Balanced Trees

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

References: Algorithms in Java, Chapter 13
Intro to Algs and Data Structs, Section 4.4

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, red-black trees, B-trees.

Summary of symbol-table implementations

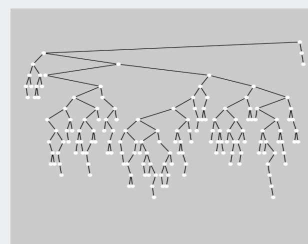
	guarantee			average case			ordered iteration?
implementation	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST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

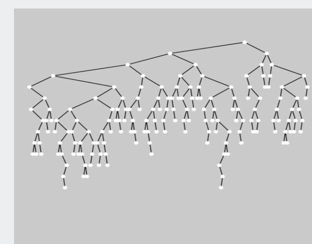
Randomized BSTs provide the desired guarantees

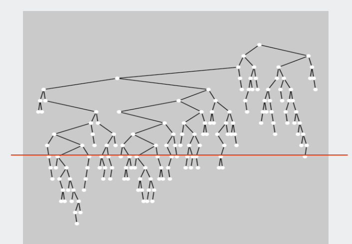
probabilistic, with exponentially small chance of error

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

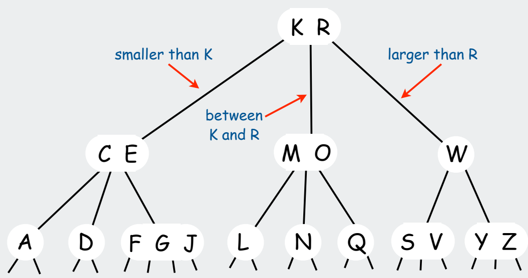
2-3-4 Tree

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.

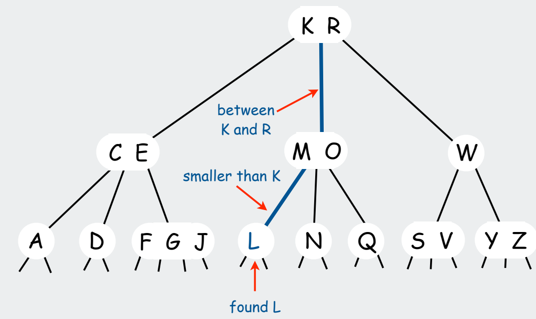


Searching in a 2-3-4 Tree

Search.

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

Ex. Search for L

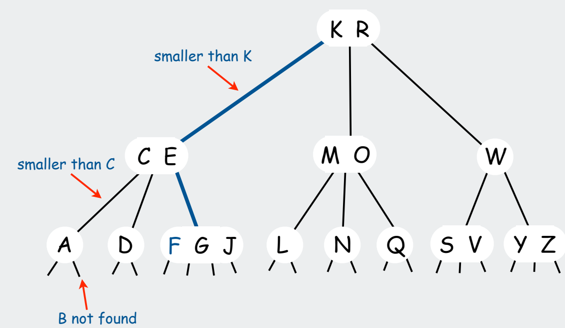


Insertion in a 2-3-4 Tree

Insert.

- Search to bottom for key.

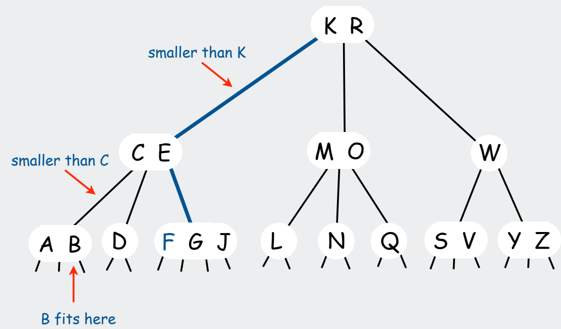
Ex. Insert B



Insertion in a 2-3-4 Tree

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

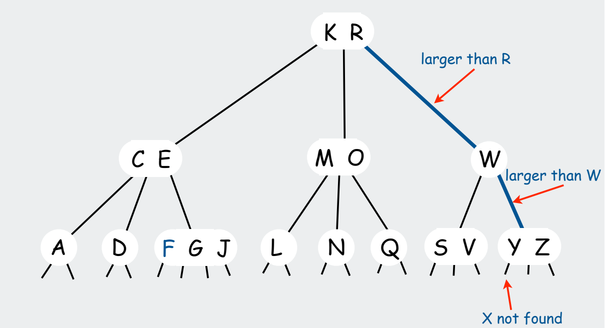
Ex. Insert B



Insertion in a 2-3-4 Tree

Insert.
- Search to bottom for key.

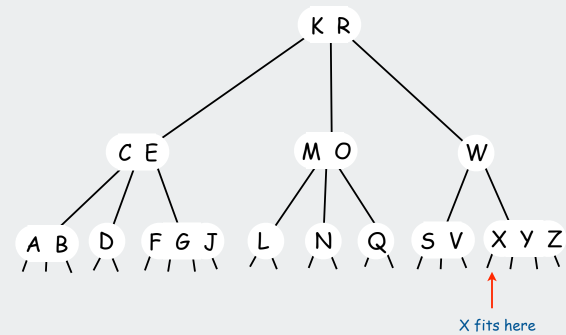
Ex. Insert X



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.

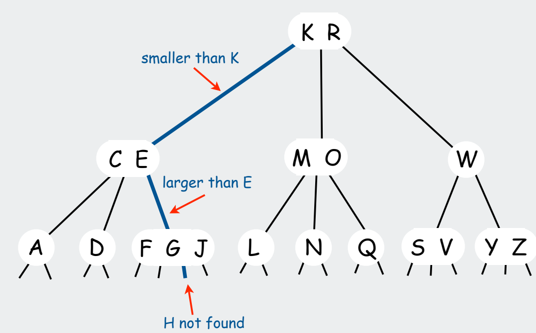
Ex. Insert X



Insertion in a 2-3-4 Tree

Insert.
- Search to bottom for key.

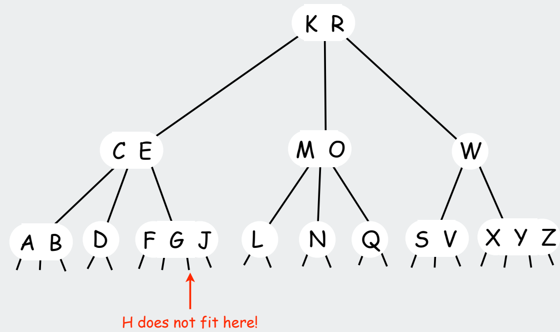
Ex. Insert H



Insertion in a 2-3-4 Tree

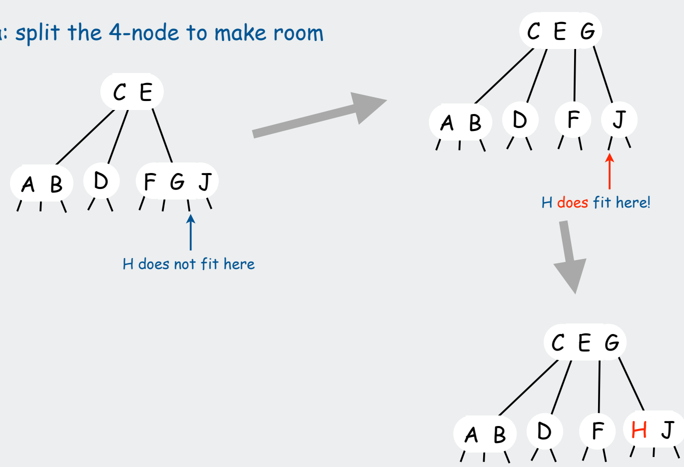
Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.
- 4-node at bottom: ??

Ex. Insert H



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

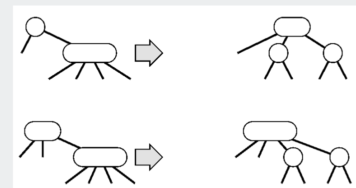
Idea: split the 4-node to make room



Problem: Doesn't work if parent is a 4-node
Solution 1: Split the parent (and continue splitting 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 last node is not a 4-node.
- Transformations to split 4-nodes:



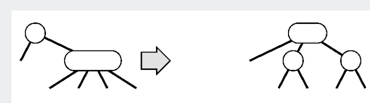
Invariant. Current node is not a 4-node.

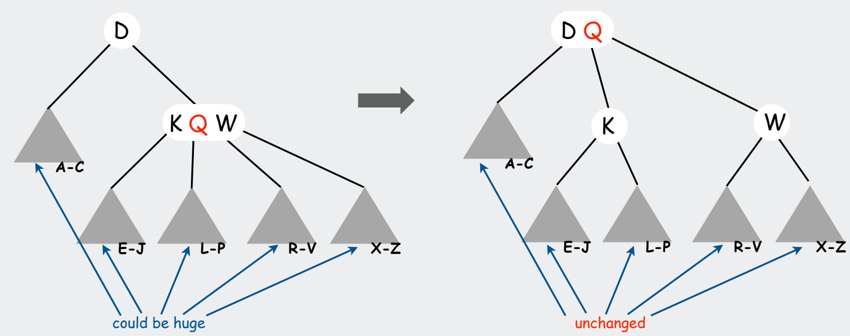
Consequence. Insertion at bottom is easy since it's not a 4-node.

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

Local transformations that work anywhere in the tree

Ex. Splitting a 4-node attached to a 2-node

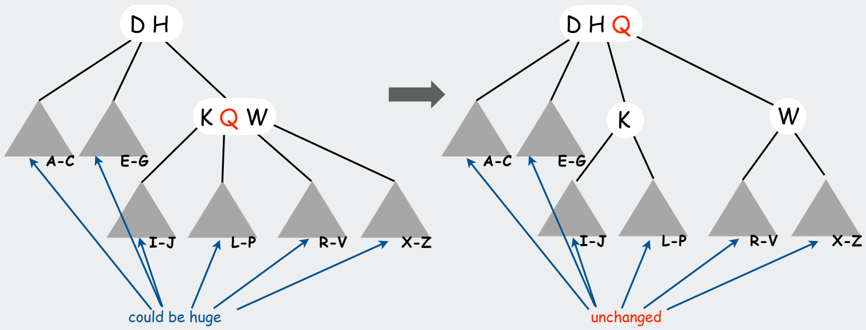




Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree

Ex. Splitting a 4-node attached to a 3-node



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

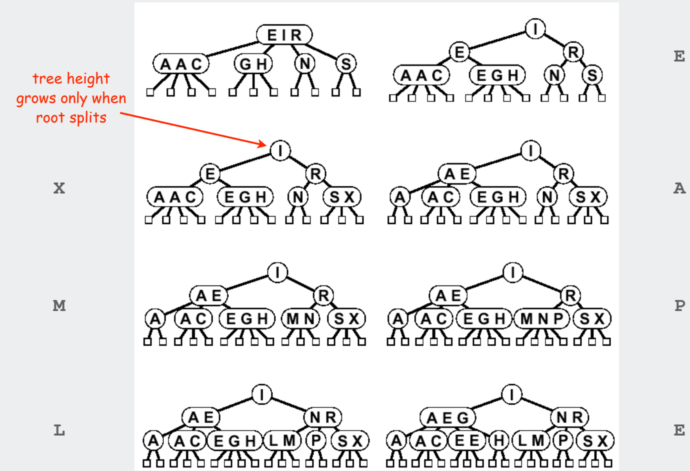
Local transformations that work anywhere in the tree

Splitting a 4-node attached to a 4-node never happens when we split nodes on the way down the tree.

Invariant. Current node is not a 4-node.

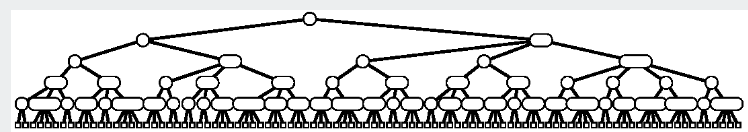
2-3-4 Tree

Tree grows up from the bottom.



2-3-4 Tree: Balance

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 say tuned for an easier way.

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

Summary of symbol-table implementations

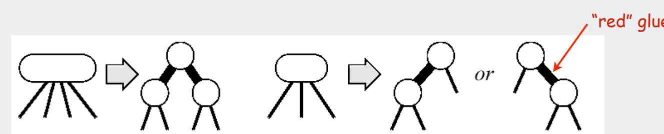
	guarantee			average case			ordered
implementation	search	insert	delete	search	insert	delete	iteration?
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	7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

constants depend upon implementation

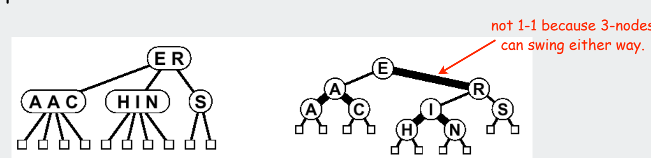
Red-black trees (Guibas-Sedgewick, 1979)

Represent 2-3-4 tree as a BST.

- Use "internal" edges for 3- and 4- nodes.



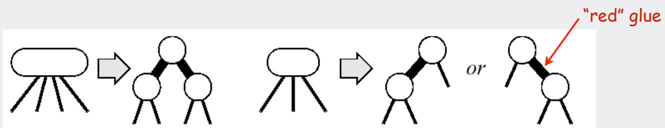
- Correspondence between 2-3-4 trees and red-black trees.



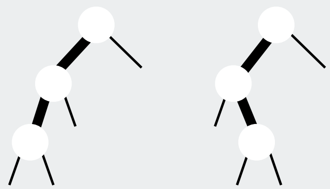
Red-Black Tree

Represent 2-3-4 tree as a BST.

- Use "internal" edges for 3- and 4- nodes.

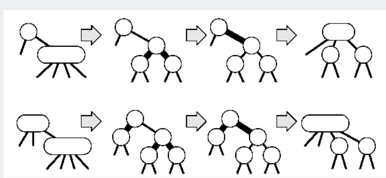


- Disallowed: two red edges in-a-row.



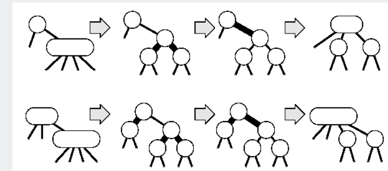
Red-Black Tree: Splitting Nodes

Two easy cases. Switch colors.

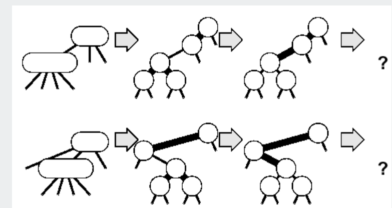


Red-Black Tree: Splitting Nodes

Two easy cases. Switch colors.



Two hard cases. Use rotations.



do single rotation

do double rotation

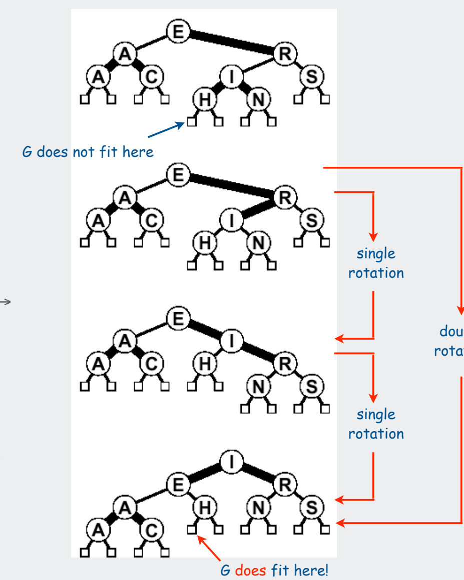
Rotations in a red-black tree

to insert G:

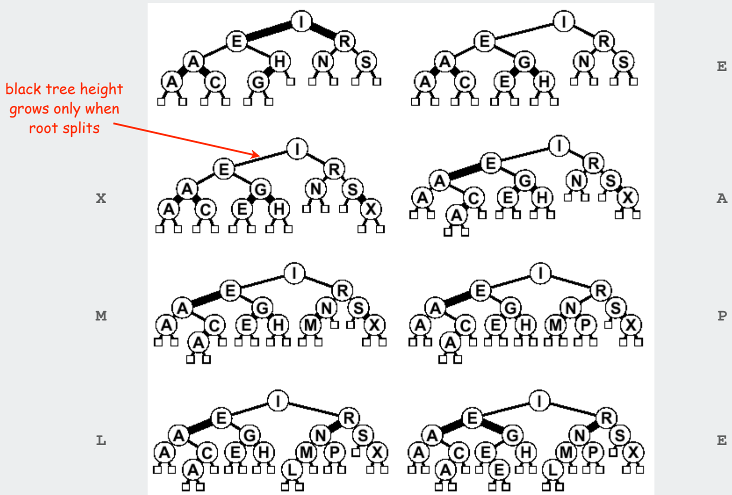
change colors

right rotate R →

left rotate E →



Red-Black Tree: Insertion



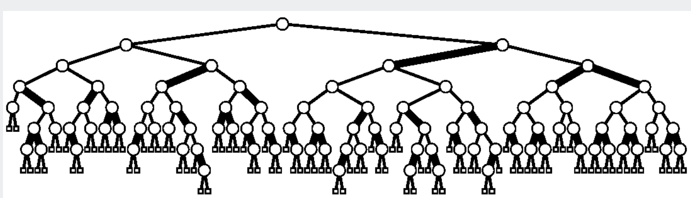
Red-Black Tree: Balance

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

Property B. Never two red links in-a-row.

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

Property D. Height of tree is lg N in the average case.



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.

Runs faster because of better balance in tree.

Insert implementation for red-black trees (skeleton)

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

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

   public void put(Key key, Val val)
   {
     root = put(root, key, value, false);
     root.color = BLACK;
   }
}
```

color of incoming link → (boolean color;)

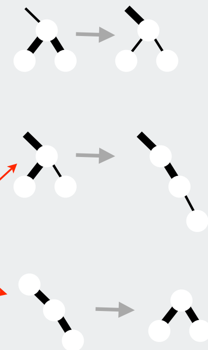
extra argument [stay tuned] → (false)

Insert implementation for red-black trees (strategy)

Search as usual
- if key found reset value, as usual
- if key not found add a new red node at the bottom in the usual way

Split 4-nodes on the way down the tree.
- flip colors
- may leave two red links in a row (unbalanced 4-node) higher up in the tree

Perform rotations on the way up the tree.
- look for two red links in a row
- perform bottom rotation if directions are different
- perform top rotation to balance 4-nodes
- symmetric cases for left and right

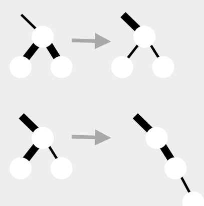


Nonrecursive top-down implementation possible, but requires keeping track of great-grandparent on search path (!) and lots of cases.

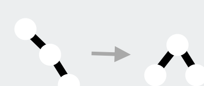
Insert implementation for red-black trees

```
private Node insert(Node x, Key key, Value value, 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, value, 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, value, 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;
}
```

extra argument is true iff x is a right child



Caution: extremely tricky recursive code. Read extremely carefully!

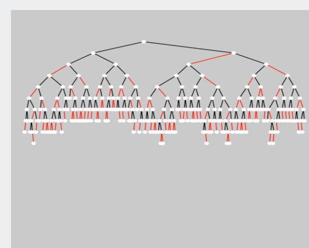


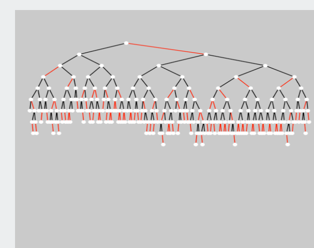
Summary of symbol-table implementations

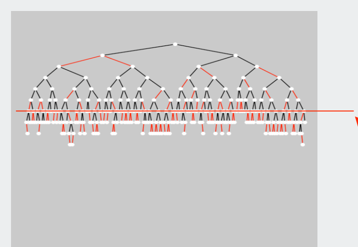
	guarantee			average case			ordered iteration?
implementation	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes
randomized BST	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	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes

exact value of coefficient unknown but extremely close to 1

Typical random red-black trees







$N = 250$

$\lg N \approx 8$

$\lg N - 1 \approx 7$

average node depth

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.

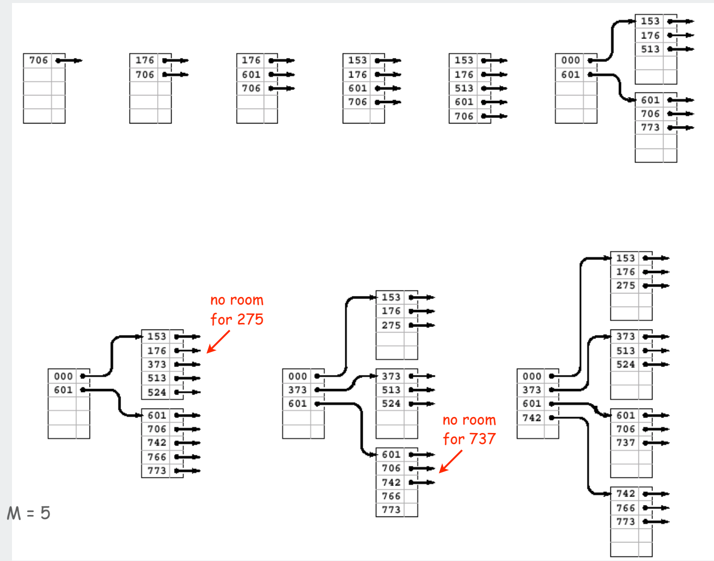
Space-time tradeoff.

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

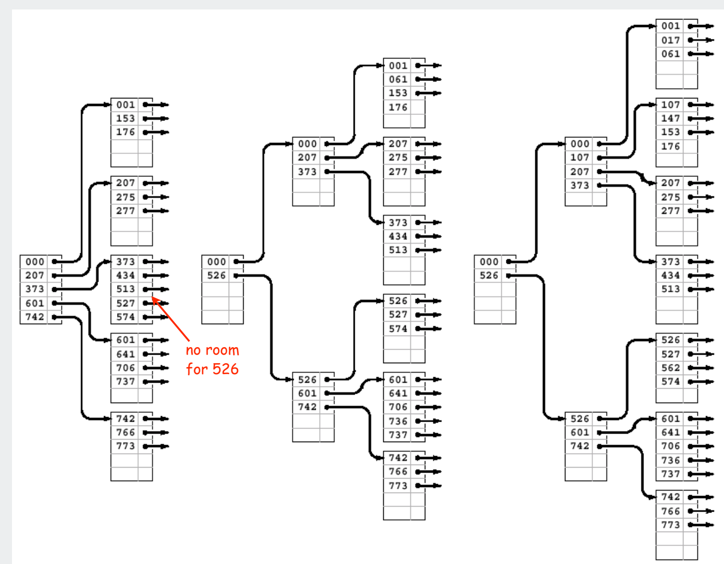
Bottom line. Number of page accesses is $\log_M N$ per op.

3 or 4 in practice!

B-Tree Example



B-Tree Example (cont)



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	1.44 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

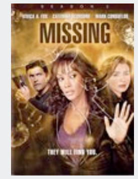
Red-black trees: widely used as system symbol tables

- 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

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.

Balanced trees summary

Goal. ST implementation with lg N guarantee for all ops.

- Difference in quality of guarantee is immaterial.
- Easy to implement other ops: randomized BST.
- Fast average case: red-black tree.
- Algorithms are variations on a theme: rotations when inserting.

Abstraction extends to give search algorithms for huge files.

- B-tree.

Next lecture: Can we do better??