3.3 Balanced Search Trees

- 2–3 search trees
- red–black BSTs
- B-trees (see book or videos)
## 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 (unordered list)</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>
</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>

**Challenge.** Guarantee performance.

**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
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 diagram:
- 3-node: E, J
  - Smaller than E: A, C
  - Between E and J: H
  - Larger than J: L
- 2-node: M, R
  - 2-node: M
  - 2-node: R
  - Null link: P, S, X
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.
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
insert $S$
2–3 tree construction demo

2–3 tree

```
      L
     /|
    / \
   E   R
 /   /|
A   H   P
   /     \
S   X
```

2–3 tree
2–3 tree: global properties

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

**Pf.** Each transformation maintains symmetric order and perfect balance.

---

**root**

```
  a b c
```

---

**parent is a 2-node**

**left**

```
  a b c
  |    |
  d    
```

---

**middle**

```
  a b c
  a   c
b   d
```

---

**right**

```
  a
  b    d
  c    d
```

---

**parent is a 3-node**

**left**

```
  a b c
d e   
```

---

**middle**

```
  a e
  b c d
```

---

**right**

```
  a b d
c e   
```

---
Splitting a 4-node is a **local** transformation: constant number of operations.
Balanced search trees: quiz 1

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: \( \log_3 n \approx .631 \lg n \). [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 (unordered list)</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>
</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.

```
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);
}
```

Fantasy code

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 a 2-node.
   • Cannot map from BST back to 2–3 tree.

Approach 2. Regular BST with red “glue” nodes.
   • Wastes space for extra node.
   • Code probably messy.

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.
Left-leaning red–black BSTs: 1–1 correspondence with 2–3 trees

Key property. 1–1 correspondence between 2–3 and LLRB.
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"
Which is a red–black BST?

A.  
```
      5
     / 
    3   8
   /   / 
  2   6   7
 /     /   
1     4    9
```

B.  
```
      4
     / 
    3   7
   /   / 
  2   6   9
```

C.  
```
      5
     / 
    3   8
   /   / 
  2   6   9
```

D.  
```
      5
     / 
    3   7
   /   / 
  2   4   9
   
```

22
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) ⇒ 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
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 ]

To restore color invariant: apply elementary ops (rotations and color flips).
Elementary red–black BST operations

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

![Diagram of a tree with elements E, S, and x, showing a left rotation.]

```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

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.

![Diagram of rotate S right (before)](image)

```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

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

![Right rotation diagram]

```
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.

<table>
<thead>
<tr>
<th><img src="https://via.placeholder.com/150" alt="Diagram" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>private void flipColors(Node h)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>assert !isRed(h);</td>
</tr>
<tr>
<td>assert isRed(h.left);</td>
</tr>
<tr>
<td>assert isRed(h.right);</td>
</tr>
<tr>
<td>h.color = RED;</td>
</tr>
<tr>
<td>h.left.color = BLACK;</td>
</tr>
<tr>
<td>h.right.color = BLACK;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Invariants. Maintains symmetric order and perfect black balance.
Elementary red–black BST operations

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

```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.
Insertion into a LLRB tree

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

[left side diagram]

- search ends at this null link
- red link to new node containing a converts 2-node to 3-node

[right side diagram]

- search ends at this null link
- attached new node with red link
- rotated left to make a legal 3-node
Insertion into 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.

Add new node here

Right link red so rotate left

to maintain symmetric order and perfect black balance

to restore color invariants
Warmup 2. Insert into a tree with exactly 2 nodes.

**larger**

```
  a
  b
  c
```

- search ends at this null link
- attached new node with red link
- colors flipped to black

**smaller**

```
  a
  b
  c
```

- search ends at this null link
- attached new node with red link
- rotated right
- colors flipped to black

**between**

```
  a
  b
  c
```

- search ends at this null link
- attached new node with red link
- rotated left
- rotated right
- colors flipped to black
**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).

To maintain symmetric order and perfect black balance.
To restore color invariants.

**Inserting H**

[Diagram showing insertion process with nodes labeled A, C, E, H, R, S]
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

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
Insertion into a LLRB tree: visualization

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

N = 255
max = 8
avg = 7.0
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.

**Empirical observation.** Height of tree is $\sim 1.0 \lg n$ in typical applications.
### 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 hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</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>
</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>
<tr>
<td>red–black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

Hidden constant $c$ is small (at most $2 \log n$ compares)
Threaded set. Implement the following API:

```java
public class ThreadedSet

    ThreadedSet() create an empty threaded set

    void add(String s) add the string to the set
    (if it is not already in the set)

    boolean contains(String s) is the string s in the set?

    String previousKey(String s) the string added to the set immediately before s
    (null if s is the first string added or s not in set)
```

Performance requirement. $\log n$ time per operation (worst case).
War story: why red–black?

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
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
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.
Red–black BSTs in the wild

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