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
<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</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>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

- **Challenge.** Guarantee performance.
Balanced Search Trees

- 2-3 search trees
- Red-black BSTs
- B-trees
- Geometric applications of BSTs
You can read it as 2 or 3 children tree
Allow 1 or 2 keys per node.
• 2-node: one key, two children.
• 3-node: two keys, three children.
2-3 tree

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

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

```
2-3 tree
```

```
                M
               /   \\
            E     R
           /   \   /   \    \\
          A   C  H   L  P  S  X

- smaller than E
- larger than J
- between E and J
```
Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).
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)
2-3 tree demo

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

search for H

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

search for B

B is less than E (go left)
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.
• Compare search key against keys in node.
• Find interval containing search key.
• Follow associated link (recursively).

search for B

link is null
(search miss)
Problem with Binary Search Tree: when the tree grows from leaves, it is possible to always insert to same branch. (worst-case)

Instead of growing the tree from bottom, try to grow upwards.
- If there is space in a leaf, simply insert it
- Otherwise push nodes from bottom to top, if done recursively the tree will be balanced as it grows (increasing the height by introducing a new root)

If we keep on inserting to same branch;

**BST:**
```
  9
 / 
8   
|   
7   
|   
6
```

**2 or 3 Tree:**
```
  8
 / 
6,7 9
```

---

15
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)
```

```
E J
```

```
A C
```

```
H
```

```
L
```

```
R
```

```
P
```

```
S X
```
Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.
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 2-node at bottom.
- Search for key, as usual.
- Replace 2-node with 3-node.
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)
```

```
AC
```

```
H
```

```
KL
```

```
P
```

```
SX
```

```
M Z
```

```
E J
```

```
R
```

```

```
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.

insert Z

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.

insert Z

replace 3-node with temporary 4-node containing 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.

`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.
**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.

**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.
• 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 of 2-3 tree demo](image-url)
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)
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

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
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
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).

**Successful search for H**

1. H is less than M so look to the left
2. H is between E and L so look in the middle
3. found H so return value (search hit)

**Unsuccessful search for B**

1. B is less than M so look to the left
2. B is less than E so look to the left
3. B is between A and C so look in the middle
4. link is null so B is not in the tree (search miss)
Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.
**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.
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.
- If you reach the root and it's a 4-node, split it into three 2-nodes.
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.
Perfect balance. Every path from root to null link has same length.

Tree height.
- Worst case:
- Best case:
Perfect balance. Every path from root to null link has same length.

Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

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
Multiple Node Types

- In 2-3 Trees, the algorithm automatically balances the tree.
- However, we have to keep track of two different node types, complicating the source code.
  - Nodes with one key
  - Nodes with two keys

- Instead of multiple nodes:
  - Multiple edge types; red and black
  - Rotations instead of Split
1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.
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.
  - We will only allow one red link to simulate 2 keys in node
  - A node with two red links would be the same as having 3 keys
• Red links lean left (correct ordering)
**Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees**

**Key property.** 1–1 correspondence between 2–3 and LLRB.
Observation. Search is the same as for elementary BST (ignore color).

but runs faster because of better balance

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;
            else if (cmp == 0) return x.val;
    }
    return null;
}

Remark. Most other ops (e.g., ceiling, selection, iteration) 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

h.left.color is RED

h.right.color is BLACK
**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

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

Invariants. Maintains symmetric order and perfect black balance.

```java
private Node rotateLeft(Node h) {
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```
Elementary red-black BST operations

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 $x$;
}

Invariants. Maintains symmetric order and perfect black balance.
Right rotation. Orient a left-leaning red link to (temporarily) lean right.

```java
private Node rotateRight(Node h) {
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

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

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

private void flipColors(Node h) {
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}

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

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

```
private void flipColors(Node h) {
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

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

Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.
Insertion in a LLRB tree

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

- Left:
  - Insertion in a LLRB tree
  - Search ends at this null link
  - Red link to new node containing a
  - Converts 2-node to 3-node

- Right:
  - Search ends at this null link
  - Attached new node with red link
  - Rotated left to make a legal 3-node
**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 2. Insert into a tree with exactly 2 nodes.

Insertion in a LLRB tree

Think of this as a split in 2-3 tree
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).
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 S
Red-black BST insertion

insert E
Red-black BST insertion

insert A
Red-black BST insertion

insert A

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

both children red
(flip colors)

---

Diagram:
- Node E is red.
- Node A is red.
- Node S is red.

The diagram shows a three-node red-black tree with the root E, and A and S as its children, both having red colors.

Nodes A and S are children of E, and both A and S are red, indicating they are violating the red-black tree property for red children.
Red-black BST insertion

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

red–black BST
Red-black BST insertion

red–black BST
Red-black BST insertion

insert R
Red-black BST insertion
Red-black BST insertion

red–black BST
Red-black BST insertion

insert C
Red-black BST insertion

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

red-black BST

```
  E
 /   \\  
C     S
 / \   /   \   
A   R B     C
```

78
Red-black BST insertion

red–black BST

```
  E
 /   \
C     S
 /     /
A     R
```
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 insertion

red-black BST
Red-black BST insertion

red-black BST
Red-black BST insertion

red-black BST
Red-black BST insertion

insert X
Red-black BST insertion

insert X

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

red–black BST
Red-black BST insertion

red–black BST

A

C

E

H

R

S

X
Red-black BST insertion

red-black BST
Red-black BST insertion

insert M
Red-black BST insertion

insert M

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

red-black BST
Red-black BST insertion

insert P
Red-black BST insertion

insert P

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

insert P

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

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

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

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

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

red–black BST

![Red-black BST Insertion Diagram]
Red-black BST insertion

red–black BST

![Red-black BST insertion diagram]
Red-black BST insertion

red–black BST

![Red-black BST diagram](image-url)
Red-black BST insertion

insert L
Red-black BST insertion

insert L

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

red–black BST

```
   M
  /   \
 E     R
 / \   / \ \
C   L  P   X
|   |   |   |
A   H  S   
```
LLRB tree insertion trace

Standard indexing client.

![Tree insertion trace diagram]
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;
}
```

insert at bottom (and color red)
lean left
balance 4-node
split 4-node
only a few extra lines of code provides near-perfect balance
Insertion in a LLRB tree: visualization

255 insertions in ascending order
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
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.
ST implementations: frequency counter

Costs for java FrequencyCounter 8 < tale.txt using BST

Costs for java FrequencyCounter 8 < tale.txt using RedBlackBST
## 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>
<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>
</tbody>
</table>

* exact value of coefficient unknown but extremely close to 1
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-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.

Choose $M$ as large as possible so that $M$ links fit in a page, e.g., $M = 1024$.
• Start at root.
• Find interval for search key and take corresponding link.
• Search terminates in external node.

Searching in a B-tree

(searching for E)

follow this link because
E is between * and K

follow this link because
E is between D and H

search for E in
this external node

 searching in a B-tree set (M = 6)
**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.

![Inserting a new key into a B-tree set](image)
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.

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.
- White: unoccupied portion of page.
- Black: occupied portion of page.
- Full page, about to split.
- Full page splits into two half-full pages then 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.
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,...
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.
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.
Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.
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.

**Quadtrees.** Recursively divide space into four quadrants.

**BSP tree.** Recursively divide space into two regions.
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.
Idea. Recursively divide space into 4 quadrants.

Implementation. 4-way tree (actually a trie).

Benefit. Good performance in the presence of clustering.

Drawback. Arbitrary depth!

```java
public class QuadTree
{
    private Quad quad;
    private Value val;
    private QuadTree NW, NE, SW, SE;
}
```
Quadtree: larger example

Curse of dimensionality

k-d range search. Orthogonal range search in $k$-dimensions.
Main application. Multi-dimensional databases.

3d space. Octrees: recursively subdivide 3d space into 8 octants.
100d space. Centrees: recursively subdivide 100d space into $2^{100}$ centres?
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!
**Goal.** Simulate the motion of $N$ particles, mutually affected by gravity.

**Brute force.** For each pair of particles, compute force.  

$$F = \frac{G m_1 m_2}{r^2}$$

http://www.youtube.com/watch?v=ua7YiN4eL_w
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.
• 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.