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 (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>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.
Search in a 2-3 tree

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

**Successful search for H**

- H is less than M so look to the left.

**Unsuccessful search for B**

- B is less than M so look to the left.

found H so return value (search hit)

B is between A and C so look in the middle
link is null so B is not in the tree (search miss)
Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.
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.

why middle key?

![Diagram](image)

search for Z ends at this 3-node
replace 3-node with temporary 4-node containing Z
replace 2-node with new 3-node containing middle key
split 4-node into two 2-nodes pass middle key to parent
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.
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

Standard indexing client.
2-3 tree construction trace

The same keys inserted in ascending order.

insert A

P

R

S

X
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: $\lg N$. [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.
- 2-3 search trees
- red-black BSTs
- B-trees
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

```java
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
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

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

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.

```
private Node rotateRight(Node h) {
    assert 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 BST operations

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

![Diagram of a tree with nodes labeled A, E, S, and their children categorized by color]

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

Insertion in a LLRB tree: overview
Insertion in a LLRB tree

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

```
left

root

search ends at this null link

root

red link to new node containing a converts 2-node to 3-node

right

root

search ends at this null link

root

attached new node with red link

rotated left to make a legal 3-node
```
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
**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).
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 insertion demo
LLRB tree insertion trace

Standard indexing client.
LLRB tree insertion trace

Standard indexing client (continued).

red-black BST  corresponding 2-3 tree
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;
}
```

only a few extra lines of code to provide 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>Guarantee average case</th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
<th>Search Hit</th>
<th>Insert</th>
<th>Delete</th>
<th>Ordered iteration?</th>
<th>Operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sequential search (linked list)</strong></td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>binary search (ordered array)</strong></td>
<td></td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
<td>?</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td><strong>2-3 tree</strong></td>
<td></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>
<td>compareTo()</td>
</tr>
<tr>
<td><strong>red-black BST</strong></td>
<td></td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>yes</td>
<td>compareTo()</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.
- ...

---

**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*  
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
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)](image)

- **Sentinel key:** The first key in the tree.
- **External 3-node:** Contains client keys.
- **External 5-node (full):** Contains client keys and 4 copies of other keys.
- **2-node:** Contains 2 client keys.
- **Internal 3-node:** Contains 3 client keys and 2 copies of other keys.
- **Each red key is a copy of min key in subtree:** Helps in search.
- **All nodes except the root are 3-, 4- or 5-nodes:** Ensures balance in the tree.

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.
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.

![Diagram of B-tree insertion](image-url)

Inserting a new key into a B-tree set
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.

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.