Left-Leaning
Red-Black Trees

Robert Sedgewick
Princeton University

Original version: Data structures seminar at Dagstuhl (Feb 2008)
  • red-black trees made simpler (!)
  • full delete() implementation
This version: Analysis of Algorithms meeting at Maresias (Apr 2008)
  • back to balanced 4-nodes
  • back to 2-3 trees (!)
  • scientific analysis
Addendum: observations developed after talk at Maresias

Java code at www.cs.princeton.edu/~rs/talks/LLRB/Java
Movies at www.cs.princeton.edu/~rs/talks/LLRB/movies
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:*

```plaintext
> 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 fulfils 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 poster](image-url)
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

LLRB Trees

Deletion

Analysis
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**

- Smaller than K
- Smaller than C
- B not found
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**

```
Key          | Children
-------------|---------
K            | R
C            | E
A            | B
D
F            | G
J
M            | O
L
N            | Q
S            | V
Y            | Z
W
```
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

*Insert.*

- Search to bottom for key.

---

**Ex: Insert X**

- Insertion process:
  - Search to the bottom for the key X.
  - Key X not found.
  - Insertion continues as needed.
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

- smaller than K
- larger than E
- H not found
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
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.

Could be huge

Unchanged
Splitting a 4-node below a 3-node is a local transformation that works anywhere in the tree.

- Introduction
- 2-3-4 Trees
- LLRB Trees
- Deletion
- Analysis

Diagram:

- Original tree before splitting
- Split result with local transformation

Notes:
- "could be huge" for the original tree
- "unchanged" for the split result
Growth of a 2-3-4 tree

happens **upwards** from the bottom

- **insert A**
  - A

- **insert S**
  - A S

- **insert E**
  - A E S

- **insert R**
  - A R S

- **insert C**
  - E
  - A C
  - R S

- **insert D**
  - E
  - A C D
  - R S

- **insert I**
  - E
  - A C D
  - I R S

Tree grows up one level after splitting the 4-node and inserting the new element.
Growth of a 2-3-4 tree (continued)

happens upwards from the bottom

- insert N
- insert B
- split 4-node to C E R, and then insert
- split 4-node to E R, and then insert
- insert X
- tree grows up one level
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.getTheCorrectChild(key) != null) {
        x = x.getTheCorrectChild(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

LLRB Trees

Deletion

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

1. Represent 2-3-4 tree as a BST.
2. Use "internal" red 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
**Left-leaning red-black trees**

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

---

**Key Properties**

- elementary BST search works
- easy-to-maintain correspondence with 2-3-4 trees
- trees therefore have perfect black-link balance
Left-leaning red-black trees

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

Disallowed

• right-leaning 3-node representation

• two 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.
    }

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

**Important 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;  // associative model (no duplicate keys)
  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 stack-based single-pass algorithm
Balanced tree code

is based on local transformations known as rotations

In red-black trees, we only rotate red links (to maintain perfect black-link balance)

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

```java
private Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = x.right.color;
    x.right.color = RED;
    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 if necessary** to get correct 3-node or 4-node representation
Splitting a 4-node is accomplished with a color flip

Flip the colors of the three nodes

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

Key points:
- preserves perfect black-link balance
- passes a RED link up the tree
- reduces problem to inserting (that link) into parent
Splitting a 4-node in a LLRB tree

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

1. Flip colors, which passes red link up one level
2. Rotate if necessary to get correct representation in parent
   (using precisely the same transformations as for insert at bottom)

**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. Flip colors, which passes red link up one level
2. Rotate if necessary to get correct representation in parent
   (using precisely the same transformations as for insert at bottom)

Parent is a 3-node: three cases
Inserting and splitting nodes in LLRB trees

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

Search as usual
- if key found reset value, as usual
- if key not found  insert new red node at the bottom
- might leave right-leaning red or two reds in a row higher up in the tree

Split 4-nodes on the way down the tree.
- flip color
- might leave right-leaning red or two reds in a row higher up in the tree

**NEW TRICK:** Do rotates on the way **UP** the tree.
- left-rotate any right-leaning link on search path
- right-rotate top link if two reds in a row found
- trivial with recursion (do it after recursive calls)
- no corrections needed elsewhere
Insert code for LLRB trees

is based on four 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
   if (isRed(h.left) && isRed(h.right))
       colorFlip(h);
   ```

3. Enforce left-leaning condition.
   
   ```java
   if (isRed(h.right))
       h = rotateLeft(h);
   ```

4. Balance a 4-node.
   
   ```java
   if (isRed(h.left) && isRed(h.left.left))
       h = rotateRight(h);
   ```
Insert implementation for LLRB trees

is a few lines of code added to elementary BST insert

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

    if (isRed(h.left) && isRed(h.right))
        colorFlip(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 = rotateLeft(h);

    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);

    return h;

}  
```

- **insert at the bottom**
- **split 4-nodes on the way down**
- **standard BST insert code**
- **fix right-leaning reds on the way up**
- **fix two reds in a row on the way up**
LLRB (top-down 2-3-4) insert movie
Q. What happens if we move color flip to the end?

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

    if (isRed(h.left) && isRed(h.right))
        colorFlip(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 = rotateLeft(h);

    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);

    return h;
}
```
Introduction

2-3-4 Trees
LLRB Trees
Deletion
Analysis

A surprise

Q. What happens if we move color flip to the end?

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

    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 = rotateLeft(h);

    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);

    if (isRed(h.left) && isRed(h.right))
        colorFlip(h);

    return h;
}
```
Q. What happens if we move color flip to the end?

A. It becomes an implementation of 2-3 trees (!)

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

    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 = rotateLeft(h);

    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);

    if (isRed(h.left) && isRed(h.right))
        colorFlip(h);

    return h;
}
```

*Insert in 2-3 tree:*
- attach new node with red link
- 2-node → 3-node
- 3-node → 4-node
- split 4-node
- pass red link up to parent and repeat
- no 4-nodes left!
Insert implementation for 2-3 trees (!)

is a few lines of code added to elementary BST insert

```java
private Node insert(Node h, Key key, Value val) {
    if (h == null)
        return new Node(key, val, RED);
    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 = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))
        colorFlip(h);
    return h;
}
```

- **insert at the bottom**
- **standard BST insert code**
- **fix right-leaning reds on the way up**
- **fix two reds in a row on the way up**
- **split 4-nodes on the way up**
LLRB (bottom-up 2-3) insert movie
Why revisit red-black trees?

Which do you prefer?

private Node insert(Node x, Key key, Value val, boolean sw) {
    if (x == null)
        return new Node(key, value, 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;
}
Why revisit red-black trees?

Take your pick:

TreeMap.java

Adapted from CLR by experienced professional programmers (2004)

Wrong scale!

Why revisit red-black trees?

Which do you prefer?

```java
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, RED);
    else if (cmp > 0)
        x.right = insert(x.right, key, val, RED);
    else
    if (isRed(x.left) && isRed(x.left.left))
        x = rotateRight(x);
    if (isRed(x.left) && isRed(x.left.left))
        x.left = rotateRight(x.left);
    x.color = BLACK; x.left.color = RED;
    return x;
}
```

Very tricky

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

Very straightforward

150

46

33

Lines of code for insert (lower is better!)
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
- take your pick: top-down 2-3-4 or bottom-up 2-3

![Diagram of tree operations]

Implements widely used implementations

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

**Same ideas simplify implementation of other operations**

- delete min, max
- arbitrary delete
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
- take your pick: top-down 2-3-4 or bottom-up 2-3

Improves widely used implementations

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

Same ideas simplify implementation of other operations

- delete min, max
- arbitrary delete
Lessons learned from insert() implementation also simplify delete() implementations

1. Color flips and rotations preserve perfect black-link balance.
2. Fix right-leaning reds and eliminate 4-nodes on the way up.

```
private Node fixUp(Node 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))
        colorFlip(h);
    return h;
}
```

Delete strategy (works for 2-3 and 2-3-4 trees)

- invariant: current node is not a 2-node
- introduce 4-nodes if necessary
- remove key from bottom
- eliminate 4-nodes on the way up
Warmup 1: delete the maximum

1. Search down the right spine of the tree.
2. If search ends in a 3-node or 4-node: just remove it.
   
   ![Diagram showing deletion process](image)

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

   ![Diagram showing balance transformation](image)

**Note:** LLRB representation reduces number of cases (as for insert)
Warmup 1: delete the maximum
by carrying a red link down the right spine of the tree.

Invariant: either $h$ or $h$.right is RED
Implication: deletion easy at bottom

1. Rotate red links to the right
2. Borrow from sibling if necessary
   - when $h$.right and $h$.right.left are both BLACK
   - Two cases, depending on color of $h$.left.left

```java
private Node moveRedRight(Node h) {
    colorFlip(h);
    if (isRed(h.left.left)) {
        h = rotateRight(h);
        colorFlip(h);
    }
    return h;
}
```
deleteMax() implementation for LLRB trees

is otherwise a few lines of code

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

private Node deleteMax(Node h)
{
    if (isRed(h.left))
        h = rotateRight(h);

    if (h.right == null)
        return null;

    if (!isRed(h.right) && !isRed(h.right.left))
        h = moveRedRight(h);

    h.left = deleteMax(h.left);
    return fixUp(h);
}
```

lean 3-nodes to the right
remove node on bottom level
(h must be RED by invariant)
borrow from sibling if necessary
move down one level
fix right-leaning red links
and eliminate 4-nodes on the way up
deleteMax() example 1

push reds down

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

2. D
   H
   B
   A
   C
   E
   G
   I
   J
   K
   M
   N
   O

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

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

fix right-leaning reds on the way up

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

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

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

(remove maximum)

8. D
   H
   L
   N
   B
   A
   C
   E
   G
   J
   K
   M
   N

(not nothing to fix!)
deleteMax() example 2

push reds down

1

D
B
A
C
E
F
G
I
J
K
L
N

remove maximum

4

D
B
A
C
F
E
G
I
J
K
L
M

fix right-leaning reds on the way up

2

D
B
A
C
H
F
E
G
J
K
L
N

3

D
B
A
C
H
F
E
G
J
I
K
L
N

4

D
B
A
C
H
F
E
G
J
K
L
M

5

D
B
A
C
F
E
G
J
I
K
L
M

6

D
B
A
C
F
E
G
J
K
L
M
Warmup 2: delete the minimum

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

Invariant: either h or h.left is RED
Implication: deletion easy at bottom

Borrow from sibling
• if h.left and h.left.left are both BLACK
• two cases, depending on color of h.right.left

```java
private Node moveRedLeft(Node h) {
    colorFlip(h);  
    if (isRed(h.right.left)) {
        h.right = rotateRight(h.right);  
        h = rotateLeft(h);  
        colorFlip(h);
    }
    return h;
}
```
deleteMin() implementation for LLRB trees

is 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);

    return fixUp(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
  and eliminate 4-nodes
  on the way up
**deleteMin() example**

- **push reds down**

  1. Original tree
  2. Push reds down
  3. Fix right-leaning reds on the way up
  4. Push reds down
  5. Fix right-leaning reds on the way up
  6. Push reds down
  7. Fix right-leaning reds on the way up
  8. Push reds down

- **remove minimum**

  5. Remove minimum
LLRB deleteMin() 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:

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

`deleteMin(right child of D)`

`flip colors, delete node`

`fix right-leaning red link`

`then delete the successor`

replace its key, value with those of its successor
Deleting an arbitrary node at the bottom can be implemented with the same helper methods used for deleteMin() and deleteMax().

Invariant: \( h \) or one of its children is RED

- search path goes left: use moveRedLeft().
- search path goes right: use moveRedRight().
- delete node at bottom
- fix right-leaning reds on the way up
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);
    }
    return fixUp(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 1/5 the code
Introduction

2-3-4 Trees

LLRB Trees

Deletion

Analysis
Worst-case analysis

follows immediately from 2-3-4 tree correspondence

1. All trees have perfect black balance.
2. No two red links in a row on any path.

Shortest path: \( \lg N \) (all black)
Longest path: \( 2 \lg N \) (alternating red-black)

**Theorem:** *With red-black BSTs as the underlying data structure, we can implement an ordered symbol-table API that supports insert, delete, delete the minimum, delete the maximum, find the minimum, find the maximum, rank, select the kth largest, and range count in guaranteed logarithmic time.*

Red-black trees are the method of choice for many applications.
One remaining question

that is of interest in typical applications

The number of *searches* far exceeds the number of inserts.

Q. What is the cost of a typical search?

A. If each tree node is equally likely to be sought, compute the internal path length of the tree and divide by N.

```
N: 8

total internal path length: 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 = 13

average search cost: 13/8 = 1.625
```

Q. What is the expected internal path length of a tree built with randomly ordered keys (average cost of a search)?
Average-case analysis of balanced trees

deserves another look!

Main questions:

Is average path length in tree built from random keys \( \sim c \lg N \) ?
If so, is \( c = 1 \) ?
Average-case analysis of balanced trees
deserves another look!

Main questions:

Is average path length in tree built from random keys $\sim c \lg N$?
If so, is $c = 1$?

Experimental evidence

Ex: Tufte plot of average path length in 2-3 trees

- $N = 100, 200, \ldots, 50,000$
- 100 trees each size
Average-case analysis of balanced trees

deserves another look!

Main questions:

Is average path length in tree built from random keys \( \sim c \lg N \)?
If so, is \( c = 1 \)?

Experimental evidence strongly suggests YES!

Ex: Tufte plot of average path length in 2-3 trees

- \( N = 100, 200, \ldots, 50,000 \)
- 100 trees each size
Experimental evidence can suggest and confirm hypotheses.

**Average path length in (top-down) 2-3-4 tree built from random keys**

**Average path length in 2-3 tree built from random keys**
Average-case analysis of balanced trees

deserves another look!

Main questions:

Is average path length in tree built from random keys $\sim c \lg N$?
If so, is $c = 1$?

Some known facts:

- worst case gives easy $2 \lg N$ upper bound
- fringe analysis of gives upper bound of $c_k \lg N$ with $c_k > 1$
- analytic combinatorics gives path length in random trees

Are simpler implementations simpler to analyze?

Is the better experimental evidence that is now available helpful?

A starting point: study balance at the root (left subtree size)
Left subtree size in left-leaning 2-3 trees

Exact distributions

4

5

6

7
Left subtree size in left-leaning 2-3 trees

Limiting distribution?

smoothed version (32-64)
Left subtree size in left--leaning 2-3 trees

Tufte plot
view of highway for bus driver who has had one Caipirinha too many?
Left subtree size in left-leaning 2-3 trees

Limiting distribution?

10,000 trees for each size
smooth factor 10
An exercise in the analysis of algorithms

Find a proof!

Average path length in 2-3 tree built from random keys

$\lg N - 1.5$
Addendum: Observations
Observation 1

The percentage of red nodes in a 2-3 tree is between 25 and 25.5%
Observation 2

The height of a 2-3 tree is $\sim 2 \ln N$ (!!!)

Very surprising because the average path length in an elementary BST is also $\sim 2 \ln N \approx 1.386 \ln N$
Observation 3

The percentage of red nodes on each path in a 2-3 tree rises to about 25%, then drops by 2 when the root splits.
Observation 4

In aggregate, the observed number of red links per path log-alt ernates between periods of steady growth and not-so-steady decrease (because root-split times vary widely)