3.3 Balanced Trees

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

Symbol table review

<table>
<thead>
<tr>
<th>Implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>search insert delete</td>
<td>search hit insert delete</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N N N N/2 N N/2</td>
<td>no equals()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>N N lg N N lg N N/2 N/2</td>
<td>yes compareTo()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N N N 1.39 lg N 1.39 lg N ? yes compareTo()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Goal log N log N log N log N log N yes compareTo()

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

2-3 tree

- Allow 1 or 2 keys per node.
  - 2-node: one key, two children.
  - 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.
Perfect balance. Every path from root to null link has same length.
Search in a 2-3 tree

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

Insertion in a 2-3 tree

**Case 1.** Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

**Case 2.** Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
Insertion in a 2-3 tree

**Case 2.** Insert into a 3-node at bottom.
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it’s a 4-node, split it into three 2-nodes.

Remark. Splitting the root increases height by 1.

2-3 tree construction trace

The same keys inserted in ascending order.

Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of operations.
Global properties in a 2-3 tree

Invariant. Symmetric order.
Invariant. Perfect balance.

Pf. Each transformation maintains order and balance.

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

Tree height.
• Worst case: \( \lg N \) [all 2-nodes]
• Best case: \( \log_3 N \approx 0.631 \lg N \) [all 3-nodes]
• Between 12 and 20 for a million nodes.
• Between 18 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>( N )</td>
<td>( N )</td>
<td>( N/2 )</td>
<td>( N/2 )</td>
</tr>
<tr>
<td>(linked list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( N/2 )</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( N )</td>
<td>1.39 ( \lg N )</td>
<td>1.39 ( \lg N )</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>( c \lg N )</td>
<td>( c \lg N )</td>
<td>( c \lg N )</td>
<td>( c \lg N )</td>
</tr>
</tbody>
</table>

constants depend upon implementation
2-3 tree: implementation?

Direct implementation is complicated, because:
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

**Bottom line.** Could do it, but there’s a better way.

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

1. Represent 2-3 tree as a BST.
2. Use “internal” left-leaning links as “glue” for 3-nodes.

An equivalent definition

A BST such that:
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

```
red links "glue" nodes within a 3-node
black links connect 2-nodes and 3-nodes
```

```
larger key is root
```

```
"perfect black balance"
```
Left-leaning red-black trees: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.

Red-black tree representation

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

Red-black tree representation

Private static final boolean RED = true;
Private static final boolean BLACK = false;

Private class Node
{
Key key;
Value val;
Node left, right;
boolean color; // color of parent link
}

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

Search implementation for red-black trees

Observation. Search is the same as for elementary BST (ignore color).

Remark. Many other ops (e.g., ceiling, selection, iteration) are also identical.

Elementary red-black tree operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

Invartants. Maintains symmetric order and perfect black balance.
Elementary red-black tree operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

```
private Node rotateRight(Node h) {
    assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black tree operations

Color flip. Recolor to split a (temporary) 4-node.

```
private void flipColors(Node h) {
    assert !isRed(h) && isRed(h.left) && isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black tree operations

```
{insert C
  add new node here
  right link red so rotate left
}
```

Warmup 1. Insert into a tree with exactly 1 node.
**Insertion in a LLRB tree**

**Case 1.** Insert into a 2-node at the bottom.
- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.

**Case 2.** Insert into a 3-node at the bottom.
- Do standard BST insert; color new link red.
- Rotate to make lean left (if needed).

**Insertion in a LLRB tree: passing red links up the tree**

**Case 2.** Insert into a 3-node at the bottom.
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

**Warmup 2.** Insert into a tree with exactly 2 nodes.

**Case 2.** Insert into a 3-node at the bottom.
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat Case 1 or Case 2 up the tree (if needed).
**Standard indexing client.**

Insertion in a LLRB tree: Java implementation

Same code for both cases.
- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.

```java
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else h.val = val;
    if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
    if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  && isRed(h.right))     h = flipColors(h);
    return h;
}
```

**Insertion in a LLRB tree: visualization**

255 insertions in ascending order

N = 255
max = 8
avg = 7.0
opt = 7.0
Insertion in a LLRB tree: visualization

N = 255
max = 8
avg = 7.0
opt = 7.0

Insertion in a LLRB tree: visualization

N = 50

Insertion in a LLRB tree: visualization

N = 255
max = 10
avg = 7.3
opt = 7.0

Insertion in a LLRB tree: visualization

Balance in LLRB trees

Proposition. Height of tree is \( \leq 2 \lg N \) in the worst case.

Pf.
- Every path from root to null link has same number of black links.
- Never two red links in-a-row.

Property. Height of tree is \( \sim 1.00 \lg N \) in typical applications.
Why left-leaning trees?

Simplified code.
- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.
- Delete min/max.
- Arbitrary delete.

Improves widely-used algorithms.
- AVL trees, 2-3 trees, 2-3-4 trees.
- Red-black trees.

Bottom line. Left-leaning red-black trees are the simplest balanced BST to implement and the fastest in practice.
**File system model**

**Page.** Contiguous block of data (e.g., a file or 4096-byte chunk).

**Probe.** First access to a page (e.g., from disk to memory).

**Model.** Time required for a probe is much larger than time to access data within a page.

**Goal.** Access data using minimum number of probes.

---

**B-trees (Bayer-McCreight, 1972)**

**B-tree.** Generalize 2-3 trees by allowing up to \( M-1 \) key-link pairs per node.
- At least 2 key-link pairs at root.
- At least \( M/2 \) key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

**Anatomy of a B-tree set (\( M = 6 \))**

- 2-node
- 3-node
- 4-node
- 5-node (full)
- External 3-node
- External 4-node
- 3-node
- External
- Internal
- External 3-node (full)
- 2-node
- External 4-node
- 3-node
- 2-node
- 3-node
- 4-node
- 5-node (full)
- External 3-node
- Internal
- External 4-node
- 3-node

**Searching in a B-tree**

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.

**Insertion in a B-tree**

- Search for new key.
- Insert at bottom.
- Split nodes with \( M \) key-link pairs on the way up the tree.
Proposition. A search or an insertion in a B-tree of order M with N keys requires between \( \log_{M/2} N \) and \( \log_{M-1} N \) probes.

**Pf.** All internal nodes (besides root) have between \( M/2 \) and \( M-1 \) links.

In practice. Number of probes is at most 4.

Optimization. Always keep root page in memory.

Balance in B-tree

Building a large B tree

B-tree variants. B+ tree, B*tree, B# tree, ...

B-trees (and variants) are 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

Red-black trees are widely used as system symbol tables.

- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, linux/rbtree.h.

Common sense. Sixth sense. Together they’re the FBI’s newest team.
Red-black trees in the wild

Antonio is at the COMPUTER as Dena explains herself to Nicole and Melissa. The COMPUTER screen is covered with happy
REFERENCE BOOKS. Wrinkly.sidebar, maps and notes of PRINTOUTS.

JESS
It was the red door again.

PAUL
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...

PAUL
Red, black, black, red.

SUE
Yes, I’m sure that’s what it is.

ANTONIO
Red, black, black, red.

ANTONIO
It could be a POSSIBILITY of a binary
search tree. The red nodes create
edges from each node to a
black node that are the same color.

JESS
Does that help you with girls?

Antoni is tapping away at a computer keyboard. She looks
something.