Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

### Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

---

### Text

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after $N$ inserts)</th>
<th>average case (after $N$ random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N  N  N</td>
<td>N/2  N  N/2  no</td>
<td>equals()</td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$lg , N$</td>
<td>$lg , N$</td>
<td>$lg , N$</td>
<td>$lg , N$</td>
</tr>
<tr>
<td>BST</td>
<td>N  N  N</td>
<td>$1.39 , lg , N$</td>
<td>$1.39 , lg , N$</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>
</tbody>
</table>

- **Challenge.** Guarantee performance.
2-3 tree

Allow 1 or 2 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.

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

Symmetric order. Inorder traversal yields keys in ascending order.

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

**search for H**

![Diagram showing search for H]

**search for B**

![Diagram showing search for B]
2-3 tree demo

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)

search for B

link is null
(search miss)

2-3 tree demo

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

insert K

K is less than M
(go left)

insert K

K is greater than J
(go right)
2-3 tree demo

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

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

2-3 tree demo

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 K

replace 2-node with 3-node containing K

insert Z

Z is greater than M (go right)
2-3 tree demo

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

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

2-3 tree demo

2-3 tree demo
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.

split 4-node into two 2-nodes
(pass middle key to parent)

insert Z

insert Z

insert L

convert 3-node into 4-node
2-3 tree demo

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

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

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

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

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

Search in a 2-3 tree

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

Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.
**Insertion in a 2-3 tree**

**Case 2.** 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.

**Local transformations in a 2-3 tree**

Splitting a 4-node is a **local** transformation: constant number of operations.

**Global properties in a 2-3 tree**

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

Guaranteed logarithmic performance for search and insert.

---

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</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/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>N</td>
<td>N</td>
<td>$\lg N$</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.39 $\lg N$</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>$c \lg N$</td>
<td>$c \lg N$</td>
<td>$c \lg N$</td>
<td>$c \lg N$</td>
</tr>
</tbody>
</table>

Constants depend 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.
Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

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.

Key property. 1–1 correspondence between 2–3 and LLRB.
Search implementation for red-black BSTs

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

public Val 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., ceiling, selection, iteration) are also identical.

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.

Red-black BST representation

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

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

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

rotate S right (before)

greater than S

less than E between E and S

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;
}

rotate S right (after)

greater than S

less than E between E and S

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

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

flip colors (before)

less than A between A and E between E and S greater than S

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;
}

flip colors (after)

less than A between A and E between E and S greater than S

Invariants. Maintains symmetric order and perfect black balance.
**Insertion in a LLRB tree: overview**

**Basic strategy.** Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.

**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.

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

**Warmup 2.** Insert into a tree with exactly 2 nodes.
**Insertion in a LLRB tree**

**Case 2.** Insert into a 3-node at the bottom.
- Do standard BST insert; color new link red.
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

**Red-black BST insertion**

**Insertion in 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.
- 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 insertion

insert A

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

Red-black BST insertion
both children red
(flip colors)

Red-black BST insertion
both children red
(flip colors)
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

Red-black BST

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST

Red-black BST insertion

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

red-black BST

\[
\begin{array}{c}
E \\
C \\
S \\
A \\
R \\
\end{array}
\]

Red-black BST insertion

red-black BST

\[
\begin{array}{c}
E \\
C \\
S \\
A \\
R \\
\end{array}
\]

Red-black BST insertion

insert H

\[
\begin{array}{c}
E \\
C \\
S \\
A \\
R \\
H \\
\end{array}
\]

Red-black BST insertion

two left reds in a row (rotate S right)

\[
\begin{array}{c}
E \\
C \\
S \\
A \\
R \\
H \\
\end{array}
\]
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

red-black BST

Red-black BST insertion

insert X

right link red (rotate S left)
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

Red-black BST insertion

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

- **Insert P**

  - Insertion point: P
  - Parent: E
  - Left child: M
  - Right child: X
  - Two red children (flip colors)

Red-black BST insertion

- **Right link red (rotate E left)**

  - Right child: R
  - Parent: E
  - Left child: M
  - Right child: S

Red-black BST insertion

- **Two left reds in a row (rotate R right)**

  - Left child: R
  - Right child: X
  - Parent: E
  - Left child: P

Red-black BST insertion

- **Two red children (flip colors)**

  - Left child: M
  - Right child: R
  - Parent: E
  - Left child: C
  - Right child: H
  - Parent: A
Red-black BST insertion

two red children
(flip colors)

red-black BST

Red-black BST insertion

red-black BST

Red-black BST insertion

red-black BST
Red-black BST insertion

insert L

Red-black BST insertion

right link red (rotate H left)

Red-black BST insertion

red-black BST

LLRB tree insertion trace

Standard indexing client.

insert S

red-black BST
corresponding 2-3 tree
Standard indexing client (continued).

**Insertion in a LLRB tree**: Java implementation

Same code for both 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;
}
```

**Remark.** Only a few extra lines of code to standard BST insert.
Insertion in a LLRB tree: visualization

Remark. Only a few extra lines of code to standard BST insert.

\[ \begin{align*}
N & = 255 \\
\text{max} & = 10 \\
\text{avg} & = 7.3 \\
\text{opt} & = 7.0
\end{align*} \]

255 random insertions

Balance in LLRB trees

Proposition. Height of tree is \( \leq 2 \lg N \) in the worst case.

\[ \text{Pf.} \]
\[ \begin{itemize}
\item Every path from root to null link has same number of black links.
\item Never two red links in-a-row.
\end{itemize} \]

Property. Height of tree is \( \approx 1.00 \lg N \) in typical applications.

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>worst-case cost (after N insert)</th>
<th>average case (after N random inserts)</th>
<th>ordered insertion?</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>N2</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>\lg N</td>
<td>N</td>
<td>N</td>
<td>\lg 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>1.39 \lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-3 tree</td>
<td>c \lg N</td>
<td>c \lg N</td>
<td>c \lg N</td>
<td>c \lg N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>red-black BST</td>
<td>2 \lg N</td>
<td>2 \lg N</td>
<td>2 \lg N</td>
<td>1.00 \lg N *</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\* exact value of coefficient unknown but extremely close to 1

War story: why red-black?

Xerox PARC innovations. [1970s]

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

![Xerox Alto](image)
Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to \( M - 1 \) key-link pairs per node.
- At least 2 key-link pairs at root.
- At least \( M / 2 \) key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

Properties of B-trees

- At least 2 key-link pairs at root.
- At least \( M / 2 \) key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

File system model

Page. Contiguous block of data (e.g., a file or 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.

Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.
Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with \( M \) key-link pairs on the way up the tree.

Proposition. A search or an insertion in a B-tree of order \( M \) with \( N \) keys requires between \( \log_{M-1} N \) and \( \log_{M/2} N \) probes.

Pf. All internal nodes (besides root) have between \( M/2 \) and \( M-1 \) links.

In practice. Number of probes is at most 4.

Optimization. Always keep root page in memory.

Building a large B tree

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.

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

B-trees (and variants) are widely used for file systems and databases.
- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.
**Balanced Search Trees**

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs

**Geometric applications of BSTs**

- kd trees

---

**2-d orthogonal range search**

Extension of ordered symbol-table to 2d keys.
- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- **Range search**: find all keys that lie in a 2d range.
- **Range count**: number of keys that lie in a 2d range.

**Geometric interpretation.**
- Keys are point in the **plane**.
- Find/count points in a given **h-v** rectangle.

**Applications.** Networking, circuit design, databases,...

**2d orthogonal range search: grid implementation**

Grid implementation.
- Divide space into $M$-by-$M$ grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add $(x, y)$ to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.
2d orthogonal range search: grid implementation costs

Space-time tradeoff.
- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

Choose grid square size to tune performance.
- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: $\sqrt{N}$-by-$\sqrt{N}$ grid.

Running time. [if points are evenly distributed]
- Initialize data structure: $N$.
- Insert point: 1.
- Range search: 1 per point in range.

Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.
- Lists are too long, even though average length is short.
- Need data structure that gracefully adapts to data.

Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.
Ex. USA map data.

Space-partitioning trees

Use a tree to represent a recursive subdivision of 2d space.

Grid. Divide space uniformly into squares.
2d tree. Recursively divide space into two halfplanes.
Quadtree. Recursively divide space into four quadrants.
BSP tree. Recursively divide space into two regions.
Space-partitioning trees: applications

Applications.
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.

Kd tree

Kd tree. Recursively partition $k$-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.

Efficient, simple data structure for processing $k$-dimensional data.
- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!

N-body simulation

Goal. Simulate the motion of $N$ particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force. $F = \frac{G m_1 m_2}{r^2}$

Appel algorithm for N-body simulation

Key idea. Suppose particle is far, far away from cluster of particles.
- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and center of mass of aggregate particle.
Appel algorithm for N-body simulation

- Build 3d-tree with $N$ particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.

**Impact.** Running time per step is $N \log N$ instead of $N^2 \Rightarrow \text{enables new research.}$