are ignored.
- The rows in the synopsis are ordered such that the coverage of all possible RB cases is easily comprehensible.

before				*case*	*rotation*	*assignment*	*after*				*next*	*Δh*
P	**C**	**S**	**D**				**P**	**C**	**S**	**D**		
—				**D1**							✓	
●	●	●	●	**D2**		**N:=P**	?				?	1
●	●	◒	●	**D3**	**P↶S**	**N:=N**	◒		●	◒	**D6**	0
							◒	◒	●	●	**D5**	0
							◒	●	●	●	**D4**	0
◒	●	●	●	**D4**			●	●	◒	●	✓	
◓	◒	●	●	**D5**	**C↷S**	**N:=N**	◓		●	◒	**D6**	0
◓		●	◒	**D6**	**P↶S**		●		◓	●	✓	

Deletion

This synopsis shows in its *before* columns, that all possible cases[31] of color constellations are covered.

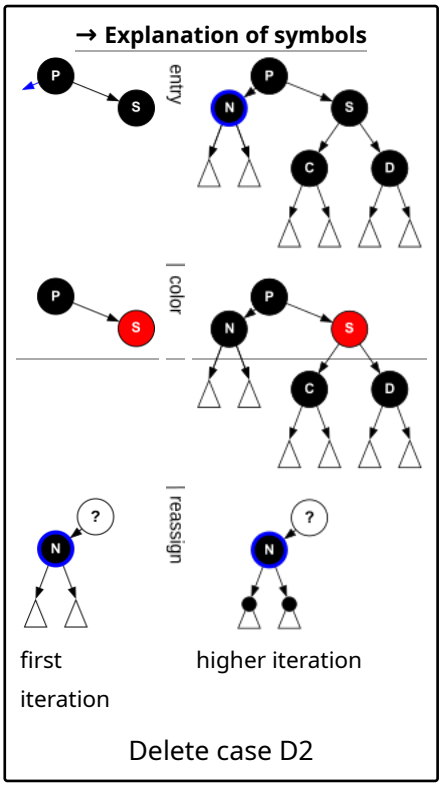
- The column *rotation* indicates whether a rotation contributes to the rebalancing.
- The column *assignment* shows an assignment of **N** before entering a subsequent iteration step. This possibly induces a reassignment of the other nodes **P**, **C**, **S**, **D** also.
- If something has been changed by the case, this is shown in the column group *after*.
- A ✓ sign in column *next* signifies that the rebalancing is complete with this step. If the column *after* determines exactly one case, this case is given as the subsequent one, otherwise there are question marks.
- The loop is where the problem of rebalancing is escalated $\Delta h = 1$ level higher in the tree in that the parent **P** becomes the new current node **N**. So it takes maximally h iterations to repair the tree (where h is the height of the tree). Because the probability of escalation decreases exponentially with each iteration the total rebalancing cost is constant on average, indeed amortized constant. (Just as an aside: Mehlhorn & Sanders point out: "AVL trees do *not* support constant amortized update costs."[16]:165,158 This is true for the rebalancing after a deletion, but not AVL insertion.[32])
- Out of the body of the loop there are exiting branches to the cases **3**, **6**, **5**, **4**, and **1**; section "Delete case 3" of its own has three different exiting branches to the cases **6**, **5** and **4**.
- Rotations occur in cases **6** and **5 + 6** and **3 + 5 + 6** – all outside the loop. Therefore, at most three rotations occur in total.

Delete case 1

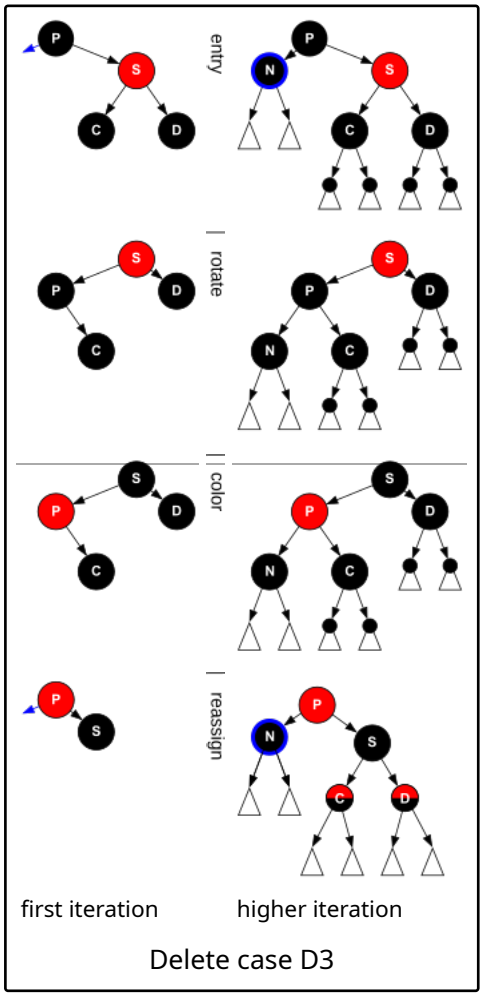
The current node **N** is the new root. One black node has been removed from every path, so the RB-properties are preserved. The black height of the tree decreases by 1.

Delete case 2

P, **S**, and **S**'s children are black. After painting **S** red all paths passing through **S**, which are precisely those paths *not* passing through **N**, have one less black node. Now all paths in the subtree rooted by **P** have the same number of black nodes, but one fewer than the paths that do not pass through **P**, so requirement 4 may still be violated. After relabeling **P** to **N** the loop invariant is fulfilled so that the rebalancing can be iterated on one black level (= 1 tree level) higher.



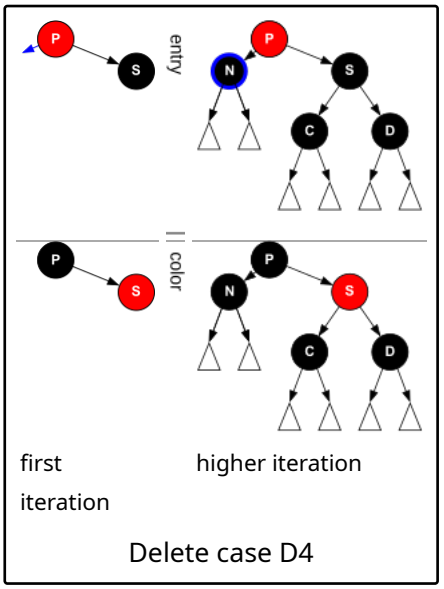
Delete case 3



The sibling **S** is red, so **P** and the nephews **C** and **D** have to be black. A `dir`-rotation at **P** turns **S** into **N**'s grandparent. Then after reversing the colors of **P** and **S**, the path through **N** is still short one black node. But **N** now has a red parent **P** and after the reassignment a black sibling **S**, so the transformations in cases 4, 5, or 6 are able to restore the RB-shape.

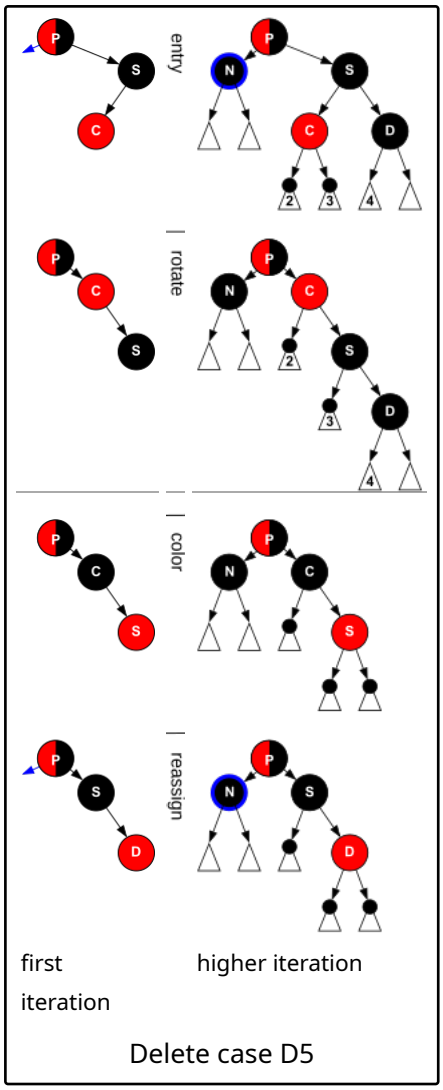
Delete case 4

The sibling **S** and **S**'s children are black, but **P** is red. Exchanging the colors of **S** and **P** does not affect the number of black nodes on paths going through **S**, but it does add one to the number of black nodes on paths going through **N**, making up for the deleted black node on those paths.

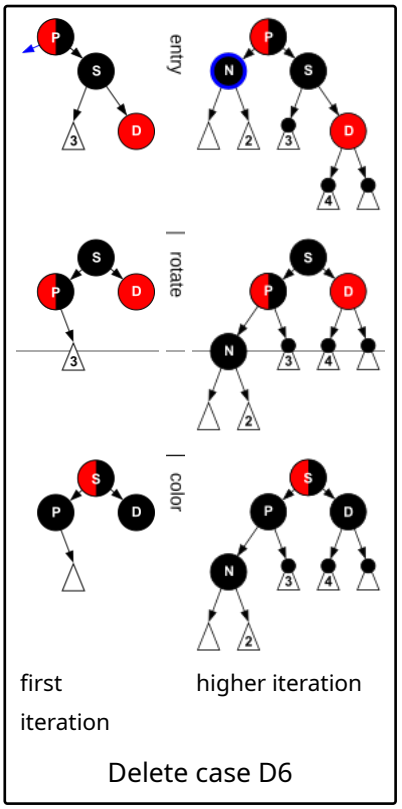


Delete case 5

The sibling **S** is black, **S**'s close child **C** is red, and **S**'s distant child **D** is black. After a `(1-dir)`-rotation at **S** the nephew **C** becomes **S**'s parent and **N**'s new sibling. The colors of **S** and **C** are exchanged. All paths still have the same number of black nodes, but now **N** has a black sibling whose distant child is red, so the constellation is fit for case D6. Neither **N** nor its parent **P** are affected by this transformation, and **P** may be red or black (◐ in the diagram).



Delete case 6



The sibling **S** is black, **S**'s distant child **D** is red. After a `dir`-rotation at **P** the sibling **S** becomes the parent of **P** and **S**'s distant child **D**. The colors of **P** and **S** are exchanged, and **D** is made black. The whole subtree still has the same color at its root **S**, namely either red or black (◐ in the diagram), which refers to the same color both before and after the transformation. This way requirement 3 is preserved. The paths in the subtree not passing through **N** (i.o.w. passing through **D** and node **3** in the diagram) pass through the same number of black nodes as before, but **N** now has one additional black ancestor: either **P** has become black, or it was black and **S** was added as a black grandparent. Thus, the paths passing through **N** pass through one additional black node, so that requirement 4 is restored and the total tree is in RB-shape.

Because the algorithm transforms the input without using an auxiliary data structure and using only a small amount of extra storage space for auxiliary variables it is in-place.

Proof of bounds

For $h \in \mathbb{N}$ there is a red–black tree of height h with

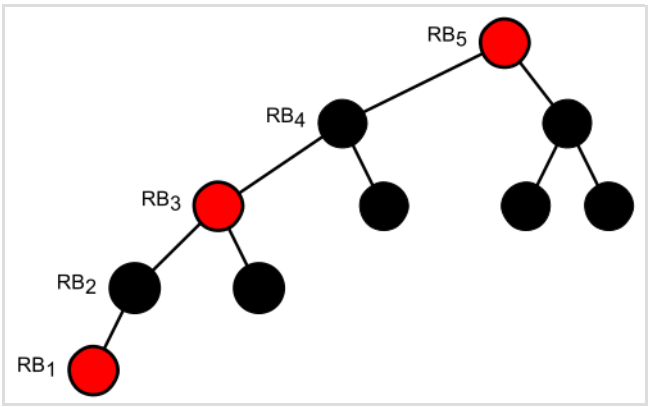


Figure 2: Red–black trees RB_h of heights h=1,2,3,4,5,
each with minimal number 1,2,4,6 resp. 10 of nodes.

$$m_h = 2^{\lfloor (h+1)/2 \rfloor} + 2^{\lfloor h/2 \rfloor} - 2$$
$$= \begin{cases} 2 \cdot 2^{\frac{h}{2}} - 2 = 2^{\frac{h}{2}+1} - 2 & \text{if } h \text{ even} \\ 3 \cdot 2^{\frac{h-1}{2}} - 2 & \text{if } h \text{ odd} \end{cases}$$

nodes ($\lfloor \ \rfloor$ is the floor function) and there is no red–black tree of this tree height with fewer nodes—therefore it is **minimal**.
Its black height is $\lceil h/2 \rceil$ (with black root) or for odd h (then with a red root) also $(h-1)/2$.

Proof

For a red–black tree of a certain height to have minimal number of nodes, it must have exactly one longest path with maximal number of red nodes, to achieve a maximal tree height with a minimal black height. Besides this path all other nodes have to be black.[15]:444 Proof sketch If a node is taken off this tree it either loses height or some RB property.

The RB tree of height $h = 1$ with red root is minimal. This is in agreement with

$$m_1 = 2^{\lfloor (1+1)/2 \rfloor} + 2^{\lfloor 1/2 \rfloor} - 2 = 2^1 + 2^0 - 2 = 1 \ .$$

A minimal RB tree (RB_h in figure 2) of height $h > 1$ has a root whose two child subtrees are of different height. The higher child subtree is also a minimal RB tree, RB_{h-1}, containing also a longest path that defines its height $h-1$; it has m_{h-1} nodes and the black height $\lfloor (h-1)/2 \rfloor =: s$. The other subtree is a perfect binary tree of (black) height s having $2^s - 1 = 2^{\lfloor (h-1)/2 \rfloor} - 1$ black nodes—and no red node. Then the number of nodes is by induction

$$m_h = \underbrace{(m_{h-1})}_{\text{(higher subtree)}} + \underbrace{(1)}_{\text{(root)}} + \underbrace{(2^{\lfloor (h-1)/2 \rfloor} - 1)}_{\text{(second subtree)}}$$

resulting in

$$m_h = (2^{\lfloor h/2 \rfloor} + 2^{\lfloor (h-1)/2 \rfloor} - 2) + 2^{\lfloor (h-1)/2 \rfloor}$$
$$= 2^{\lfloor h/2 \rfloor} + 2^{\lfloor (h+1)/2 \rfloor} - 2 \ \blacksquare$$

The graph of the function m_h is convex and piecewise linear with breakpoints at $(h = 2k \mid m_{2k} = 2 \cdot 2^k - 2)$ where $k \in \mathbb{N}$. The function has been tabulated as $m_h =$ A027383(h–1) for $h \geq 1$ (sequence A027383 in the OEIS).

Solving the function for h

The inequality $9 > 8 = 2^3$ leads to $3 > 2^{3/2}$, which for odd h leads to

$$m_h = 3 \cdot 2^{(h-1)/2} - 2 = \left(3 \cdot 2^{-3/2}\right) \cdot 2^{(h+2)/2} - 2 > 2 \cdot 2^{h/2} - 2.$$

So in both, the even and the odd case, h is in the interval

$$\log_2(n+1) \quad \leq h \leq \quad 2\log_2(n+2) - 2 = 2\log_2(n/2+1) \quad \left[< 2\log_2(n+1) \right]$$

(perfect binary tree) (minimal red–black tree)

with n being the number of nodes.[33]

Conclusion

A red–black tree with n nodes (keys) has tree height $h \in O(\log n)$.

Set operations and bulk operations

In addition to the single-element insert, delete and lookup operations, several set operations have been defined on red–black trees: union, intersection and set difference. Then fast *bulk* operations on insertions or deletions can be implemented based on these set functions. These set operations rely on two helper operations, *Split* and *Join*. With the new operations, the implementation of red–black trees can be more efficient and highly-parallelizable.[34] In order to achieve its time complexities this implementation requires that the root is allowed to be either red or black, and that every node stores its own **black height**.

- *Join*: The function *Join* is on two red–black trees t_1 and t_2 and a key k, where $t_1 < k < t_2$, i.e. all keys in t_1 are less than k, and all keys in t_2 are greater than k. It returns a tree containing all elements in t_1, t_2 also as k.

  If the two trees have the same black height, *Join* simply creates a new node with left subtree t_1, root k and right subtree t_2. If both t_1 and t_2 have black root, set k to be red. Otherwise k is set black.
  If the black heights are unequal, suppose that t_1 has larger black height than t_2 (the other case is symmetric). *Join* follows the right spine of t_1 until a black node c, which is balanced with t_2. At this point a new node with left child c, root k (set to be red) and right child t_2 is created to replace c. The new node may invalidate the red–black invariant because at most three red nodes can appear in a row. This can be fixed with a double rotation. If double red issue propagates to the root, the root is then set to be black, restoring the properties. The cost of this function is the difference of the black heights between the two input trees.

- *Split*: To split a red–black tree into two smaller trees, those smaller than key x, and those larger than key x, first draw a path from the root by inserting x into the red–black tree. After this insertion, all values less than x will be found on the left of the path, and all values greater than x will be found on the right. By applying *Join*, all the subtrees on the left side are merged bottom-up using keys on the path as intermediate nodes from bottom to top to form the left tree, and the right part is symmetric.

  For some applications, *Split* also returns a Boolean value denoting if x appears in the tree. The cost of *Split* is $O(\log n)$, order of the height of the tree. This algorithm actually has nothing to do with any special properties of a red–black tree, and may be used on any tree with a *join* operation, such as an AVL tree.

The join algorithm is as follows:

```
function joinRightRB(T_L, k, T_R):
    if (T_L.color=black) and (T_L.blackHeight=T_R.blackHeight):
        return Node(T_L,⟨k,red⟩,T_R)
    T'=Node(T_L.left,⟨T_L.key,T_L.color⟩,joinRightRB(T_L.right,k,T_R))
    if (T_L.color=black) and (T'.right.color=T'.right.right.color=red):
        T'.right.right.color=black;
        return rotateLeft(T')
    return T' /* T''[recte T'] */

function joinLeftRB(T_L, k, T_R):
  /* symmetric to joinRightRB */

function join(T_L, k, T_R):
    if T_L.blackHeight>T_R.blackHeight:
        T'=joinRightRB(T_L,k,T_R)
        if (T'.color=red) and (T'.right.color=red):
            T'.color=black
        return T'
    if T_R.blackHeight>T_L.blackHeight:
        /* symmetric */
    if (T_L.color=black) and (T_R.color=black):
        return Node(T_L,⟨k,red⟩,T_R)
    return Node(T_L,⟨k,black⟩,T_R)
```

The split algorithm is as follows:

```
function split(T, k):
    if (T = NULL) return (NULL, false, NULL)
    if (k = T.key) return (T.left, true, T.right)
    if (k < T.key):
        (L',b,R') = split(T.left, k)
        return (L',b,join(R',T.key,T.right))
    (L',b,R') = split(T.right, k)
    return (join(T.left,T.key,L'),b,T.right)
```

The union of two red–black trees t_1 and t_2 representing sets A and B, is a red–black tree t that represents $A \cup B$. The following recursive function computes this union:

```
function union(t_1, t_2):
    if t_1 = NULL return t_2
    if t_2 = NULL return t_1
    (L_1,b,R_1)=split(t_1,t_2.key)
    proc1=start:
        T_L=union(L_1,t_2.left)
    proc2=start:
        T_R=union(R_1,t_2.right)
    wait all proc1,proc2
    return join(T_L, t_2.key, T_R)
```

Here, *split* is presumed to return two trees: one holding the keys less its input key, one holding the greater keys. (The algorithm is non-destructive, but an in-place destructive version exists also.)

The algorithm for intersection or difference is similar, but requires the *Join2* helper routine that is the same as *Join* but without the middle key. Based on the new functions for union, intersection or difference, either one key or multiple keys can be inserted to or deleted from the red–black tree. Since *Split* calls *Join* but does not deal with the balancing criteria of red–black trees directly, such an implementation is usually called the "join-based" implementation.

The complexity of each of union, intersection and difference is $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ for two red–

black trees of sizes m and $n(\geq m)$. This complexity is optimal in terms of the number of comparisons. More importantly, since the recursive calls to union, intersection or difference are independent of each other, they can be executed in parallel with a parallel depth $O(\log m \log n)$.[34] When $m = 1$, the join-based implementation has the same computational directed acyclic graph (DAG) as single-element insertion and deletion if the root of the larger tree is used to split the smaller tree.

Parallel algorithms

Parallel algorithms for constructing red–black trees from sorted lists of items can run in constant time or $O(\log \log n)$ time, depending on the computer model, if the number of processors available is asymptotically proportional to the number n of items where $n \to \infty$. Fast search, insertion, and deletion parallel algorithms are also known.[35]

The join-based algorithms for red–black trees are parallel for bulk operations, including union, intersection, construction, filter, map-reduce, and so on.

Parallel bulk operations

Basic operations like insertion, removal or update can be parallelised by defining operations that process bulks of multiple elements. It is also possible to process bulks with several basic operations, for example bulks may contain elements to insert and also elements to remove from the tree.

The algorithms for bulk operations aren't just applicable to the red–black tree, but can be adapted to other sorted sequence data structures also, like the 2–3 tree, 2–3–4 tree and (a,b)-tree. In the following different algorithms for bulk insert will be explained, but the same algorithms can also be applied to removal and update. Bulk insert is an operation that inserts each element of a sequence I into a tree T.

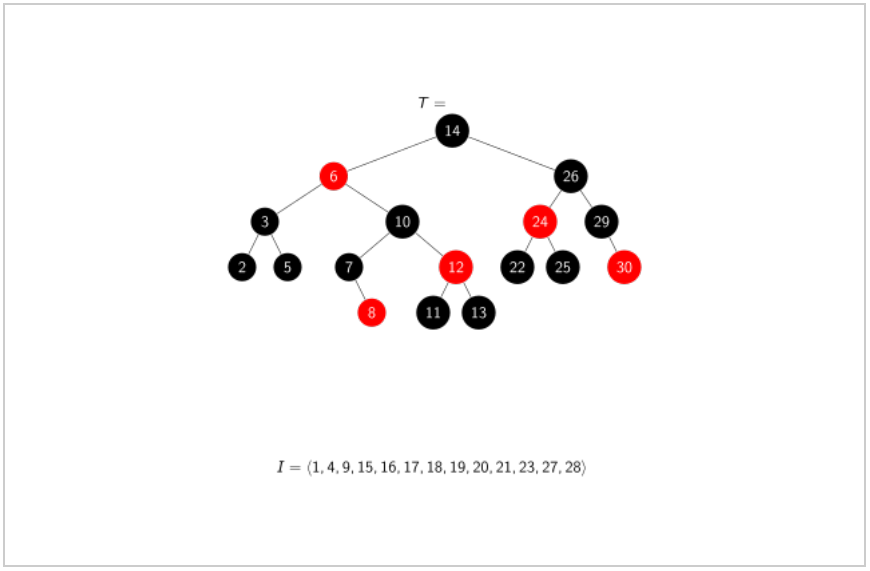
Join-based

This approach can be applied to every sorted sequence data structure that supports efficient join- and split-operations.[36] The general idea is to split I and T in multiple parts and perform the insertions on these parts in parallel.

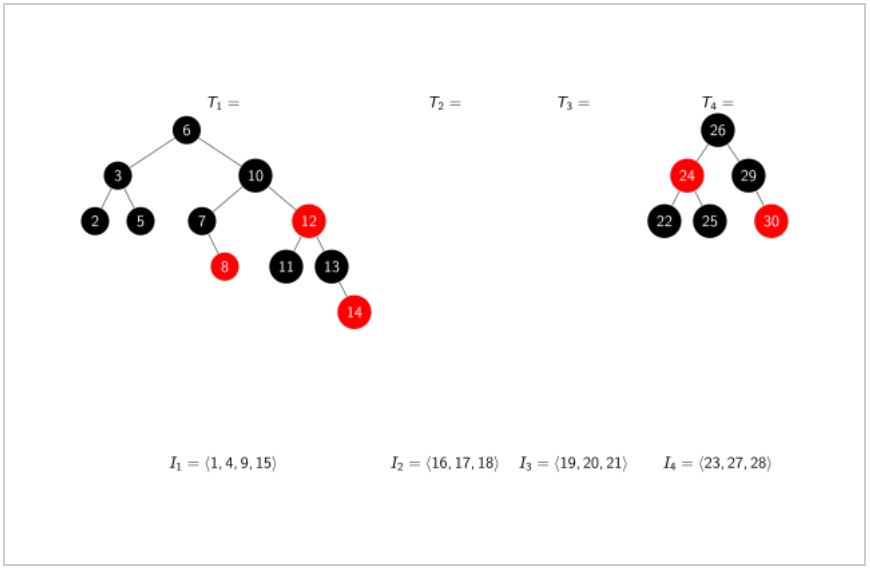
1. First the bulk I of elements to insert must be sorted.
2. After that, the algorithm splits I into $k \in \mathbb{N}^+$ parts $\langle I_1, \cdots, I_k \rangle$ of about equal sizes.
3. Next the tree T must be split into k parts $\langle T_1, \cdots, T_k \rangle$ in a way, so that the ranges of I_m and T_n overlap for corresponding parts only ($m = n$); in other words, thanks to the ordering, for every $j \in \mathbb{N}^+ \mid 1 \leq j < k$ following constraints hold:
    1. $\mathbf{last}(I_j) < \mathbf{first}(T_{j+1})$
    2. $\mathbf{last}(T_j) < \mathbf{first}(I_{j+1})$
4. Now the algorithm inserts each element of I_j into T_j sequentially. This step must be performed for every j, which can be done by up to k processors in parallel.
5. Finally, the resulting trees will be joined to form the final result of the entire operation.

Note that in Step 3 the constraints for splitting I assure that in Step 5 the trees can be joined again and the

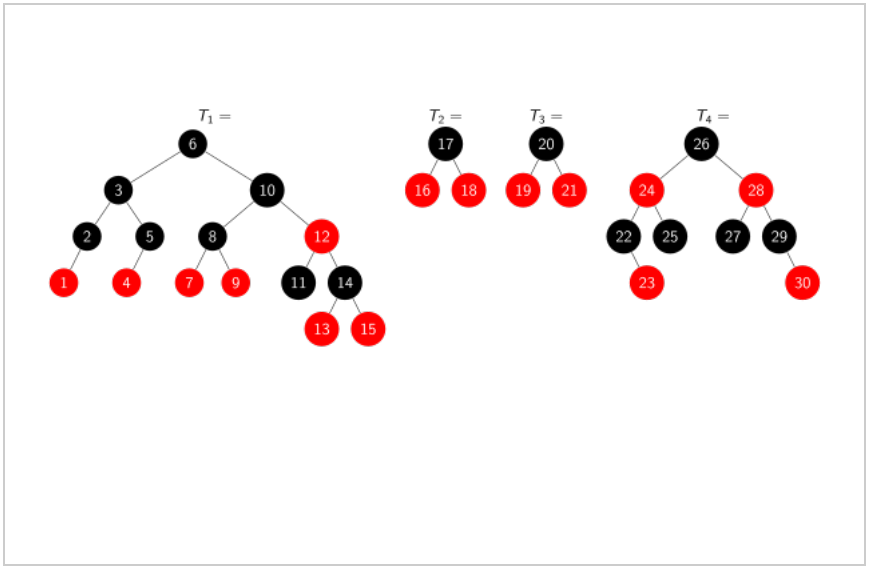
resulting sequence is sorted.



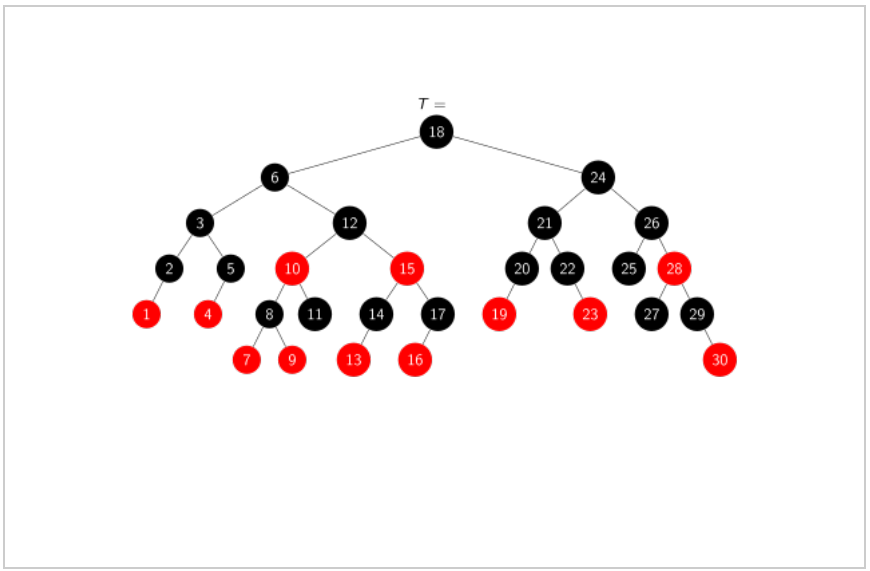
initial tree



split I and T



insert into the split T



join T

The pseudo code shows a simple divide-and-conquer implementation of the join-based algorithm for bulk-insert. Both recursive calls can be executed in parallel. The join operation used here differs from the version explained in this article, instead join2 is used, which misses the second parameter k.

```
bulkInsert(T, I, k):
    I.sort()
    bulklInsertRec(T, I, k)

bulkInsertRec(T, I, k):
    if k = 1:
        forall e in I: T.insert(e)
    else
        m := ⌊size(I) / 2⌋
        (T1, _, T2) := split(T, I[m])
        bulkInsertRec(T1, I[0 .. m], ⌈k / 2⌉)
            || bulkInsertRec(T2, I[m + 1 .. size(I) - 1], ⌊k / 2⌋)
        T ← join2(T1, T2)
```

Execution time

Sorting I is not considered in this analysis.

#recursion levels	$\in O(\log k)$
T(split) + T(join)	$\in O(\log \|T\|)$
insertions per thread	$\in O\left(\frac{\|I\|}{k}\right)$
T(insert)	$\in O(\log \|T\|)$
T(bulkInsert) with k = #processors	$\in O\left(\log k \log \|T\| + \frac{\|I\|}{k} \log \|T\|\right)$

This can be improved by using parallel algorithms for splitting and joining. In this case the execution time is $\in O\left(\log |T| + \frac{|I|}{k} \log |T|\right)$.[37]

Work

#splits, #joins	$\in O(k)$
W(split) + W(join)	$\in O(\log \|T\|)$
#insertions	$\in O(\|I\|)$
W(insert)	$\in O(\log \|T\|)$
W(bulkInsert)	$\in O(k \log \|T\| + \|I\| \log \|T\|)$

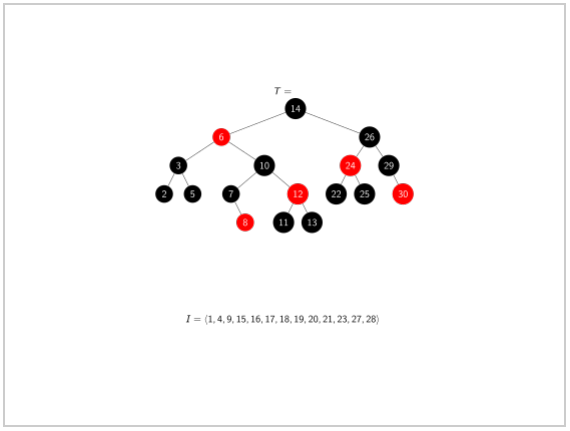
Pipelining

Another method of parallelizing bulk operations is to use a pipelining approach.[38] This can be done by breaking the task of processing a basic operation up into a sequence of subtasks. For multiple basic operations the subtasks can be processed in parallel by assigning each subtask to a separate processor.

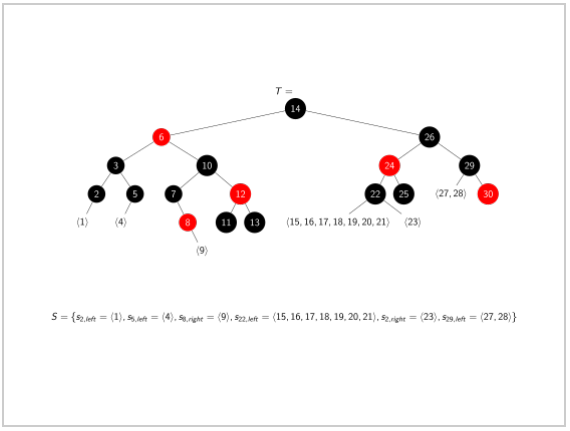
1. First the bulk I of elements to insert must be sorted.
2. For each element in I the algorithm locates the according insertion position in T. This can be done in parallel for each element $\in I$ since T won't be mutated in this process. Now I must be divided into subsequences S according to the insertion position of each element. For example $s_{n,left}$ is the subsequence of I that contains the elements whose insertion position would be to the left of node n.
3. The middle element $m_{n,dir}$ of every subsequence $s_{n,dir}$ will be inserted into T as a new node n'. This can be done in parallel for each $m_{n,dir}$ since by definition the insertion position of each $m_{n,dir}$ is unique. If $s_{n,dir}$ contains elements to the left or to the right of $m_{n,dir}$, those will be contained in a new set of subsequences S as $s_{n',left}$ or $s_{n',right}$.
4. Now T possibly contains up to two consecutive red nodes at the end of the paths form the root to the leaves, which needs to be repaired. Note that, while repairing, the insertion position of elements $\in S$ have to be updated, if the corresponding nodes are affected by rotations.

- If two nodes have different nearest black ancestors, they can be repaired in parallel. Since at most four nodes can have the same nearest black ancestor, the nodes at the lowest level can be repaired in a constant number of parallel steps.
- This step will be applied successively to the black levels above until T is fully repaired.

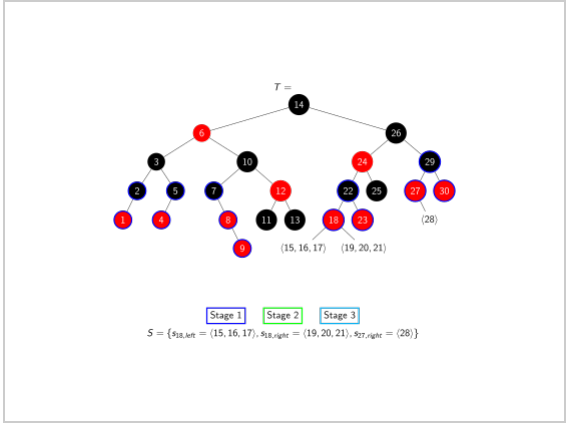
5. The steps 3 to 5 will be repeated on the new subsequences until S is empty. At this point every element $\in \boldsymbol{I}$ has been inserted. Each application of these steps is called a *stage*. Since the length of the subsequences in S is $\in O(|I|)$ and in every stage the subsequences are being cut in half, the number of stages is $\in O(\log |I|)$.
    - Since all stages move up the black levels of the tree, they can be parallelised in a pipeline. Once a stage has finished processing one black level, the next stage is able to move up and continue at that level.



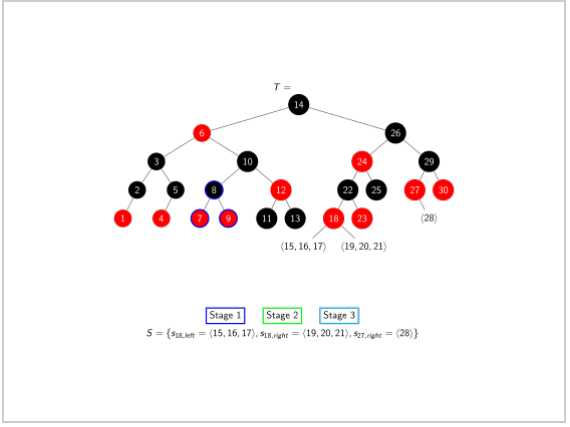
Initial tree



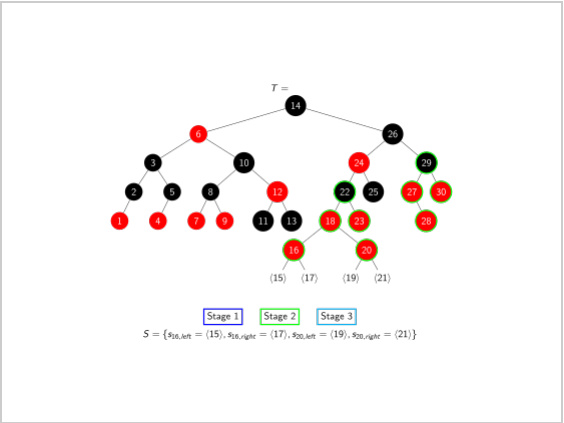
Find insert positions



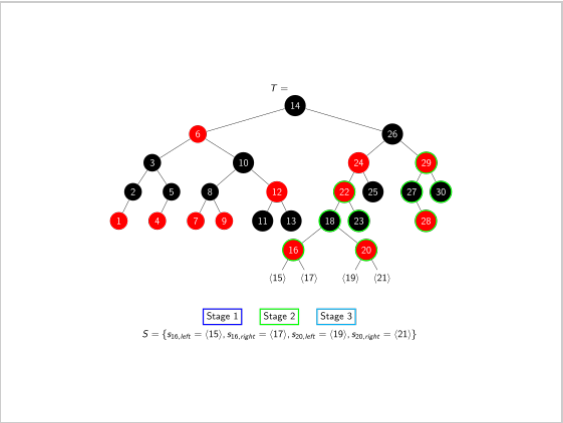
Stage 1 inserts elements



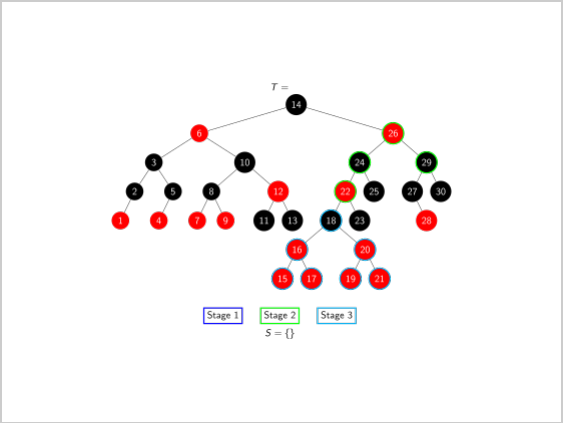
Stage 1 begins to repair nodes



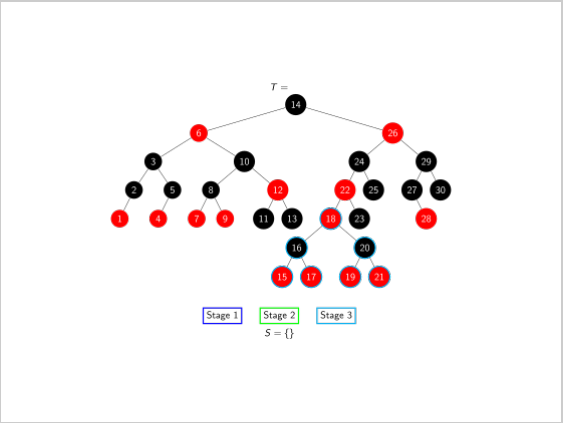
Stage 2 inserts elements



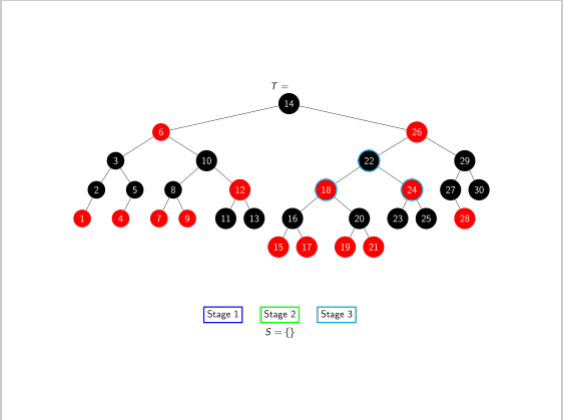
Stage 2 begins to repair nodes



Stage 3 inserts elements



Stage 3 begins to repair nodes



Stage 3 continues to repair nodes

Execution time

Sorting I is not considered in this analysis. Also, $|I|$ is assumed to be smaller than $|T|$, otherwise it would be more efficient to construct the resulting tree from scratch.

T(find insert position)	$\in O(\log \|T\|)$
#stages	$\in O(\log \|I\|)$
T(insert) + T(repair)	$\in O(\log \|T\|)$

T(bulkInsert) with $|I|$ ~ #processors $\in O(\log|I| + 2 \cdot \log|T|)$
$= O(\log|T|)$

Work

W(find insert positions)	$\in O(\|I\| \log\|T\|)$
#insertions, #repairs	$\in O(\|I\|)$
W(insert) + W(repair)	$\in O(\log\|T\|)$
W(bulkInsert)	$\in O(2 \cdot \|I\| \log\|T\|)$
	$= O(\|I\| \log\|T\|)$