Balanced Trees (**Red**-**Black** Trees)
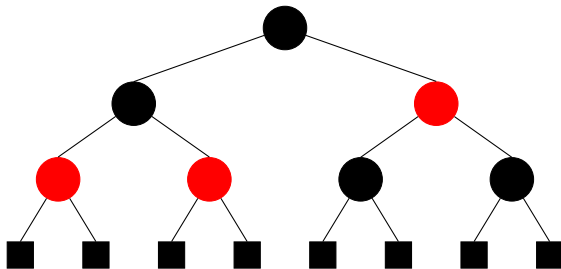
Warren Kim

Quick Definition

> **Definition**
>
> *A **red**-**black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ performance.*

Applications

Red-Black Trees have a variety of applications. Some include:

$\rightarrow$ Linux CPU scheduler (Completely Fair Scheduler)

$\rightarrow$ Linux Virtual Memory Areas (VMA)

$\rightarrow$ STL Data Structures (e.g. C++'s `std::map`, Java's `HashMap`)

$\rightarrow$ Graph algorithm optimizations (for AI/ML)[e.g. K-mean clustering]

$\rightarrow$ Priority Queues (e.g. Range Queries)

Motivation

Why do we want balanced binary trees?

$\rightarrow$ Raw binary search tree performance is highly dependant on input order.

$\rightarrow$ We want to ensure $\mathcal{O}(\log n)$ performance.
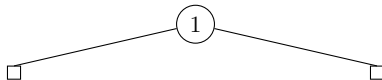
Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:
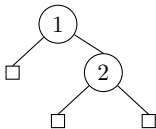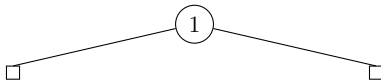
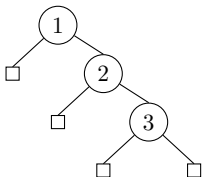$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

Example
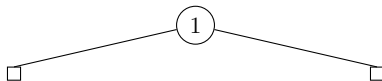
Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:
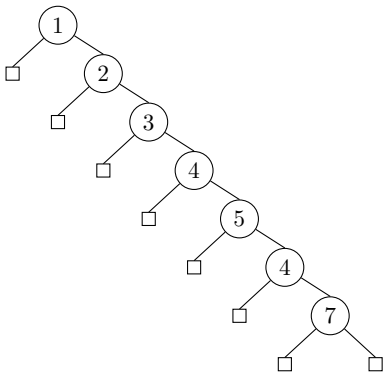
$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

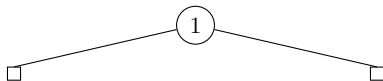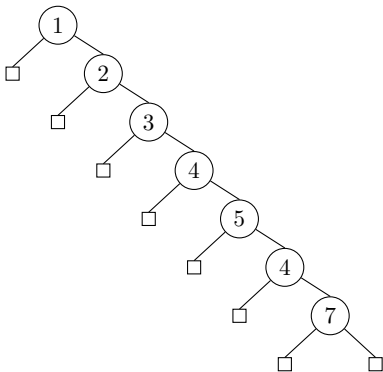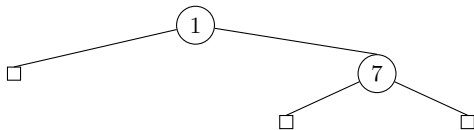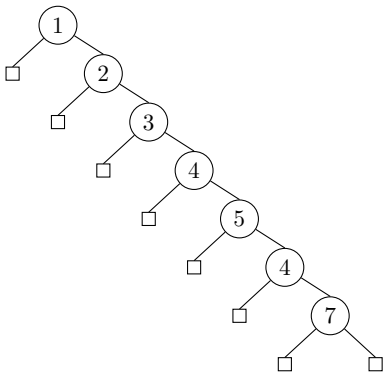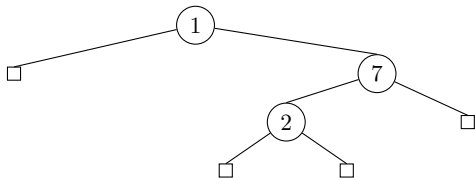# Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$\{1, 2, 3, 4, 5, 6, 7\}$ $\{1, 7, 2, 6, 3, 5, 4\}$

Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:
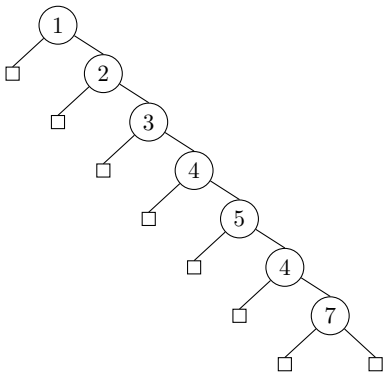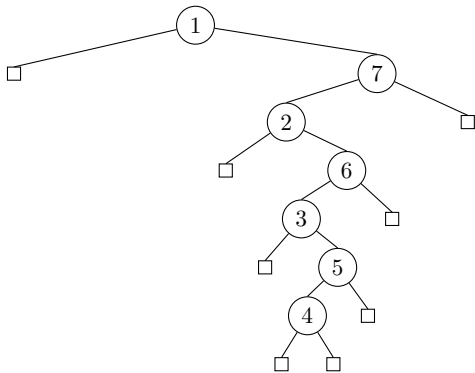
$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

Example

Suppose we have the input set $\{1, 2, 3, 4, 5, 6, 7\}$ and consider two input orders:

$$\{1, 2, 3, 4, 5, 6, 7\} \qquad\qquad \{1, 7, 2, 6, 3, 5, 4\}$$

Intuition

How do we balance binary trees?

We can *dynamically* balance the tree!

$\rightarrow$ We can add metadata[1] to our `Node` struct.

$\rightarrow$ We can define a set of conditions that enforce balance.

---

[1]Metadata: Additional member variables

Definition and Properties

> **Definition**
>
> A **red**-**black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:

(i) *Color:* Every node is either **red** or **black**

```cpp
enum Color { RED, BLACK };

struct Node {
    Color color;
    Node *left;
    Node *right;
    Node *parent;

    int data;
};
```
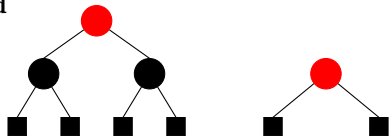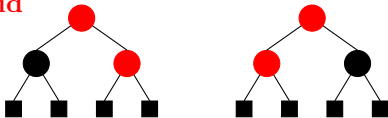
Definition and Properties

> ## Definition
>
> *A **red**-**black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:*

(i) *Color:* Every node is either *red* or **black**

(ii) *Internal:* A **red** node does not have a **red** child
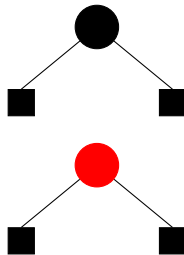
Valid

Invalid

Definition and Properties

Definition

*A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:*

(i) *Color:* Every node is either **red** or **black**

(ii) *Internal:* A **red** node does not have a **red** child
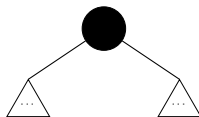
(iii) *External:* All nil nodes are **black**

Definition and Properties

> ### Definition
>
> A **red**-**black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:

(i) *Color:* Every node is either **red** or **black**

(ii) *Internal:* A **red** node does not have a **red** child

(iii) *External:* All nil nodes are **black**
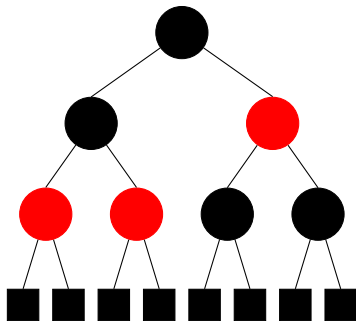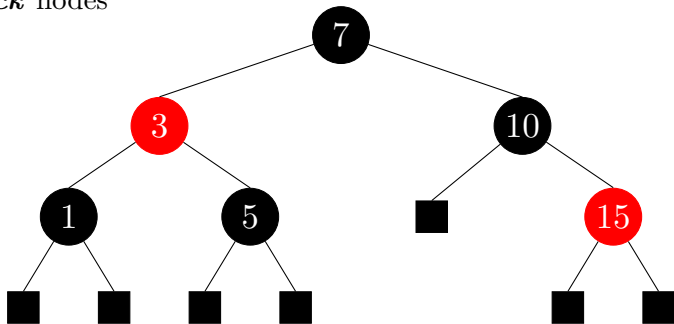
(iv) *Root:* The root node is always **black**

Definition and Properties

> ### Definition
>
> *A **red**-**black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:*

(i) *Color:* Every node is either **red** or **black**

(ii) *Internal:* A **red** node does not have a **red** child

(iii) *External:* All nil nodes are **black**

(iv) *Root:* The root node is always **black**

(v) *Depth:* Every path from the root to *any* nil node passes through the same number of **black** nodes

Depth Property

(v) Depth: Every path from the root to *any* nil node passes through the same number of **black** nodes

Valid

Depth Property

(v) Depth: Every path from the root to *any* nil node passes through the same number of **black** nodes
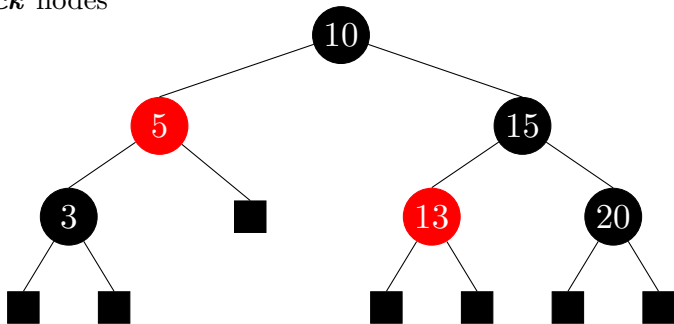
Invalid

Depth Property

(v) Depth: Every path from the root to *any* nil node passes through the same number of **black** nodes

Valid

Definition and Properties

> **Definition**
>
> *A **red-black** tree is a type of **self-balancing** binary search tree that guarantees $\mathcal{O}(\log n)$ search, insertion, and deletion operations with the following properties:*

(i) *Color:* Every node is either **red** or **black**

(ii) *Internal:* A **red** node does not have a **red** child

(iii) *External:* All nil nodes are **black**

(iv) *Root:* The root node is always **black**

(v) *Depth:* Every path from the root to *any* nil node passes through the same number of **black** nodes

Insertion

Suppose we have a node z to insert into our **red-black** tree. Then,

Insertion

Suppose we have a node z to insert into our **red-black** tree. Then,

(i) Like a BST, insert z.

Insertion

Suppose we have a node z to insert into our **_red-black_** tree. Then,

(i) Like a BST, insert z.

(ii) Color z **red**.

Insertion

Suppose we have a node z to insert into our **red-black** tree. Then,

(i) Like a BST, insert z.

(ii) Color z **red**.

(iii) Fix double **red** violations, if any.

Insertion

Suppose we have a node z to insert into our **red-*black*** tree. Then,

(i) Like a BST, insert z.

(ii) Color z **red**.

(iii) Fix double **red** violations, if any.

(iv) Recursively fix violations upward.

Double Red Violations

Recall *Property (ii): A **red** node does not have a **red** child.*

When we insert our node z (**red** by definition), its parent may be **red**. Below are examples of such cases.
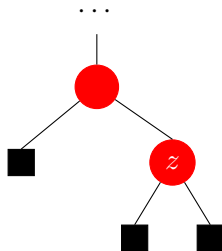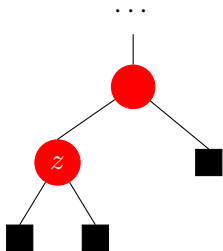
Fixing Double Red Violations

Terminology

With respect to inserted node z,

   $\rightarrow$ *Parent* (p): z's direct parent

   $\rightarrow$ *Uncle* (u): p's sibling

   $\rightarrow$ *Grandparent* (g): p's parent
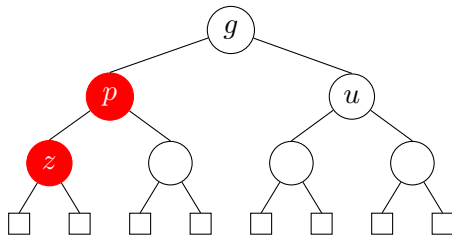
Fixing Double Red Violations

Terminology

With respect to inserted node z,

→ *Parent* (p): z's direct parent

→ *Uncle* (u): p's sibling

→ *Grandparent* (g): p's parent
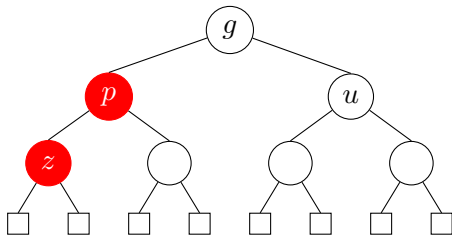
There are two cases:

Fixing Double Red Violations

Terminology

With respect to inserted node z,

→ *Parent* (p): z's direct parent

→ *Uncle* (u): p's sibling

→ *Grandparent* (g): p's parent
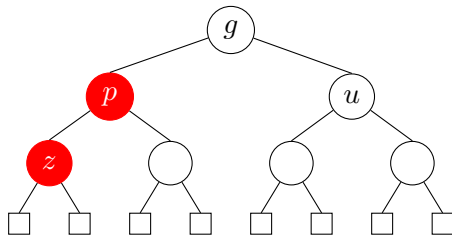
There are two cases:

(i) **Recolor:** If both the *parent* and *uncle* are ***red***, perform a *recolor*.

Fixing Double Red Violations

Terminology

With respect to inserted node z,
→ *Parent* (p): z's direct parent
→ *Uncle* (u): p's sibling
→ *Grandparent* (g): p's parent
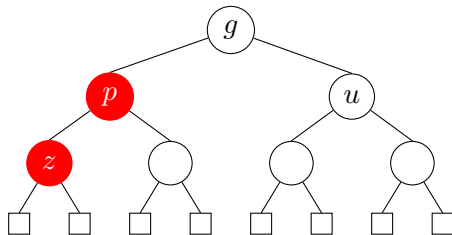
There are two cases:
(i) **Recolor:** If both the *parent* and *uncle* are **red**, perform a *recolor*.
(ii) **Restructure:** If the *parent* is **red** but the *uncle* is **black**, perform a *tri-node restructure*.

Recolor
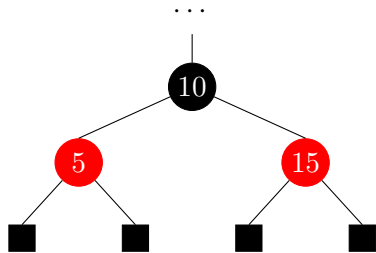
Recolor
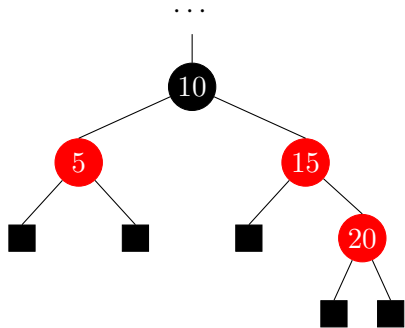
Recolor
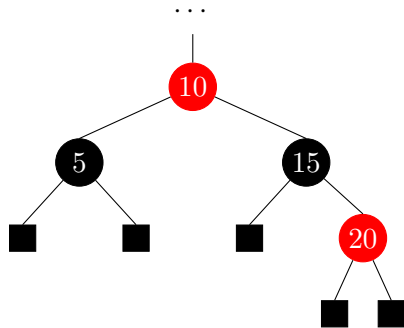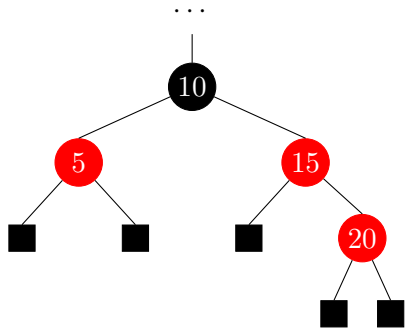
Recolor (Recursive)
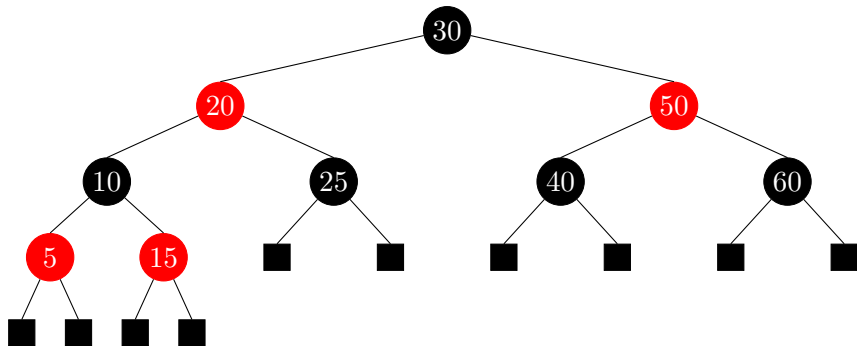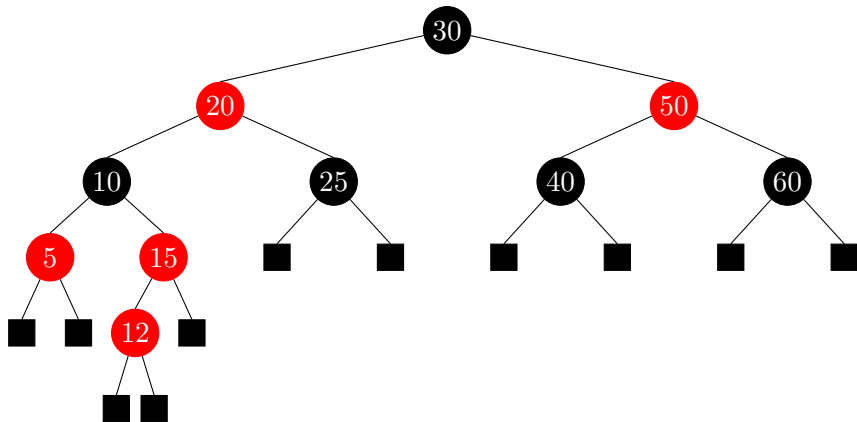
Recolor (Recursive)
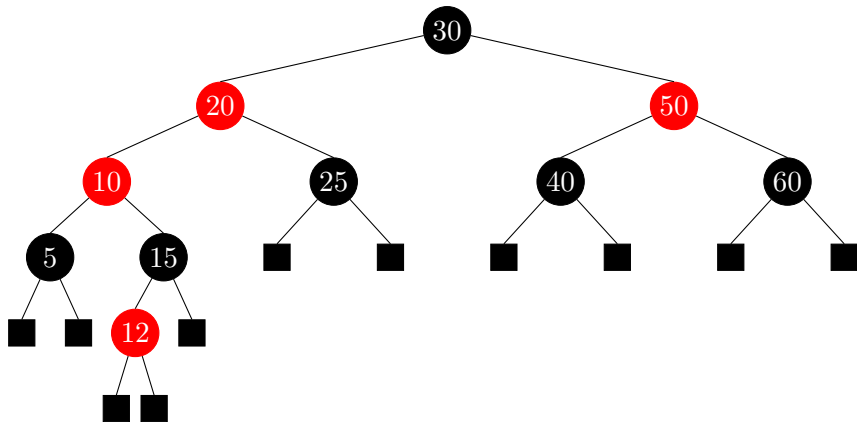
Recolor (Recursive)
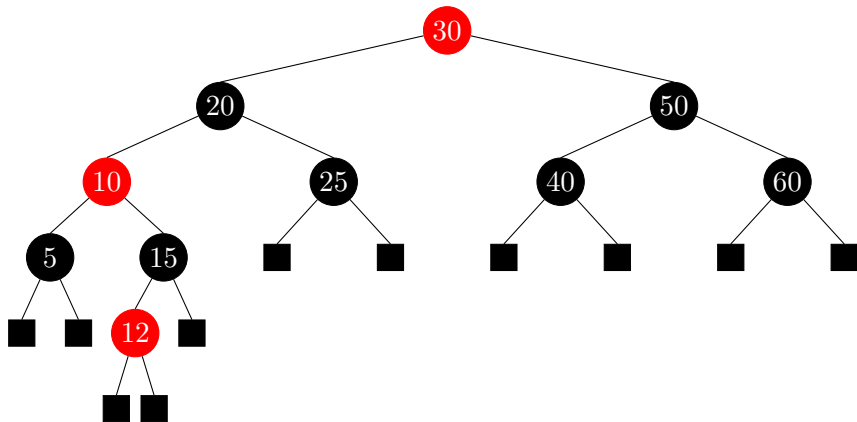
Recolor (Recursive)

Recolor (Recursive)
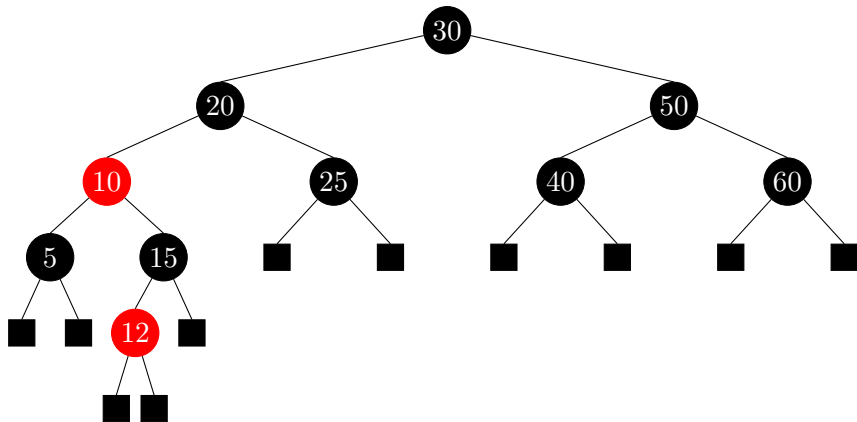
Tri-Node Restructure

There are four cases:
- *(i)* Left-Left
- *(ii)* Right-Right
- *(iii)* Left-Right
- *(iv)* Right-Left

Case: Left-Left (Simple)
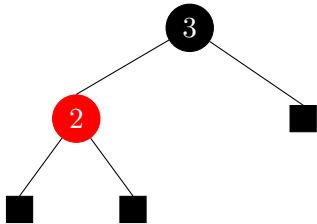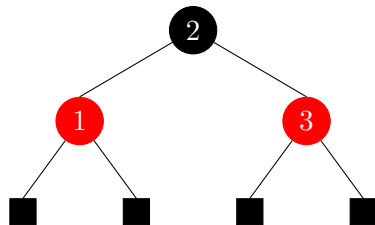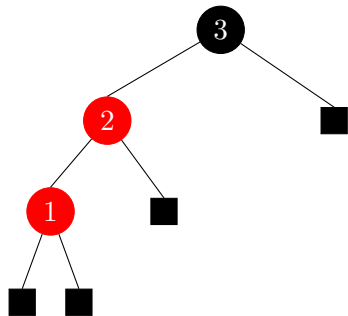
Case: Left-Left (Simple)

Case: Left-Left (Simple)

Case: Left-Left (General)
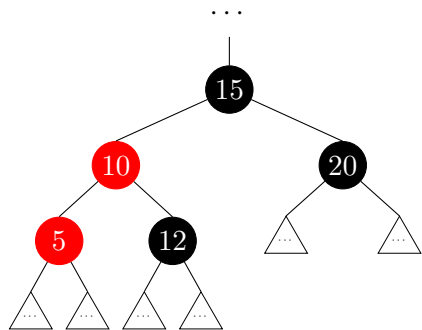
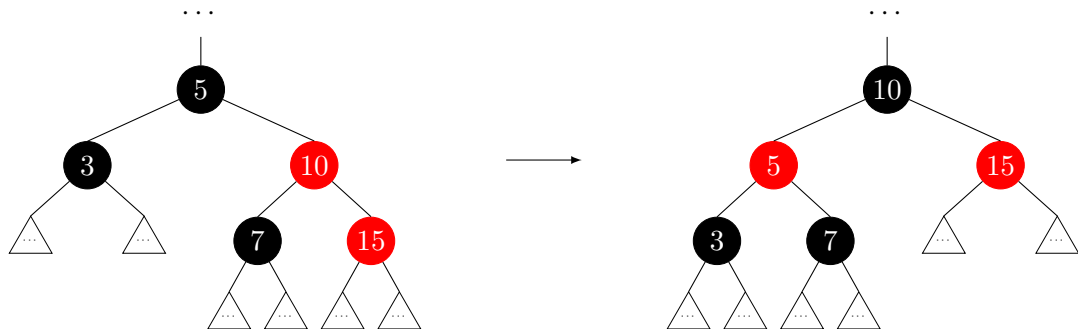Here, ⟨triangle⟩ represents a subtree and $\cdots$ represents the rest of the tree.

Case: Left-Left (General)
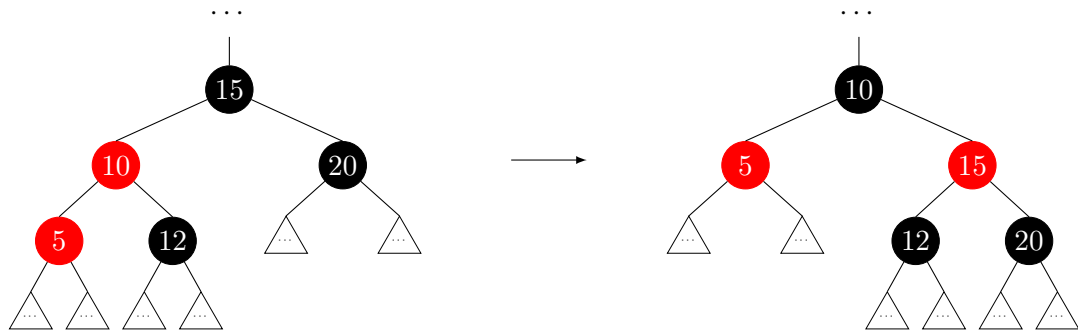
Here, △ represents a subtree and ⋯ represents the rest of the tree.

Case: Right-Right (Simple)
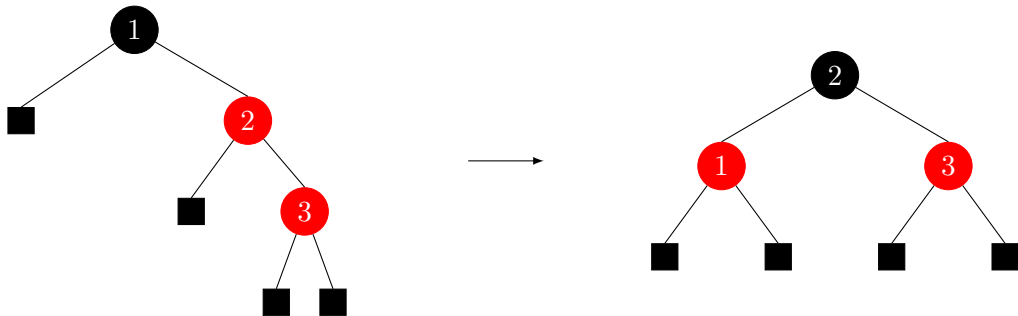
Case: Right-Right (General)

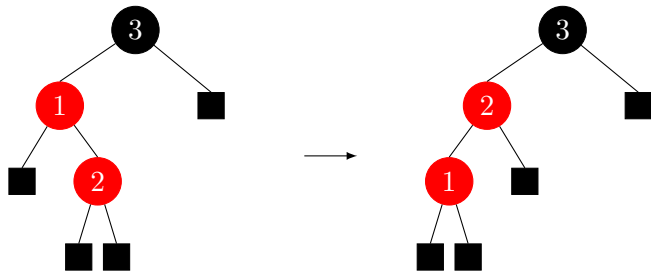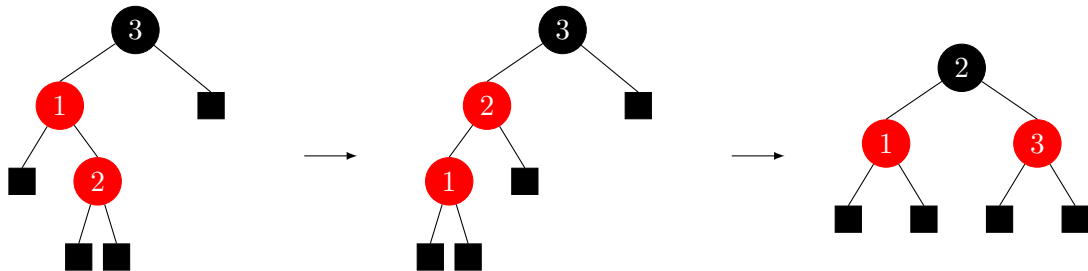Here, $\triangle$ represents a subtree and $\cdots$ represents the rest of the tree.

Case: Left-Right (Simple)
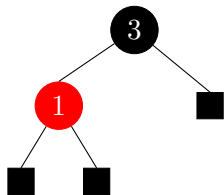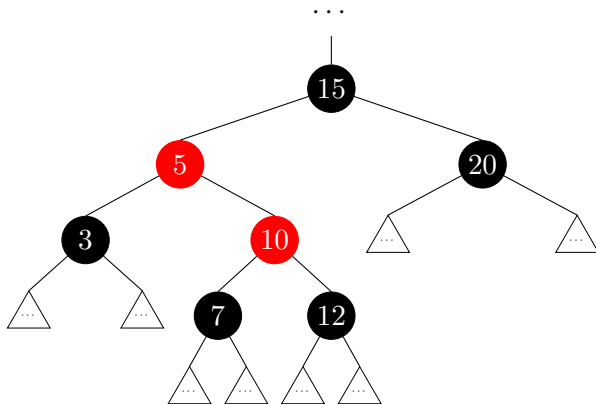
Case: Left-Right (Simple)
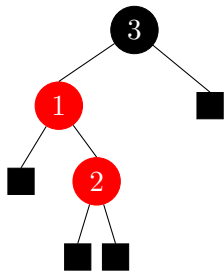
Case: Left-Right (Simple)
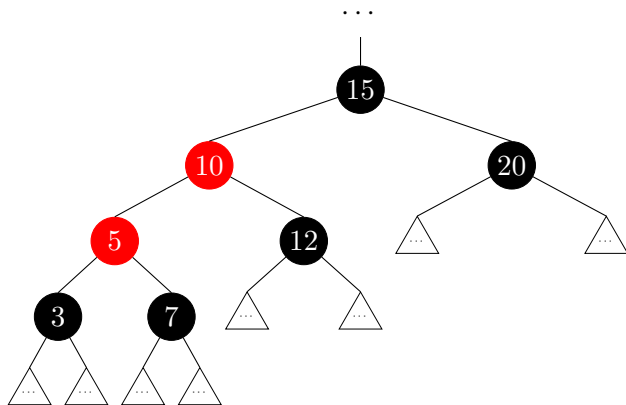
Case: Left-Right (Simple)

Case: Left-Right (General)

Step 1

Here, △ represents a subtree and $\cdots$ represents the rest of the tree.
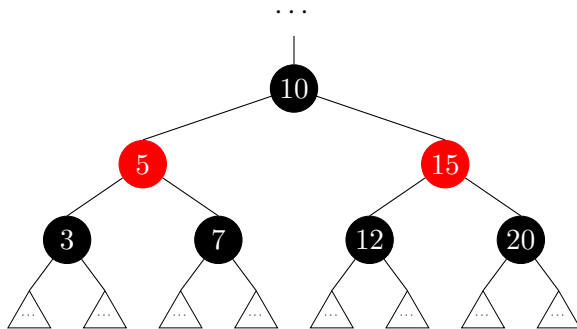
Case: Left-Right (General)

Step 2

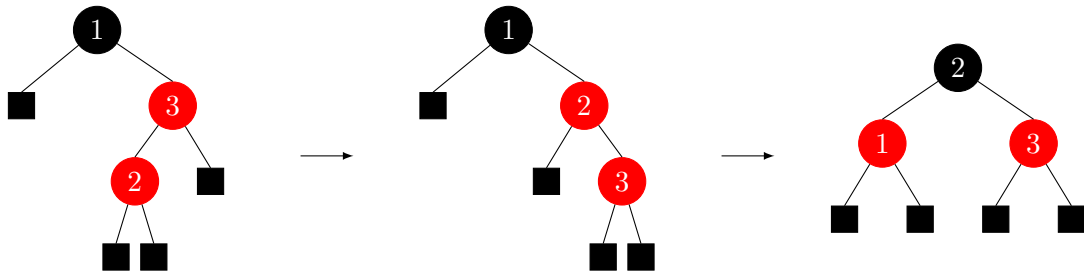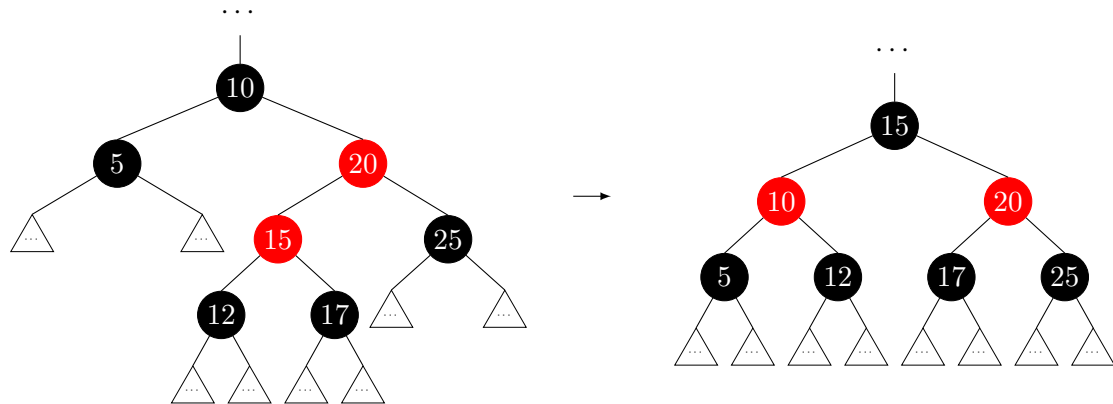Here, △ represents a subtree and $\cdots$ represents the rest of the tree.

Case: Left-Right (General)

Step 3

Here, △ represents a subtree and ⋯ represents the rest of the tree.

Case: Right-Left (Simple)

Case: Right-Left (General)

Here, △ represents a subtree and ⋯ represents the rest of the tree.

Time and Space Complexities

$\rightarrow$ **_Insertion_**: $\mathcal{O}(\log n)$
$\rightarrow$ **_Deletion_**: $\mathcal{O}(\log n)$
$\rightarrow$ **_Search_**: $\mathcal{O}(\log n)$
$\rightarrow$ **_Space_**: $\mathcal{O}(n)$

End

Thank you!

Appendix

Below are slides that didn't make the cut.

Corollaries

> **Proposition**
>
> *If a node z has exactly one child, c, then* **(a)** *c is **red**,* **(b)** *z is **black**, and* **(c)** *c has no children.*

Proof. Suppose we have a valid **red-black** tree. Consider a node z with exactly one child. Without loss of generality, choose z's left node to be the child and call it c.

(a) z passes through no **black** nodes on the right side by assumption. If c were **black**, then z would pass through 1 **black** node, a contradiction since this violates the *depth property*.

(b) By **(a)**, z's child is **red** and by the *internal property*, z is **black**.

(c) Since z passes through no **black** nodes on the right side by assumption, z cannot pass through any **black** nodes on the left side by the *the depth property*. Then, since c is **red** by **(a)**, c has only nil nodes $\qquad\square$

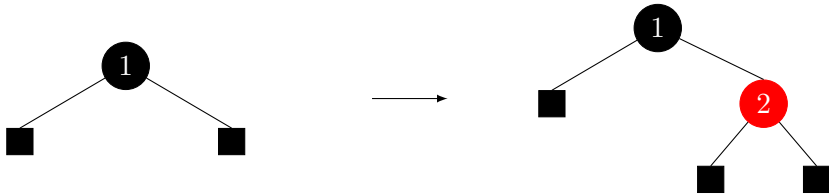# Height of a Red-Black Tree

> **Theorem**
>
> *A **red-black** tree with n nodes has a height h that is $\mathcal{O}(\log n)$.*

Proof. Suppose we have a **red-black** tree with n nodes and height h. Let b be the number of **black** nodes on the shortest path from root to any leaf. In the worst case, the longest path alternates between **red** and **black** nodes and thus has a height of $2b$. Then, h is bounded above by $2b$; that is, $h \leq 2b$. There are $2^b - 1 \leq n$ nodes in this tree. Solving for b, we get $b \leq \log(n+1)$. Substituting b, we get $b \leq \log(n+1) \leq h \leq 2b \leq 2\log(n+1)$ so h is bounded below by $\log(n+1)$ and above by $2\log(n+1)$; that is, $\log(n+1) \leq h \leq 2\log(n+1)$. So, h is $\mathcal{O}(\log n)$. $\qquad\square$

Red-Black Tree: Example

Red-Black Tree: Example

Red-Black Tree: Example

Red-Black Tree: Example

Case: Right-Right

Red-Black Tree: Example

Case: Right-Right

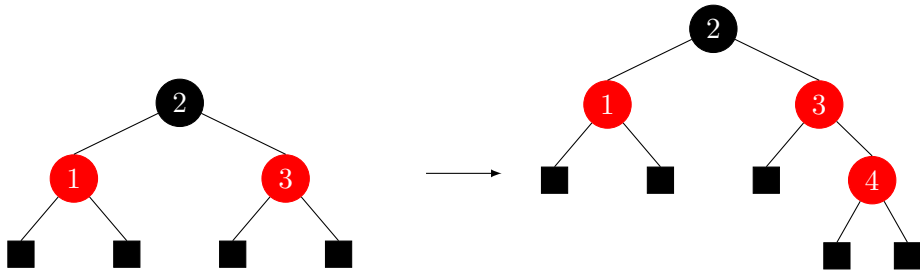# Red-Black Tree: Example

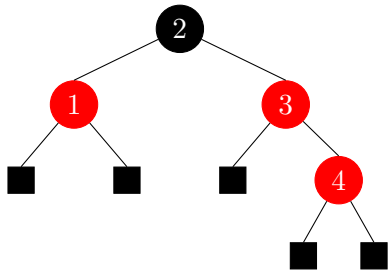**Case: Right-Right**

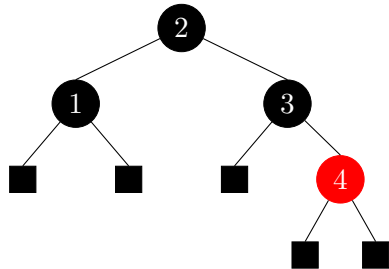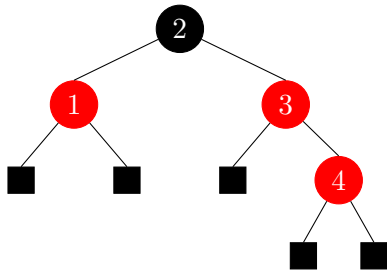# Red-Black Tree: Example

Red-Black Tree: Example

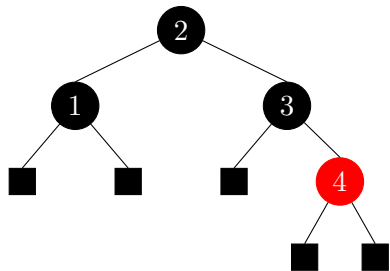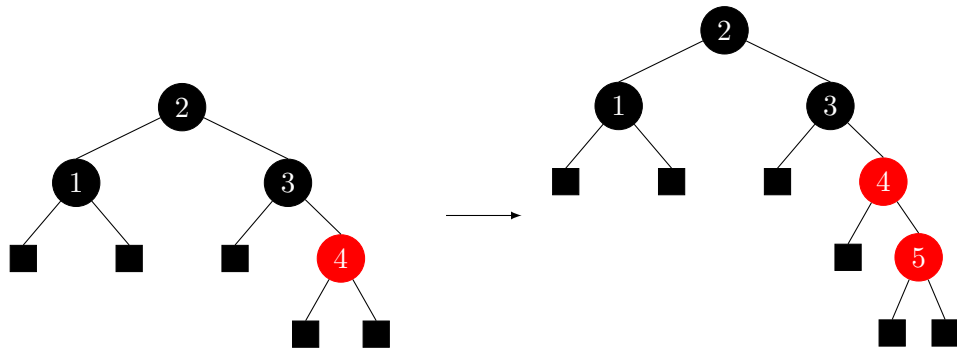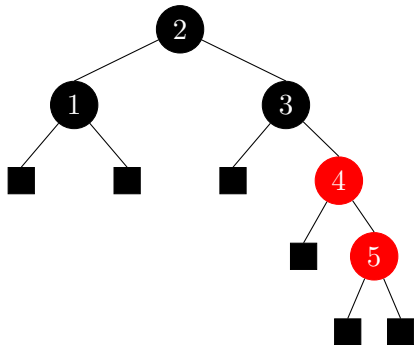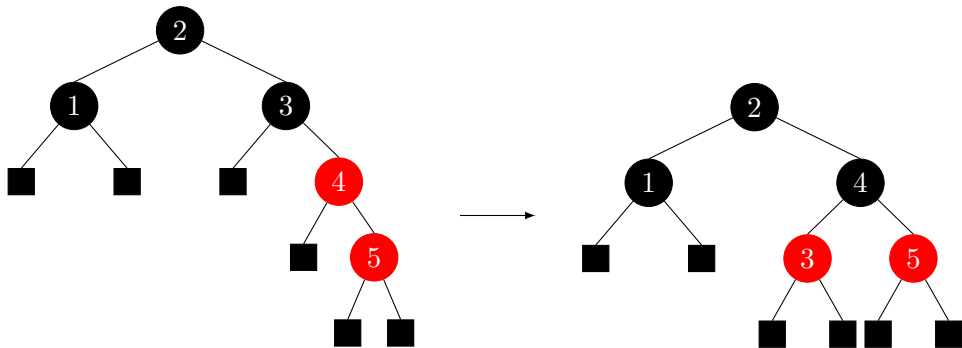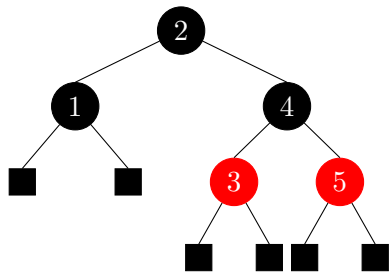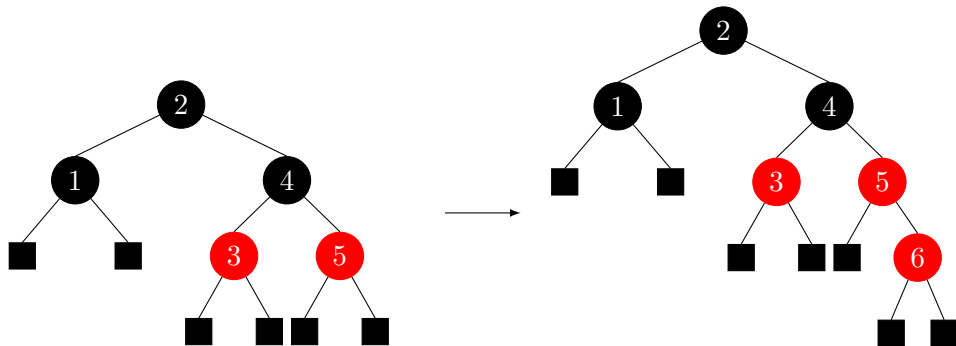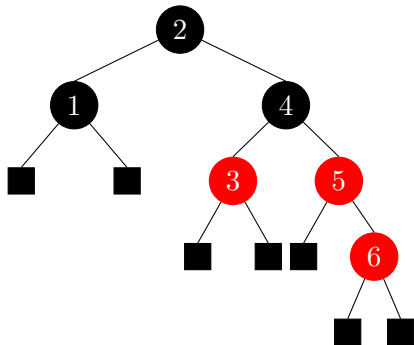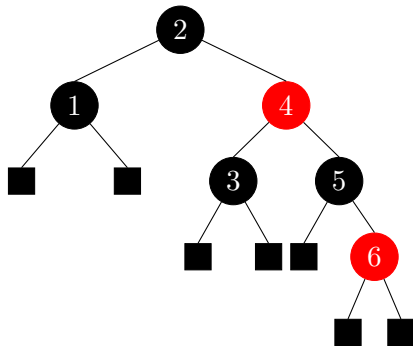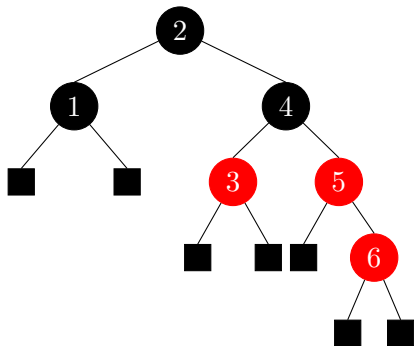*Case: Recolor*

Red-Black Tree: Example

Case: Recolor

Red-Black Tree: Example

Case: Recolor

Red-Black Tree: Example

Red-Black Tree: Example

Case: Right-Right

Red-Black Tree: Example

Case: Right-Right

Red-Black Tree: Example

Case: Right-Right

Red-Black Tree: Example

Red-Black Tree: Example
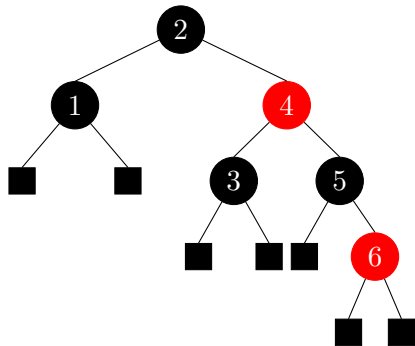

Case: Recolor

Red-Black Tree: Example

Case: Recolor

Red-Black Tree: Example

Case: Recolor

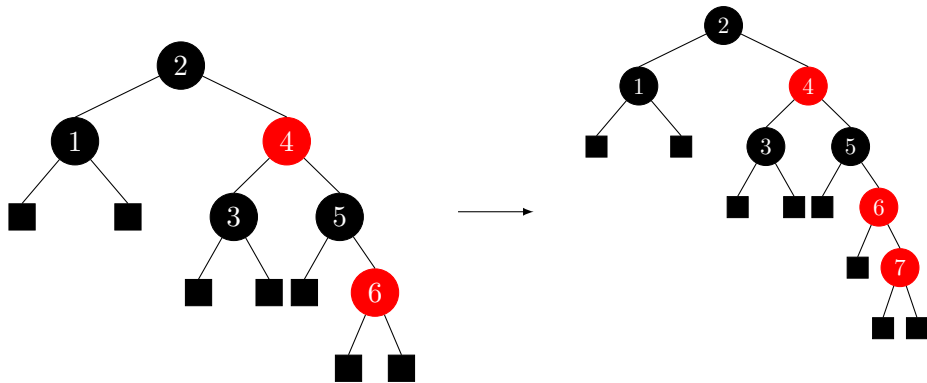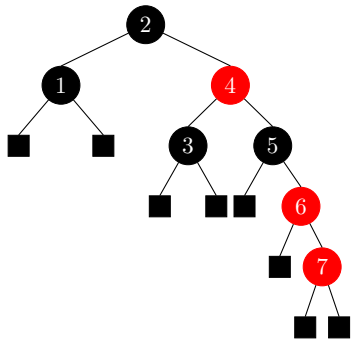# Red-Black Tree: Example

Red-Black Tree: Example

Case: Right-Right

Red-Black Tree: Example

Red-Black Tree: Example