3.3 **Balanced Search Trees**

- 2–3 search trees
- red–black BSTs
### BST: ordered symbol table operations summary

<table>
<thead>
<tr>
<th></th>
<th>sequential search</th>
<th>binary search</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>insert</td>
<td>( N )</td>
<td>( N )</td>
<td>( h )</td>
</tr>
<tr>
<td>min / max</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( h )</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>rank</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>select</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( h )</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>( N \log N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
</tbody>
</table>

- \( h \) = height of BST
- \( N \) is proportional to \( \log N \) if keys inserted in random order

Order of growth of running time of ordered symbol table operations
Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.

**Bottom line.** Tree shape depends on order of insertion.
Symbol table review

### Table

<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>goal</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>

**Challenge.** Guarantee performance.

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

- 2–3 search trees
- red–black BSTs
Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

**Symmetric order.** In order traversal yields keys in ascending order.

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

2–3 tree
Search.

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

**search for H**
2–3 tree demo: search

Search.

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

search for H

H is less than M (go left)

- E J
  - A C
  - H
  - L
- R
  - P
  - S X
2–3 tree demo: search

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

search for H

H is between E and J (go middle)
Search.

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

2–3 tree demo: search

search for H

found H (search hit)
2–3 tree demo: search

Search.

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

search for B

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

Search.

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

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

search for B

B is between A and C (go middle)
2–3 tree demo: search

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

**search for B**

![Diagram of 2-3 tree with search for B]
Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.
2–3 tree demo: insertion

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K

K is greater than J (go right)
2–3 tree demo: insertion

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

insert K

search ends here
2–3 tree demo: insertion

Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.

\[\text{insert } K\]
2–3 tree demo: insertion

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K
2–3 tree demo: insertion

Insert 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.
2–3 tree demo: insertion

Insert 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.

insert Z
2–3 tree demo: insertion

Insert 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.

insert Z

search ends here
Insert 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.

**2–3 tree demo: insertion**

**Insert Z**

.replace 3-node with temporary 4-node containing Z
2–3 tree demo: insertion

Insert 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.

insert Z
2–3 tree demo: insertion

Insert 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.

*insert Z*

split 4-node into two 2-nodes
(pass middle key to parent)
2–3 tree demo: insertion

Insert 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.

insert Z
2–3 tree demo: insertion

Insert 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.

insert Z
Insert 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.
2–3 tree demo: insertion

Insert 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 L
2–3 tree demo: insertion

Insert 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 L

split 4-node (move L to parent)
2–3 tree demo: insertion

Insert 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 L
2–3 tree demo: insertion

Insert 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 L
```

Diagram:

```
split 4-node
(move L to parent)
```

```
A C  H  P  S X
```

```
E L R
```
2–3 tree demo: insertion

Insert 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.

height of tree increases by 1

insert L
2–3 tree demo: insertion

Insert 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 L
2–3 tree construction demo

insert S
2–3 tree demo: construction
2–3 tree demo: construction

insert E

classify S

classify E

classify E

classify E

convert 2-node into 3-node
2–3 tree demo: construction

insert E
2–3 tree demo: construction

2–3 tree
2–3 tree demo: construction

insert A

convert 3-node into 4-node
2–3 tree demo: construction

insert A
2–3 tree demo:  construction

insert A

split 4-node
(move E to parent)
2–3 tree demo: construction

insert A

```
    E
   /|
  A  S
```

2–3 tree demo: construction
2–3 tree demo: construction

insert R

convert 2-node into 3-node
2–3 tree demo: construction

insert R
2–3 tree demo: construction

2–3 tree

```
        E
       / \
      A   RS
```
2–3 tree demo: construction

insert C

classifyFXSnodeFintoFSnode

convert 2-node into 3-node
insert C
2–3 tree demo: construction

2–3 tree
2–3 tree demo: construction

insert H

convert 3-node into 4-node
2–3 tree demo: construction

insert H
2–3 tree demo: construction

insert H

split 4-node
(move R to parent)
insert H
2–3 tree demo: construction

2–3 tree
2-3 tree demo: construction

insert X

convert 2-node into 3-node
2–3 tree demo: construction

insert X
2–3 tree demo: construction

2–3 tree
insert M

convert 2-node into 3-node
2–3 tree demo: construction

insert M
2–3 tree demo: construction

2–3 tree

![2-3 tree diagram]

- E R
- A C
- H M
- S X
2–3 tree demo: construction

insert P

convert 3-node into 4-node
2–3 tree demo: construction

insert P
insert P

split 4-node
(move L to parent)
2–3 tree demo: construction

insert P
2–3 tree demo: construction

insert P

split 4-node (move M to parent)
2–3 tree demo: construction

insert P
2–3 tree demo: construction

2–3 tree

```
   M
  / | \
 E  R
 /   \
A C   H P   S X
```
2–3 tree demo: construction

insert L

convert 2-node into 3-node
2–3 tree demo: construction

insert L

convert 2-node into 3-node
2–3 tree demo: construction
2–3 tree: global properties

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

**Pf.** Each transformation maintains symmetric order and perfect balance.
Splitting a 4-node is a **local** transformation: constant number of operations.
Perfect balance. Every path from root to null link has same length.

Tree height.
- Worst case: \( \log N \) [all 2-nodes]
- Best case: \( \log_3 N \approx 0.631 \log N \) [all 3-nodes]
- The tree will have a height of between 12 and 20 for a million nodes.
- The tree will have a height of 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>search</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>equals()</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>hit</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td><strong>sequential search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td>equals()</td>
</tr>
<tr>
<td><strong>binary search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓</td>
</tr>
<tr>
<td><strong>BST</strong></td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\sqrt{N}$</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>✓</td>
</tr>
<tr>
<td><strong>2–3 tree</strong></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</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.

fantasy code

```java
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
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, wasted link.
- 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.

- **Encoding a 3-node with two 2-nodes connected by a left-leaning red link**

```
a  b
less than a  between a and b  greater than b
```

- **Correspondence between red-black and 2-3 trees**
  - Red links "glue" nodes within a 3-node
  - Black links connect 2-nodes and 3-nodes

```
E  J  M
A  C  H  L  P  S  X
```

```
E  J  M
A  C  H  L  P  S  X
```

```
 corresponding red–black BST
```

"larger key is root"
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"
**Search implementation for red–black BSTs**

**Observation.** Search is the same as for elementary BST (ignore color), but runs faster because of better balance.

```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.** 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) ⇒ 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 ]

![Diagram](image)

**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.

\[
\text{rotate E left (before)}
\]

```
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](rotation.png)

\[
\text{rotate } S \text{ right (before)}
\]

```
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.

![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.

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

Insertion into a LLRB tree

- **Left Case**: Insert into a single 2-node (two cases).
  - Search ends at this null link.
  - Red link to new node containing a converts 2-node to 3-node.

- **Right Case**: Search ends at this null link.
  - Attached new node with red link.
  - Rotated left to make a legal 3-node.

**Diagram:**
- **Left Side**: Root node with two children, one red link.
- **Right Side**: Root node with one child, red link.
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.

To maintain symmetric order and perfect black balance.
To fix color invariants.
Insertion into a LLRB tree

Warmup 2. Insert into a tree with exactly 2 nodes.
Insertion into 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).

To maintain symmetric order and perfect black balance.
To fix color invariants.
Insertion into 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

insert S
Red-black BST construction demo

insert E
Red-black BST construction demo

insert A
Red-black BST construction demo

insert A

two left reds in a row
(rotate S right)
Red-black BST construction demo

Both children red
(flip colors)
both children red
(flip colors)
Red-black BST construction demo

red–black BST
Red-black BST construction demo

red–black BST

```
    E
   / \
  A   S
```
insert R
Red-black BST construction demo

red-black BST
Red-black BST construction demo

red–black BST

```
A  E
  |
  R
  |
  S
```
Red-black BST construction demo

insert C
Red-black BST construction demo

right link red
(rotate A left)
Red-black BST construction demo

red–black BST
red–black BST
Red-black BST construction demo

red–black BST

A

C

E

S

R
Red-black BST construction demo

insert $H$
Red-black BST construction demo

two left reds in a row (rotate S right)
Red-black BST construction demo

Both children red (flip colors)
Red-black BST construction demo

right link red
(rotate E left)
Red-black BST construction demo

red–black BST

![Red-black BST construction diagram](image_url)
Red-black BST construction demo

red–black BST

A

C

E

H

R

S
Red-black BST construction demo

deck of cards

deck of cards

deck of cards

deck of cards

deck of cards

deck of cards

deck of cards

deck of cards

Red-black BST construction demo

insert X
insert X

right link red (rotate S left)
red–black BST
Red-black BST construction demo

red–black BST

```
    R
   / \
  E   X
 / \ / \  
C  H S  
/ \ /   /
A  H  S  
```
Red-black BST construction demo

red–black BST

```
       R
      / 
     E   X
    /   /
   C    H
  /     /
 A     S
```

Red-black BST construction demo

insert M
Red-black BST construction demo

insert M

right link red (rotate H left)
Red-black BST construction demo

red–black BST

Graph of a red-black BST:
- Root: R
- Branches: E, X
- Subtrees:
  - E: C, M
  - X: S
  - C: A
  - M: H
  - S: None
Red-black BST construction demo

insert P

two red children (flip colors)
Red-black BST construction demo

right link red
(rotate E left)
Red-black BST construction demo

two left reds in a row
(rotate R right)
Red-black BST construction demo

two red children
(flip colors)
Red-black BST construction demo

two red children
(flip colors)
Red-black BST construction demo

red–black BST
Red-black BST construction demo

red–black BST

![Red-black BST Diagram]
Red-black BST construction demo

red-black BST
Insertion into 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;
}
```

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

N = 255
max = 8
avg = 7.0
opt = 7.0

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
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.

**Property.** 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</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>
<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).
Red–black BSTs in the wild

Common sense. Sixth sense. Together they're the FBI's newest team.
Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS 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
Budget 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 simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?
War story:  why red–black?

Xerox PARC innovations.  [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- 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
Brno 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 $2 \log N$. ” — expert witness
File system model

**Page.** Contiguous block of data (e.g., a 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.
B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2–3 trees by allowing up to $M$ keys per node.
- At least $\lceil M/2 \rceil$ keys in all nodes (except root).
- Every path from root to leaf has same number of links.

Choose $M$ as large as possible so that $M$ keys fit in a page
($M = 1,024$ is typical)
Search in a B-tree

- Start at root.
- Check if node contains key.
- Otherwise, find interval for search key and take corresponding link.

could use binary search (but all ops are considered free)

A C D F _ _
I J K L O _
Q R T _ _ _
V W X Y Z _

a B-tree (M = 6)
Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with $M + 1$ keys on the way back up the B-tree (moving middle key to parent).

![B-tree diagram](image-url)
Balance in B-tree

**Proposition.** A search or an insertion in a B-tree of order $M$ with $N$ keys requires between $\sim \log_M N$ and $\sim \log_{M/2} N$ probes.

**Pf.** All nodes (except possibly root) have between $\lfloor M/2 \rfloor$ and $M$ keys.

**In practice.** Number of probes is at most 4. 

$M = 1024$; $N = 62$ billion

$\log_{M/2} N \leq 4$
What of the following does the B in B-tree not mean?

A. Bayer
B. Balanced
C. Binary
D. Boeing
E. I don't know.

“the more you think about what the B in B-trees could mean, the more you learn about B-trees and that is good.”
– Rudolph Bayer
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.

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

**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.