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

Next 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

This version: Combinatorics and Probability seminar at University of Pennsylvania (Oct 2008)

- added focus on analytic combinatorics

Java code at www.cs.princeton.edu/~rs/talks/LLRB/Java

Movies at www.cs.princeton.edu/~rs/talks/LLRB/movies

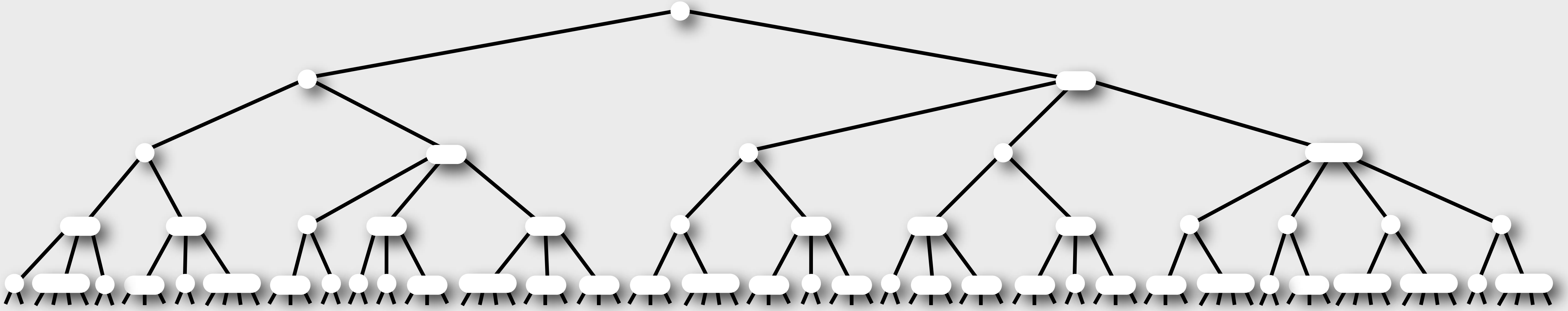
Introduction

2-3-4 Trees

Red-Black Trees

Left-Leaning RB Trees

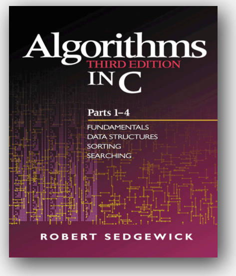
Deletion

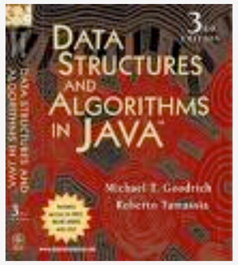
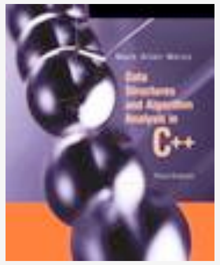


Red-black trees

are now found throughout our computational infrastructure

Textbooks on algorithms



. . .

Library search function in many programming environments

. . .

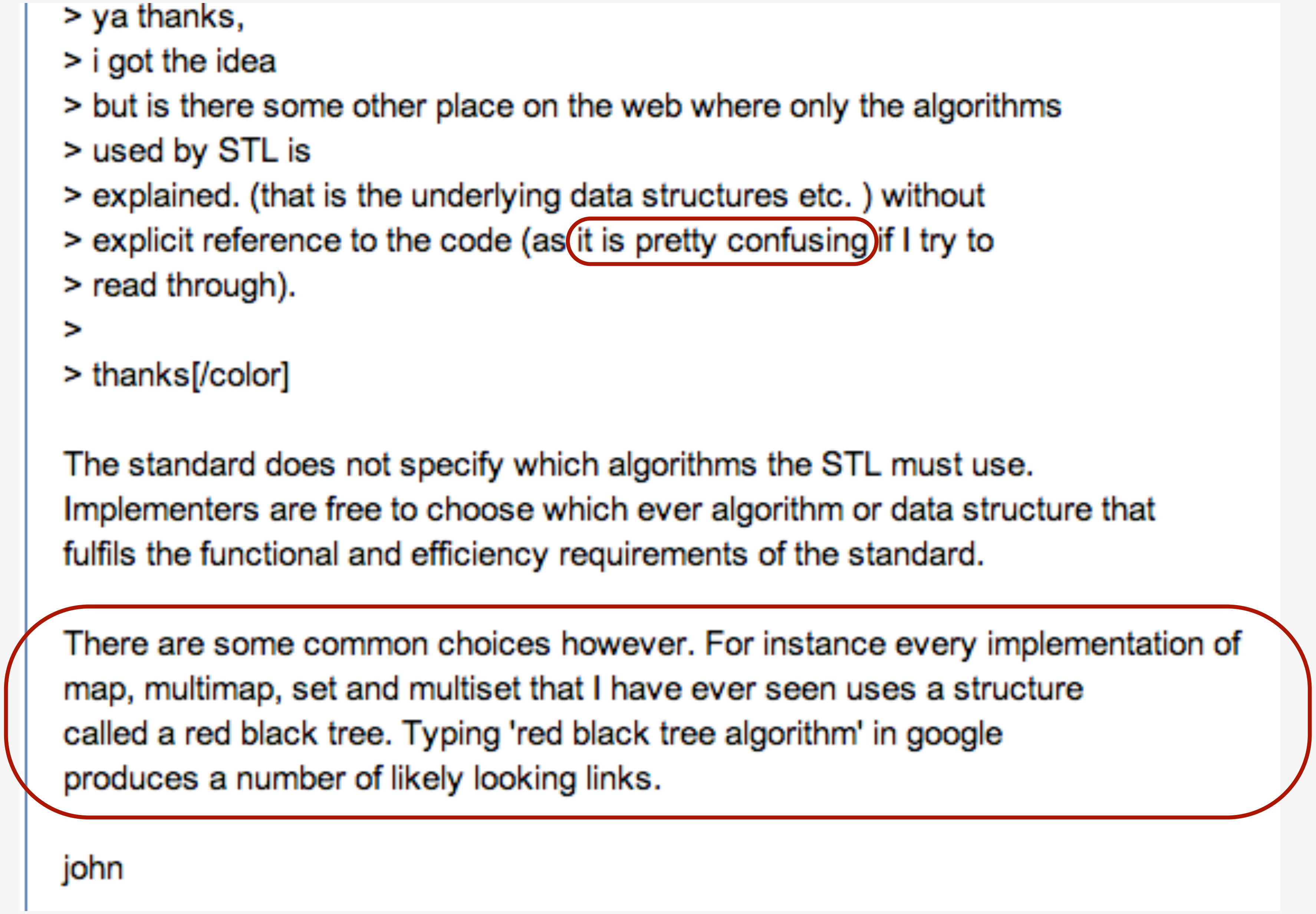
Popular culture (stay tuned)

Worth revisiting?

Red-black trees

are now found throughout our computational infrastructure

Typical:



> 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.

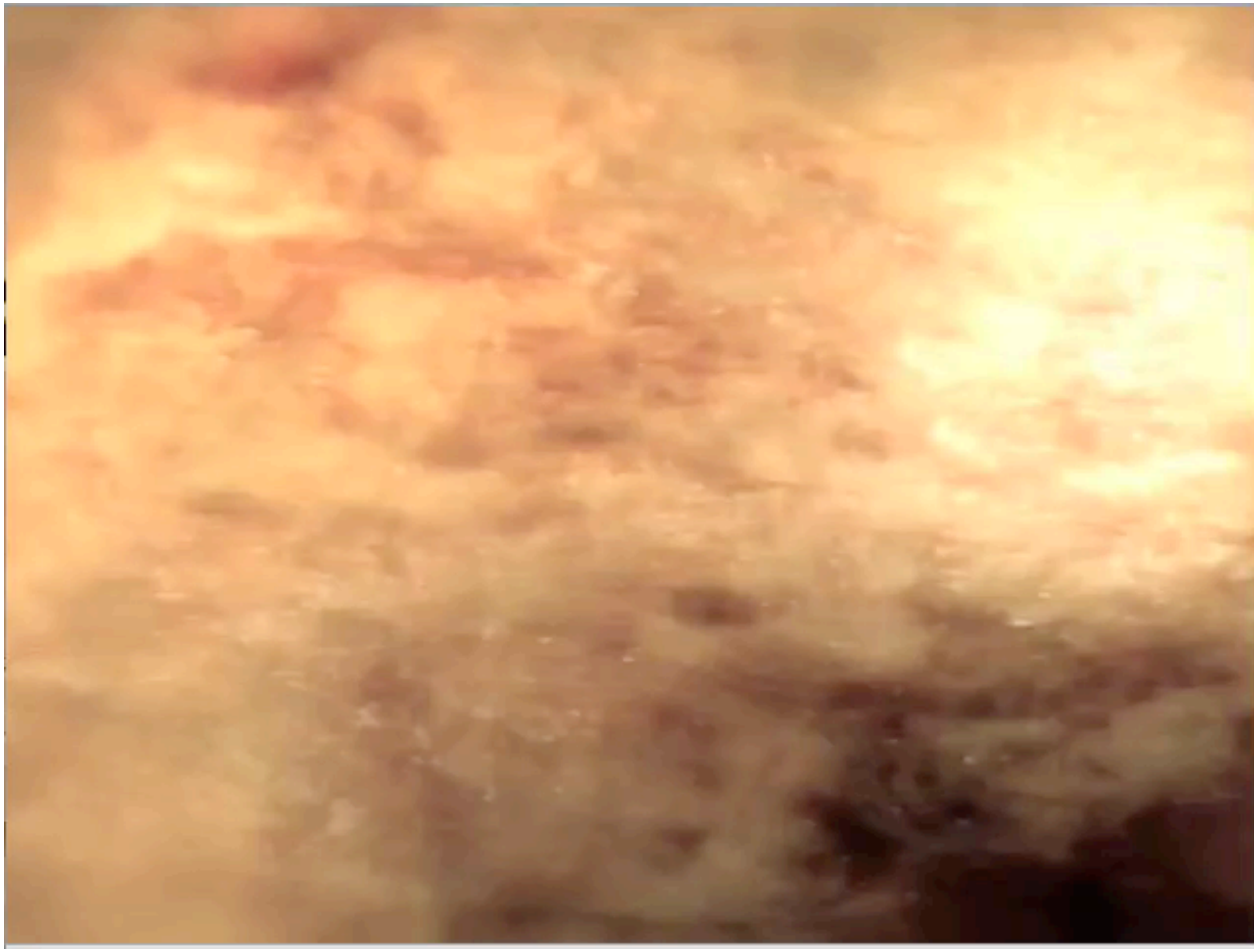
There are some common choices however. For instance every implementation of map, multimap, set and multiset that I have ever seen uses a structure called a red black tree. Typing 'red black tree algorithm' in google produces a number of likely looking links.

john

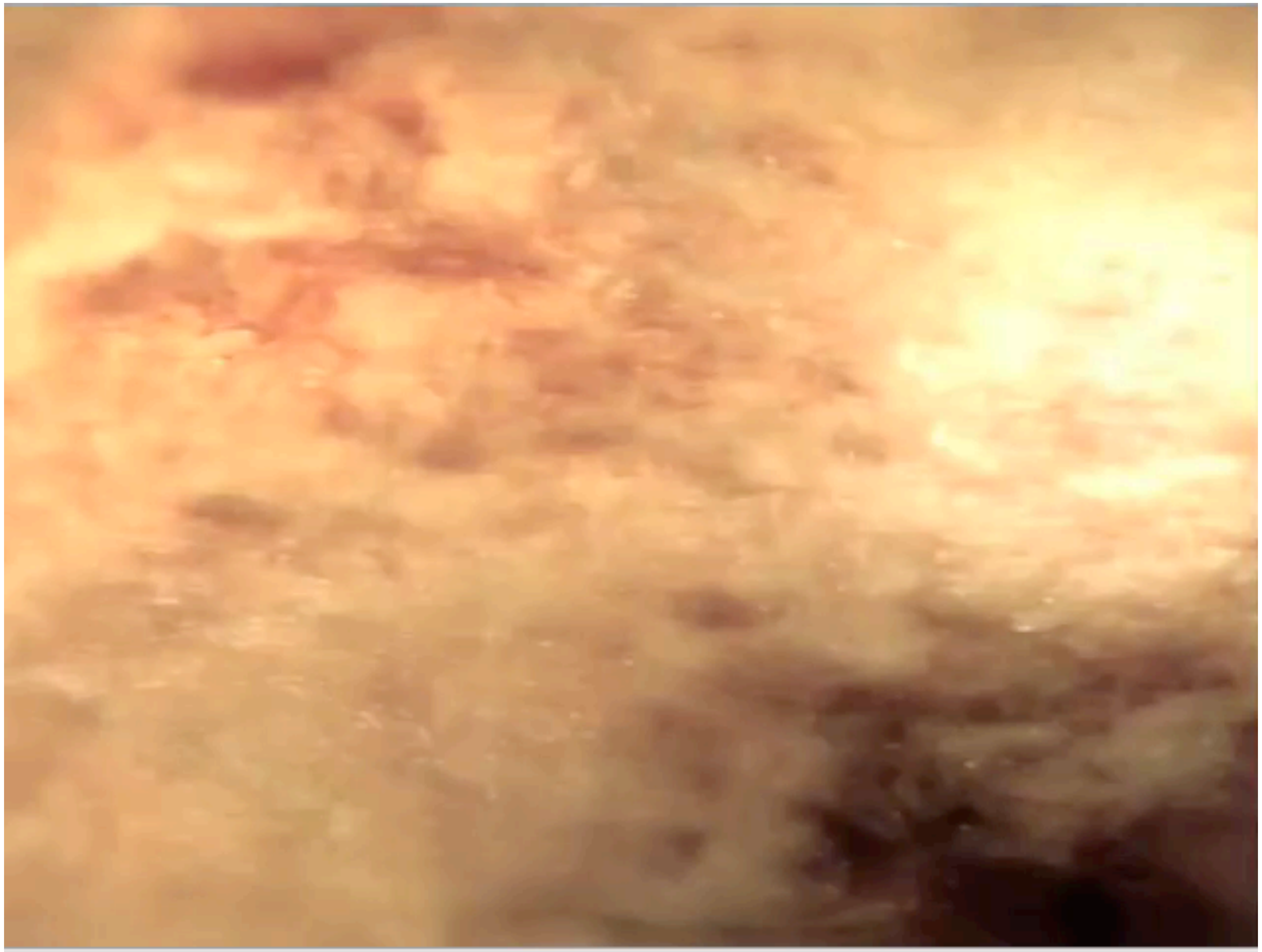
Digression:

Red-black trees are found in popular culture??

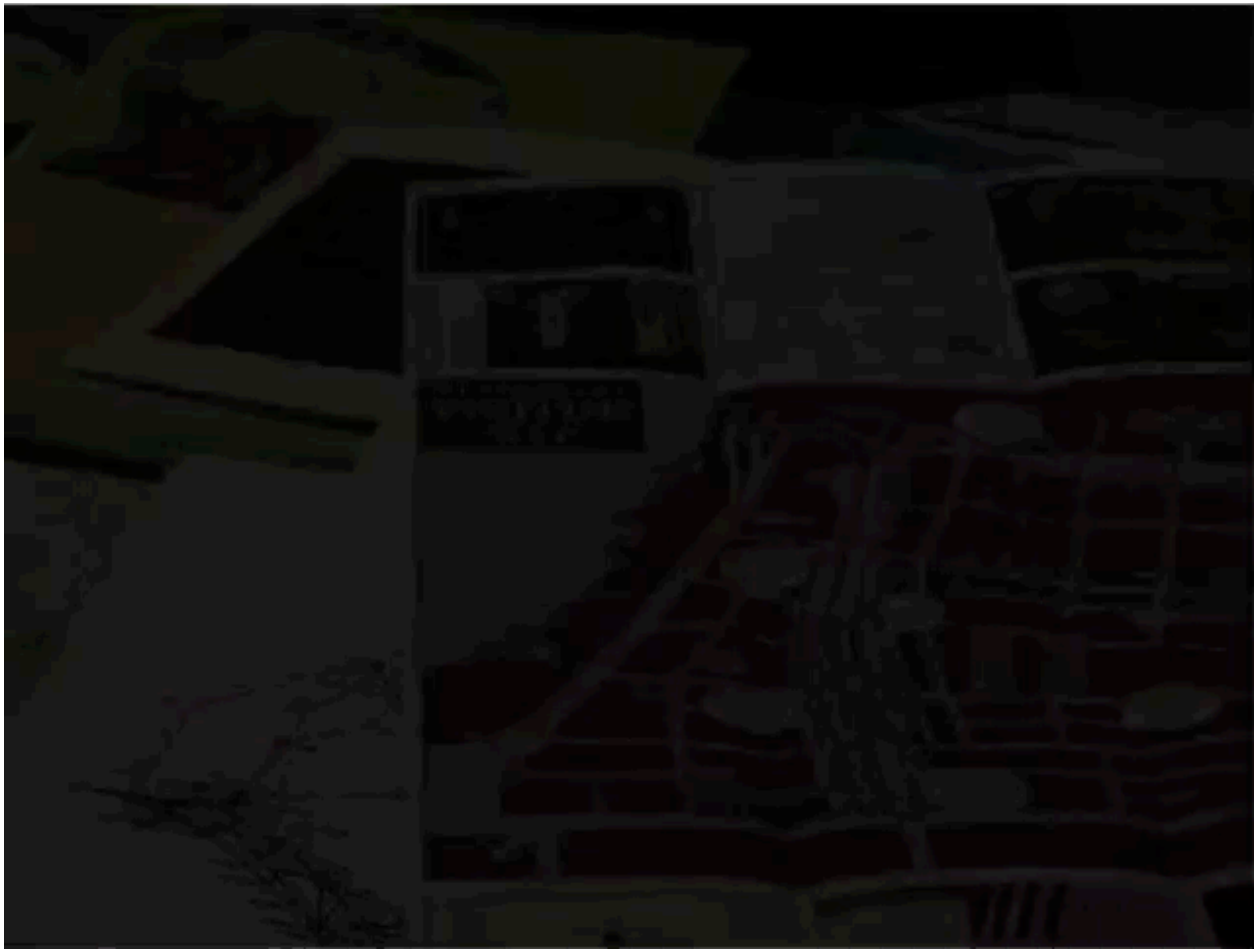
Mystery: black door?



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.

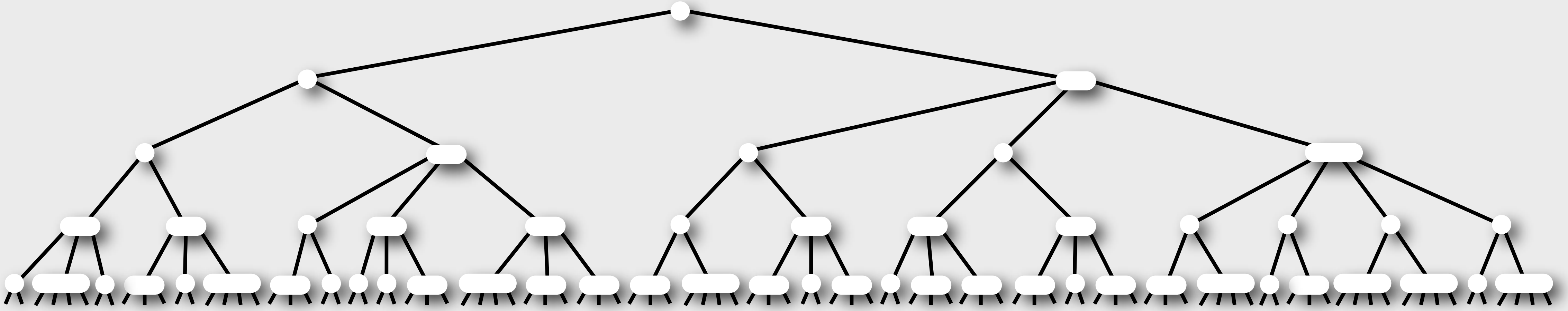
Introduction

2-3-4 Trees

LLRB Trees

Deletion

Analysis



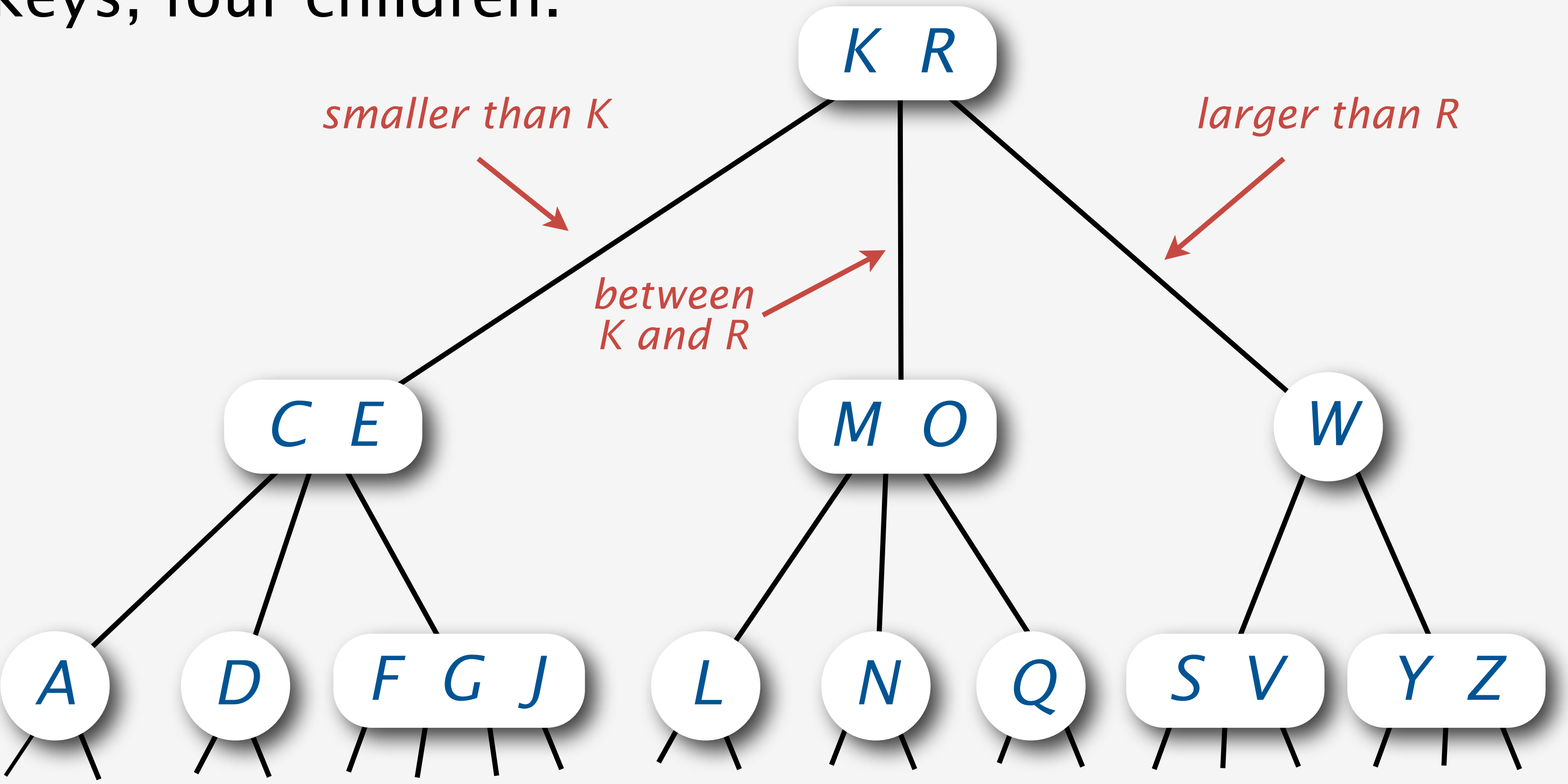
2-3-4 Tree

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.



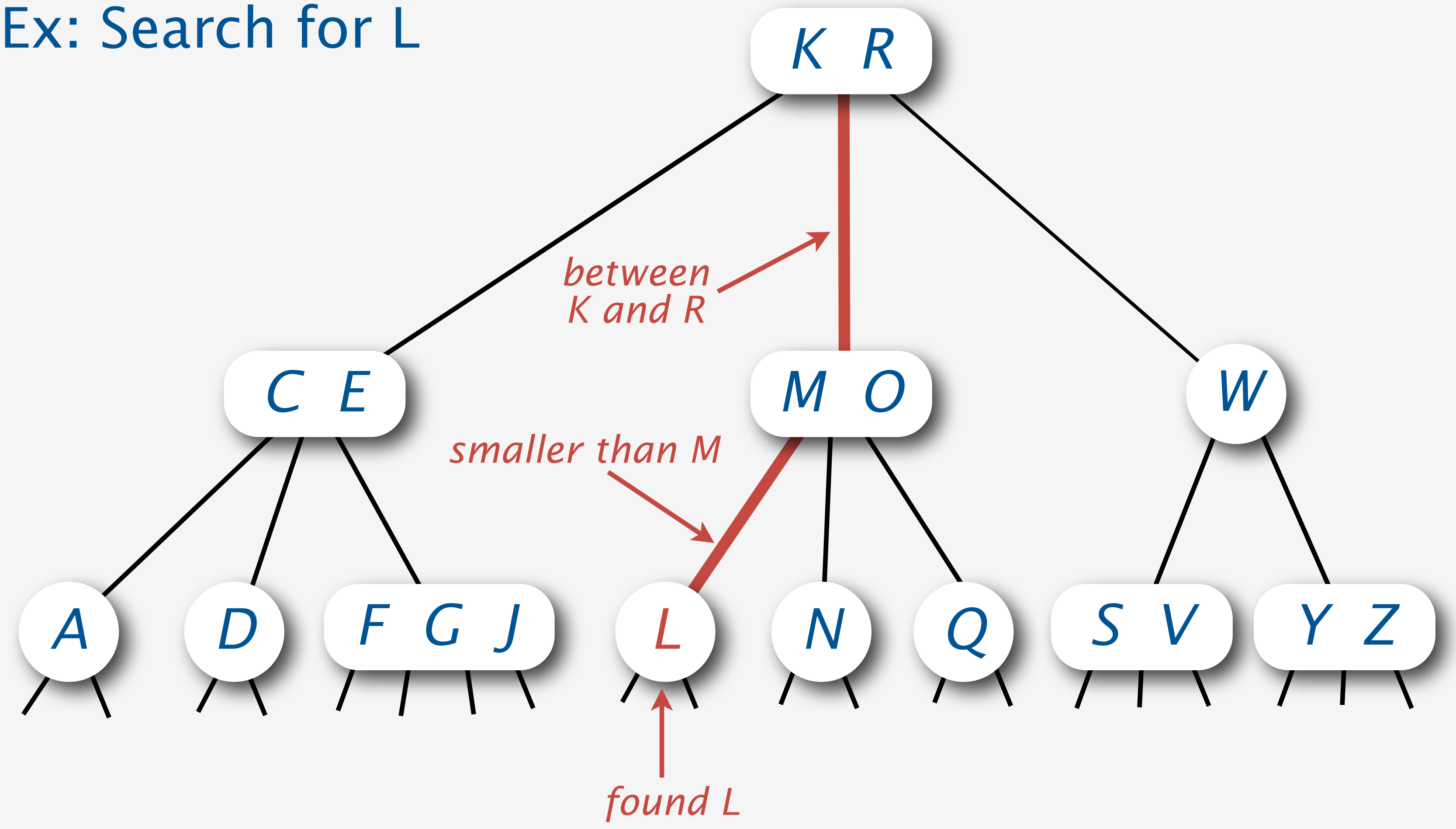
Search in a 2-3-4 Tree

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



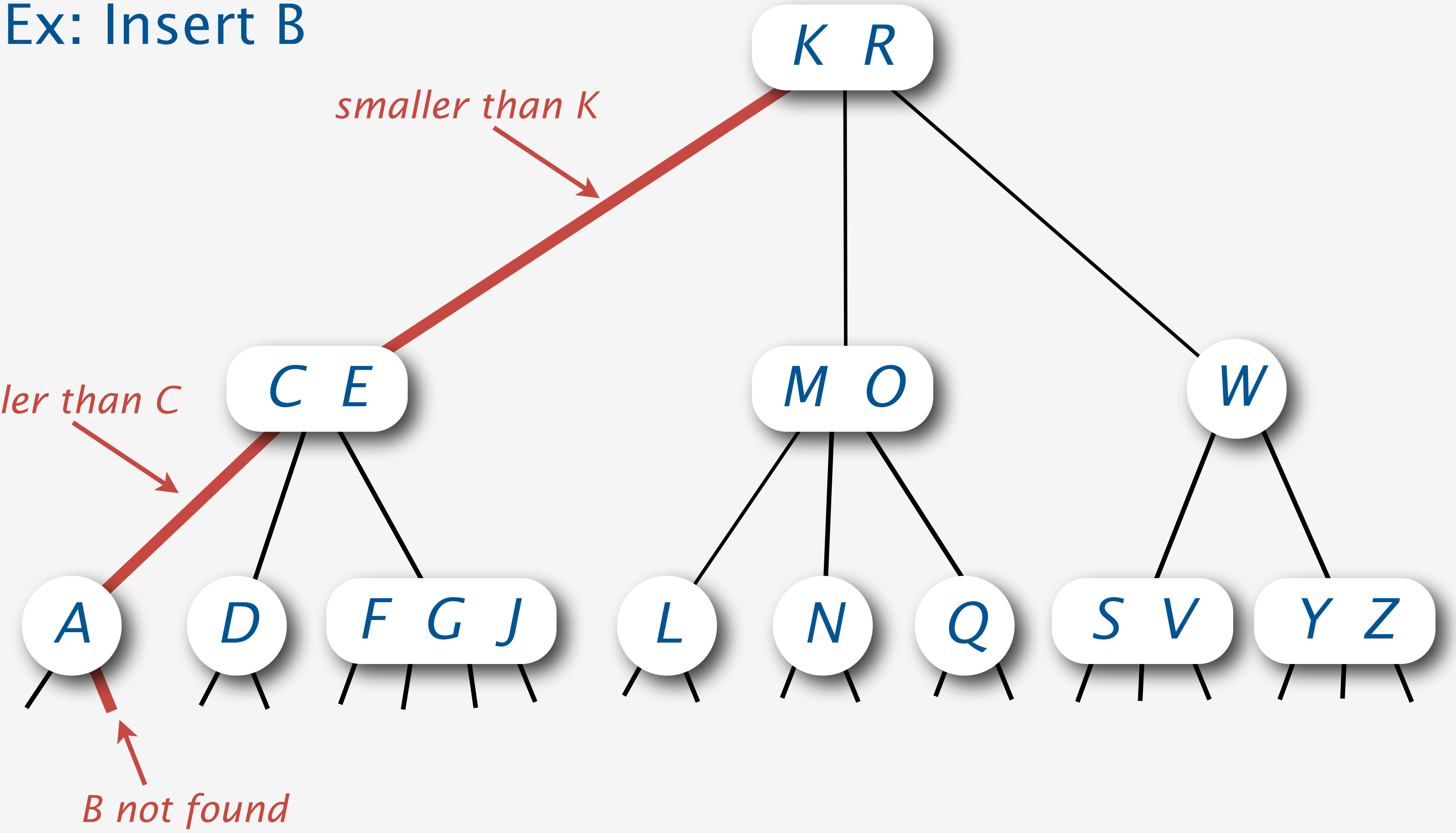
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

- Search to bottom for key.

Ex: Insert B



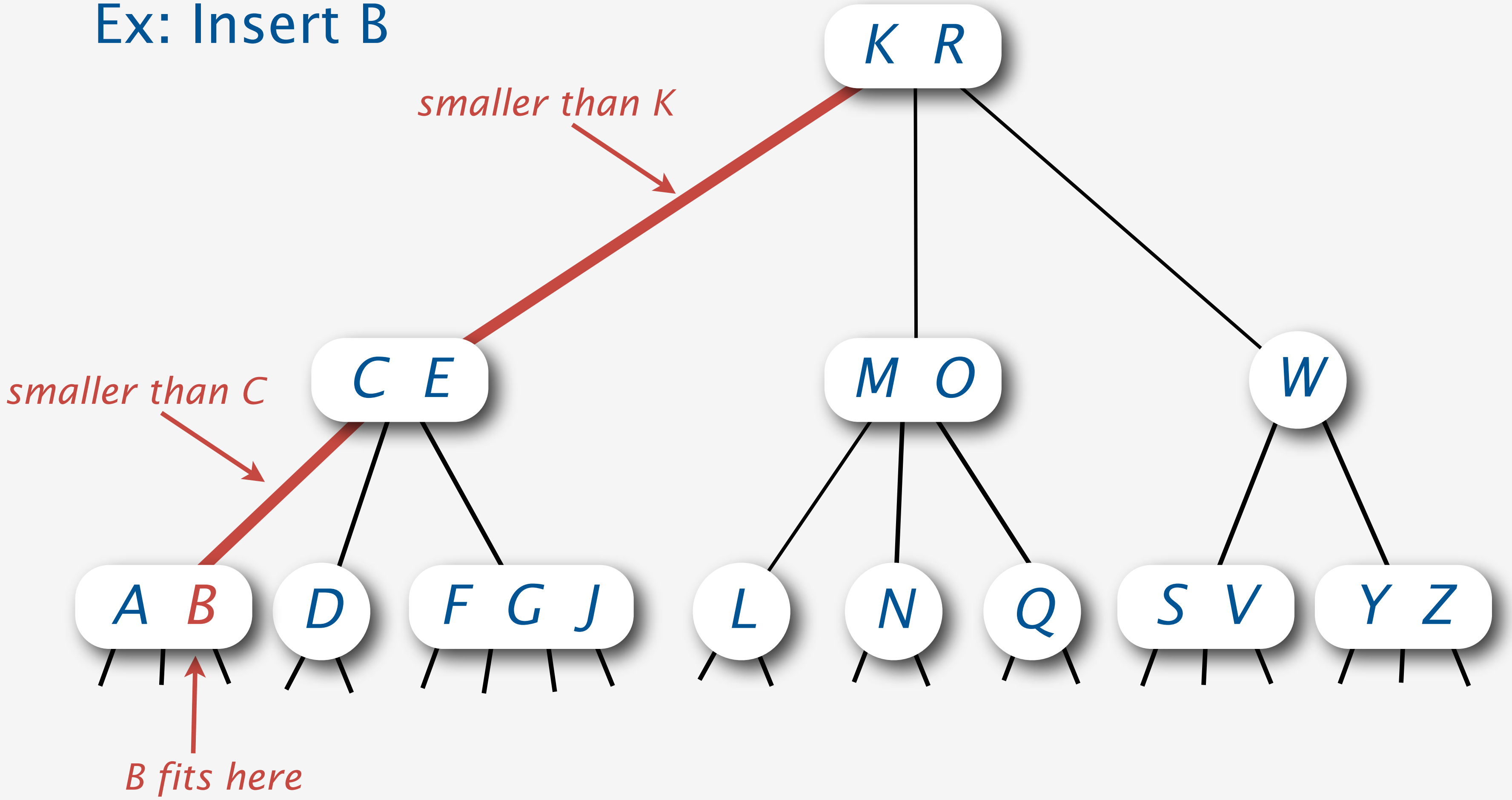
Insertion in a 2-3-4 Tree

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



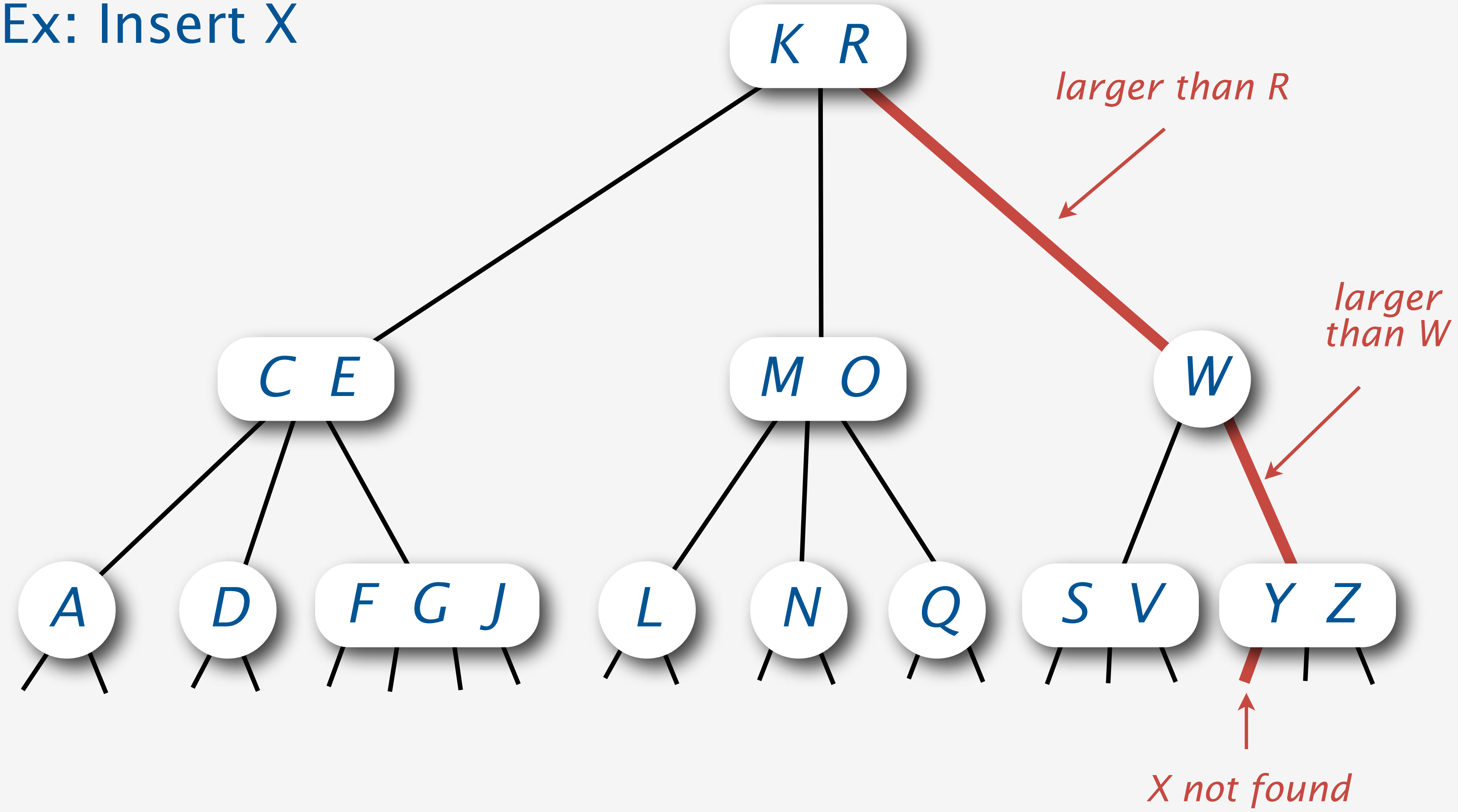
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

- Search to bottom for key.

Ex: Insert X



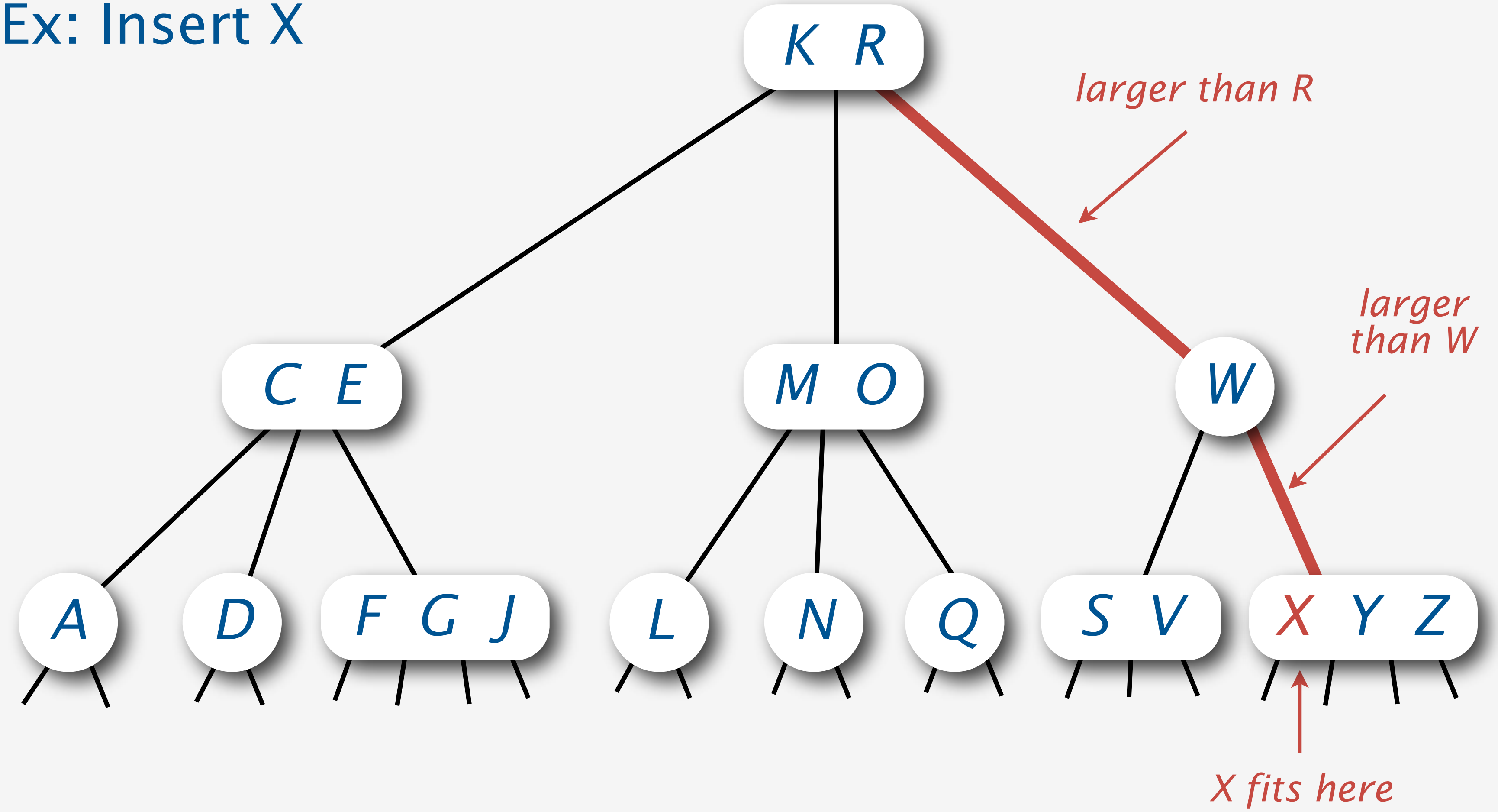
Insertion in a 2-3-4 Tree

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



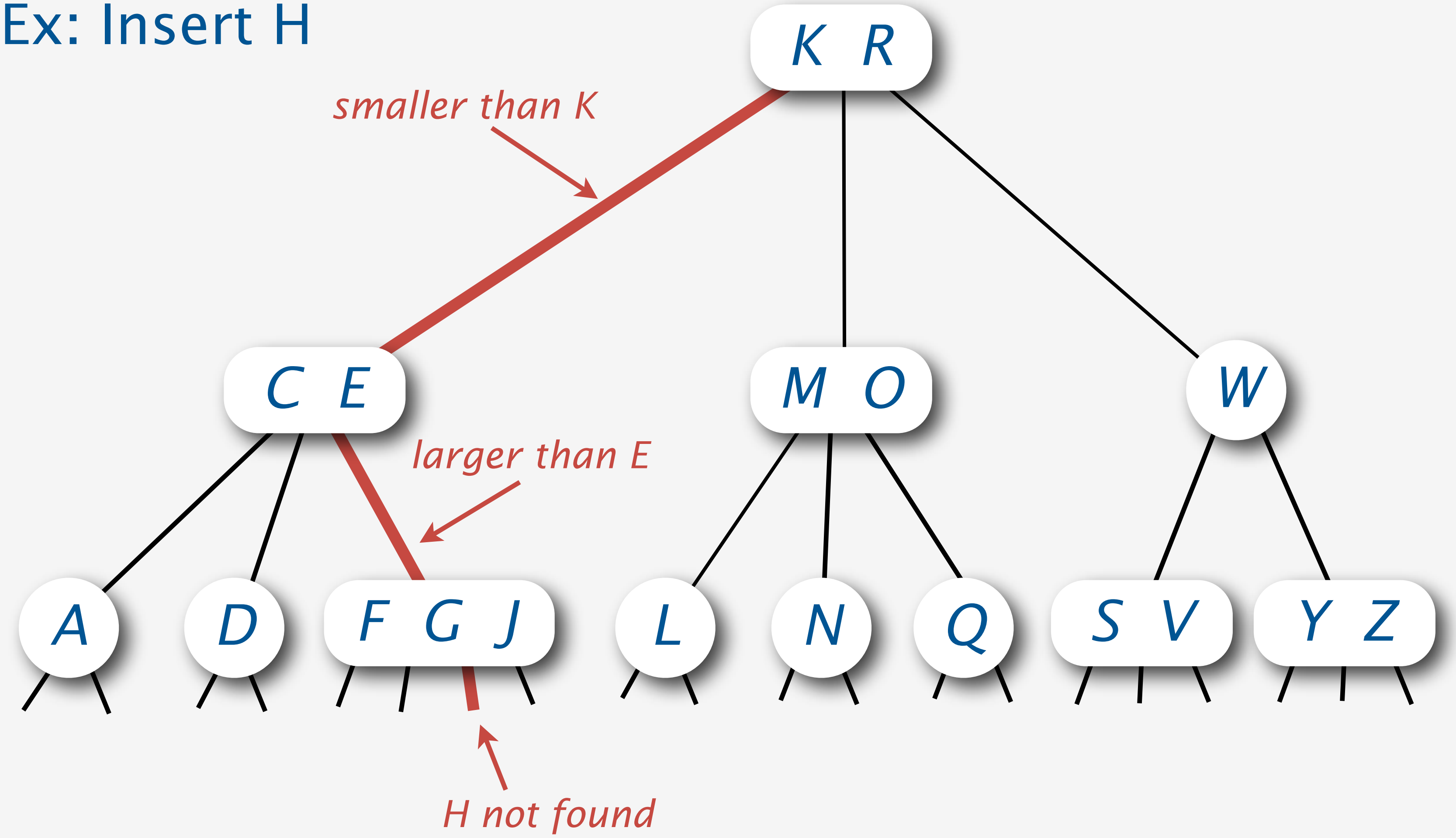
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

- Search to bottom for key.

Ex: Insert H



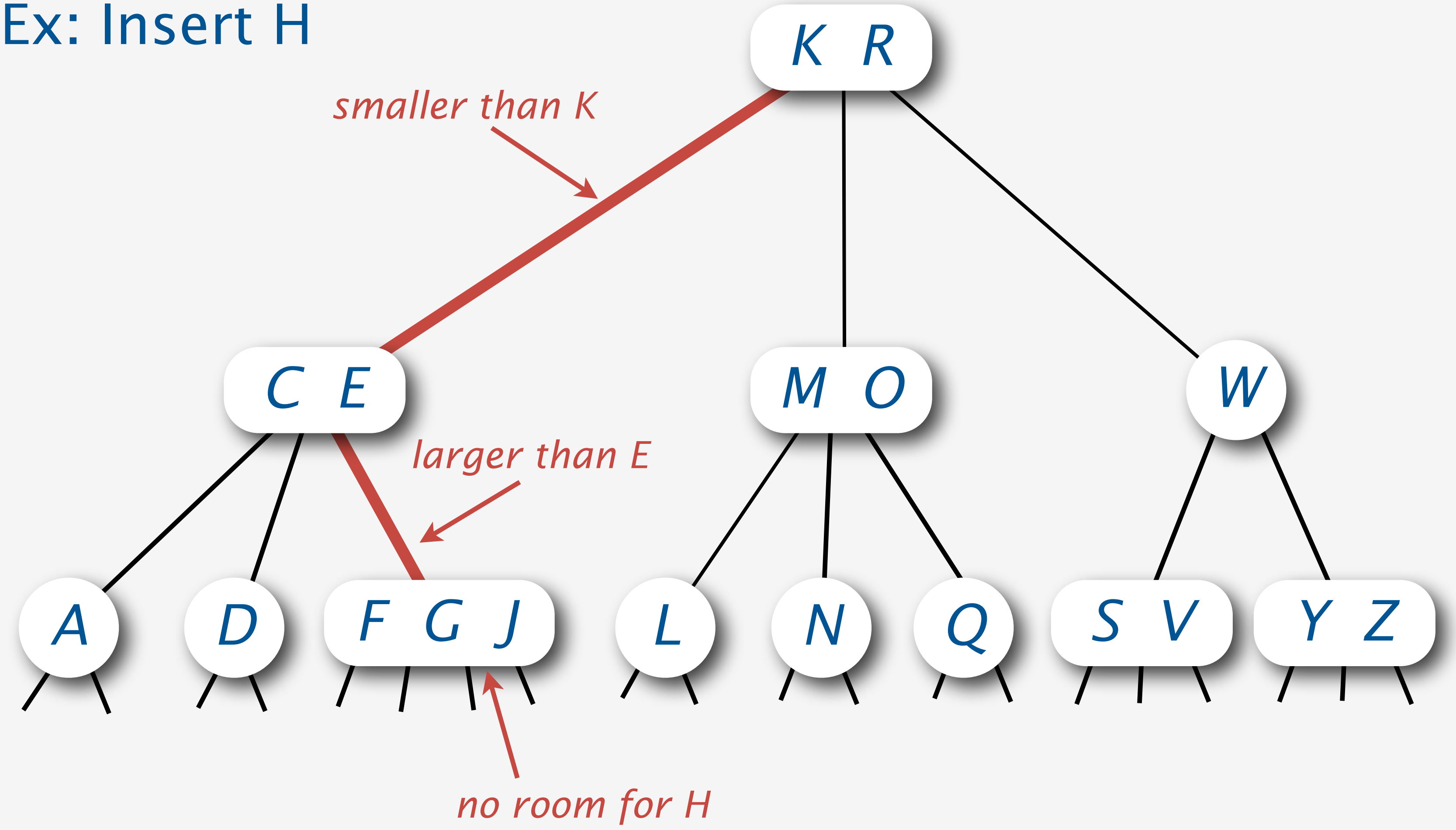
Insertion in a 2-3-4 Tree

Add new keys at the bottom of the tree.

Insert.

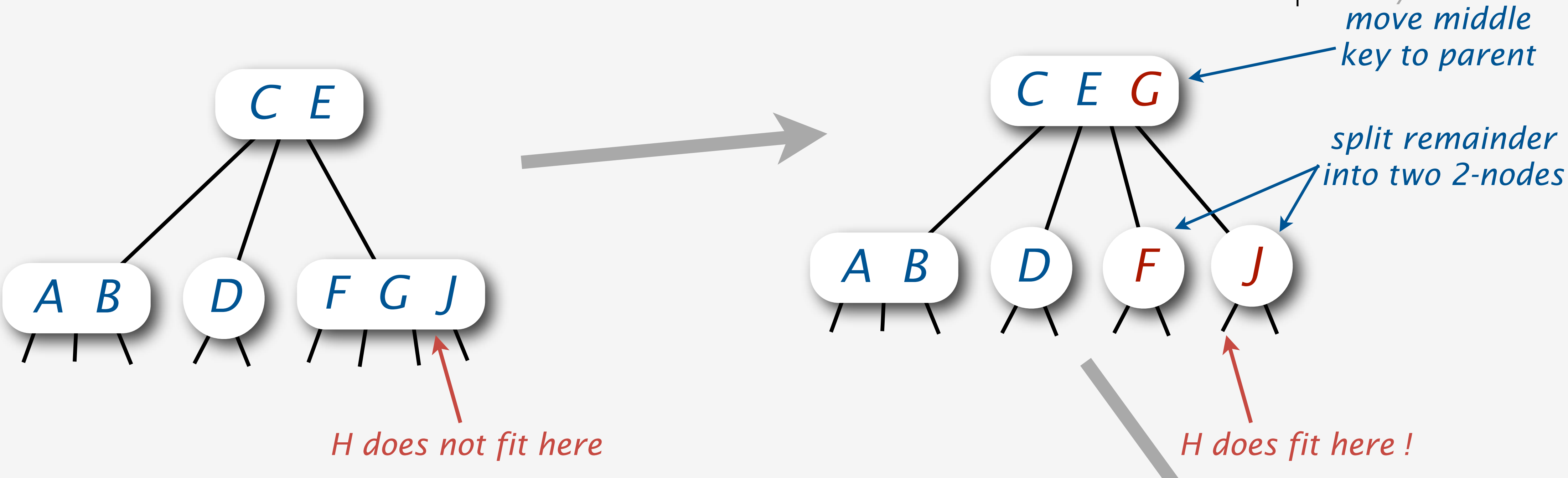
- Search to bottom for key.
- 2-node at bottom: convert to a 3-node.
- 3-node at bottom: convert to a 4-node.
- 4-node at bottom: no room for new key.

Ex: Insert H



Splitting 4-nodes in a 2-3-4 tree

is an effective way to make room for insertions

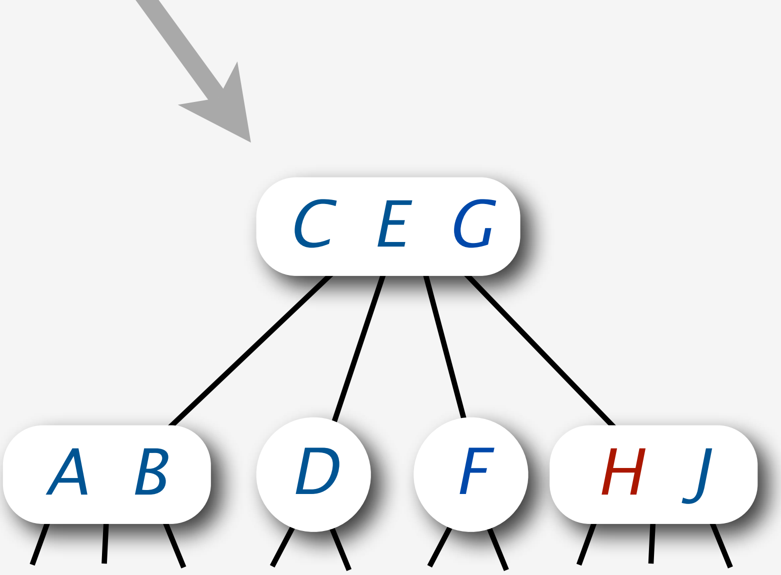


move middle key to parent

split remainder into two 2-nodes

H does not fit here

H does fit here !



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

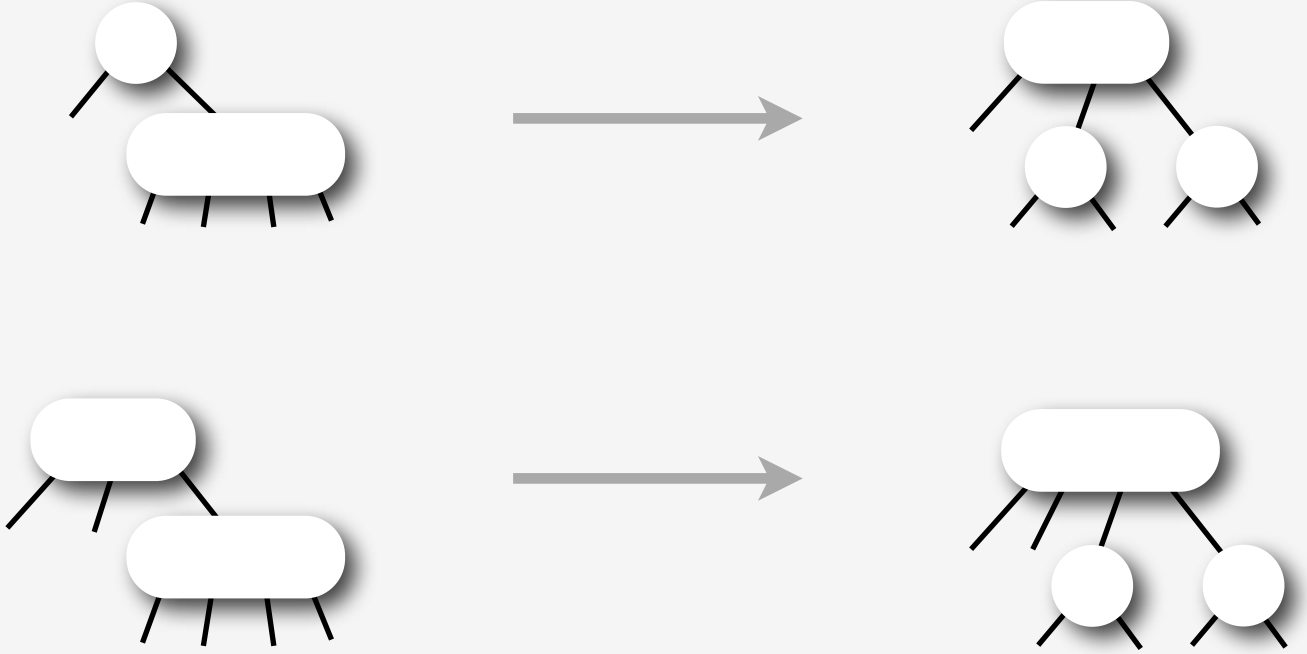
Top-down solution (Guibas-Sedgewick, 1978)

- 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:



local transformations that work anywhere in the tree

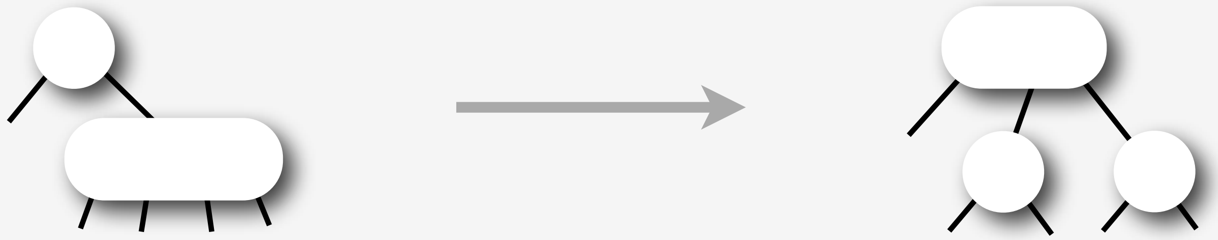
Invariant: "Current" node is not a 4-node

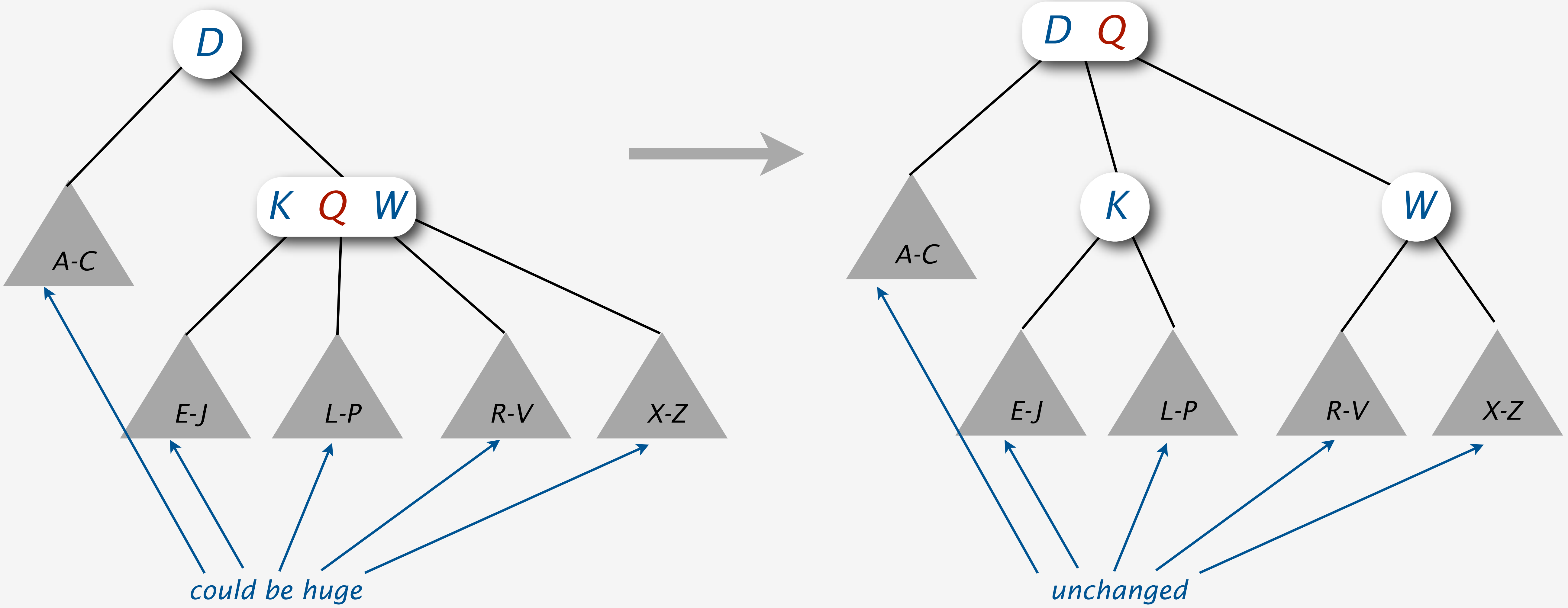
Consequences:

- 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

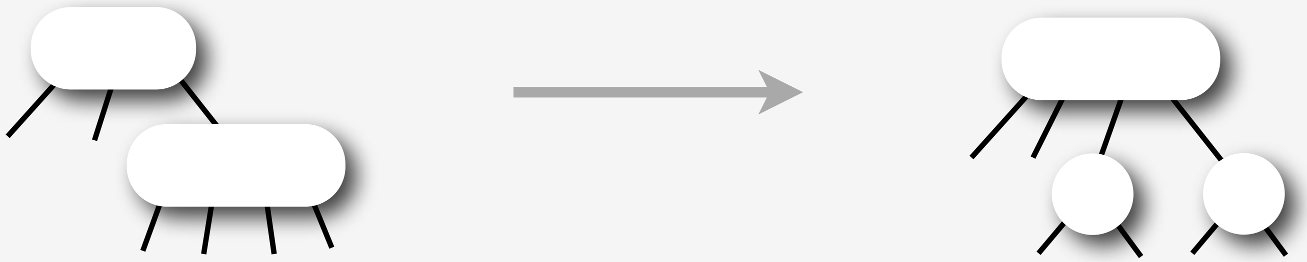
is a local transformation that works anywhere in the tree

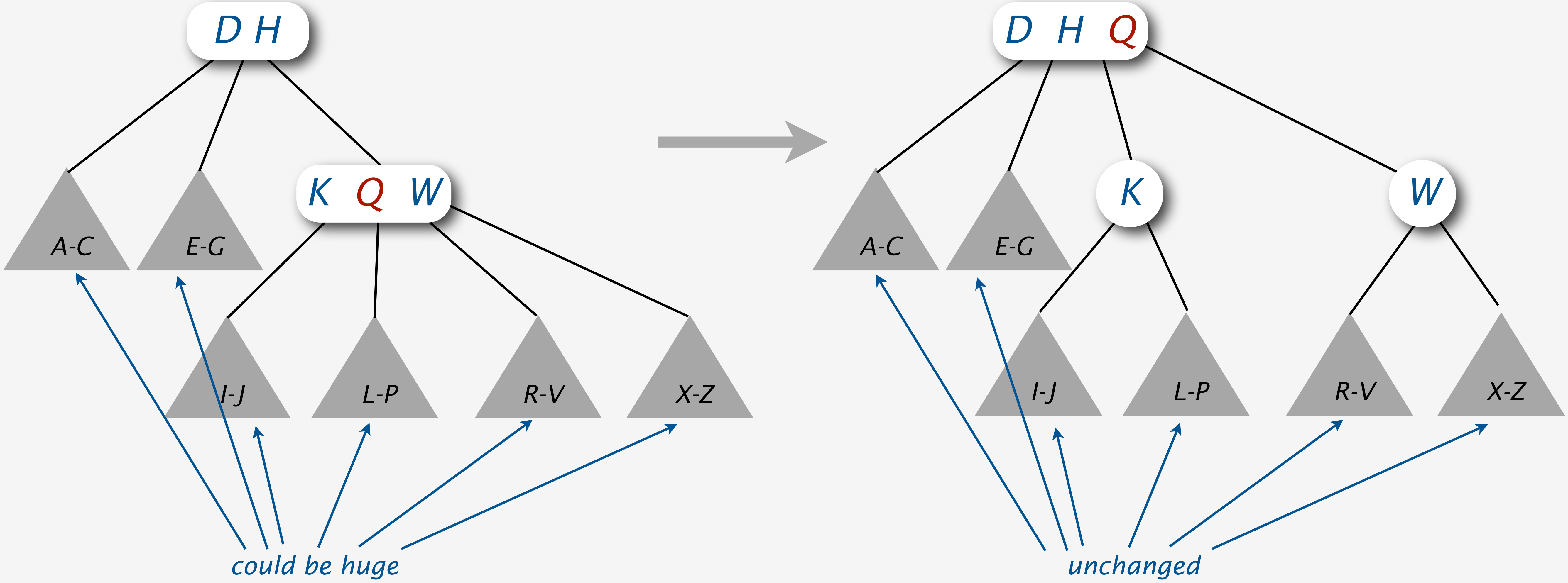




Splitting a 4-node below a 3-node

is a local transformation that works anywhere in the tree

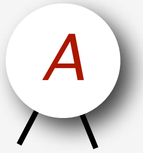




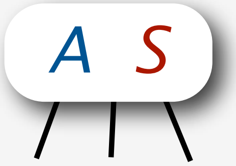
Growth of a 2-3-4 tree

happens upwards from the bottom

insert A

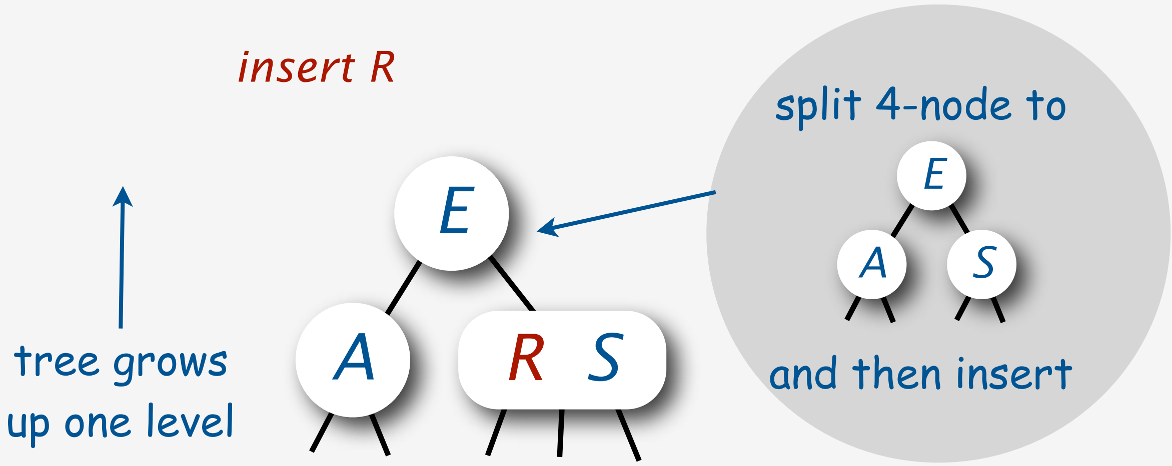


insert S



insert E

insert R

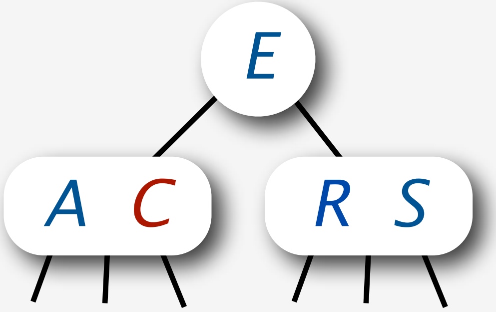


split 4-node to

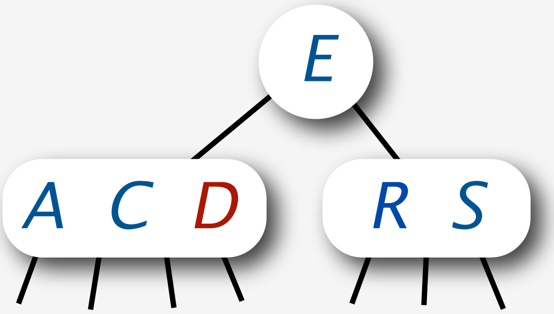
and then insert

tree grows up one level

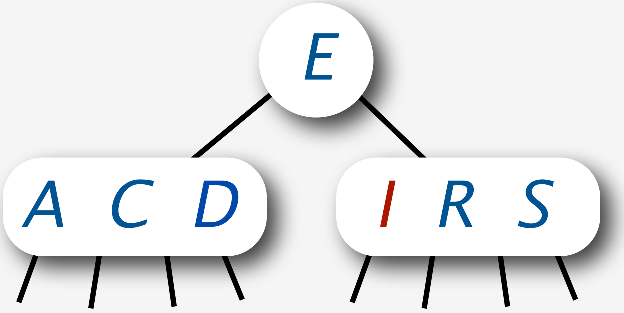
insert C



insert D

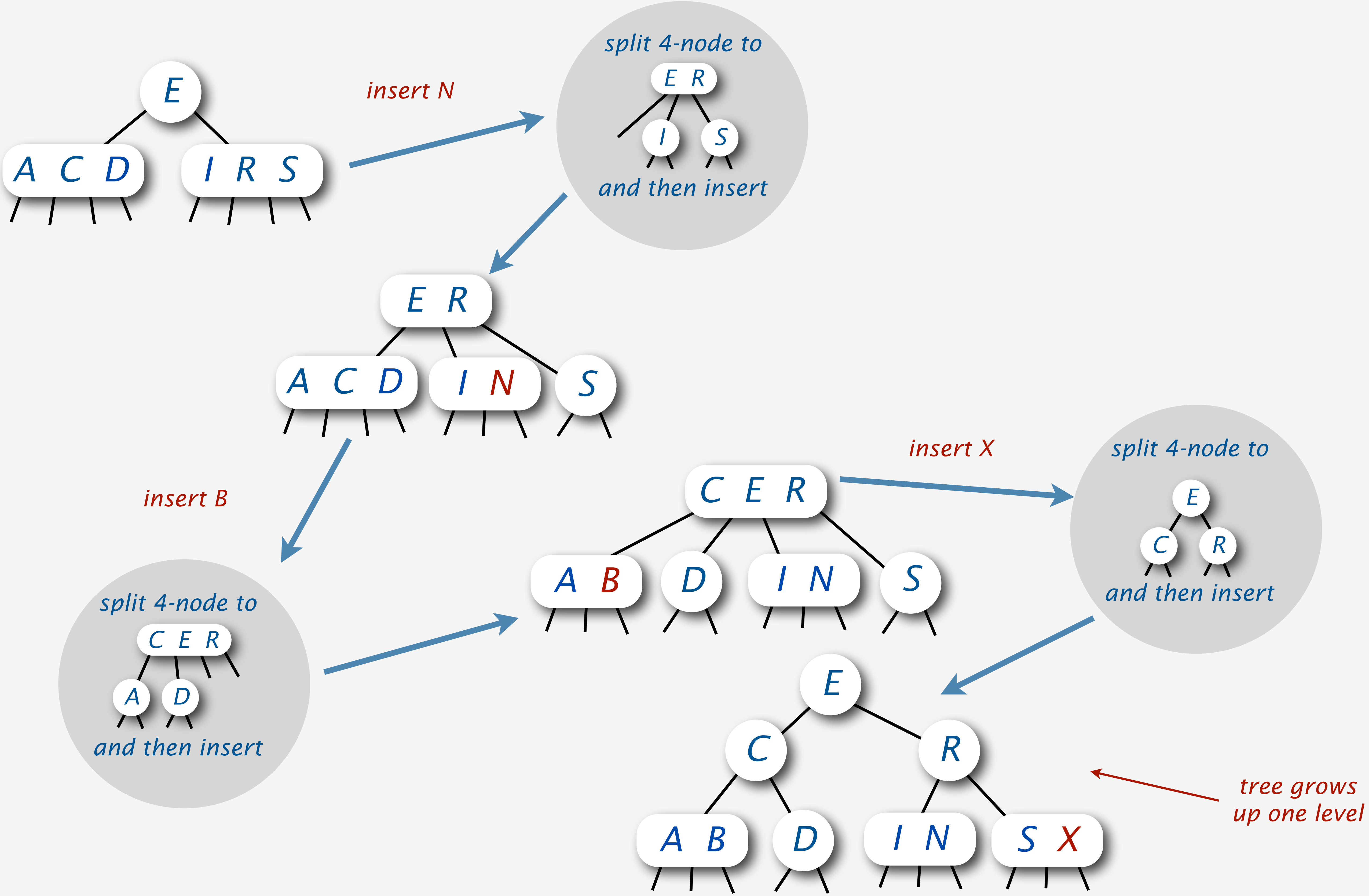


insert I



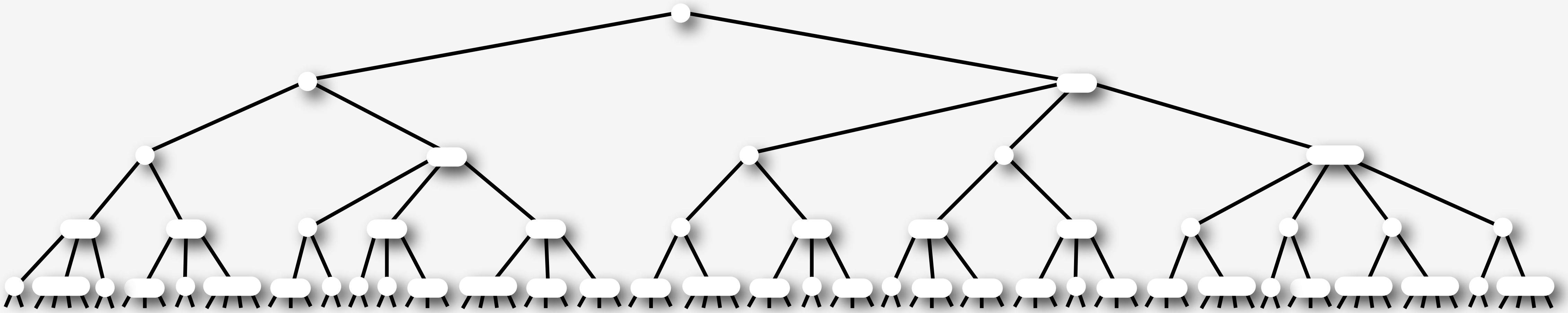
Growth of a 2-3-4 tree (continued)

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: log4 N = 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.

fantasy code

```
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.

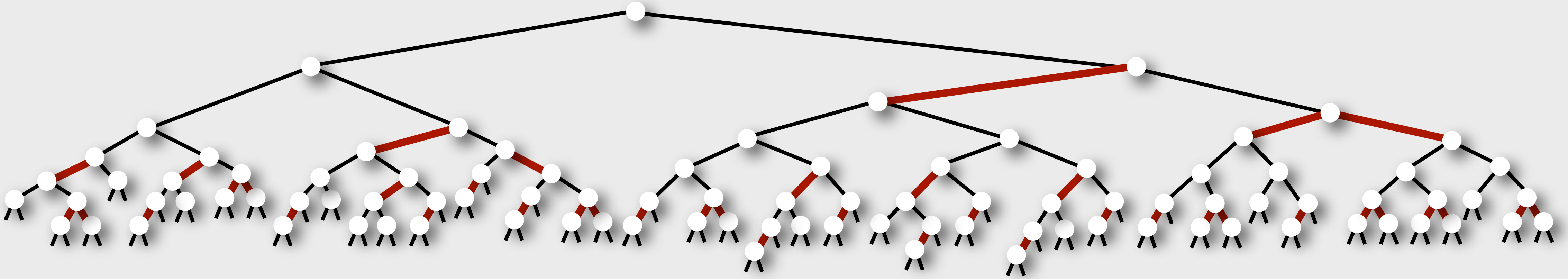
Introduction

2-3-4 Trees

LLRB Trees

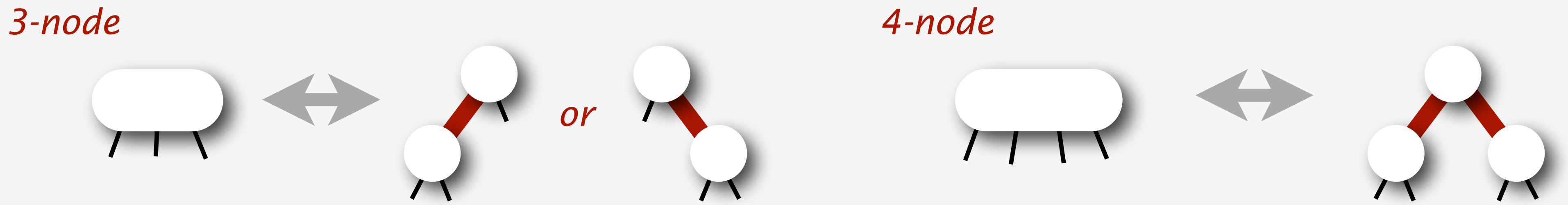
Deletion

Analysis



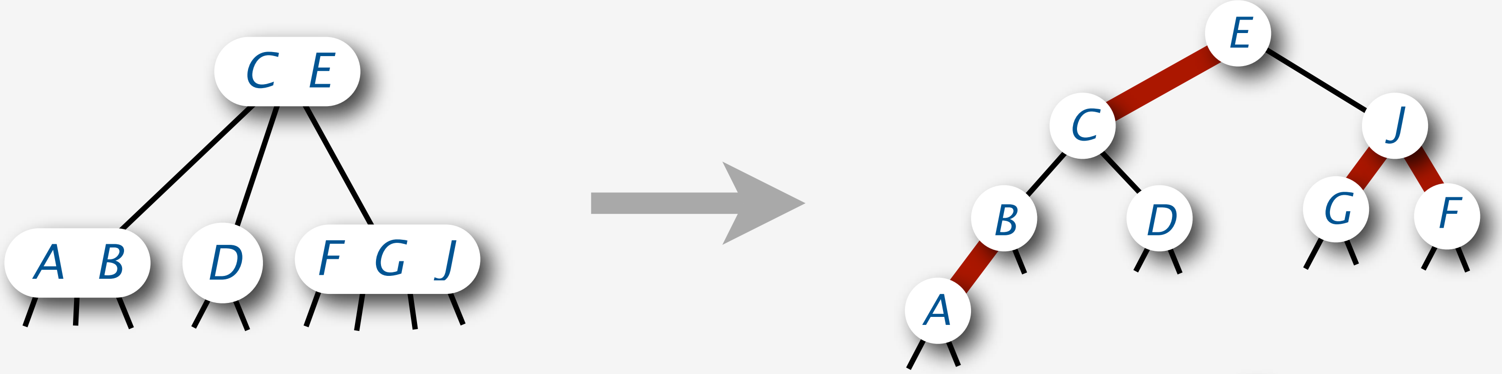
Red-black trees (Guibas-Sedgewick, 1978)

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

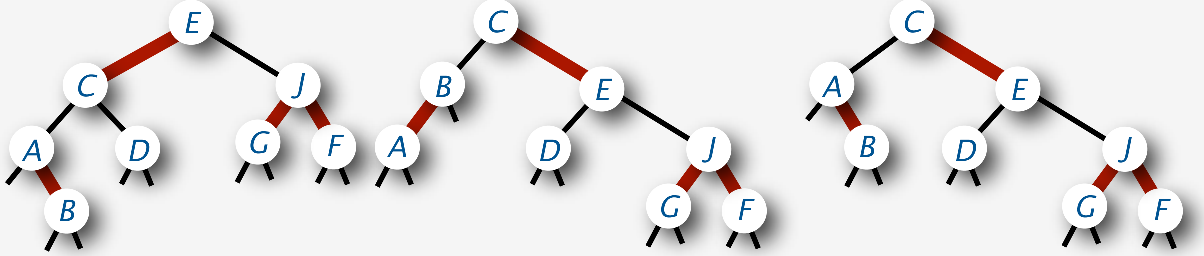


Key Properties

- 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)



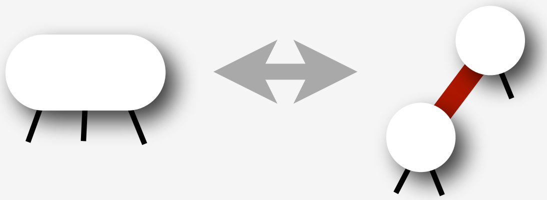
Many variants studied (details omitted.)

NEW VARIANT (this talk): Left-leaning red-black trees

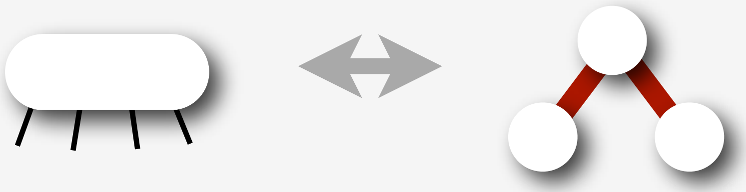
Left-leaning red-black trees

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.

3-node

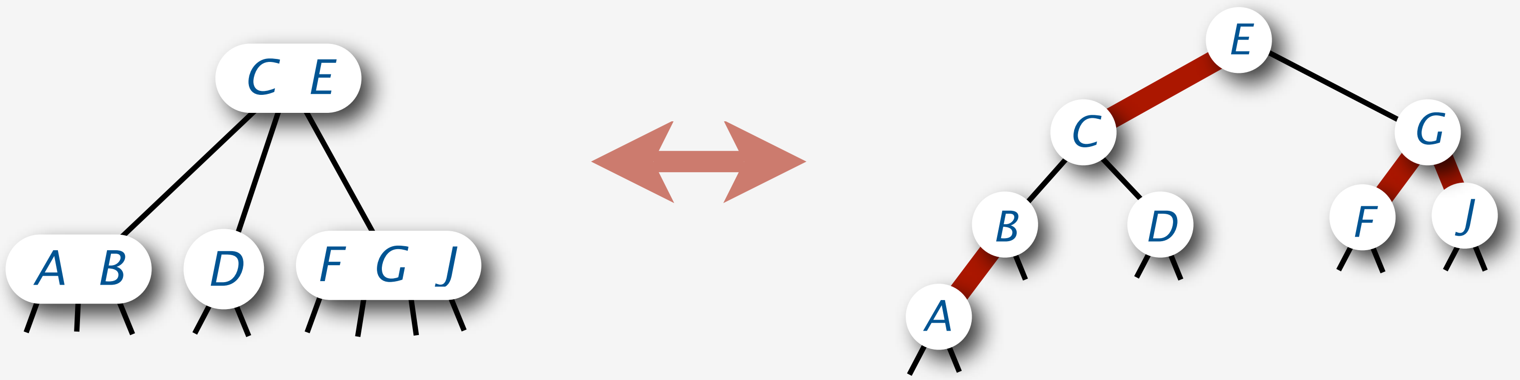


4-node



Key Properties

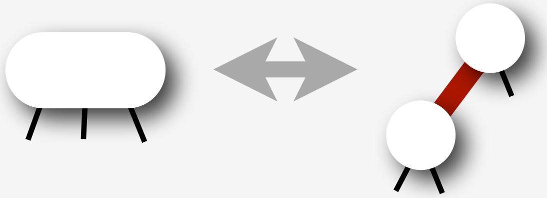
- elementary BST search works
- easy-to-maintain 1-1 correspondence with 2-3-4 trees
- trees therefore have perfect black-link balance



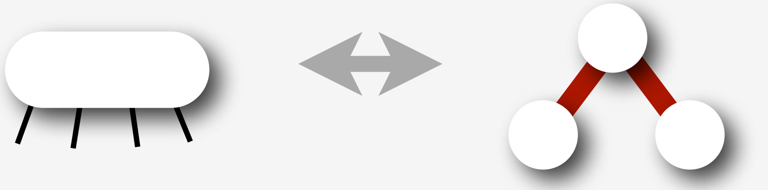
Left-leaning red-black trees

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.

3-node

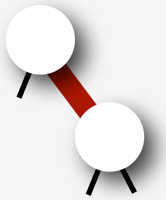


4-node



Disallowed

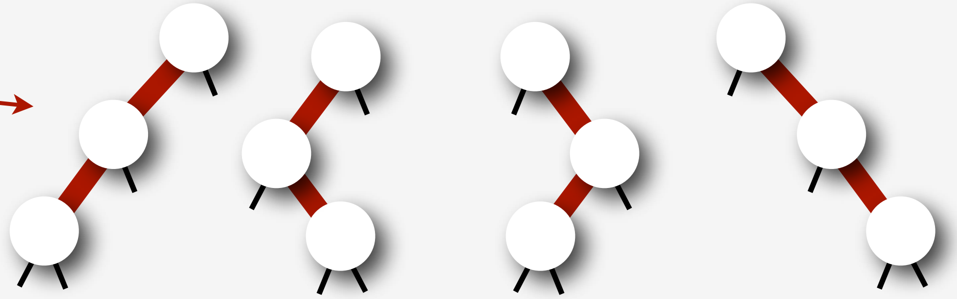
- right-leaning 3-node representation



standard red-black trees allow this one

- two reds in a row

original version of left-leaning trees used this 4-node representation



single-rotation trees allow all of these

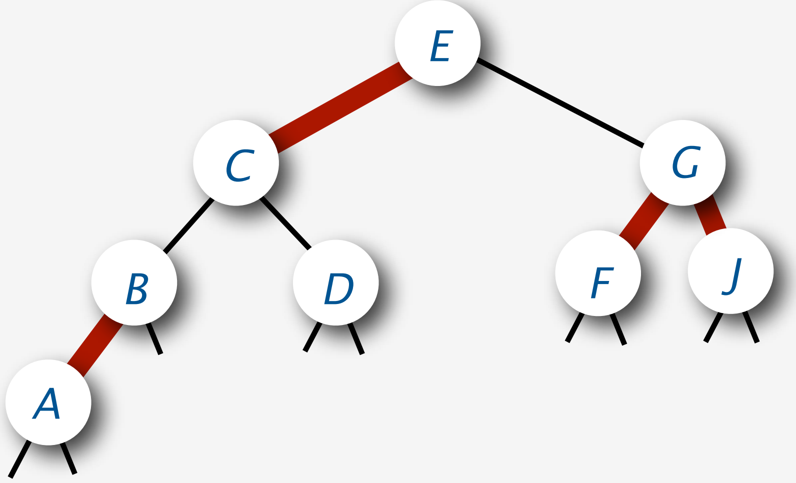
Java data structure for red-black trees

adds one bit for color to elementary BST data structure

```
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.
}
```

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);
}
```

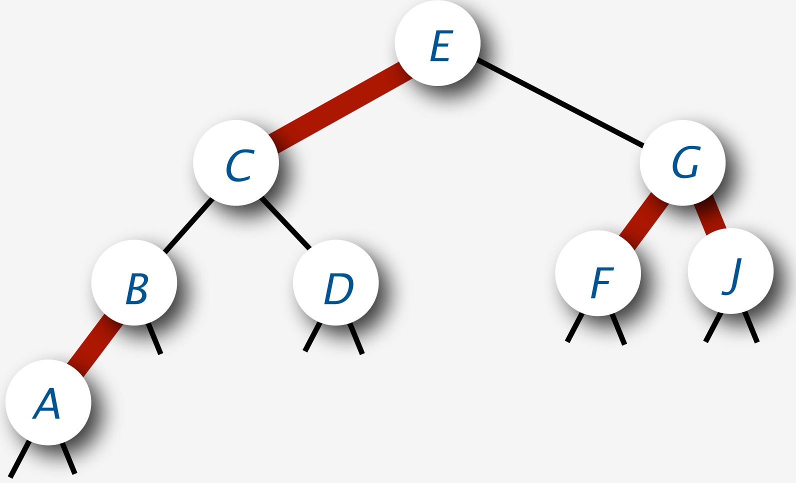
Search implementation for red-black trees

is the same as for elementary BSTs

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

BST (and LLRB tree) search implementation

```
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

```
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*

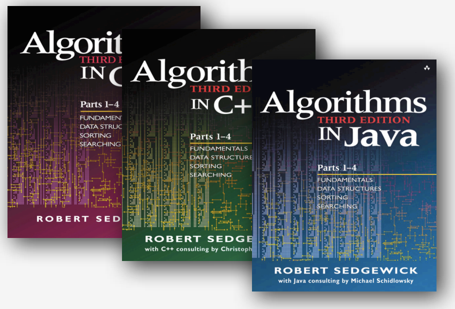
```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val);

   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;
}
```

associative model (no duplicate keys)

Nonrecursive


. . .

Recursive



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

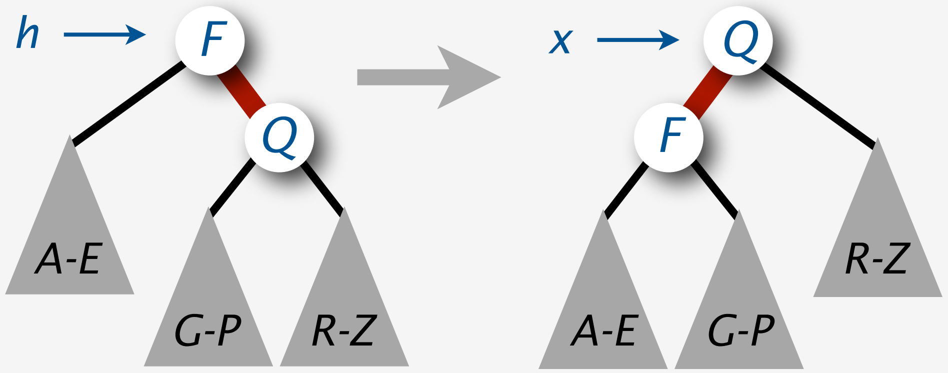
- simplifies correctness proof
- simplifies code for balanced BST implementations
- could remove recursion to get stack-based single-pass algorithm

Balanced tree code

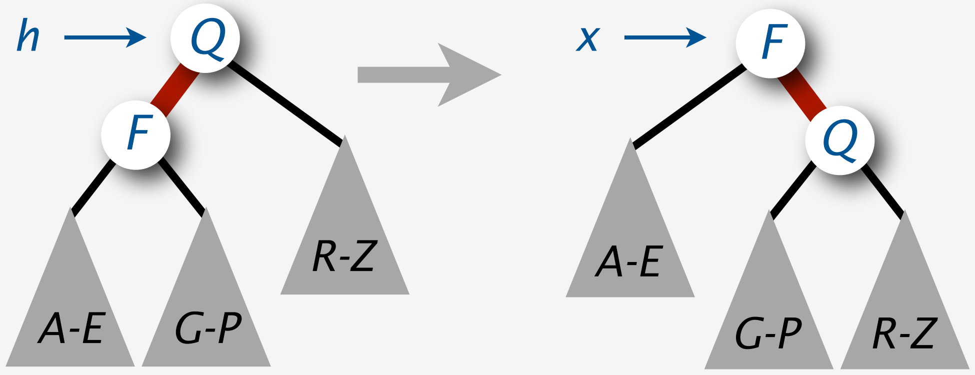
is based on local transformations known as rotations

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

```
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;
}
```



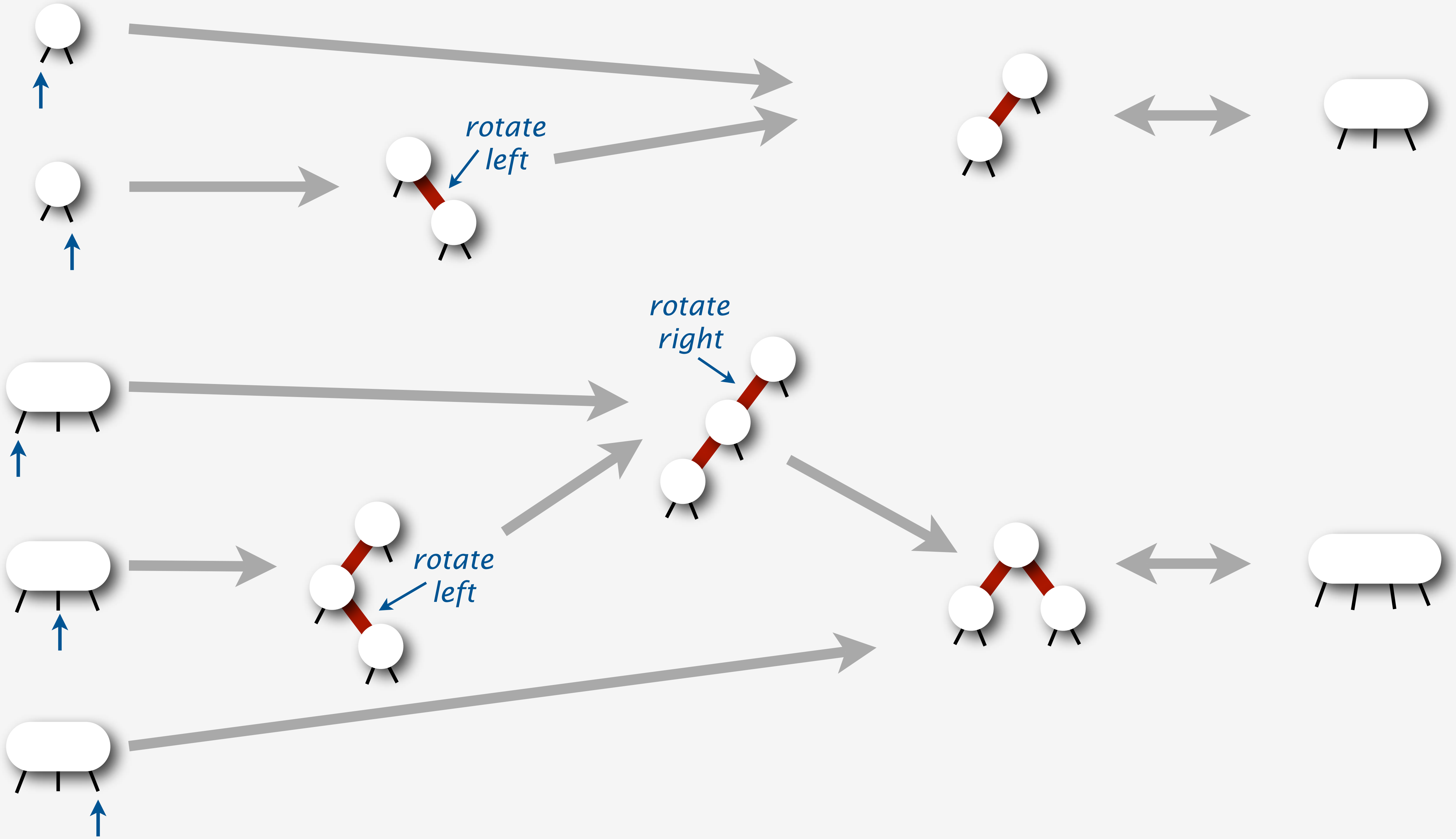
```
private Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = x.right.color;
   x.right.color = RED;
   return x;
}
```



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

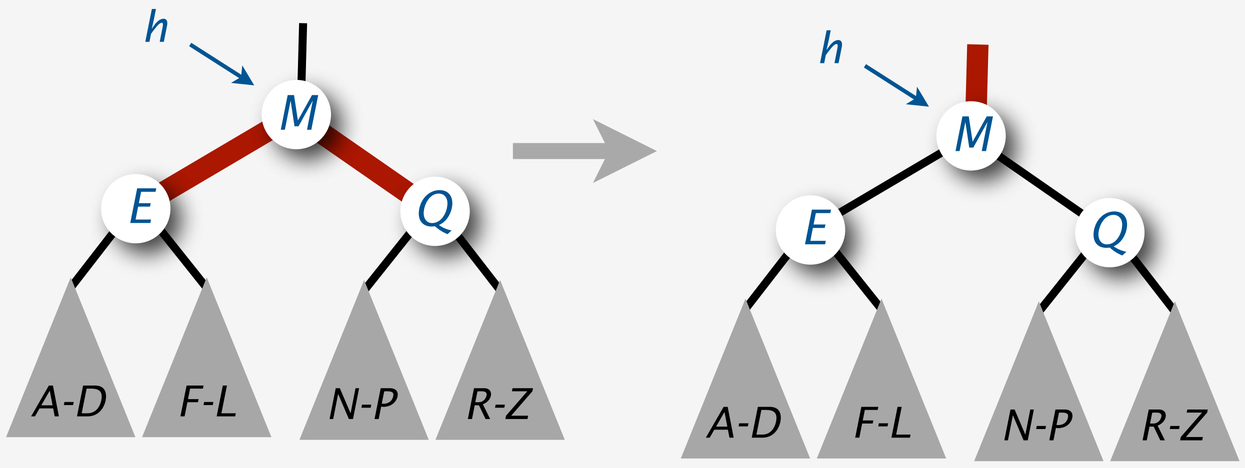


Splitting a 4-node

is accomplished with a color flip

Flip the colors of the three nodes

```
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:

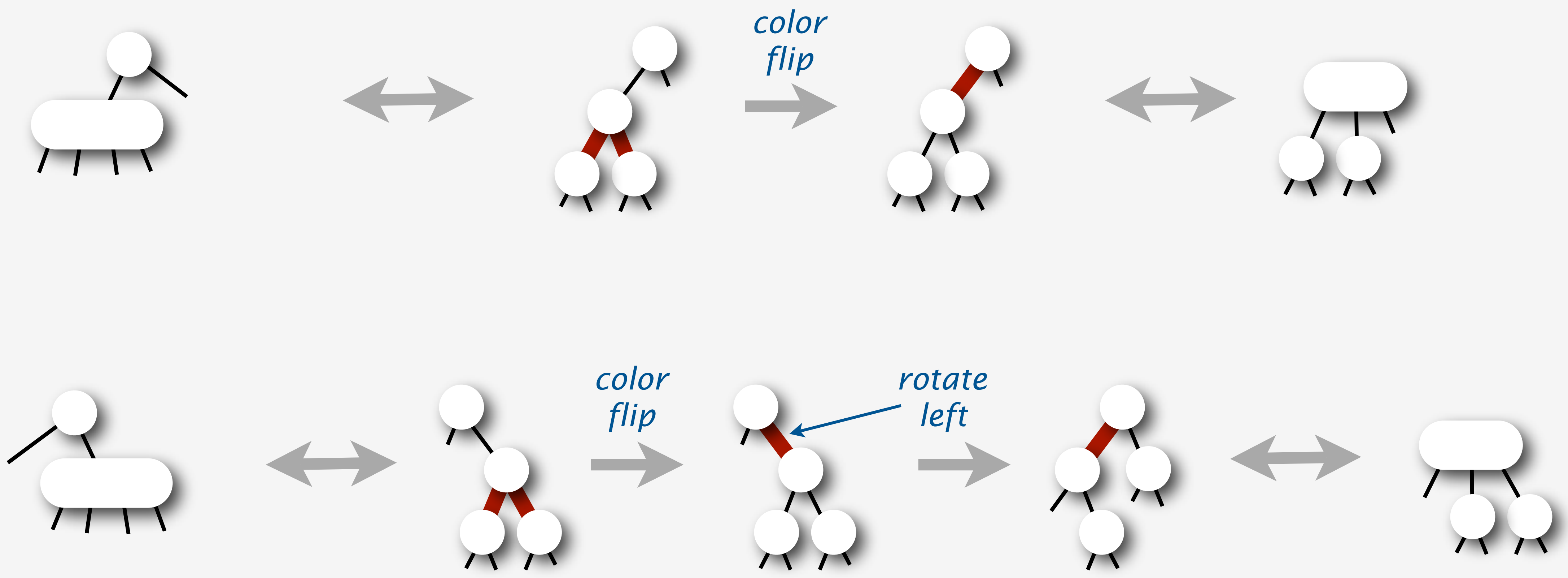
- preserves prefect black-lin balance
- passes a RED link up the tree
- reduces problem to inserting (that link) into parent

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

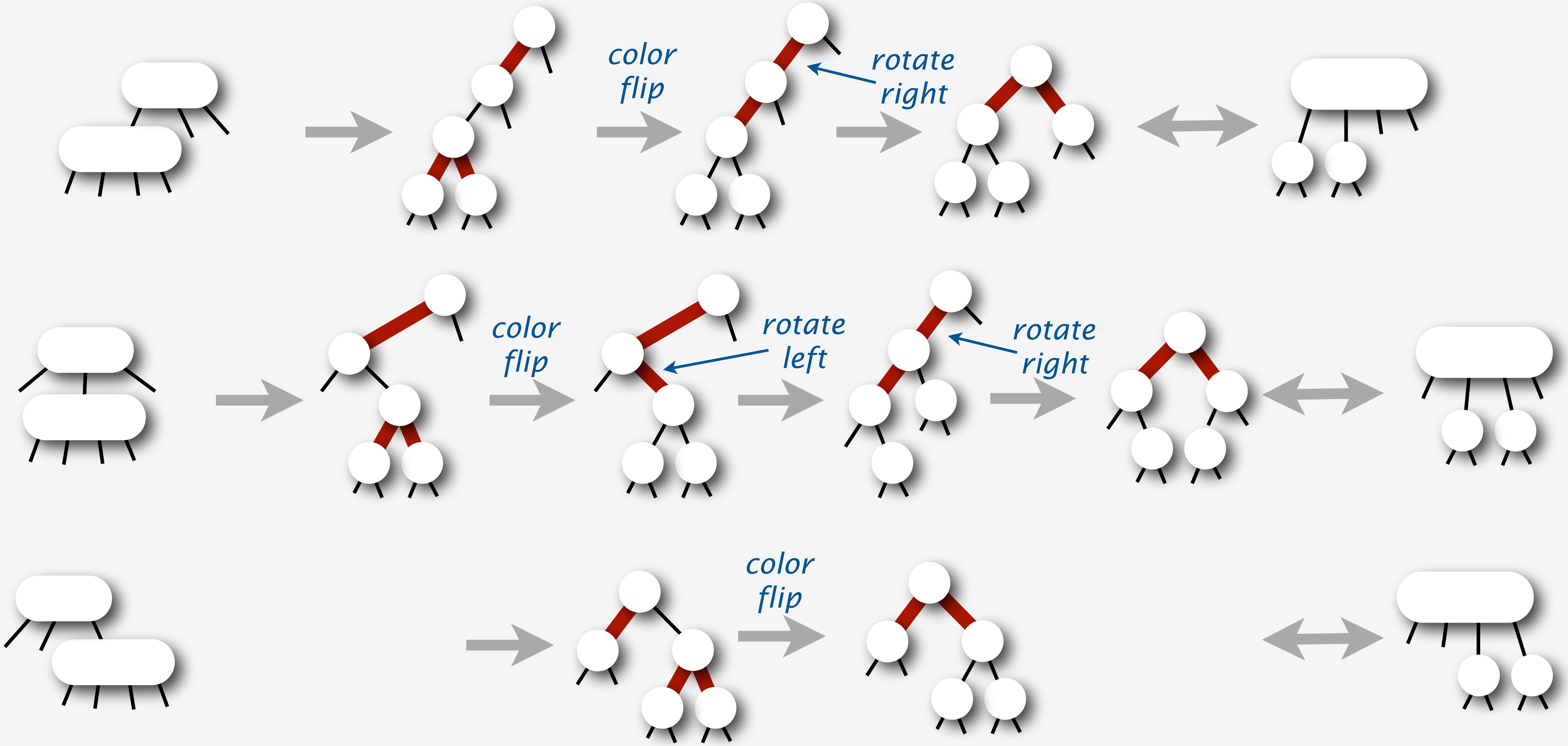


Splitting a 4-node in a LLRB tree

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

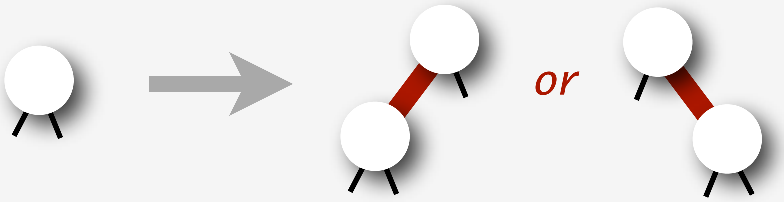
1. Flip colors, which passes red link up one level
2. Rotate if necessary to get correct representation in parent
   (using precisely the same transformations as for insert at bottom)

Parent is a 3-node: three cases



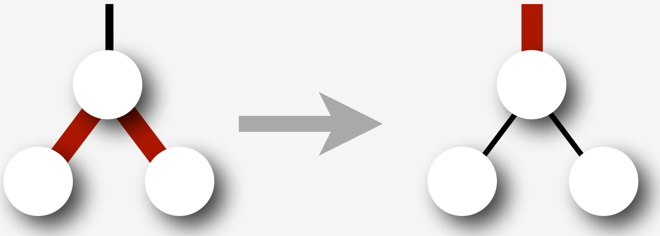
Inserting and splitting nodes in LLRB trees

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



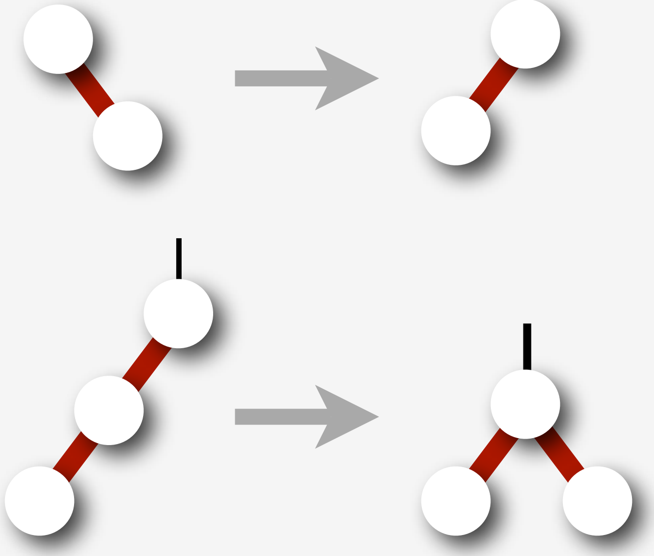
Search as usual

- 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.

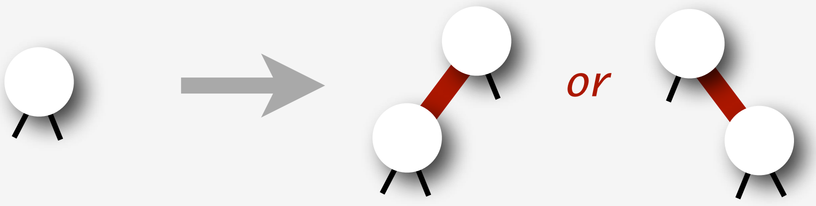
- left-rotate any right-leaning link on search path
- right-rotate top link if two reds in a row found
- trivial with recursion (do it after recursive calls)
- no corrections needed elsewhere

Insert code for LLRB trees

is based on four simple operations.

1. Insert a new node at the bottom.

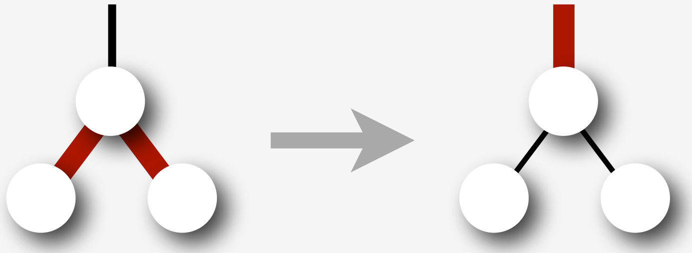
```
if (h == null)
      return new Node(key, value, RED);
```



could be right or left

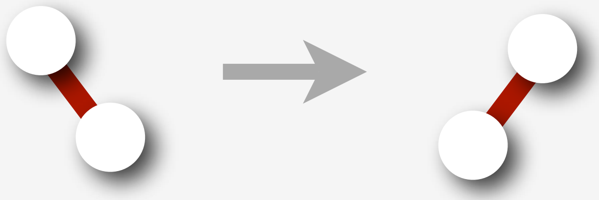
2. Split a 4-node.

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



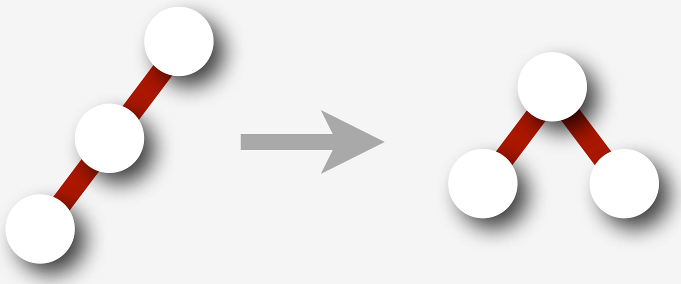
3. Enforce left-leaning condition.

```
if (isRed(h.right))
      h = rotateLeft(h);
```



4. Balance a 4-node.

```
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

```
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

A surprise

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

```
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;
}
```

A surprise

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

```
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;
}
```

A surprise

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

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

```
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

```
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;
}
```

very tricky

```
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;
}
```

Left-Leaning Red-Black Trees
Robert Sedgewick
Princeton University

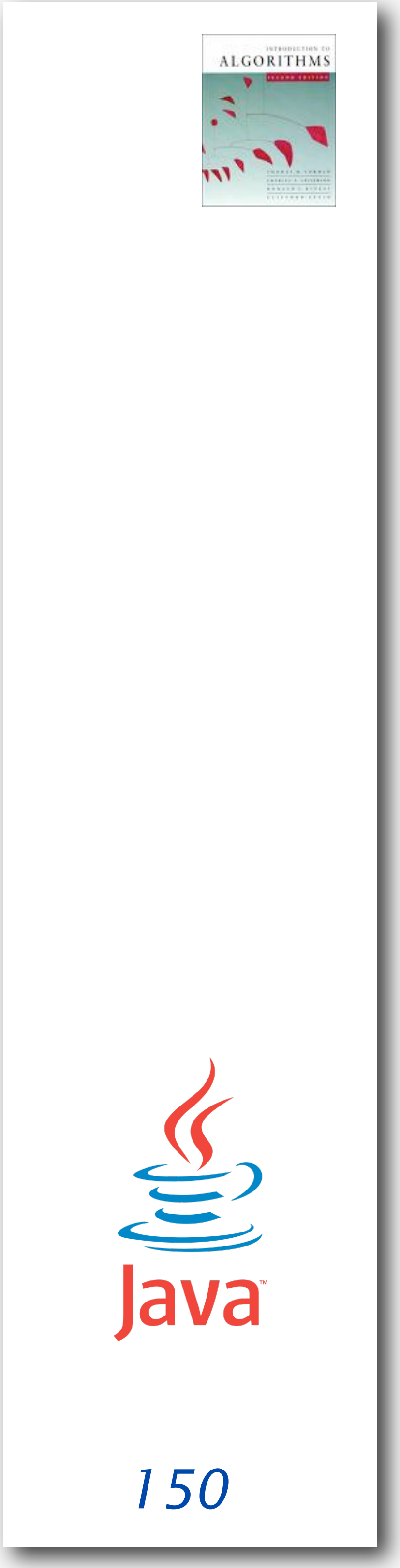
straightforward

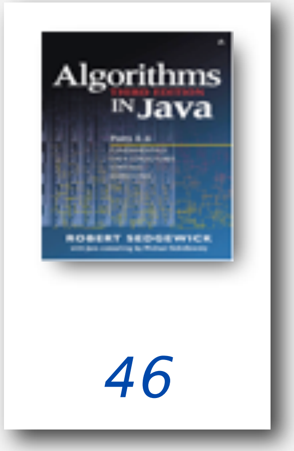
Why revisit red-black trees?

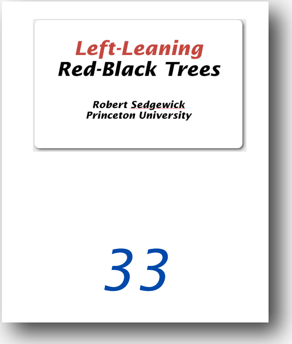
Take your pick:

TreeMap.java

Adapted from CLR by experienced professional programmers (2004)

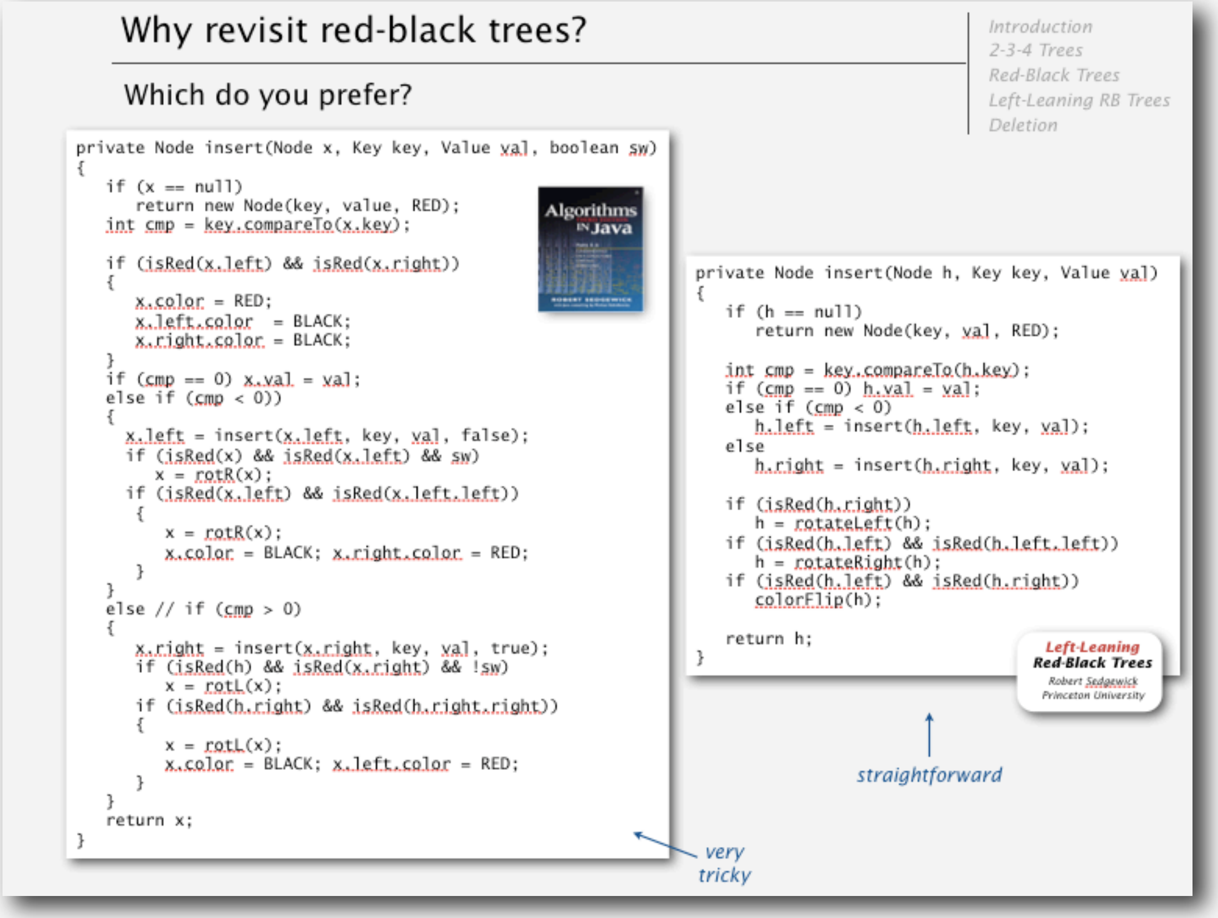






lines of code for insert (lower is better!)

wrong scale!



Why revisit red-black trees?

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

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;
}
```

very tricky

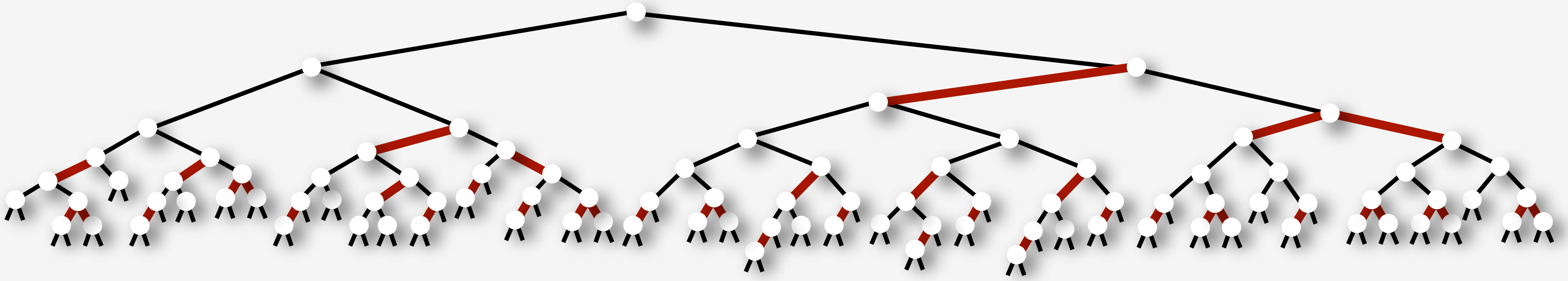
```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);

   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;
}
```

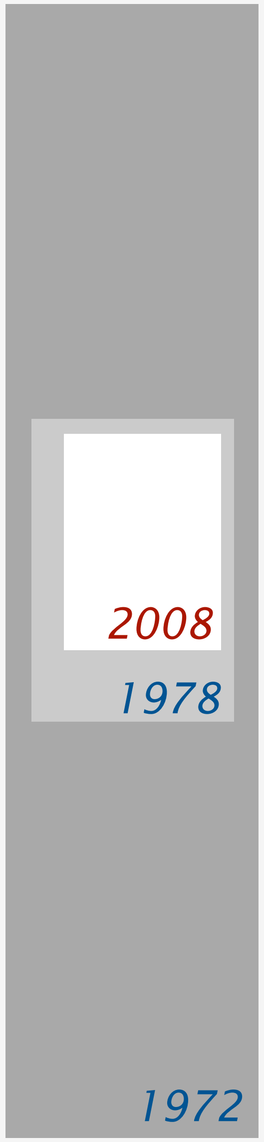
straightforward

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

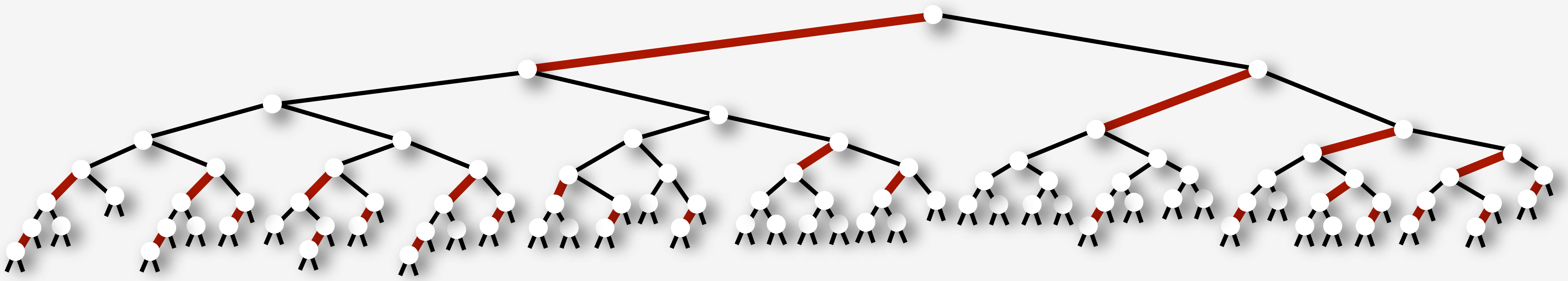
Same ideas simplify implementation of other operations

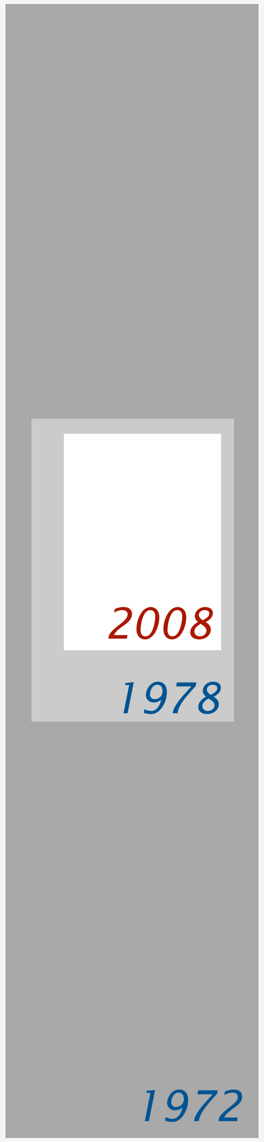
- delete min, max
- arbitrary delete

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

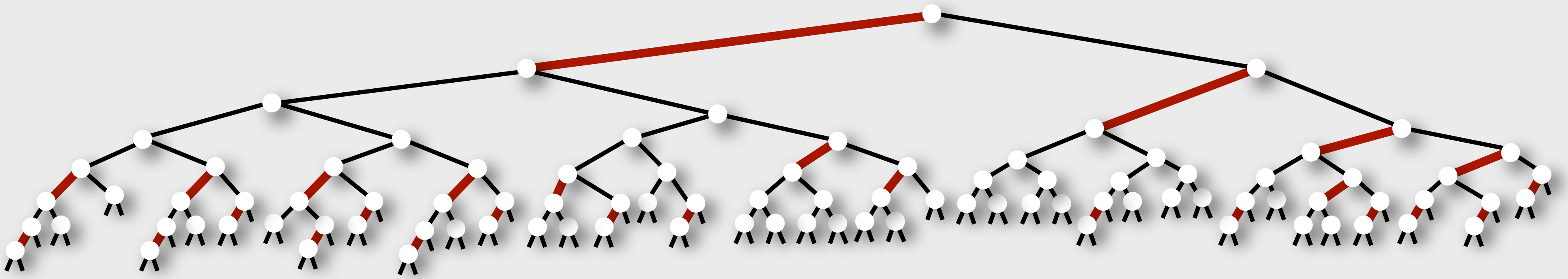
Introduction

2-3-4 Trees

LLRB Trees

Deletion

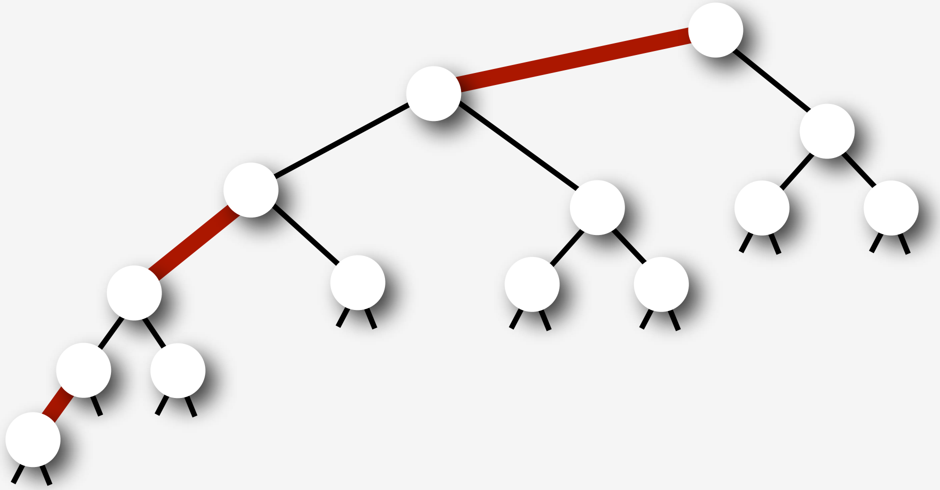
Analysis



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.

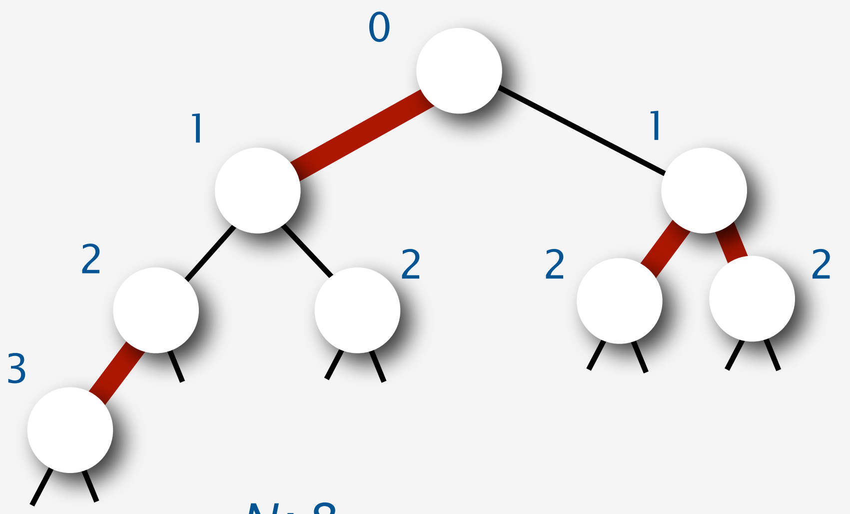
One remaining question

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

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)?

Analytic Combinatorics

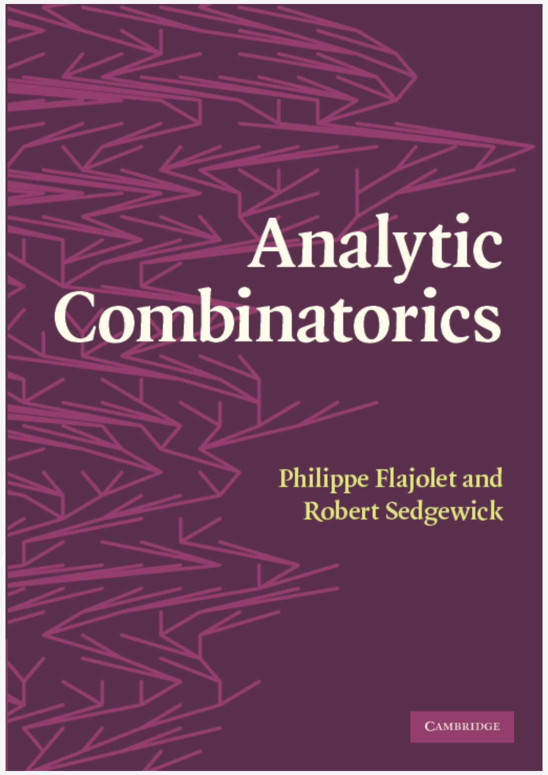
is a modern basis for studying discrete structures

Developed by

Philippe Flajolet and many coauthors

based on

classical combinatorics and analysis



Coming in 2008, now available on the web

Generating functions (GFs) encapsulate sequences

Symbolic methods treat GFs as formal objects

- formal definition of combinatorial constructions
- direct association with generating functions

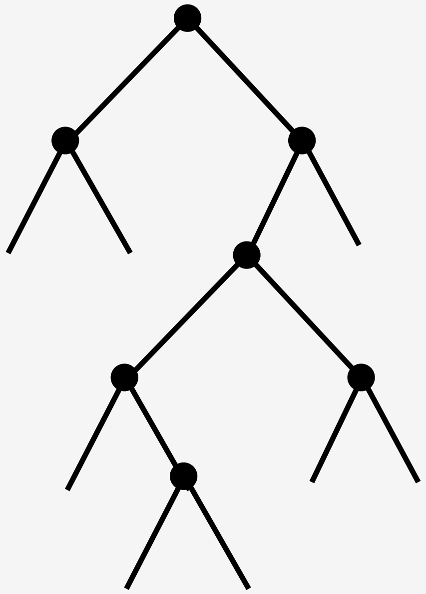
Complex asymptotics treat GFs as functions in the complex plane

- Study them with singularity analysis and other techniques
- Accurately approximate original sequence

Analysis of algorithms: classic example

A binary tree is a node connected to two binary trees.

How many binary trees with N nodes?



Given a recurrence relation

$$B_N = B_0 \ B_{N-1} + \ldots + B_k \ B_{N-1-k} + \ldots + B_{N-1} \ B_0$$

introduce a generating function

$$B(z) \equiv B_0 z^0 + B_1 z^1 + B_2 z^2 + B_3 z^3 + \ldots$$

multiply both sides by z^N and sum to get an equation

$$B(z) = 1 + z \ B(z)^2$$

that we can solve algebraically

$$B(z) = \frac{1 - \sqrt{1 - 4z}}{2z}$$

Quadratic equation

and expand to get coefficients

$$B_N = \frac{1}{N+1}\binom{2N}{N}$$

Binomial theorem

that we can approximate

$$B_N \sim \frac{4^N}{N\sqrt{\pi N}}$$

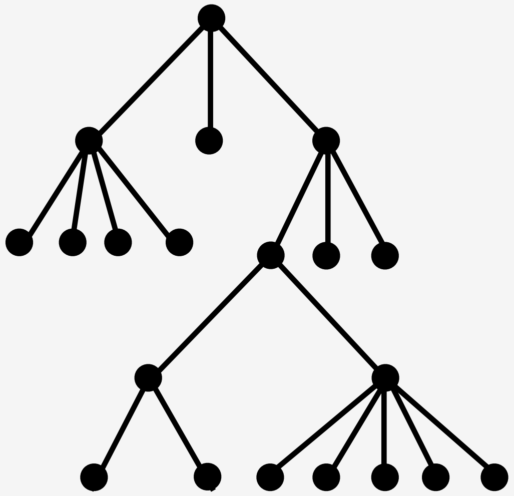
Stirling's approximation

Basic challenge: need a new derivation for each problem

Analytic combinatorics: classic example

A tree is a node connected to a sequence of trees

How many trees with N nodes?



Combinatorial constructions

$$\langle G\rangle = \varepsilon + \langle G\rangle + \langle G\rangle\times\langle G\rangle + \langle G\rangle\times\langle G\rangle\times\langle G\rangle + \ldots$$

directly map to GFs

$$G(z) = 1 + G(z) + G(z)^2 + G(z)^3 + \ldots$$

that we can manipulate algebraically

$$G(z) = \frac{1 - \sqrt{1 - 4z}}{2}$$

by quadratic equation

since $G(z) = \dfrac{1}{1 - G(z)}$,

so $G(z)^2 - G(z) + z = 0$

and treat as a complex function to approximate growth

$$G_N \sim \frac{4^N}{2N\,\Gamma(½)\sqrt{N}} = \frac{4^N}{2N\sqrt{\pi N}}$$

First principle: location of singularity determines exponential growth

Second principle: nature of singularity determines subexponential factor

NOTE: exact formula not needed!

Analytic combinatorics: singularity analysis

is a key to extracting coefficient asymptotics

Exponential growth factor

- depends on location of dominant singularity
- is easily extracted

Ex: $[z^N](1 - bz)^c = b^N [z^N](1 - z)^c$

Polynomial growth factor

- depends on nature of dominant singularity
- can often be computed via contour integration

Ex:

$$[z^N](1 - z)^c = \frac{1}{2\pi i}\int_C \frac{(1 - z)^c}{z^{N+1}}dz$$

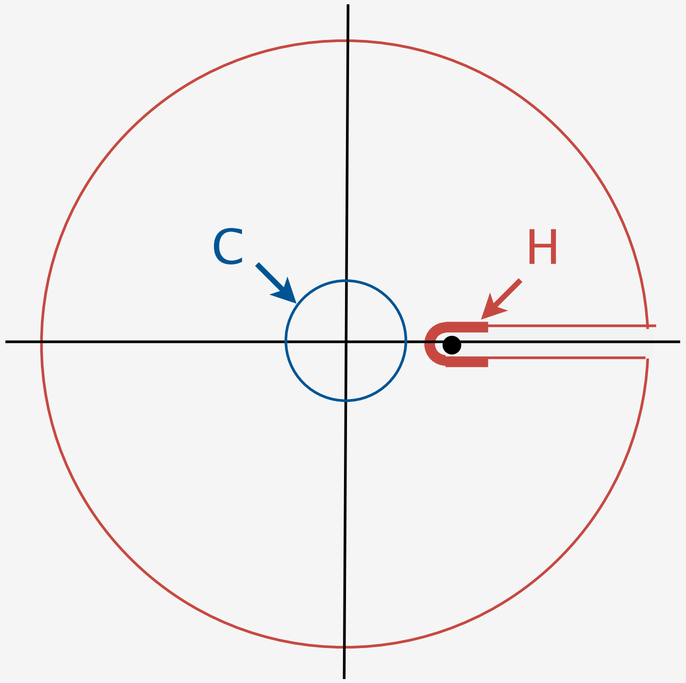
Cauchy coefficient formula

$$\sim \frac{1}{2\pi i}\int_H \frac{(1 - z)^c dz}{z^{N+1}}$$

Hankel contour

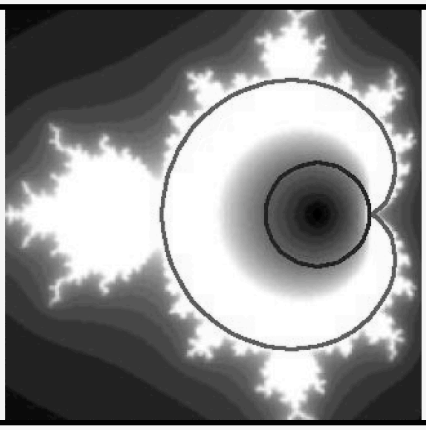
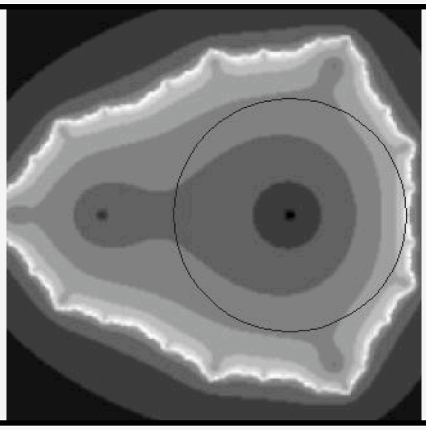
$$\sim \frac{1}{\Gamma(c)N^{c+1}}$$

many details omitted!



Warmup: tree enumeration

is classic analytic combinatorics

	binary trees	2-3 trees (Odlyzko, 1982)
combinatorial construction	$<B> = \square + <B> \times <B>$	$E<\square> = \square + E<\square \mapsto (\square\square + \square\square\square)>$
generating function	$B(z) = z + B(z)^2$	$E(z) = z + E(z^2 + z^3)$
domain of analyticity		
radius of convergence	$1/4$	$1/\varphi$
asymptotic growth	$B_N \bowtie 4^N$	$E_N \bowtie \varphi^N$
asymptotic approximation	$B_N \sim 4^N / N\sqrt{\pi N}$	$E_N \sim \varphi^N p(\log N) / N$

periodic function

Exercises in tree enumeration

Left-leaning 2-3 trees

2-3-4 trees

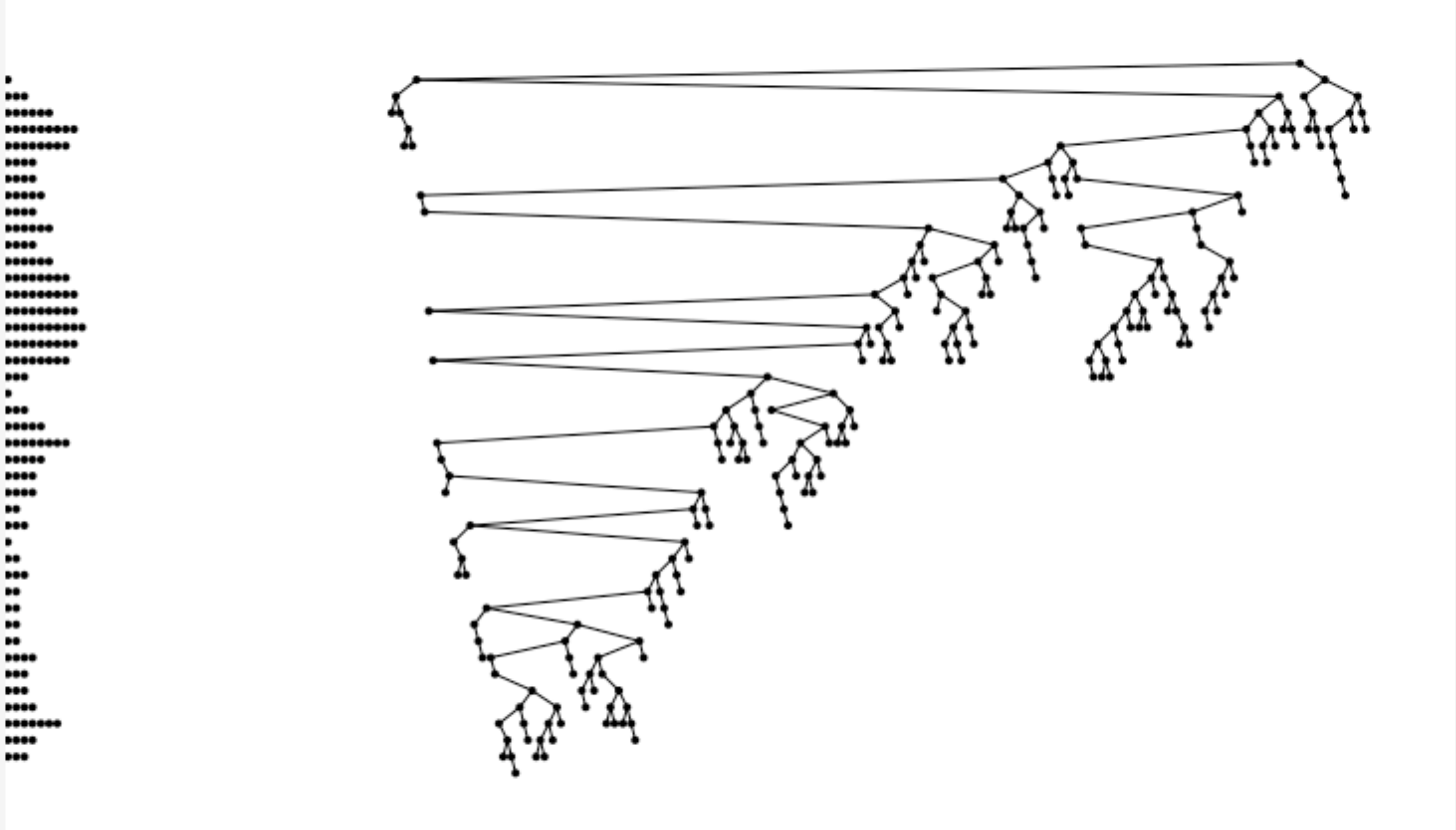
top-down 2-3-4 trees

path length in 2-3 trees (all trees equally likely)

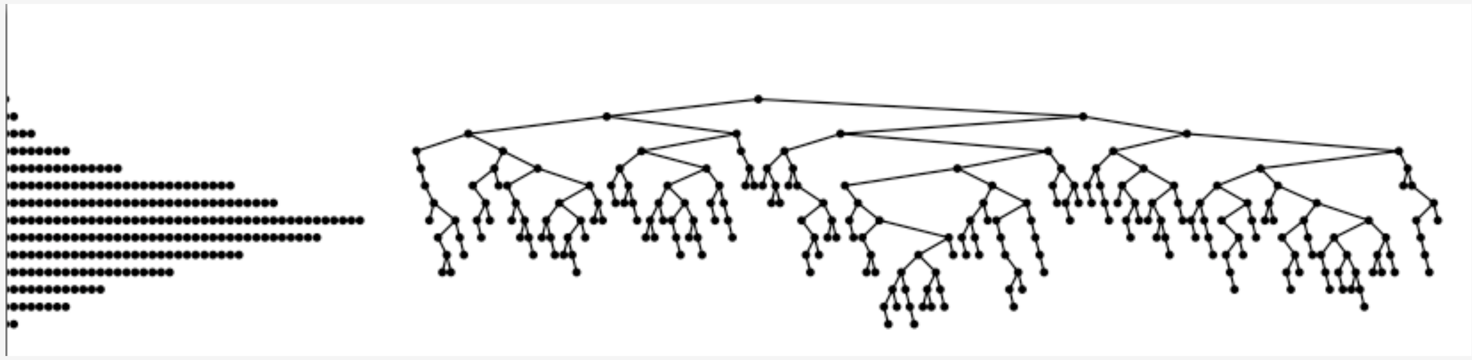
Path length in search trees

is a property of permutations, not trees

random binary tree



random binary search tree



Confronting this fact is the essential challenge in the analysis

Average-case analysis of balanced search trees

is a longstanding open problem

Main questions:

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

Average-case analysis of balanced search trees

is a longstanding open problem

Main questions:

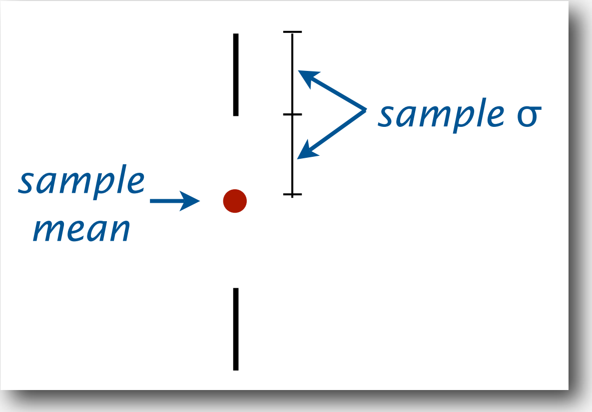
> Is average path length in tree built from random keys ~ c lg N ?
> If so, is c = 1 ?

Experimental evidence

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

- N = 100, 200, . . . , 50,000
- 100 trees each size

Tufte plot



Average-case analysis of balanced search trees

is a longstanding open problem

Main questions:

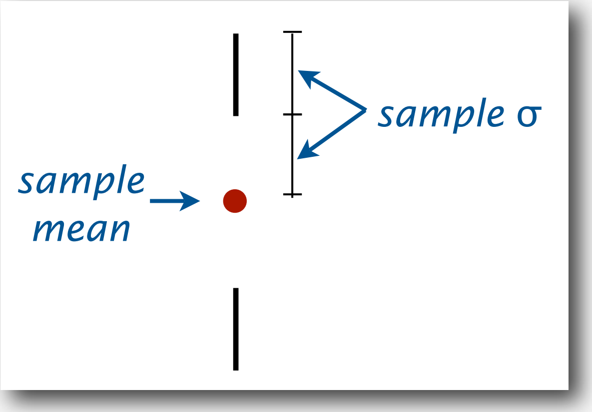
> Is average path length in tree built from random keys ~ c lg N ?
> If so, is c = 1 ?

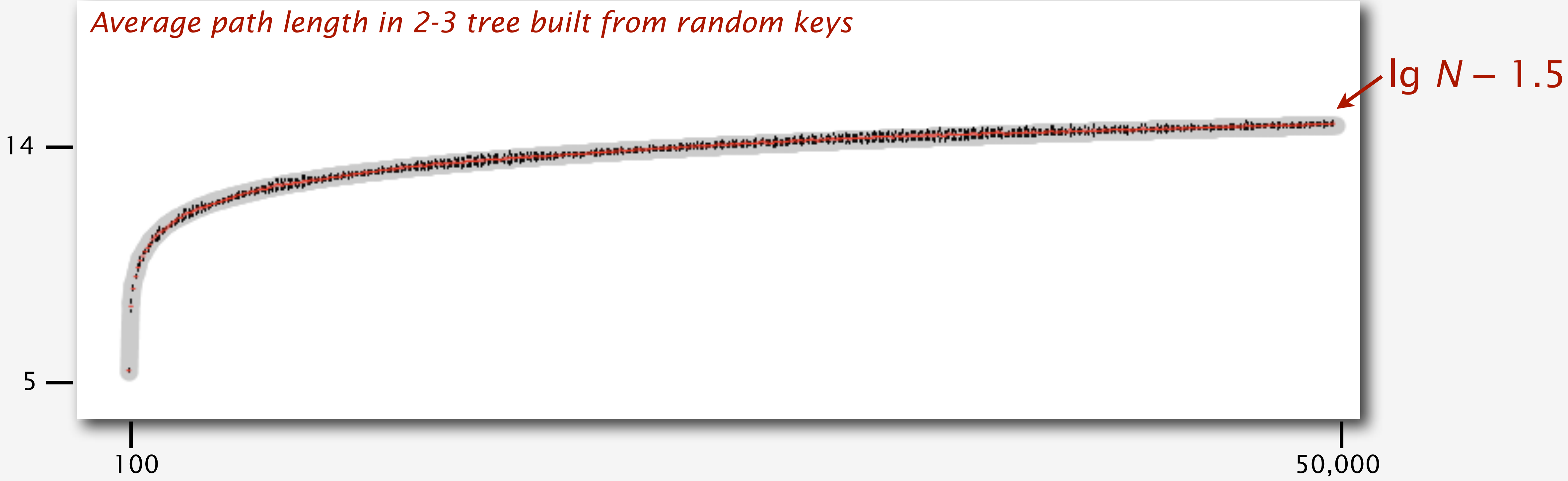
Experimental evidence strongly suggests YES!

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

- N = 100, 200, . . . , 50,000
- 100 trees each size

Tufte plot

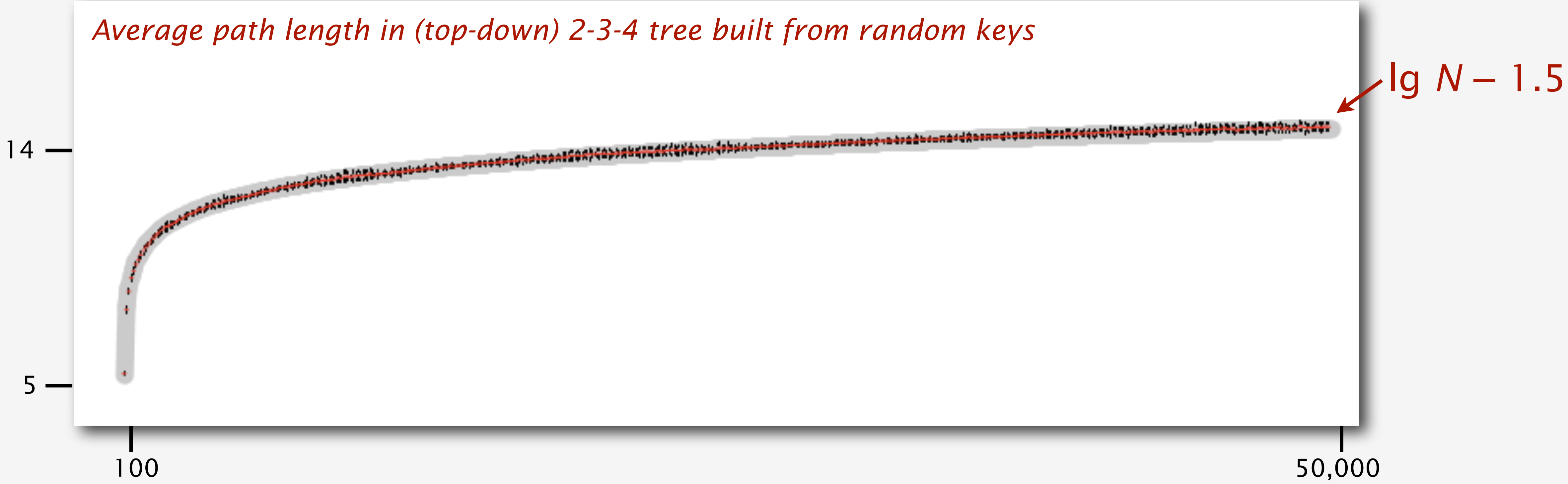




Experimental evidence

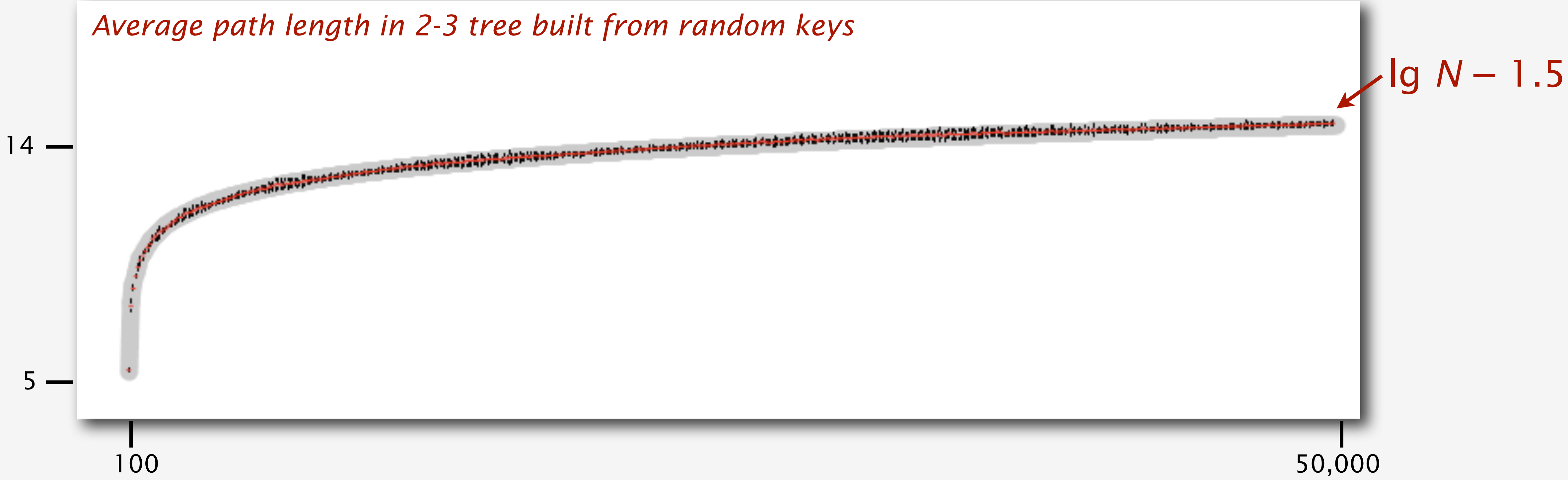
can suggest and confirm hypotheses

Ex: Does one of the algorithms lead to significantly faster search?



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

lg $N - 1.5$



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

lg $N - 1.5$

Hypothesis: No.

Average-case analysis of balanced search trees

is a longstanding open problem

Main questions:

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

Some known facts:

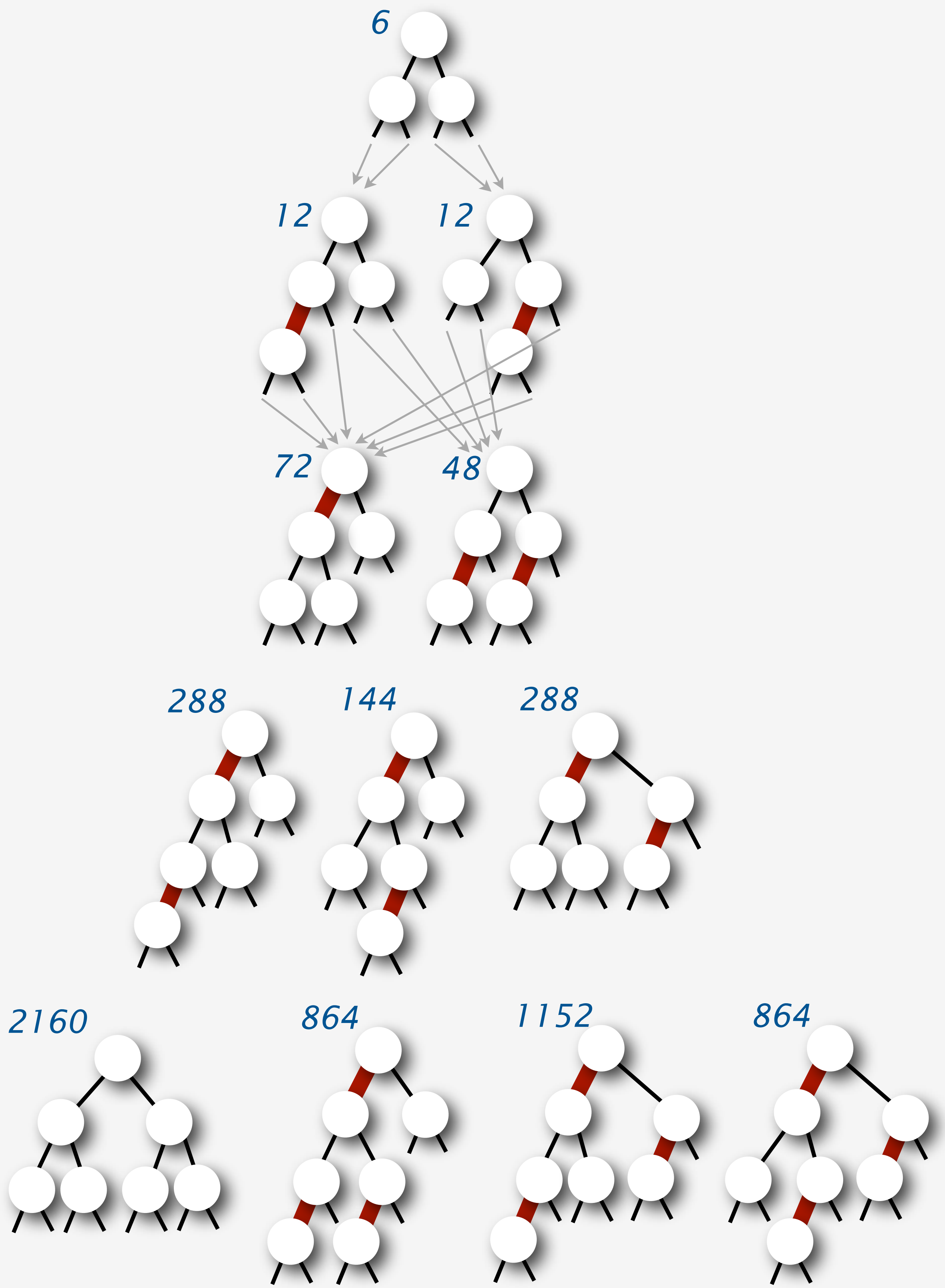
- worst case gives easy 2 lg N upper bound
- fringe analysis of gives upper bound of c_k lgN with $c_k > 1$
- analytic combinatorics gives path length in random trees

Are simpler implementations simpler to analyze?

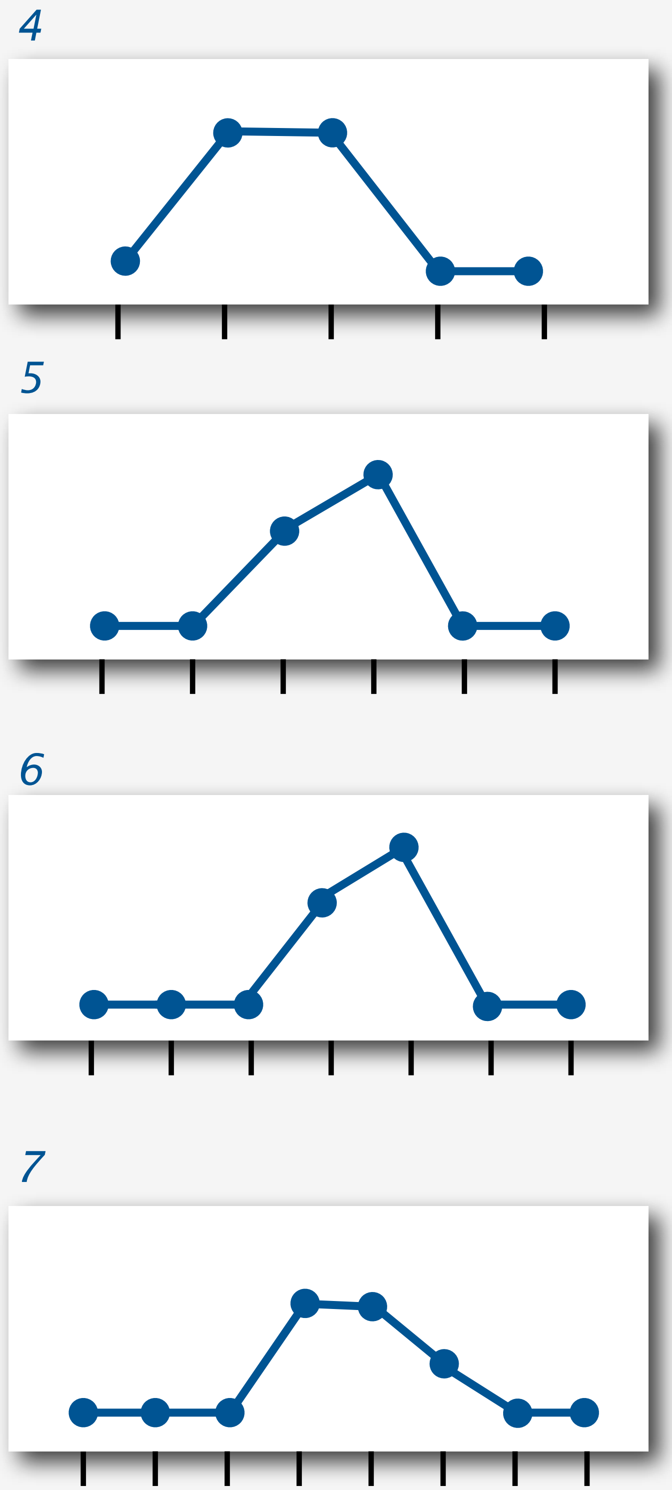
Is the better experimental evidence that is now available helpful?

A starting point: study balance at the root (left subtree size)

Left subtree size in left-leaning 2-3 trees

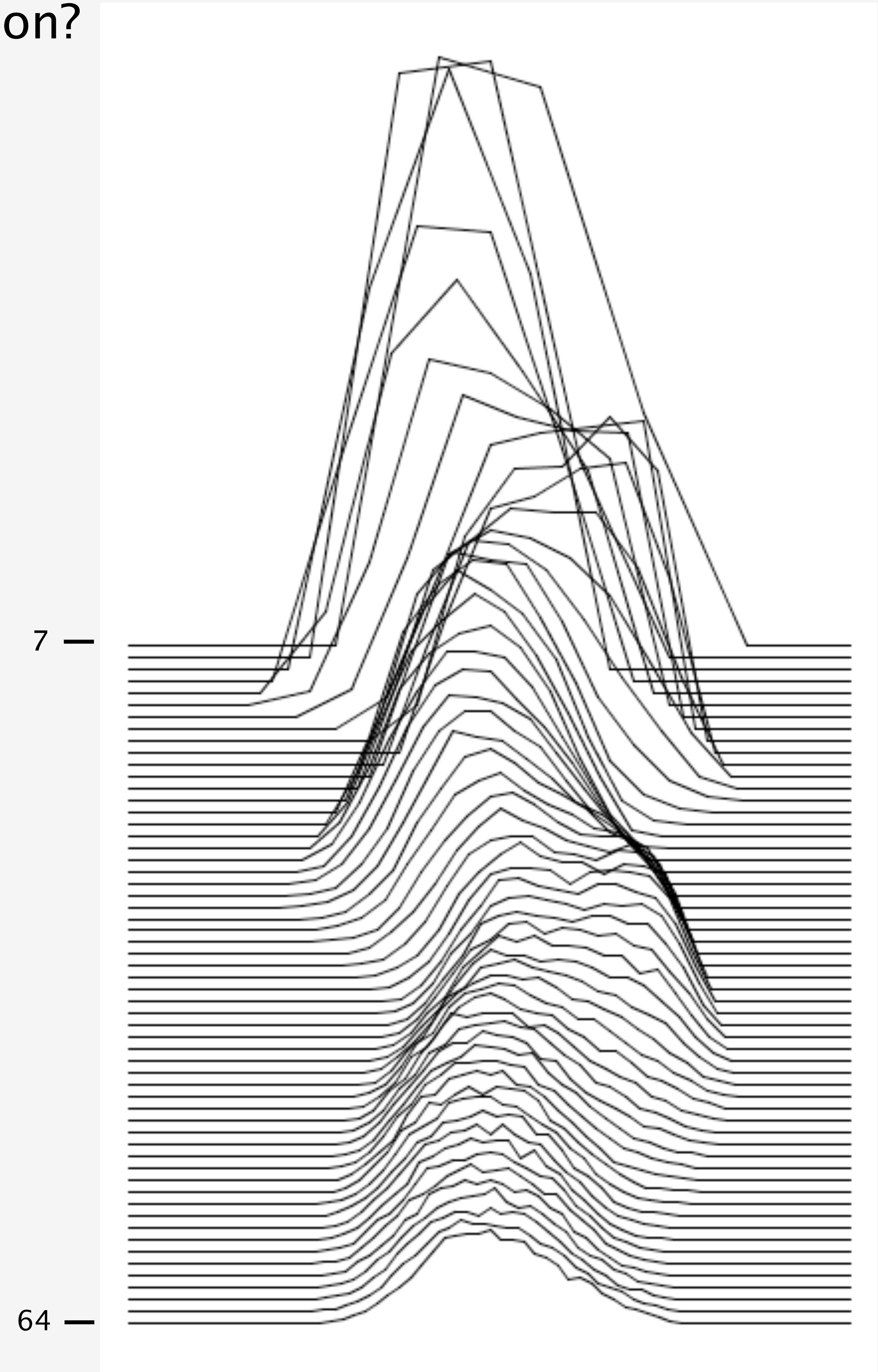


Exact distributions

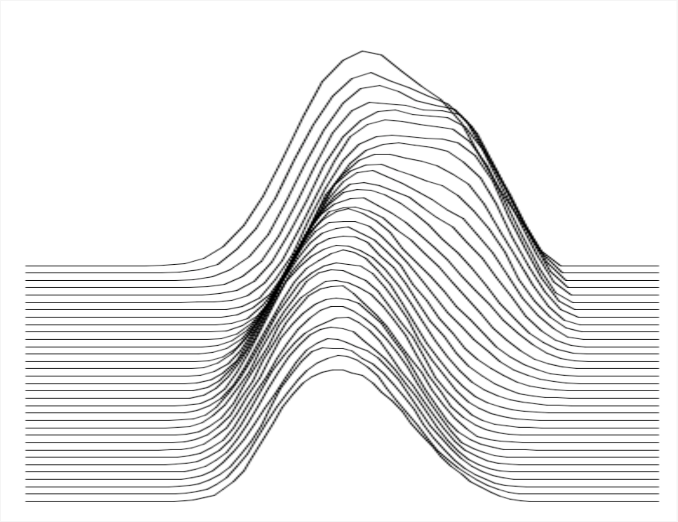


Left subtree size in left-leaning 2-3 trees

Limiting distribution?

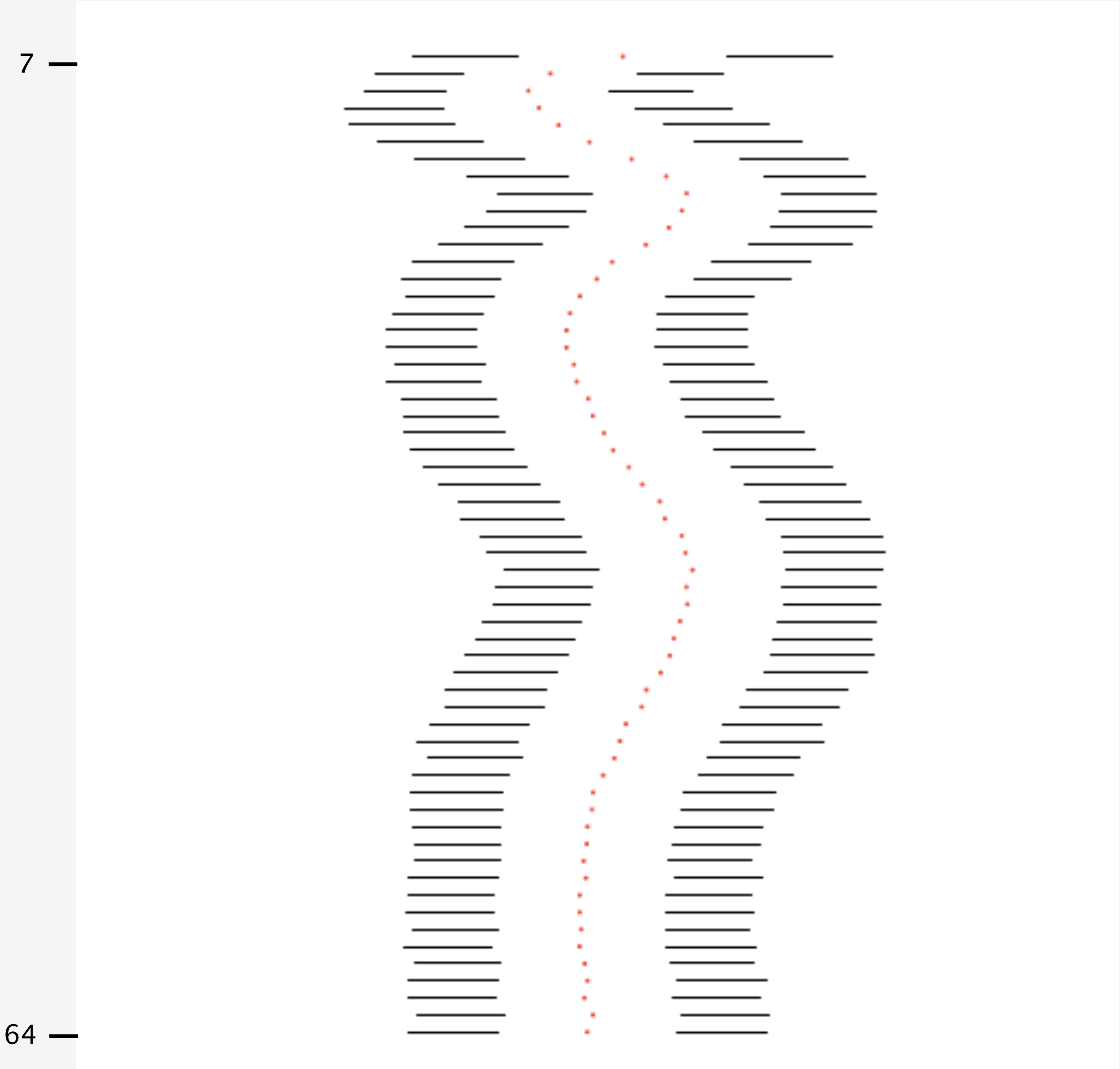


smoothed version (32-64)



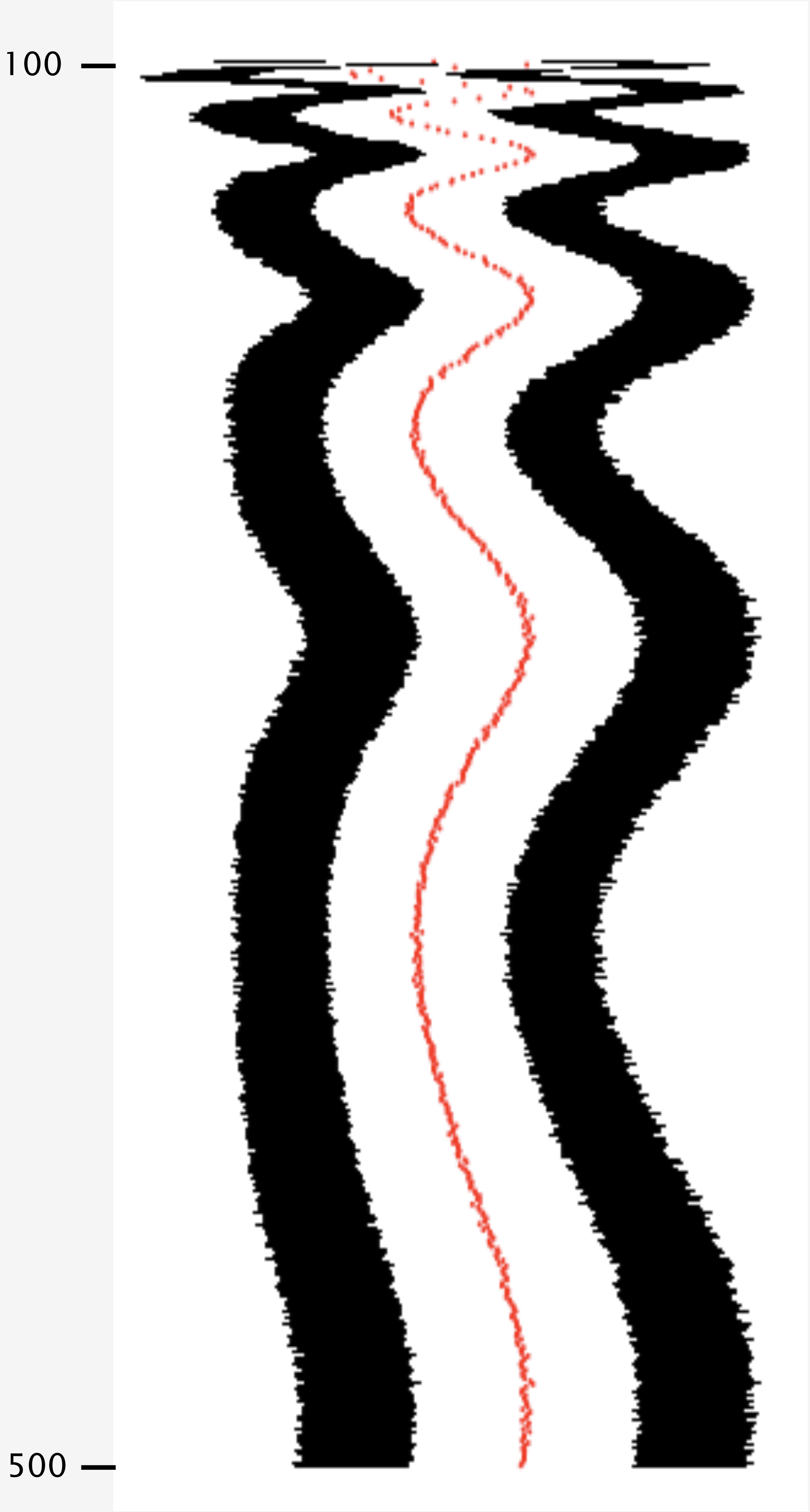
Left subtree size in left-leaning 2-3 trees

Tufte plot



Left subtree size in left-leaning 2-3 trees

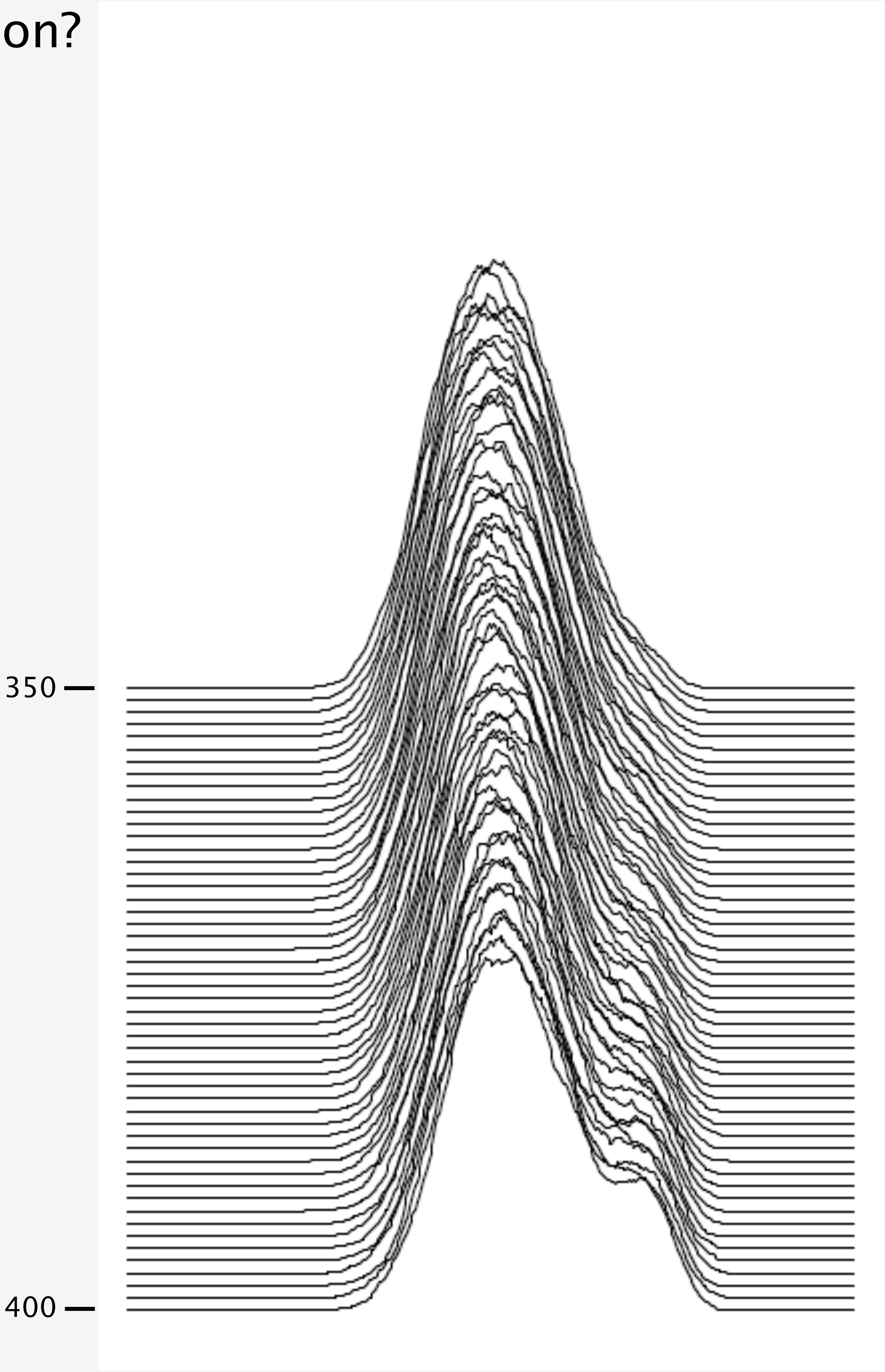
Tufte plot



view of highway for bus driver who has had one Caipirinha too many ?

Left subtree size in left-leaning 2-3 trees

Limiting distribution?

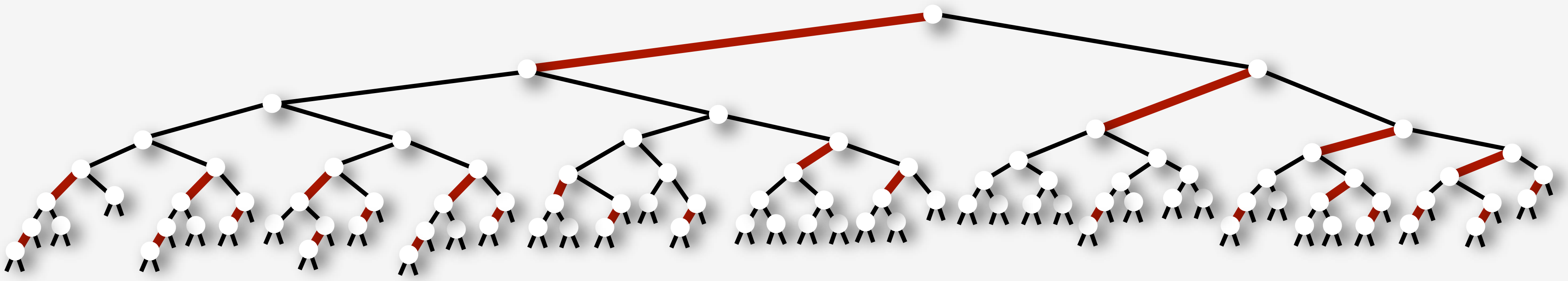


10,000 trees for each size

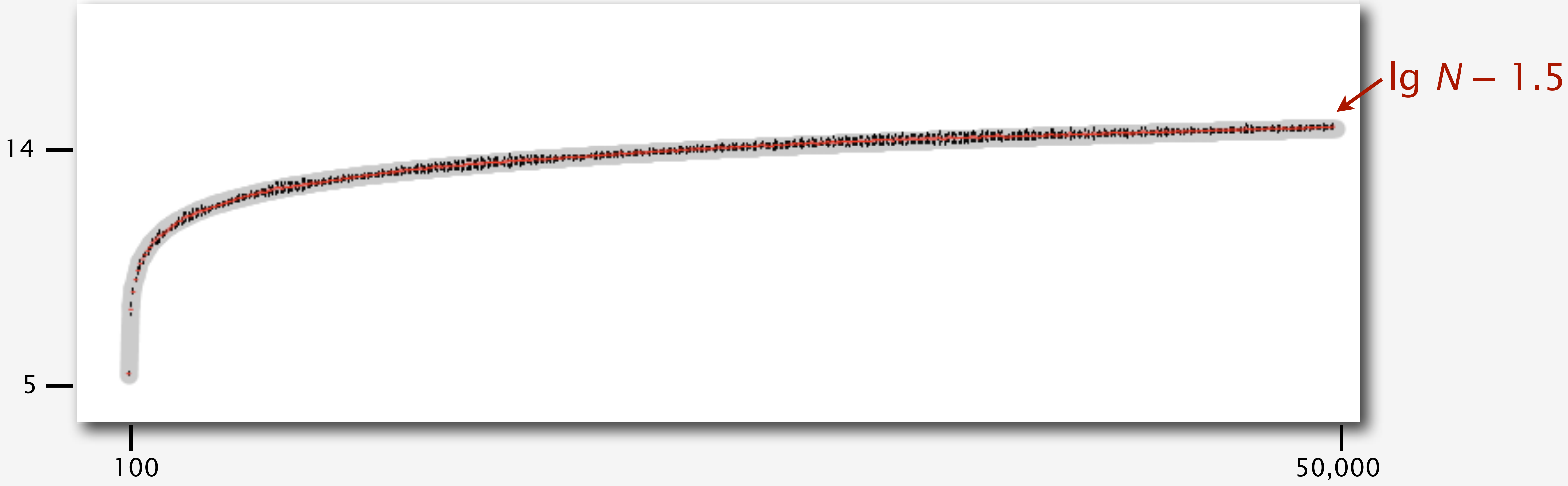
smooth factor 10

An exercise in the analysis of algorithms

Find a proof !



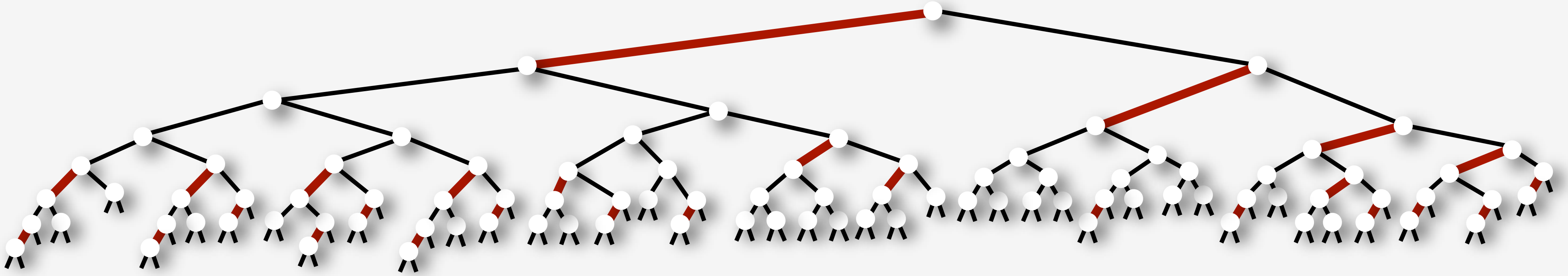
Average path length in 2-3 tree built from random keys



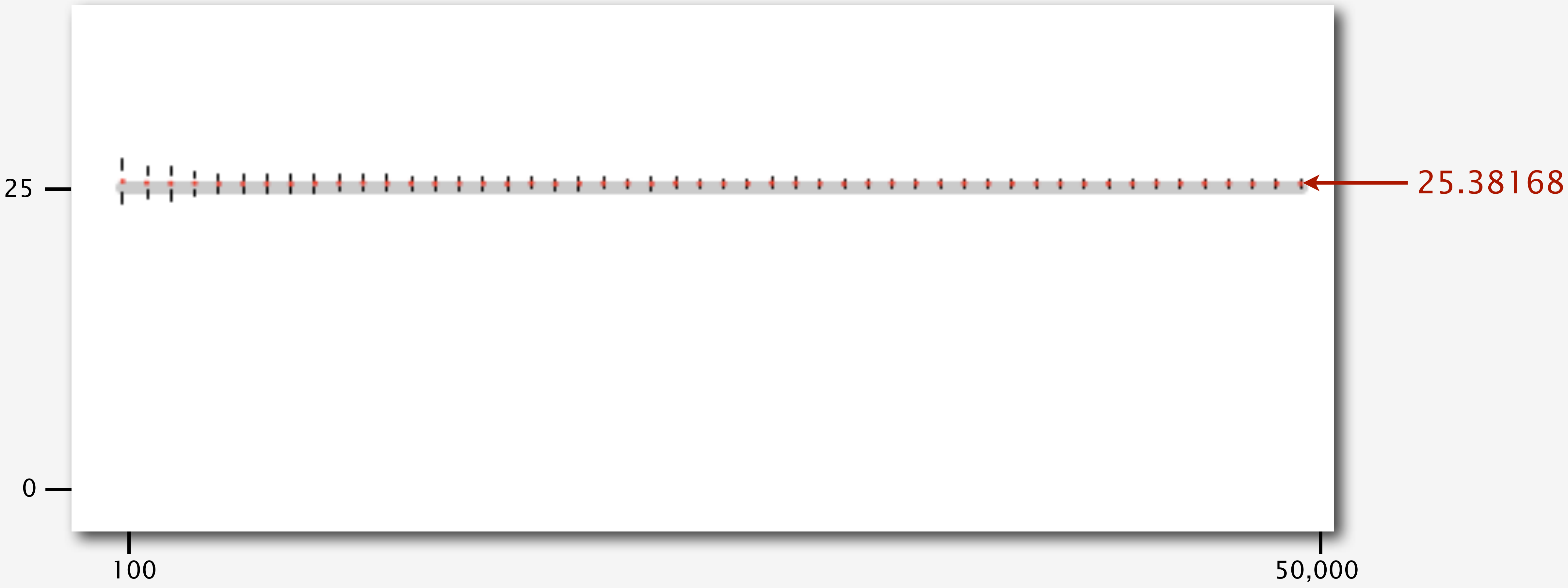
Addendum: Observations

Observation 1

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

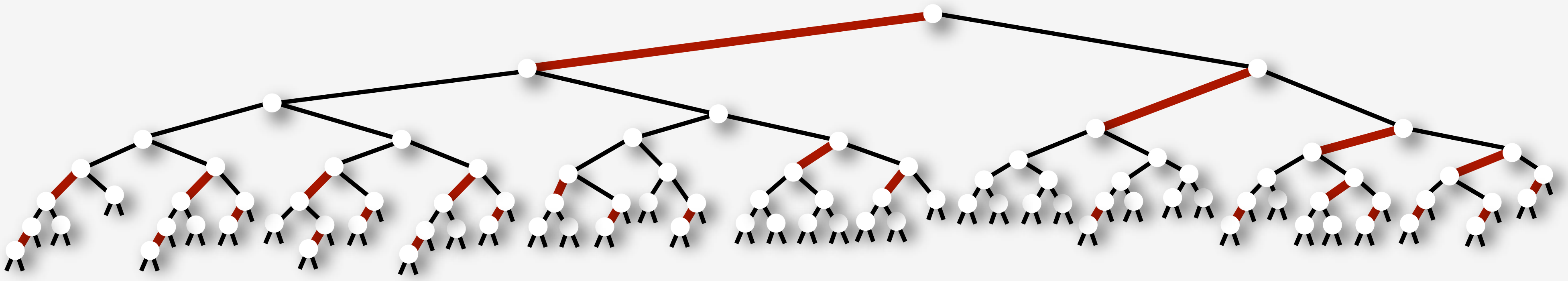


Percentage of red nodes in 2-3 tree built from random keys

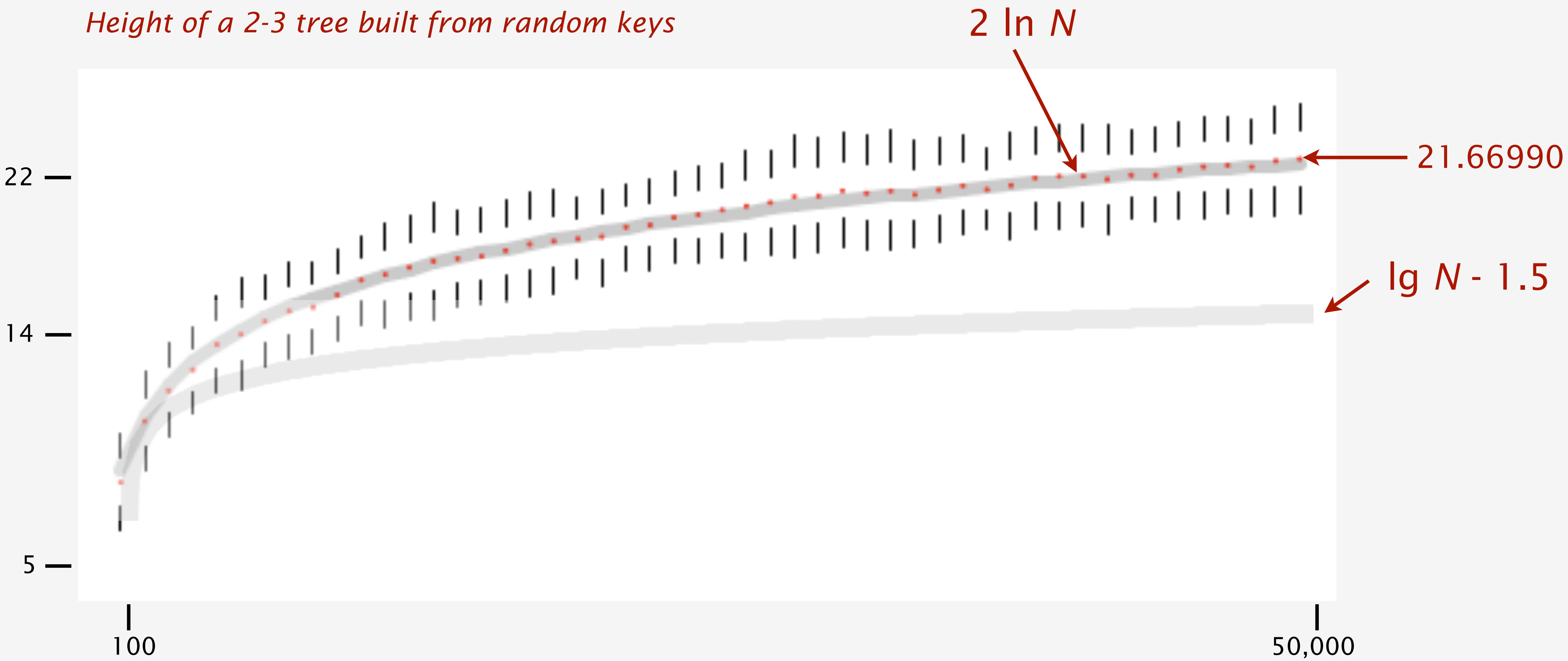


Observation 2

The **height** of a 2-3 tree is ~2 ln N (!!!)



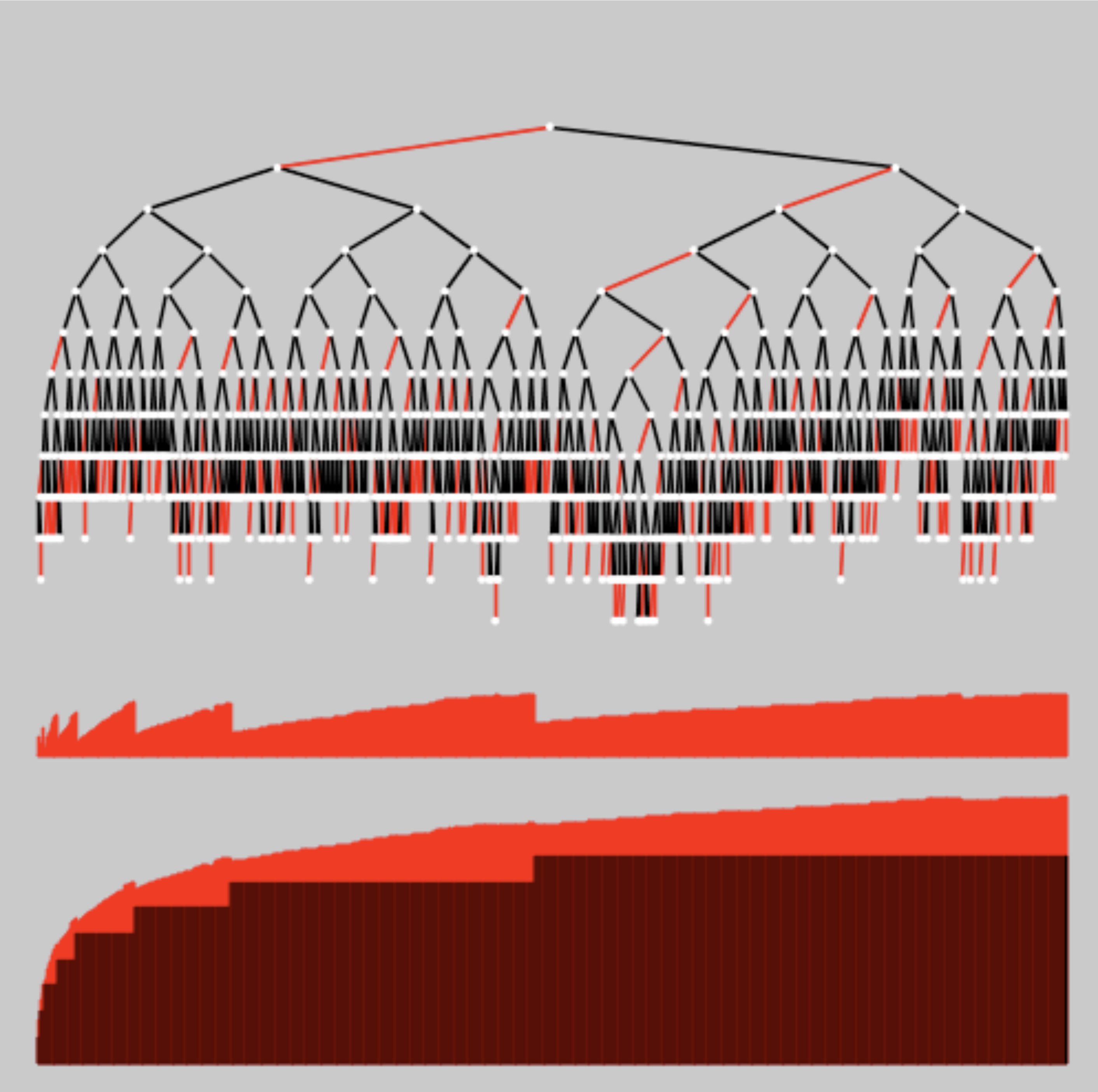
Height of a 2-3 tree built from random keys



Very surprising because the average path length in an elementary BST is also ~2 ln *N ≈ 1.386* lg *N*

Observation 3

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



Observation 4

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