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 $\sim c \log N$ 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.

Typical random BSTs

Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered array</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
<td>yes</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.39 $\lg N$</td>
<td>yes</td>
</tr>
<tr>
<td>randomized BST</td>
<td>$7 \lg N$</td>
<td>$7 \lg N$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Randomized BSTs provide the desired guarantees

This lecture: Can we do better?

Typical random BSTs

$N = 250$
$\lg N \approx 8$
$1.39 \lg N \approx 11$

average node depth
**2-3-4 trees**
- 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.

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

**Example** Search for L

**Insertion**
- Search to bottom for key.

**Example** 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

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

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 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.

Local transformations that work anywhere in the tree

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

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

Local transformations that work anywhere in the tree

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

<table>
<thead>
<tr>
<th>A-C</th>
<th>I-J</th>
<th>L-P</th>
<th>R-V</th>
<th>X-Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-J</td>
<td>L-P</td>
<td>R-V</td>
<td>X-Z</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Q</td>
<td>W</td>
<td>K</td>
<td>W</td>
</tr>
</tbody>
</table>

could be huge  unchanged

<table>
<thead>
<tr>
<th>E-G</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
</tr>
<tr>
<td>H</td>
</tr>
</tbody>
</table>

A-C  E-G
D   H
Q

Local transformations that work anywhere in the tree

Splitting a 4-node attached to a 3-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.

Tree height.

Worst case: \( \log N \) [all 2-nodes]

Best case: \( \log_4 N = 1/2 \log N \) [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.

Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Average Case</th>
<th>Insert</th>
<th>Delete</th>
<th>Ordered Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered array</td>
<td>(\lg N)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 (\lg N)</td>
<td>1.38 (\lg N)</td>
<td>1.38 (\lg N)</td>
<td>yes</td>
</tr>
<tr>
<td>randomized BST</td>
<td>(7 \lg N)</td>
<td>(7 \lg N)</td>
<td>(7 \lg N)</td>
<td>(7 \lg N)</td>
<td>1.38 (\lg N)</td>
<td>1.38 (\lg N)</td>
<td>1.38 (\lg N)</td>
<td>yes</td>
</tr>
<tr>
<td>2-3-4 tree</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>(c \lg N)</td>
<td>yes</td>
</tr>
</tbody>
</table>

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" glue

not 1-1 because 3-nodes can swing either way.
Red-Black Tree

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

```
    or
```

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

Red-Black Tree: Splitting Nodes

Two easy cases. Switch colors.

Red-Black Tree: Splitting Nodes

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 →

G does fit here!
Red-Black Tree: Insertion

black tree height grows only when root splits

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 \(2 \log N + 2\) in the worst case.
Property D. Height of tree is \(\log N\) in the average case.

Search implementation for red-black trees

Search code is the same as elementary BST. Runs faster because of better balance in tree.

Insert implementation for red-black trees (skeleton)

```java
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;
}
```

```java
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, val, false);
        root.color = BLACK;
    }
}
```
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.

Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Average Case</th>
<th>Ordered Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>No</td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
<td>Yes</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>No</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N/2</td>
<td>Yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>Yes</td>
</tr>
<tr>
<td>randomized BST</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>7 lg N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>Yes</td>
</tr>
<tr>
<td>2-3-4 tree</td>
<td>c lg N</td>
<td>c lg N</td>
<td>c lg N</td>
<td>c lg N</td>
<td>c lg N</td>
<td>Yes</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>lg N</td>
<td>lg N</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Exact value of coefficient unknown but extremely close to 1

Typical random red-black trees

\[ N = 250 \]
\[ \lg N \approx 8 \]
\[ \lg N - 1 \approx 7 \]
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.

\[ \text{3 or 4 in practice!} \]
Summary of symbol-table implementations

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered array</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
</tr>
<tr>
<td>unordered list</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.44 lg N</td>
<td>yes</td>
</tr>
<tr>
<td>randomized BST</td>
<td>7 lg N</td>
<td>1.44 lg N</td>
<td>yes</td>
</tr>
<tr>
<td>2-3-4 tree</td>
<td>c lg N</td>
<td>c lg N</td>
<td>yes</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>lg N</td>
<td>yes</td>
</tr>
<tr>
<td>B-tree</td>
<td>1</td>
<td>1</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

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??

Red-black trees in the wild

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