3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs
- B-trees (see book or videos)

https://algs4.cs.princeton.edu
# 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</td>
</tr>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</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>$\log n$</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 and left-leaning red–black BSTs.
3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs
- B-trees
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 key(s) in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H
2–3 tree: insertion

Insertion into a 2-node at bottom.

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

\[
\begin{array}{c}
\text{insert } G \\
\end{array}
\]
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
2–3 tree construction demo
What is the maximum height of a 2–3 tree with $n$ 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 root to null link has same length.

Tree height.

- 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 nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. 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 ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>log $n$</td>
<td>$n$</td>
<td>$n$</td>
<td>log $n$</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>log $n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2–3 tree</td>
<td>(\log n)</td>
<td>(\log n)</td>
<td>(\log n)</td>
<td>(\log n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

but hidden constant $c$ is large (depends 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.

```java
public void put(Key key, Value val) {
    Node x = root;
    while (x.getTheCorrectChild(key) != null) {
        x = x.getTheCorrectChild(key);
        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
- B-trees
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 two 2-nodes.
- Can’t (uniquely) map from BST back to 2–3 tree.

**Approach 2.** Regular BST with red “glue” nodes.
- Wastes space for extra node.
- Messy code.

**Approach 3.** Regular BST with red “glue” links.
- Widely used in practice.
- Arbitrary restriction: 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.

![Diagram showing 2-3 tree and corresponding red-black BST with left-leaning red links as glue for 3-nodes.](image-url)
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)

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.

Symmetric order

Color invariants

“Perfect black balance”
Which LLRB tree corresponds to the following 2–3 tree?

A. 
```
    J
   / |
E   Z
 /   |
A   H  L
```
B. 
```
    L
   / |
G   Z
 /   |
E   J
 /   |
A   H
```
C. Both A and B.
D. Neither A nor B.
Search implementation for red–black BSTs

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

```
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 if (cmp == 0) return x.val;
    }
    return null;
}
```

**Remark.** Many other ops (floor, iteration, rank, 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;
    boolean color; // color of parent link
}

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)
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:

- 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.

Invariants. Maintains symmetric order and perfect black balance.
**Left rotation.** Orient a (temporarily) right-leaning red link to lean left.

![Diagram showing left rotation](image)

```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;
}
```

**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

**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.

![Flip colors diagram]

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

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

- Do standard BST insert.  \(\leftarrow\) to preserve symmetric order
- Color new link red.  \(\leftarrow\) to preserve perfect black balance
- Repeat up the tree until color invariants restored:
  - two left red links in a row?  \(\Rightarrow\) rotate right
  - left and right links both red?  \(\Rightarrow\) color flip
  - right link only red?  \(\Rightarrow\) rotate left
Insertion into a LLRB tree

- Do standard BST insert.
- Color new link red.
- Repeat up the tree until color invariants restored:
  - two left red links in a row? \(\Rightarrow\) rotate right
  - left and right links both red? \(\Rightarrow\) color flip
  - right link only red? \(\Rightarrow\) rotate left
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:
  - right link only red? ⇒ rotate left
  - two left red links in a row? ⇒ rotate right
  - left and right links both red? ⇒ 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 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 provides near-perfect balance
Insertion into a LLRB tree: visualization

n = 255
height = 9
average depth = 6.3

255 insertions in random order
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

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 refined arguments show height \( \leq 2 \log_2 n \). ]
### ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guaranteesearch</th>
<th>Guaranteinsert</th>
<th>Guarantee delete</th>
<th>Average casesearch</th>
<th>Guaranteinsert</th>
<th>Guarante delete</th>
<th>Ordered ops?</th>
<th>Key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>✔</td>
<td>equals()</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\sqrt{n}$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>2–3 tree</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>✔</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Hidden constant $c$ is small (at most $2 \log_2 n$ compares)
Why named red–black BSTs?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- 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
Balanced 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, linux/rbtree.h.

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

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

- Windows: NTFS.
- Mac OS X: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, 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.

19 Retweets 324 Likes

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.

2 Retweets 22 Likes

Simon Sharwood @ssharwood · 26 Feb 2017
Replying to @cyberomin
seriously? am reporter for @theregister and would love to know more about your experience

2 Retweets 22 Likes

https://twitter.com/cyberomin/status/835888786462625792
Red–black BST validation. Given a binary tree (with keys and color in nodes), does it represent a red–black BST?

- Symmetric order?
- Color invariants?
- Perfect black balance?
War story 3: red–black BSTs

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