Today’s Outline

- Multidimensional search trees
- Range Queries
- k-D Trees
- Quad Trees

Multi-D Search ADT

- Dictionary operations
  - create
  - destroy
  - find
  - insert
  - delete
  - range queries
- Each item has \( k \) keys for a \( k \)-dimensional search tree
- Searches can be performed on one, some, or all the keys or on ranges of the keys

Applications of Multi-D Search

- Astronomy (simulation of galaxies) - 3 dimensions
- Protein folding in molecular biology - 3 dimensions
- Lossy data compression - 4 to 64 dimensions
- Image processing - 2 dimensions
- Graphics - 2 or 3 dimensions
- Animation - 3 to 4 dimensions
- Geographical databases - 2 or 3 dimensions
- Web searching - 200 or more dimensions

Range Query

A range query is a search in a dictionary in which the exact key may not be entirely specified.

Range queries are the primary interface with multi-D data structures.

Range Query Examples:

Two Dimensions

- Search for items based on just one key
- Search for items based on ranges for all keys
- Search for items based on a function of several keys: e.g., a circular range query
Range Querying in 1-D
Find everything in the rectangle...

Range Querying in 1-D with a BST
Find everything in the rectangle...

1-D Range Querying in 2-D

2-D Range Querying in 2-D

\( k \)-D Trees
- Split on the next dimension at each succeeding level
- If building in batch, choose the median along the current dimension at each level
  - guarantees logarithmic height and balanced tree
- In general, add as in a BST

\( k \)-D tree node
- keys [value]
- dimension
- left
- right

The dimension that this node splits on

Find in a \( k \)-D Tree
\( \text{find}(<x_1, x_2, \ldots, x_k>, \text{root}) \) finds the node which has the given set of keys in it or returns null if there is no such node

- \( \text{Node} * \text{find}(<\text{keyVector}, \text{keys}, \text{root}) \{ \)
- \( \text{int dim} = \text{root}->\text{dimension}; \)
- \( \text{if} (\text{root} == \text{NULL}) \)
  - \( \text{return root}; \)
- \( \text{else if} (\text{root}->\text{keys} == \text{keys}) \)
  - \( \text{return root}; \)
- \( \text{else if} (\text{keys}[\text{dim}] < \text{root}->\text{keys}[\text{dim}]) \)
  - \( \text{return find}(<\text{keys, root}->\text{left}); \)
- \( \text{else} \)
  - \( \text{return find}(<\text{keys, root}->\text{right}); \)
- \} \)

runtime:
Find Example

Building a 2-D Tree (1/4)

Building a 2-D Tree (2/4)

Building a 2-D Tree (3/4)

Building a 2-D Tree (4/4)

k-D Tree
2-D Range Querying in 2-D Trees

Search every partition that intersects the rectangle.
Check whether each node (including leaves) falls into the range.

Range Query in a 2-D Tree

```
print_range(int xlow, xhigh, ylow, yhigh, Node * root) {
  if (root == NULL) return;
  if (xlow <= root.x && root.x <= xhigh &&
      ylow <= root.y && root.y <= yhigh) {
    print(root);
    if (root.dim == "x")
      print_range(root.left);
    if (root.dim == "y")
      print_range(root.right);
  }
}
```

runtime: O(depth of tree)

Range Query in a k-D Tree

```
print_range(int low[MAXD], high[MAXD], Mode * root) {
  if (root == NULL) return;
  inrange = true;
  for (i=0; i[MAXD]; i++)
    if (root.coord[i] < low[i]) inrange = false;
    if (high[i] < root.coord[i]) inrange = false;
  if (inrange) print(root);
  if (low[root.dim] <= root.coord[root.dim])
    print_range(root.left);
  if (root.coord[root.dim] <= high[root.dim])
    print_range(root.right);
}
```

runtime: O(N)

Other Shapes for Range Querying

Search every partition that intersects the shape (circle).
Check whether each node (including leaves) falls into the shape.

k-D Trees Can Be Inefficient

(But not when built in batch!)

```
insert(<5,0>)
insert(<6,9>)
insert(<9,3>)
insert(<6,5>)
insert(<7,7>)
insert(<8,6>)
```
suck factor: O(n)

k-D Trees Can Be Inefficient

(But not when built in batch!)

```
insert(<5,0>)
insert(<6,9>)
insert(<9,3>)
insert(<6,5>)
insert(<7,7>)
insert(<8,6>)
```
suck factor: O(n)
Quad Trees

- Split on all (two) dimensions at each level
- Split key space into equal size partitions (quadrants)
- Add a new node by adding to a leaf, and, if the leaf is already occupied, split until only one node per leaf

<table>
<thead>
<tr>
<th>quadrant</th>
<th>quad tree node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,1</td>
<td></td>
</tr>
<tr>
<td>0,0,0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>keys</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Find Example

\( \text{find(<10,2>, i.e., c)} \)
\( \text{find(<5,6>, i.e., d)} \)

Find in a Quad Tree

\( \text{find(<x, y>, root)} \) finds the node which has the given pair of keys in it or returns quadrant where the point should be if there is no such node

```java
Node *i find(Key x, Key y, Node *i root) {
    if (root == NULL)
        return NULL; // Empty tree
    if (root->isLeaf)
        return root; // Key may not actually be here
    int quad = getQuadrant(x, y, root);
    return find(x, y, root->quadrants[quad]);
}
```

runtime: \(O(\text{depth})\)

Building a Quad Tree (1/5)

Building a Quad Tree (2/5)

Building a Quad Tree (3/5)
Building a Quad Tree (4/5)

Building a Quad Tree (5/5)

Quad Tree Example

Quad Trees Can Suck

Quad Trees Can Suck

2-D Range Querying in Quad Trees
2-D Range Query in a Quad Tree

```c
int print_range(int xlow, int xhigh, int ylow, int yhigh, Node *root)
{
    if (root == NULL) return;
    if (xlow <= root.x && xhigh >= root.x && ylow <= root.y && yhigh >= root.y)
        print(root);
    if (xlow < root.x && xhigh >= root.x && ylow <= root.y)
        print_range(root.lower_left);
    if (xlow <= root.x && xhigh >= root.x && ylow > root.y)
        print_range(root.upper_left);
    if (root.x < xhigh && ylow <= root.y)
        print_range(root.lower_right);
    if (root.x < xhigh && yhigh >= root.y)
        print_range(root.upper_right);
}
```

Runtime: O(N)

Insert Example

```c
insert(<10, 7>, x)
```

- Find the spot where the node should go.
- If the space is unoccupied, insert the node.
- If it is occupied, split until the existing node separates from the new one.

Delete Example

```c
delete(<10, 2>) (i.e., c)
```

• Find and delete the node.
• If its parent has just one child, delete it.
• Propagate!

Nearest Neighbor Search

```c
getNearestNeighbor(<1>, 4>)
```

• Find a nearby node (do a find).
• Do a circular range query.
• As you get results, tighten the circle.
• Continue until no closer node in query.

Quad Trees vs. k-D Trees

- k-D Trees
  - Density balanced trees
  - Number of nodes is \( O(n) \) where \( n \) is the number of points
  - Height of the tree is \( O(\log n) \) with batch insertion
  - Supports insert, find, nearest neighbor, range queries
- Quad Trees
  - Number of nodes is \( O(n(1 + \log(\Delta/n))) \) where \( n \) is the number of points and \( \Delta \) is the ratio of the width (or height) of the key space and the smallest distance between two points
  - Height of the tree is \( O(\log n + \log \Delta) \)
  - Supports insert, delete, find, nearest neighbor, range queries

To Do / Coming Attractions

- Read (a little) about k-D trees in Weiss 12.6
- (More) Heaps ’o fun
  - leftist heaps & binomial queues