Balanced Trees

- 2-3-4 trees
- red-black trees
- B-trees

References:
Algorithms in Java, Chapter 13
http://www.cs.princeton.edu/introalgsds/44balanced

Symbol Table Review

Symbol table: key-value pair abstraction.

- Insert a value with specified key.
- Search for value given key.
- Delete value with given key.

Randomized BST.

- Guarantee of ~c lg N time per operation (probabilistic).
- Need subtree count in each node.
- Need random numbers for each insert/delete op.

This lecture. 2-3-4 trees, left-leaning red-black trees, B-trees.

new for Fall 2007!

Summary of symbol-table implementations

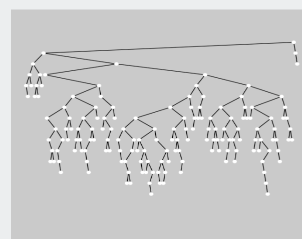
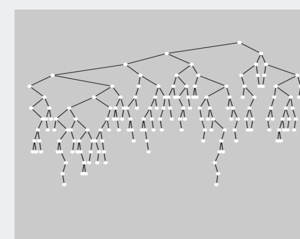
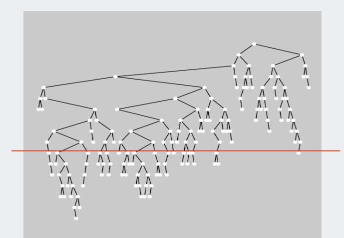
implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.39 lg N	1.39 lg N	1.39 lg N	yes

Randomized BSTs provide the desired guarantees

probabilistic, with exponentially small chance of quadratic time

This lecture: Can we do better?

Typical random BSTs

N = 250
lg N ≈ 8
1.39 lg N ≈ 11

average node depth

- 2-3-4 trees
- red-black trees
- B-trees

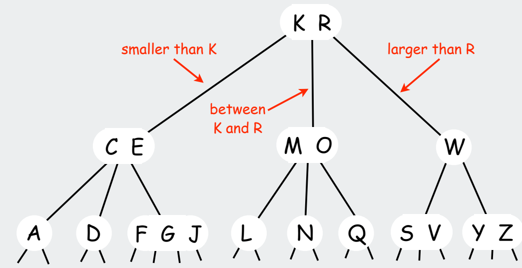
2-3-4 Tree

2-3-4 tree. Generalize node to allow multiple keys; keep tree balanced.

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.

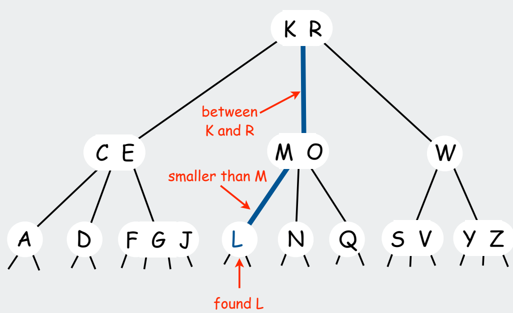


Searching in a 2-3-4 Tree

Search.

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

Ex. Search for L

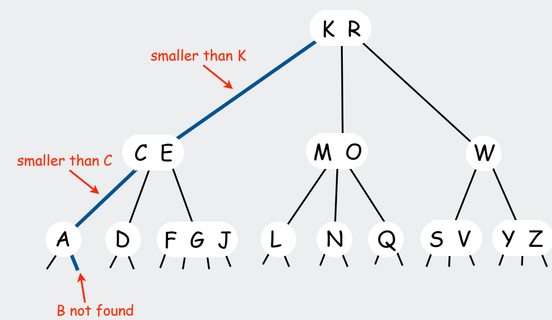


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.

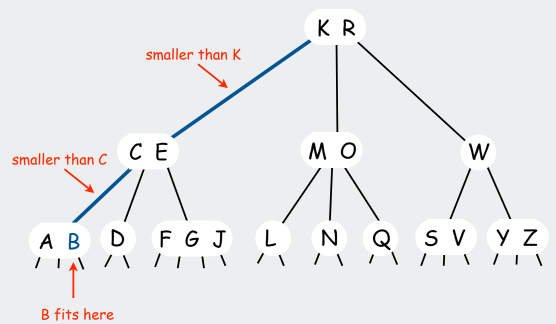
Ex. Insert B



Insertion in a 2-3-4 Tree

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

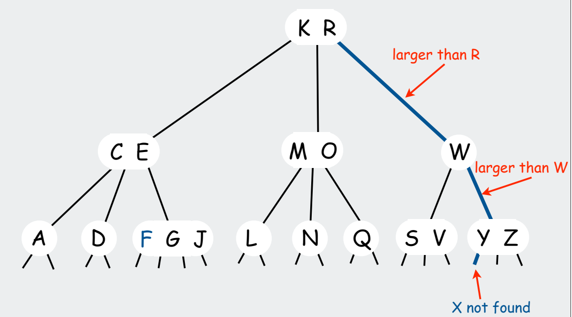
Ex. Insert B



Insertion in a 2-3-4 Tree

Insert.
- Search to bottom for key.

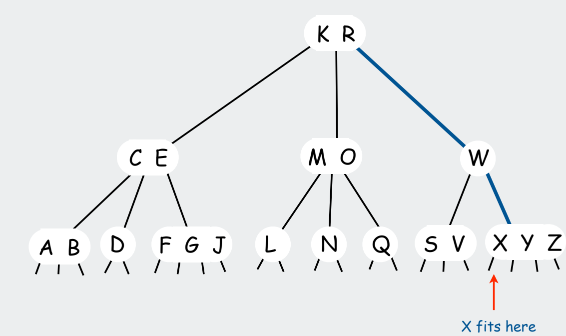
Ex. Insert X



Insertion in a 2-3-4 Tree

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

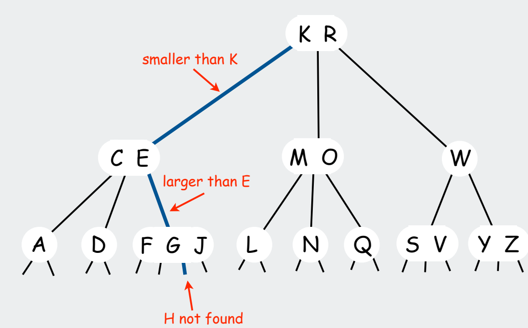
Ex. Insert X



Insertion in a 2-3-4 Tree

Insert.
- Search to bottom for key.

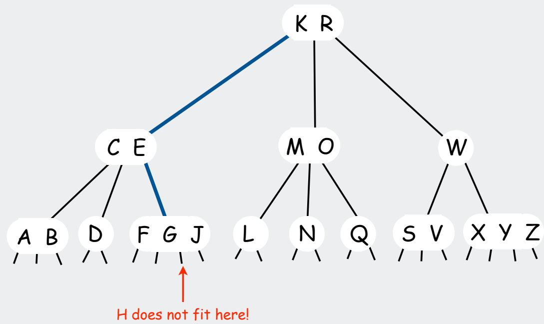
Ex. Insert H



Insertion in a 2-3-4 Tree

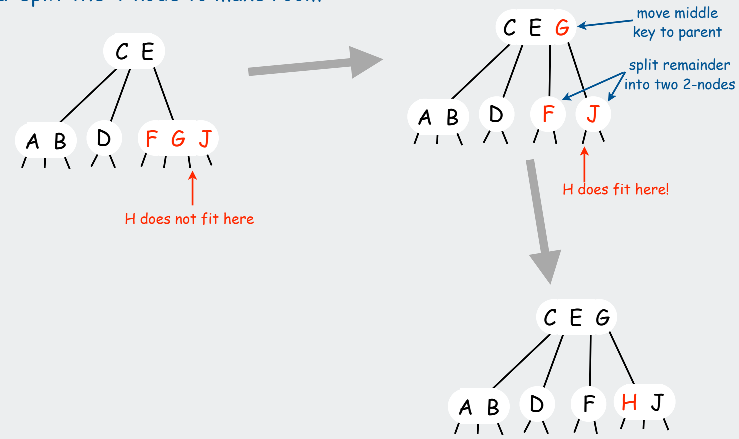
Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.
- 4-node at bottom: ??

Ex. Insert H



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

Idea: split the 4-node to make room



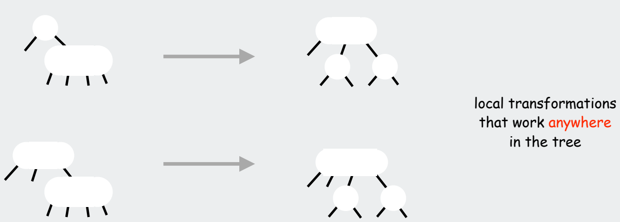
Problem: Doesn't work if parent is a 4-node

Solution 1: Split the parent (and continue splitting up while necessary).

Solution 2: Split 4-nodes on the way down.

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

Idea: split 4-nodes on the way down the tree.
- Ensures that most recently seen node is not a 4-node.
- Transformations to split 4-nodes:

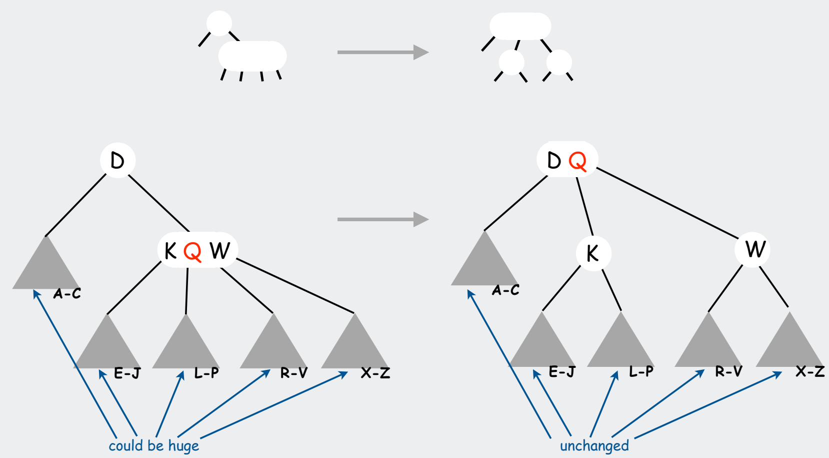


Invariant. Current node is not a 4-node.

Consequences
- 4-node below a 4-node case never happens
- insertion at bottom node is easy since it's not a 4-node.

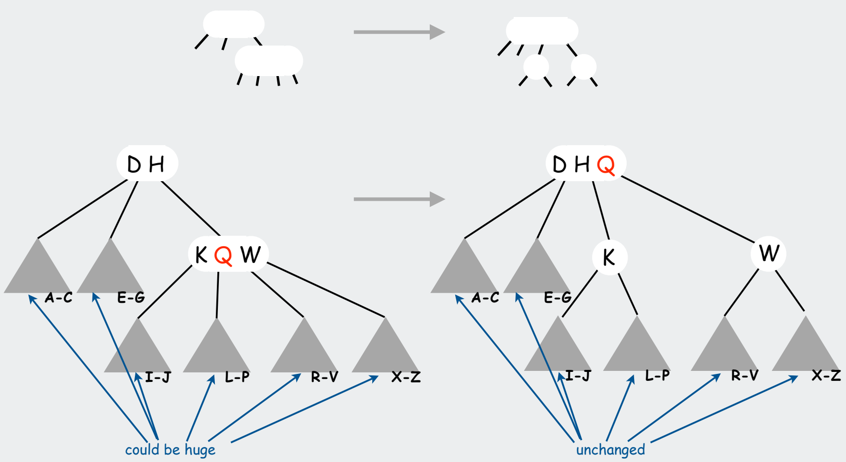
Splitting a 4-node below a 2-node in a 2-3-4 tree

A local transformation that works anywhere in the tree



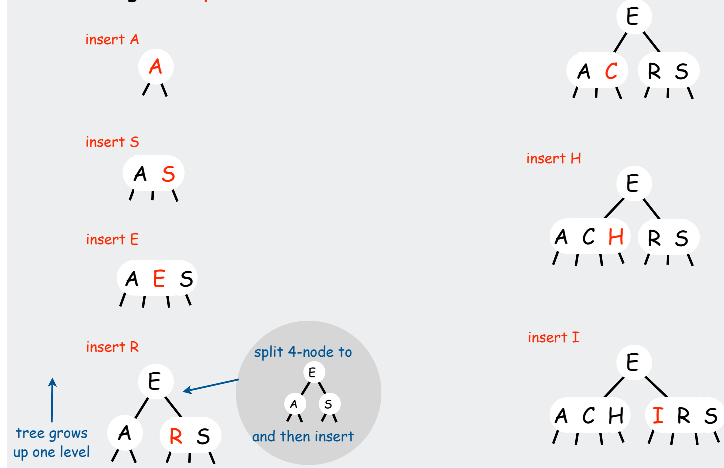
Splitting a 4-node below a 3-node in a 2-3-4 tree

A local transformation that works anywhere in the tree



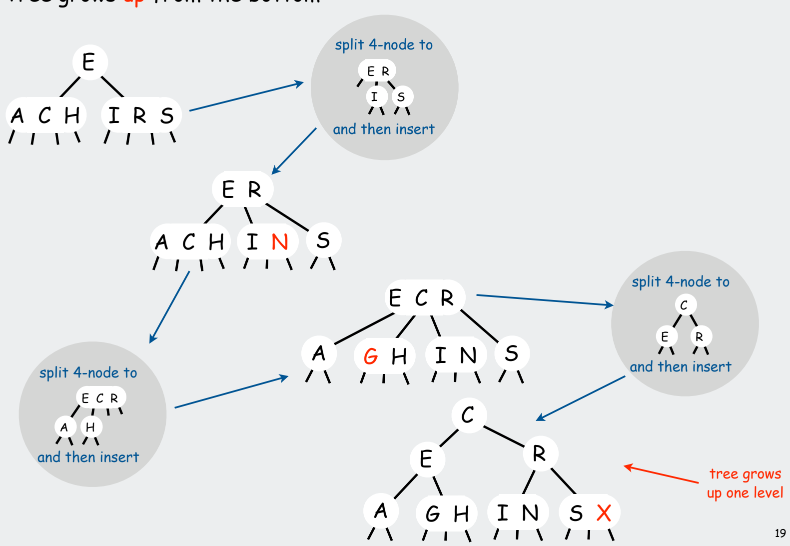
Growth of a 2-3-4 tree

Tree grows up from the bottom



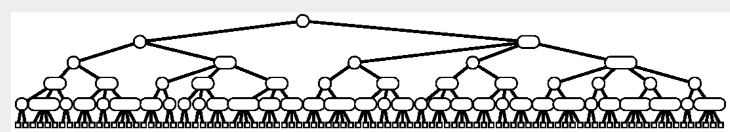
Growth of a 2-3-4 tree (continued)

Tree grows up from the bottom



Balance in 2-3-4 trees

Key property: All paths from root to leaf have same length.



Tree height.
- Worst case: lg N [all 2-nodes]
- Best case: $\log_4 N = 1/2 \lg N$ [all 4-nodes]
- Between 10 and 20 for a million nodes.
- Between 15 and 30 for a billion nodes.

2-3-4 Tree: Implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Implementation of `getChild()` involves multiple compares.
- Large number of cases for `split()`, `make3Node()`, and `make4Node()`.

```
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);
}
```

fantasy code

Bottom line: could do it, but stay tuned for an easier way.

Summary of symbol-table implementations

implementation	guarantee search	guarantee insert	guarantee delete	average case search	average case insert	average case delete	ordered iteration?
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes

constants depend upon implementation

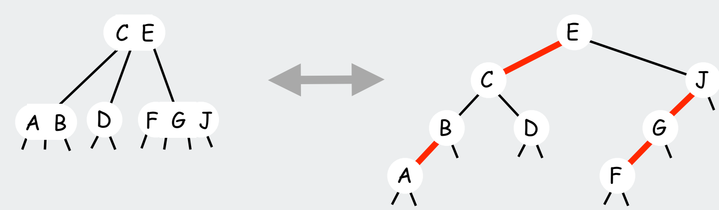
- 2-3-4 trees
- **red-black trees**
- B-trees

Left-leaning red-black trees (Guibas-Sedgewick, 1979 and Sedgewick, 2007)

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
- 1-1 correspondence between 2-3-4 and left-leaning red-black trees

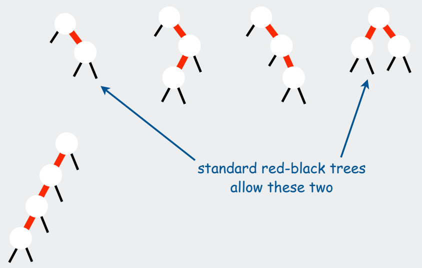


Left-leaning red-black trees

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

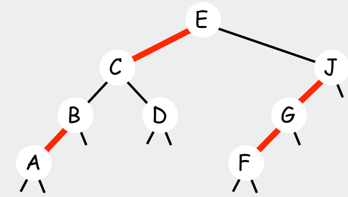
Disallowed:

- right-leaning red edges
- three red edges in a row



Search implementation for red-black trees

```
public Val 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;
}
```



Search code is the same as elementary BST (ignores the color)
[runs faster because of better balance in tree]

Note: iterator code is also the same.

Insert implementation for red-black trees (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
          implements Iterable<Key>
{
   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 void put(Key key, Value val)
   {
      root = put(root, key, val);
      root.color = BLACK;
   }
}
```

color of incoming link

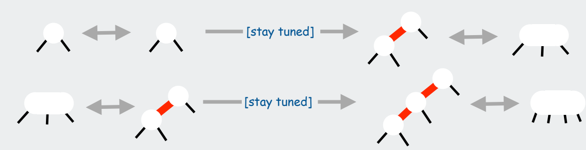
helper method to test node color

```
private boolean isRed(Node x)
{
   if (x == null) return false;
   return (x.color == RED);
}
```

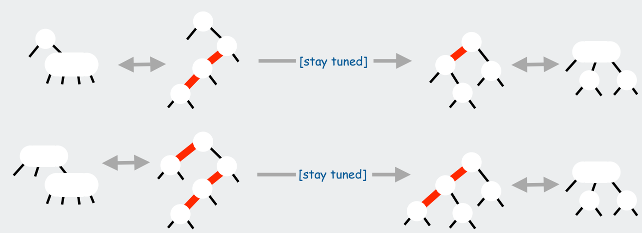

Insert implementation for left-leaning red-black trees (strategy)

Basic idea: maintain 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 a new red node at the bottom



3. Split 4-nodes on the way DOWN the tree.

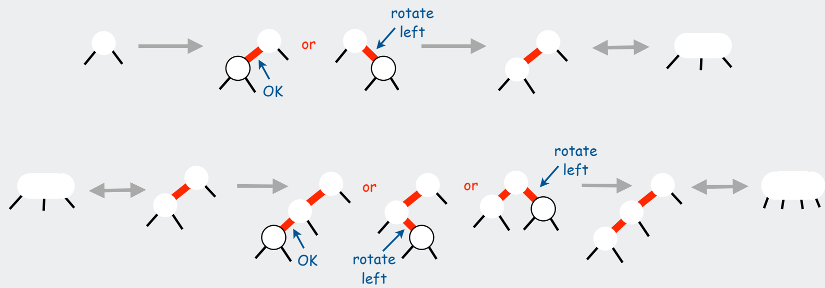


Inserting a new node at the bottom in a LLRB tree

Maintain 1-1 correspondence with 2-3-4 trees

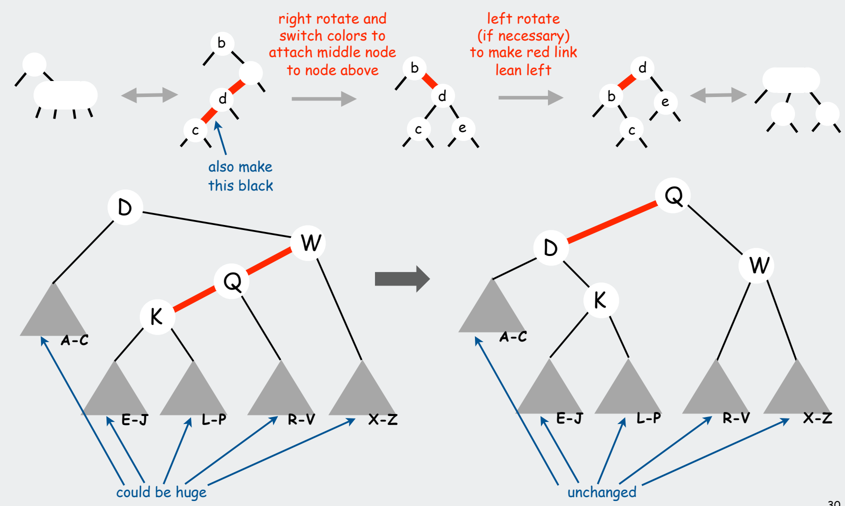
1. Add new node as usual, with red link to glue it to node above

2. Rotate left if necessary to make link lean left



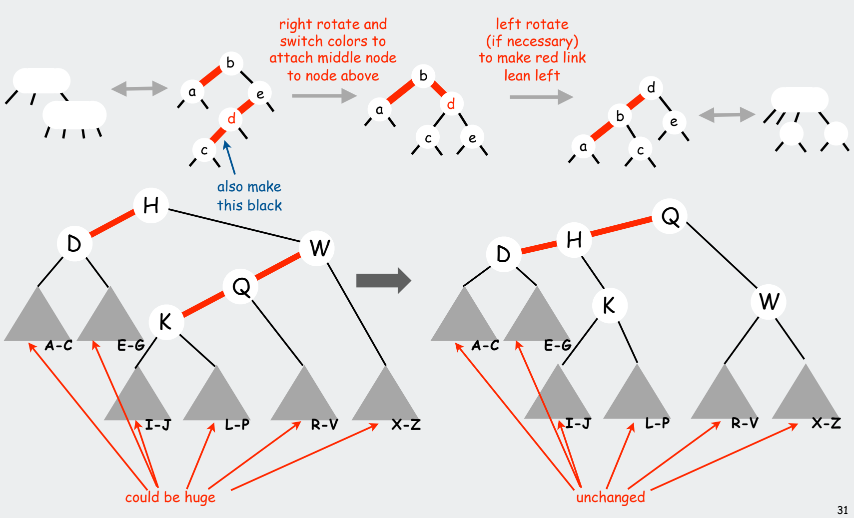
Splitting a 4-node below a 2-node in a left-leaning red-black tree

Maintain correspondence with 2-3-4 trees



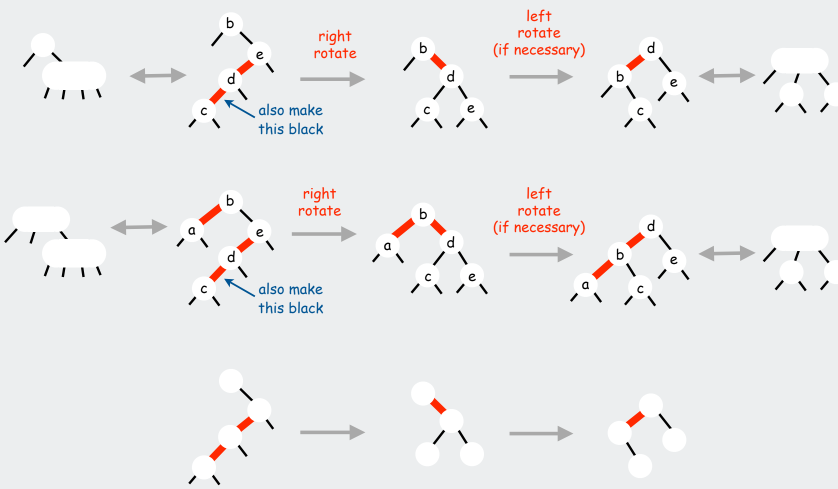
Splitting a 4-node below a 3-node in a left-leaning red-black tree

Maintain correspondence with 2-3-4 trees



Splitting 4-nodes a left-leaning red-black tree

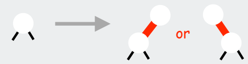
The two transformations are the same



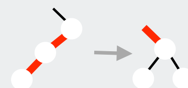
Insert implementation for left-leaning red-black trees (strategy revisited)

Basic idea: maintain 1-1 correspondence with 2-3-4 trees

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

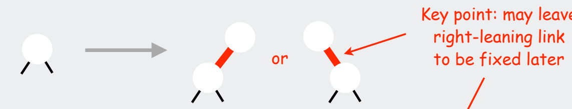


NEW TRICK: enforce left-leaning condition on the way UP the tree.
- left-rotate any right-leaning link on search path
- trivial with recursion (do it after recursive calls)
- no other right-leaning links elsewhere

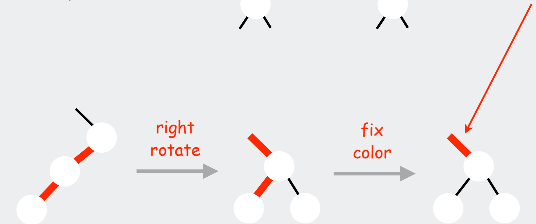
Note: nonrecursive top-down implementation possible, but requires keeping track of great-grandparent on search path (!) and lots of cases.

Insert implementation for left-leaning red-black trees (basic operations)

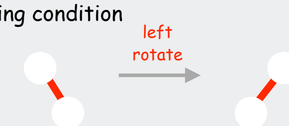
Insert a new node at bottom



Split a 4-node



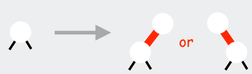
Enforce left-leaning condition



Insert implementation for left-leaning red-black trees (code for basic operations)

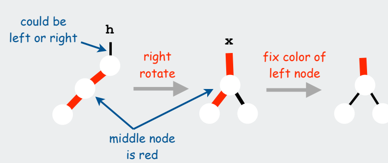
Insert a new node at bottom

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



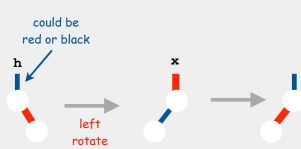
Split a 4-node

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



Enforce left-leaning condition

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



Insert implementation for left-leaning red-black trees (code)

```
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;
}
```

← insert new node at bottom

← split 4-nodes on the way down

← search

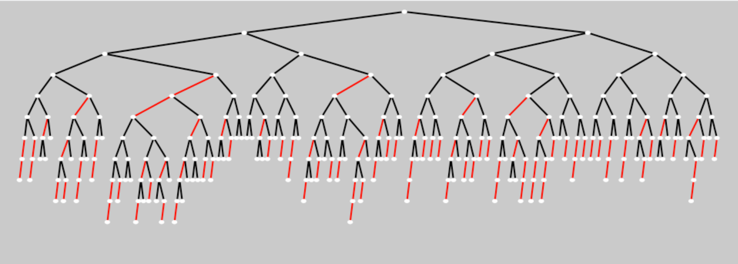
← enforce left-leaning condition on the way back up

Balance in left-leaning red-black trees

Proposition A. Every path from root to leaf has same number of black links.

Proposition B. Never three red links in-a-row.

Proposition C. Height of tree is less than 3 lg N + 2 in the worst case.



Property D. Height of tree is ~lg N in typical applications.

Property E. Nearly all 4-nodes are on the bottom in the typical applications.

Why left-leaning trees?

Take your pick:

old code (that students had to learn in the past)

```
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;
}
```

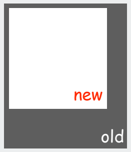
extremely tricky

new code (that you have to learn)

```
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) x.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;
}
```

straightforward
(if you've paid attention)

Why left-leaning trees?



Simplified code

- left-leaning restriction reduces number of cases
- recursion gives two (easy) chances to fix each node
- short inner loop

Same ideas simplify implementation of other operations

- delete min
- delete max
- delete

Built on the shoulders of many, many old balanced tree algorithms

- AVL trees
- 2-3 trees
- 2-3-4 trees
- skip lists

Bottom line: Left-leaning red-black trees are the simplest to implement

and at least as efficient

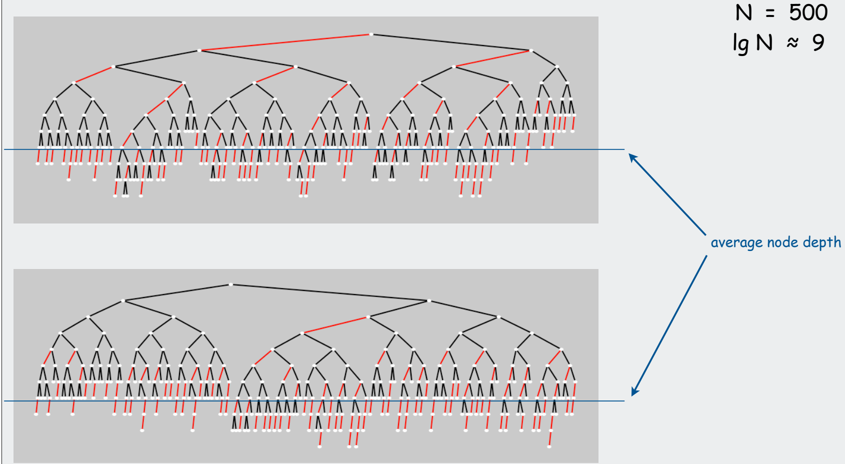
Summary of symbol-table implementations

implementation	guarantee search	guarantee insert	guarantee delete	average case search	average case insert	average case delete	ordered iteration?
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes

exact value of coefficient unknown
but extremely close to 1

Typical random left-leaning red-black trees

$N = 500$
$\lg N \approx 9$



- 2-3-4 trees
- red-black trees
- **B-trees**

B-trees (Bayer-McCreight, 1972)

B-Tree. Generalizes 2-3-4 trees by allowing up to M links per node.

Main application: file systems.

- Reading a page into memory from disk is expensive.
- Accessing info on a page in memory is free.
- Goal: minimize # page accesses.
- Node size M = page size.

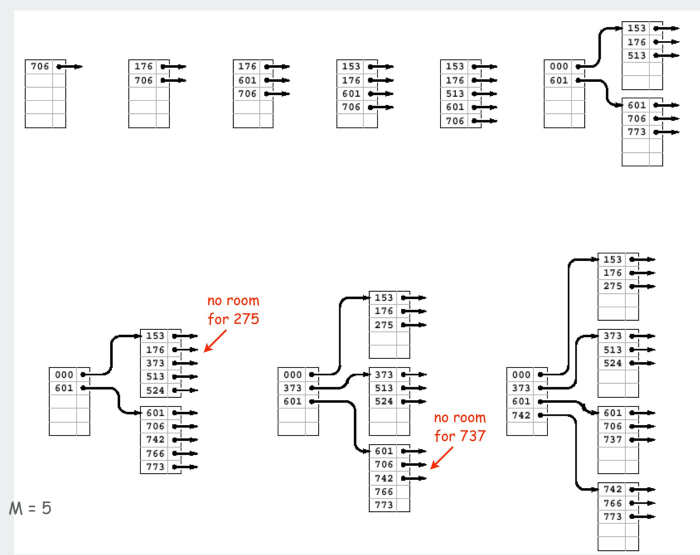
Space-time tradeoff.

- M large $\Rightarrow$ only a few levels in tree.
- M small $\Rightarrow$ less wasted space.
- Typical M = 1000, N < 1 trillion.

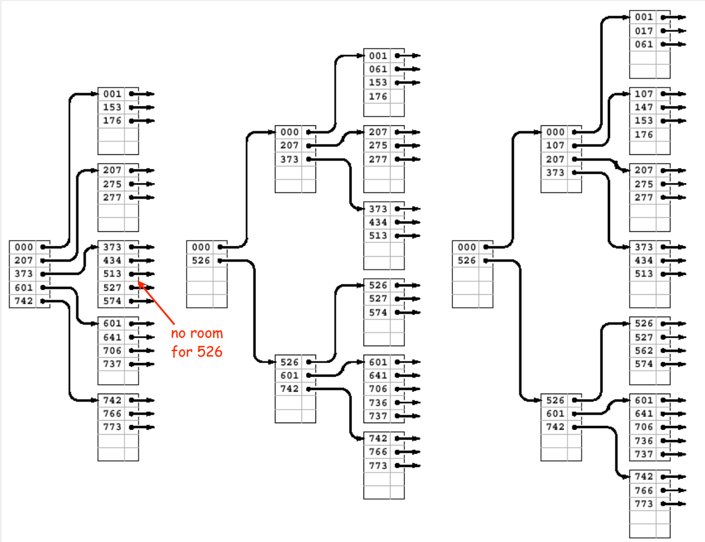
Bottom line. Number of page accesses is $\log_M N$ per op.

in practice: 3 or 4 (!)

B-Tree Example



B-Tree Example (cont)



Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.44 lg N	1.44 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.44 lg N	1.44 lg N	1.44 lg N	yes
2-3-4 tree	c lg N	c lg N		c lg N	c lg N		yes
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes
B-tree	1	1	1	1	1	1	yes

B-Tree. Number of page accesses is $\log_M N$ per op.

Balanced trees in the wild

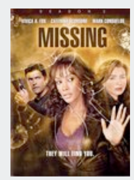
Red-black trees: widely used as system symbol tables
- Java: **java.util.TreeMap, java.util.TreeSet.**
- C++ STL: map, multimap, multiset.
- Linux kernel: **linux/rbtree.h.**

B-Trees: widely used for file systems and databases
- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL

Bottom line: ST implementation with lg N guarantee for all ops.
- Algorithms are variations on a theme: rotations when inserting.
- Easiest to implement, optimal, fastest in practice: LLRB trees
- Abstraction extends to give search algorithms for huge files: B-trees

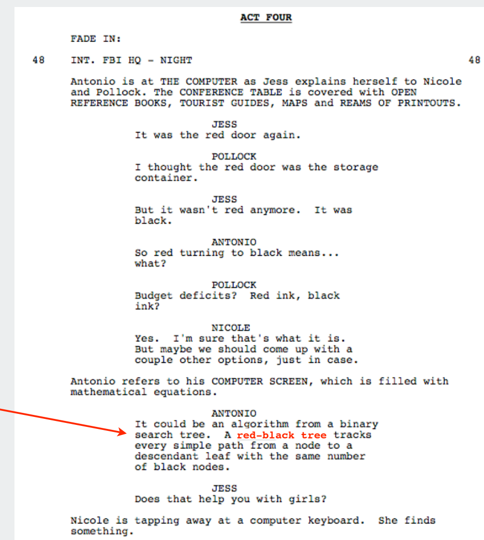
After the break: Can we do better??

Red-black trees in the wild



Common sense. Sixth sense.
Together they're the FBI's newest team.

!!



ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

Nicole is tapping away at a computer keyboard. She finds something.