Balanced Trees

Four height-balanced trees:

Red-Black binary tree
  Faster than AVL for insertion and removal

Adelsen-Velskii Landis (AVL) binary tree
  Faster than red-black for lookup

B-tree
  n-way balanced tree

Nguyen-Wong B-trees
  Composite pattern n-way balanced tree

Demo LLRBTree

Insert: 10 20 30 40 50 60 70 80 90

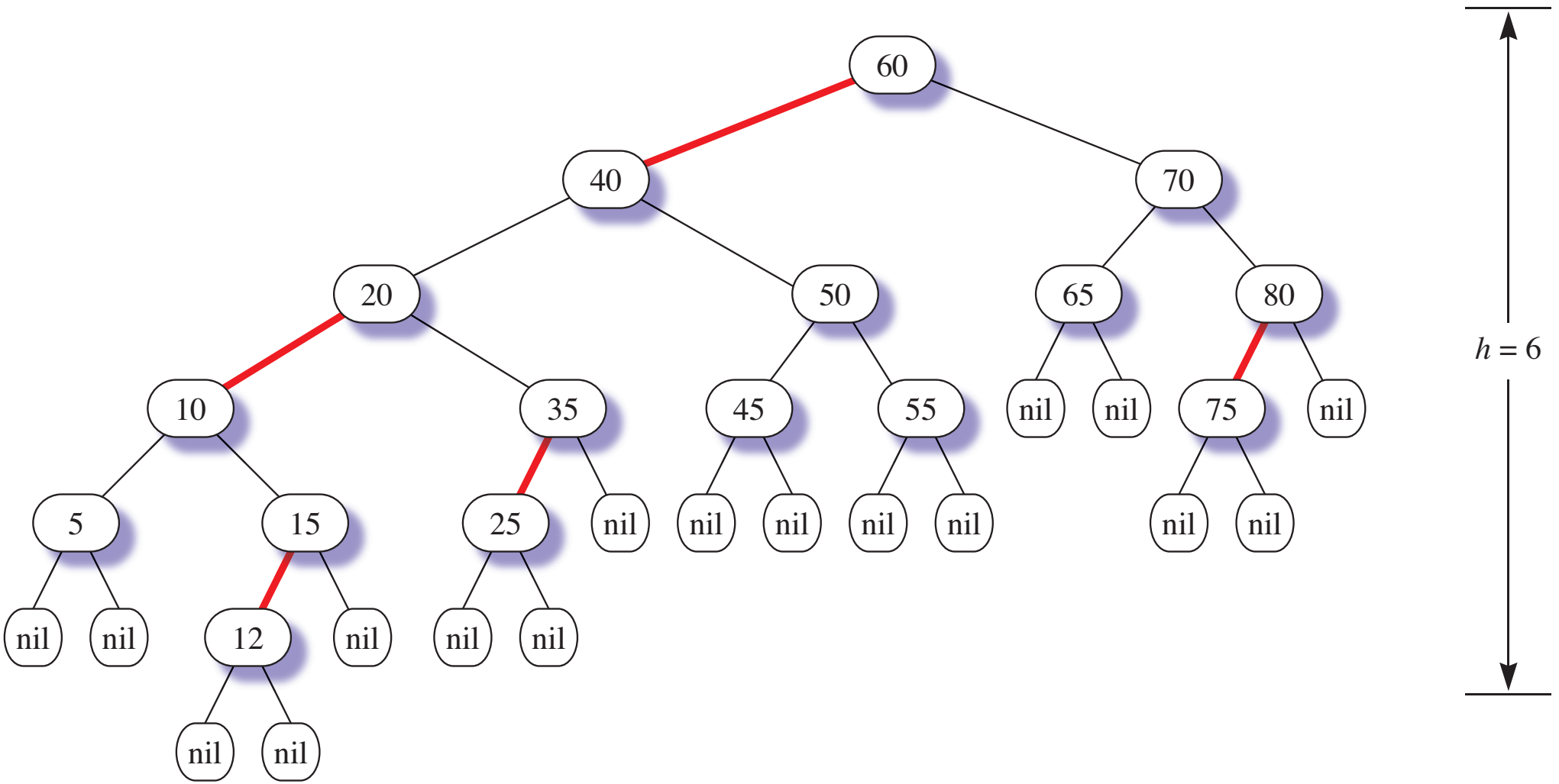
Properties of a red-black tree

- A red-black tree is a binary search tree.
- Every link is either red or black.
- Every child that is an empty tree is a leaf with a black link to its parent.
- No path from the root to any leaf has two red links in a row.
- All paths from each node to all its leaves have the same number of black links.

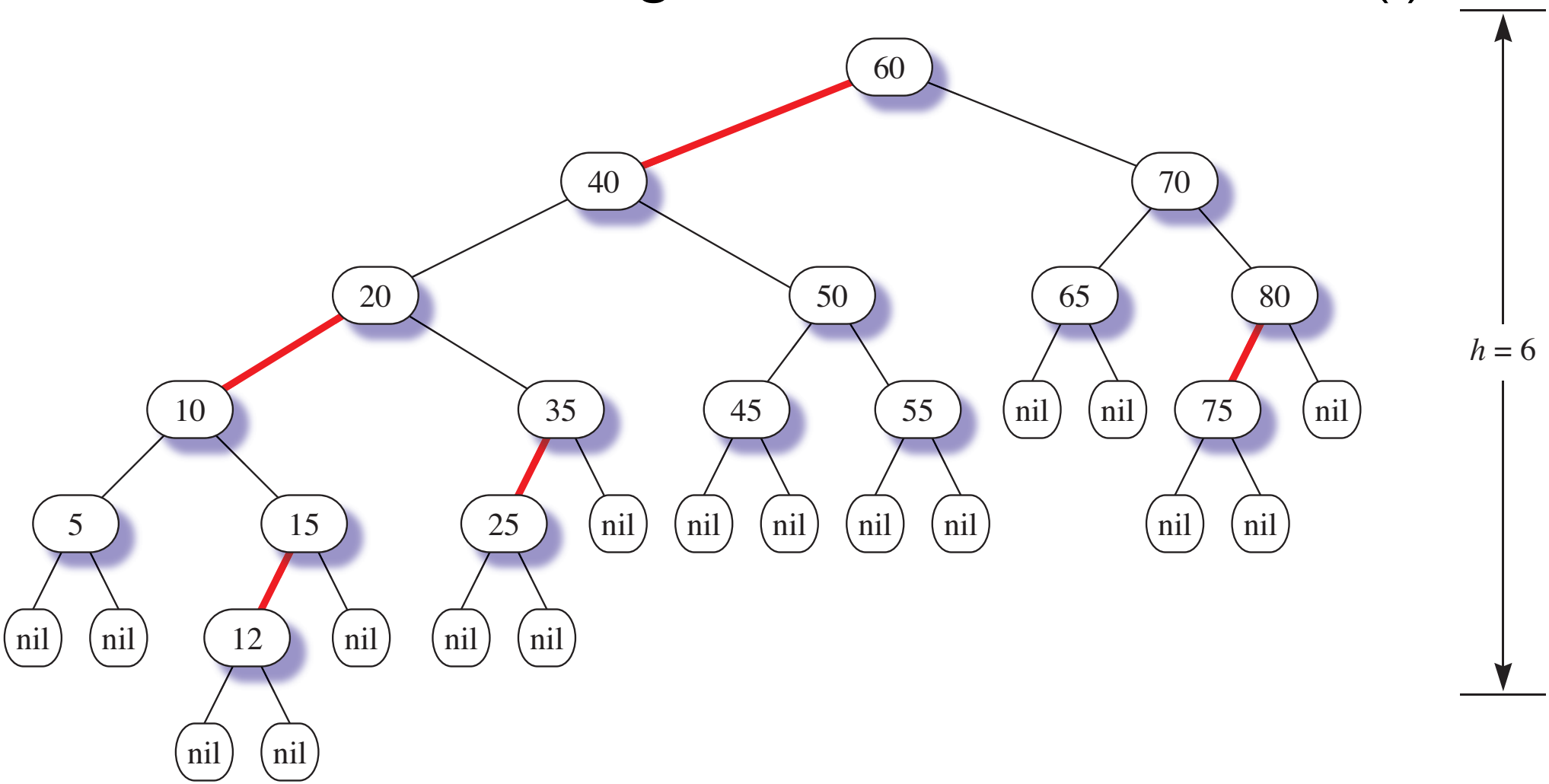
Properties of a left-leaning red-black tree

- A left-leaning red-black tree is a red-black tree.
- Every red link is to a left child.

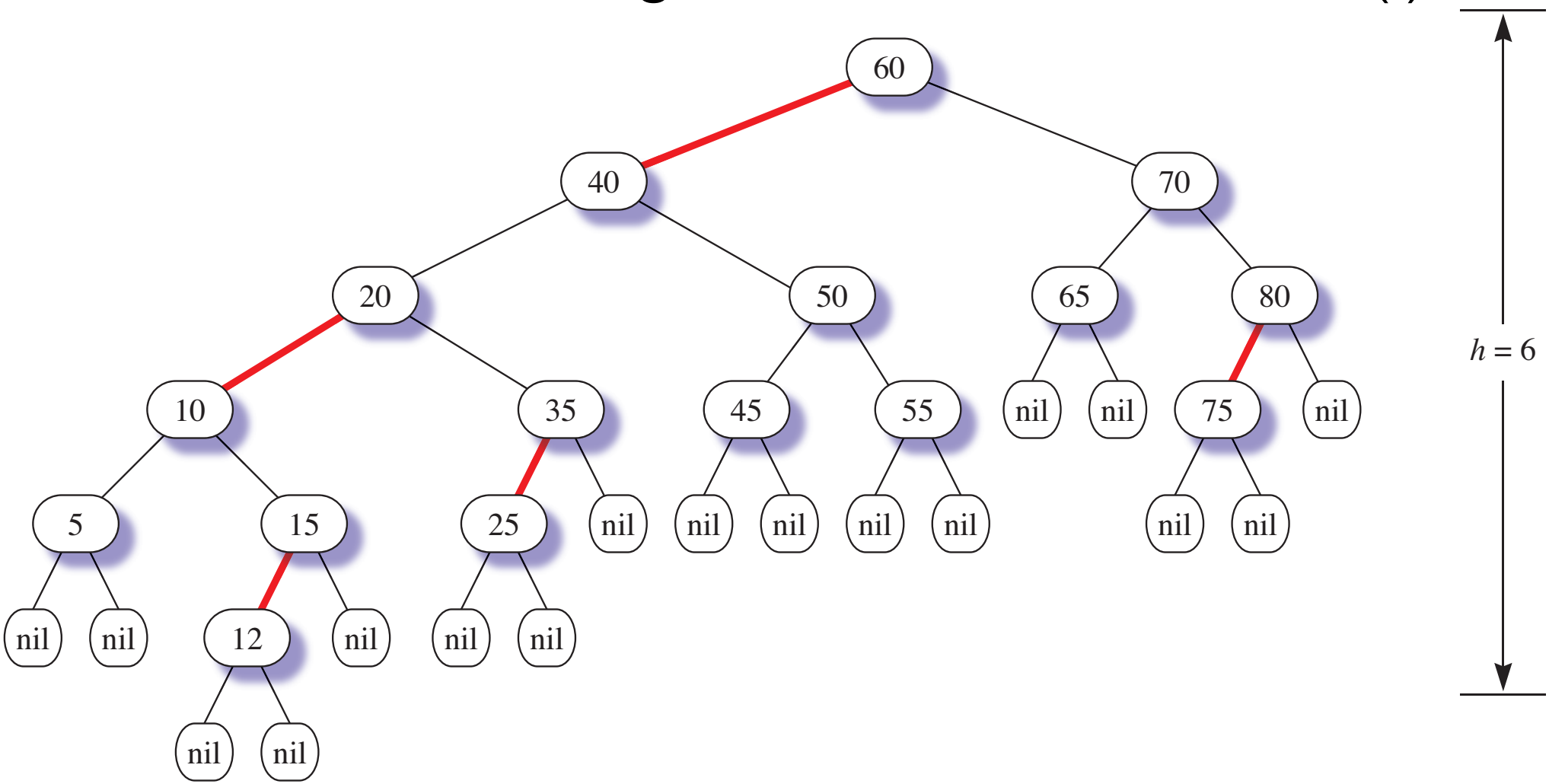
Sedgewick, has shown that this restriction greatly simplifies the code for both insertion and deletion.


60
40
70
20
50
65
80
10
35
45
55
nil
nil
75
nil
5
15
25
nil
nil
nil
nil
nil
nil
nil
nil
nil
nil
12
nil
nil
nil
nil
nil
nil
h = 6

The number of black links from the root to any leaf is called the black height of the tree, denoted bh ( ).



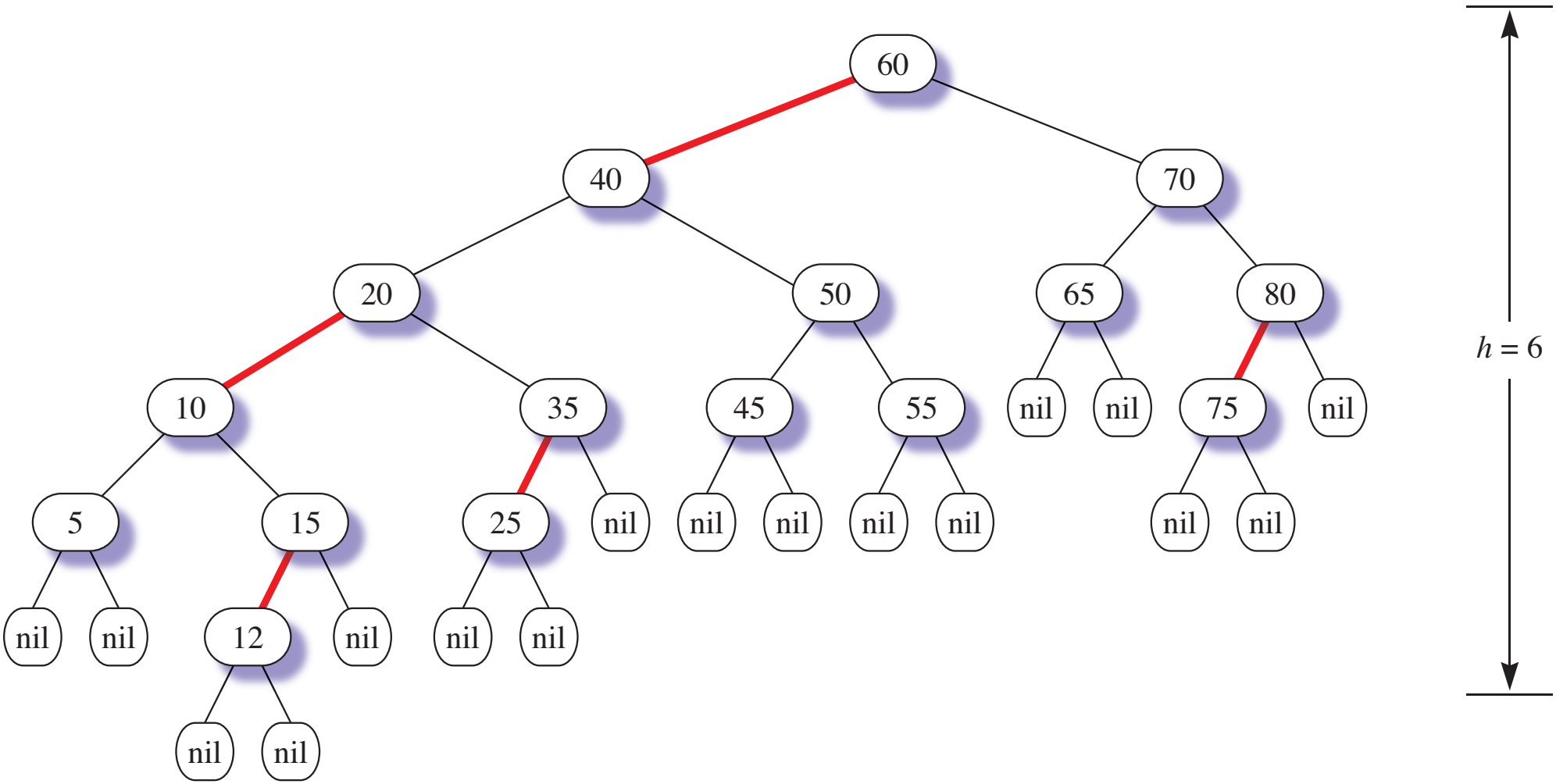
The number of black links from the root to any leaf is called the black height of the tree, denoted bh ( ).



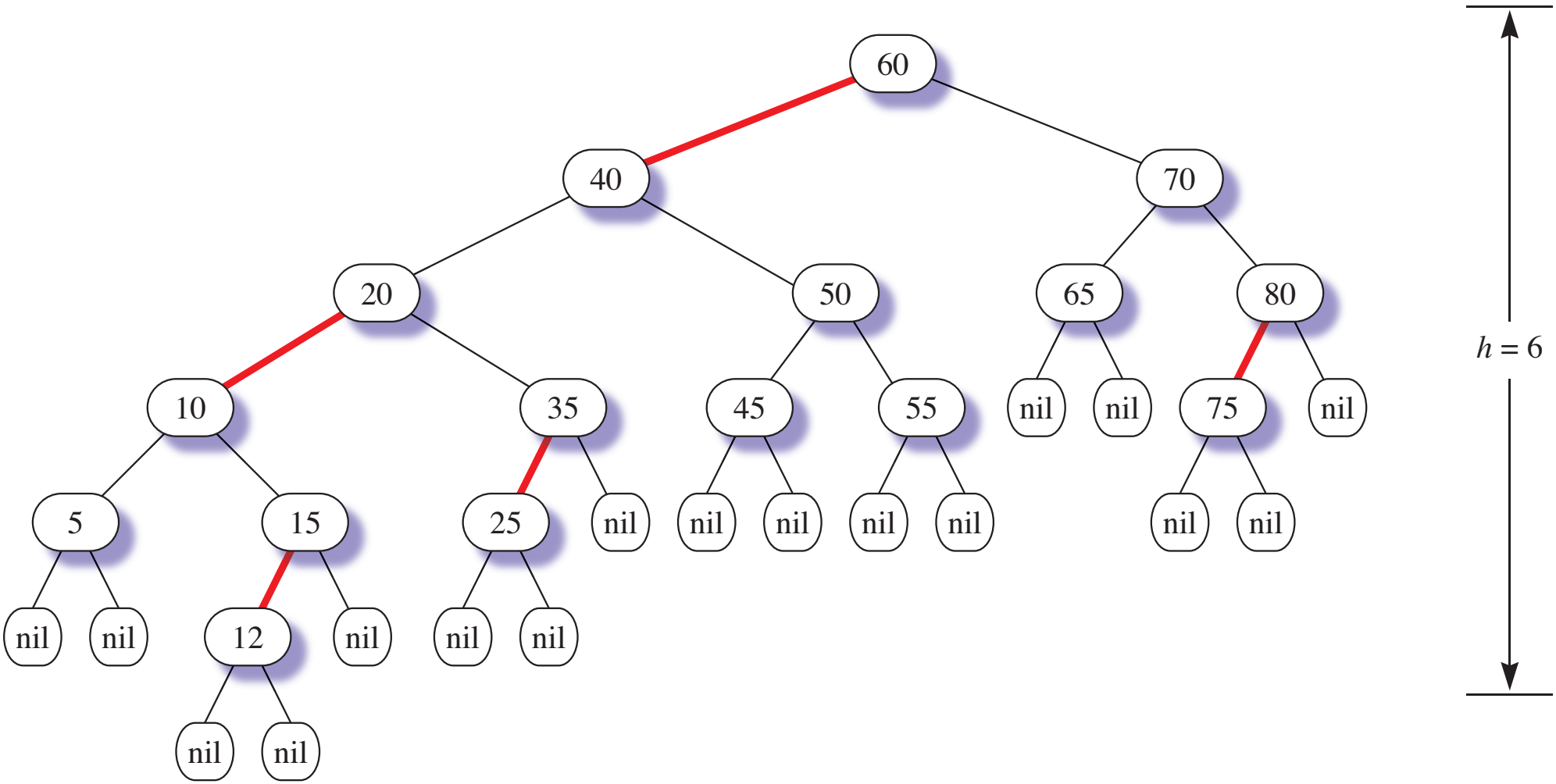
bh (60) = 3

Performance of LLRBTree

Lemma 1: No path from the root to a leaf is more than twice as long as any other. In a red-black tree of height h rooted at x, $\text{bh}(x) \geq h/2$.



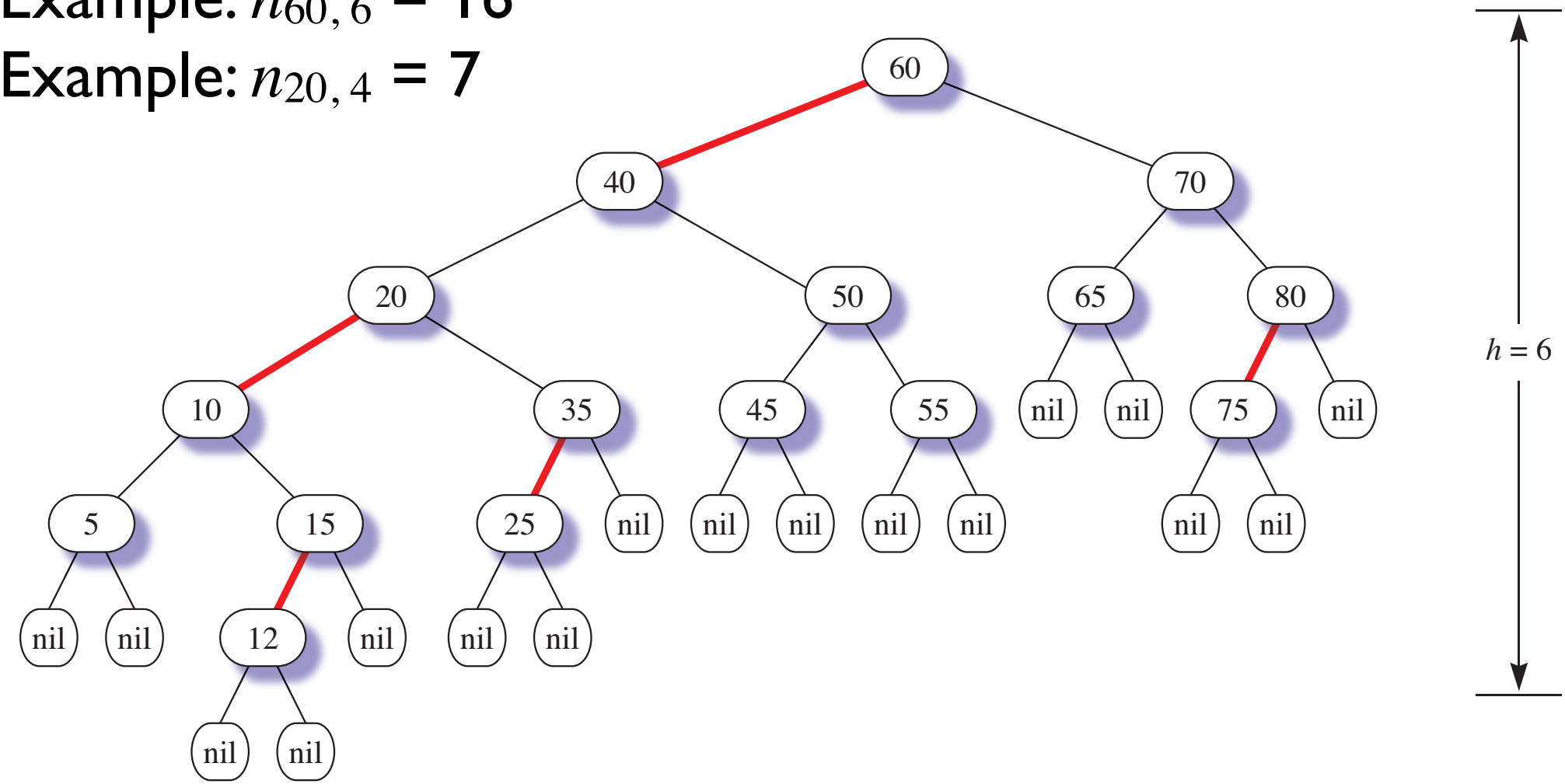
Define $n_{x,\,h}$ as the number of internal nodes in a red-black tree of height h rooted at node x.



Define $n_{x,\,h}$ as the number of internal nodes in a red-black tree of height h rooted at node x.

Example: $n_{60,\,6} = 16$

Example: $n_{20,\,4} = 7$



Lemma 2: $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$.

Lemma 2: $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Lemma 2: $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

Lemma 2: $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$

Lemma 2: $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\text{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle \\
& n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$n_{x,h} \geq 2^{\text{bh}(x)} - 1$

$=$ ⟨Take as the base case that the root is nil⟩

$n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1$

$=$ ⟨With a root of nil, there are no internal nodes⟩

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\text{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle \\
& n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, there are no internal nodes} \rangle \\
& 0 \geq 2^{\text{bh}(\text{nil})} - 1
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\text{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle \\
& n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, there are no internal nodes} \rangle \\
& 0 \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, the black height is } 0 \rangle
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\text{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle \\
& n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, there are no internal nodes} \rangle \\
& 0 \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, the black height is } 0 \rangle \\
& 0 \geq 2^0 - 1
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$$
\begin{aligned}
& n_{x,h} \geq 2^{\text{bh}(x)} - 1 \\
= \quad & \langle \text{Take as the base case that the root is nil} \rangle \\
& n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, there are no internal nodes} \rangle \\
& 0 \geq 2^{\text{bh}(\text{nil})} - 1 \\
= \quad & \langle \text{With a root of nil, the black height is } 0 \rangle \\
& 0 \geq 2^0 - 1 \\
= \quad & \langle \text{Math} \rangle
\end{aligned}
$$

Lemma 2: $n_{x,h} \geq 2^{\text{bh}(x)} - 1$.

Proof: The proof is by mathematical induction on the height of the tree.

Base case:

$n_{x,h} \geq 2^{\text{bh}(x)} - 1$

$=$ $\langle$Take as the base case that the root is nil$\rangle$

$n_{\text{nil},h} \geq 2^{\text{bh}(\text{nil})} - 1$

$=$ $\langle$With a root of nil, there are no internal nodes$\rangle$

$0 \geq 2^{\text{bh}(\text{nil})} - 1$

$=$ $\langle$With a root of nil, the black height is 0$\rangle$

$0 \geq 2^0 - 1$

$=$ $\langle$Math$\rangle$

true ∎

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

Link from x to child c is red => bh(c) = bh(x).
Link from x to child c is black => bh(c) = bh(x) – 1.

Minimum number of nodes in tree rooted at x
=> both children have minimum
=> links from x to both are black

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$n_{x,h}$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$\begin{aligned} & n_{x,h} \\ = \quad & \langle \text{Definition of binary tree} \rangle \end{aligned}$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1}
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\mathrm{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\mathrm{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle \\
& 1 + 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle \\
& 1 + 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle \\
& 1 + 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle \\
& 2 \cdot 2^{\text{bh}(x)-1} - 1
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle \\
& 1 + 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle \\
& 2 \cdot 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle
\end{aligned}
$$

Induction case: Must prove that $n_{x,h} \geq 2^{\text{bh}(x)} - 1$ for a height h tree using $n_{w,h-1} \geq 2^{\text{bh}(w)} - 1$ for a height $h-1$ tree as the inductive hypothesis.

$$
\begin{aligned}
& n_{x,h} \\
= \quad & \langle \text{Definition of binary tree} \rangle \\
& 1 + n_{l,h-1} + n_{r,h-1} \\
\geq \quad & \langle \text{Inductive hypothesis, twice} \rangle \\
& 1 + 2^{\text{bh}(l)} - 1 + 2^{\text{bh}(r)} - 1 \\
\geq \quad & \langle \text{Minimum occurs when links to } l \text{ and } r \text{ are black} \rangle \\
& 1 + 2^{\text{bh}(x)-1} - 1 + 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle \\
& 2 \cdot 2^{\text{bh}(x)-1} - 1 \\
= \quad & \langle \text{Math} \rangle \\
& 2^{\text{bh}(x)} - 1 \quad \blacksquare
\end{aligned}
$$

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

$n_{x,h}$

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

$$
\begin{aligned}
& n_{x,h} \\
\geq \quad & \langle \text{Lemma 2} \rangle
\end{aligned}
$$

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

$$
\begin{aligned}
& n_{x,h} \\
\geq & \quad \langle \text{Lemma 2} \rangle \\
& 2^{\text{bh}(x)} - 1
\end{aligned}
$$

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

$$
\begin{aligned}
& n_{x,h} \\
\geq & \quad \langle \text{Lemma 2} \rangle \\
& 2^{\text{bh}(x)} - 1 \\
\geq & \quad \langle \text{Lemma 1} \rangle
\end{aligned}
$$

Lemma 3: $n_{x,h} \geq 2^{h/2} - 1$.

Proof:

$$
\begin{aligned}
& n_{x,h} \\
\geq & \quad \langle \text{Lemma 2} \rangle \\
& 2^{\text{bh}(x)} - 1 \\
\geq & \quad \langle \text{Lemma 1} \rangle \\
& 2^{h/2} - 1 \quad \blacksquare
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$n \geq 2^{h/2} - 1$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ &= \quad \langle \text{Math} \rangle \end{aligned}$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1 \\
= & \quad \langle \text{Take lg of both sides, monotonicity of } \lg() \rangle
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1 \\
= & \quad \langle \text{Take lg of both sides, monotonicity of lg}() \rangle \\
& h/2 \leq \lg(n+1)
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1 \\
= & \quad \langle \text{Take lg of both sides, monotonicity of lg}() \rangle \\
& h/2 \leq \lg(n+1) \\
= & \quad \langle \text{Math} \rangle
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1 \\
= & \quad \langle \text{Take lg of both sides, monotonicity of } \lg() \rangle \\
& h/2 \leq \lg(n+1) \\
= & \quad \langle \text{Math} \rangle \\
& h \leq 2\lg(n+1) \quad \blacksquare
\end{aligned}
$$

Theorem, red-black tree height bound: A red-black tree with n internal nodes has height $h \leq 2\lg(n+1)$.

Proof: Starting with Lemma 3,

$$
\begin{aligned}
& n \geq 2^{h/2} - 1 \\
= & \quad \langle \text{Math} \rangle \\
& 2^{h/2} \leq n + 1 \\
= & \quad \langle \text{Take lg of both sides, monotonicity of lg}() \rangle \\
& h/2 \leq \lg(n+1) \\
= & \quad \langle \text{Math} \rangle \\
& h \leq 2\lg(n+1) \quad \blacksquare
\end{aligned}
$$

For comparison, height-balanced AVL trees have $h \leq 1.44\lg(n+2) - 0.33$. They are more balanced than red-black trees and have faster retrieval (1.44 coefficient compared to 2 for red-black trees) but have slower insertion and removal.

Implementation of LLRBTree

```
// ========== LLRBTree ==========
template<class T>
class LLRBTree {
private:
    shared_ptr<Node<T>> _root;

public:
    LLRBTree() = default; //Constructor
    // Post: This left-leaning red-black tree
    // is initialized to be empty.
```

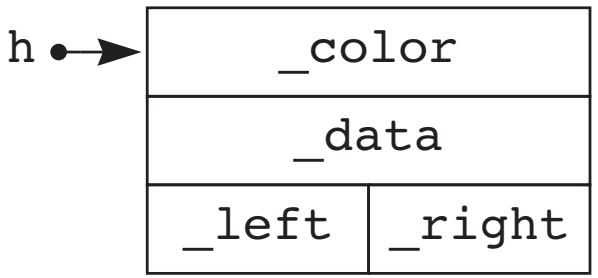
```
    bool contains(T const &data);
    // Post: Returns true if this tree contains data.

    void insert(T const &data);
    // Post: data is stored in this tree in order
    // with no duplicates.

    void remove(T const &data);
    // Post: If data is found, then it is removed
    // from this tree.

    void toStream(ostream &os) const;
    // Post: A string representation of this tree
    // is streamed to os.
};
```

```
// ========== Node ==========
template<class T>
class Node {
    friend class LLRBTree<T>;

private:
    shared_ptr<Node> _left;
    T _data;
    shared_ptr<Node> _right;
    bool _color; // Color of link from parent.
```

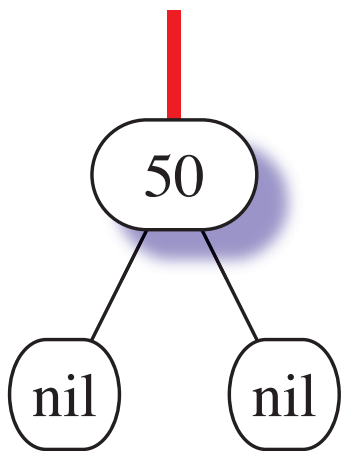
```
public:
    explicit Node(T data);
    // Post: This node is allocated with _data set to data,
    // _color set to RED, and _left, _right set to nullptr.
```

```
private:
    shared_ptr<Node<T>> fixup(shared_ptr<Node<T>> h);
    void flipColors(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> insert(shared_ptr<Node<T>> h, T const &data);
    bool isRed(shared_ptr<Node<T>> h);
    T min(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> moveRedLeft(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> moveRedRight(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> remove(shared_ptr<Node<T>> h, T const &data);
    shared_ptr<Node<T>> removeMin(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> rotateLeft(shared_ptr<Node<T>> h);
    shared_ptr<Node<T>> rotateRight(shared_ptr<Node<T>> h);
    void toStream(shared_ptr<Node<T>> h,
                  string prRight,
                  string prRoot,
                  string prLeft,
                  ostream &os) const;
};
```

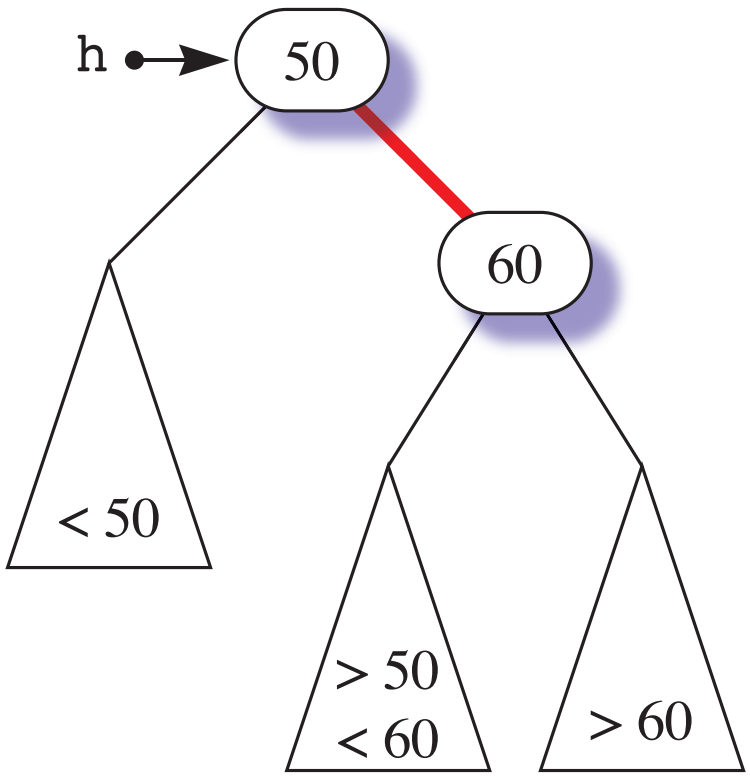
```
const bool RED = true;
const bool BLACK = false;

// ========= Node =========
template<class T>
private:
    shared_ptr<Node> _left;
    T _data;
    shared_ptr<Node> _right;
    bool _color; // Color of link from parent.

// ========= isRed =========
// Post: Returns false if h == nullptr.
// Otherwise returns true if h node is RED.
template<class T>
bool Node<T>::isRed(shared_ptr<Node<T>> h) {
    return h == nullptr ? BLACK : h->_color;
}
```



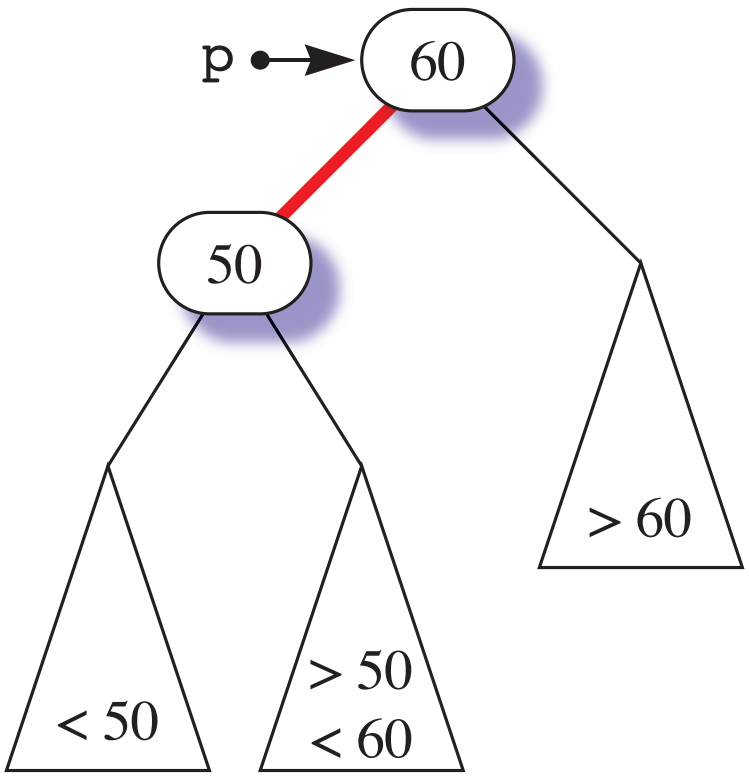
```
// ========== Constructor ==========
template<class T>
Node<T>::Node(T data):
    _color(RED),
    _data(data) {
}
```



The rotate left operation

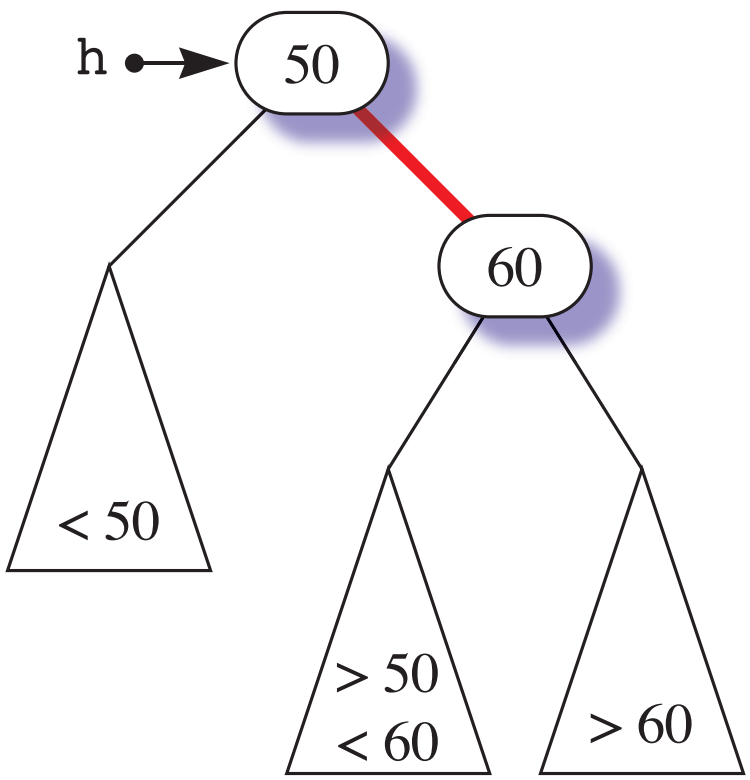


(a) Before rotate left.

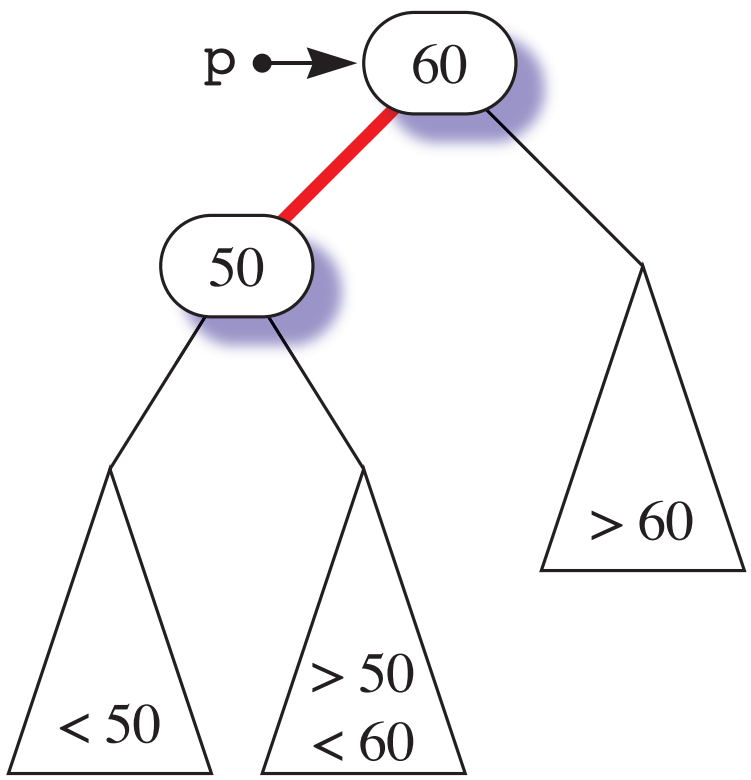


(b) After rotate left.

The rotate left operation



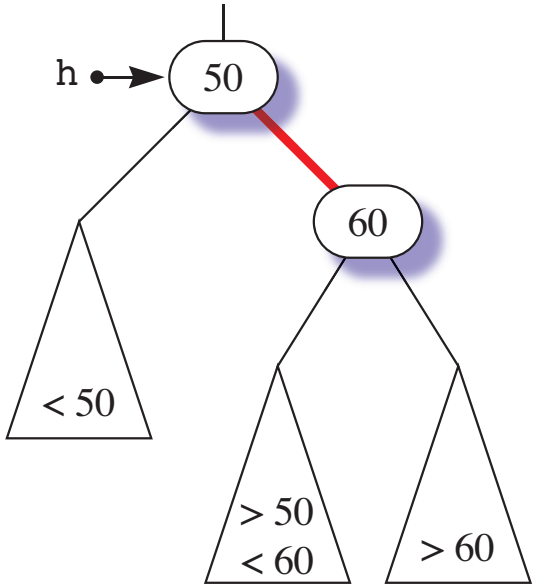
(a) Before rotate left.



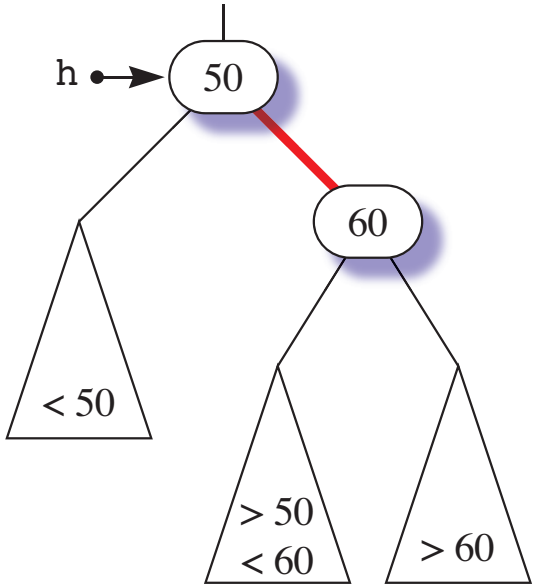
(b) After rotate left.

The black height and the order are maintained.

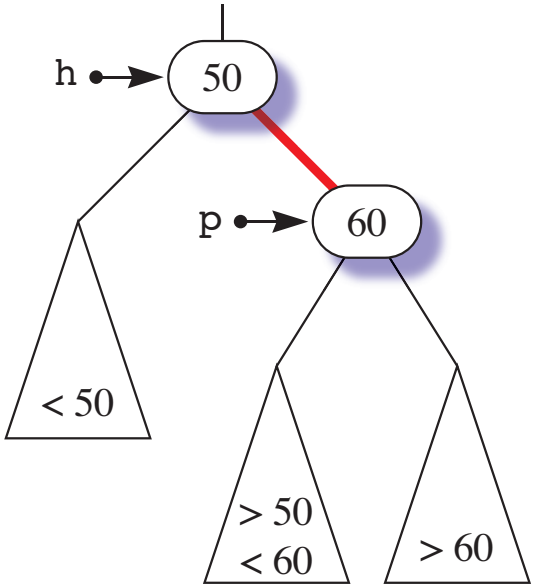
```
// ========= rotateLeft =========
// Pre: h->_right is RED.
// Post: h is rotated left.
// Post: A pointer p to the root node of the modified tree is returned.
// Post: p->left is RED.
// Unchanged: Color of link to new root from its parent.
template<class T>
shared_ptr<Node<T>> Node<T>::rotateLeft(shared_ptr<Node<T>> h) {
    if (!isRed(h->_right)) {
        cerr << "rotateLeft precondition violated: "
             << "h->_right is not RED" << endl;
        throw -1;
    }
    shared_ptr<Node<T>> p = h->_right;
    h->_right = p->_left;
    p->_left = h;
    p->_color = h->_color;
    h->_color = RED;
    return p;
}
```



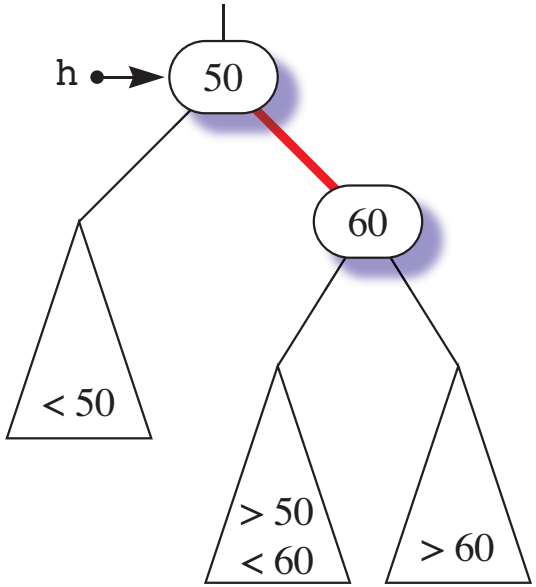
(a) Initial tree.



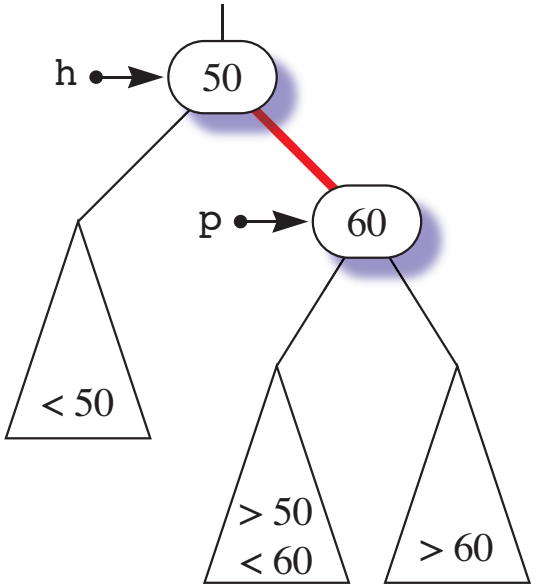
(a) Initial tree.



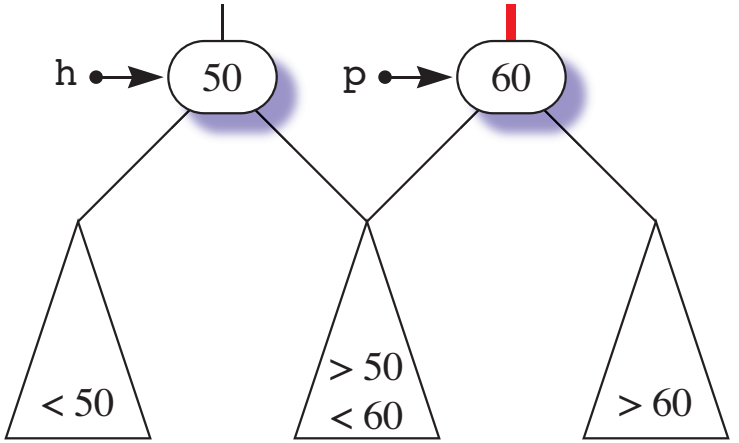
(b) `p = h->_right`



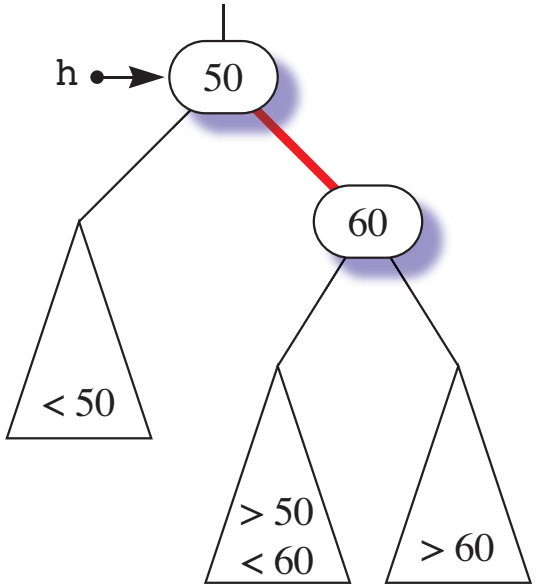
(a) Initial tree.



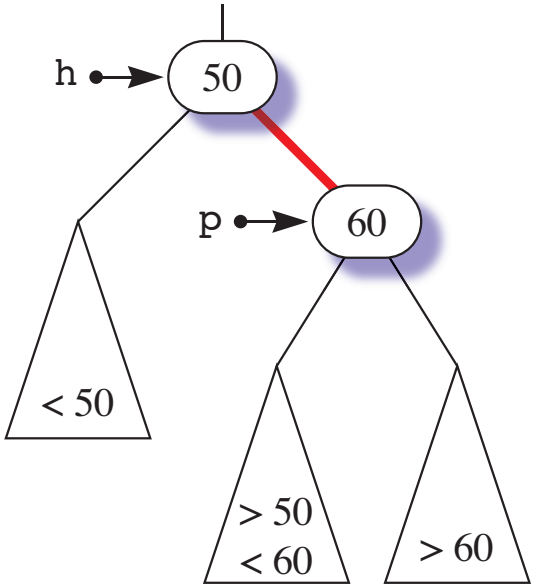
(b) `p = h->_right`



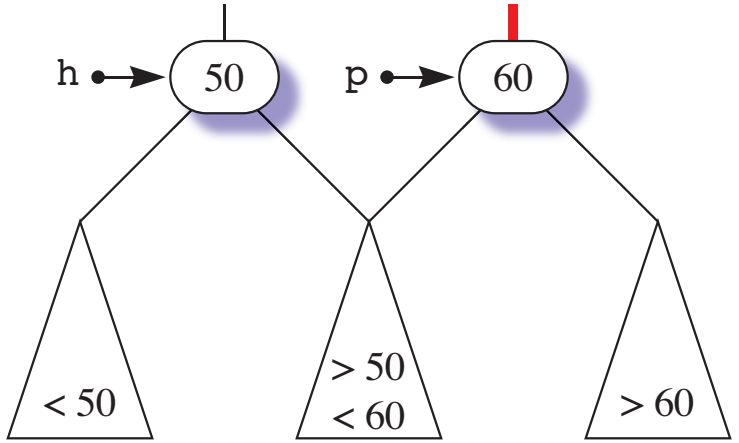
(c) `h->_right = p->_left`



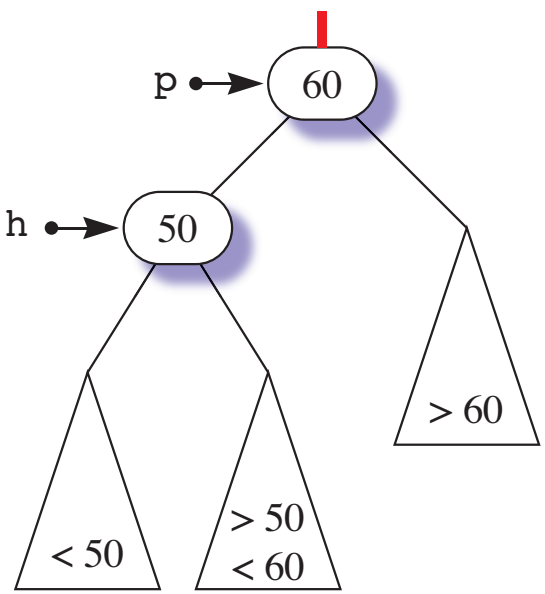
(a) Initial tree.



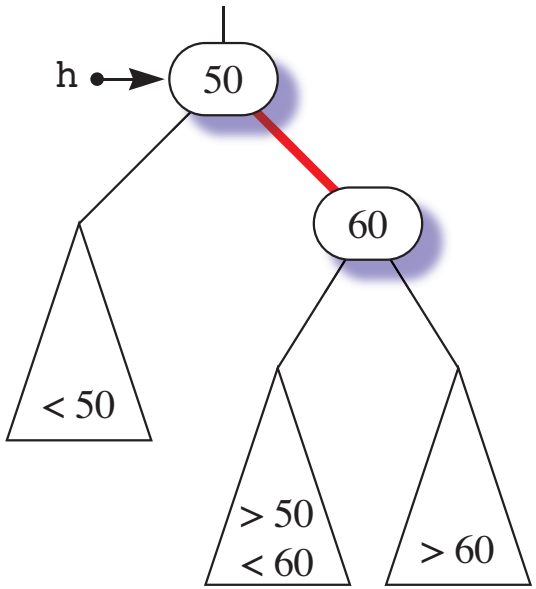
(b) `p = h->_right`



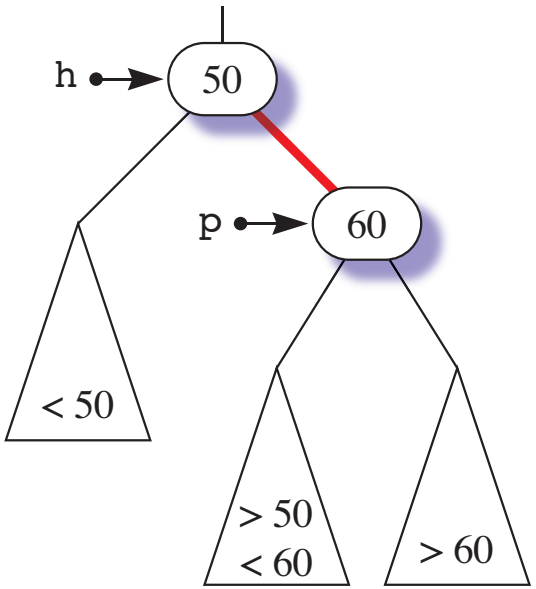
(c) `h->_right = p->_left`



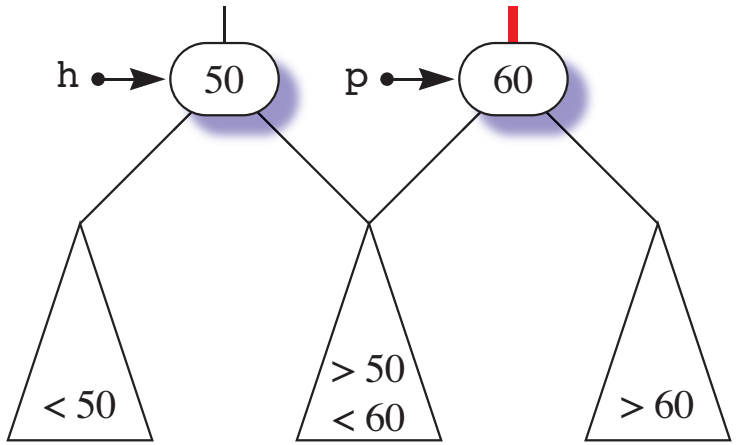
(d) `p->_left = h`



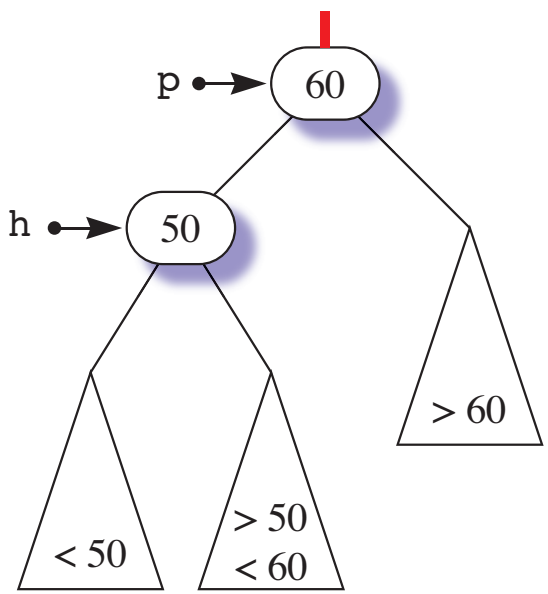
(a) Initial tree.



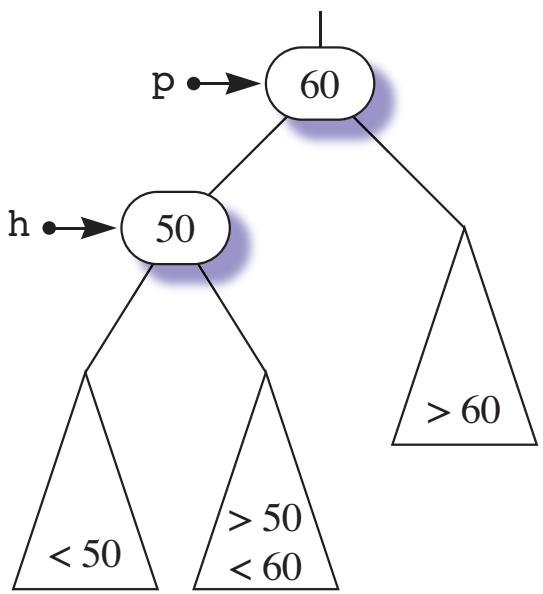
(b) `p = h->_right`



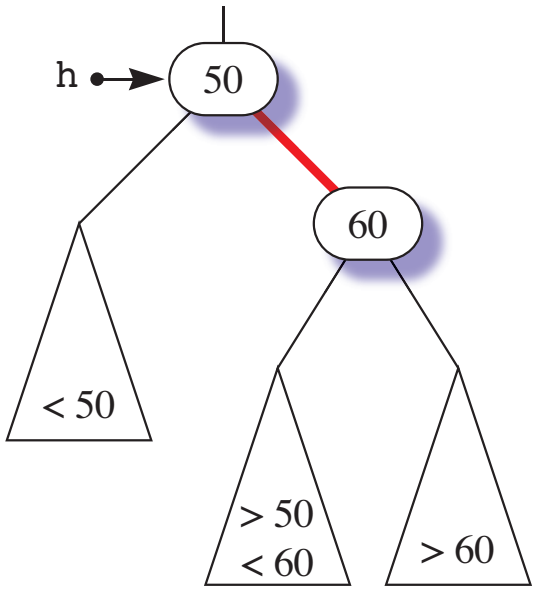
(c) `h->_right = p->_left`



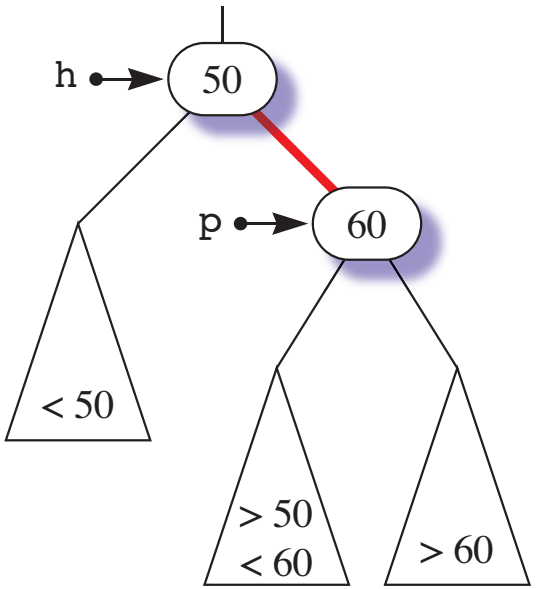
(d) `p->_left = h`



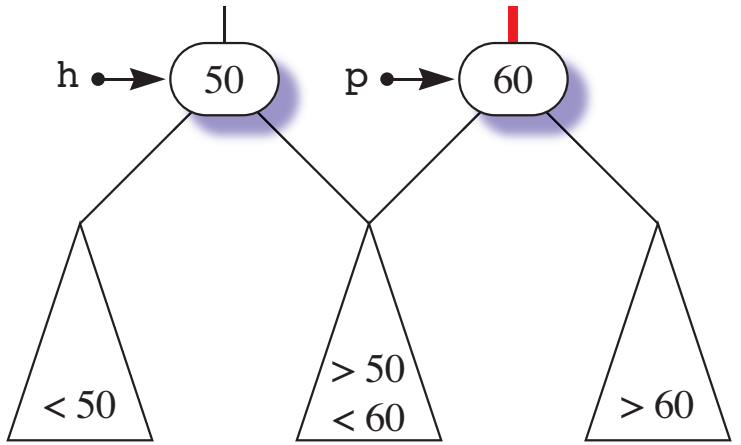
(e) `p->_color = h->_color`



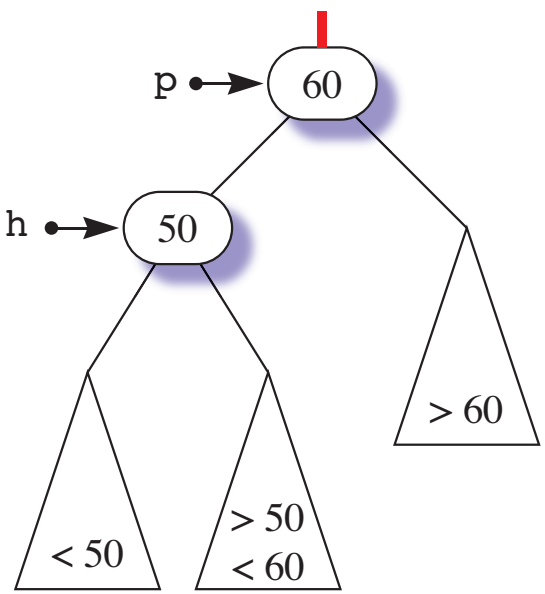
(a) Initial tree.



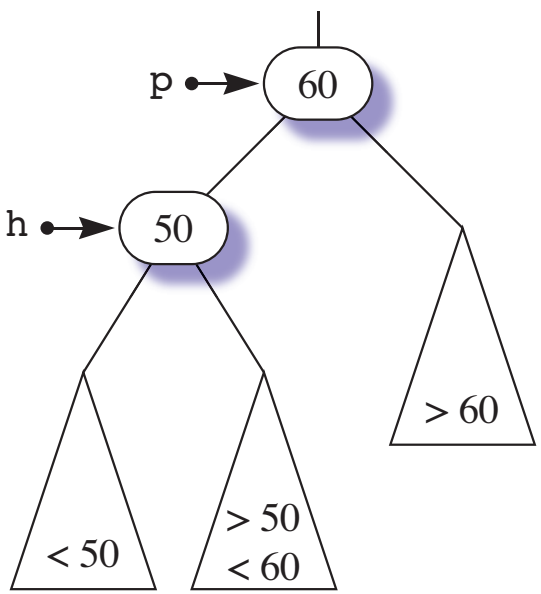
(b) `p = h->_right`



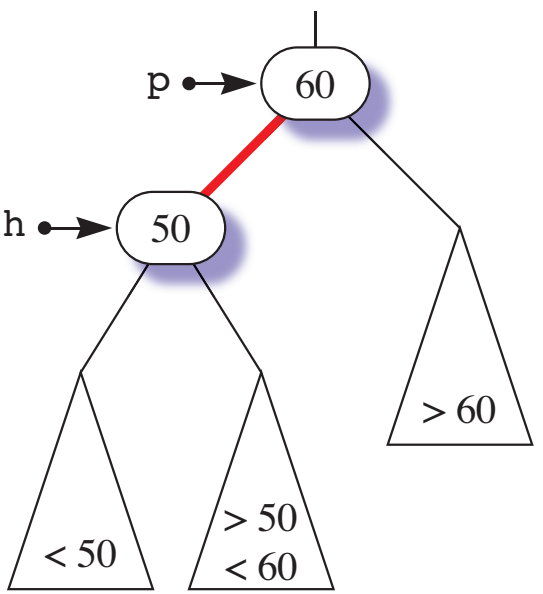
(c) `h->_right = p->_left`



(d) `p->_left = h`

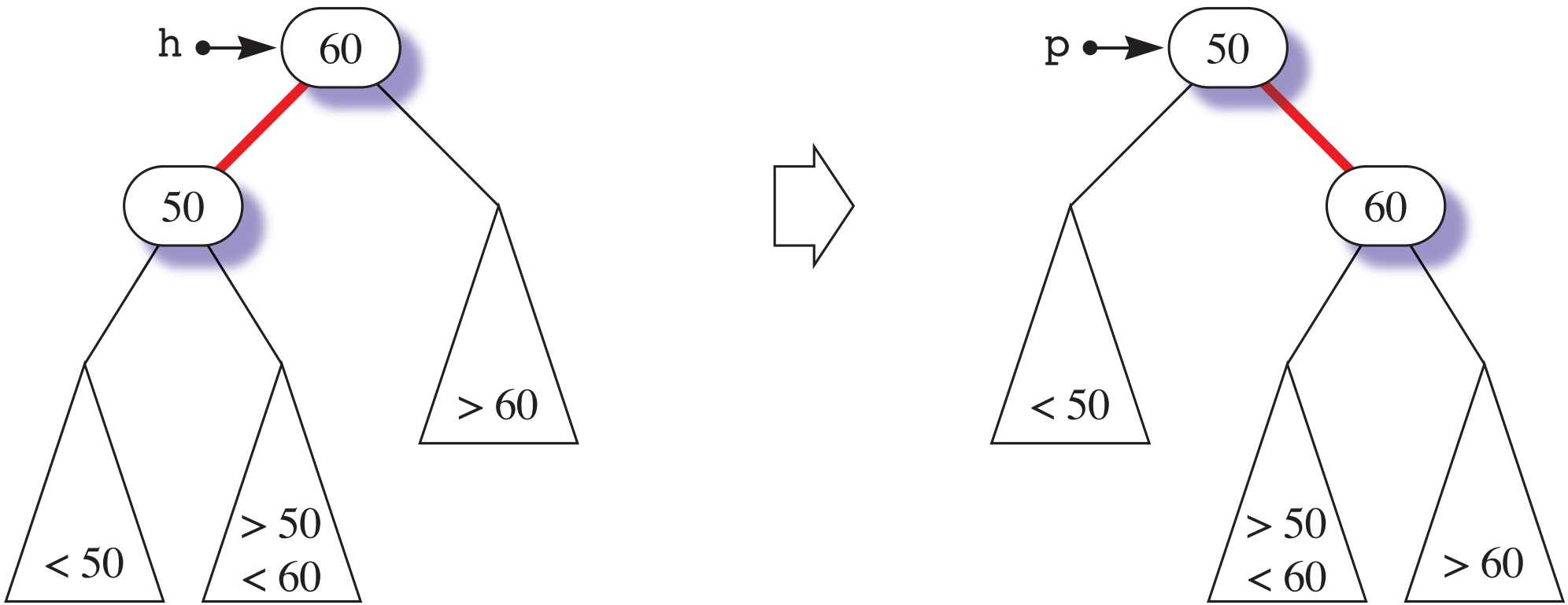


(e) `p->_color = h->_color`



(f) `h->_color = RED`

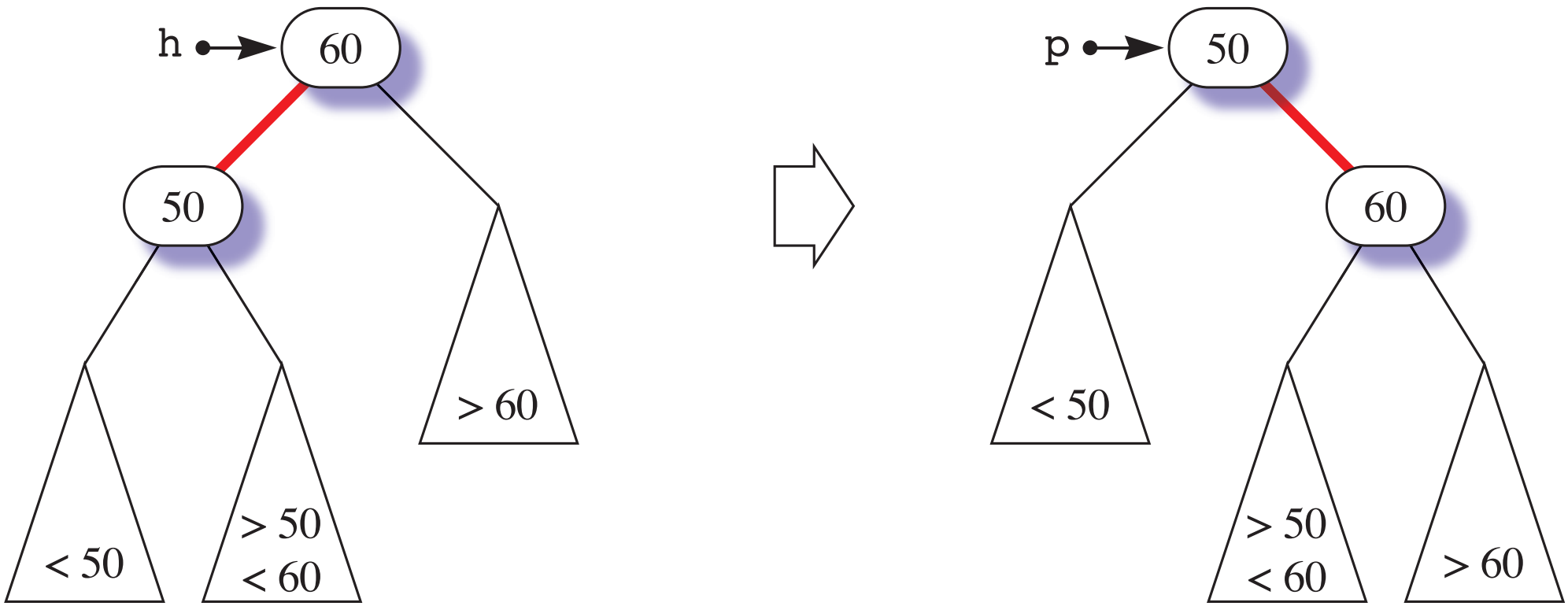
The rotate right operation



(a) Before rotate right.

(b) After rotate right.

The rotate right operation



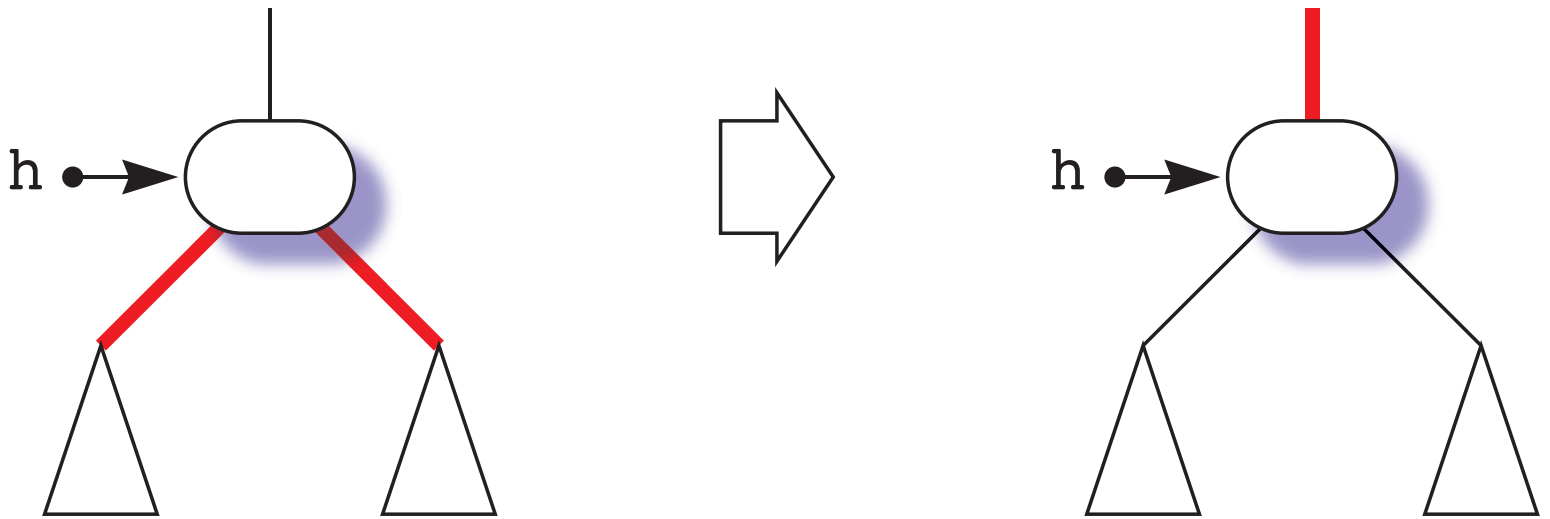
(a) Before rotate right.

(b) After rotate right.

The black height and the order are maintained.

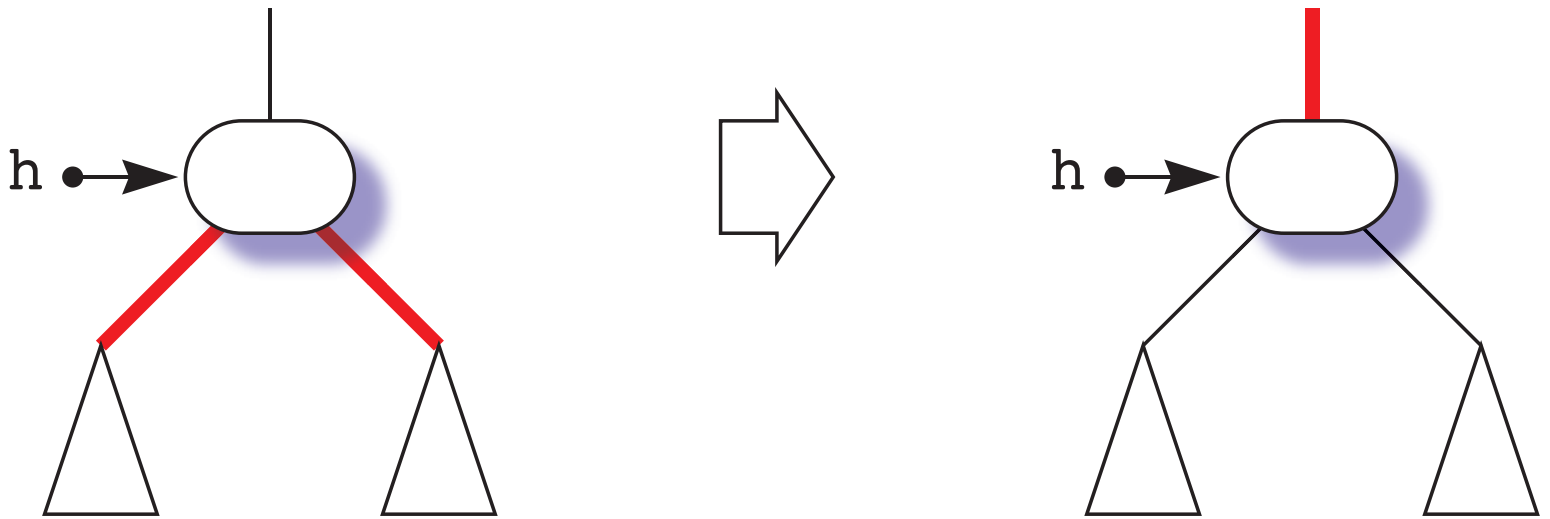
```
// ========= rotateRight =========
// Pre: h->_left is RED.
// Post: h is rotated right.
// Post: A pointer p to the root node of the modified tree is returned.
// Post: p->right is RED.
// Unchanged: Color of link to new root from its parent.
template<class T>
shared_ptr<Node<T>> Node<T>::rotateRight(shared_ptr<Node<T>> h) {
    cerr << "rotateRight(): Exercise for the student." << endl;
    throw -1;
}
```

The flip colors operation



(a) Before flip colors. **(b)** After flip colors.

The flip colors operation



(a) Before flip colors. **(b)** After flip colors.

The black height is maintained if `h` is black and `h->_left` and `h->_right` are both red.

```
// ========= flipColors =========
// Pre: Children of h have the same color.
// Post: The colors of h and its children are flipped.
template<class T>
void Node<T>::flipColors(shared_ptr<Node<T>> h) {
    if (isRed(h->_left) != isRed(h->_right)) {
        cerr << "flipColors precondition violated: "
             << "Children of h have different colors." << endl;
        throw -1;
    }
    h->_color = !h->_color;
    cerr << "flipColors(): Exercise for the student." << endl;
    throw -1;
```

Inserting into LLRBTree

```
// ========= insert =========
// Post: data is stored in this tree in order with no duplicates.
template<class T>
void LLRBTree<T>::insert(T const &data) {
    _root = _root->insert(_root, data);
    _root->_color = BLACK;
}

// Post: val is stored in root node h in order as a leaf.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::insert(shared_ptr<Node<T>> h, T const &data) {
    if (h == nullptr) {
        return make_shared<Node<T>>(data);
    }
    if (data < h->_data) {
        h->_left = insert(h->_left, data);
    }
    else if (data > h->_data) {
        h->_right = insert(h->_right, data);
    } // else no duplicates.
    return fixup(h);
}
```

Before calling `fixup(h)`, which properties might be violated?

- A red-black tree is a binary search tree.
- Every link is either red or black.
- Every child that is an empty tree is a leaf with a black link to its parent.
- No path from the root to any leaf has two red links in a row.
- All paths from each node to all its leaves have the same number of black links.
- A left-leaning red-black tree is a red-black tree.
- Every red link is to a left child.

Before calling `fixup(h)`, which properties might be violated?

- A red-black tree is a binary search tree.
- Every link is either red or black.
- Every child that is an empty tree is a leaf with a black link to its parent.
- No path from the root to any leaf has two red links in a row.
- All paths from each node to all its leaves have the same number of black links.
- A left-leaning red-black tree is a red-black tree.
- Every red link is to a left child.

The purpose of `fixup(h)`
is to correct two possible problems.

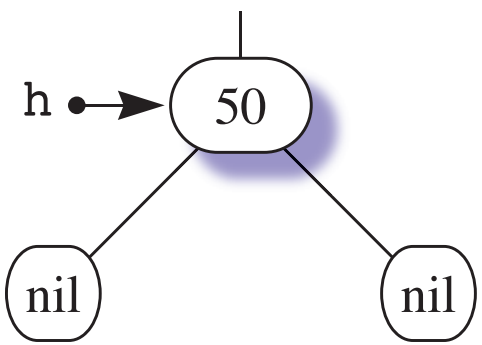
- The additional red link might be to a right child.
- The additional red link might produce two red links in a row.

`fixup(h)` makes three adjustments.

- If the red link is a right link, make it a left link.
- If there are two red left links in a row, rotate right in preparation for the last adjustment.
- If the left and right links are both red, flip colors.

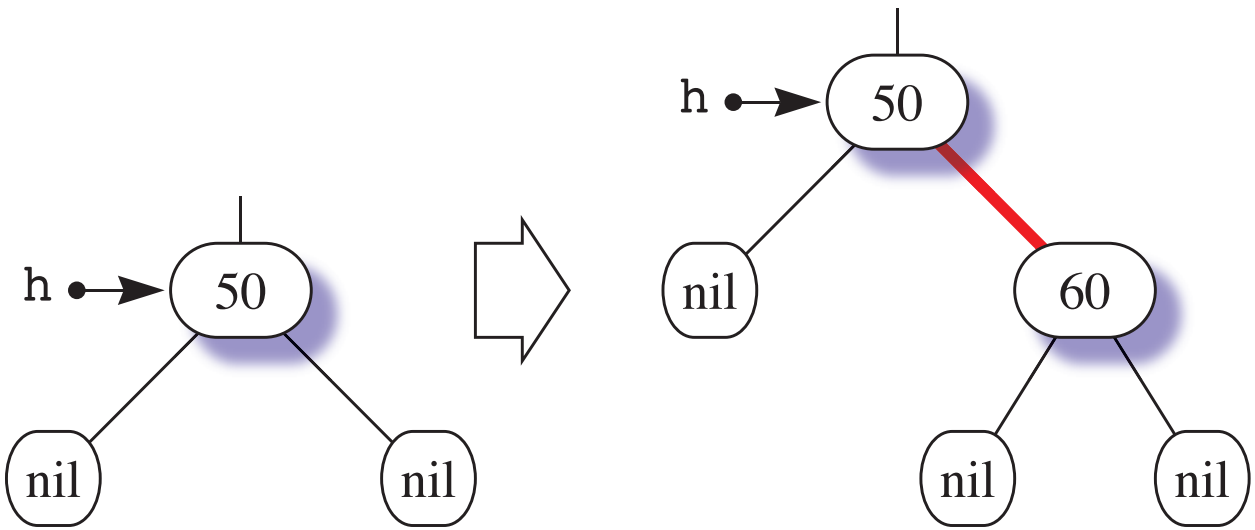
```
// ========= fixup =========
// Pre: h->_left and h->_right are roots of left-leaning red-black trees.
// Post: h is a root of a left-leaning red-black tree.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::fixup(shared_ptr<Node<T>> h) {
    if (isRed(h->_right)) {
        h = rotateLeft(h);
    }
    if (isRed(h->_left) && isRed(h->_left->_left)) {
        h = rotateRight(h);
    }
    if (isRed(h->_left) && isRed(h->_right)) {
        flipColors(h);
    }
    return h;
}
```

Inserting as a right leaf with a black sibling.



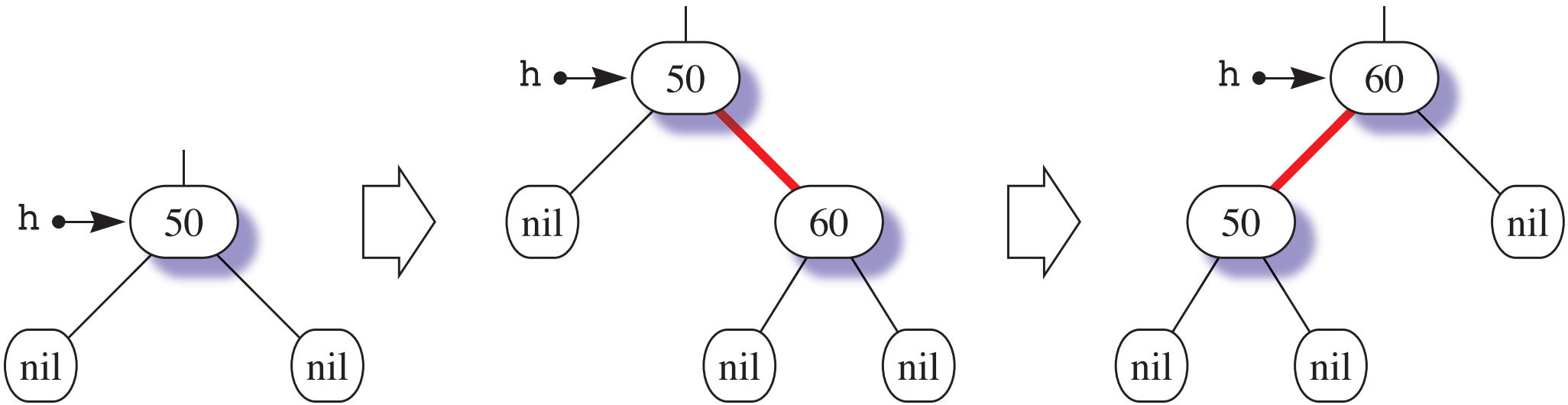
(a) Initial tree.

Inserting as a right leaf with a black sibling.



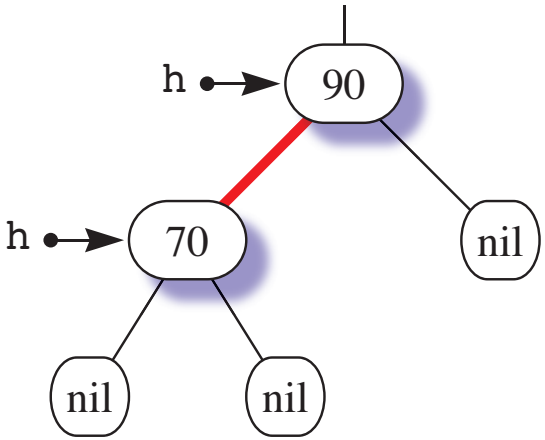
(a) Initial tree. **(b)** Insert 60.

Inserting as a right leaf with a black sibling.



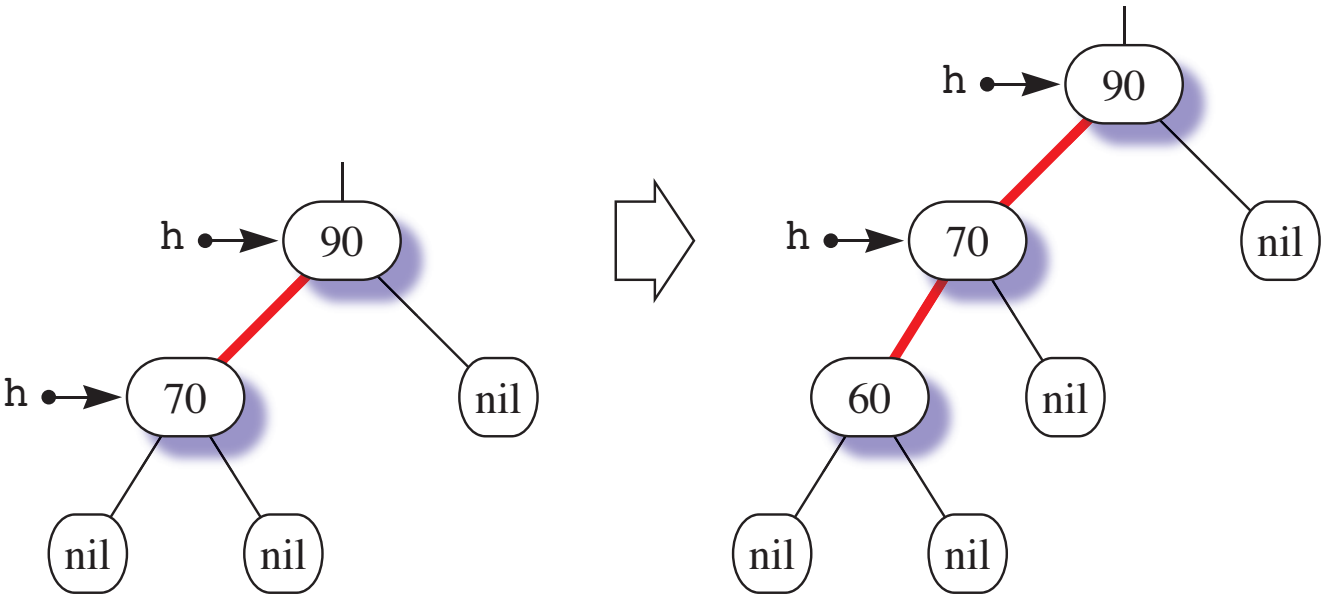
(a) Initial tree. **(b)** Insert 60. **(c)** Rotate left.

Inserting as a left leaf with a red parent.



(a) Initial tree.

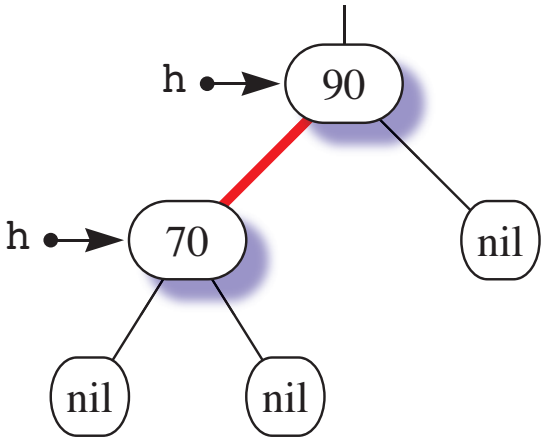
Inserting as a left leaf with a red parent.



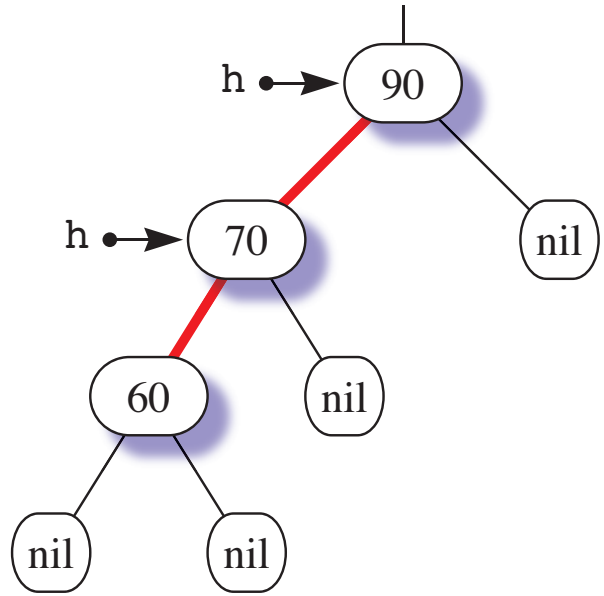
(a) Initial tree.

(b) Insert 60.

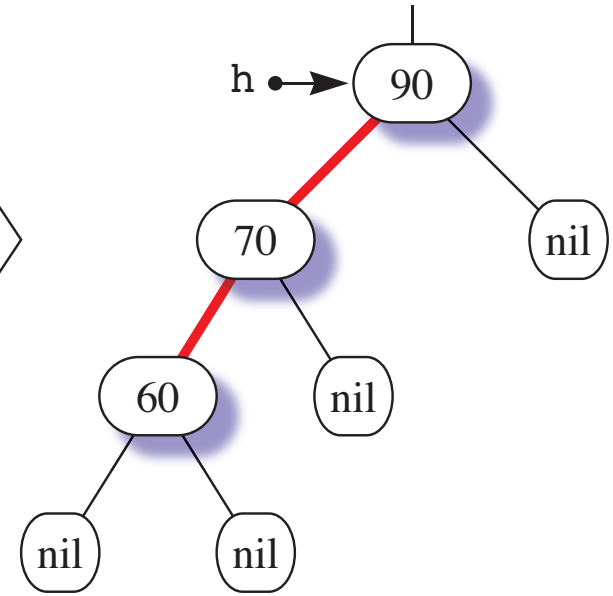
Inserting as a left leaf with a red parent.



(a) Initial tree.

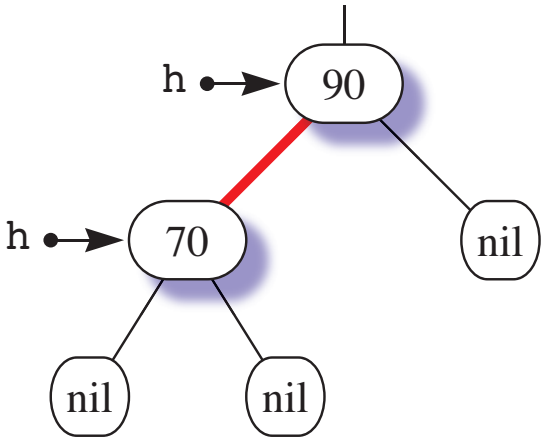


(b) Insert 60.

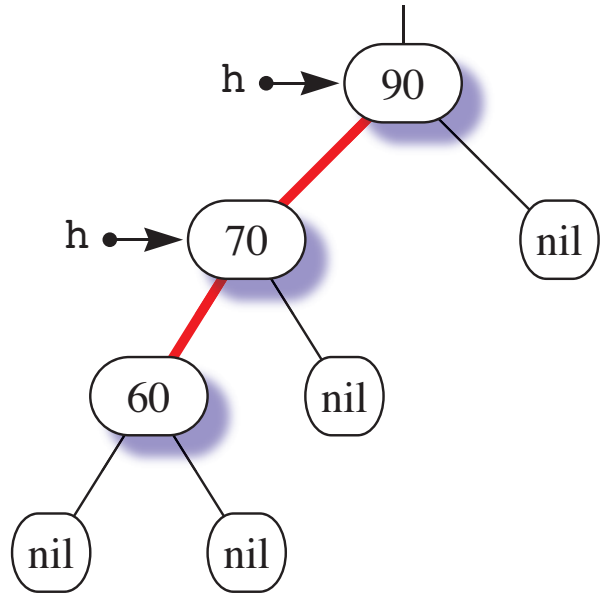


(c) No fixup.

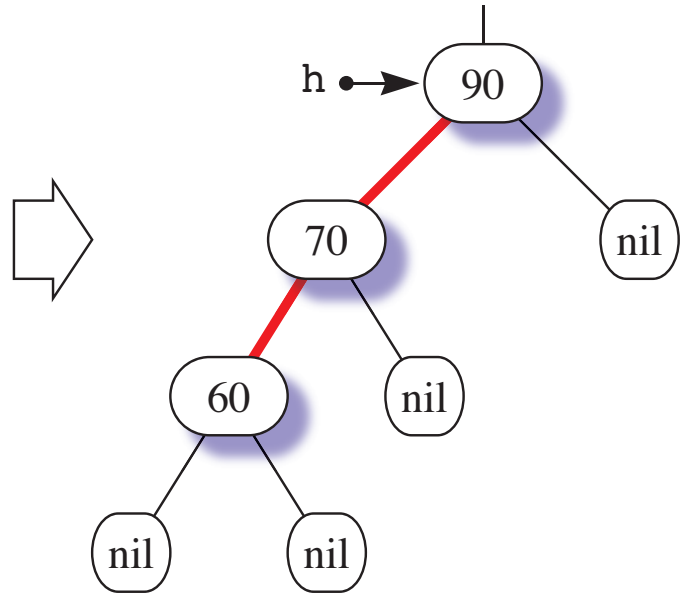
Inserting as a left leaf with a red parent.



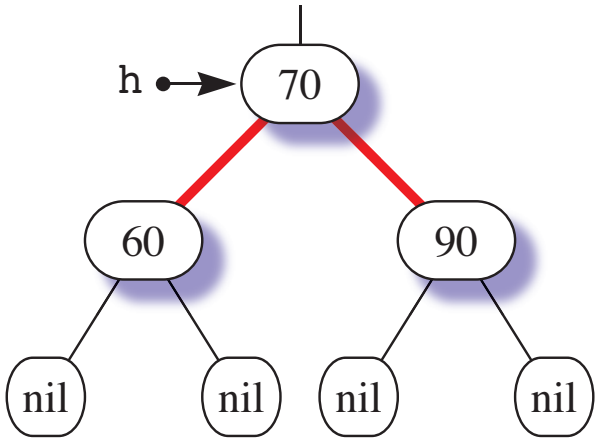
(a) Initial tree.



(b) Insert 60.

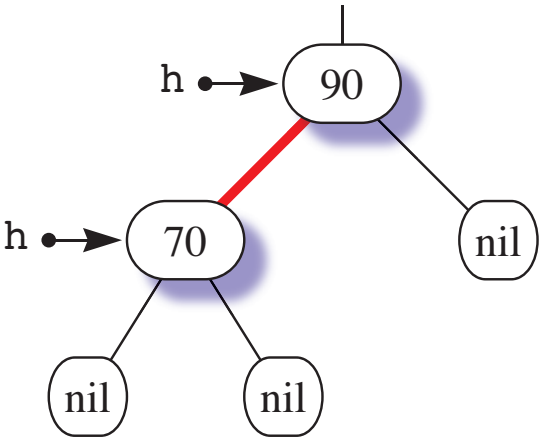


(c) No fixup.

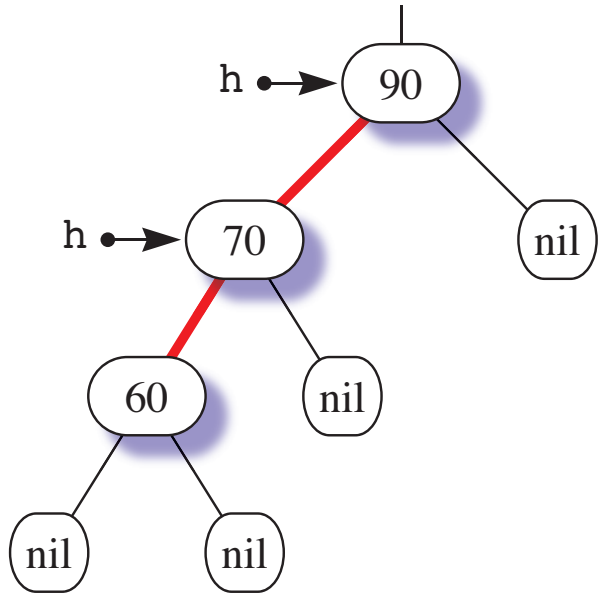


(d) Rotate right.

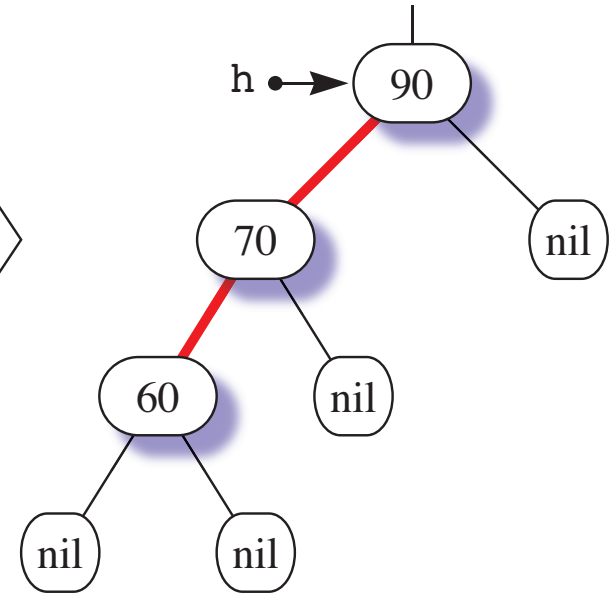
Inserting as a left leaf with a red parent.



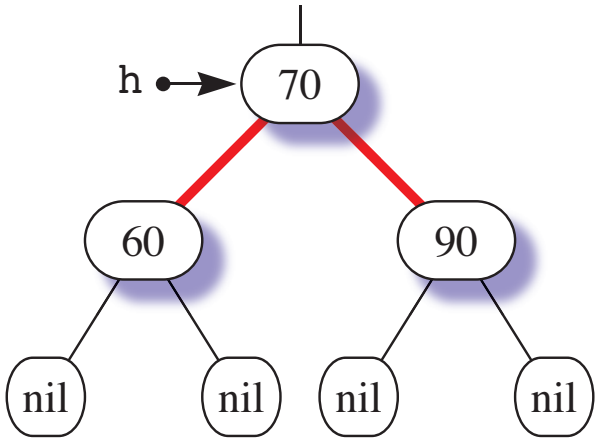
(a) Initial tree.



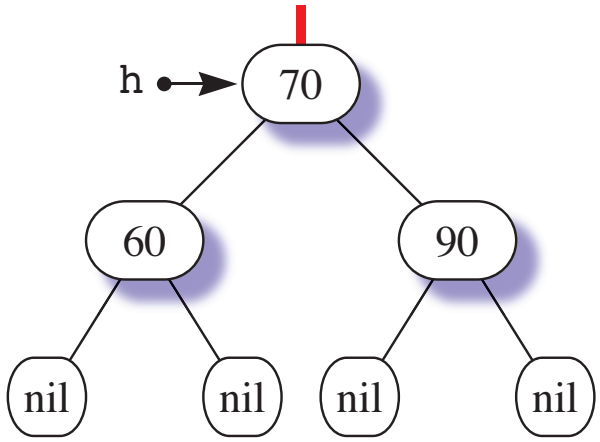
(b) Insert 60.



(c) No fixup.

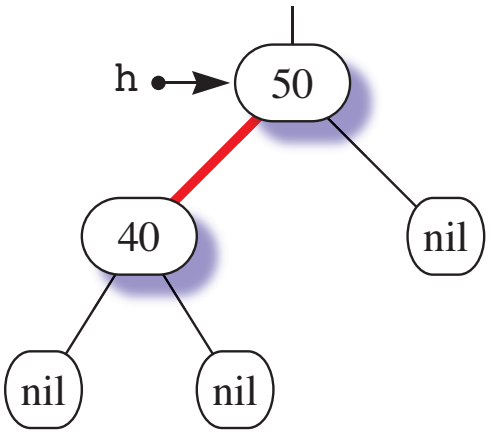


(d) Rotate right.



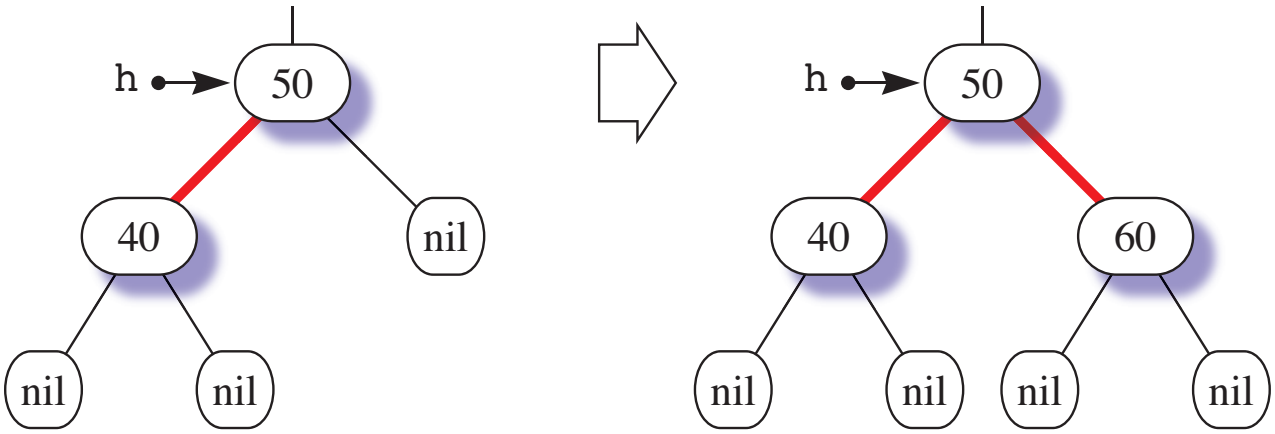
(e) Flip colors.

Inserting as a right leaf with a red sibling.



(a) Initial tree.

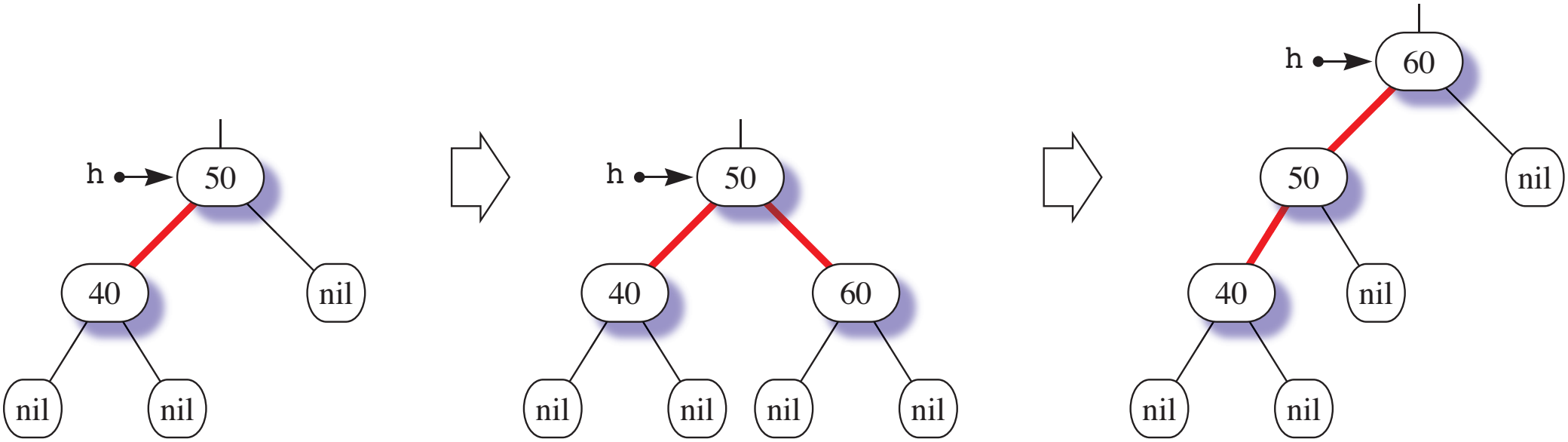
Inserting as a right leaf with a red sibling.



(a) Initial tree.

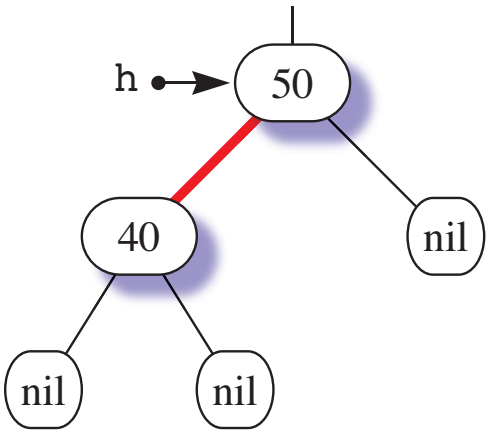
(b) Insert 60.

Inserting as a right leaf with a red sibling.

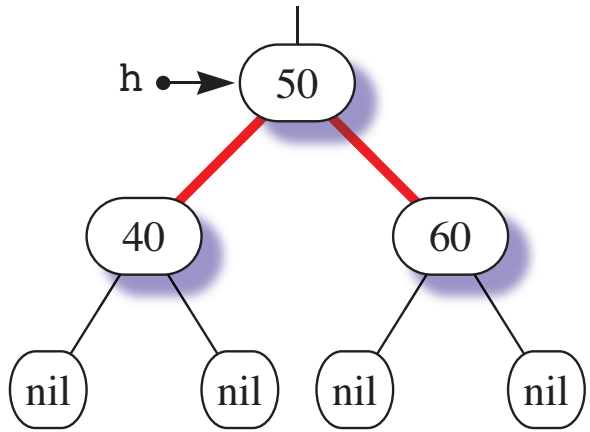


(a) Initial tree. **(b)** Insert 60. **(c)** Rotate left.

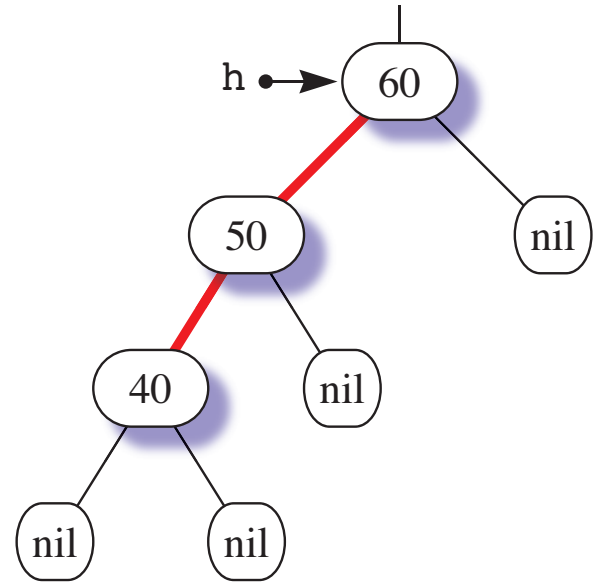
Inserting as a right leaf with a red sibling.



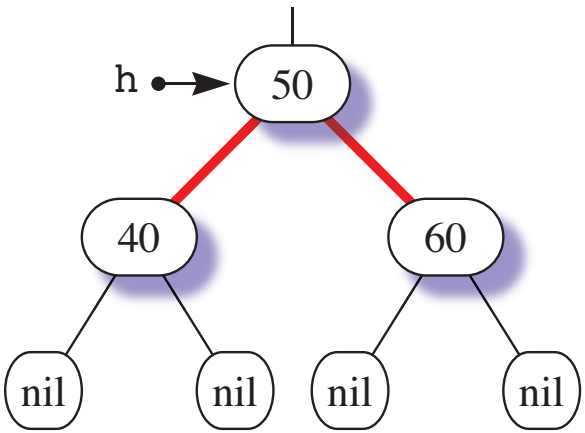
(a) Initial tree.



(b) Insert 60.

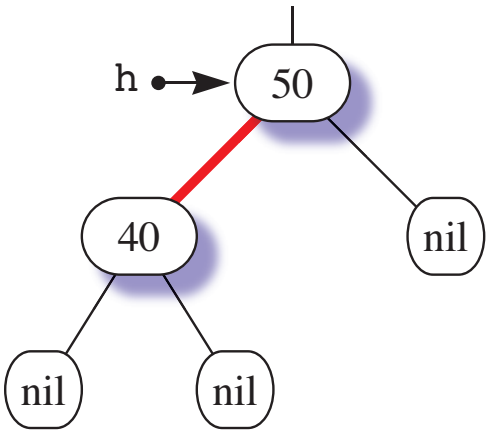


(c) Rotate left.

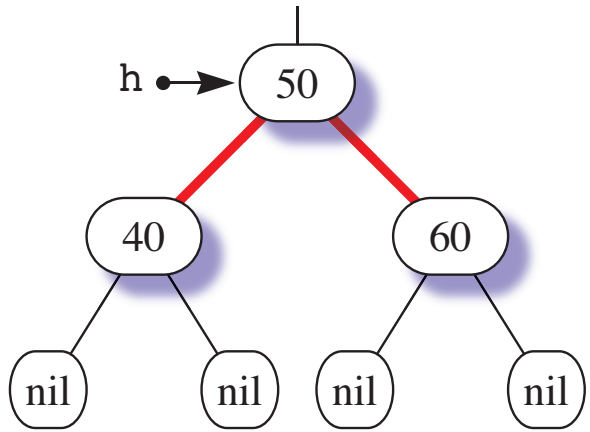


(d) Rotate right.

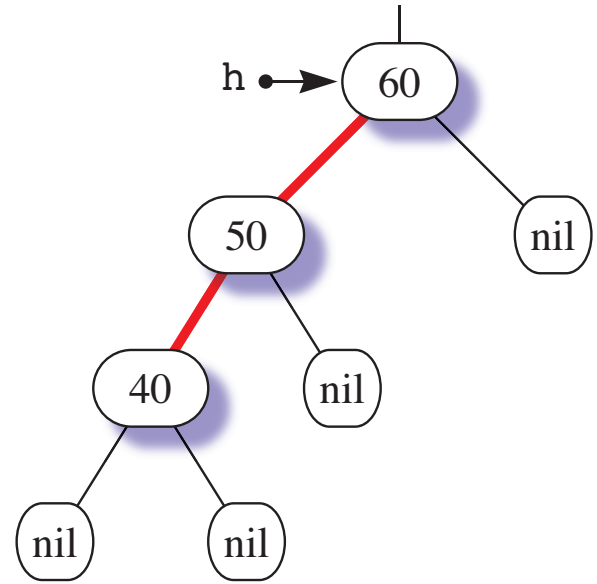
Inserting as a right leaf with a red sibling.



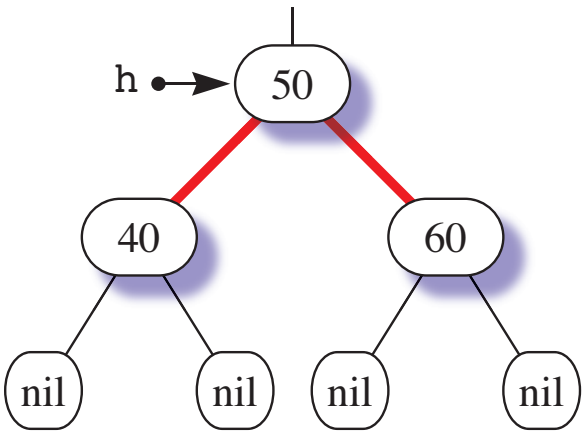
(a) Initial tree.



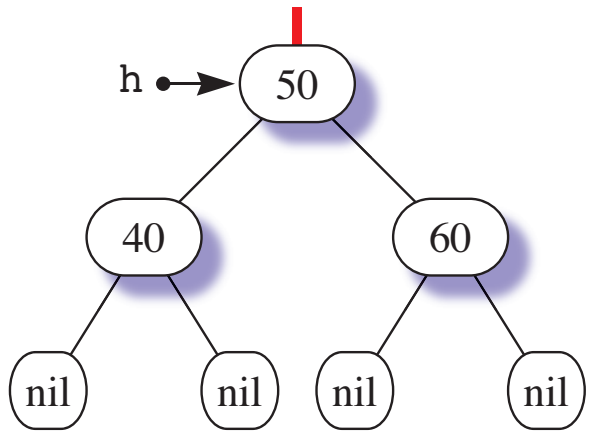
(b) Insert 60.



(c) Rotate left.

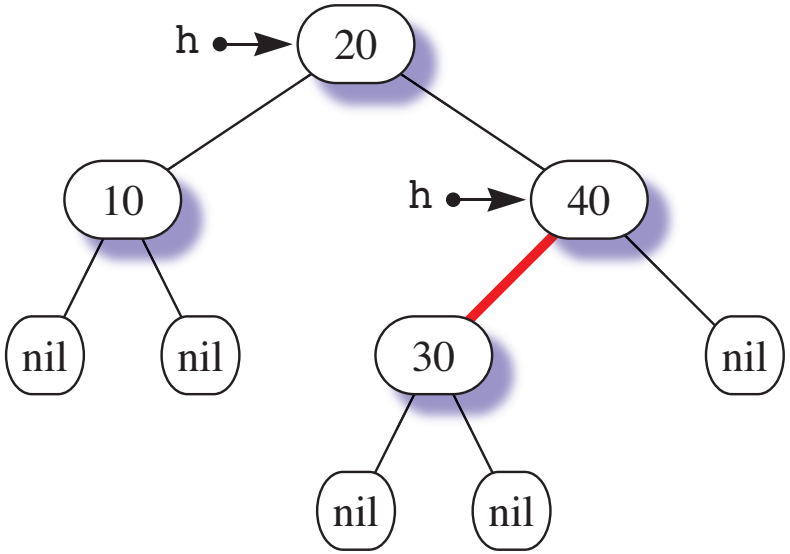


(d) Rotate right.

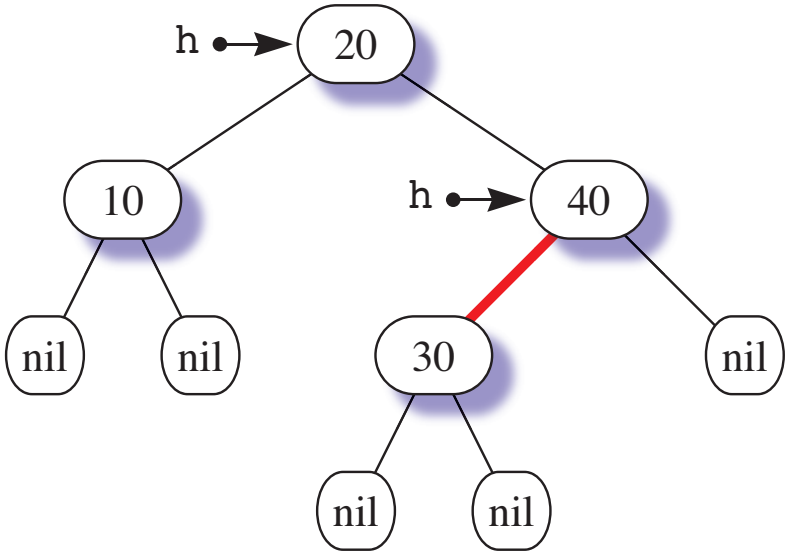


(e) Flip colors.

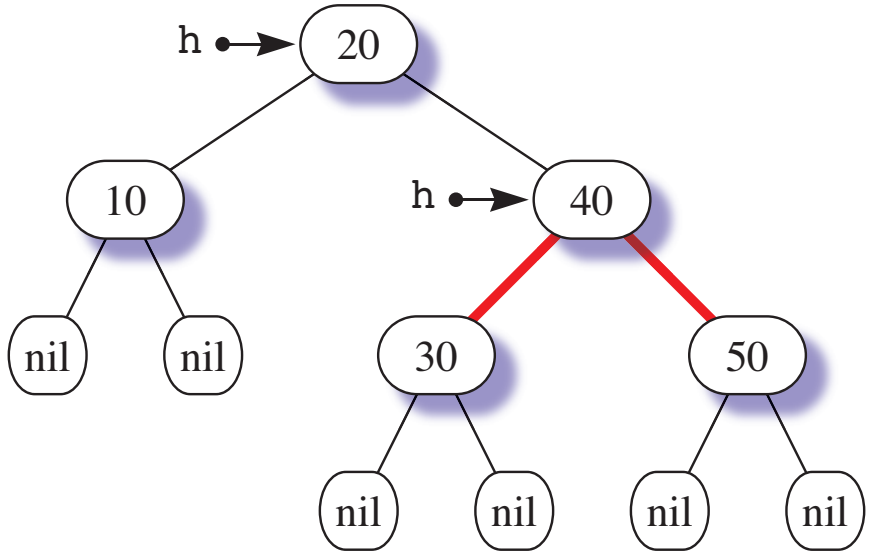
Inserting 50 into LLRBTree



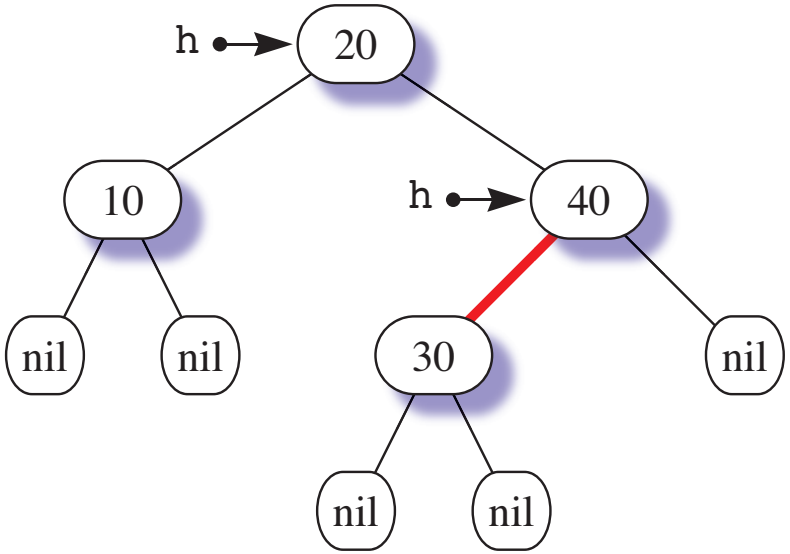
(a) Initial tree.



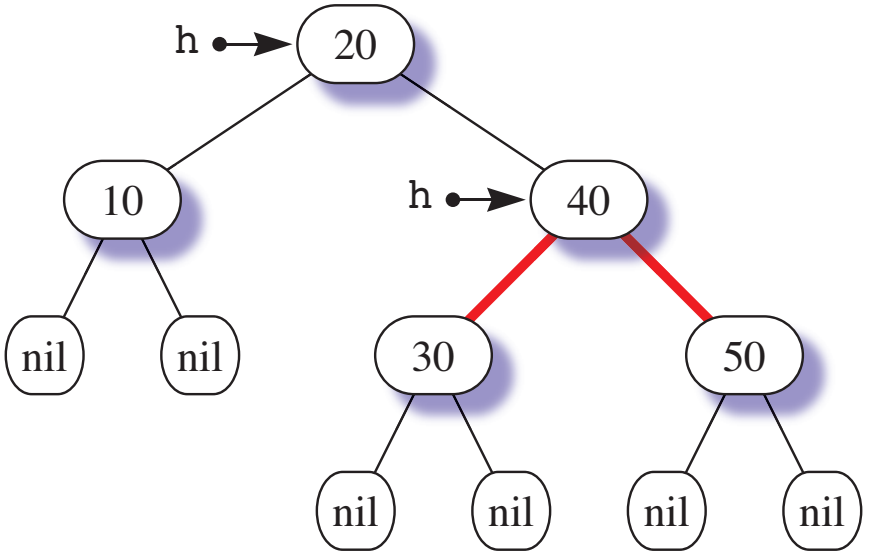
(a) Initial tree.



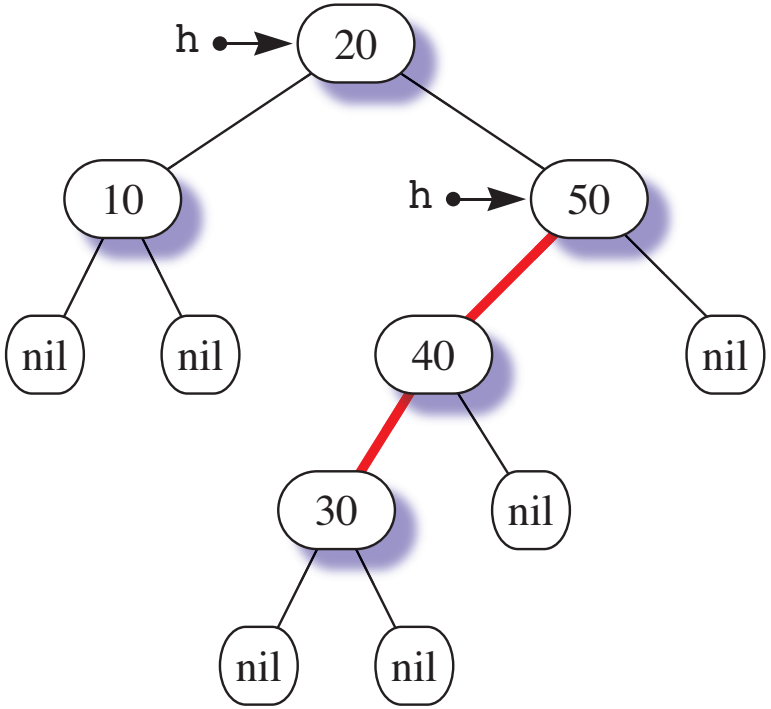
(b) `h->right = insert(h->right, data)`



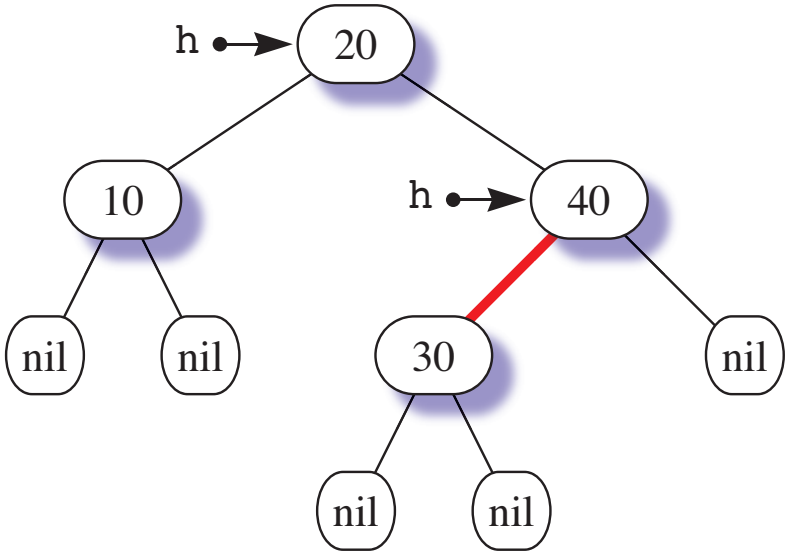
(a) Initial tree.



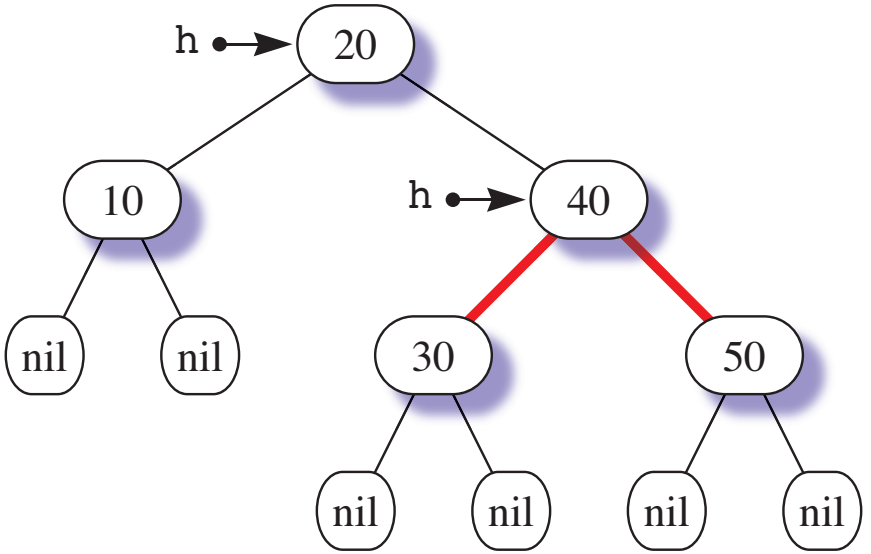
(b) `h->right = insert(h->right, data)`



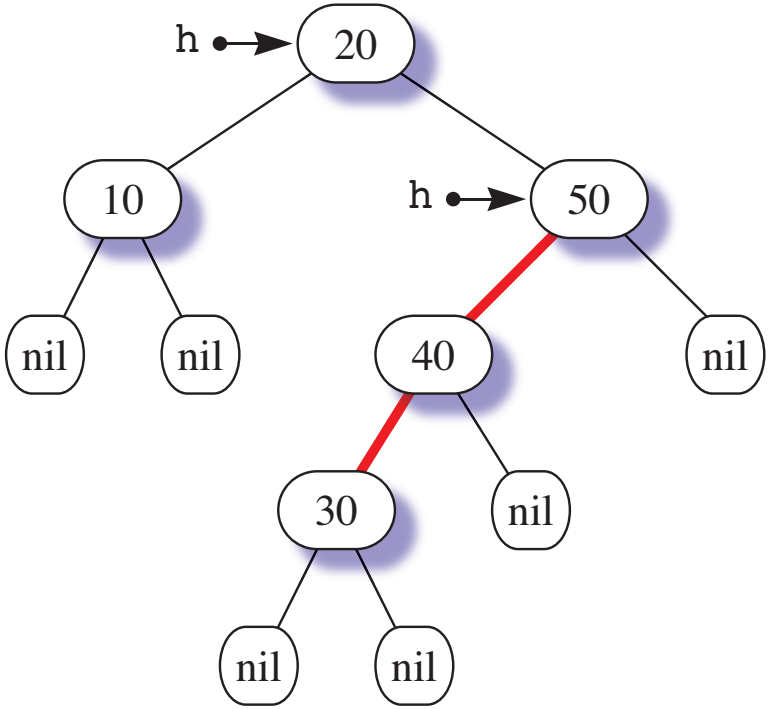
(c) `h = rotateLeft(h)`



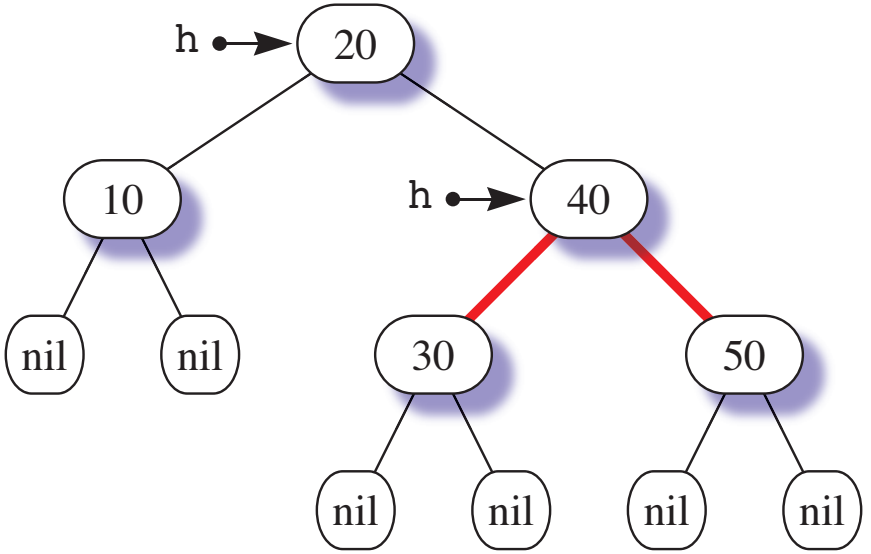
(a) Initial tree.



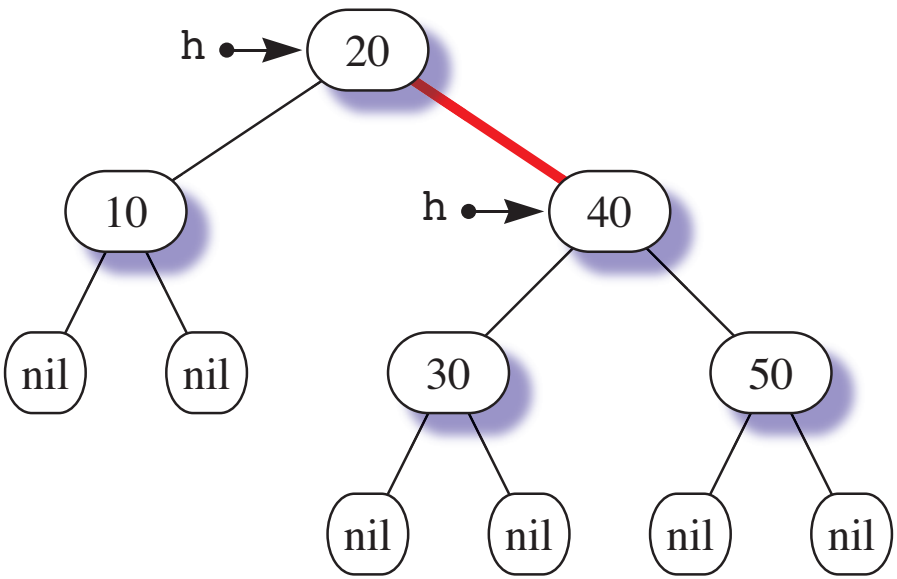
(b) `h->right = insert(h->right, data)`



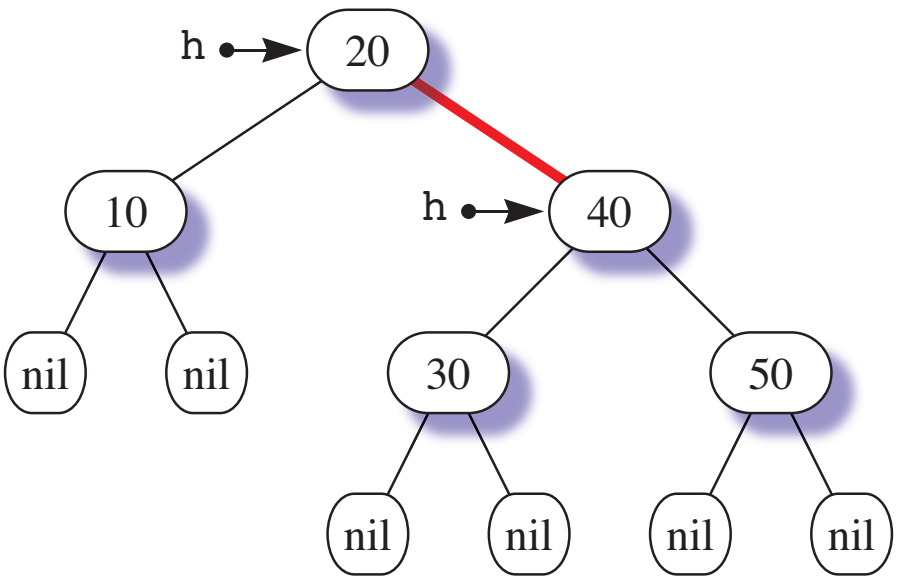
(c) `h = rotateLeft(h)`



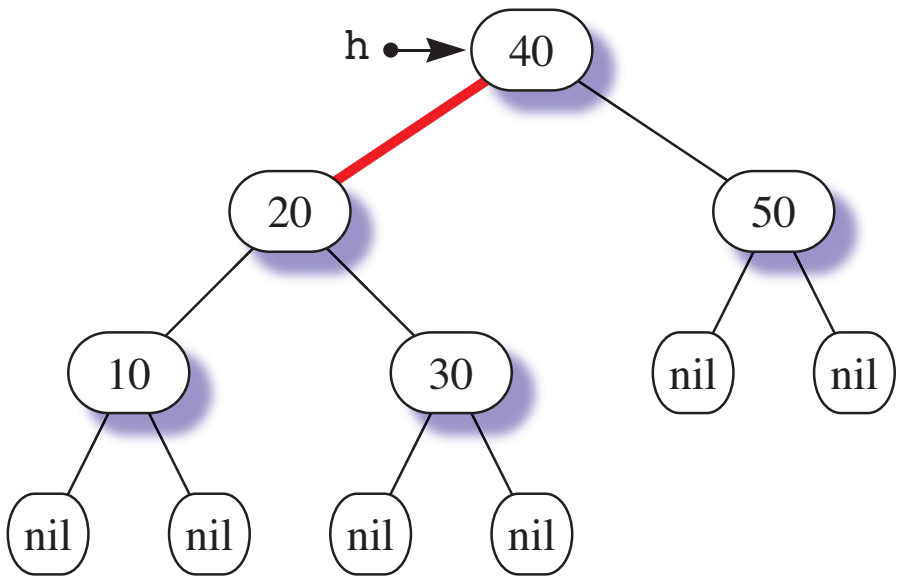
(d) `h = rotateRight(h)`



(e) `flipColors(h)`



(e) `flipColors(h)`



(f) `h = rotateLeft(h)`

Deleting from LLRBTree

The remove operation

- Find the node to remove.
- Find the minimum of the right child of the node to remove, which is the successor node.
- Copy the data from the successor to the node to remove with the `min()` operation.
- Remove the minimum of the right child.

The remove operation

- Find the node to remove.
- Find the minimum of the right child of the node to remove, which is the successor node.
- Copy the data from the successor to the node to remove with the `min()` operation.
- Remove the minimum of the right child.

Useful fact

- The minimum of a LLRBTree has a nil left child.
- Therefore, it must have a nil right child.
  - The right child cannot be red because the tree is left-leaning.
  - If the right child were not nil, the successor would not have a unique black height.
- Conclusion: We do not need a remRoot operation!
- We can simply delete the successor.

Delete minimum strategy

- If the node to be deleted has a red link to it, the node can be deleted without affecting the black height requirement.
- Problem: The node to be deleted might have a black link to it.
- Solution: Push red links down on the way to the minimum, which creates right-leaning reds.
- Fix the right-leaning reds on the way up.

The moveRedLeft operation

- The purpose of the moveRedLeft operation is to push the red links down on the way to the minimum to guarantee that the minimum will be connected to its parent with a red link.
- It is only called on the way down if there are two left black links in a row.

```
// ========= removeMin =========
// Pre: h != nullptr.
// Post: The minimum value is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::removeMin(shared_ptr<Node<T>> h) {
    if (h == nullptr) {
        cerr << "removeMin precondition violated: "
             << "h == nullptr" << endl;
        throw -1;
    }
    if (h->_left == nullptr) {
        return nullptr;
    }
    if (!isRed(h->_left) && !isRed(h->_left->_left)) {
        h = moveRedLeft(h);
    }
    h->_left = removeMin(h->_left);
    return fixup(h);
}
```

```
// ========= removeMin =========
// Pre: h != nullptr.
// Post: The minimum value is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::removeMin(shared_ptr<Node<T>> h) {
    if (h == nullptr) {
        cerr << "removeMin precondition violated: "
             << "h == nullptr" << endl;
        throw -1;
    }
    if (h->_left == nullptr) {
        return nullptr;
    }
    if (!isRed(h->_left) && !isRed(h->_left->_left)) {
        h = moveRedLeft(h);
    }
    h->_left = removeMin(h->_left);
    return fixup(h);
}
```

Guaranteed to have red link from parent.

```
// ========= removeMin =========
// Pre: h != nullptr.
// Post: The minimum value is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::removeMin(shared_ptr<Node<T>> h) {
    if (h == nullptr) {
        cerr << "removeMin precondition violated: "
             << "h == nullptr" << endl;
        throw -1;
    }
    if (h->_left == nullptr) {
        return nullptr;
    }
    if (!isRed(h->_left) && !isRed(h->_left->_left)) {
        h = moveRedLeft(h);
    }
    h->_left = removeMin(h->_left);
    return fixup(h);
}
```

Called on the way down if two left black links in a row

```
// ========= removeMin =========
// Pre: h != nullptr.
// Post: The minimum value is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::removeMin(shared_ptr<Node<T>> h) {
    if (h == nullptr) {
        cerr << "removeMin precondition violated: "
             << "h == nullptr" << endl;
        throw -1;
    }
    if (h->_left == nullptr) {
        return nullptr;
    }
    if (!isRed(h->_left) && !isRed(h->_left->_left)) {
        h = moveRedLeft(h);
    }
    h->_left = removeMin(h->_left);
    return fixup(h);
}
```

Recursively move down the left to find the minimum.

```
// ========= removeMin =========
// Pre: h != nullptr.
// Post: The minimum value is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::removeMin(shared_ptr<Node<T>> h) {
    if (h == nullptr) {
        cerr << "removeMin precondition violated: "
             << "h == nullptr" << endl;
        throw -1;
    }
    if (h->_left == nullptr) {
        return nullptr;
    }
    if (!isRed(h->_left) && !isRed(h->_left->_left)) {
        h = moveRedLeft(h);
    }
    h->_left = removeMin(h->_left);
    return fixup(h);
}
```

Same fixup as with insert!

```
// ========= moveRedLeft =========
// Pre: Both h->_left and h->_left->_left are BLACK.
// Post: Either h->_left or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::moveRedLeft(shared_ptr<Node<T>> h) {
    if (isRed(h->_left) || isRed(h->_left->_left)) {
        cerr << "moveRedLeft precondition violated: "
             << "h->_left is RED or h->_left->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_right->_left)) {
        h->_right = rotateRight(h->_right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}
```

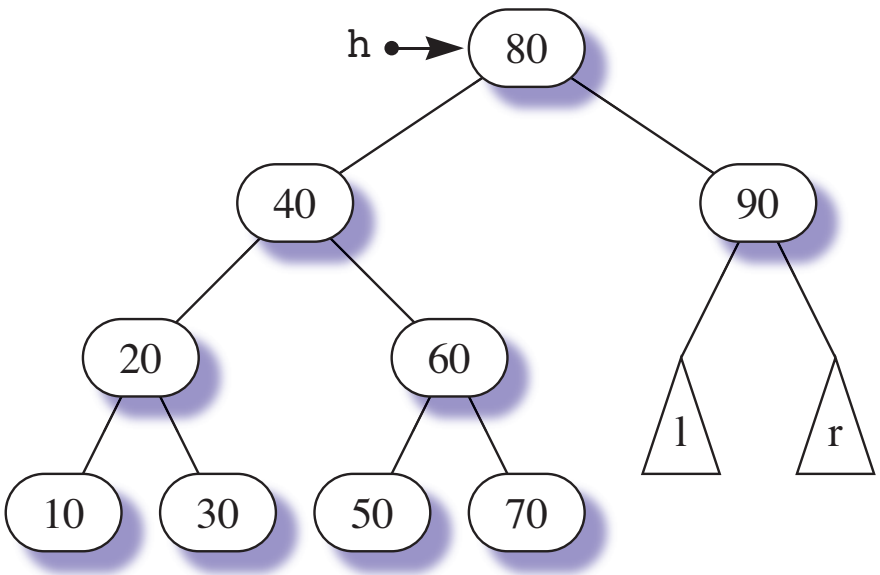
```
// ========= moveRedLeft =========
// Pre: Both h->_left and h->_left->_left are BLACK.
// Post: Either h->_left or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::moveRedLeft(shared_ptr<Node<T>> h) {
    if (isRed(h->_left) || isRed(h->_left->_left)) {
        cerr << "moveRedLeft precondition violated: "
             << "h->_left is RED or h->_left->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_right->_left)) {
        h->_right = rotateRight(h->_right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}
```

Move red left might only flip the colors.

```
// ========= moveRedLeft =========
// Pre: Both h->_left and h->_left->_left are BLACK.
// Post: Either h->_left or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::moveRedLeft(shared_ptr<Node<T>> h) {
    if (isRed(h->_left) || isRed(h->_left->_left)) {
        cerr << "moveRedLeft precondition violated: "
             << "h->_left is RED or h->_left->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_right->_left)) {
        h->_right = rotateRight(h->_right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}
```

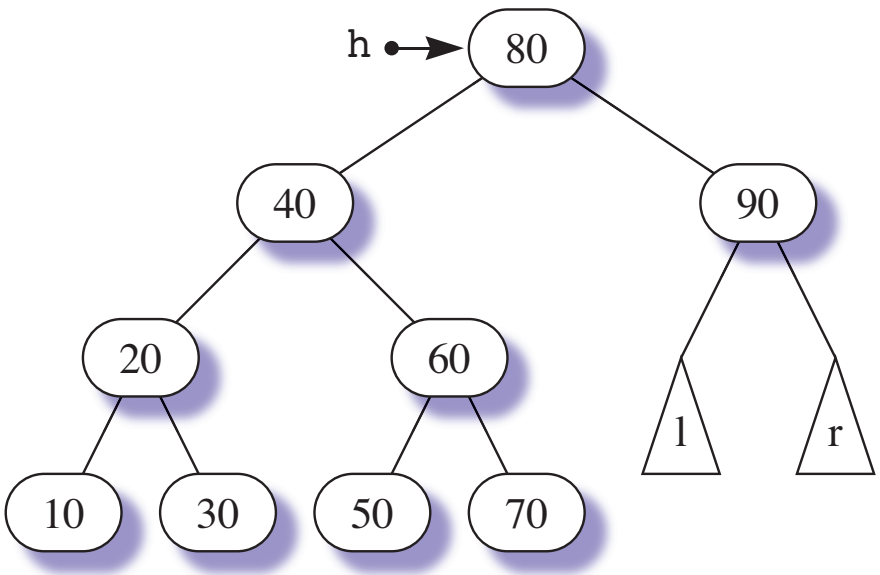
This condition could not be repaired by fixup on the way back up.

Deleting minimum. Move red left only flips colors.

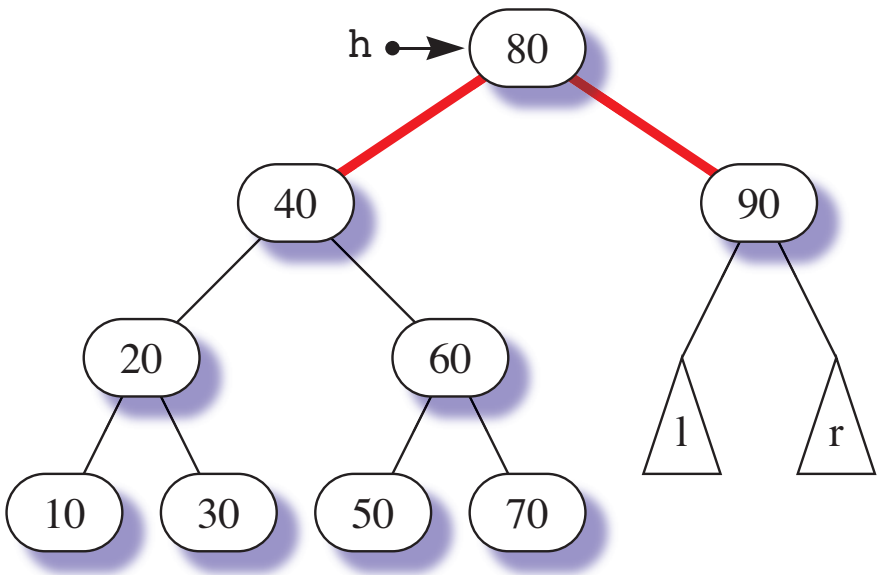


(a) Initial tree.

Deleting minimum. Move red left only flips colors.

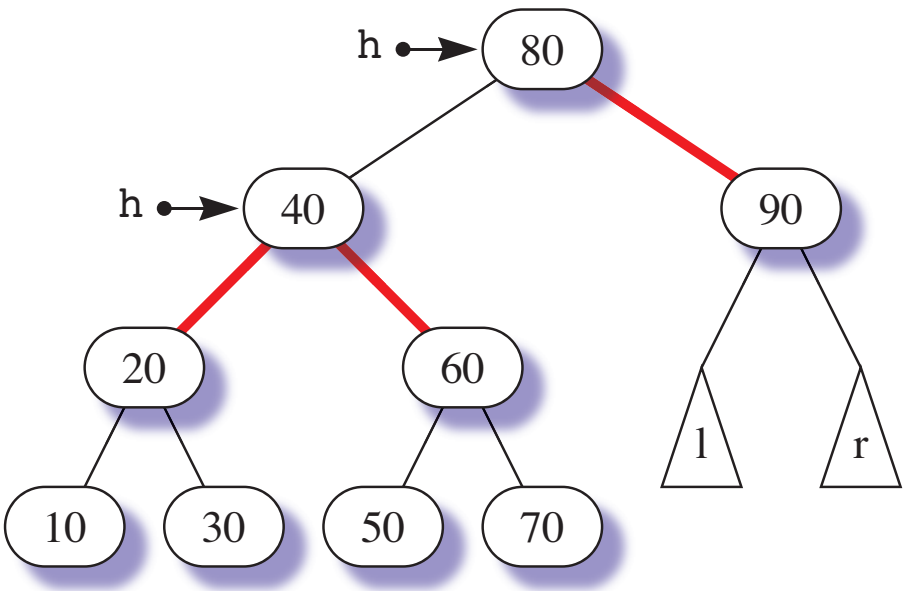


(a) Initial tree.



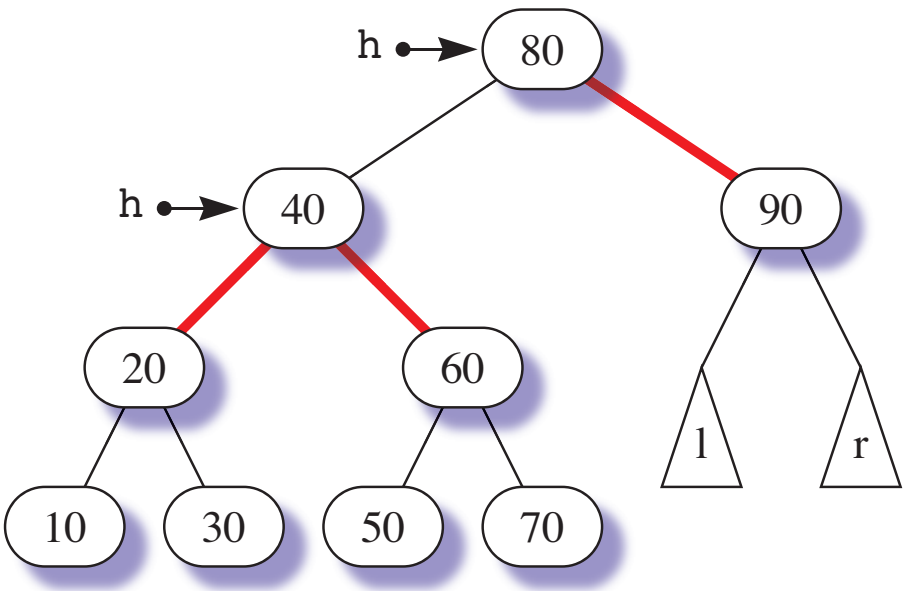
(b) `moveRedLeft()` only flips colors.

Deleting minimum. Move red left only flips colors.

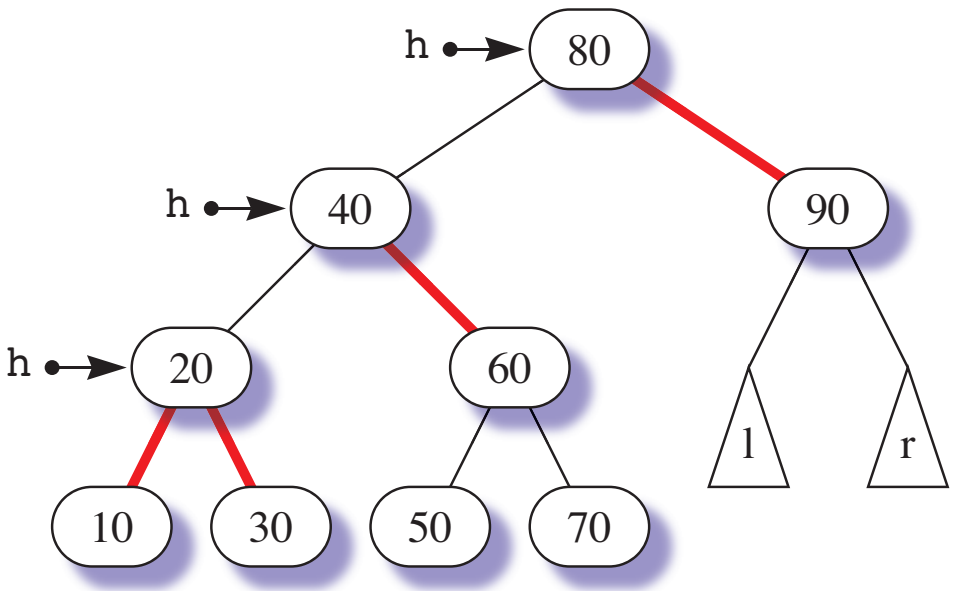


(c) `moveRedLeft()` only flips colors.

Deleting minimum. Move red left only flips colors.

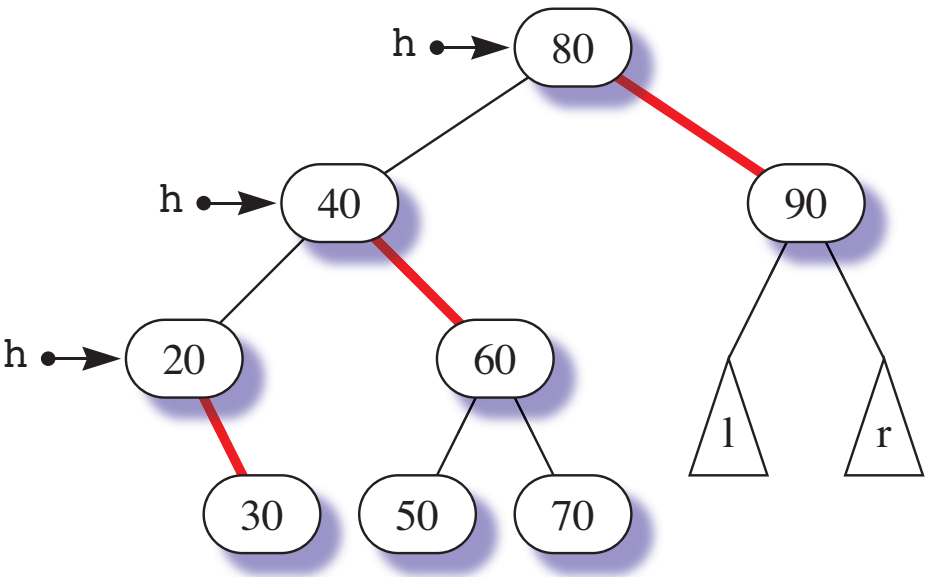


(c) `moveRedLeft()` only flips colors.



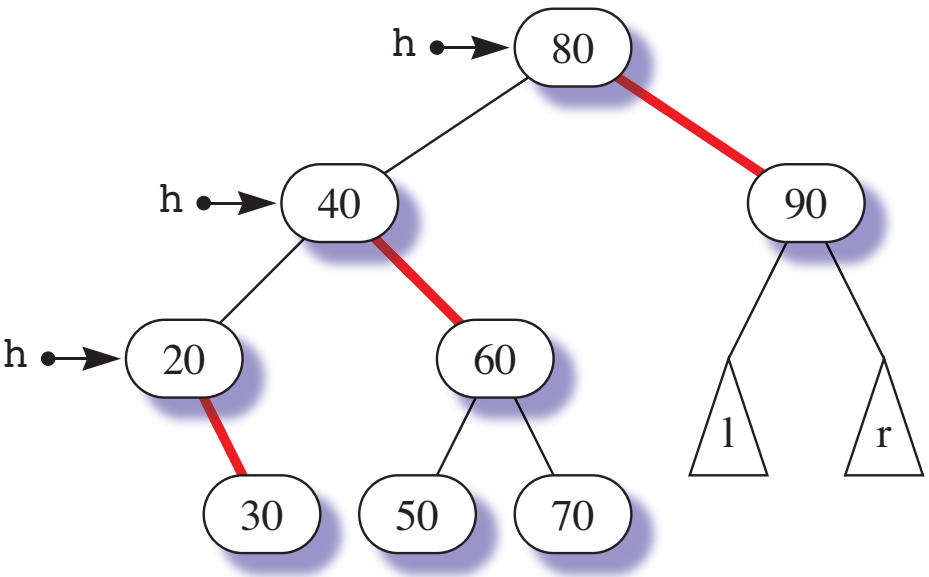
(d) `moveRedLeft()` only flips colors.

Deleting minimum. Move red left only flips colors.

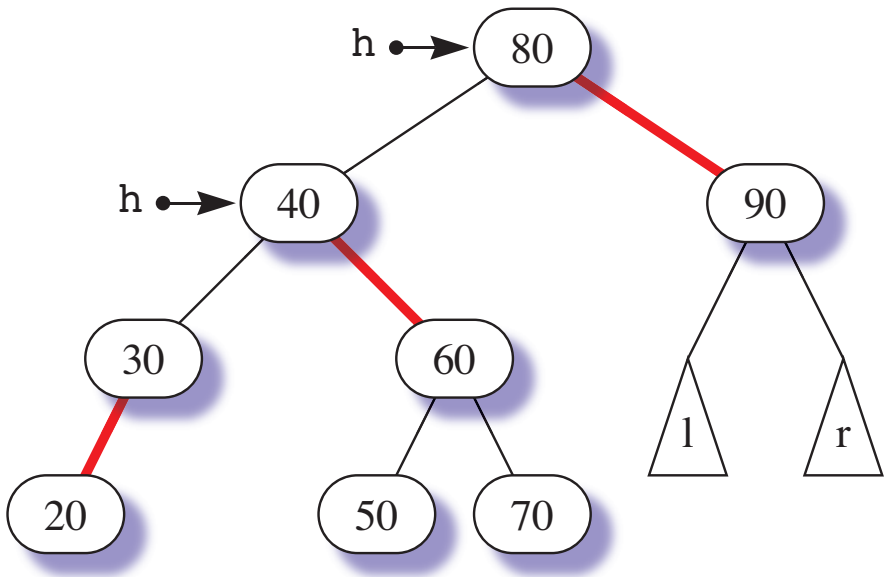


(e) Delete minimum with red parent link.

Deleting minimum. Move red left only flips colors.

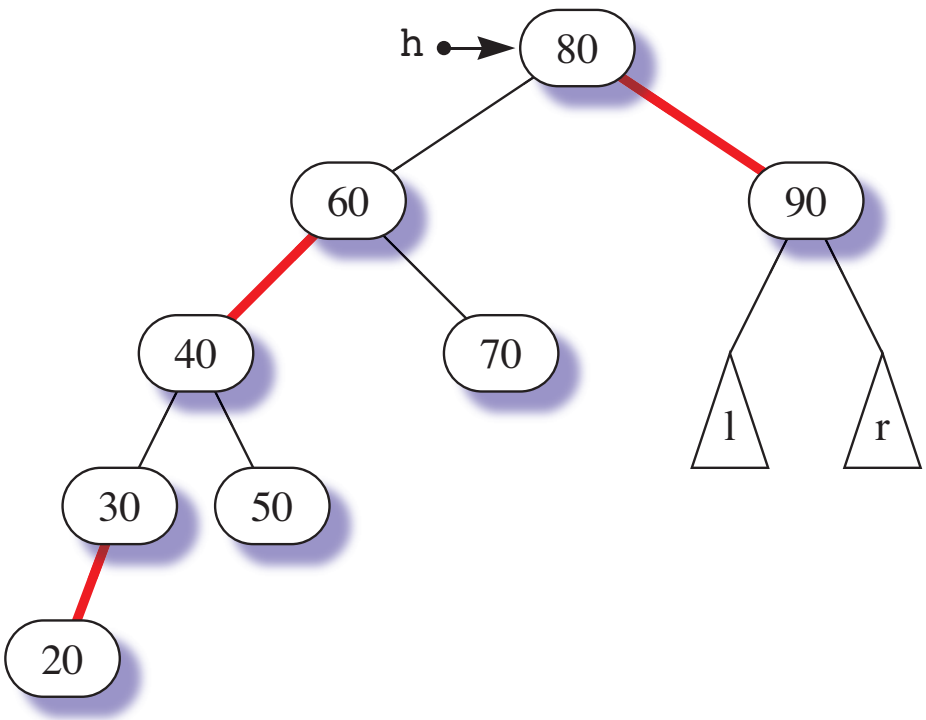


(e) Delete minimum with red parent link.



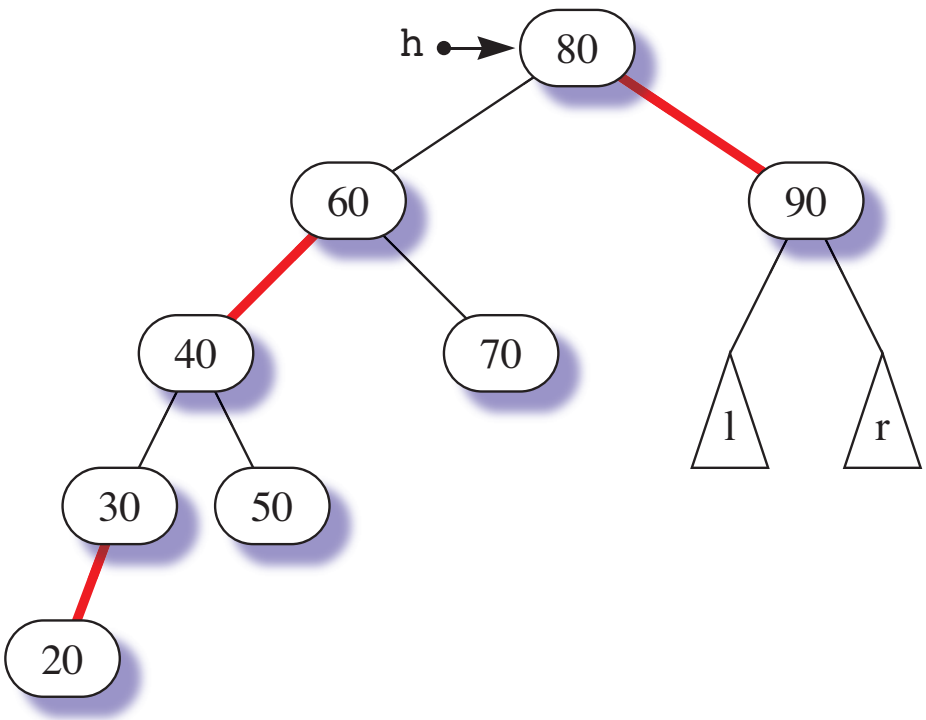
(f) Fixup with `rotateLeft()`.

Deleting minimum. Move red left only flips colors.

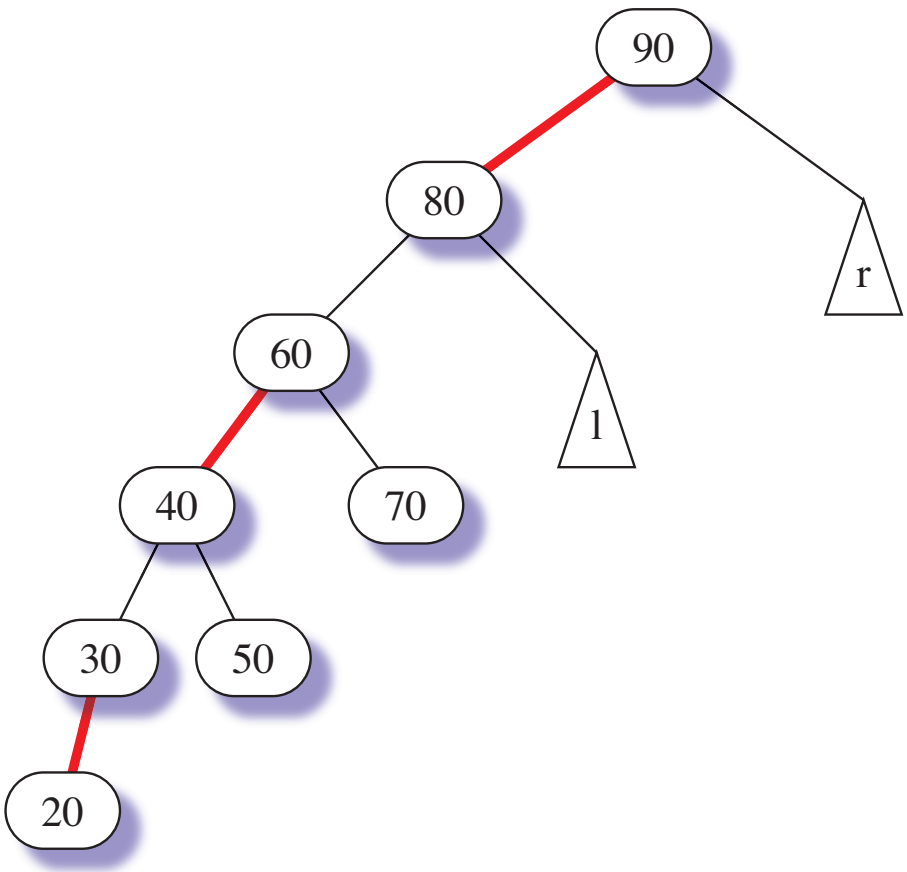


(g) Fixup with `rotateLeft()`.

Deleting minimum. Move red left only flips colors.

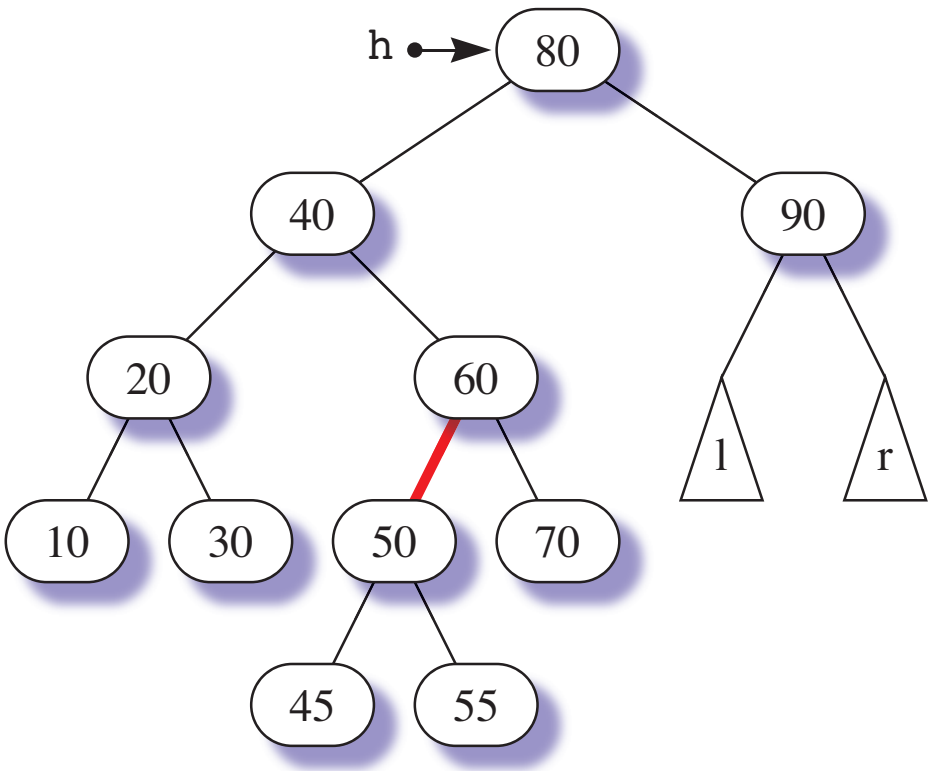


(g) Fixup with `rotateLeft()`.



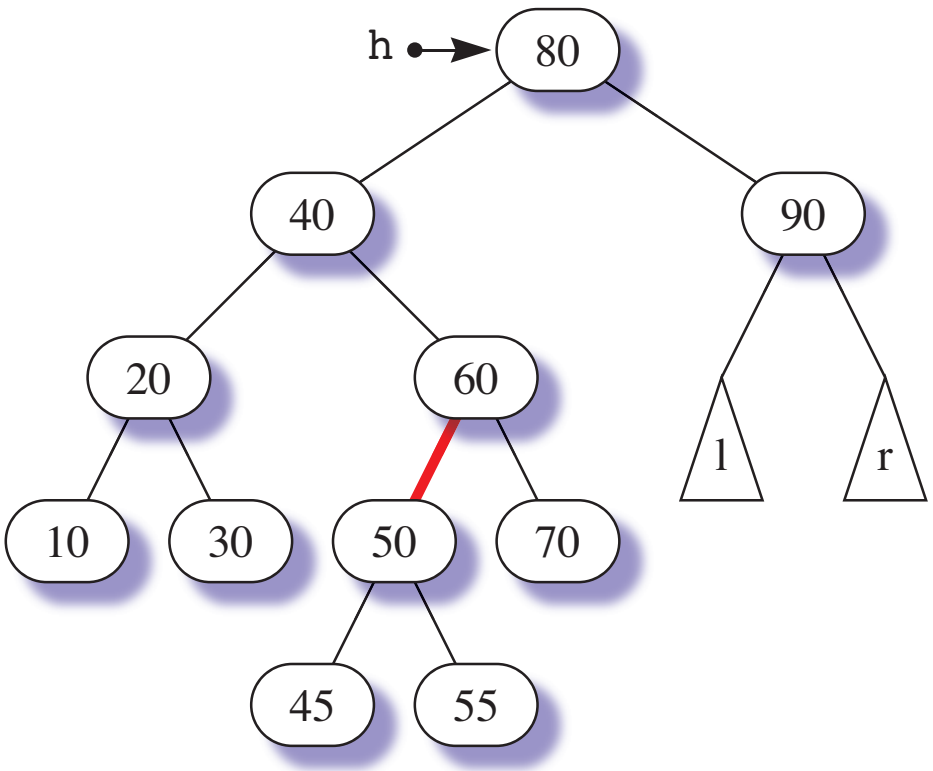
(h) Fixup with `rotateLeft()`.

Deleting minimum.
Move red left does more than flipping colors.

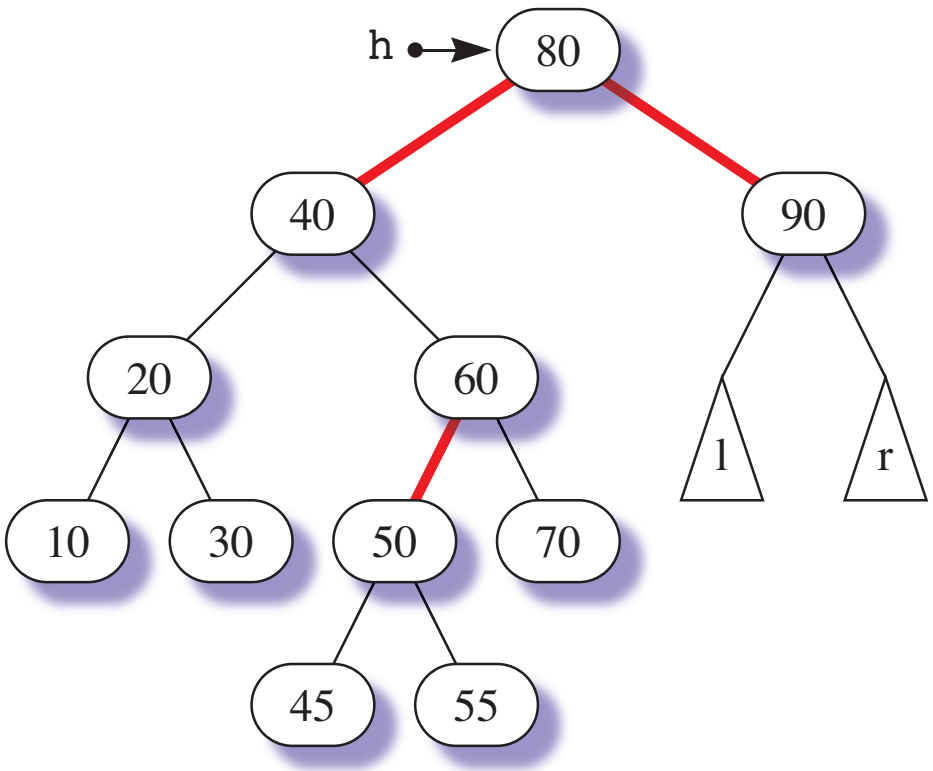


(a) Initial tree.

Deleting minimum.
Move red left does more than flipping colors.

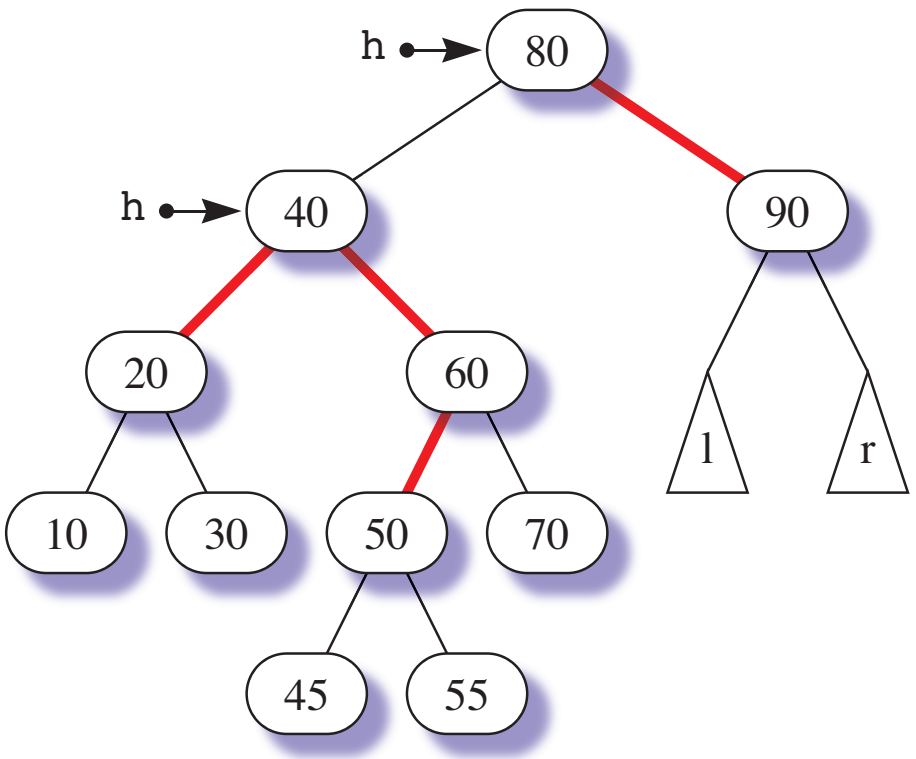


(a) Initial tree.



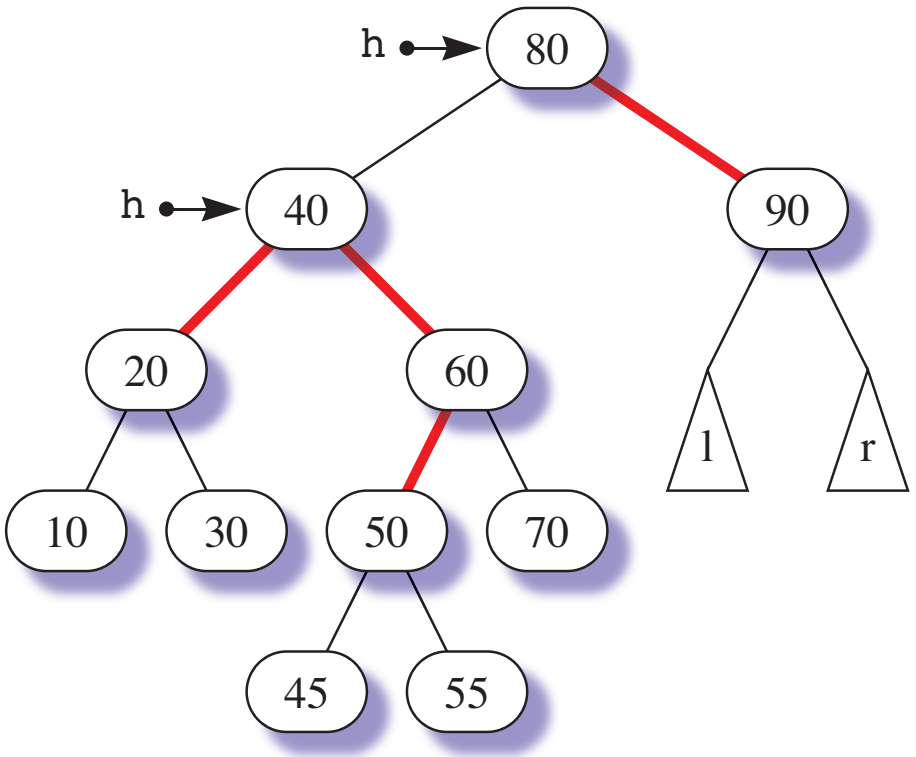
(b) `moveRedLeft()` only flips colors.

Deleting minimum.
Move red left does more than flipping colors.

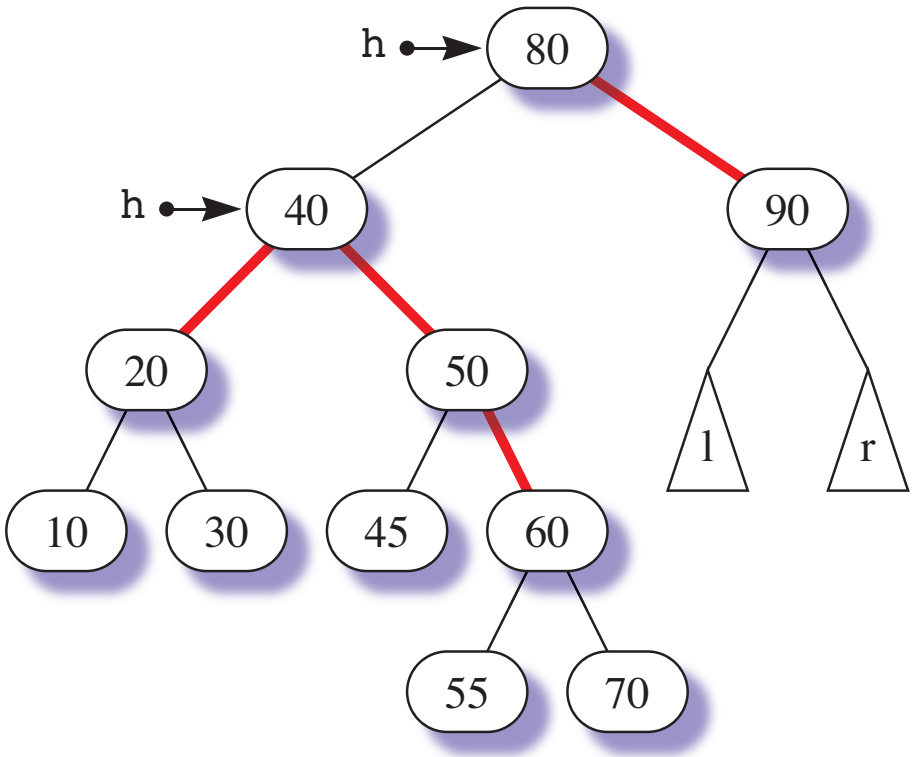


(c) First, `moveRedLeft()` flips colors.

Deleting minimum.
Move red left does more than flipping colors.

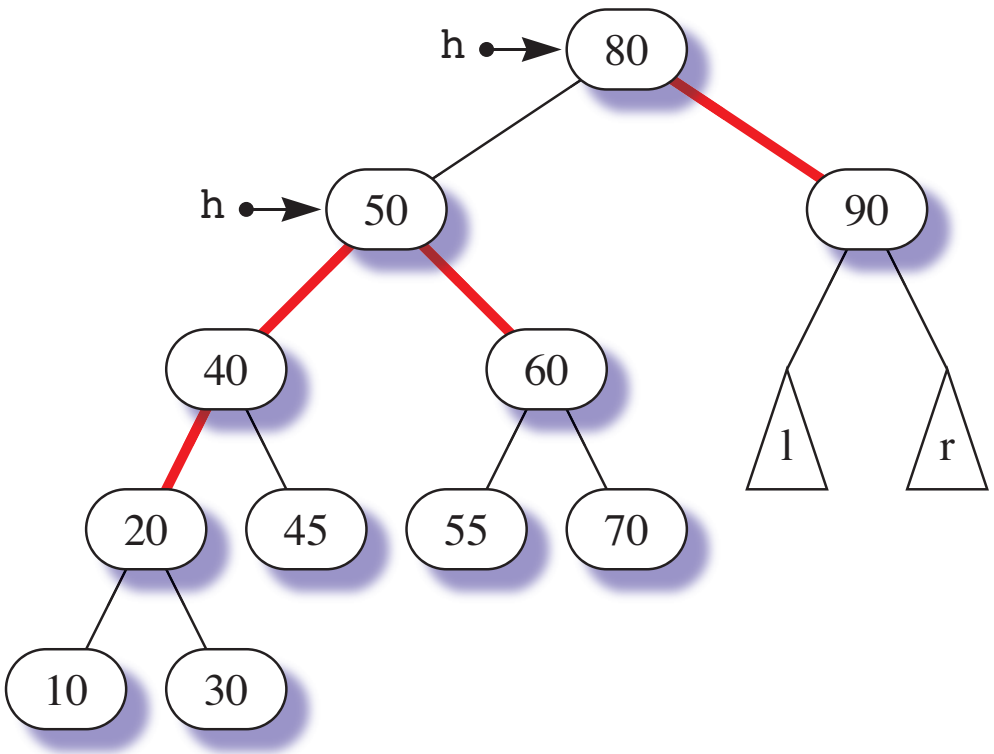


(c) First, `moveRedLeft()` flips colors.



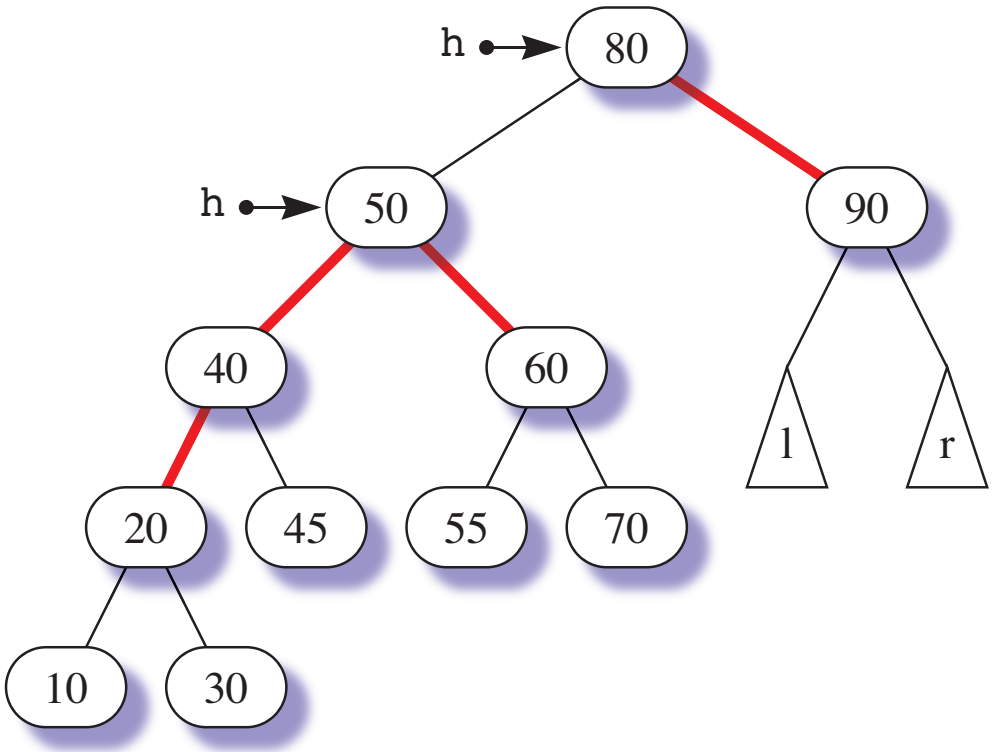
(d) `h->_right = rotateRight(h->_right)`.

Deleting minimum.
Move red left does more than flipping colors.

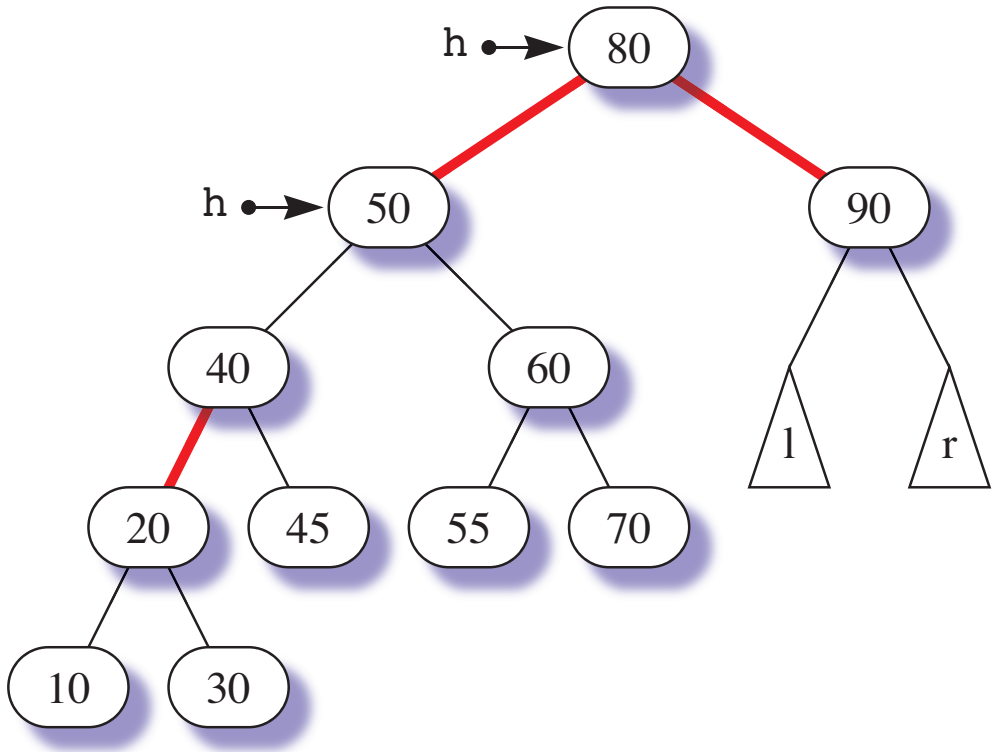


(e) `h = rotateLeft(h);`

Deleting minimum.
Move red left does more than flipping colors.

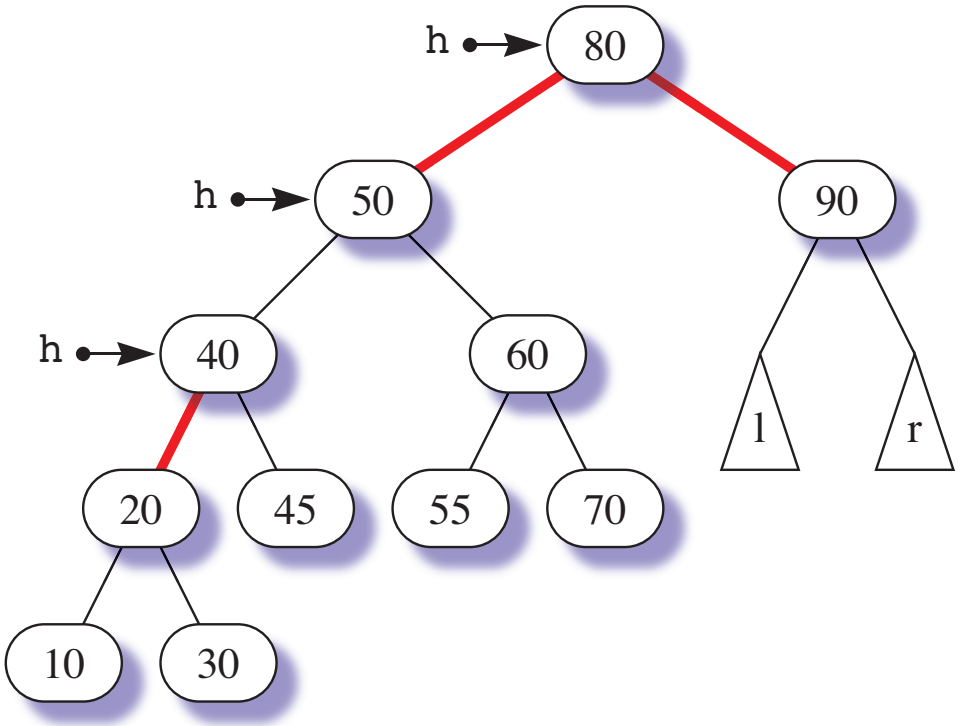


(e) `h = rotateLeft(h);`



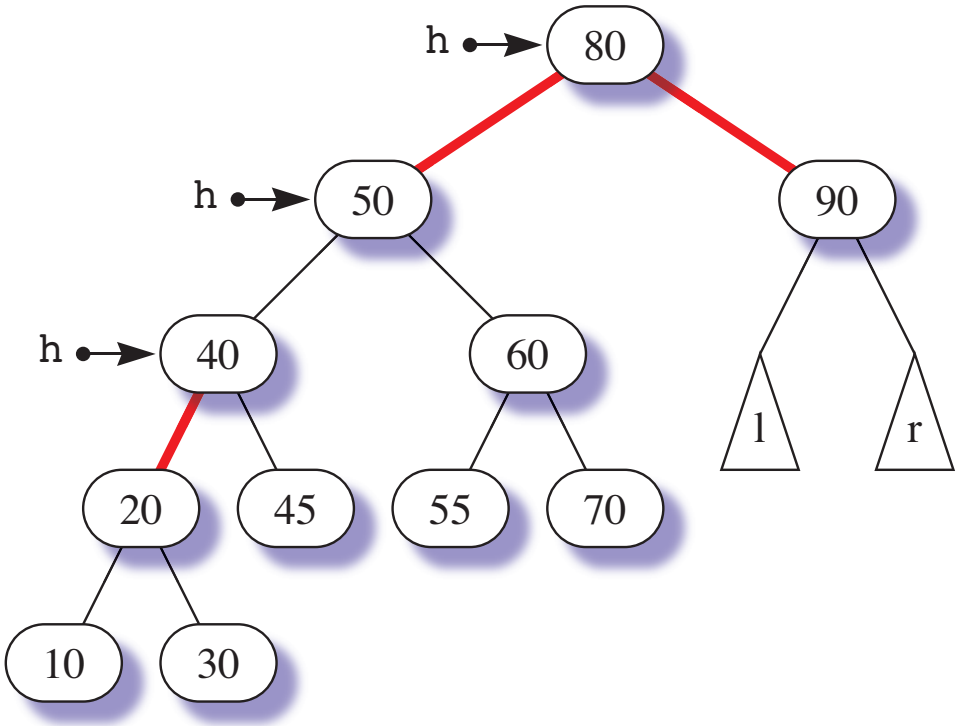
(f) `h = flipColors(h);`

Deleting minimum.
Move red left does more than flipping colors.

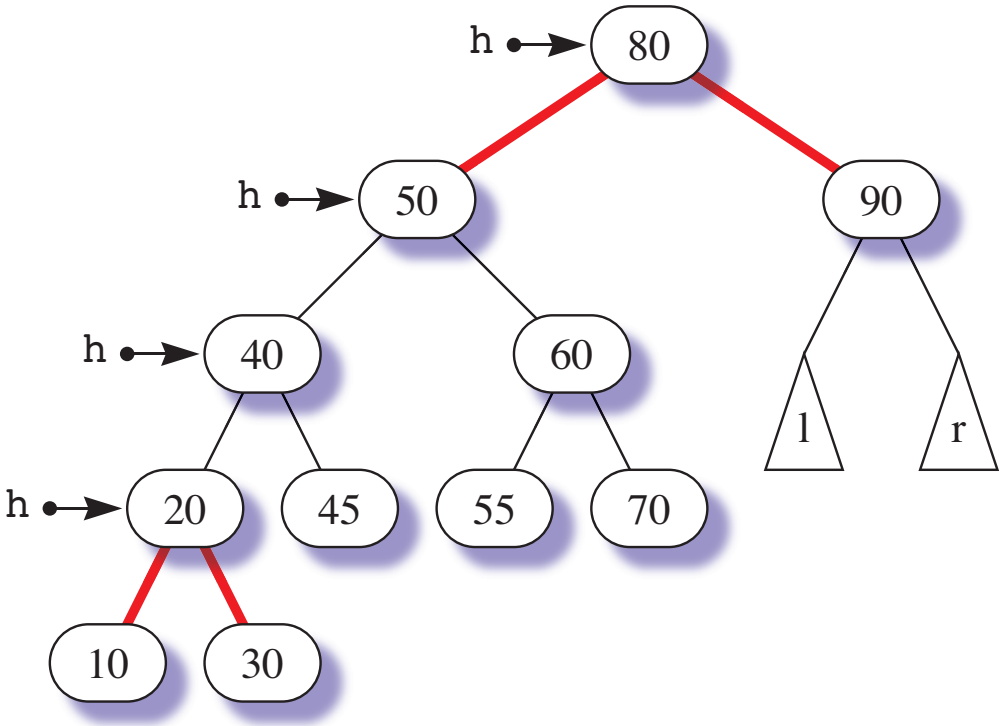


(g) `moveRedLeft()` is not called.

Deleting minimum.
Move red left does more than flipping colors.

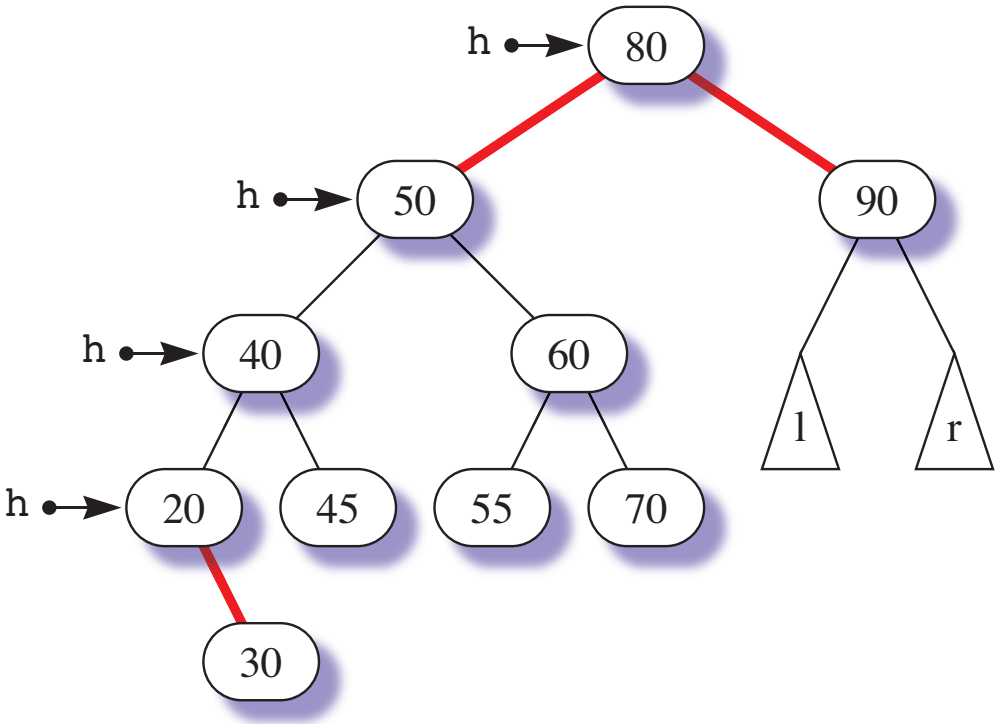


(g) `moveRedLeft()` is not called.



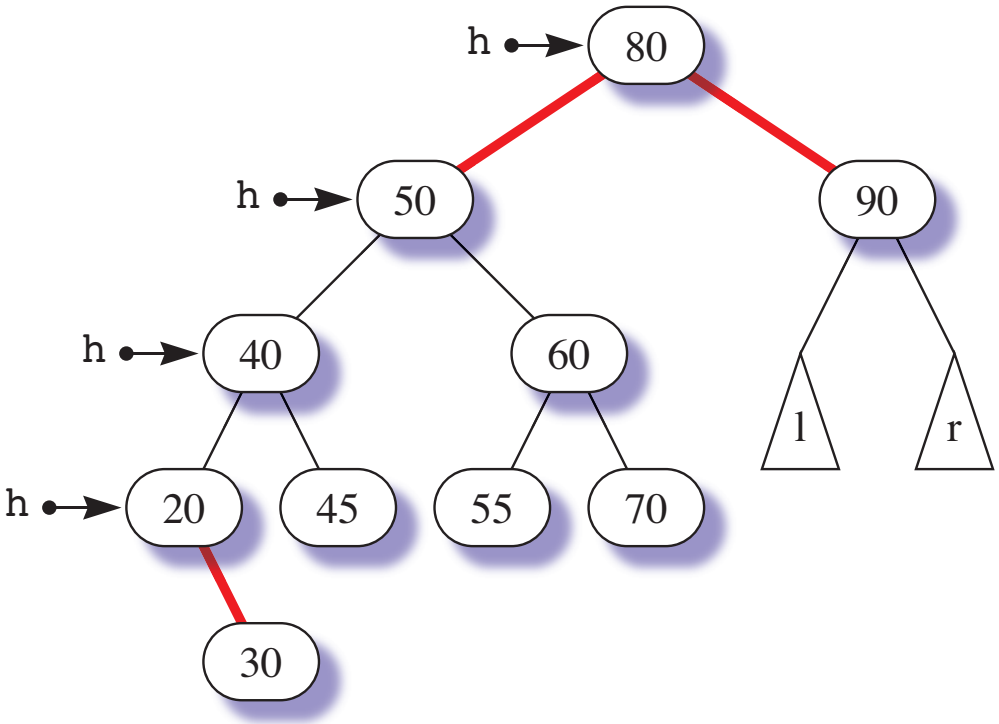
(h) `moveRedLeft()` only flips colors.

Deleting minimum.
Move red left does more than flipping colors.

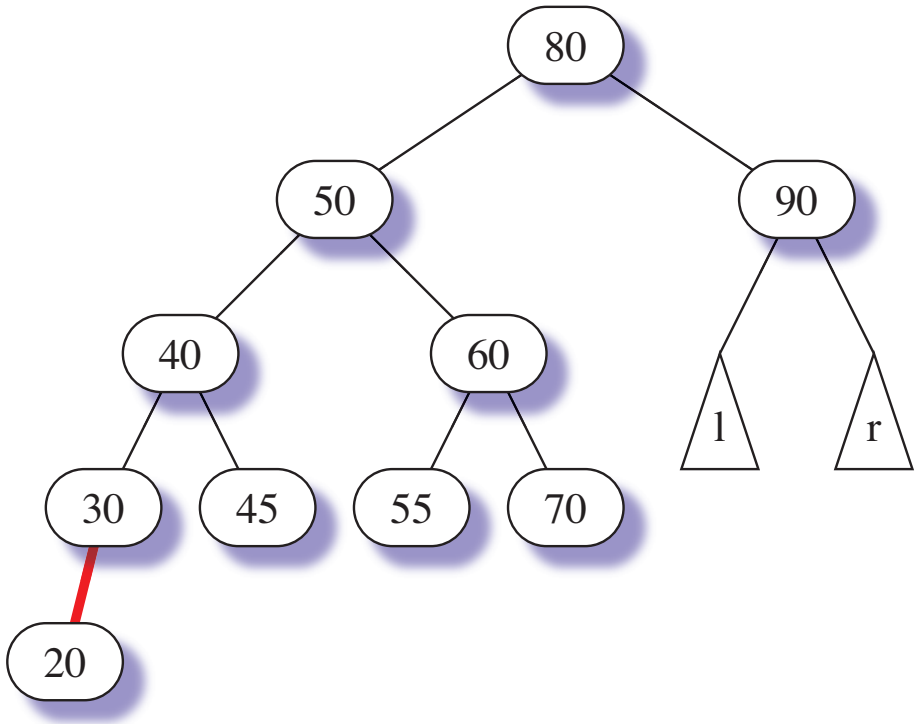


(i) Delete minimum with red parent link.

Deleting minimum.
Move red left does more than flipping colors.



(i) Delete minimum with red parent link.



(j) After four fixups.

The remove operation

- Find the node to remove.
- Find the minimum of the right child of the node to remove, which is the successor node.
- Copy the data from the successor to the node to remove with the `min()` operation.
- Remove the minimum of the right child.

Used to copy the data from the successor node.

```
// ========= min =========
// Pre: h != nullptr.
// Post: The minimum of tree rooted at h is returned.
template<class T>
T Node<T>::min(shared_ptr<Node<T>> h) {
    cerr << "min(): Exercise for the student." << endl;
    throw -1;
}
```

The remove operation

- Find the node to remove.
- Find the minimum of the right child of the node to remove, which is the successor node.
- Copy the data from the successor to the node to remove with the `min()` operation.
- Remove the minimum of the right child.

Find the node to remove

- Same strategy: need to push the red links down.
- Finding the node to remove may require moving to the right as well as to the left.
- If you move to the left on the way to the node to remove, call `moveRedLeft()` as before.
- If you move to the right, call `moveRedRight()`.

```
// ========= moveRedRight =========
// Pre: Both h->_right and h->_right->_left are BLACK.
// Post: Either h->_right or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
<class T>
shared_ptr<Node<T>> Node<T>::moveRedRight(shared_ptr<Node<T>> h) {
    if (isRed(h->_right) || isRed(h->_right->_left)) {
        cerr << "moveRedRight precondition violated: "
             << "h->_right is RED or h->_right->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_left->_left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}
```

```
// ========= moveRedRight =========
// Pre: Both h->_right and h->_right->_left are BLACK.
// Post: Either h->_right or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
<class T>
shared_ptr<Node<T>> Node<T>::moveRedRight(shared_ptr<Node<T>> h) {
    if (isRed(h->_right) || isRed(h->_right->_left)) {
        cerr << "moveRedRight precondition violated: "
             << "h->_right is RED or h->_right->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_left->_left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}
```

Move red right might only flip the colors.

```
// ========= moveRedRight =========
// Pre: Both h->_right and h->_right->_left are BLACK.
// Post: Either h->_right or one of its children is RED.
// Post: A pointer to the root node of the modified tree is returned.
<class T>
shared_ptr<Node<T>> Node<T>::moveRedRight(shared_ptr<Node<T>> h) {
    if (isRed(h->_right) || isRed(h->_right->_left)) {
        cerr << "moveRedRight precondition violated: "
             << "h->_right is RED or h->_right->_left is RED" << endl;
        throw -1;
    }
    flipColors(h);
    if (isRed(h->_left->_left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}
```

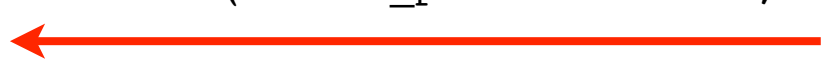
This condition could not be repaired by fixup on the way back up.

The remove operation

```
// ========= remove =========
// Post: If data is found, then it is removed from this tree.
template<class T>
void LLRBTree<T>::remove(T const &data) {
    if (!contains(data)) {
        return;
    }
    _root = _root->remove(_root, data);
    if (_root) {
        _root->_color = BLACK;
    }
}
```

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```



```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Found, or move down to the right.

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Found and can return immediately.

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Possible adjustment with moveRedRight.

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Found the node to remove.

```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Copy data from successor node.

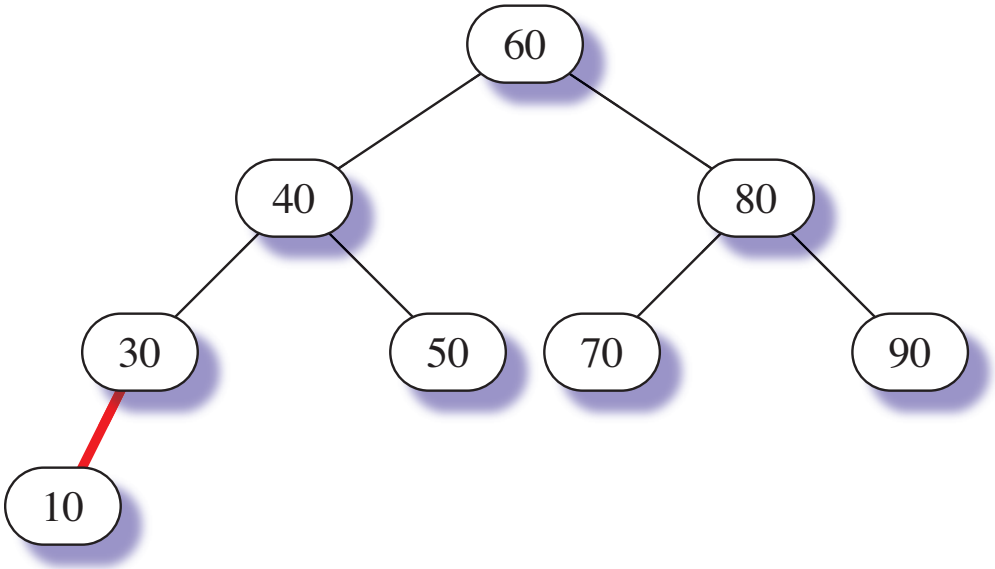
```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Delete successor node.

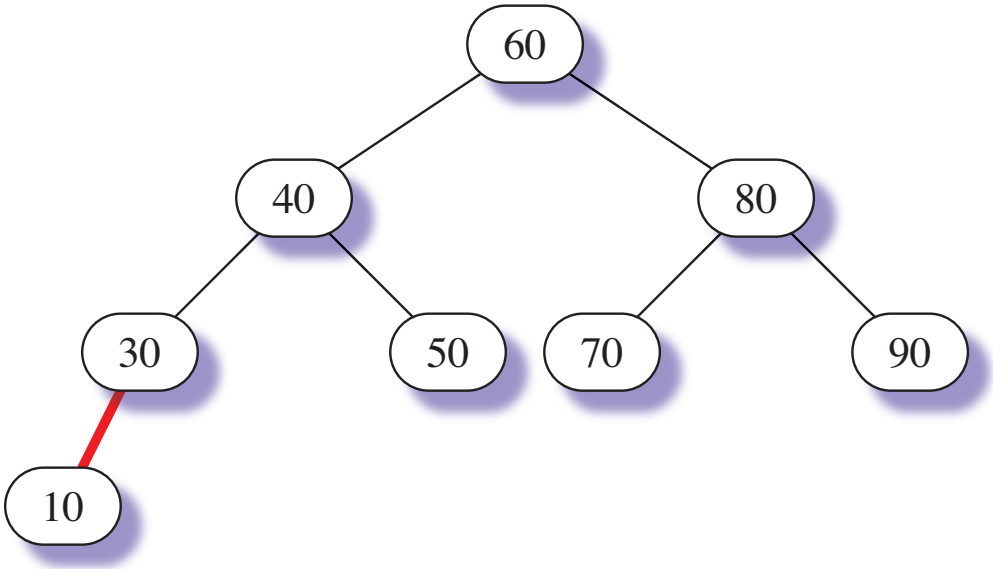
```
// Post: data is removed from root node h.
// Post: A pointer to the root node of the modified tree is returned.
template<class T>
shared_ptr<Node<T>> Node<T>::remove(shared_ptr<Node<T>> h, T const &data) {
    if (data < h->_data) {
        if (!isRed(h->_left) && !isRed(h->_left->_left)) {
            h = moveRedLeft(h) ;
        }
        h->_left = remove(h->_left, data);
    }
    else {
        if (isRed(h->_left)) {
            h = rotateRight(h);
        }
        if ((data == h->_data) && (h->_right == nullptr)) {
            return nullptr;
        }
        if (!isRed(h->_right) && !isRed(h->_right->_left)) {
            h = moveRedRight(h);
        }
        if (data == h->_data) {
            h->_data = min(h->_right);
            h->_right = removeMin(h->_right);
        }
        else {
            h->_right = remove(h->_right, data);
        }
    }
    return fixup(h);
}
```

Move down to the right.

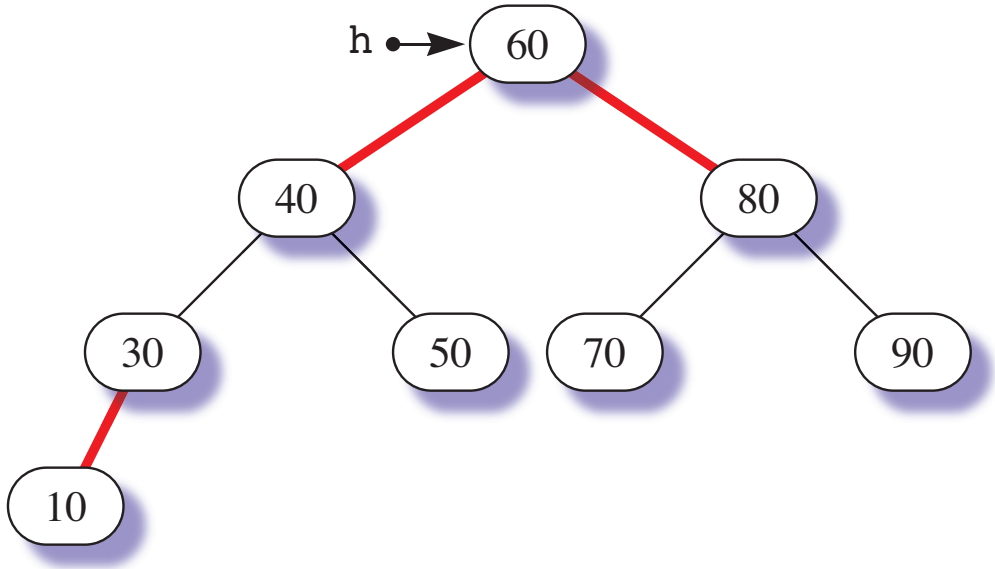
Removing 50 from an LLRBTree



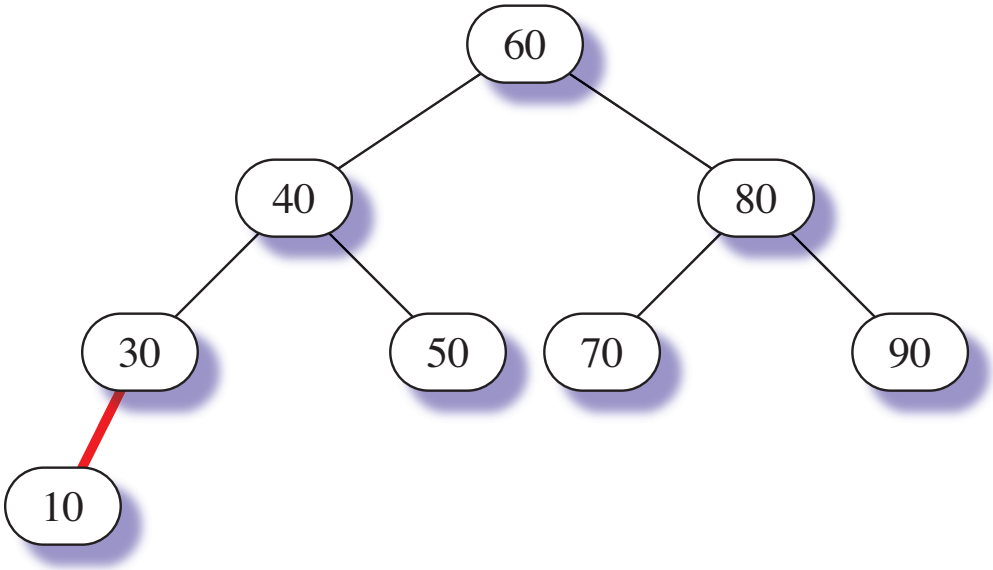
(a) Initial tree.



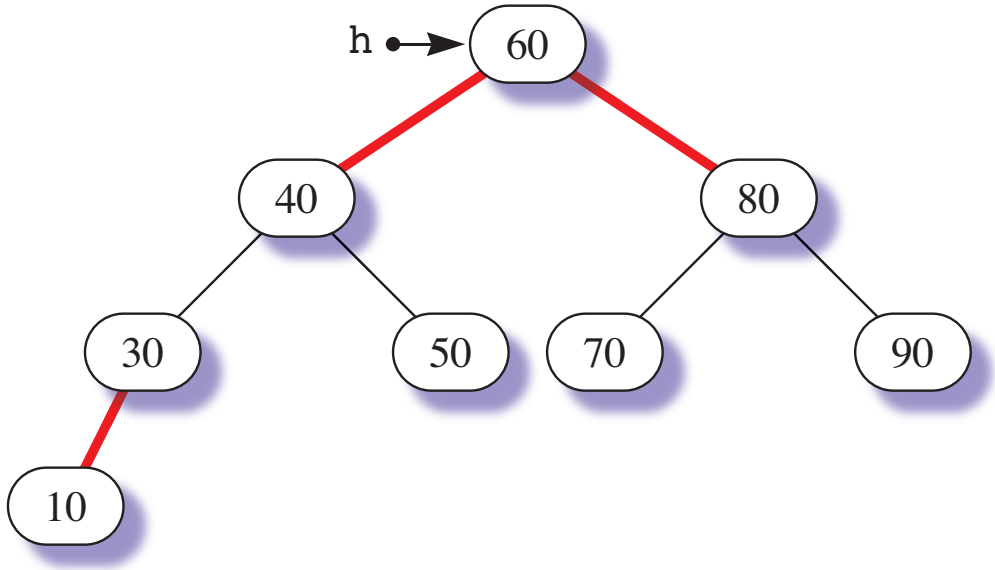
(a) Initial tree.



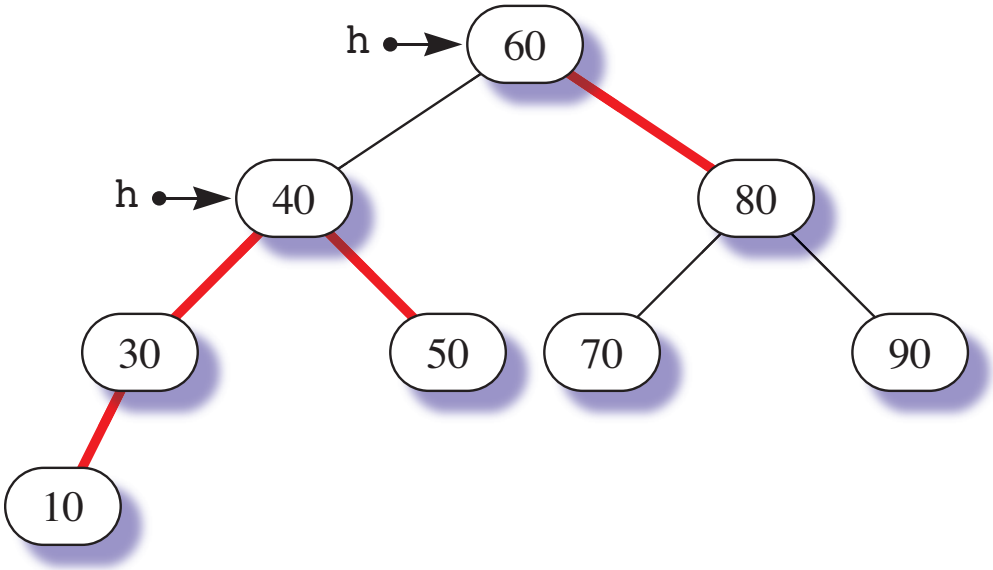
(b) `moveRedLeft()`, only flips colors.



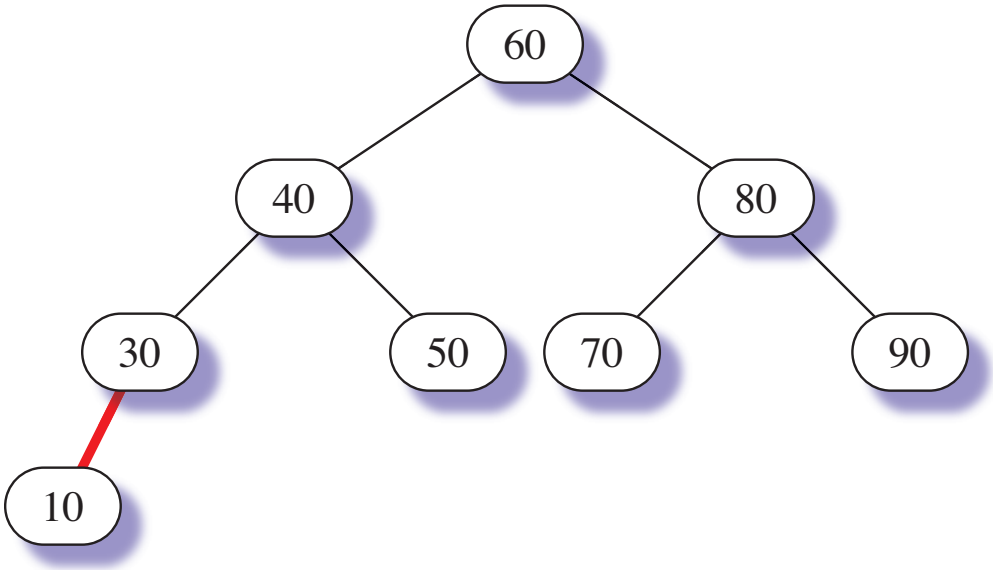
(a) Initial tree.



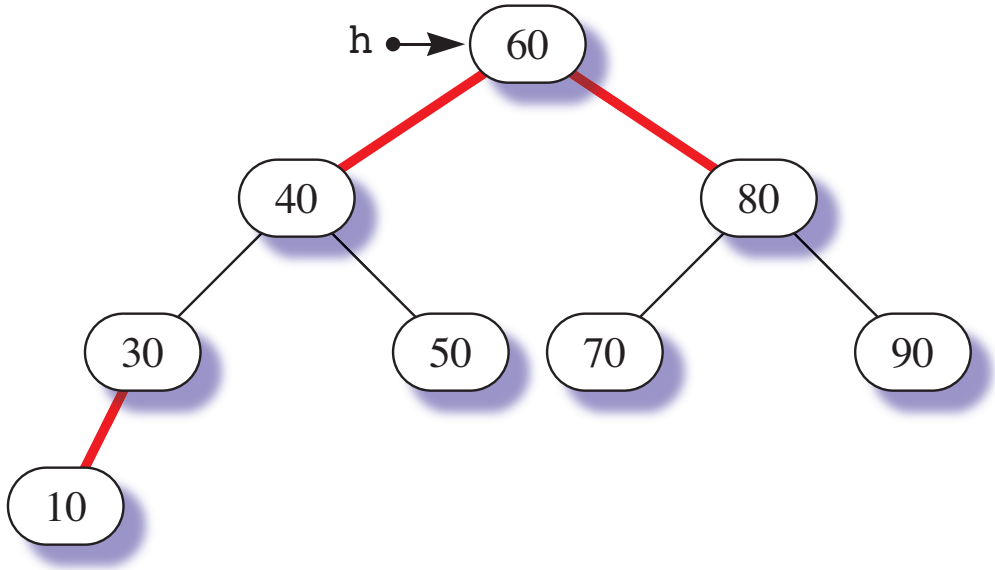
(b) `moveRedLeft()`, only flips colors.



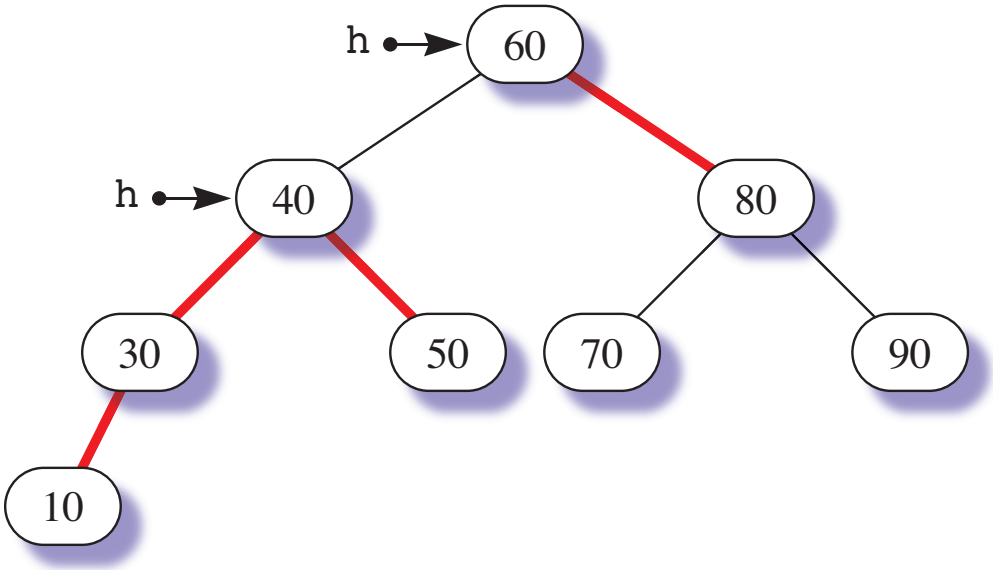
(c) `moveRedRight()`, `h = flipColors(h)`.



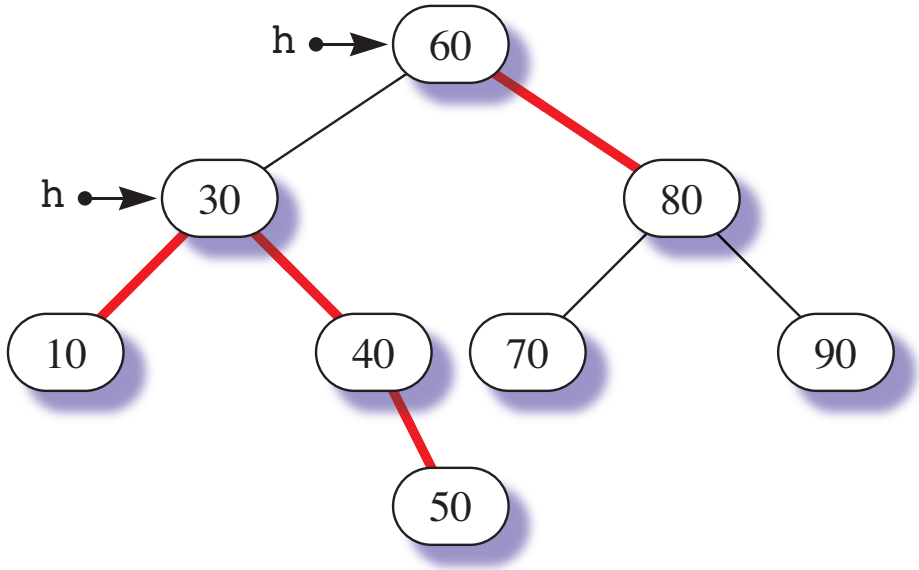
(a) Initial tree.



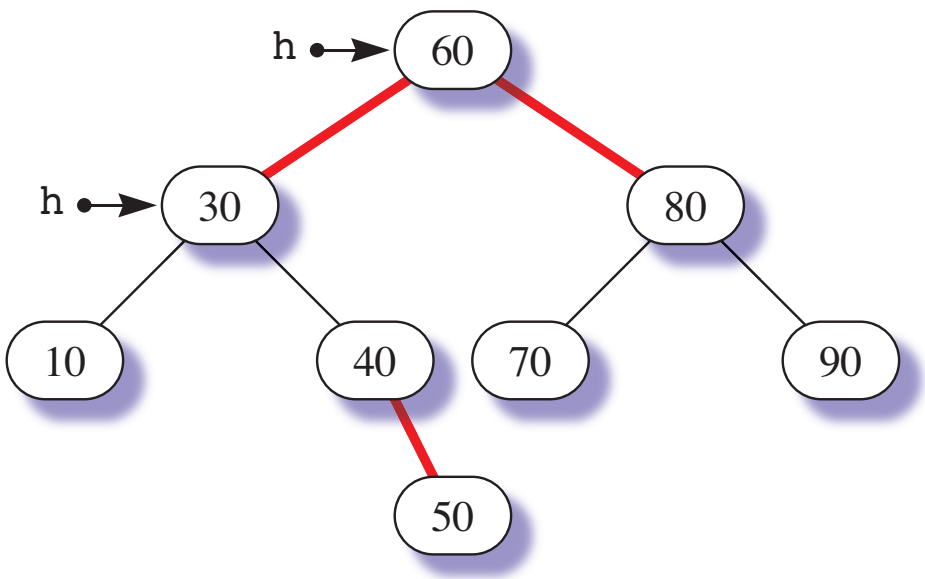
(b) `moveRedLeft()`, only flips colors.



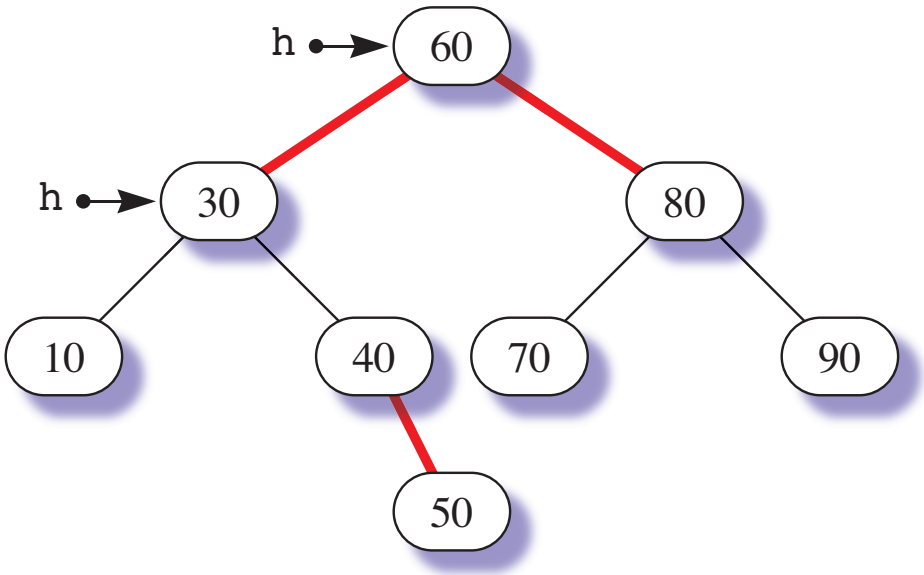
(c) `moveRedRight()`, `h = flipColors(h)`.



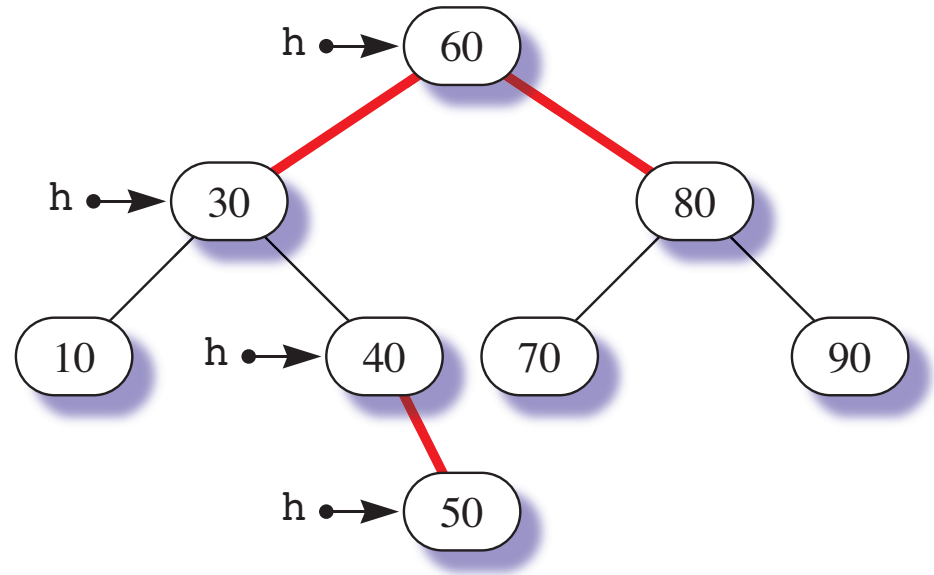
(d) `rotateRight(h)`.



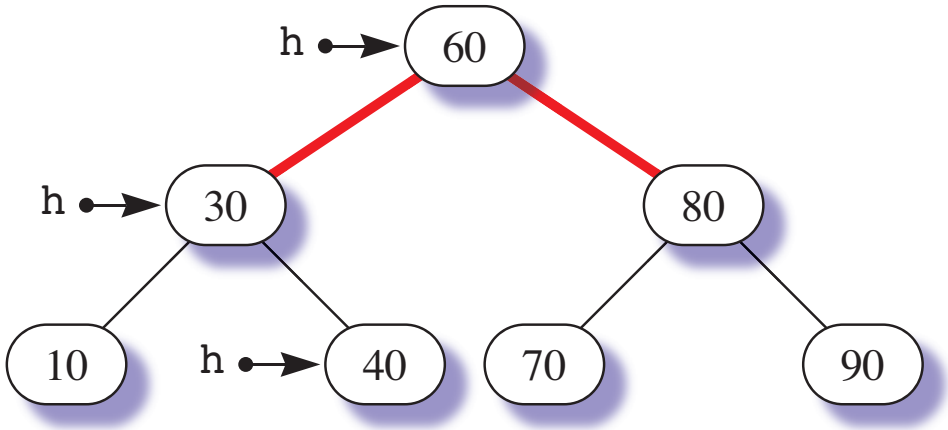
(e) `h = flipColors(h)`.



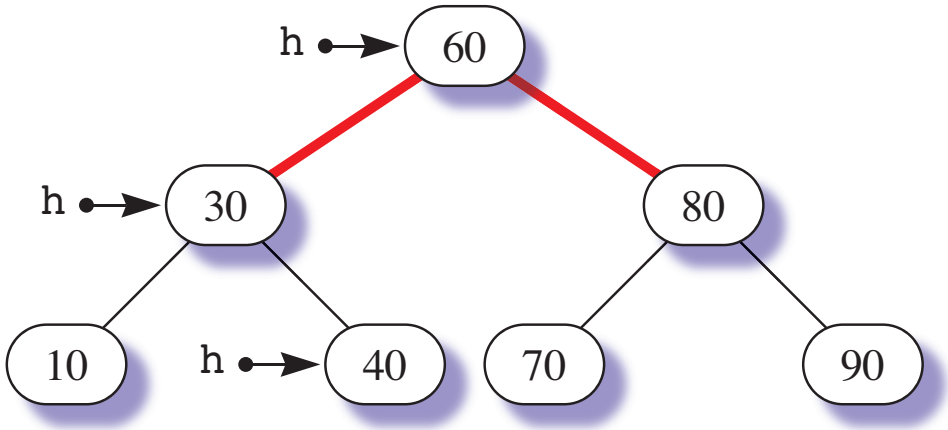
(e) `h = flipColors(h)`.



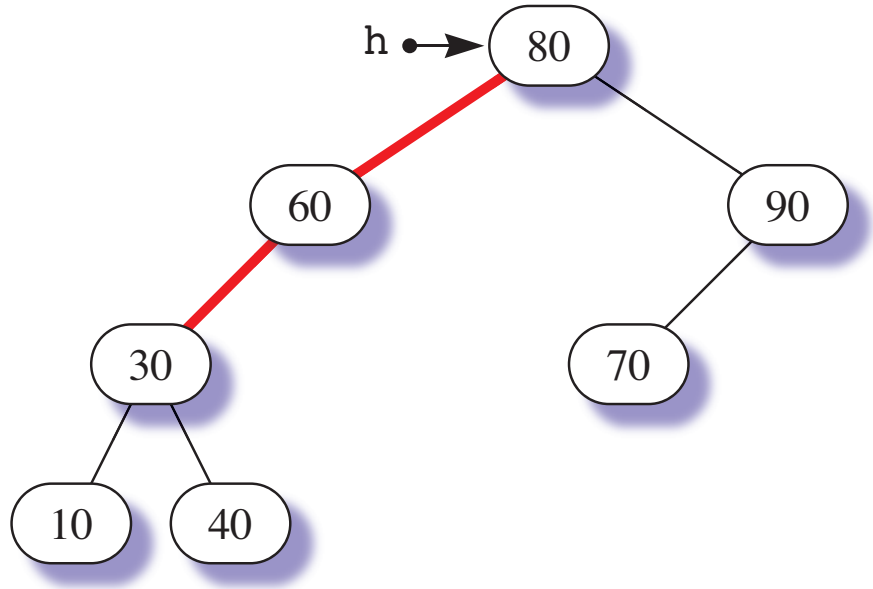
(f) `remove()`, twice.



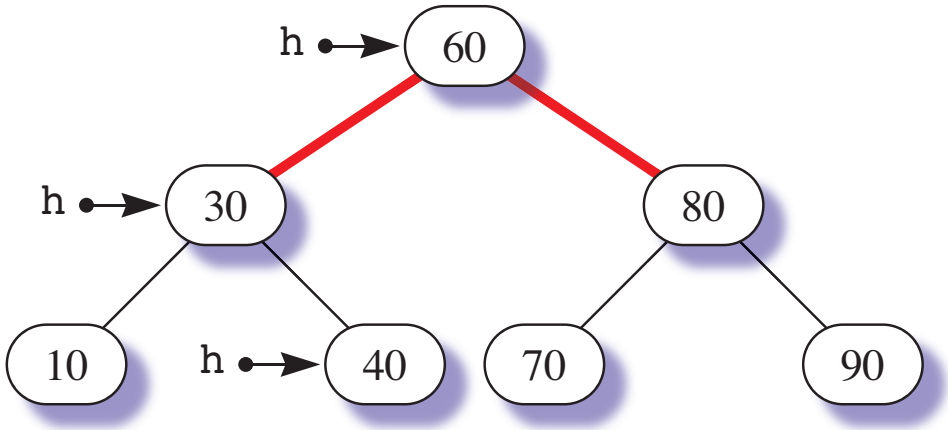
(g) `delete h`.



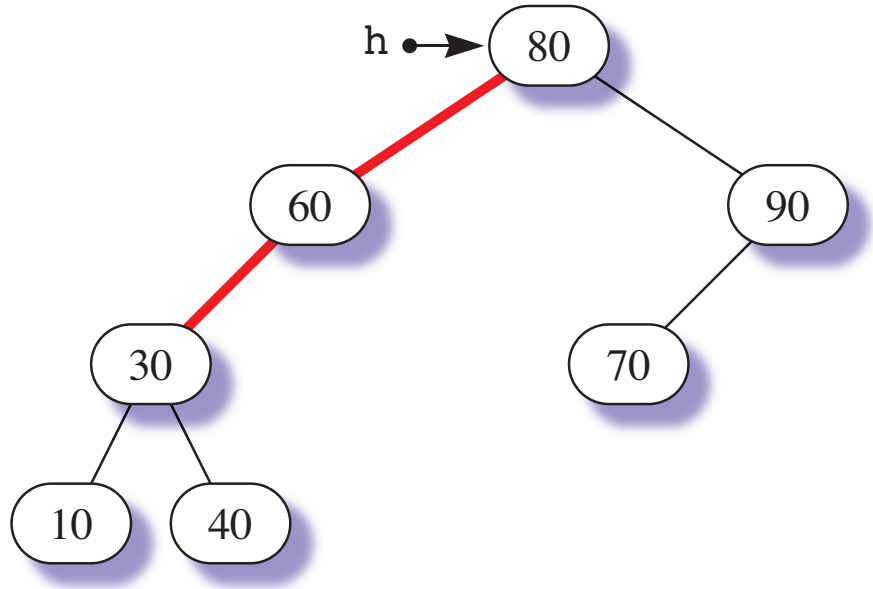
(g) `delete h`.



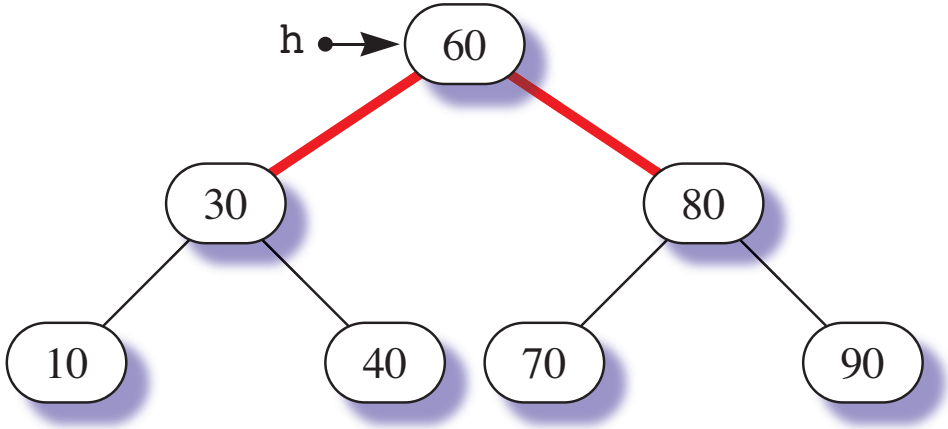
(h) Fixup with `rotateLeft()`.



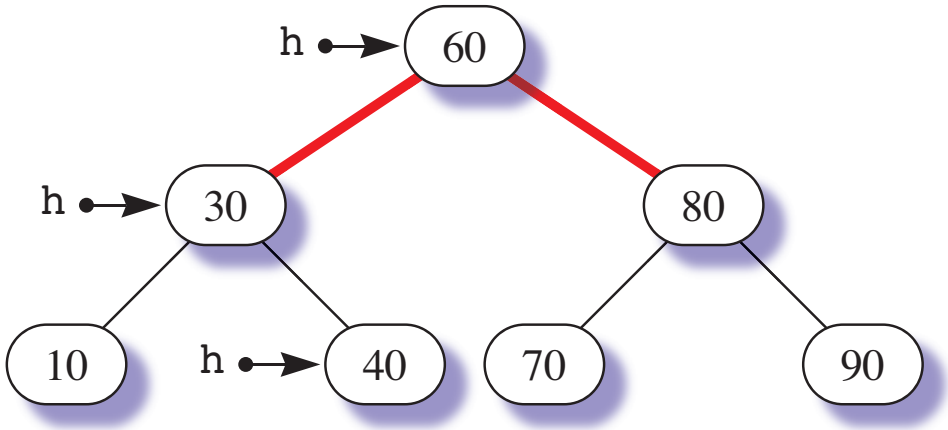
(g) `delete h`.



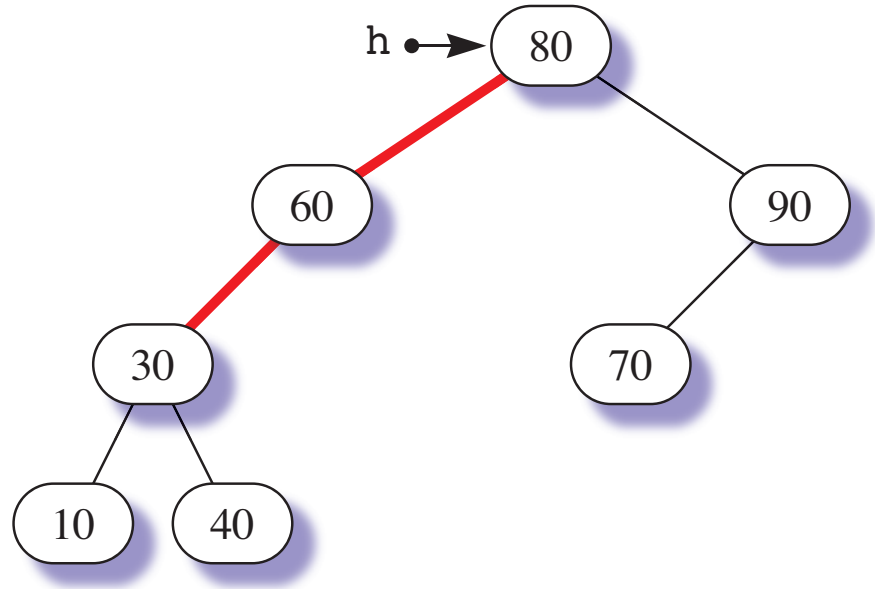
(h) Fixup with `rotateLeft()`.



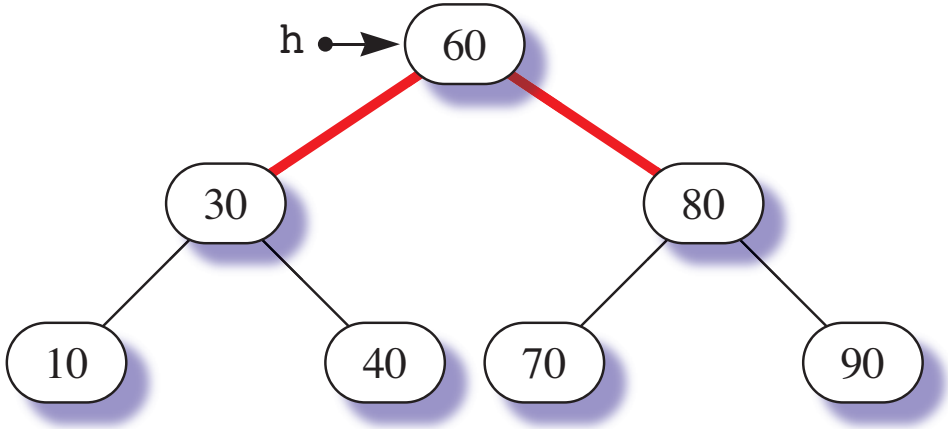
(i) Fixup with `rotateRight()`.



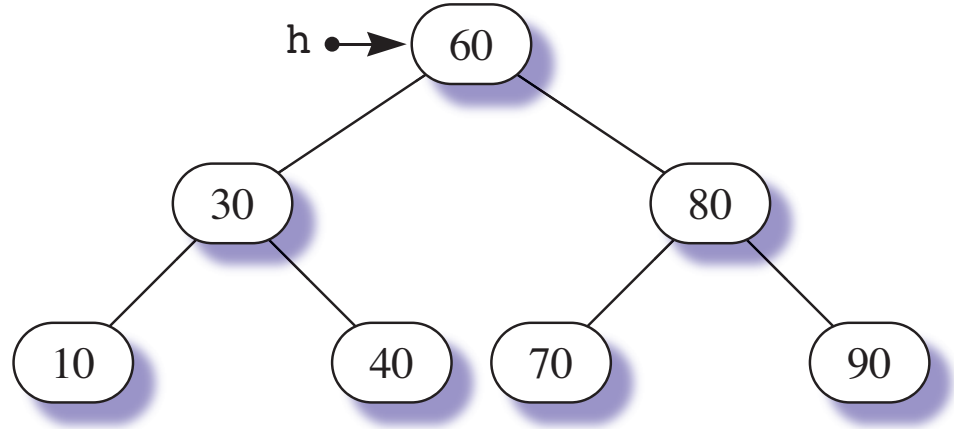
(g) `delete h`.



(h) Fixup with `rotateLeft()`.



(i) Fixup with `rotateRight()`.



(j) Fixup with `flipColors()`.

Insert fixup, general red-black tree

RB-INSERT-FIXUP(T, z)

while $z.p.color$ == RED
  if $z.p$ == $z.p.p.left$
    $y = z.p.p.right$
    if $y.color$ == RED
      $z.p.color = \text{BLACK}$
      $y.color = \text{BLACK}$
      $z.p.p.color = \text{RED}$
      $z = z.p.p$
    else if z == $z.p.right$
      $z = z.p$
      LEFT-ROTATE(T, z)
      $z.p.color = \text{BLACK}$
      $z.p.p.color = \text{RED}$
      RIGHT-ROTATE$(T, z.p.p)$
  else (same as **then** clause with "right" and "left" exchanged)
$T.root.color = \text{BLACK}$

Case 1

Case 2

Case 3

Remove fixup, general red-black tree

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK
            x.p.color = RED
            LEFT-ROTATE(T, x.p)
            w = x.p.right
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED
            x = x.p
        else if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                RIGHT-ROTATE(T, w)
                w = x.p.right
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            LEFT-ROTATE(T, x.p)
            x = T.root
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1

Case 2

Case 3

Case 4