Left-Leaning
Red-Black Trees

Robert Sedgewick
Princeton University
Introduction

2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees
Deletion
Red-black trees

are now found throughout our computational infrastructure

Textbooks on algorithms

Library search function in many programming environments

Popular culture (stay tuned)

Worth revisiting?
Red-black trees are now found throughout our computational infrastructure.

Typical:

> ya thanks,
> i got the idea
> but is there some other place on the web where only the algorithms
> used by STL is
> explained. (that is the underlying data structures etc. ) without
> explicit reference to the code (as it is pretty confusing) if I try to
> read through).
>
> thanks[/color]

The standard does not specify which algorithms the STL must use. Implementers are free to choose which ever algorithm or data structure that fulfills the functional and efficiency requirements of the standard.

There are some common choices however. For instance every implementation of map, multimap, set and multiset that I have ever seen uses a structure called a red black tree. Typing 'red black tree algorithm' in google produces a number of likely looking links.

john
Digression:

Red-black trees are found in popular culture??

![Missing Season 2 DVD Cover](image.png)
Mystery: black door?
Mystery: red door?
An explanation?
Primary goals

Red-black trees (Guibas-Sedgewick, 1978)

- reduce code complexity
- minimize or eliminate space overhead
- unify balanced tree algorithms
- single top-down pass (for concurrent algorithms)
- find version amenable to average-case analysis

Current implementations

- maintenance
- migration
- space not so important (?)
- guaranteed performance
- support full suite of operations

Worth revisiting?
Primary goals

Red-black trees (Guibas-Sedgewick, 1978)

- reduce code complexity
- minimize or eliminate space overhead
- unify balanced tree algorithms
- single top-down pass (for concurrent algorithms)
- find version amenable to average-case analysis

Current implementations

- maintenance
- migration
- space not so important (??)
- guaranteed performance
- support full suite of operations

Worth revisiting?  YES. Code complexity is out of hand.
Introduction

2-3-4 Trees

Red-Black Trees

Left-Leaning RB Trees

Deletion
2-3-4 Tree

Generalize BST node to allow multiple keys. Keep tree in perfect balance.

**Perfect balance.** Every path from root to leaf has same length.

Allow 1, 2, or 3 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.
Search in a 2-3-4 Tree

Compare node keys against search key to guide search.

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

Ex: Search for L
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

**Insert.**

- Search to bottom for key.

Ex: Insert B

```
A  D  F  G  J  L  N  Q  S  V  Y  Z
```

- B not found
- C not found
- K not found

```
K  R
/  \\
C  E
   /  \\
  A  D
    /  \\
   F  G  J
```

- smaller than K
- smaller than C
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to a 3-node.

Ex: Insert B

- B fits here.
- Smaller than C.
- Smaller than K.
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

• Search to bottom for key.
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

**Insert.**

- Search to bottom for key.
- 3-node at bottom: convert to a 4-node.

---

**Ex: Insert X**

- Larger than R
- Larger than W
- X fits here
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

**Insert.**

- Search to bottom for key.

Ex: Insert H
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

- Search to bottom for key.
  - 2-node at bottom: convert to a 3-node.
  - 3-node at bottom: convert to a 4-node.
  - 4-node at bottom: no room for new key.

Ex: Insert H

```
A      D      F      G      J
C      E

K      R
M      O
W
L      N      Q
S      V      Y      Z

smaller than K
larger than E
no room for H
```
Splitting 4-nodes in a 2-3-4 tree

is an effective way to make room for insertions

**Problem:** Doesn’t work if parent is a 4-node

**Bottom-up solution (Bayer, 1972)**
- Use same method to split parent
- Continue up the tree while necessary

**Top-down solution (Guibas-Sedgewick, 1978)**
- Split 4-nodes on the way down
- Insert at bottom
Splitting 4-nodes on the way down ensures that the “current” node is not a 4-node

Transformations to split 4-nodes:

Invariant: “Current” node is not a 4-node

Consequences:

• 4-node below a 4-node case never happens
• Bottom node reached is always a 2-node or a 3-node
Splitting a 4-node below a 2-node is a local transformation that works anywhere in the tree.
Splitting a 4-node below a 3-node is a local transformation that works anywhere in the tree.
Growth of a 2-3-4 tree

happens *upwards* from the bottom

- **insert A**
  
  \[ \text{A} \]

- **insert S**
  
  \[ \text{A S} \]

- **insert E**
  
  \[ \text{A E S} \]

- **insert R**
  
  \[ \begin{array}{c}
  \text{E} \\
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{R}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array} \quad \text{split 4-node to}
  \begin{array}{c}
  \text{E}
  \end{array} \\
  \begin{array}{c}
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array}
  \text{and then insert}
  \begin{array}{c}
  \text{E}
  \end{array} \quad \begin{array}{c}
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{R}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array} \quad \text{tree grows up one level} \]

- **insert C**
  
  \[ \begin{array}{c}
  \text{E}
  \end{array} \quad \begin{array}{c}
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{C}
  \end{array} \quad \begin{array}{c}
  \text{R}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array} \]

- **insert D**
  
  \[ \begin{array}{c}
  \text{E}
  \end{array} \quad \begin{array}{c}
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{C}
  \end{array} \quad \begin{array}{c}
  \text{D}
  \end{array} \quad \begin{array}{c}
  \text{R}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array} \]

- **insert I**
  
  \[ \begin{array}{c}
  \text{E}
  \end{array} \quad \begin{array}{c}
  \text{A}
  \end{array} \quad \begin{array}{c}
  \text{C}
  \end{array} \quad \begin{array}{c}
  \text{D}
  \end{array} \quad \begin{array}{c}
  \text{I}
  \end{array} \quad \begin{array}{c}
  \text{R}
  \end{array} \quad \begin{array}{c}
  \text{S}
  \end{array} \]
Growth of a 2-3-4 tree (continued)

happens **upwards** from the bottom
Balance in 2-3-4 trees

Key property: All paths from root to leaf are the same length

Tree height.
- Worst case: \( \lg N \) [all 2-nodes]
- Best case: \( \log_4 N = \frac{1}{2} \lg N \) [all 4-nodes]
- Between 10 and 20 for 1 million nodes.
- Between 15 and 30 for 1 billion nodes.

Guaranteed logarithmic performance for both search and insert.
Direct implementation of 2-3-4 trees is complicated because of code complexity.

Maintaining multiple node types is cumbersome.

- Representation?
- Need multiple compares to move down in tree.
- Large number of cases for splitting.
- Need to convert 2-node to 3-node and 3-node to 4-node.

```java
private void insert(Key key, Val val) {
    Node x = root;
    while (x.getChild(key) != null) {
        x = x.getChild(key);
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
    return x;
}
```

**Bottom line:** Could do it, but stay tuned for an easier way.
Introduction

2-3-4 Trees

Red-Black Trees

Left-Leaning RB Trees

Deletion
Red-black trees (Guibas-Sedgewick, 1978)

1. Represent 2-3-4 tree as a BST.
2. Use "internal" edges for 3- and 4- nodes.

Key Properties

- elementary BST search works
- easy to maintain a correspondence with 2-3-4 trees (and several other types of balanced trees)

Note: correspondence is not 1-1. (3-nodes can lean either way)

Many variants studied (details omitted.)

NEW VARIANT (this talk): Left-leaning red-black trees
Introduction
2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees
Deletion
Left-leaning red-black trees

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning edges for 3- and 4- nodes.

Key Properties

- elementary BST search works
- easy-to-maintain correspondence with 2-3-4 trees
Left-leaning red-black trees

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning edges for 3- and 4- nodes.

Disallowed

- right-leaning edges

- three reds in a row
Java data structure for red-black trees

adds **one bit for color** to elementary BST data structure

```java
public class BST<Key extends Comparable<Key>, Value> {
    private static final boolean RED = true;
    private static final boolean BLACK = false;
    private Node root;

    private class Node {
        Key key;
        Value val;
        Node left, right;
        boolean color;
        Node(Key key, Value val, boolean color) {
            this.key = key;
            this.val = val;
            this.color = color;
        }
    }

    public Value get(Key key) {
        // Search method.
        return null;
    }

    public void put(Key key, Value val) {
        // Insert method.
        return null;
    }
}
```

*constants*

*color of incoming link*

*helper method to test node color*

private boolean isRed(Node x) {
    if (x == null) return false;
    return (x.color == RED);
}
Search implementation for red-black trees

is the same as for elementary BSTs

( but typically runs faster because of better balance in the tree).

**BST (and LLRB tree) search implementation**

```java
public Value get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp == 0) return x.val;
        else if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
    }
    return null;
}
```

**Note:** Other BST methods also work

- order statistics
- iteration

**Ex: Find the minimum key**

```java
public Key min() {
    Node x = root;
    while (x != null) x = x.left;
    if (x == null) return null;
    else return x.key;
}
```
Insert implementation for LLRB trees

is best expressed in a **recursive** implementation

---

**Recursive insert() implementation for elementary BSTs**

```java
private Node insert(Node h, Key key, Value val) {
    if (h == null) {
        return new Node(key, val);
    }

    int cmp = key.compareTo(h.key);
    if (cmp == 0) h.val = val;
    else if (cmp < 0) {
        h.left = insert(h.left, key, val);
    } else {
        h.right = insert(h.right, key, val);
    }

    return h;
}
```

---

**Note:** effectively travels down the tree and then up the tree.

- simplifies correctness proof
- simplifies code for balanced BST implementations
- could remove recursion to get single-pass algorithm
Insert implementation for LLRB trees

follows directly from 1-1 correspondence with 2-3-4 trees

1. If key found on recursive search, reset value, as usual.
2. If key not found, insert at the bottom.

3. Split 4-nodes on the way down
Balanced tree code

is based on local transformations known as rotations

```java
private Node rotL(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    return x;
}
```

```java
private Node rotR(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    return x;
}
```
Insert a new node at the bottom in a LLRB tree follows directly from 1-1 correspondence with 2-3-4 trees

1. Add new node as usual, with red link to glue it to node above
2. **Rotate left if necessary** to make link lean left
Splitting a 4-node in a LLRB tree

follows directly from 1-1 correspondence with 2-3-4 trees

1. Rotate **right** to balance the 4-node
2. Flip colors to pass **red** link up one level
3. **Rotate left if necessary** to make link lean left

*Parent is a 2-node: two cases*
Splitting a 4-node in a LLRB tree

follows directly from 1-1 correspondence with 2-3-4 trees

1. Rotate **right** to balance the 4-node
2. Flip colors to pass **red** link up one level
3. **Rotate left if necessary** to make link lean left

*Parent is a 3-node: three cases*
Splitting a 4-node in a LLRB tree

follows directly from 1-1 correspondence with 2-3-4 trees

1. Rotate **right** to balance the 4-node
2. Flip colors to pass **red** link up one level
3. **Rotate left if necessary** to make link lean left

**Key point:** The transformations are all the same.
Inserting and splitting nodes in LLRB trees

are easier when left rotates are done on the way up the tree.

Search as usual

• if key found reset value, as usual
• if key not found insert a new red node at the bottom [might be right-leaning red link]

Split 4-nodes on the way down the tree.

• right-rotate and flip color
• might leave right-leaning link higher up in the tree

NEW TRICK: enforce left-leaning condition on the way up the tree.

• left-rotate any right-leaning link on search path
• trivial with recursion (do it after recursive calls)
• no other right-leaning links elsewhere
Insert code for LLRB trees

is based on three simple operations.

1. Insert a new node at the bottom.

   ```java
   if (h == null)
     return new Node(key, value, RED);
   ```

2. Split a 4-node.

   ```java
   private Node splitFourNode(Node h) {
     x = rotR(h);
     x.left.color = BLACK;
     return x;
   }
   ```

3. Enforce left-leaning condition.

   ```java
   private Node leanLeft(Node h) {
     x = rotL(h);
     x.color = x.left.color;
     x.left.color = RED;
     return x;
   }
   ```
Insert implementation for LLRB trees

is a few lines of code added to elementary BST insert

private Node insert(Node h, Key key, Value val)
{
    if (h == null)
        return new Node(key, val, RED);

    if (isRed(h.left))
        if (isRed(h.left.left))
            h = splitFourNode(h);

    int cmp = key.compareTo(h.key);
    if (cmp == 0) h.val = val;
    else if (cmp < 0)
        h.left = insert(h.left, key, val);
    else
        h.right = insert(h.right, key, val);

    if (isRed(h.right))
        h = leanLeft(h);

    return h;
}
LLRB insert movie
Why revisit red-black trees?

Take your pick:

```java
private Node insert(Node x, Key key, Value val, boolean sw) {
    if (x == null)
        return new Node(key, val, RED);
    int cmp = key.compareTo(x.key);
    if (isRed(x.left) && isRed(x.right))
        {  
            x.color = RED;
            x.left.color = BLACK;
            x.right.color = BLACK;
        }
    if (cmp == 0) x.val = val;
    else if (cmp < 0) {
        x.left = insert(x.left, key, val, false);
        if (isRed(x) && isRed(x.left) && sw)
            x = rotR(x);
        if (isRed(x.left) && isRed(x.left.left))
            {  
                x = rotR(x);
                x.color = BLACK; x.right.color = RED;
            }
    } else // if (cmp > 0)
    {  
        x.right = insert(x.right, key, val, true);
        if (isRed(h) && isRed(x.right) && !sw)
            x = rotL(x);
        if (isRed(h.right) && isRed(h.right.right))
            {  
                x = rotL(x);
                x.color = BLACK; x.left.color = RED;
            }
    }
    return x;
}
```

```java
private Node insert(Node h, Key key, Value val) {
    int cmp = key.compareTo(h.key);
    if (h == null)
        return new Node(key, val, RED);
    if (isRed(h.left))
        if (isRed(h.left.left))
            {  
                h = rotR(h);
                h.left.color = BLACK;
            }
    if (cmp == 0) h.val = val;
    else if (cmp < 0)
        h.left = insert(h.left, key, val);
    else
        h.right = insert(h.right, key, val);
    if (isRed(h.right))
        {  
            h = rotL(h);
            h.color = h.left.color;  
            h.left.color = RED;
        }
    return h;
}
```

**Very tricky**

**Straightforward**

---

*Introduction*

2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees
Deletion
Why revisit red-black trees?

Take your pick:

TreeMap.java

Adapted from CLR by experienced professional programmers (2004)

Introduction
2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees
Deletion

Why left-leaning trees?

Take your pick:

private Node insert(Node x, Key key, Value val, boolean red) {
    if (x == null) {
        return new Node(key, val, RED);
    }
    int cmp = key.compareTo(x.key);
    if (cmp < 0) {
        x.left = insert(x.left, key, val, false);
        if (isRed(x.left) && !isRed(x.right)) {
            x.color = RED;
            x.left.color = BLACK;
            x.right.color = BLACK;
        }
        if (cmp == 0) x.val = val;
    } else if (cmp > 0) {
        x.right = insert(x.right, key, val, false);
        if (isRed(x.right) && !isRed(x.left)) {
            x.color = BLACK;
            x.right.color = RED;
        }
        x = put(x);
        x.color = BLACK;
        if (isRed(x.left)) {
            if (isRed(x.right)) {
                x.left = put(x.left);
                x.color = RED;
                x.left.color = BLACK;
                x = x.left;
            }
            if (isRed(x.right)) {
                x.right = put(x.right);
                x.color = RED;
                x.right.color = BLACK;
                x = x.right;
            }
        } else if (isRed(x.left) && isRed(x.left.left)) {
            x.left = rotateLeft(x); // x.left 
        } else if (isRed(x.right) && isRed(x.right.right)) {
            x.right = rotateRight(x); // x.right 
        } else {
            x = put(x);
        }
    }
    return x;
}

Wrong scale!
Why revisit red-black trees?

LLRB implementation is far simpler than previous attempts.

- left-leaning restriction reduces number of cases
- recursion gives two (easy) chances to fix each node
- short inner loop more than compensates for slight increase in height

Improves widely used algorithms

- AVL, 2-3, and 2-3-4 trees
- red-black trees

Same ideas simplify implementation of other operations

- delete min, max
- arbitrary delete
Introduction
2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees

Deletion
Warmup 1: delete the minimum

1. Search down the left spine of the tree.
2. If search ends in a 3-node or 4-node: just remove it.

3. Removing a 2-node would destroy balance
   - transform tree on the way down the search path
   - Invariant: current node is not a 2-node

Note: LLRB representation reduces number of cases (as for insert)
Warmup 1: delete the minimum

Carry a red link **down** the left spine of the tree.

Invariant: either $h$ or $h.\text{left}$ is **RED**

Implication: deletion easy at bottom

Need to adjust tree only when $h.\text{left}$ and $h.\text{left}.\text{left}$ are both **BLACK**

Two cases, depending on color of $h.\text{right}.\text{left}$

```java
private Node moveRedLeft(Node h) {
    h.color = BLACK;
    h.left.color = RED;
    if (isRed(h.right.left)) {
        h.right = rotR(h.right);
        h = rotL(h);
    } else h.right.color = RED;
    return h;
}
```

**Easy case:** $h.\text{right}.\text{left}$ is **BLACK**

**Color flip**

**Harder case:** $h.\text{right}.\text{left}$ is **RED**

**Color flip and rotate right**

**Rotate left**

$h.\text{left}.\text{left}$ turns **RED**
Leaving right red links on the search path

simplifies the code, complicates the proof.

1. Does each transformation preserve balance?

2. Does each transformation preserve correspondence with 2-3-4 trees?
deleteMin() implementation for LLRB trees

is otherwise a few lines of code

```java
public void deleteMin()
{
    root = deleteMin(root);
    root.color = BLACK;
}

private Node deleteMin(Node h)
{
    if (h.left == null)
        return null;

    if (!isRed(h.left) && !isRed(h.left.left))
        h = moveRedLeft(h);

    h.left = deleteMin(h.left);

    if (isRed(h.right))
        h = leanLeft(h);

    return h;
}
```

- remove node on bottom level (h must be RED by invariant)
- push red link down if necessary
- move down one level
- fix right-leaning red links on the way up the tree
deleteMin() example

1. \[ \text{push reds down} \]
   \[
   \begin{array}{c}
   1 \quad D \quad H \\
   A \quad B \quad C \quad E \quad F \quad G \quad I \quad J \quad K \quad L \quad M \quad N \quad O
   \end{array}
   \]

2. \[ \text{fix right-leaning reds on the way up} \]
   \[
   \begin{array}{c}
   2 \quad D \quad H \\
   A \quad B \quad C \quad E \quad F \quad G \quad I \quad J \quad K \quad L \quad M \quad N \quad O
   \end{array}
   \]

3. \[ \text{remove minimum} \]
   \[
   \begin{array}{c}
   3 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad L \quad M \quad N \quad O
   \end{array}
   \]

4. \[ \]
   \[
   \begin{array}{c}
   4 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad L \quad M \quad N \quad O
   \end{array}
   \]

5. \[ \]
   \[
   \begin{array}{c}
   5 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad L \quad M \quad N \quad O
   \end{array}
   \]

6. \[ \]
   \[
   \begin{array}{c}
   6 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad M \quad N \quad O
   \end{array}
   \]

7. \[ \]
   \[
   \begin{array}{c}
   7 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad M \quad N \quad O
   \end{array}
   \]

8. \[ \]
   \[
   \begin{array}{c}
   8 \quad D \quad H \\
   A \quad B \quad C \quad E \quad G \quad I \quad J \quad K \quad M \quad N \quad O
   \end{array}
   \]

Introduction
2-3-4 Trees
Red-Black Trees
Left-Leaning RB Trees
Deletion
LLRB deleteMin() movie
Warmup 2: delete the maximum

is similar, but slightly different (since trees lean left).

```java
private Node deleteMax(Node h) {
    if (h.right == null) {
        if (h.left != null)
            h.left.color = BLACK;
        return h.left;
    }
    if (isRed(h.left))
        h = leanRight(h);
    if (!isRed(h.right) && !isRed(h.right.left))
        h = moveRedRight(h);
    h.right = deleteMax(h.right);
    if (isRed(h.right))
        h = leanLeft(h);
    return h;
}
```

```java
private Node moveRedRight(Node h) {
    h.color = BLACK;
    h.right.color = RED;
    if (isRed(h.left.left))
        h = rotR(h);
    h.color = RED;
    h.left.color = BLACK;
    else h.left.color = RED;
    return h;
}
```
**deleteMax() example**

**push reds down**

1. D  
   - A  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N  
   - O

2. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

**fix right-leaning reds on the way up**

3. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

4. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

**remove maximum**

5. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

6. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

7. D  
   - H  
   - L  
   - B  
   - C  
   - E  
   - G  
   - I  
   - J  
   - K  
   - M  
   - N

*nothing to fix!*
LLRB deleteMax() movie
Deleting an arbitrary node

involves the same general strategy.

1. Search down the left spine of the tree.
2. If search ends in a 3-node or 4-node: just remove it.

3. Removing a 2-node would destroy balance
   • transform tree on the way down the search path
   • Invariant: current node is not a 2-node

Difficulty:
   • Far too many cases!
   • LLRB representation dramatically reduces the number of cases.

Q: How many possible search paths in two levels?
A: \[ 9 \times 6 + 27 \times 9 + 81 \times 12 = 1269 \] (!!)
Deleting an arbitrary node

reduces to `deleteMin()`

A standard trick:

```
h.key = min(h.right);
h.value = get(h.right, h.key);
h.right = deleteMin(h.right);
```

![Diagram of deletion process](image)

1. **to delete D**
   - `deleteMin(right child of D)`
   - `flip colors, delete node`
   - `fix right-leaning red link`

2. **replace its key, value with those of its successor**

3. **then delete the successor**
Deleting an arbitrary node at the bottom can be implemented with the same helper methods used for \texttt{deleteMin()} and \texttt{deleteMax()}. 

\textbf{Invariant:} h or one of its children is \textbf{RED}

- search path goes left: use \texttt{moveRedLeft()}. 
- search path goes right: use \texttt{moveRedRight()}. 
- delete node at bottom 
- fix right-leaning reds on the way up 

A few loose ends remain . . . et voilà! (see next page)
delete() implementation for LLRB trees

private Node delete(Node h, Key key) {
    int cmp = key.compareTo(h.key);
    if (cmp < 0) {
        if (!isRed(h.left) && !isRed(h.left.left))
            h = moveRedLeft(h);
        h.left = delete(h.left, key);
    } else {
        if (isRed(h.left)) h = leanRight(h);
        if (cmp == 0 && (h.right == null))
            return null;
        if (!isRed(h.right) && !isRed(h.right.left))
            h = moveRedRight(h);
        if (cmp == 0) {
            h.key = min(h.right);
            h.value = get(h.right, h.key);
            h.right = deleteMin(h.right);
        } else h.right = delete(h.right, key);
    } if (isRed(h.right)) h = leanLeft(h);
    return h;
}
LLRB delete() movie
Alternatives

Red-black-tree implementations in widespread use:
- are based on pseudocode with “case bloat”
- use parent pointers (!)
- 400+ lines of code for core algorithms

Left-leaning red-black trees
- you just saw all the code
- single pass (remove recursion if concurrency matters)
- <80 lines of code for core algorithms
- less code implies faster insert, delete
- less code implies easier maintenance and migration

accomplishes the same result with less than 1/4 the code