3.3 Balanced Search Trees

- 2-3 search trees
- red-black BSTs
- 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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</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>N</td>
<td>N</td>
<td>lg N</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>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

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

**Invariants.** Maintains symmetric order and perfect balance.

**Pf.** Each transformation maintains symmetric order and perfect balance.

![Diagram of 2-3 tree with invariants](image)

2-3 tree: performance

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

![Diagram of perfect balance](image)

**Tree height.**
- **Worst case:** $\lg N$, [all 2-nodes]
- **Best case:** $\log_3 N \approx .631 \lg N$, [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></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\lg N$</td>
</tr>
<tr>
<td>BST</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</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 BSTs (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.

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

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

Red-black BST representation

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

Elementary red-black BST operations

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

Invariants. Maintains symmetric order and perfect black balance.
Elementary red-black BST operations

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

\[
\begin{align*}
\text{private Node rotateRight}(\text{Node } h) \{ \\
\text{assert } (h \neq \text{null}) \&\& \text{isRed}(h.\text{left}); \\
\text{Node } x = h.\text{left}; \\
h.\text{left} = x.\text{right}; \\
x.\text{right} = h; \\
x.\text{color} = h.\text{color}; \\
h.\text{color} = \text{RED}; \\
\text{return } x; \\
\}
\end{align*}
\]

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.

\[
\begin{align*}
\text{insert } C \\
\text{add new node here} \\
\text{right link red so rotate left} \\
\end{align*}
\]

Elementary red-black BST operations

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

\[
\begin{align*}
\text{private void flipColors}(\text{Node } h) \{ \\
\text{assert } !\text{isRed}(h) \&\& \text{isRed}(h.\text{left}) \&\& \text{isRed}(h.\text{right}); \\
h.\text{color} = \text{RED}; \\
h.\text{left}.\text{color} = \text{BLACK}; \\
h.\text{right}.\text{color} = \text{BLACK}; \\
\}
\end{align*}
\]

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree

Warmup 1. Insert into a tree with exactly 1 node.

\[
\begin{align*}
\text{left} \\
\text{search ends at this null link} \\
\text{root} \\
\text{red link to new node containing a converts 2-node to 3-node} \\
\end{align*}
\]
**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 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.
- Search ends at this null link.
  - Search ends at this null link.
  - Rotated left.
  - Rotated right.
  - Colors flipped to black.

**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).
- Repeat case 1 or case 2 up the tree (if needed).
LLRB tree construction trace

Standard indexing client.

Insertion in a LLRB tree: Java implementation

- 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
        if (cmp == 0)
            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))     flipColors(h);
    return h;
}
```

LLRB tree construction trace

Standard indexing client (continued).

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

255 insertions in descending order

Balance in LLRB trees

**Proposition.** Height of tree is $\leq 2 \log_2 N$ 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 $\sim 1.00 \log_2 N$ in typical applications.
**ST implementations: frequency counter**

- Costs for `java FrequencyCounter 8 < tale.txt` using RedBlackBST:
  - 14350 operations, cost 12.9.
- Costs for `java FrequencyCounter 8 < tale.txt` using BST:
  - 14350 operations, cost 13.9.

**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>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>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>
<tr>
<td>red-black BST</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
</tbody>
</table>

* exact value of coefficient unknown but extremely close to 1.

**Why left-leaning trees?**

**Old code (that students had to learn in the past)**

```java
private Node put(Node x, Key key, Value val, boolean sw) {
    if (x == null)
        return new Node(key, val, 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.left = put(x.left, key, val, false); if (isRed(h) && isRed(x.left) && sw)
            x = rotateRight(x); if (isRed(h.left) && isRed(h.left.left))
            x = rotateRight(x); x.color = BLACK; x.right.color = RED; }
    else if (cmp > 0)
        { x.right = put(x.right, key, val, true); if (isRed(h) && isRed(x.right) && !sw)
            x = rotateLeft(x); if (isRed(h.right) && isRed(h.right.right))
            x = rotateLeft(x); x.color = BLACK; x.left.color = RED; }
    else x.val = val;
    return x;
}
```

**New code (that you have to learn)**

```java
public 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))
        flipColors(h);
    return h;
}
```

**Bottom line.** Left-leaning red-black BSTs are among the simplest balanced BSTs to implement and among the fastest in practice.

**Why left-leaning red-black BSTs?**

**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 balanced search trees.**
- AVL trees, splay trees, randomized BSTs, ...
- 2-3 trees, 2-3-4 trees.
- Red-black BSTs.
War story: why red-black?

Xerox PARC innovations. [1970s]
- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.

Xerox Alto

**A Hierarchical Framework For Balanced Trees**

L. G. A. & M. P. McIlroy

Department of Computer Science
Carnegie-Mellon University

Abstract

In this paper we present a unified framework for the implementation and study of balanced tree algorithms. We have been able to use this framework to develop a number of balanced tree algorithms which are simple to implement, have good asymptotic performance, and have properties about the structure that are easy to understand. We describe three algorithms which are based on this framework: two algorithms for 2-3 search trees and one algorithm for red-black balanced search trees.

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).

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

Cost model. Number of probes.

Goal. Access data using minimum number of probes.

War story: red-black BSTs

Telephone company contracted with database provider to build real-time database to store customer information.

Database implementation.
- Red-black BST search and insert; Hibbard deletion.
- Exceeding height limit of 80 triggered error-recovery process.

Extended telephone service outage.
- Main cause = height bounded exceeded!
- Telephone company sues database provider.
- Legal testimony:

"If implemented properly, the height of a red-black BST with \( N \) keys is at most \( 2 \log N \)." — expert witness
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.

**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.

**Searching in a B-tree**
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.

**Balance in B-tree**

**Proposition.** A search or an insertion in a B-tree of order \( M \) with \( N \) keys requires between \( \log_M N - 1 \) and \( \log_M 2N \) 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.
Building a large B tree

Building a large B-tree full page, about to split

Building a large B-tree external nodes (line segment of length proportional to number of keys in that node)

Balanced 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.

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 BSTs in the wild

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

Red-black BSTs in the wild