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 ops?</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</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$\frac{1}{2}N$</td>
</tr>
<tr>
<td>(unordered 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>$\frac{1}{2}N$</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>goal</td>
<td>$\log N$</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.
2-3 tree demo

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

search for H

Insertion into a 2-3 tree

Insertion into a 2-node at bottom.
- Add new key to 2-node to create a 3-node.

insert G

2-3 tree construction demo

Insertion 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.

insert Z

insert S
2-3 tree construction 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: \( \log_3 N \approx 0.631 \lg N \). [all 3-nodes]
• Between 12 and 20 for a million nodes.
• Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed logarithmic performance for search and insert.

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.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search insert delete</td>
<td>search hit insert delete</td>
<td>ordered</td>
<td>key interface</td>
<td></td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>( N )</td>
<td>( N )</td>
<td>( \frac{1}{2} N )</td>
<td>( \frac{1}{2} N )</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>( \lg N )</td>
<td>( N )</td>
<td>( \frac{1}{2} N )</td>
<td>( \frac{1}{2} N )</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>

3.3 Balanced Search Trees

‣ 2-3 search trees
‣ red-black BSTs
‣ B-trees

fantasy code

public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?

Approach 1: regular BST.
- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.

Approach 2: regular BST with "glue" nodes.
- Wastes space, wasted link.
- Code probably messy.

Approach 3: regular BST with red "glue" links.
- Widely used in practice.
- Arbitrary restriction: red links lean left.

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.

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.

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).

```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., floor, iteration, selection) are also identical.

Red-black BST representation

Each node is pointed to by precisely one link (from its parent) \( \Rightarrow \) 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;
    int N; // # nodes in this subtree
    boolean color; // color of link from
    //   parent to this node
}

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

Elementary red-black BST operations

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

```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;
}
```

Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

During internal operations, maintain:
- Symmetric order.
- Perfect black balance.
  [ but not necessarily color invariants ]

How? Apply elementary red-black BST operations: rotation and color flip.
**Elementary red-black BST operations**

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

rotate E left  
(after)

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

```java
cprivate cNode crotateLeftkNode cl{
   cassert cisRedkhqrightl;
   cNode cx = chqright;
   chqrightc=cxqleft;
   cxqleftc=ch;
   cxqcolorc=chqcolor;
   chqcolorc=cRED;
   creturn cx;
}
```

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

rotate S right  
(before)

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

```java
cprivate cNode crotateRightkNode cl{
   cassert cisRedkh.leftl;
   cassert cisRedkhqleftl;
   cassert cisRedkhqrightl;
   chqcolorc=cRED;
   chqleftqcolorc=cBLACK;
   chqrightqcolorc=cBLACK;
   c}
```

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

rotate S right  
(after)

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

```java
cprivate cNode crotateRightkNode cl{
   cassert cisRedkh.leftl;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

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

flip colors  
(before)

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

```java
cprivate void flipColorskNode cl{
   cassert cisRedh();
   cassert cisRedh.left();
   cassert cisRedh.right();
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```
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

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.

Insert C

Case 2. Insert into a single 3-node (three cases).
- If search ends at this null link, rotate left.
- If search ends at this null link, rotate right.
- If search ends at this null link, rotate rotated left.
- if rotated left, rotated right.

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

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).

Red-black BST construction demo

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
</table>

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).

Red-black BST construction demo

<table>
<thead>
<tr>
<th>red–black BST</th>
</tr>
</thead>
</table>
Insertion in a LLRB tree: Java implementation

Same code for all 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;
}
```

Insertion in a LLRB tree: visualization

255 insertions in ascending order

255 insertions in descending order

255 random insertions
Balance in LLRB trees

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

**Proof.**
- 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 \( \sim 1.0 \lg N \) in typical applications.

War story: why red-black?

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

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Key Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>( N )</td>
<td>( 1/2 N )</td>
<td>equals()</td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( 1.39 \lg N )</td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td>2-3 tree</td>
<td>( c \lg N )</td>
<td>( c \lg N )</td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td>red-black BST</td>
<td>( 2 \lg N )</td>
<td>( 1.0 \lg N )</td>
<td>compareTo()</td>
<td></td>
</tr>
</tbody>
</table>

* exact value of coefficient unknown but extremely close to 1

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
3.3 BALANCED SEARCH TREES

- 2-3 search trees
- red-black BSTs
- B-trees

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.

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.

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.

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. $M = 1024; N = 62$ billion $\log_{M/2} N \leq 4$

**Optimization.** Always keep root page in memory.

Building a large B-tree

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.
- Emacs: conservative stack scanning.

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

B-trees (and variants) are widely used for file systems and databases.
- Windows: NTFS.
- 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.

ACT FOUR

Antonio is at the computer as Jess explains herself to Nicole and Pollack. The conference table is covered with open reference books, tourist guides, maps, and sheets 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
Bad ink 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 single path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?