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>search</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>N</td>
<td>N/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>search hit</td>
<td>N/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td></td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>delete</td>
<td>N/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>lg N</td>
<td>yes</td>
<td>compareTo()</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>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>N</td>
<td>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

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).

2-3 tree demo

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**

2-3 tree demo

```
  S
 /   \\  \
  X   A  C
 |     |
 P     H
 |     |    
 R     M
```

H is between E and J (go middle)

```
  H  E  J   R
 /   |   |
 A   H   L
```

**search for H**

2-3 tree demo

```
 M
 /   \\  \
 E  J   R
 |     |
 A   H   L
```

found H (search hit)

**search for B**

2-3 tree demo

```
  B
 /   \\  \
  M   E  J
 |     |
 A   H   L
```

B is less than M (go left)

**search for B**

2-3 tree demo

```
  B
 /   \\  \
  E   J   R
 |     |
 A   H   L
```

B is less than E (go left)
**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)

- A
- C

search for B

link is null
(search miss)

- A
- C

**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)

- A
- C
- H
- L
- P
- S

insert K

K is greater than J
(go right)

- A
- C
- H
- L
- P
- S
2-3 tree demo

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

insert K

2-3 tree demo

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

insert K

2-3 tree demo

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

insert K

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

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.

Z is greater than R (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.

search ends here

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.

replace 3-node with temporary 4-node containing 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.

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.

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.

insert L

convert 3-node into 4-node
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 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.

**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 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:
- Best case:

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 Ht</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</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>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.

"perfect black balance"

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.

Key property. 1–1 correspondence between 2–3 and LLRB.

red links "glue" nodes within a 3-node
black links connect 2-nodes and 3-nodes
larger key is root
horizontal red links
Search implementation for red-black BSTs

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

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)

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)

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

Flip colors
(before)

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)

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.
Red-black BST insertion

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).

Insertion in 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).
• Repeat case 1 or case 2 up the tree (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.
• 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 insertion

insert A

both children red (flip colors)

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
Red-black BST insertion

red-black BST

![Red-black BST insertion](image1)

Red-black BST insertion

red-black BST

![Red-black BST insertion](image2)

Red-black BST insertion

insert R

![Red-black BST insertion](image3)

Red-black BST insertion

red-black BST

![Red-black BST insertion](image4)
Red-black BST insertion

red-black BST

Red-black BST insertion

right link red
(rotate A left)

Red-black BST insertion

Red-black BST insertion
Red-black BST insertion

red-black BST

Red-black BST insertion

Red-black BST insertion

insert H

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

right link red (rotate E left)

red-black BST
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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Red-black BST

Red-black BST

Red-black BST

A

E

C

H

S

R

X

A

E

C

H

S

R

X

E

C

H

S

A

E

C

H

S

R

X

E

C

H

M

S

A

E

C

H

M

S

A
Red-black BST insertion

Insert M

Red-black BST insertion

Red-black BST

Insert P

Red-black BST insertion

Two red children (flip colors)

Insert P
Red-black BST insertion

**insert P**

```
A
H
P
```

```
C
M

E
```

```
X
```

two red children (flip colors)

Red-black BST insertion

```
A
H
P
```

```
C
M

E
```

```
X
```

right link red (rotate E left)

Red-black BST insertion

```
A
H
P
```

```
C
M

E
```

```
X
```

two left reds in a row (rotate R right)

Red-black BST insertion

```
A
H
P
```

```
C
M

E
```

```
X
```

two red children (flip colors)

Red-black BST insertion
Red-black BST insertion

two red children
(flip colors)

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.

red-black BST corresponding 2-3 tree
**LLRB tree insertion trace**

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.

255 random insertions

**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 insertion?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<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 N N N</td>
<td>N/2 N N/2 N/2 no</td>
<td>equals()</td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N N N</td>
<td>lg N N/2 N/2 yes</td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N N N</td>
<td>1.39 lg N 1.39 lg N 1</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>2-3 tree</td>
<td>c lg N c lg N c lg N</td>
<td>c lg N c lg N c lg N yes</td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td>red-black BST</td>
<td>2 lg N 2 lg N 2 lg N</td>
<td>1.00 lg N 1.00 lg N 1.00 lg N yes</td>
<td>compareTo()</td>
<td></td>
</tr>
</tbody>
</table>

* exact value of coefficient unknown but extremely close to 1

**Balance in LLRB trees**

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

**Pf.**
- Every path from root to null link has same number of black links.
- Never two red links in-a-row.

**Property.** Height of tree is $\sim 1.00 \lg N$ in typical applications.

**War story: why red-black?**

Xerox PARC innovations. [1970s]

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

Xerox Alto

A DENDRICH FRAMEWORK FOR BALANCED TREES

LSD J. Bolin

Ruck for the Research Dept.,

Palo Alto, California,

and Carnegie-Mellon University

Robert Sedgewick

Princ in Computer Science of Princeton University

unknown

To this paper we present a different framework for its implementation
and study of balanced tree algorithms. We also have to thank this
framework for Classic—which we believe to be the best.

never two red links in-a-row.
**Balanced Search Trees**

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

---

**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.

---

**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.

---

**Anatomy of a B-tree set (\( M = 6 \))**

- All nodes except the root are 3-, 4- or 5-nodes.
- Client keys (black) are in external nodes.
- Each red key is a copy of min key in subtree.
- External 3-node (full).
- Internal 3-node.
- Each red key is a copy of min key in subtree.
- External 2-node.
- External 4-node.
- External 5-node.
- All nodes except the root are 3-, 4- or 5-nodes.

---

**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.

### Balance in B-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
Each line shows the result of inserting one key in some page. The white portion of the page is unoccupied, and the black portion is occupied. A full page, about to split, is shown when a new key is added to one of them.

### 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`.

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.
Overview

This lecture. Intersections among geometric objects.

Applications. CAD, games, movies, virtual reality, VLSI design, databases, ...

Efficient solutions. Binary search trees (and extensions).

Balanced Search Trees

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

Geometric applications of BSTs

- 1D range search
- Line segment intersection
- kd trees
- Rectangle intersection
Id range search

Extension of ordered symbol table.
- Insert key-value pair.
- Search for key \( k \).
- Delete key \( k \).
- **Range search**: find all keys between \( k_1 \) and \( k_2 \).
- **Range count**: number of keys between \( k_1 \) and \( k_2 \).

Application. Database queries.

**Geometric interpretation.**
- Keys are point on a line.
- Find/count points in a given 1d interval.

```
insert B    B
insert D    B D
insert A    A B D
insert I    A B D I
insert H    A B D H I
insert F    A B D F H I
insert P    A B D F H I P
count G to K 2
search G to K H I
```

Id range search: implementations

Unordered array. Fast insert, slow range search.
Ordered array. Slow insert, binary search for \( k_1 \) and \( k_2 \) to do range search.

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>range count</th>
<th>range search</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>( I )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
<tr>
<td>ordered array</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( R + \log N )</td>
</tr>
</tbody>
</table>

**Parameters.**
- \( N \) = number of keys.
- \( R \) = number of keys that match.

Id range count: BST implementation

**Id range count.** How many keys between \( l_0 \) and \( h_1 \)?

```
public int size(Key lo, Key hi) {
    if (contains(hi)) return rank(hi) - rank(lo) + 1;
    else              return rank(hi) - rank(lo);
}
```

**Proposition.** Running time is proportional to \( \log N \) (assuming BST is balanced).

**Pf.** Nodes examined = search path to \( l_0 \) + search path to \( h_1 \).

Id range search: BST implementation

**Id range search.** Find all keys between \( l_0 \) and \( h_1 \).
- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

**Proposition.** Running time is proportional to \( R + \log N \) (assuming BST is balanced).

**Pf.** Nodes examined = search path to \( l_0 \) + search path to \( h_1 \) + matching keys.
Geometric applications of BSTs

- 1D range search
- Line segment intersection
- kd trees

Orthogonal line segment intersection search:
sweep-line algorithm

Sweep vertical line from left to right.
- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into BST.

Orthogonal line segment intersection search:
sweep-line algorithm

Sweep vertical line from left to right.
- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into BST.
- h-segment (right endpoint): remove y-coordinate from BST.

Quadratic algorithm. Check all pairs of line segments for intersection.

Nondegeneracy assumption. All x- and y-coordinates are distinct.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.
- x-coordinates define events.
- h-segment (left endpoint): insert y-coordinate into BST.
- h-segment (right endpoint): remove y-coordinate from BST.
- v-segment: range search for interval of y-endpoints.

1d range search

Orthogonal line segment intersection search: sweep-line algorithm analysis

Proposition. The sweep-line algorithm takes time proportional to \( N \log N + R \) to find all \( R \) intersections among \( N \) orthogonal line segments.

Pf.
- Put x-coordinates on a PQ (or sort).
- Insert y-coordinates into BST.
- Delete y-coordinates from BST.
- Range searches in BST.

Bottom line. Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.

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.

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.

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

13,000 points, 1000 grid squares
half the squares are empty  half the points are in 10% of the squares
**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.

![Grid](image1.png) ![2d tree](image2.png) ![Quadtree](image3.png) ![BSP tree](image4.png)

**Insertion in a 2d tree**

Recursively partition plane into two halfplanes.

![Insertion in 2d tree](image5.png)

**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.

![Applications](image6.png)

**2d tree implementation**

**Data structure.** BST, but alternate using x- and y-coordinates as key.
- Search gives rectangle containing point.
- Insert further subdivides the plane.

![2d tree implementation](image7.png)
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

![Diagram](image1)

find all points in 2d tree that are contained in green query rectangle

search root node
check if query rectangle contains point 1

query rectangle to left of splitting line
search only in left subtree

Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

![Diagram](image2)

search left subtree
check if query rectangle contains point 3
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

![Diagram showing a 2d tree with a query rectangle intersecting the splitting line.](image)

query rectangle intersects splitting line
search bottom and top subtrees

![Diagram showing a 2d tree with a query rectangle to the left of the splitting line.](image)

query rectangle to left of splitting line
search only in left subtree

Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

![Diagram showing a 2d tree with a query rectangle containing point 4.](image)

search left subtree
check if query rectangle contains point 4

(search hit)

Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

![Diagram showing a 2d tree with a query rectangle containing point 5.](image)

search left subtree
check if query rectangle contains point 5

(search hit)
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

```
search top subtree
check if query rectangle contains point 6
```

Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

```
search only in left subtree
```
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

search left subtree
stop since empty

return from function call

return from function call
Range search in a 2d tree

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

Typical case. \( R + \log N \).
Worst case (assuming tree is balanced). \( R + \sqrt{N} \).

Range search. Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/bottom subdivision (if any could fall in rectangle).
- Recursively search right/top subdivision (if any could fall in rectangle).

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

find closest points in 2d tree to green query point

search root node
compute distance from query point to 1 (update champion nearest neighbor)
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

query point is to the left of splitting line
search left subtree first

query point is above splitting line
search top subtree first

search left subtree
compute distance from query point to 3
(update champion)

search top subtree
compute distance from query point to 6
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Search bottom subtree
compute distance from query point to 4

Search left subtree
compute distance from query point to 5
(update champion)
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.
Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search right subtree
prune since nearest neighbor
can't be in subdivision

return from function call

search right subtree next

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search right subtree
prune since nearest neighbor
can't be in subdivision

return from function call

search right subtree next

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search right subtree
prune since nearest neighbor
can't be in subdivision

return from function call

search right subtree next

Nearest neighbor search in a 2d tree

- Check distance from point in node to query point.
- Recursively search left/bottom subdivision (if it could contain a closer point).
- Recursively search right/top subdivision (if it could contain a closer point).
- Organize recursive method so that it begins by searching for query point.

search right subtree
prune since nearest neighbor
can't be in subdivision

return from function call

search right subtree next

nearest neighbor = 5
Nearest neighbor search in a 2d tree

Nearest neighbor search. Given a query point, find the closest point.
• Check distance from point in node to query point.
• Recursively search left/bottom subdivision (if it could contain a closer point).
• Recursively search right/top subdivision (if it could contain a closer point).
• Organize recursive method so that it begins by searching for query point.

Typical case. \( \log N \).
Worst case (even if tree is balanced). \( N \).

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 $N^2 \Rightarrow$ enables new research.