

Algorithms and Data Structures

B6. Red-Black Trees

Gabriele Röger

University of Basel

April 18/24, 2024

Algorithms and Data Structures

April 18/24, 2024 — B6. Red-Black Trees

B6.1 Red-Black Trees

B6.2 Insertion (and Deletion)

B6.3 Summary

B6.1 Red-Black Trees

Motivation

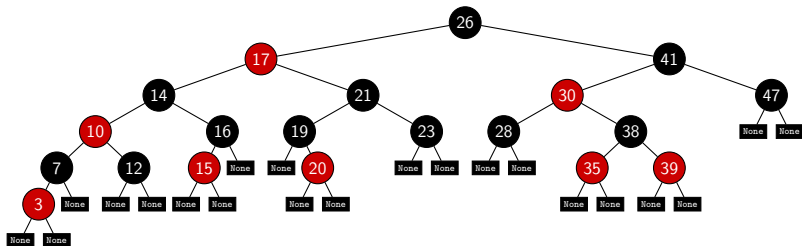
- ▶ **Binary search trees** can support many relevant operations in **linear time in the height of the tree**.
- ▶ **But:** Binary search trees **can degenerate into chains**, in which case the operations take linear time in the number n of elements (no better than with a linked list).
- ▶ **Idea:** Search-trees schemes that are in some form “balanced” and can guarantee running time $O(\log_2 n)$ in the worst case.
 - ▶ **AVL trees:** for every node, the height of the left and right subtree differs by at most 1.
 - ▶ **B-trees:** permit several keys and subtrees per node (e.g. special case: 2-3 tree).
 - ▶ **Red-Black trees:** use node colors to maintain an approximate balancing.
 - ▶ ...

Red-Black Trees: Representation

- ▶ Use one extra bit per node, storing its color, which can be either red or black.
- ▶ Each node now contains attributes `color`, `key`, `left`, `right` and `parent`.

None Leaf Nodes

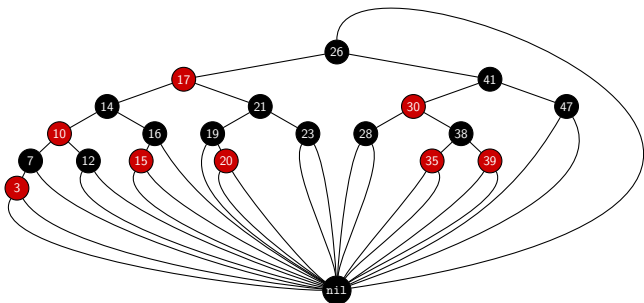
- ▶ Left, right and parent are None if there is no corresponding node.
- ▶ Because it is conceptionally and implementation-wise easier, we will represent them as actual node objects.
- ▶ These are then the leaves of the trees and the nodes holding the entries are inner nodes.



None Leaf Nodes: Sentinel

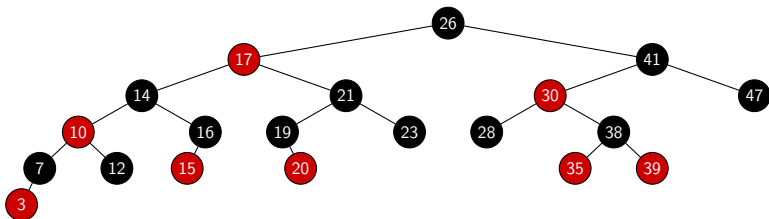
Instead of many leaf nodes, we use a single **sentinel node** `nil`.

- ▶ Implemented like a normal (black) node but used as child of many nodes.
- ▶ The sentinel also serves as parent of the root.
- ▶ Attributes for parent and children can take on arbitrary values.



Graphical Representation

On the slides, we omit the None leaf nodes/sentinel:



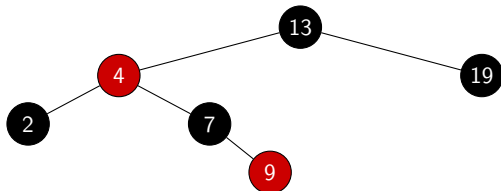
Red-Black Trees

Definition (Red-Black Tree)

A **red-black tree** is a **binary search tree** that satisfies the following **red-black properties**:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

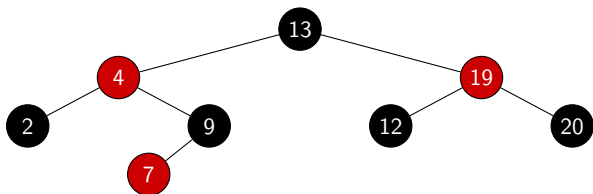
Quiz I: Is this a Red-Black Tree?



Reminder: A **red-black tree** is a **binary search tree** where:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

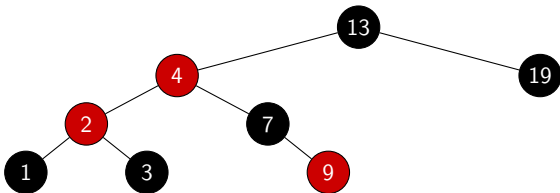
Quiz II: Is this a Red-Black Tree?



Reminder: A **red-black tree** is a **binary search tree** where:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

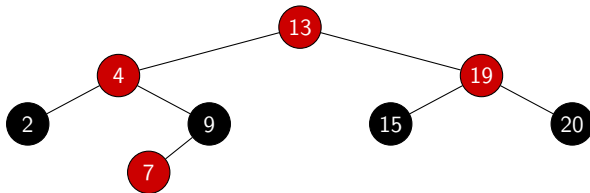
Quiz III: Is this a Red-Black Tree?



Reminder: A **red-black tree** is a **binary search tree** where:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

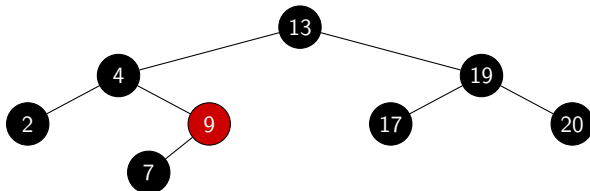
Quiz IV: Is this a Red-Black Tree?



Reminder: A **red-black tree** is a **binary search tree** where:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Quiz V: Is this a Red-Black Tree?



Reminder: A **red-black tree** is a **binary search tree** where:

- 1 Every node is either red or black.
- 2 The root is black.
- 3 Every leaf (None node) is black.
- 4 If a node is red, then both its children are black.
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Height of Red-Black Tree

Theorem

A red-black tree with n inner nodes has height at most $2 \log_2(n + 1)$.

Proof

Let the **black-height** $\text{bh}(x)$ of node x denote the number of black nodes on any simple path from, but not including, x down to a leaf.

We first show by induction on the height of x that the subtree rooted at any node x contains at least $2^{\text{bh}(x)} - 1$ inner nodes. ...

Height of Red-Black Tree

Proof (continued).

Height of x is 0: x is a leaf and the subtree rooted at x contains $2^{\text{bh}(x)} - 1 = 2^0 - 1 = 0$ inner nodes.

Inductive step: x has positive height.

Then x has two children. If a child is black, it contributes 1 to x 's black-height but not to its own. If a child is red, then it contributes to neither x 's black-height nor its own.

Therefore, each child has a black-height of $\text{bh}(x) - 1$ or $\text{bh}(x)$.

Since the height of the child is smaller than the one of x , by the inductive hypothesis the subtree rooted by each child has at least $2^{\text{bh}(x)-1} - 1$ inner nodes.

Thus, the subtree rooted by x contains at least $2(2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ inner nodes.

...

Height of Red-Black Tree

Proof (continued).

We showed that that the subtree rooted at any node x contains at least $2^{\text{bh}(x)} - 1$ inner nodes.

Let h be the height of the tree. Since both children of a red node must be black, at least half of the nodes on any simple path from the root to a leaf (not including the root) must be black.

Thus, the black-height of the root is at least $h/2$ and thus $n > 2^{h/2} - 1$.

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\log_2(n + 1) \geq h/2$, or $h \leq 2 \log_2(n + 1)$. □

Height of Red-Black Tree: Consequence

Theorem

A red-black tree with n inner nodes has height at most $2 \log_2(n + 1)$.

- ▶ The height of a red-black tree is in $O(\log_2 n)$.
- ▶ Red-black trees are binary search trees.
- ▶ On binary search trees, `search(n, k)`, `minimum(n)`, `maximum(n)`, `successor(n)`, `predecessor(n)` can run in time $O(h)$ (cf. Ch. B5).
- ▶ We can use the same implementation for red-black trees, achieving **running time $O(\log_2(n))$** for all these queries.

B6.2 Insertion (and Deletion)

Modifying Red-Black Trees

We cannot simply use the insertion and deletion implementation from binary search trees ([Why not?](#)).



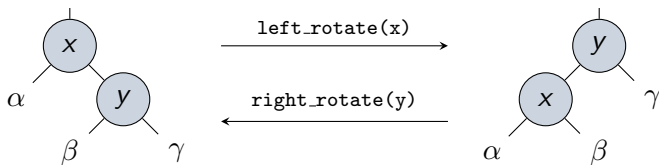
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

Insert (and delete) a number of keys into the red-black tree. What do you observe?



Rotation

- ▶ Inserting and deleting nodes as in binary search trees does not preserve the red-black property.
- ▶ Rotation is an operation that transforms the structure of the tree but preserves the binary-search-tree property.
- ▶ Two variants: left and right rotation.
- ▶ We use them to re-establish the red-black property during an insertion/deletion.

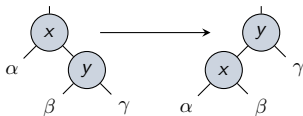


Left-Rotation

```

1 class RedBlackTree:
2     def __init__(self):
3         self.nil = Node(None, None, color=BLACK) # sentinel
4         self.root = self.nil
5
6     def left_rotate(self, x):
7         y = x.right
8         x.right = y.left
9         if y.left is not self.nil:
10            y.left.parent = x
11        y.parent = x.parent
12        if x.parent is self.nil: # x was root node
13            self.root = y
14        elif x is x.parent.left:
15            x.parent.left = y
16        else:
17            x.parent.right = y
18        y.left = x
19        x.parent = y

```



Insertion

```

1  def insert(self, key, value):
2      current = self.root
3      parent = self.nil
4      while current is not self.nil:
5          parent = current
6          if current.key > key:
7              current = current.left
8          else:
9              current = current.right
10     node = Node(key, value, color=RED)
11     node.parent = parent
12     if parent is self.nil: # tree was empty
13         self.root = node
14     elif key < parent.key:
15         parent.left = node
16     else:
17         parent.right = node
18     node.left = self.nil # explicit leaf nodes
19     node.right = self.nil
20     self.fixup(node)

```

Up to this point
pretty much like
insert in binary
search tree.

What red-black
properties can be
violated before the
fixup?

Reminder: Red-Black Trees

Definition (Red-Black Tree)

A red-black tree is a binary search tree that satisfies the following red-black properties:

- 1 Every node is either red or black.
- 2 **The root is black.**
- 3 Every leaf (None node) is black.
- 4 **If a node is red, then both its children are black.**
- 5 For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

What could be violated before the fixup? **Only 2 or 4!**

Property 2 is easy to re-establish: Just color the root black.

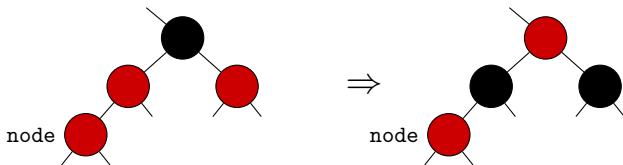
For property 4, distinguish three cases...

Fixup: Case 1

Potential problem: node and its parent are both red (the only violation of red-black property 4).

Case 1: The uncle (parent's sibling) of node is red.

- ▶ The grandparent of node cannot be red (by property 4).
- ▶ **Idea:** Make grandparent red and parent and uncle black.
- ▶ **Afterwards:** Need to fixup grandparent (its parent could be red).

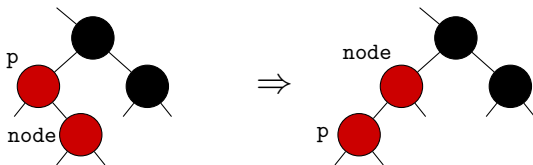


Fixup: Case 2

[Suppose node's parent is a left child.]

Case 2: The uncle of node is black and node is a right child.

- ▶ Perform a left-rotation on the parent.
- ▶ Now the red previous parent is the left child of the red node.
- ▶ This constellation corresponds to case 3 (with the previous parent in the role of the red child node) and is resolved the same way (next slide).

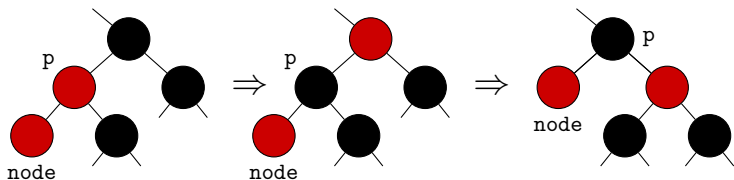


Fixup: Case 3

[Suppose node's parent is a left child.]

Case 3: The uncle of node is black and node is a left child.

- ▶ Make parent black and grandparent red.
- ▶ Afterwards, perform a right-rotation on the grandparent.



Insertion: Fixup

```

1  def fixup(self, node):
2      while node.parent.color == RED:
3          grandparent = node.parent.parent
4          if node.parent is grandparent.left:
5              uncle = grandparent.right
6              if uncle.color == RED:
7                  node.parent.color = BLACK
8                  uncle.color = BLACK
9                  grandparent.color = RED
10                 node = grandparent
11             else:
12                 if node is node.parent.right:
13                     node = node.parent
14                     self.left_rotate(node)
15                 node.parent.color = BLACK
16                 node.parent.parent.color = RED
17                 self.right_rotate(grandparent)
18             else:
19                 ...
20                 # symmetric cases 1-3, where parent is
21                 # not the left child (cf. notebook).
33     self.root.color = BLACK

```

Case 1

Case 2

Case 3

Running time: $O(h)$
(h tree height)

Insertion: Running Time

```
1  def insert(self, key, value):
2      current = self.root
3      parent = self.nil
4      while current is not self.nil:
5          parent = current
6          if current.key > key:
7              current = current.left
8          else:
9              current = current.right
10     node = Node(key, value, color=RED)
11     node.parent = parent
12     if parent is self.nil: # tree was empty
13         self.root = node
14     elif key < parent.key:
15         parent.left = node
16     else:
17         parent.right = node
18     node.left = self.nil # explicit leaf nodes
19     node.right = self.nil
20     self.fixup(node)
```

Running time:
 $O(h)$
(h tree height)

Deletion

- ▶ Deleting a node from a red-black tree is more complicated than inserting a node.
- ▶ We do not cover the details in this course.
- ▶ Deletion from a tree with n nodes is possible in time $O(\log_2 n)$.

B6.3 Summary

Summary

- ▶ **Red-black trees** are a special kind of **binary search trees** that are **approximately balanced**.
- ▶ The **height** of a red-black tree with n nodes is $O(\log_2 n)$.
- ▶ Consequently, the **query operations** only take **logarithmic time** in the size of the tree.
- ▶ The same is true for **insertion** and **deletion**.