CS 240 – Data Structures and Data Management

Module 8: Range-Searching in Dictionaries for Points

A. Storjohann
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2018

References: Goodrich & Tamassia 12.1, 12.3

version 2018–11–01 14:25

Outline

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

Outline

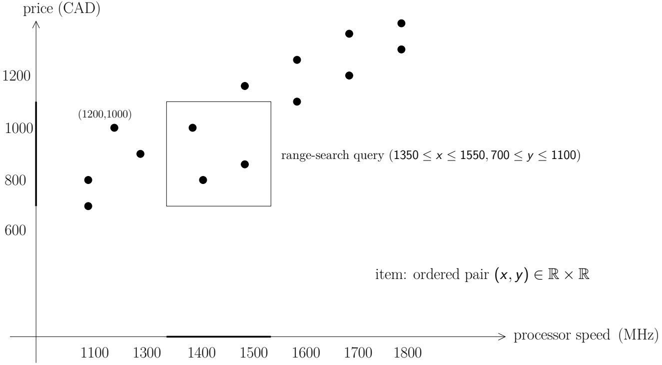
1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion

Multi-Dimensional Data

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



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 $S = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$ in the plane.

Assume: All points are within a square R.

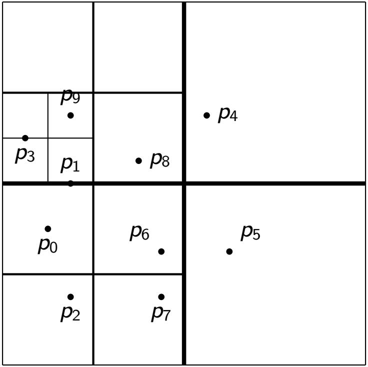
- Can find R by computing minimum and maximum x and y values in S
- Ideally the width/height of R is a power of 2

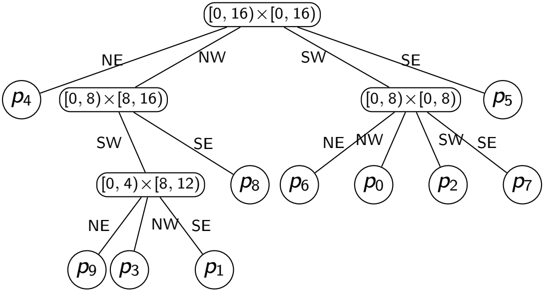
How to **build** the quadtree on S:

- 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 children $v_{NE}, v_{NW}, v_{SW}, v_{SE}$; v_i is associated with R_i
- Recursively repeat this process at each child.

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

Quadtrees example





Quadtree Dictionary Operations

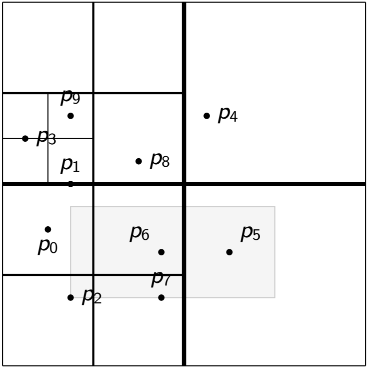
- **Search**: Analogous to binary search trees and tries
- **Insert**:
  - Search for the point
  - Split the leaf if there are two points
- **Delete**:
  - Search for the point
  - Remove the point
  - If its parent has only one child left, delete that child and continue the process toward the root.

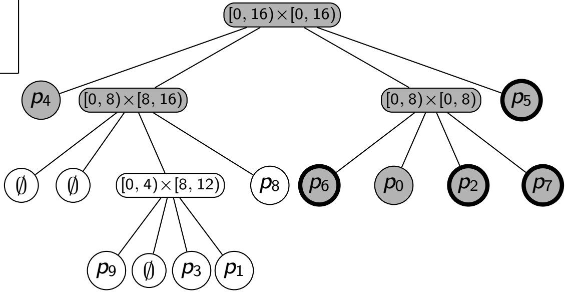
Quadtree Range Search

```
QTree-RangeSearch(T, A)
T: The root of a quadtree, A: Query rectangle
    let R be the square associated with T
    if (R ⊆ A) then
            report all points in T; return
    if (R ∩ A is empty) then
            return
    if (T stores a single point p) then
            if p is in A return p
            else return
    for each child v of T do
        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.

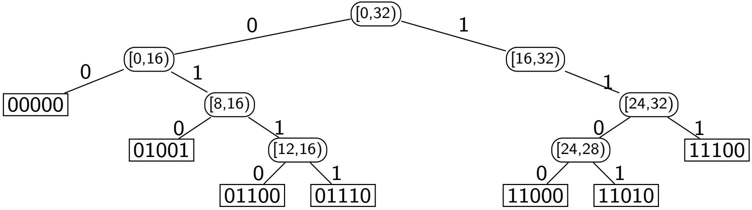
Quadtree Analysis

- 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: $\beta(S) = \dfrac{\text{sidelength of } R}{d_{min}}$
  - d_{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.

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 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.

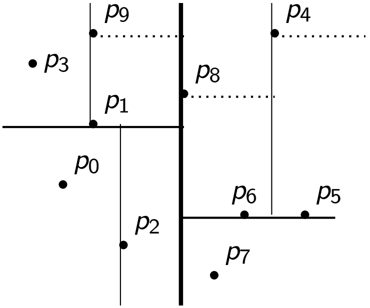
Outline

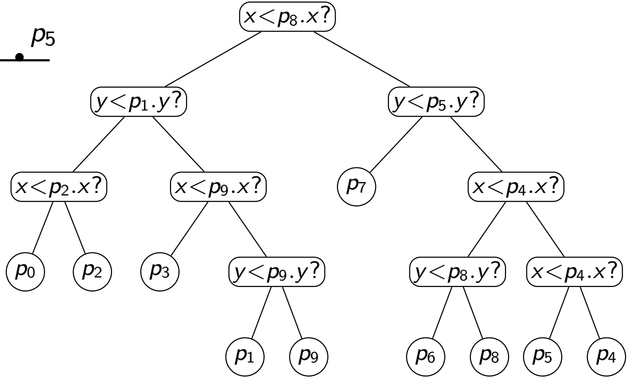
1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion

kd-trees

- 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

kd-tree example





Constructing kd-trees

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

- If $|S| \leq 1$ create a leaf and return.
- Else find $(\lfloor \frac{n}{2} \rfloor + 1)$st smallest x-coordinates X in S.
- Partition S into $S_{x<X}$ and $S_{x \geq X}$ by comparing points' x coordinate with X.
- Create left child with recursive call (splitting on y) for points $S_{x<X}$.
- Create right child with recursive call (splitting on y) for points $S_{x \geq X}$.

Building with initial y-split symmetric.

Analysis:

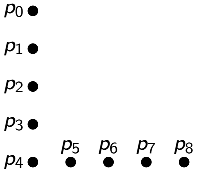
- Find median and partition in linear time.
- $\Theta(n)$ work on each level in the tree (summed over all nodes)
- Total is $\Theta(\mathit{height} \cdot n)$

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.

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 to unbalanced. (No details.)

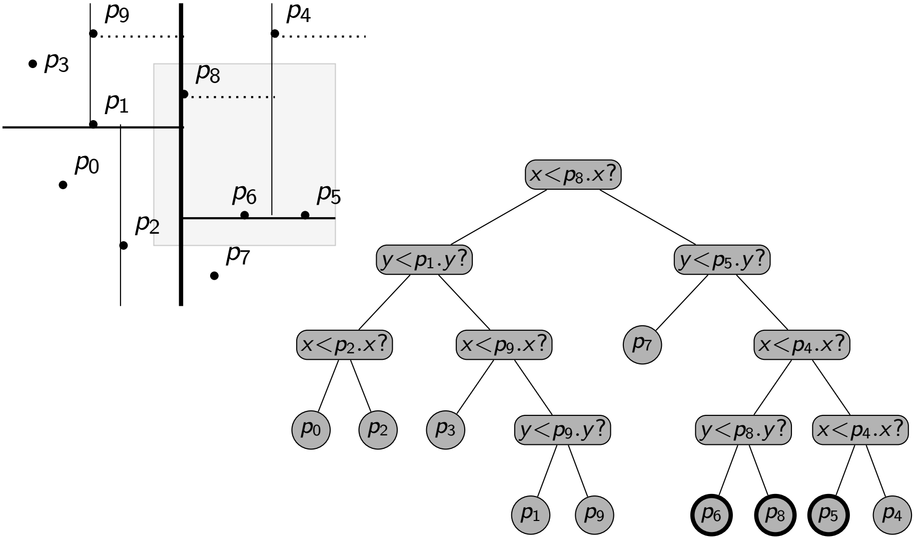
kd-tree Range Search

- 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

    if $(R \subseteq A)$ **then** report all points in T; return
    if $(R \cap A$ is empty$)$ **then** return
    if (T stores a single point p) **then**
            if p is in A return p
            else return
    if T stores split "is $x < X$"?
        $R_\ell \leftarrow R \cap \{(x, y) : x < X\}$
        $R_r \leftarrow R \cap \{(x, y) : x \geq X\}$
        kdTree-RangeSearch$(T.\mathit{left}, R_\ell, A)$
        kdTree-RangeSearch$(T.\mathit{right}, R_r, A)$
    else // root node splits by y-coordinate
        ... // symmetric

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 nodes for which *kdTreeRangeSearch* was called.
- **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 children 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

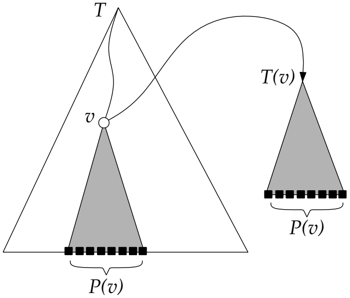
1. Range-Searching in Dictionaries for Points
   - Range Search Query
   - Quadtrees
   - kd-Trees
   - Range Trees
   - Conclusion

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 do do (1-dimensional) range search in binary search tree?

BST Range Search

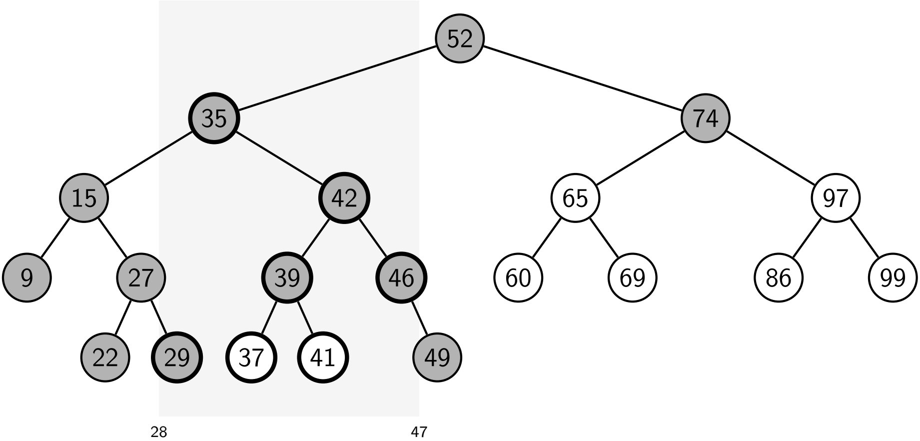
BST-RangeSearch(T, k_1, k_2)
T: root of a binary search tree, k_1, k_2: search keys
Returns keys in T that are in range $[k_1, k_2]$

if $T = null$ **then return**
if $k_1 \leq key(T) \leq k_2$ **then**
    $L \leftarrow$ BST-RangeSearch$(T.left, k_1, k_2)$
    $R \leftarrow$ BST-RangeSearch$(T.right, k_1, k_2)$
    return $L \cup \{key(T)\} \cup R$
if $key(T) < k_1$ **then**
    return BST-RangeSearch$(T.right, k_1, k_2)$
if $key(T) > k_2$ **then**
    return BST-RangeSearch$(T.left, k_1, k_2)$

Note: Keys are reported in in-order, i. e., in sorted order.

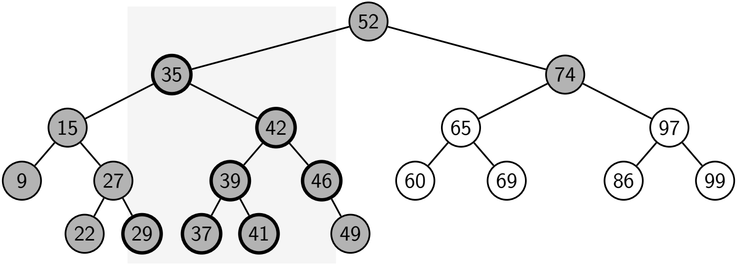
BST Range Search example

BST-RangeSearch$(T, 28, 47)$



Note: Search from 39 was unnecessary: **all** its descendants are in range.

BST Range Search re-phrased

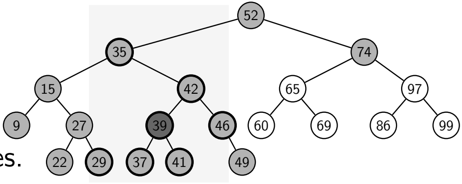


- Search for left boundary k_1: this gives path P_1
- Search for right boundary k_2: this gives path P_2
- Partition nodes of T into three groups:
  - boundary nodes: nodes in P_1 or P_2
  - inside nodes: nodes that are right of P_1 and left of P_2
  - outside nodes: nodes that are left of P_1 or right 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 (39)
  - 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

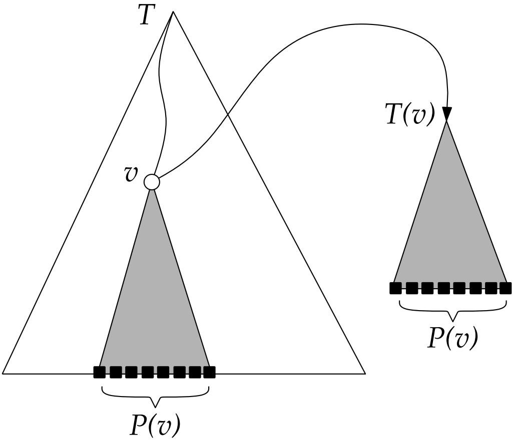
- Balanced binary search supports ranges queries in $O(\log n + s)$ time.
  - $\log n$-term comes from the height of the tree
  - s is the output-size as before
- Variants of range-searching: Only report *whether* there are items in the range, or the *number* of such items.
  - Balanced binary search 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 $P = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$
- 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.
- Each node v of T stores an **auxiliary structure** $T(v)$:
  - Let $P(v)$ be all points at descendants of v in T (including 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

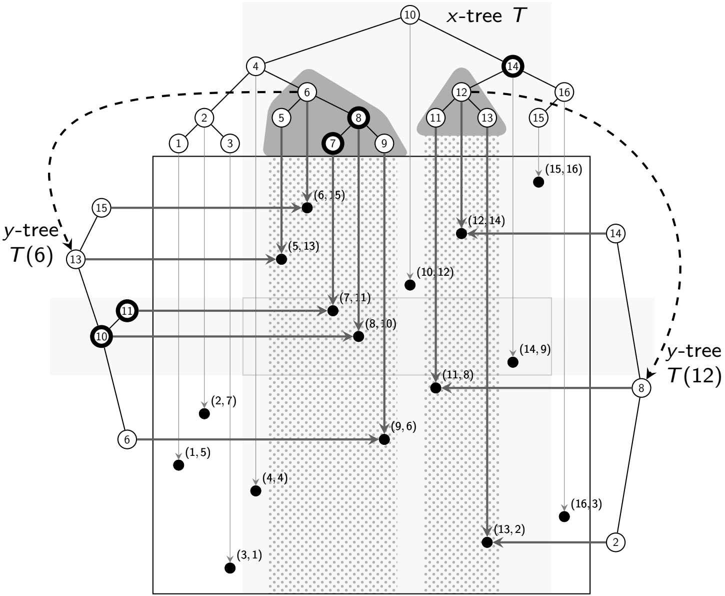
- **Search**: as in a binary search tree
- **Insert**: First, insert point by x-coordinate into T.
  Then, walk back up to the root and insert the point by y-coordinate in *all* $T(v)$ of nodes v on path to the root.
- **Delete**: analogous to insertion

- **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-RangeSearch*(T, x_1, x_2))
  - Obtain boundary, topmost outside and allocation nodes as before.
  - 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.
  - For every boundary node, test to see if the corresponding point is within the region A.

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).

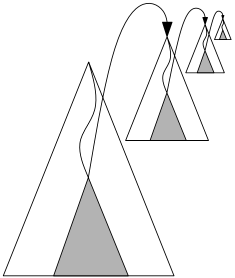
Range Trees: Higher Dimensions

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

Space	$O(n\,(\log n)^{d-1})$	kd-trees: $O(n)$
Construction time	$O(n\,(\log n)^{d-1})$	kd-trees: $O(n \log n)$
Range query time	$O(s + (\log n)^{d})$	kd-trees: $O(s + n^{1-1/d})$

(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