3.3 Balanced Search Trees

- 2-3 search trees
- red-black BSTs
- B-trees
## Symbol table review

<table>
<thead>
<tr>
<th>implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</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 (unordered 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>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

**Challenge.** Guarantee performance.

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

introduced to the world in COS 226, Fall 2007
2-3 search trees
- 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.
2-3 tree demo
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.
2-3 tree: performance

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

Tree height.
- Worst case:
- Best case:
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:** $\log_3 N \approx .631 \lg N$. [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>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</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 (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>Ig N</td>
<td>N</td>
<td>N</td>
<td>Ig N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 Ig N</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>c Ig N</td>
<td>c Ig N</td>
<td>c Ig N</td>
<td>c Ig 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.
• 2-3 search trees
• red-black BSTs
• B-trees
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.
Search implementation for red-black BSTs

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

but runs faster because of better balance

```
public Val get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

**Remark.** Most other ops (e.g., ceiling, selection, iteration) are also identical.
Red-black BST representation

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

```java
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;
}
```

null links are black

h.left.color is RED
h.right.color is BLACK
Elementary red-black BST operations

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

```
private Node rotateLeft(Node h) {
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

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

```java
private Node rotateLeft(Node h) {
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```
**Elementary red-black BST operations**

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

**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.

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

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

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

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

```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert 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 BST operations.
**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.
Warmup 2. Insert into a tree with exactly 2 nodes.

**Insertion in a LLRB tree**

- **Larger**:
  - Search ends at this null link.
  - Attached new node with red link.
  - Colors flipped to black.

- **Smaller**:
  - Search ends at this null link.
  - Attached new node with red link.
  - Rotated right.
  - Colors flipped to black.

- **Between**:
  - Search ends at this null link.
  - Attached new node with red link.
  - Rotated left.
  - Colors flipped to black.

[Diagrams showing the step-by-step process of insertion in a LLRB tree with labels for search ends, attached new nodes, rotations, and color flips.]
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).
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 insertion demo
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 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;
}
```

Insert at bottom (and color red)

Lean left

Balance 4-node

Split 4-node

Only a few extra lines of code provides near-perfect balance
Insertion in a LLRB tree: visualization

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

255 insertions in ascending order
Insertion in a LLRB tree: visualization

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

255 insertions in descending order
Insertion in a LLRB tree: visualization

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

255 random insertions
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.
## ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst-case cost (after N inserts)</th>
<th>Average case (after N random inserts)</th>
<th>Ordered iteration?</th>
<th>Key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td><strong>sequential search (unordered list)</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td><strong>binary search (ordered array)</strong></td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td><strong>2-3 tree</strong></td>
<td>c lg N</td>
<td>c lg N</td>
<td>c lg N</td>
<td>c lg N</td>
</tr>
<tr>
<td><strong>red-black BST</strong></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*
War story: why red-black?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...

XEROX. PARC

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas
Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University

Robert Sedgewick*
Program in Computer Science
Brown University

and

Providence, R. I.

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its
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 $2 \lg N$. ” — expert witness
- 2-3 search trees
- red-black BSTs
- B-trees
File system model

**Page.** Contiguous block of data (e.g., a file or 4,096-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.
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 ($M = 6$)

- Sentinels
- 2-node
- 3-node
- 4-node
- 5-node (full)
- Client keys (black) are in external nodes
- Each red key is a copy of min key in subtree
- All nodes except the root are 3-, 4- or 5-nodes

Choose $M$ as large as possible so that $M$ links fit in a page, e.g., $M = 1024$. 
Searching in a B-tree

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

Searching in a B-tree set (M = 6)
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.
Balance in B-tree

**Proposition.** A search or an insertion in a B-tree of order $M$ with $N$ keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

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

**In practice.** Number of probes is at most 4.  \[ M = 1024; N = 62 \text{ billion} \]

**Optimization.** Always keep root page in memory.
Building a large B tree

- full page, about to split
- 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

ACT FOUR

FADE IN:

INT. FBI HQ - NIGHT

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?