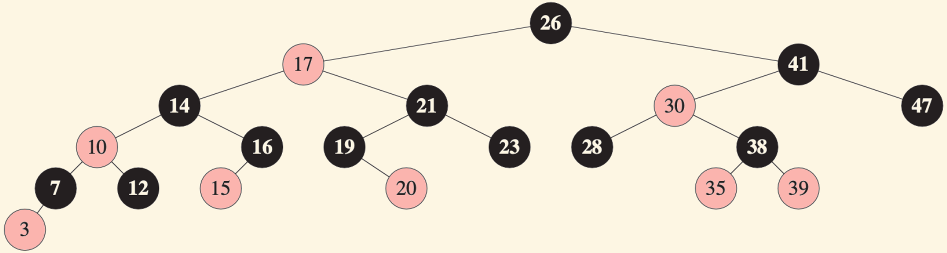
Introduction to Red-Black Trees

Michael T. Gastner (23 March 2023)



Disclaimer: These slides are based on and occasionally quote from ‘Introduction to Algorithms’ (3rd ed.) by Cormen et al. (2009), MIT Press.

Introduction

Reminder: Definition of Binary Search Tree

In this lesson, we continue a theme from the first half of this course, when we studied **binary search trees**. Remember that every node x in a binary search tree has the following attributes:

- $x.key$: the value by which we sort the nodes (e.g., numerically or alphabetically).
- $x.p$: parent of x.
- $x.left$ and $x.right$: children of x.

Binary search trees are binary trees that satisfy the following property:

Binary-search-tree property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

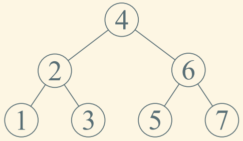
Introduction

Reminder: Definition of the Height of a Node in a Tree

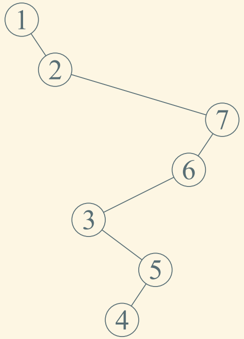
Remember that the **height of a node** in a tree is the number of edges on the longest simple downward path from the node to a leaf. The **height of a rooted tree** h is the height of its root.

The height of a binary tree with n nodes ranges from $\lfloor \log_2 n \rfloor$ for balanced trees to $n - 1$ if each internal node has exactly one child.

balanced
$h = 2$



unbalanced
$h = 6$



Introduction

A Red-Black Tree is a Special Type of Binary Search Tree

We learned earlier in this course that basic dynamic-set operations on binary search trees, such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE, require $O(h)$ time.

A **red-black tree** is a special type of binary search tree that guarantees $h = O(\log n)$, making the basic dynamic-set operations fast.

In addition to the usual node attributes key, p, $left$, and $right$, nodes in a red-black tree have a $color$ attribute. This attribute plays an important role in keeping the tree balanced as nodes are inserted and deleted.

Learning Objectives

By the End of This Lesson, You Should Be Able to . . .

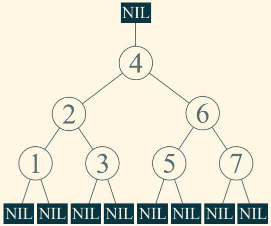
- State the defining properties of a red-black tree.
- State that the height of a red-black tree grows as $O(\log n)$.
- Explain why the logarithmic growth of the height makes red-black trees good binary search trees.
- Apply the RB-INSERT procedure to given data.

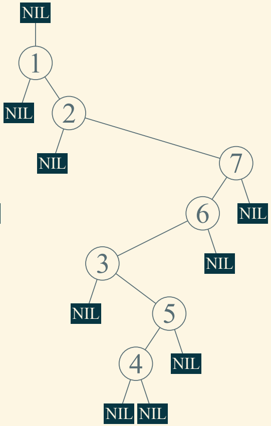
Treating NIL as a Leaf

Departure from Our Previous Interpretation of External and Internal Nodes

If a child or parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL.

In a red-black tree, we shall regard these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree.





Properties of Red-Black Trees

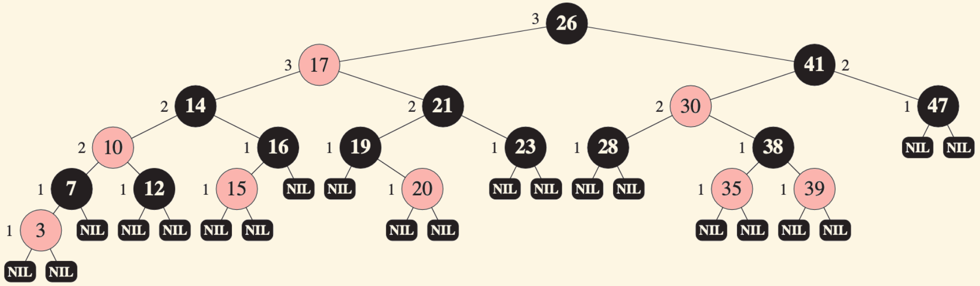
Constraints on the Colors

A red-black tree must satisfy the following **red-black properties:**

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) 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.

Illustration of a Red-Black Tree

Including the NIL Leaves

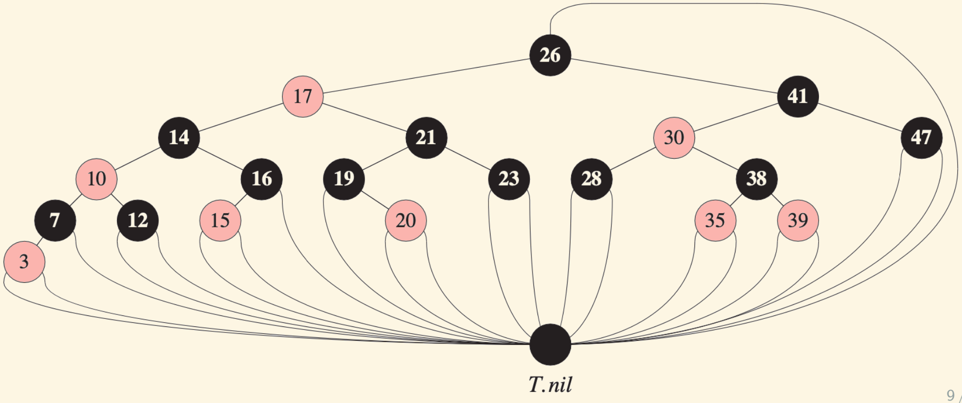


We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height of the node**, denoted by $\mathrm{bh}(x)$. In the illustration above, each non-NIL node is marked with its black-height.

We define the **black-height of a red-black tree** to be the black-height of its root.

Sentinel
Special Node to Represent NIL

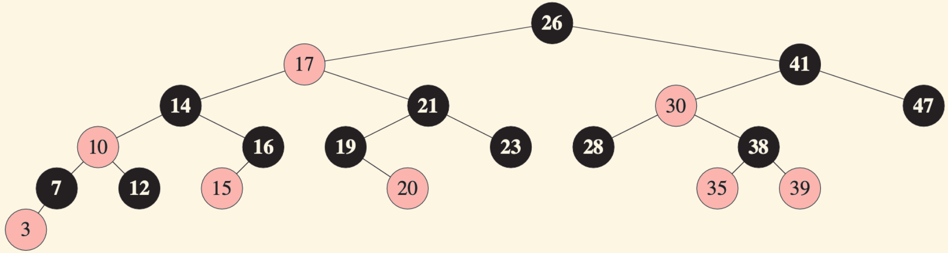
For convenience, we use a so-called **sentinel** to represent all NIL values in a red-black tree. The sentinel $T.nil$ has the same attributes as an ordinary node in the tree. Its $color$ attribute is BLACK, and its other attributes—p, $left$, $right$, and key—can take on any arbitrary values.



Only Drawing Internal Nodes

Omitting Leaves in Drawings

We generally confine our interest to the internal nodes of a red-black tree, since they hold the key values. In the remainder of this lesson, we omit the leaves when we draw red-black trees.



Why are Red-Black Trees Good Binary Search Trees?

Height of a Red-Black Tree Increases at Most Logarithmically

Lemma:

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

Outline of the proof:

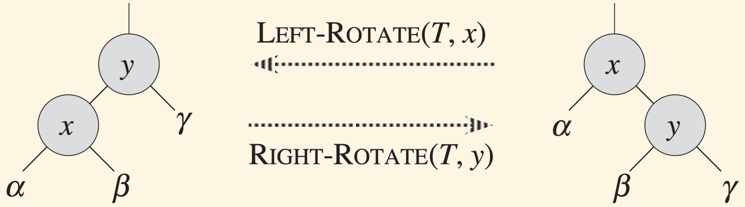
Induction on the height of a node x shows that the subtree rooted at x contains at least $2^{\mathrm{bh}(x)} - 1$ internal nodes. To complete the proof, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus, $n \geq 2^{h/2} - 1$. Moving the 1 to the left-hand side and taking logarithms on both sides yields $\log_2(n+1) \geq h/2$, or $h \leq 2\log_2(n+1)$.

For details, see Section 13.1 in Cormen et al. (2009).

Rotations

Restructuring Trees Without Violating the Binary-Search-Tree Property

When a node is inserted into or deleted from a tree, we may violate the red-black properties. To restore them, we may need to perform a **rotation**. The figure below shows how the operations LEFT-ROTATE(T, x) and RIGHT-ROTATE(T, y) convert one binary search tree into another.

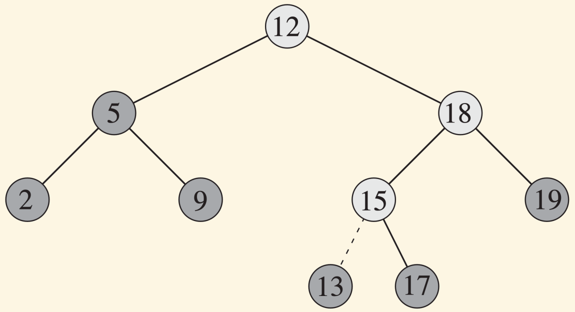


Because we already encountered rotation during our lessons on AVL trees, we will not delve into the algorithmic details in this lesson.

Insertion into a General Binary-Search Tree

Reminder

The figure below shows how to insert an item with key 13 into a general binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.



Insertion into a Red-Black Tree

Comparison of Pseudocode with Differences Highlighted

```
TREE-INSERT($T, z$)
1   $y = \text{NIL}$
2   $x = T.root$
3   while $x \neq \text{NIL}$
4       $y = x$
5       if $z.key < x.key$
6           $x = x.left$
7       else $x = x.right$
8   $z.p = y$
9   if $y == \text{NIL}$
10      $T.root = z$ // tree $T$ was empty
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  // assumes $z.left == z..right == \text{NIL}$
```

```
RB-INSERT($T, z$)
1   $y = T.nil$
2   $x = T.root$
3   while $x \neq T.nil$
4       $y = x$
5       if $z.key < x.key$
6           $x = x.left$
7       else $x = x.right$
8   $z.p = y$
9   if $y == T.nil$
10      $T.root = z$
11  elseif $z.key < y.key$
12      $y.left = z$
13  else $y.right = z$
14  $z.left = T.nil$
15  $z.right = T.nil$
16  $z.color = \text{RED}$
17  RB-INSERT-FIXUP($T, z$)
```

RB-INSERT-FIXUP

Restore Red-Black Properties

```
RB-INSERT-FIXUP$(T, z)$
 1  while $z.p.color$ == RED
 2      if $z.p == z.p.p.left$
 3          $y = z.p.p.right$
 4          if $y.color$ == RED
 5              $z.p.color$ = BLACK            // case 1: $z$'s uncle $y$ is red.
 6              $y.color$ = BLACK              // case 1
 7              $z.p.p.color$ = RED            // case 1
 8              $z = z.p.p$                    // case 1
 9          else if $z == z.p.right$
10                  $z = z.p$                  // case 2: $z$'s uncle $y$ is black and ...
11                  LEFT-ROTATE$(T, z)$        // case 2: ... $z$ is a right child.
12              $z.p.color$ = BLACK            // case 3: $z$'s uncle $y$ is black and ...
13              $z.p.p.color$ = RED            // case 3: ... $z$ is a left child.
14              RIGHT-ROTATE$(T, z.p.p)$       // case 3
15      else (same as then clause with "right" and "left" exchanged)
16  $T.root.color$ = BLACK
```

Loop Invariant
Node z is Always Red

We know from line 16 and 17 of $\textsc{RB-Insert}$ that z is red whenever $\textsc{RB-Insert-Fixup}$ is called.

Inside the **while** loop, a color is assigned to z only on lines 8 and 10.

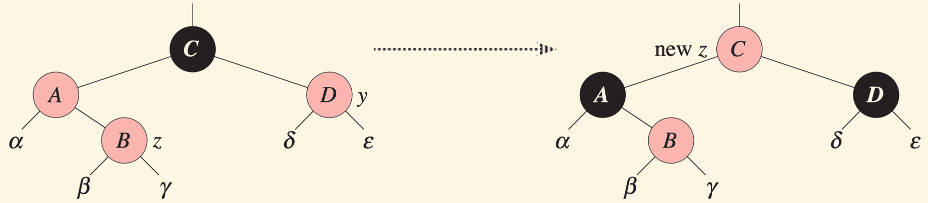
On line 8, z receives the color of its grandparent, which is guaranteed to be red because of the assignment $z.p.p.color = \textsc{red}$ on line 7.

On line 10, z receives the color of its parent $z.p.color$. We know that the parent's color is red because of the condition of the **while** loop in line 1.

In summary, node z must be always red.

Case 1: z's Uncle y Is Red

Illustration of the Subcase When z Is a Right Child

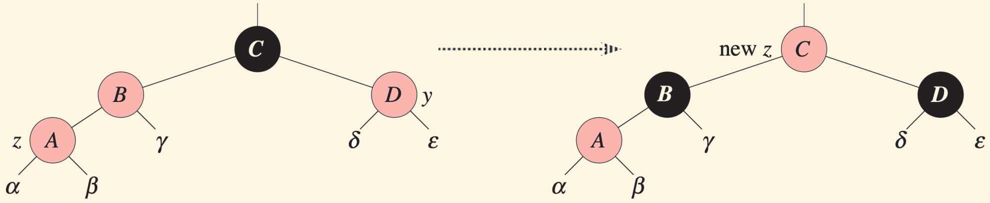


Property 4 is violated, since z and its parent $z.p$ are both red. Each of the subtrees α, β, γ, δ, and ϵ has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward simple paths from a node to a leaf have the same number of blacks.

The **while** loop continues with node z's grandparent $z.p.p$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

Case 1: z's Uncle y Is Red

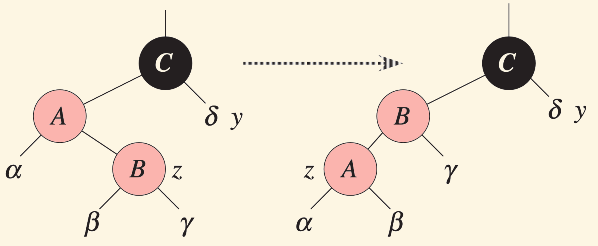
Illustration of the Subcase When z Is a Left Child



If z is a left child, we take essentially the same action as on the previous page: we change the colors of the grandparent and its children to restore property 4.

Case 2: z's Uncle y Is Black and z Is a Right Child

Case 2 Gets Converted into Case 3

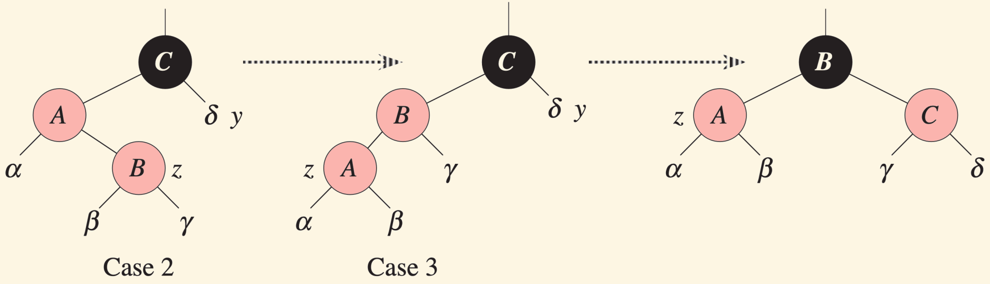


Case 2 Case 3

Property 4 is violated because z and $z.p$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. We transform case 2 into case 3 by a left rotation, which preserves property 5: all downward simple paths from a node to a leaf have the same number of black nodes.

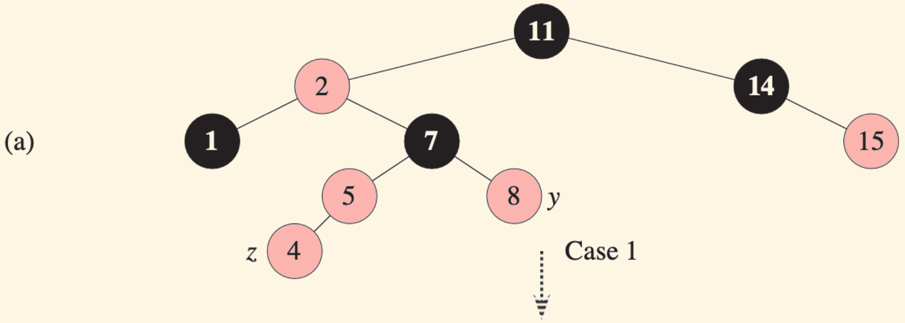
Case 3: z's Uncle y Is Black and z Is a Left Child

Restoring All Red-Black Properties



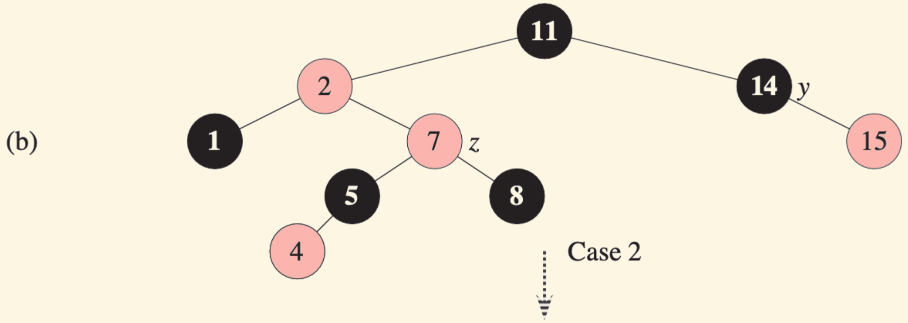
Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates because property 4 is satisfied.

Example: Complete Run of RB-INSERT-FIXUP
Part (a)



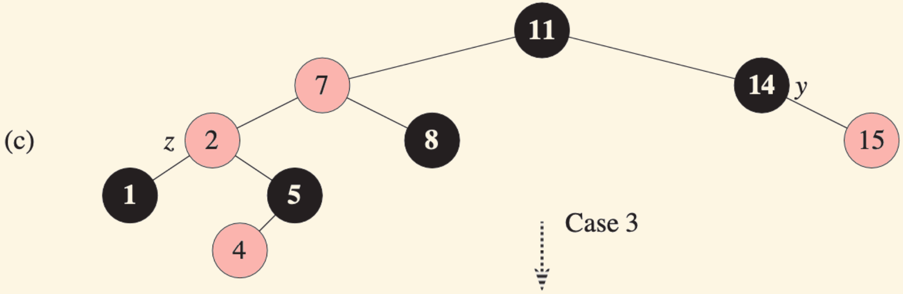
(a) A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies.

Example: Complete Run of RB-INSERT-FIXUP
Part (b)



We recolor nodes and move the pointer z up the tree, resulting in the tree shown in (b). Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies.

Example: Complete Run of RB-INSERT-FIXUP
Part (c)

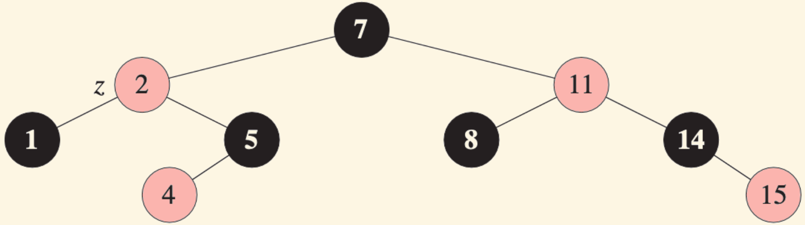


We perform a left rotation, and the tree that results is shown in (c). Now, z is the left child of its parent, and case 3 applies.

Example: Complete Run of RB-INSERT-FIXUP

Part (d). Restoring the Red-Black Properties.

(d)



Recoloring and right rotation yield the tree in (d), which is a legal red-black tree.

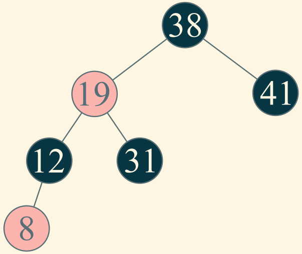
Exercise

Building a Red-Black Tree

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

Solution to Exercise

Final Red-Black Tree


38
19
41
12
31
8