3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context

https://algs4.cs.princeton.edu
### Symbol table review

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>binary search (sorted array)</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>goal</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

**Challenge.** $\Theta(\log n)$ time in worst case.

**This lecture.** 2–3 trees and left-leaning red-black BSTs.
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context
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 the root to a null link has the same length.
**Search.**
- Compare search key against key(s) in node.
- Find interval containing search key.
- Follow associated link (recursively).
2–3 tree: insertion

Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

**insert G**
2–3 tree: insertion

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**
Suppose that you insert P into the following 2–3 tree. What will be the root of the resulting 2–3 tree?

A. E
B. E R
C. M
D. P
E. R
What is the maximum height of a 2–3 tree containing $n$ keys?

A. $\sim \log_3 n$
B. $\sim \log_2 n$
C. $\sim 2 \log_2 n$
D. $\sim n$
2–3 tree: performance

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

Key property. The height of a 2–3 tree containing \( n \) keys is \( \Theta(\log n) \).
- Min: \( \log_3 n \approx 0.631 \log_2 n \). [all 3-nodes]
- Max: \( \log_2 n \). [all 2-nodes]
- Between 12 and 20 for a million keys.
- Between 18 and 30 for a billion keys.

Bottom line. Search and insert take \( \Theta(\log n) \) time in the worst case.
## ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(sorted array)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>2-3 trees</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

*but hidden constant $c$ is large (depends upon 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.

```
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);
}
```

Bottom line. Could do it, but there's a better way.
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context
How to implement 2–3 trees as binary search trees?

**Challenge.** How to represent a 3 node?

**Approach 1.** Two BST nodes.
- No way to tell a 3-node from two 2-nodes.
- Can’t (uniquely) map from BST back to 2–3 tree.

**Approach 2.** Two BST nodes, plus red “glue” node.
- Wastes space for extra node.
- Messy code.

**Approach 3.** Two BST nodes, with red “glue” link.
- Widely used in practice.
- Arbitrary restriction: red links lean left.
Left-leaning red–black BSTs

1. Represent 2–3 tree as a BST.
2. Use “internal” left-leaning red links as “glue” for 3–nodes.
Left-leaning red–black BSTs: 1–1 correspondence with 2–3 trees

**Key property.** 1–1 correspondence between 2–3 trees and LLRB trees.
An equivalent definition of LLRB trees (without reference to 2–3 trees)

**Def.** A red–black BST is a BST such that:

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

![Red-Black Tree Diagram]
Which LLRB tree corresponds to the following 2–3 tree?

A.

B.

C. Both A and B.

D. Neither A nor B.
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 {
    private Key key;
    private Value val;
    private Node left, right;
    private boolean color;
}

private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```
Review: the road to LLRB trees

BSTs (can get imbalanced)

2–3 trees (balanced but awkward to implement)

3-nodes “glued” together with red links

how we draw LLRB trees (color in links)

how we implement LLRB trees (color in nodes)
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context
Search in a red–black BST

**Observation.** Red-black BSTs are BSTs $\Rightarrow$ search is the same as for BSTs (ignore color).

![Red-black BST diagram]

```
public Value 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 return x.val;
    }
    return null;
}
```

**Remark.** Many other operations (iteration, floor, rank, selection) are also identical.
Insertion into 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 ]

**Example violations of color invariants:**

- [Diagram showing examples of violations: right-leaning red link, two red children (a temporary 4-node), left-left red (a temporary 4-node), left-right red (a temporary 4-node).]

**To restore color invariants:** perform rotations and color flips.
**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.left);
    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.

```
private Node rotateLeft(Node h) {
    assert !isRed(h.left);
    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.

```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    assert !isRed(h.right);
    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

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

![Diagram of right rotation](image)

```java
private Node rotateRight(Node h) {
    assert isRed(h.left);
    assert !isRed(h.right);
    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.

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.
Which sequence of elementary operations transforms the red-black BST at left to the one at right?

A. Color flip E; left rotate R.
B. Color flip R; left rotate E.
C. Color flip R; left rotate R.
D. Color flip R; right rotate E.
Insertion into a LLRB tree

- Do standard BST insert and color new link red.
- Repeat up the tree until color invariants restored:
  - two left red links in a row? ⇒ rotate right
  - left and right links both red? ⇒ color flip
  - only right link red? ⇒ rotate left
Insertion into a LLRB tree

- Do standard BST insert and color new link red.
- Repeat up the tree until color invariants restored:
  - two left red links in a row? ⇒ rotate right
  - left and right links both red? ⇒ color flip
  - only right link red? ⇒ rotate left

---

**Diagram**

Inserting P

1. **Red-Black BST**
   - Add new node here
   - Right link red: rotate left

2. **Red-Black BST**
   - Both children red: flip colors

3. **Red-Black BST**
   - Two lefts in a row: rotate right

4. **Red-Black BST**
   - Both children red: flip colors
Red–black BST construction demo

insert S E A R C H X M P L
Insertion into a LLRB tree: Java implementation

- Do standard BST insert and color new link red.
- Repeat up the tree until color invariants restored:
  - only right link red? \( \Rightarrow \) rotate left
  - two left red links in a row? \( \Rightarrow \) rotate right
  - left and right links both red? \( \Rightarrow \) color flip

```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)) flipColors(h);

    return h;
}
```

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

255 insertions in random order

n = 255  
height = 9  
average depth = 6.3
Insertion into a LLRB tree: visualization

n = 255
height = 7
average depth = 6.0

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

\[ n = 254 \]
\[ \text{height} = 13 \]
\[ \text{average depth} = 6.5 \]

254 insertions in descending order
Balance in LLRB trees

**Proposition.** Height of LLRB tree is \( \leq 2 \log_2 n \).

**Pf.**

- Black height = height of corresponding 2–3 tree \( \leq \log_2 n \).
- Never two red links in a row.
  \[ \Rightarrow \text{height of LLRB tree} \leq (2 \times \text{black height}) + 1 \]
  \[ \leq 2 \log_2 n + 1. \]

- [ A slightly more careful argument shows height \( \leq 2 \log_2 n \). ]
# ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>binary search (sorted array)</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>2–3 trees</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>red-black BSTs</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

hidden constant $c$ is small ($\leq 2 \log_2 n$ compares)
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs (representation)
- red–black BSTs (operations)
- context
Balanced search trees in the wild

Red–black BSTs are widely used as system symbol tables.

- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: CFQ I/O scheduler, VMAs, linux/rbtree.h.

Other balanced BSTs. AVL trees, splay trees, randomized BSTs, rank-balanced BSTs, ....

B-trees (and cousins) are widely used for file systems and databases.

- Windows: NTFS.
- Mac OS X: HFS, HFS+, APFS.
- Linux: ReiserFS, XFS, ext4, JFS, Btrfs.
- Databases: Oracle, DB2, Ingres, SQL, PostgreSQL.
**War story 1: red–black BSTs**

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

**Database implementation.**

- Red–black BST.
- Exceeding height limit of 80 triggered error-recovery process.

Extended telephone service outage.

- Main cause = height bound exceeded!
- Telephone company sues database provider.
- Legal testimony:

> “If implemented properly, the height of a red–black BST with n keys is at most \(2 \log_2 n\).” — expert witness
War story 2: red–black BSTs

I was just asked to balance a Binary Search Tree by JFK’s airport immigration. Welcome to America.

8:26 AM · 26 Feb 2017 from Manhattan, NY
8,025 Retweets 7,087 Likes

Celestine Omin @cyberomin · 26 Feb 2017

I was too tired to even think of a BST solution. I have been travelling for 23hrs. But I was also asked about 10 CS questions.

Celestine Omin @cyberomin · 26 Feb 2017

Sad thing is, if I didn’t give the Wikipedia definition for these questions, it was considered a wrong answer.

Simon Sharwood @ssharwood · 26 Feb 2017

Replying to @cyberomin

Seriously? am reporter for @theregister and would love to know more about your experience
The red–black tree song (by Sean Sandys)

I see a brand new node,
I want to paint it black.

We need a balanced tree,
we've got to paint it black.

I see a brand new node,
I want to paint it black.

No time for AVL trees,
We must paint it black.

I want to find my key in
log n time—that's all.

Rotating subtrees 'round,
sure can be a ball.

I see a brand new node,
I want to paint it black.

Can't have a lot of red nodes,
we must paint them black.

Unfortunately, coding them
can be a $!#%.

If we had half a brain,
to splay trees we would switch.

And if they're still confusing,
you should have no fear.

Because outside this class,
of them you'll never hear.

I wanna see it,
painted, painted black.

Black is nice.

I wanna see the nodes painted black.
Black is nice.

I wanna see 'em
painted, painted, painted, painted black.

Mm mm mm mm mm mm mm.
Mm mm mm mm mm-mm.
Mm mm mm mm mm mm mm.
Mm mm mm mm mm-mmm.