3.3 BALANCED SEARCH TREES

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

Symbol table / dictionary / map is a fundamental data type

Naive implementations (arrays/linked lists) are way too slow.

(Binary) search trees work well in the average case, but can grow too tall (imbalanced) in the worst case

How to balance search trees?

<table>
<thead>
<tr>
<th>application</th>
<th>purpose of search</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary</td>
<td>find definition</td>
<td>word</td>
<td>definition</td>
</tr>
<tr>
<td>book index</td>
<td>find relevant pages</td>
<td>term</td>
<td>list of page numbers</td>
</tr>
<tr>
<td>file share</td>
<td>find song to download</td>
<td>name of song</td>
<td>computer ID</td>
</tr>
<tr>
<td>financial account</td>
<td>process transactions</td>
<td>account number</td>
<td>transaction details</td>
</tr>
<tr>
<td>web search</td>
<td>find relevant web pages</td>
<td>keyword</td>
<td>list of page names</td>
</tr>
<tr>
<td>compiler</td>
<td>find properties of variables</td>
<td>variable name</td>
<td>type and value</td>
</tr>
<tr>
<td>routing table</td>
<td>route Internet packets</td>
<td>destination</td>
<td>best route</td>
</tr>
<tr>
<td>DNS</td>
<td>find IP address</td>
<td>domain name</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find domain name</td>
<td>IP address</td>
<td>domain name</td>
</tr>
<tr>
<td>genomics</td>
<td>find markers</td>
<td>DNA string</td>
<td>known positions</td>
</tr>
<tr>
<td>file system</td>
<td>find file on disk</td>
<td>filename</td>
<td>location on disk</td>
</tr>
</tbody>
</table>
## Symbol table review

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th></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>
<td>insert</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log n</td>
<td>n</td>
<td>n</td>
<td>log n</td>
<td>n</td>
</tr>
<tr>
<td>BST</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>log 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>
<td>log n</td>
</tr>
</tbody>
</table>

**Challenge.** Guarantee performance.

**This lecture.** 2–3 trees and left-leaning red–black BSTs.

optimized for teaching and coding; introduced to the world in this course!

co-invented by Bob Sedgewick
3.3 Balanced Search Trees

- 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

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

search for H
Search.

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

search for H

H is less than M
(go left)
2–3 tree demo: search

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

search for H

H is between E and J (go middle)
2–3 tree demo: search

Search.

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

search for H

![2–3 tree diagram](image-url)

found H (search hit)
Search.

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

**2–3 tree demo: search**

**search for B**

- B is less than M (go left)
Search.

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

**search for B**
2-3 tree demo: search

Search.

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

search for B

B is between A and C (go middle)
Search.

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

**search for B**

```
  M
 /   \
E     J
|     |
A     C
H     L
 P     SX
```

link is null
(search miss)
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**
2–3 tree construction demo

insert S
2–3 tree construction demo

2–3 tree
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.
- 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.
## 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>2–3 tree</td>
<td>$\log n$</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).
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.
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: 1–1 correspondence with 2–3 trees

Key property. 1–1 correspondence between 2–3 and LLRB.
Which LLRB tree corresponds to the following 2–3 tree?

A. 

B. 

C. Both A and B. 

D. Neither A nor B.
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.
Which one of the following is a red–black BST?

A. 
```
1 4 2 6 7 9 8 3
```

B. 
```
1 5 2 6 7 3
```

C. 
```
1 7 4 2 6 9 8 3
```

D. 
```
1 8 2 4 6 9 7 3
```
Search implementation for red–black BSTs

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

```java
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;
}
```

null links are black
Review: the road to Left Leaning Red Black Trees

BSTs
Can get imbalanced

Imagine 3-nodes held together by internal glue links shown in red

2-3 trees
Balanced but cumbersome

How we draw LLRB trees

How we represent LLRB trees in code
Plan for rest of this lecture

**LLRB search.** Same as BST search; see above.

**LLRB insert.** Rest of this lecture.

**LLRB delete.** Tricky; see book.

**LLRB operations.**
- Insert requires operations called rotations and color flips.
- Derived via 1-1 correspondence with 2-3 tree operations (temporarily creating and splitting a 4-node)

**Learning strategy.**
We’ll omit the correspondence to 2-3 trees in the rest of the lecture and learn the LLRB operations directly.
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]

Examples of 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 invariant: apply rotations and color flips.
Elementary red–black BST operations

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

![Diagram of left rotation](image)

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

**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.

**Exercise.** Verify that left rotation 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.

![Diagram of right rotation]

```java
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.

```
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.

Invariants. Maintains symmetric order and perfect black balance.
Insertion into a LLRB tree

- Do standard BST insert; color new link red.
- Repeat until color invariants restored:
  - Both children red? Flip colors
  - Right link red? Rotate left
  - Two left reds in a row? Rotate right
Insertion into a LLRB tree: passing red links up the tree

- Do standard BST insert; color new link red.
- Repeat until color invariants restored:
  - Both children red?  Flip colors
  - Right link red?  Rotate left
  - Two left reds in a row?  Rotate right
Red–black BST construction demo

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

Can distill down to three cases!

- Right child red; left child black: rotate left.
- Left child red; 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 provides near-perfect balance.
255 insertions in ascending order
Insertion into a LLRB tree: visualization

\[ N = 255 \]
\[ \text{max} = 8 \]
\[ \text{avg} = 7.0 \]
\[ \text{opt} = 7.0 \]

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

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

255 random insertions
What is the maximum height of a LLRB tree with $n$ keys?

A. $\sim \log_3 n$

B. $\sim \log_2 n$

C. $\sim 2 \log_2 n$

D. $\sim n$
Balance in LLRB trees

**Proposition.** Height of tree is $\leq 2 \lg n$ in the worst case.

**Pf.**
- Black height = height of corresponding 2–3 tree $\leq \lg n$.
- Never two red links in-a-row.
# ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>equals()</td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</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>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 $2 \log n$ compares)
Why “red-black”?

Xerox PARC innovations. [1970s]

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

Historical context: Guibas & Sedgewick 1978

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 embed 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

*Supported by an NSF grant.
Left-leaning Red-Black Trees

Robert Sedgewick
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract
The red-black tree model for implementing balanced search trees, introduced by Guibas and Sedgewick thirty years ago, is now found throughout our computational infrastructure. Red-black trees are described in standard textbooks and are the underlying data structure for symbol-table implementations within C++, Java, Python, BSD Unix, and many other modern systems. However, many of these implementations have sacrificed some of the original design goals (primarily in order to develop an effective implementation of the delete operation, which was incompletely specified in the original paper), so a new look is worthwhile. In this paper, we describe a new variant of red-black trees that meets many of the original design goals and leads to substantially simpler code for insert/delete, less than one-fourth as much code as in implementations in common use.
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.
- Emacs: conservative stack scanning.

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

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

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS, BTRFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.
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.
- 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 \( \lg n \).” — expert witness