Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
Multi-Dimensional Data

- Various applications
  - Attributes of a product (laptop: price, screen size, processor speed, RAM, hard drive,···)
  - Attributes of an employee (name, age, salary,···)

- Dictionary for multi-dimensional data
  A collection of $d$-dimensional items
  Each item has $d$ aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
  Operations: insert, delete, range-search query

- (Orthogonal) Range-search query: specify a range (interval) for certain aspects, and find all the items whose aspects fall within given ranges.
  Example: laptops with screen size between 11 and 13 inches, RAM between 8 and 16 GB, price between 1,500 and 2,000 CAD
Multi-Dimensional Data Example

- Each item has $d$ aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
- Aspect values ($x_i$) are numbers
- Each item corresponds to a point in $d$-dimensional space
- We concentrate on $d = 2$, i.e., points in Euclidean plane

```plaintext
<table>
<thead>
<tr>
<th>Price (CAD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 1300 1400 1500 1600 1700 1800</td>
</tr>
<tr>
<td>Processor Speed (MHz)</td>
</tr>
<tr>
<td>600 800 1000 1200</td>
</tr>
</tbody>
</table>
```

Range-search query: $(1350 \leq x \leq 1550, 700 \leq y \leq 1100)$

Item: ordered pair $(x, y) \in \mathbb{R} \times \mathbb{R}$
2-Dimensional Range Search

Options for implementing $d$-dimensional dictionaries:

- Reduce to one-dimensional dictionary:
  combine the $d$-dimensional key into one key
  Problem: Range search on one aspect is not straightforward

- Use several dictionaries: one for each dimension
  Problem: inefficient, wastes space

- Partition trees
  - A tree with $n$ leaves, each leaf corresponds to an item
  - Each internal node corresponds to a region
    - quadtrees, kd-trees

- multi-dimensional range trees
  - A binary search tree for one dimension
  - Each node has an associated binary search tree for the other dimension
Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
Quadtrees

We have \( n \) points \( S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \) in the plane.

**Assume:** All points are within a square \( R \).
- Can find \( R \) by computing minimum and maximum \( x \) and \( y \) values in \( S \)
- Ideally the width/height of \( R \) is a power of 2

How to build the quadtree on \( S \):
- Root \( r \) of the quadtree corresponds to \( R \)
- If \( R \) contains 0 or 1 points, then root \( r \) is a leaf that stores point.
- Else split: Partition \( R \) into four equal subsquares (quadrants) \( R_{NE}, R_{NW}, R_{SW}, R_{SE} \)
- Root has four subtrees \( T_{NE}, T_{NW}, T_{SW}, T_{SE} \); \( T_i \) is associated with \( R_i \)
- Recursively repeat this process at each subtree.

**Convention:** Points on split lines belong to right/top side
- We could delete leaves without point (but then need edge labels)
Quadtrees example

Easier for humans: omit empty subtrees, label edges

$[0, 16] \times [0, 16]$
Easier for humans: omit empty sub-trees, label edges

$$\{(0, 16) \times (0, 16)\}$$
Quadtree example

Easier for humans: omit empty subtrees, label edges

\[ (0, 16) \times (0, 16) \]

\[ (0, 8) \times [0, 16) \]

\[ (0, 8) \times (0, 8) \]
Easier for humans: omit empty sub-trees, label edges

\[(0, 16) \times [0, 16)\]
Quadtrees example

Easier for humans: omit empty subtrees, label edges
Quadtree Dictionary Operations

- **Search**: Analogous to binary search trees and tries
- **Insert**:
  - Search for the point
  - Split the leaf while there are two points in one region
- **Delete**:
  - Search for the point
  - Remove the point
  - If its parent has only one point left: delete parent
    (and recursively all ancestors that have only one point left)
Quadtree Range Search

\[ \text{QTree-RangeSearch}(T, A) \]

\( T \): The root of a quadtree, \( A \): Query rectangle

1. \( \text{let } R \text{ be the square associated with } T \)
2. \( \text{if } (R \subseteq A) \text{ then} \)
3. \( \quad \text{report all points in } T; \text{ return} \)
4. \( \quad \text{if } (R \cap A \text{ is empty}) \text{ then} \)
5. \( \quad \text{return} \)
6. \( \quad \text{if } (T \text{ stores a single point } p) \text{ then} \)
7. \( \quad \quad \text{if } p \text{ is in } A \text{ return } p \)
8. \( \quad \quad \text{else return} \)
9. \( \quad \text{for each child } v \text{ of } T \text{ do} \)
10. \( \quad \quad \text{QTree-RangeSearch}(v, A) \)

Note: We assume here that each node of the quadtree stores the associated square. Alternatively, these could be re-computed during the search (space-time tradeoff).
Quadtree range search example

Blue: Search stopped due to $R \cap A = \emptyset$. Green: Must continue search in children / evaluate.
Quadtree range search example

Blue: Search stopped due to $R \cap A = \emptyset$. Green: Must continue search in children / evaluate.
Crucial for analysis: what is the height of a quadtree?
  ▶ Can have very large height for bad distributions of points
  ▶ spread factor of points $S$: $\beta(S) = \frac{\text{sidelength of } R}{d_{\text{min}}}$
  ▶ $d_{\text{min}}$: minimum distance between two points in $S$
  ▶ height of quadtree: $h \in \Theta(\log \beta(S))$

- Complexity to build initial tree: $\Theta(nh)$ worst-case
- Complexity of range search: $\Theta(nh)$ worst-case even if the answer is $\emptyset$
- But in practice much faster.
Quadtree of 1-dimensional points:

```
“Points:” 0 9 12 14 24 26 28
```

Quadtrees in other dimensions

Quadtrees also easily generalize to higher dimensions (octrees, etc.) but are rarely used beyond dimension 3.
Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

<table>
<thead>
<tr>
<th>“Points:”</th>
<th>0</th>
<th>9</th>
<th>12</th>
<th>14</th>
<th>24</th>
<th>26</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>(in base-2)</td>
<td>0000</td>
<td>0100</td>
<td>0110</td>
<td>0111</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
</tr>
</tbody>
</table>

Same as a trie (with splitting stopped once key is unique)

Quadtrees also easily generalize to higher dimensions (octrees, etc.) but are rarely used beyond dimension 3.
Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

```
“Points:”  0    9    12   14    24    26    28
(in base-2) 00000 01001 01100 01110 11000 11010 11100
```

Same as a trie (with splitting stopped once key is unique)
Quadtrees in other dimensions

- **Quad-tree of 1-dimensional points:**

<table>
<thead>
<tr>
<th>Points:</th>
<th>0</th>
<th>9</th>
<th>12</th>
<th>14</th>
<th>24</th>
<th>26</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000</td>
<td>01001</td>
<td>01100</td>
<td>01110</td>
<td>11000</td>
<td>11010</td>
<td>11100</td>
</tr>
<tr>
<td>(in base-2)</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>24</td>
<td>26</td>
<td>28</td>
</tr>
</tbody>
</table>

```
(0,32) __________ 1
  |          |
(0,16)  0  [0,16)  [0,32)
```

- Quadtrees also easily generalize to higher dimensions (octrees, etc.) but are rarely used beyond dimension 3.
Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

  “Points:”  0  9  12  14  24  26  28
  (in base-2) 00000 01001 01100 01110 11000 11010 11100

  Same as a trie (with splitting stopped once key is unique)
Quadtree of 1-dimensional points:

```
“Points:”
0  9  12  14
```
```
(0,32)
```
```
0
```
```
[0,16)
```
```
01001
```
```
[0,32)
```
```
0
```
```
[16,32)
```
```
1
```
```
[24,32)
```
```
1
```

Same as a trie (with splitting stopped once key is unique)

Quadtrees also easily generalize to higher dimensions (octrees, etc.) but are rarely used beyond dimension 3.
Quadtree summary

- Very easy to compute and handle
- No complicated arithmetic, only divisions by 2 (bit-shift!) if the width/height of $R$ is a power of 2
- Space potentially wasteful, but good if points are well-distributed
- Variation: We could stop splitting earlier and allow up to $S$ points in a leaf (for some fixed bound $S$).
- Variation: Store pixelated images by splitting until each region has the same color.
Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
We have \( n \) points \( S = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\} \).

Quadtrees split square into quadrants regardless of where points are.

(Point-based) kd-tree idea: Split the region such that (roughly) half the point are in each subtree.

Each node of the kd-tree keeps track of a *splitting line* in one dimension (2D: either vertical or horizontal).

**Convention:** Points on split lines belong to right/top side.

Continue splitting, switching between vertical and horizontal lines, until every point is in a separate region.

(There are alternatives, e.g., split by the dimension that has better aspect ratios for the resulting regions. No details.)
kd-tree example
kd-tree example
kd-tree example
kd-tree example
kd-tree example
Constructing kd-trees

Build kd-tree with initial split by $x$ on points $S$:

- If $|S| \leq 1$ create a leaf and return.
- Else $X := \text{Quick-Select}(S, \lfloor n/2 \rfloor)$ (select by $x$-coordinate)
- Partition $S$ by $x$-coordinate into $S_{x<X}$ and $S_{x\geq X}$
- Create left subtree recursively (splitting by $y$) for points $S_{x<X}$.
- Create right subtree recursively (splitting by $y$) for points $S_{x\geq X}$.

Building with initial $y$-split symmetric.

Analysis:

- Find $X$ and partition $S$ in $\Theta(n)$ expected time.
- $\Theta(n)$ expected time on each level in the tree
- Total is $\Theta(\text{height} \cdot n)$ expected time
- This can be reduced to $\Theta(n \log n + \text{height} \cdot n)$ \textit{worst-case} time by pre-sorting (no details).
kd-tree height

Assume first that the points are in *general position* (no two points have the same x-coordinate or y-coordinate).

- Then the split always puts $\left\lfloor \frac{n}{2} \right\rfloor$ points on one side and $\left\lceil \frac{n}{2} \right\rceil$ points on the other.
- So height $h(n)$ satisfies the recursion $h(n) \leq h(\left\lceil \frac{n}{2} \right\rceil) + 1$.
- This resolves to $h(n) \leq \lceil \log(n) \rceil$.
- So can build the $kd$-tree in $\Theta(n \log n)$ time and $O(n)$ space.
kd-tree height

Assume first that the points are in *general position* (no two points have the same x-coordinate or y-coordinate).

- Then the split always puts $\lfloor \frac{n}{2} \rfloor$ points on one side and $\lceil \frac{n}{2} \rceil$ points on the other.
- So height $h(n)$ satisfies the recursion $h(n) \leq h(\lceil \frac{n}{2} \rceil) + 1$.
- This resolves to $h(n) \leq \lceil \log(n) \rceil$.
- So can build the $kd$-tree in $\Theta(n \log n)$ time and $O(n)$ space.

If points share coordinates, then height can be infinite!

This could be remedied by modifying the splitting routine. (No details.)
kd-tree Dictionary Operations

- **Search** (for single point): as in binary search tree using indicated coordinate
- **Insert**: search, insert as new leaf.
- **Delete**: search, remove leaf and unary parents.

**Problem:** After insert or delete, the split might no longer be at exact median and the height is no longer guaranteed to be $O(\log n)$ even for points in general position.

This can be remedied by allowing a certain imbalance and re-building the entire tree when it becomes too unbalanced. (No details.)
kd-tree Range Search

- Note: every node is again associated with a region.
- If not stored explicitly this can be computed during a search.
- Rest of range search is very similar to the one for quad-trees.

\[
kdTree-RangeSearch(T, R, A)
\]

\(T\): The root of a kd-tree, \(R\): region associated with \(T\), \(A\): query rectangle

1. \textbf{if} \((R \subseteq A)\) \textbf{then} report all points in \(T\); return
2. \textbf{if} \((R \cap A\) is empty) \textbf{then} return
3. \textbf{if} \((T\) stores a single point \(p\)) \textbf{then}
   4. \textbf{if} \(p\) is in \(A\) return \(p\)
   5. \textbf{else} return
4. \textbf{if} \(T\) stores split “is \(x < X\)?”
5. \(R_\ell \leftarrow R \cap \{(x, y) : x < X\}\)
6. \(R_r \leftarrow R \cap \{(x, y) : x \geq X\}\)
7. \(kdTree-RangeSearch(T.\text{left}, R_\ell, A)\)
8. \(kdTree-RangeSearch(T.\text{right}, R_r, A)\)
9. \(\text{else} \ // \ root \ node \ splits \ by \ y\)-coordinate
10. \textbf{else} // symmetric
kd-tree: Range Search Example

Blue: Search stopped due to $R \cap A = \emptyset$.
Pink: Search stopped due to $R \subseteq A$. 

Biedl, Petrick, Veksler (SCS, UW)
kd-tree: Range Search Example

Blue: Search stopped due to $R \cap A = \emptyset$. Pink: Search stopped due to $R \subseteq A$. 
kd-tree: Range Search Complexity

- The complexity is $O(s + Q(n))$ where
  - $s$ is the number of keys reported (output-size)
  - $s$ can be anything from 0 to $n$.
  - No range-search can work in $o(s)$ time since it must report the points.
  - $Q(n)$ is the number of “green” nodes:
    - $kdTreeRangeSearch$ was called.
    - Neither $R \subseteq A$ nor $R \cap A = \emptyset$

- **Can show:** $Q(n)$ satisfies the following recurrence relation (no details):
  $$Q(n) \leq 2Q(n/4) + O(1)$$
  - This solves to $Q(n) \in O(\sqrt{n})$
  - Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$
kd-tree: Higher Dimensions

- **kd-trees for** $d$-**dimensional space:**
  - At the root the point set is partitioned based on the first coordinate
  - At the subtrees of the root the partition is based on the second coordinate
  - At depth $d - 1$ the partition is based on the last coordinate
  - At depth $d$ we start all over again, partitioning on first coordinate

- **Storage:** $O(n)$

- **Construction time:** $O(n \log n)$

- **Range query time:** $O(s + n^{1 - 1/d})$

This assumes that $o(n)$ points share coordinates and $d$ is a constant.
Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
Towards Range Trees

- Both Quadtrees and kd-trees are intuitive and simple.
- But: both may be very slow for range searches.
- Quadtrees are also potentially wasteful in space.

New idea: Range trees

- Somewhat wasteful in space, but much faster range search.
- Have a binary search tree $T$ (sorted by $x$-coordinate); this is the primary structure
- Each node $v$ of $T$ has an auxiliary structure $T(v)$: a binary search tree (sorted by $y$-coordinate)

Must understand first: How to do (1-dimensional) range search in binary search tree?
BST Range Search

\[ \text{BST-RangeSearch}(T, k_1, k_2) \]

* \(T\): root of a binary search tree, \(k_1, k_2\): search keys

Returns keys in \(T\) that are in range \([k_1, k_2]\)

1. if \(T = \text{null}\) then return
2. if \(k_1 \leq \text{key}(T) \leq k_2\) then
3. \(L \leftarrow \text{BST-RangeSearch}(T.\text{left}, k_1, k_2)\)
4. \(R \leftarrow \text{BST-RangeSearch}(T.\text{right}, k_1, k_2)\)
5. return \(L \cup \{\text{key}(T)\} \cup R\)
6. if \(\text{key}(T) < k_1\) then
7. return \(\text{BST-RangeSearch}(T.\text{right}, k_1, k_2)\)
8. if \(\text{key}(T) > k_2\) then
9. return \(\text{BST-RangeSearch}(T.\text{left}, k_1, k_2)\)

Note: Keys are reported in in-order, i.e., in sorted order.
BST Range Search example

\[\text{BST-RangeSearch}(T, 28, 45)\]
BST Range Search example

$BST\text{-}RangeSearch(T, 28, 45)$

Note: Search from 39 was unnecessary: all its descendants are in range.
BST Range Search example

\textbf{BST-RangeSearch}( T, 28, 45)

Note: Search from 39 was unnecessary: all its descendants are in range.
BST Range Search example

\[ \text{BST-RangeSearch}(T, 28, 45) \]
BST Range Search example

\textit{BST-RangeSearch}(T, 28, 45)

Note: Search from 39 was unnecessary: \textbf{all} its descendants are in range.
BST Range Search re-phrased

- Search for left boundary $k_1$: this gives path $P_1$
- Search for right boundary $k_2$: this gives path $P_2$
- Partition nodes of $T$ into three groups:
  - boundary nodes: nodes in $P_1$ or $P_2$
  - inside nodes: nodes that are right of $P_1$ and left of $P_2$
  - outside nodes: nodes that are left of $P_1$ or right of $P_2$
- Report all inside nodes
- Test each boundary node and report it if it is in range
BST Range Search analysis

Assume that the binary search tree is balanced:
- Search for path $P_1$: $O(\log n)$
- Search for path $P_2$: $O(\log n)$
- $O(\log n)$ boundary nodes
- But could have many inside nodes.

We only need the topmost of them: allocation node $v$ not in $P_1$ or $P_2$, but parent is in $P_1$ or $P_2$ (but not both)
- if parent is in $P_1$, then $v$ is right child
- if parent is in $P_2$, then $v$ is left child

$O(\log n)$ allocation nodes. For each of them report all descendants.
- This is no faster overall, but allocation nodes will be important for 2d.

As before, test each boundary node and report it if it is in range

Run-time:

$O(\text{# boundary nodes} + \text{# reported points}) = O(\log n + s)$
BST Range Search analysis

Assume that the binary search tree is balanced:

- Search for path $P_1$: $O(\log n)$
- Search for path $P_2$: $O(\log n)$
- $O(\log n)$ boundary nodes
- But could have many inside nodes.

- We only need the topmost of them: allocation node $v$
  - not in $P_1$ or $P_2$, but parent is in $P_1$ or $P_2$ (but not both)
  - if parent is in $P_1$, then $v$ is right child
  - if parent is in $P_2$, then $v$ is left child

- $O(\log n)$ allocation nodes. For each of them report all descendants.
  - This is no faster overall, but allocation nodes will be important for 2d.

- As before, test each boundary node and report it if it is in range

- Run-time: $O(\# \text{ boundary nodes} + \# \text{ reported points}) = O(\log n + s)$
BST Range Search summary

- Balanced binary search supports ranges queries in $O(\log n + s)$ time.
  - $\log n$-term comes from the height of the tree
  - $s$ is the output-size as before
- Variants of range-searching: Only report whether there are items in the range, or the number of such items.
  - Balanced binary search trees support both in $O(\log n)$ time.
- We could have achieved the same result with a sorted array:
  - Binary search for $k_1$, binary search for $k_2$
  - Report all keys between the returned indices
- But range search in BST is a key ingredient for search in higher dimension.
2-dimensional Range Trees

- We have $n$ points $P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$
- A range tree is a tree of trees (a multi-level data structure)
- **Primary structure**: Binary search tree $T$ that stores $P$ and uses $x$-coordinates as keys.
  We assume $T$ is balanced, so it has height $O(\log n)$.
- Each node $v$ of $T$ stores an auxiliary structure $T(v)$:
  - Let $P(v)$ be all points in subtree of $v$ in $T$ (including point at $v$)
  - $T(v)$ stores $P(v)$ in a binary search tree, using the $y$-coordinates as key
  - Note: $v$ is not necessarily the root of $T(v)$
Range Tree Structure

$T$: binary search tree on $x$-coordinate

$P(v)$: points in subtree of $v$ (including point at $v$)

$T(v)$: binary search tree on $y$-coordinate of all points on $P(v)$
Range Tree Space Analysis

- Primary tree uses $O(n)$ space.
- Associate tree $T(v)$ uses $O(|P(v)|)$ space (where $P(v)$ are the points at descendants of $v$ in $T$)
- **Key insight:** $w \in P(v)$ means that $v$ is an ancestor of $w$ in $T$
  - Every node has $O(\log n)$ ancestors in $T$
  - Every node belongs to $O(\log n)$ sets $P(v)$
  - So $\sum_{v} |P(v)| \leq n \cdot O(\log n)$
- Range tree space usage: $O(n \log n)$
Range Trees: Dictionary Operations

- **Search**: as in a binary search tree
- **Insert**: First, insert point by $x$-coordinate into $T$. Then, walk back up to the root and insert the point by $y$-coordinate in all $T(v)$ of nodes $v$ on path to the root.
- **Delete**: analogous to insertion

**Problem**: Want binary search trees to be balanced.
- This makes Insert/Delete very slow if we use AVL-trees. (A rotation at $v$ changes $P(v)$ and hence requires a re-build of $T(v)$.)
- Instead of rotations, can do something similar as for kd-trees: Allow certain imbalance, rebuild entire subtree if violated. (No details.)
Range Trees: Range Search

- A two stage process
- To perform a range search query \( A = [x_1, x_2] \times [y_1, y_2] \):
  - Perform a range search (on the \( x \)-coordinates) for the interval \([x_1, x_2]\) in primary tree \( T \) (\( BST\text{-RangeSearch}(T, x_1, x_2) \))
  - Obtain boundary, topmost outside and allocation nodes as before.
  - For every boundary node, test to see if the corresponding point is within the region \( A \).
  - For every allocation node \( v \), perform a range search (on the \( y \)-coordinates) for the interval \([y_1, y_2]\) in \( T(v) \).
  - We know that all \( x \)-coordinates of points in \( T(v) \) are within range.
Range tree range search example
Range tree range search example

primary tree $T$

Biedl, Petrick, Veksler (SCS, UW)  
CS240 – Module 8
Winter 2019 33 / 37
Range tree range search example

- Primary tree $T$
- Range queries:
  1. $(1, 5)$
  2. $(2, 7)$
  3. $(3, 1)$
  4. $(4, 4)$
  5. $(5, 13)$
  6. $(6, 15)$
  7. $(7, 11)$
  8. $(8, 10)$
  9. $(9, 6)$
  10. $(10, 12)$
  11. $(11, 8)$
  12. $(12, 14)$
  13. $(13, 2)$
  14. $(14, 9)$
  15. $(15, 16)$
Range tree range search example
Range tree range search example
Range tree range search example
Range Trees: Query Run-time

- $O(\log n)$ time to find boundary and allocation nodes in primary tree.
- There are $O(\log n)$ allocation nodes.
- $O(\log n + s_v)$ time for each allocation node $v$, where $s_v$ is the number of points in $T(v)$ that are reported.
- Two allocation nodes have no common point in their trees.
  - $\Rightarrow$ every point is reported in at most one auxiliary structure.
  - $\Rightarrow \sum s_v \leq s$

Time for range-query in range tree: $O(s + \log^2 n)$

This can be reduced further to $O(s + \log n)$ (no details).
Range Trees: The issue of duplicates

- ADT Dictionary promises: no key $k$ appears twice
- But: primary tree might have duplicates.
  E.g. points (40, 10), (40, 20), (40, 50) give primary tree

Search in BST must find all copies (if in range)
- Search for left boundary path $P_1$: If equal, go left.
- Search for right boundary path $P_2$: If equal, go right.
Range Trees: Higher Dimensions

- Range trees can be generalized to $d$-dimensional space.

<table>
<thead>
<tr>
<th>Category</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>$O(n (\log n)^{d-1})$</td>
</tr>
<tr>
<td>Construction time</td>
<td>$O(n (\log n)^{d-1})$</td>
</tr>
<tr>
<td>Range query time</td>
<td>$O(s + (\log n)^d)$</td>
</tr>
</tbody>
</table>

(Note: $d$ is considered to be a constant.)
Range Trees: Higher Dimensions

- Range trees can be generalized to $d$-dimensional space.

<table>
<thead>
<tr>
<th>Property</th>
<th>$O(n \log n)^{d-1}$</th>
<th>kd-trees: $O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construction time</td>
<td>$O(n \log n)^{d-1}$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Range query time</td>
<td>$O(s + (\log n)^d)$</td>
<td>$O(s + n^{1-1/d})$</td>
</tr>
</tbody>
</table>

(Note: $d$ is considered to be a constant.)

- Space/time trade-off compared to kd-trees.
Outline

1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion
Comparison of range query data structures

- **Quadtrees**
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions

- **kd-trees**
  - linear space
  - query-time $O(\sqrt{n})$
  - inserts/deletes destroy balance
  - care needed for duplicate coordinates

- **range trees**
  - fastest range search $O(\log^2 n)$
  - wastes some space
  - insert and delete more complicated