Programming Languages

Red-black Trees

Dr Russ Ross
Utah Tech University—Department of Computing

Fall 2026

Balanced search trees

A binary search tree performs well when its height stays small. Ordered input can turn an ordinary search tree into a linked list, making lookup and insertion $O(n)$.

Red-black trees attach one bit of color to each node and maintain enough structure to keep the height $O(\log n)$.

- They are widely used as ordered maps and sets.
- Their balancing rules have a particularly compact functional presentation.
- The implementation gives us a substantial example of algebraic data types and patterns that reach through several levels of a value.

Based on Chris Okasaki, *Purely Functional Data Structures*, §3.3.

Red-black invariants

Every node is either red or black. Empty leaves count as black.

1. A red node never has a red child.

2. Every path from a node to an empty leaf contains the same number of black nodes.

The shortest root-to-leaf path can contain only black nodes. The longest can alternate red and black, so it is at most twice as long.

The binary-search invariant remains unchanged: values in the left subtree are smaller, and values in the right subtree are larger.

The SML data type

An algebraic data type gives each tree one of two shapes:

```
- datatype color = R | B;
- datatype 'a rbtree =
      Lf
    | Br of color * 'a * 'a rbtree * 'a rbtree;
> datatype color = B | R
> datatype 'a rbtree = Br of color * 'a * 'a rbtree * 'a rbtree | Lf
```

`Lf` represents an empty leaf. `Br (color, value, left, right)` represents a branch.

SML values are immutable here. An insertion constructs a new path from the root to the inserted value while reusing untouched subtrees. Both the old root and the new root remain valid.

Lookup in SML

```
- fun lookup Lf _ = false
    | lookup (Br (_, elt, left, right)) x =
        if x < elt then lookup left x
        else if x > elt then lookup right x
        else true;
> val lookup = fn : int rbtree -> int -> bool
```

The pattern determines the tree's shape and binds its components:

- `Lf` is empty.
- `Br (_, elt, left, right)` is a branch. `_` ignores its color.

The recursive call receives only the subtree that can contain `x`.

Insertion in SML

```
- fun insert root x =
    let fun ins Lf = Br (R, x, Lf, Lf)
          | ins (node as Br (color, elt, left, right)) =
              if x < elt then
                balance (color, elt, ins left, right)
              else if x > elt then
                balance (color, elt, left, ins right)
              else node
        val Br (_, elt, left, right) = ins root
    in Br (B, elt, left, right) end;
> val insert = fn : int rbtree -> int -> int rbtree
```

The insertion strategy

Three changes turn ordinary binary-search-tree insertion into red-black insertion:

1. A new node starts red. This preserves the number of black nodes on every path.
2. Each recursive return passes through `balance`, which repairs a red parent with a red child.
3. The final root becomes black. This removes a red-red violation that reaches the top.

The persistent algorithm allocates only nodes on the search path and any nodes reconstructed by balancing. It shares every untouched subtree with the previous version.

What `balance` repairs

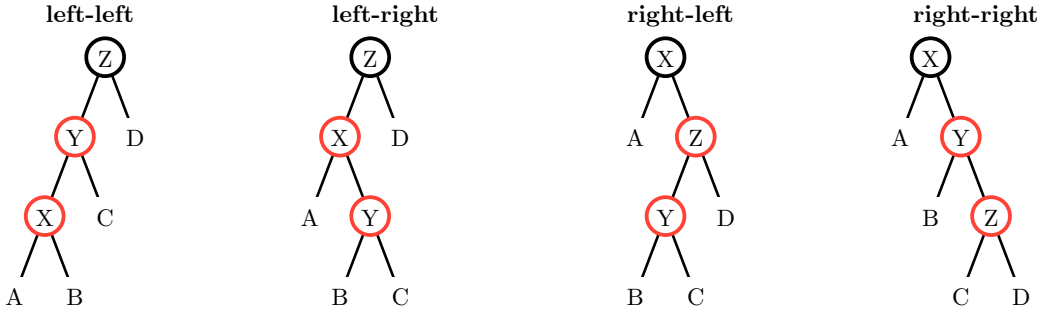
Making a new leaf red preserves the black-height invariant, but a red parent may now have a red child.

`balance` acts like the `Br` constructor except when it receives:

- a black node,
- with a red child,
- that itself has a red child.

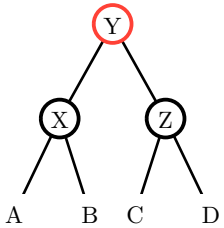
Each repair produces a red node with two black children. It removes the current red-red violation and may create one between the new red root and its parent. Recursive returns let that violation move toward the root.

The four red-red violations



The shapes differ only in where two consecutive red links occur.

One balanced result



Every case preserves the in-order sequence

$$A, X, B, Y, C, Z, D.$$

All four become

```
y (x (A, B), z (C, D))
```

with `y` red and `x` and `z` black.

The four SML patterns

```
- fun balance (B, z, Br (R, y, Br (R, x, a, b), c), d) = ...
    | balance (B, z, Br (R, x, a, Br (R, y, b, c)), d) = ...
    | balance (B, x, a, Br (R, z, Br (R, y, b, c), d)) = ...
    | balance (B, x, a, Br (R, y, b, Br (R, z, c, d))) =

        Br (R, y, Br (B, x, a, b), Br (B, z, c, d))

    | balance body = Br body;
> val balance = fn : color * 'a * 'a rbtree * 'a rbtree -> 'a rbtree
```

Each pattern names all seven in-order components directly. The first three right-hand sides are abbreviated; they construct the same tree as the fourth case.

Why the SML version is compact

The left side of each equation simultaneously:

- verifies that the node, child, and grandchild have the required colors,
- distinguishes one of the four shapes,
- takes the old nodes apart, and
- names `a`, `x`, `b`, `y`, `c`, `z`, and `d` for reconstruction.

The right side describes the desired tree as one expression. Immutability means the program never needs to describe an intermediate state or ask whether some node has another owner.

The Rust data type

The Rust version deliberately changes the storage model. Each node owns its children, and insertion mutates one tree in place.

```
#[derive(Clone, Copy, Eq, PartialEq)]
enum Color { Red, Black }

enum Tree<T> {
    Empty,
    Node {
        color: Color,
        value: T,
        left: Box<Tree<T>>,
        right: Box<Tree<T>>,
    },
}
```

`Box<Tree<T>>` gives each recursive child a known-size pointer representation. Unlike `Rc`, `Box` has one owner, which makes moving and mutating nodes straightforward.

Lookup in Rust

```
impl<T: Ord> Tree<T> {
    fn contains(&self, sought: &T) -> bool {
        match self {
            Tree::Empty => false,
            Tree::Node { value, left, right, .. } => {
                if sought < value {
                    left.contains(sought)
                } else if sought > value {
                    right.contains(sought)
                } else {
                    true
                }
            }
        }
    }
}
```

`&self` is a shared borrow. Lookup can inspect the tree but cannot change or take ownership of it. Matching a borrowed tree binds `value`, `left`, and `right` as references, so the recursive call borrows a child rather than moving it.

Insertion changes the tree

```
impl<T: Ord> Tree<T> {
    fn insert(&mut self, value: T) {
        self.insert_inner(value);
        if let Tree::Node { color, .. } = self {
            *color = Color::Black;
        }
    }

    fn insert_inner(&mut self, value: T) {
        // descend, insert a red leaf, then balance `self`
    }
}
```

The receiver changes from `&self` to `&mut self`: one exclusive borrow authorizes mutation and prevents any simultaneous access to the same tree.

`insert` no longer returns another root. The caller retains ownership of the same `Tree<T>` value while its contents change.

Recursive mutable insertion

```
fn insert_inner(&mut self, new_value: T) {
    match self {
        Tree::Empty => {
            *self = Tree::node(Color::Red, new_value);
            return;
        }
        Tree::Node { value, left, right, .. } => {
            if new_value < *value {
                left.insert_inner(new_value);
            } else if new_value > *value {
                right.insert_inner(new_value);
            } else {
                return;
            }
        }
    }
    self.balance();
}
```

The child borrow ends before `self.balance()`. Rust must see that `left`, `right`, and `value` are no longer in use before it can borrow the whole node mutably again.

`new_value` moves exactly once: into the new leaf or into one recursive call. No `Clone` bound is required.

Rust cannot use the same operation

The SML pattern consumes a conceptual input tuple and constructs a result. Our Rust method instead receives `&mut self` and must rearrange nodes that already exist.

A single deep `match` would borrow `self`, its child, and its grandchild. Those borrows would remain active through the match arm, exactly when a rotation needs an exclusive borrow of the whole subtree.

The Rust design therefore separates two phases:

1. Inspect the tree briefly to classify its shape.
2. End those borrows, then rotate whole `Box` values.

This separation satisfies the ownership rules and keeps values in their existing allocations.

Classifying a borrowed tree

```
fn is_red<T>(tree: &Tree<T>) -> bool {
    matches!(tree, Tree::Node { color: Color::Red, .. })
}

fn has_red_left<T>(tree: &Tree<T>) -> bool {
    matches!(tree, Tree::Node { left, .. } if is_red(left))
}

fn has_red_right<T>(tree: &Tree<T>) -> bool {
    matches!(tree, Tree::Node { right, .. } if is_red(right))
}
```

Each helper takes only `&Tree<T>`, a temporary shared borrow. It reports shape information without retaining references to any node.

`matches!` still uses structural patterns, but `Box` creates an ownership boundary: stable Rust patterns do not recursively unpack arbitrary smart pointers.

Moving a right child into place

```
fn rotate_left<T>(tree: &mut Tree<T>) {
    let mut pivot = {
        let Tree::Node { right, .. } = tree else { unreachable!() };
        std::mem::replace(right, Box::new(Tree::Empty))
    };

    {
        let Tree::Node { right, .. } = tree else { unreachable!() };
        let Tree::Node { left, .. } = pivot.as_mut()
            else { unreachable!() };
        std::mem::swap(right, left);
    }

    std::mem::swap(tree, pivot.as_mut());
    let Tree::Node { left, .. } = tree else { unreachable!() };
    *left = pivot;
}
```

`replace` moves the right child into `pivot` and leaves a valid `Empty` value behind. Two swaps then reconnect the inner subtree and put the old root below the pivot. No `T` value is copied or cloned.

Why the rotation uses scopes

```
let mut pivot = {
    let Tree::Node { right, .. } = tree else { unreachable!() };
    std::mem::replace(right, Box::new(Tree::Empty))
}; // the borrow of `tree.right` ends here

{
    let Tree::Node { right, .. } = tree else { unreachable!() };
    let Tree::Node { left, .. } = pivot.as_mut()
        else { unreachable!() };
    std::mem::swap(right, left);
} // both field borrows end here

std::mem::swap(tree, pivot.as_mut());
```

Rust will not mutably borrow `tree` as a whole while a mutable borrow of one of its fields remains in use. The inner blocks state exactly when each field borrow ends.

The temporary `Empty` is not observable outside the function. Every operation preserves a valid `Tree<T>`, even during rearrangement.

The mirror rotation

```
fn rotate_right<T>(tree: &mut Tree<T>) {
    let mut pivot = {
        let Tree::Node { left, .. } = tree else { unreachable!() };
        std::mem::replace(left, Box::new(Tree::Empty))
    };

    {
        let Tree::Node { left, .. } = tree else { unreachable!() };
        let Tree::Node { right, .. } = pivot.as_mut()
            else { unreachable!() };
        std::mem::swap(left, right);
    }

    std::mem::swap(tree, pivot.as_mut());
    let Tree::Node { right, .. } = tree else { unreachable!() };
    *right = pivot;
}
```

The two functions are mirror images. A double rotation calls one rotation on a child and then the opposite rotation on the whole subtree.

Classifying the four cases

```
fn balance(&mut self) {
    let Tree::Node { color, left, right, .. } = self else {
        return;
    };
    if *color != Color::Black {
        return;
    }

    let left_left   = is_red(left)  && has_red_left(left);
    let left_right  = is_red(left)  && has_red_right(left);
    let right_left  = is_red(right) && has_red_left(right);
    let right_right = is_red(right) && has_red_right(right);

    // `color`, `left`, and `right` are not used below, so their
    // borrows end before the rotations begin.
```

The four Boolean values play the role of the four deep SML patterns. They retain only facts, not references, so the later mutation can borrow `self` exclusively.

Rotating and recoloring

```
        if left_left {
            rotate_right(self);
        } else if left_right {
            let Tree::Node { left, .. } = self else { unreachable!() };
            rotate_left(left);
            rotate_right(self);
        } else if right_left {
            let Tree::Node { right, .. } = self else { unreachable!() };
            rotate_right(right);
            rotate_left(self);
        } else if right_right {
            rotate_left(self);
        } else {
            return;
        }

        blacken_children(self);
    }
```

Single rotations handle the outside cases. Double rotations first turn an inside case into an outside case.

The promoted node was red and stays red. `blacken_children` changes its two children to black, producing the same colors as the SML result.

Recoloring after a rotation

```
fn blacken_children<T>(tree: &mut Tree<T>) {
    let Tree::Node { left, right, .. } = tree else {
        unreachable!();
    };

    for child in [left, right] {
        let Tree::Node { color, .. } = child.as_mut() else {
            unreachable!();
        };
        *color = Color::Black;
    }
}
```

The array temporarily contains two independent mutable references. Rust accepts them because `left` and `right` are disjoint fields obtained from one exclusive borrow.

The `unreachable!` cases depend on facts established by the classifier. A complete library might encode more of those preconditions in helper return types, but that would add machinery to this example.

Two meanings of the same algorithm

Persistent SML

- Deep patterns recognize and decompose each input shape.
- One expression constructs the result.
- Untouched subtrees remain shared.
- Previous roots remain valid.

Mutable Rust

- Short borrows classify the shape.
- Rotations move existing `Box` values.
- Scopes mark when field borrows end.
- The original root changes in place.

Both versions implement the same ordering and color transformation. Their syntax reflects different answers to a semantic question: should insertion produce another tree or modify this tree?