

CS 240 – Data Structures and Data Management

Module 8: Range-Searching in Dictionaries for Points

Armin Jamshidpey, Éric Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

Outline

8 Range-Searching in Dictionaries for Points

- Range Searches
- Multi-Dimensional Data
- Quadtrees
- kd-Trees
- Range Trees

Outline

8 Range-Searching in Dictionaries for Points

- Range Searches
- Multi-Dimensional Data
- Quadtrees
- kd-Trees
- Range Trees

Range searches

- So far: *search*(k) looks for *one* specific item.
- **range-search**: look for *all* items in a given range.
 - ▶ Input: A **range**, i.e., an interval $Q = (x, x')$ (open or closed)
 - ▶ Want: Report all KVPs in the dictionary whose key k satisfies $k \in Q$

Example:

5	10	11	17	19	33	45	51	55	59
---	----	----	----	----	----	----	----	----	----

range-search($(18, 45]$) should return $\{19, 33, 45\}$

- As usual n denotes the number of input-items.
- Let s be the **output-size**, i.e., the number of items in the range.
- We need $\Omega(s)$ time simply to report the items.
- Note that sometimes $s = 0$ and sometimes $s = n$; we therefore keep it as a separate parameter when analyzing the run-time.

Typical run-time: $O(\log n + s)$.

Outline

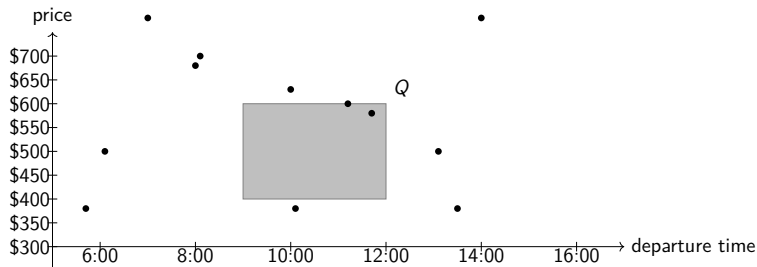
8 Range-Searching in Dictionaries for Points

- Range Searches
- **Multi-Dimensional Data**
- Quadtrees
- kd-Trees
- Range Trees

Multi-dimensional data

Range searches are of special interest for **multi-dimensional data**.

Example: flights that leave between 9am and noon, and cost \$400-\$600



- Each item has d **aspects** (coordinates): $(x_0, x_1, \dots, x_{d-1})$ so corresponds to a point in d -dimensional space
- We concentrate on $d = 2$, i.e., points in Euclidean plane
- (Orthogonal) **d -dimensional range search**: Given a **query rectangle** $Q = [x_1, x'_1] \times \dots \times [x_d, x'_d]$, find all points that lie within Q .

Multi-dimensional range search

The time for range searches depends on how the points are stored.

- Two naive ideas that do not work well:
 - ▶ Store a 1-dimensional dictionary (where the key is some combination of the aspects.)
Problem: Range search on one aspect is not straightforward
 - ▶ Search in separate dictionaries for aspects and take intersection.
Problem: inefficient, intersection may be empty.
- **Better idea:** Design new data structures specifically for points.
 - ▶ Quadtrees
 - ▶ kd-trees
 - ▶ range-trees

Assumption: Points are in **general position**:

- No two points on a horizontal line.
- No two points on a vertical line.

This simplifies presentation; data structures can be generalized.

Outline

8 Range-Searching in Dictionaries for Points

- Range Searches
- Multi-Dimensional Data
- **Quadtrees**
- kd-Trees
- Range Trees

Quadtrees

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

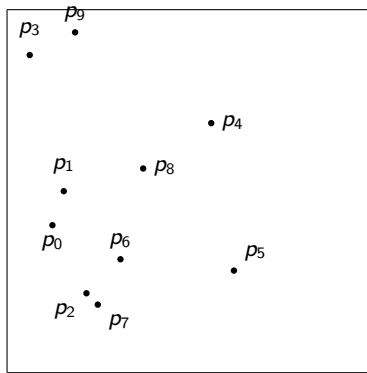
Find a **bounding box** $R = [0, 2^k) \times [0, 2^k)$: a square containing all points.

- Assume (after translation) that all coordinates are non-negative.
- Find max-coordinate in P , use the smallest k such that it is $< 2^k$.

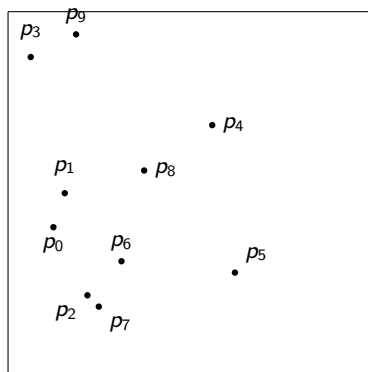
Structure (and also how to *build* the quadtree that stores P):

- Root r of the quadtree is associated with region 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**)
 $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Partition P into sets $P_{NE}, P_{NW}, P_{SW}, P_{SE}$ of points in these regions.
 - ▶ **Convention**: Points on split lines belong to right/top side
- Recursively build tree T_i for points P_i in region R_i and make them children of the root.

Quadtrees: Example

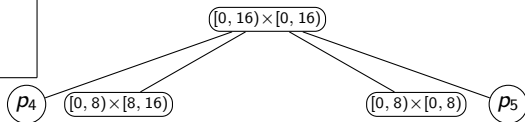
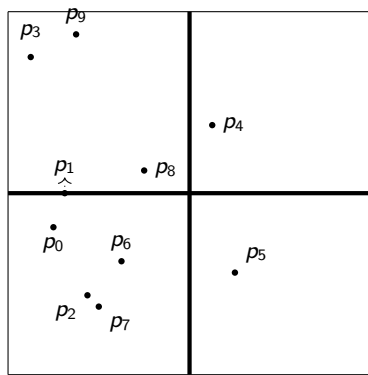


Quadtrees: Example

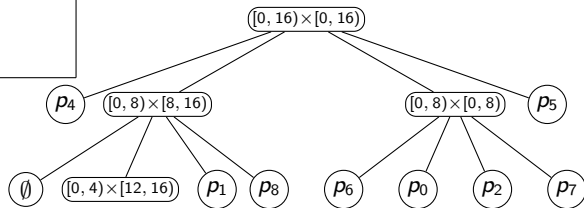
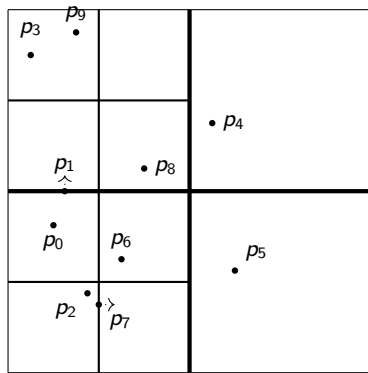


$$[0, 16) \times [0, 16)$$

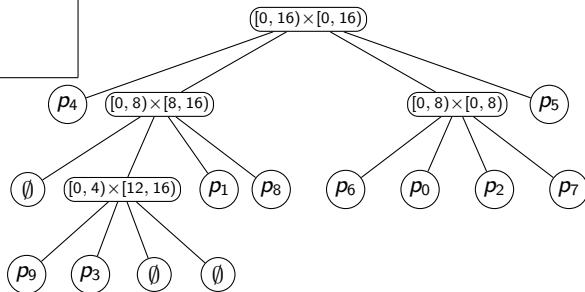
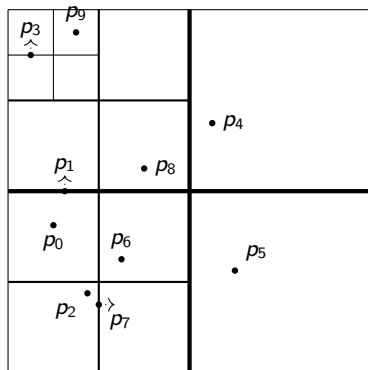
Quadtrees: Example



Quadtrees: Example



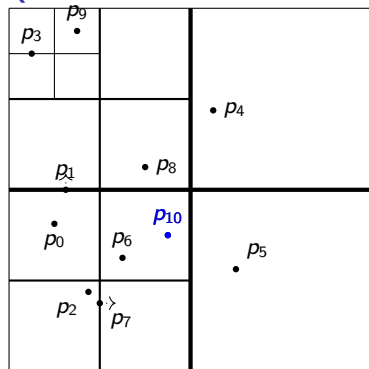
Quadtrees: Example



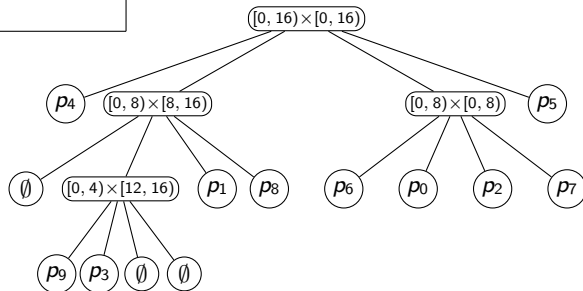
Quadrees: Dictionary operations

- *search*(for single point): 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)

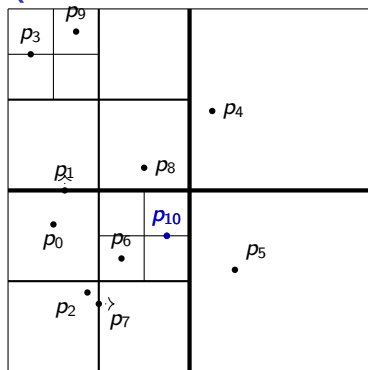
Quadtrees insertion: Example



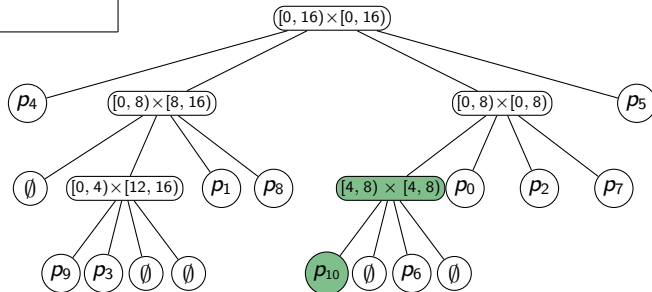
insert(p_{10})



Quadtrees insertion: Example



insert(p_{10})



Quadtrees: Range search

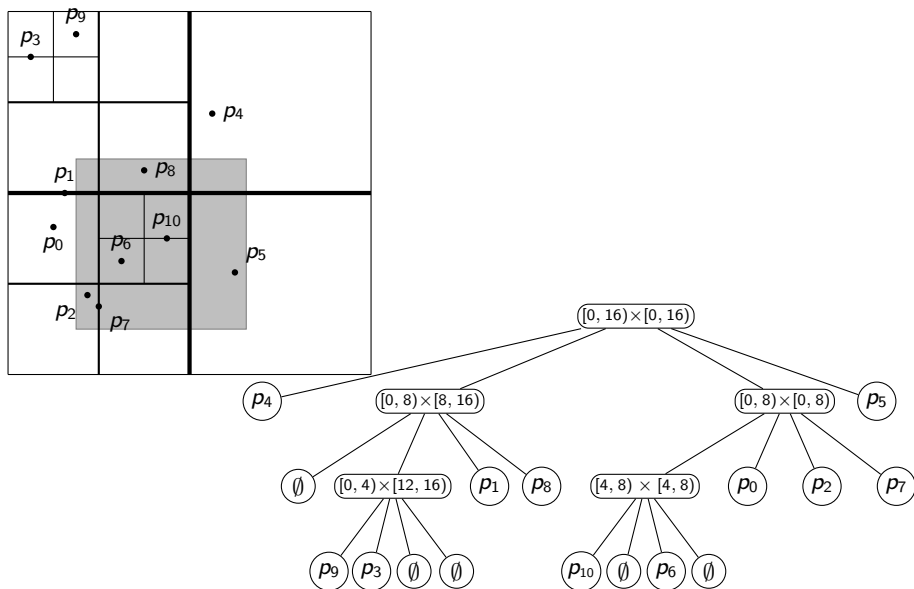
QTree::range-search($r \leftarrow \text{root}$, Q)

r : The root of a quadtree, Q : Query-rectangle

1. $R \leftarrow$ region associated with node r
2. **if** ($R \subseteq Q$) **then** // inside node, stop search
 report all points below r and **return**
3. **else if** ($R \cap Q$ is empty) **then return** // outside node, stop search
 // boundary node, recurse
4. **if** (r is a leaf) **then**
5. $p \leftarrow$ point stored at r
6. **if** p is not NULL and in Q **then** report it
7. **return**
8. **for** each child v of r **do** *QTree::range-search*(v , Q)

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).

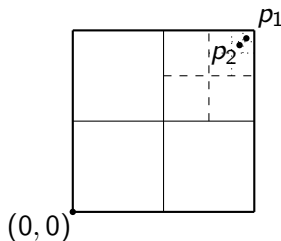
Quadtrees range search: Example



Quadtrees: Analysis

Complexity of range search:

- In worst-case, we look at nearly all nodes, even if the answer is $\emptyset$.
- The number nodes could be $\Theta(nh)$, where h is the height.
- Can have very large height for bad distributions of points.



(Even with $n = 3$ points, the height can be arbitrarily large.)

In practice, quad-trees work quite well. Theoretical evidence (no details):

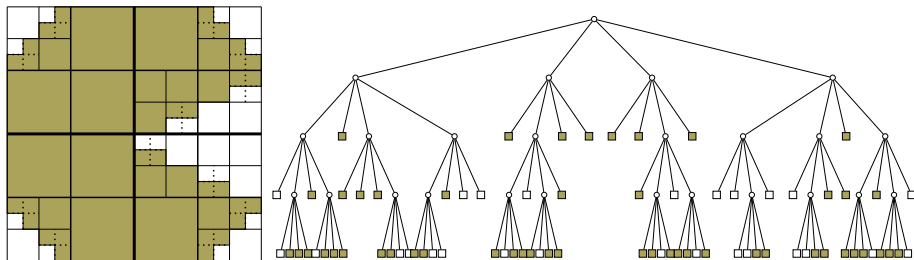
- For n randomly chosen points, the expected height is $O(\log n)$.
- The height depends on the **spread factor**:

$$\frac{\text{sidelength of bounding box}}{\text{minimum distance between points in } P}$$

The height is in $\Theta(\log(\text{spread factor}))$

Quadtrees: Summary

- Very easy to compute and handle
- No complicated arithmetic, only divisions by 2 if the width/height of bounding box R is a power of 2
- Space potentially wasteful, but good if points are well-distributed
- Quadtrees also easily generalize to higher dimensions (split into octants $\rightarrow$ octrees, etc.)
- Variation: Use quad-tree to store pixelated images.



Outline

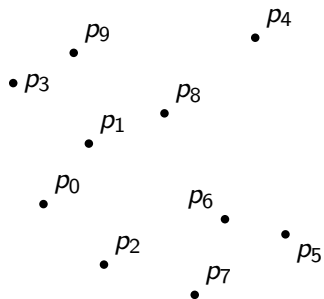
8 Range-Searching in Dictionaries for Points

- Range Searches
- Multi-Dimensional Data
- Quadtrees
- **kd-Trees**
- Range Trees

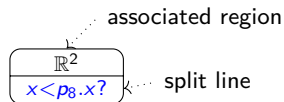
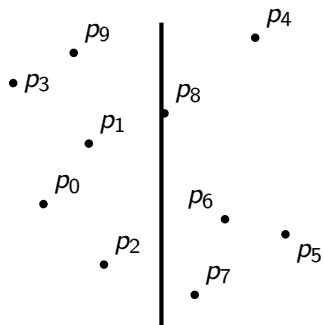
kd-trees

- We have n points $P = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$
- Quadtrees: Split region into quadrants regardless of where points are
- kd-tree idea: Split region based on where points are.
 - ▶ We split at upper median of coordinates
 - ↪ roughly half of 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

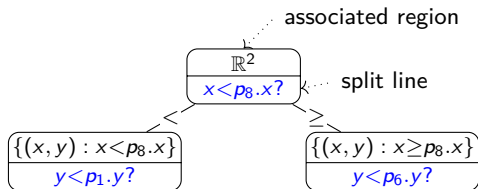
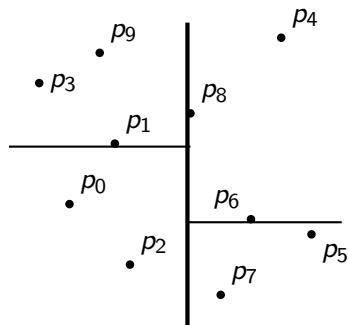
kd-trees: Example



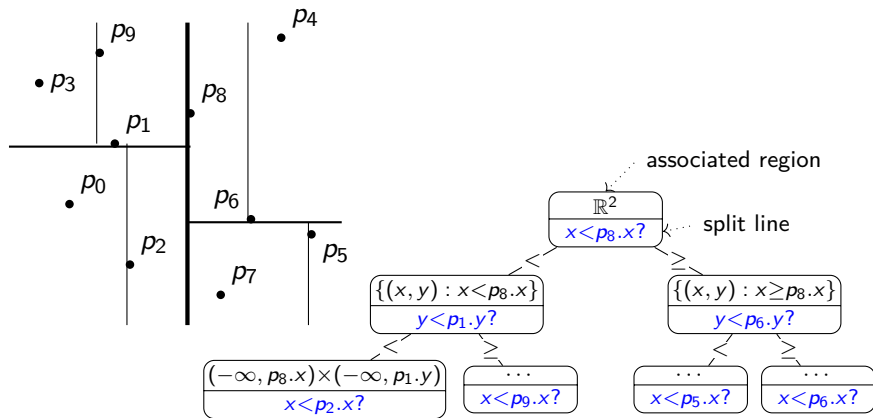
kd-trees: Example



kd-trees: Example

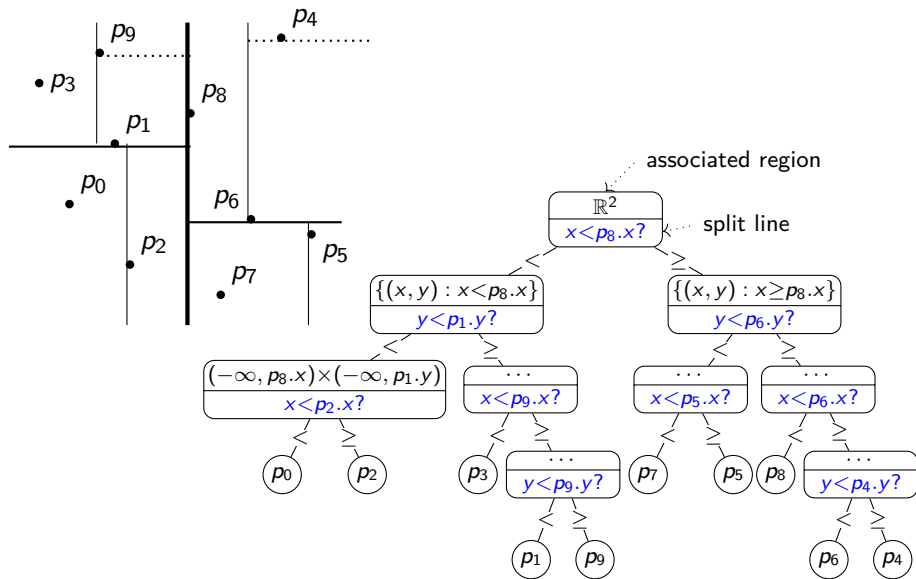


kd-trees: Example



For ease of drawing, we will usually not list the associated regions.

kd-trees: Example



For ease of drawing, we will usually not list the associated regions.

kd-trees: Construction

The algorithm to build a kd-tree is immediate from the definition of a kd-tree:

To build a kd-tree with initial split by x on points P :

- If $|P| \leq 1$ create a leaf and return.
- Else $X := \text{randomized-quick-select}(P, \lfloor \frac{n}{2} \rfloor)$ (select by x -coordinate)
- Partition P by x -coordinate into $P_{x < X}$ and $P_{x \geq X}$
 - ▶ $\lfloor \frac{n}{2} \rfloor$ points on one side and $\lceil \frac{n}{2} \rceil$ points on the other.
(Recall: Points in general position.)
- Create left subtree recursively (splitting by y) for points $P_{x < X}$.
- Create right subtree recursively (splitting by y) for points $P_{x \geq X}$.

Building with initial y -split symmetric.

kd-trees: Analysis

Run-time:

- Find X and partition P takes $\Theta(n)$ expected time.
- Both subtrees have $\approx n/2$ points.

$$T^{\text{exp}}(n) \leq 2T^{\text{exp}}(n/2) + O(n) \quad (\text{sloppy recurrence})$$

This resolves to $O(n \log n)$ expected time.

- This can be reduced to $O(n \log n)$ *worst-case* time by pre-sorting
- This is tight (exercise)

Height: $h(1) = 0, h(n) \leq h(n/2) + 1$

- This resolves to $O(\log n)$
- This is tight: $h \in \Omega(\log(\text{number of nodes})) \subseteq \Omega(\log n)$.

Space: All interior nodes have exactly two children.

- Therefore have $n - 1$ interior nodes, so $2n - 1$ nodes in total

kd-trees: Dictionary operations

- *search* (for single point): as in binary search tree using indicated coordinate
- *insert*: search, insert as new leaf.
- *delete*: search, remove leaf.

Problem: After insert or delete, the split might no longer be at exact median and the height is no longer guaranteed to be $O(\log_2 n)$.

We can maintain $O(\log n)$ height by occasionally re-building entire subtrees. (No details.)

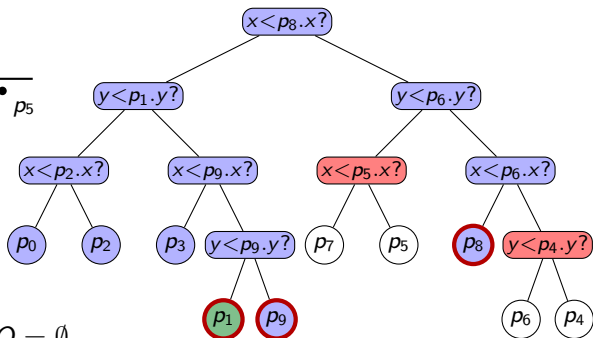
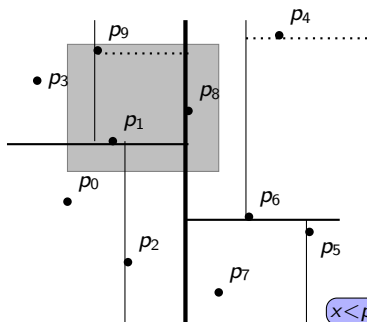
kd-trees do not handle insertion/deletion well.

kd-trees: Range search

Range search is *exactly* as for quad-trees, except that there are only two children and leaves always store points.

```
kdTree::range-search( $r \leftarrow \text{root}$ ,  $Q$ )
1.  $R \leftarrow$  region associated with node  $r$ 
2. if ( $R \subseteq Q$ ) then                                // inside node, stop search
    report all points below  $r$  and return
3. else if ( $R \cap Q$  is empty) then return // outside node, stop search
                                           // boundary node, recurse
4. if ( $r$  is a leaf) then
5.      $p \leftarrow$  point stored at  $r$ 
6.     if  $p$  is in  $Q$  then report it
7.     return
8. for each child  $v$  of  $r$  do kdTree::range-search( $v$ ,  $Q$ )
```

kd-tree range search: Example



- **Outside** node: $R \cap Q = \emptyset$.
- **Inside** node: $R \subseteq Q$.
- **Boundary** node: Neither of the above.

kd-tree range search: Analysis

- We spend $O(1)$ time at each visited node, except in line 2.
- All calls to line 2 together take $O(s)$ time (recall: $s = \text{output-size}$)
 - ▶ Line 2 at node z takes time $O(\text{size of subtree at } z)$
 - ▶ This size is $O(\text{number of leaves below } z)$
 - ▶ The points at these leaves are reported.
- **Observe:** $\# \text{ visited nodes} \leq 2 \cdot \underbrace{\# \text{ boundary-nodes}}_{\beta(n)} + 1$
 - ▶ At a visited node, the parent was a boundary-node (except at root)
 - ▶ Every boundary-node has at most two children.
- **Can show:** $\beta(n)$ satisfies the following recurrence relation:

$$\beta(n) \leq 2\beta(n/4) + O(1)$$

- This implies $\beta(n) \in O(\sqrt{n})$
- Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$

kd-trees: 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:** $O(n)$
- **Height:** $O(\log n)$
- **Construction time:** $O(n \log n)$
- **Range search time:** $O(s + n^{1-1/d})$

This assumes that d is a constant.

Outline

8 Range-Searching in Dictionaries for Points

- Range Searches
- Multi-Dimensional Data
- Quadtrees
- kd-Trees
- Range Trees

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**

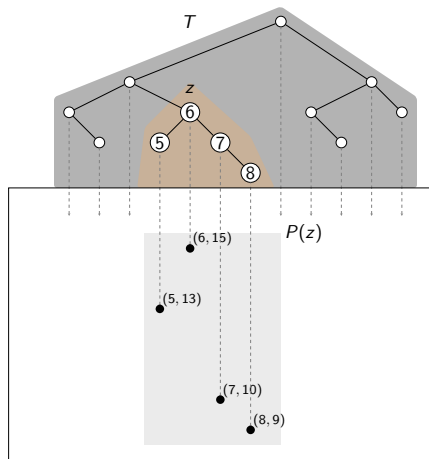
- **Tree of trees** (a *multi-level* data structure)
 - ▶ So far, nodes in our trees stored a key-value pair and references to children and (maybe) the parent
 - ▶ But we can store much more in a node
 - ▶ Here: Each node stores in another binary search tree
- They are wasteful in space, but permit much faster range search.

2-dimensional range trees

Primary structure:

Balanced binary search tree T that stores P and uses *x -coordinates* as keys.

For each node z of T , let $P(z)$ be all points at descendants of z (including point at z)

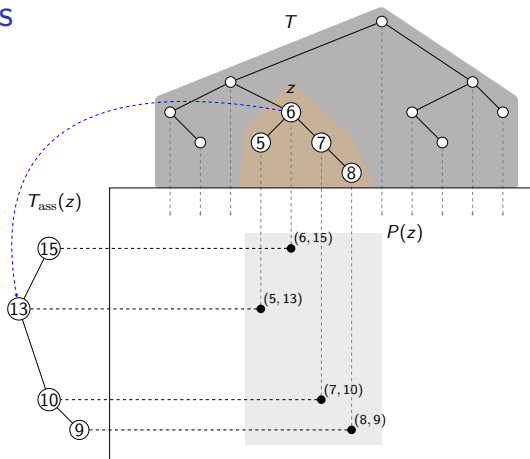


2-dimensional range trees

Primary structure:

Balanced binary search tree T that stores P and uses *x-coordinates* as keys.

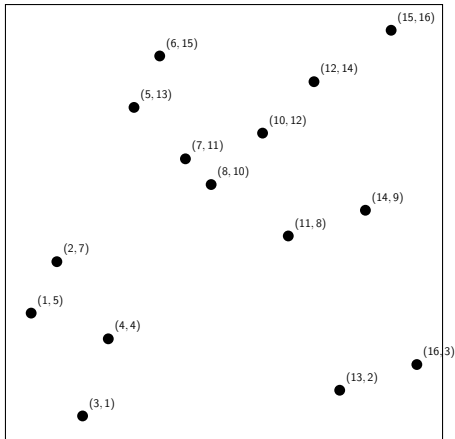
For each node z of T , let $P(z)$ be all points at descendants of z (including point at z)



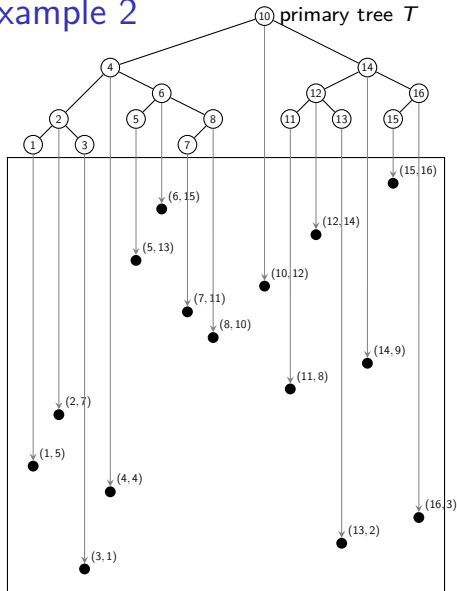
Every node z of T stores an **associate structure** $T_{\text{ass}}(z)$:

- $T_{\text{ass}}(z)$ stores $P(z)$ in a balanced binary search tree, using the *y-coordinates* as key
- Note: Point of z is not necessarily the root of $T_{\text{ass}}(z)$

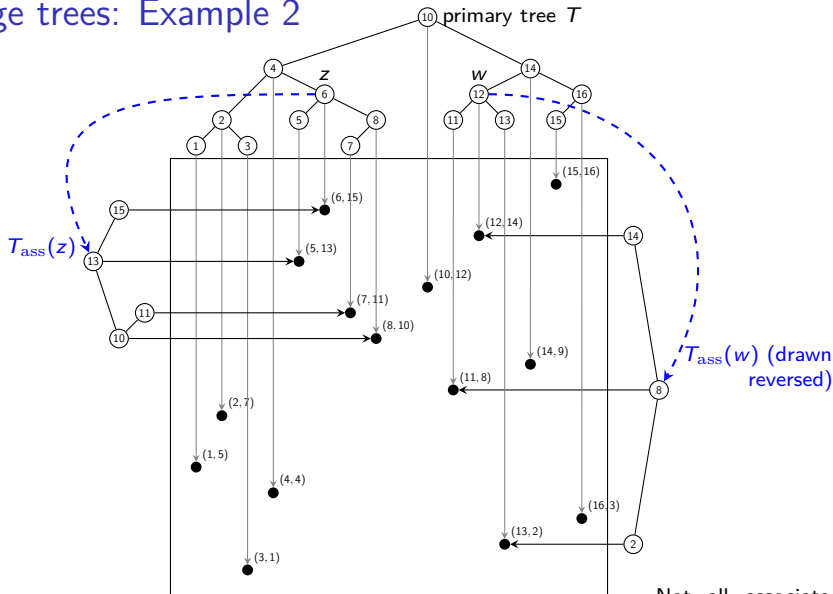
Range trees: Example 2



Range trees: Example 2



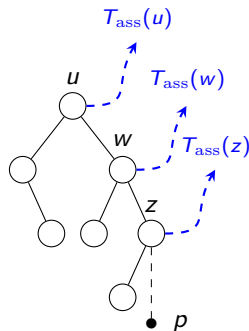
Range trees: Example 2



Not all associate trees are shown.

Range trees: Space analysis

- Primary tree T uses $O(n)$ space and has height $O(\log n)$.
- How many nodes do all associate trees together have?

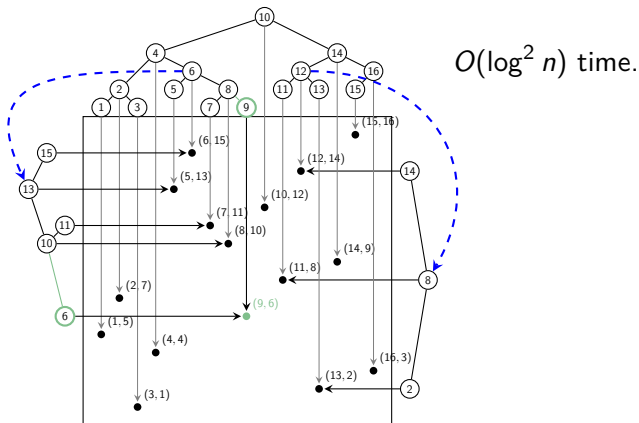


- ▶ Consider a point p stored at node z in T :
 - ★ $T_{\text{ass}}(z)$ stores p
 - ★ $T_{\text{ass}}(\text{parent}(z))$ stores p
 - ★ $T_{\text{ass}}(\text{parent}(\text{parent}(z)))$ stores p , etc.
 - ★ No other associate trees store p
- ▶ **Key insight:** p is in associate tree $T_{\text{ass}}(u)$ if and only if u is an ancestor of node z that stores p .
- ▶ So every point belongs to $O(\log n)$ associate trees.
- ▶ So all associate trees together use $O(n \log n)$ space.

A range-tree with n points uses $O(n \log n)$ space.

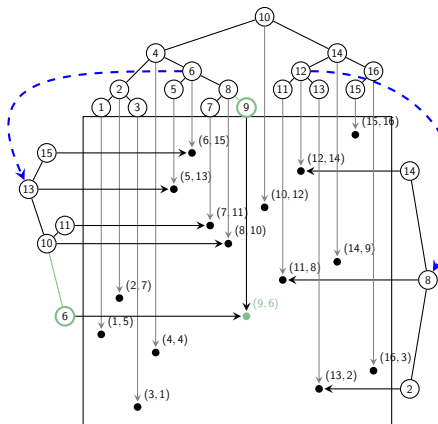
Range trees: Dictionary operations

- *search*: search by x -coordinate in T
- *insert/delete*: First, insert/delete point by x -coordinate into T . Then, walk back up to the root and insert/delete the point by y -coordinate in *all* associate trees $T_{\text{ass}}(z)$ of nodes z on path.



Range trees: Dictionary operations

- *search*: search by x-coordinate in T
- *insert/delete*: First, insert/delete point by x-coordinate into T . Then, walk back up to the root and insert/delete the point by y-coordinate in *all* associate trees $T_{\text{ass}}(z)$ of nodes z on path.



$O(\log^2 n)$ time.

Problem: Need to keep T balanced

- ▶ Cannot afford to do rotations. (A rotation at z changes $P(z)$, so requires a re-build of $T_{\text{ass}}(z)$.)
- ▶ **Solution:** Completely rebuild highly unbalanced subtrees (no details)

Range trees: Range search

Range search for $Q = [x_1, x_2] \times [y_1, y_2]$ is a two stage process:

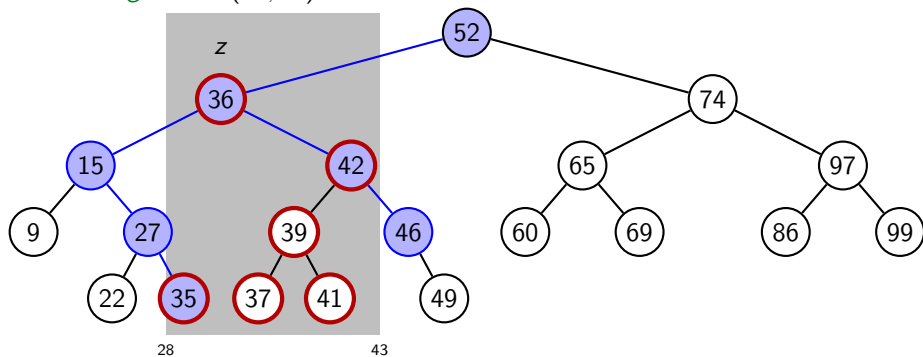
- Perform a range search (on the x -coordinates) for the interval $[x_1, x_2]$ in primary tree T but do **not** report all points.
- Instead, return $O(\log n)$ nodes of T with special properties.
- Use these to determine the points in range, using the associated trees.

So must understand first: How to do (1-dimensional) range search in binary search tree?

- There is an easy way to do this recursively, similar to *kd-tree::range-search* ($\rightsquigarrow$ course notes)
- For use in range trees, we do a more complicated approach that returns relevant nodes of T .

BST range search

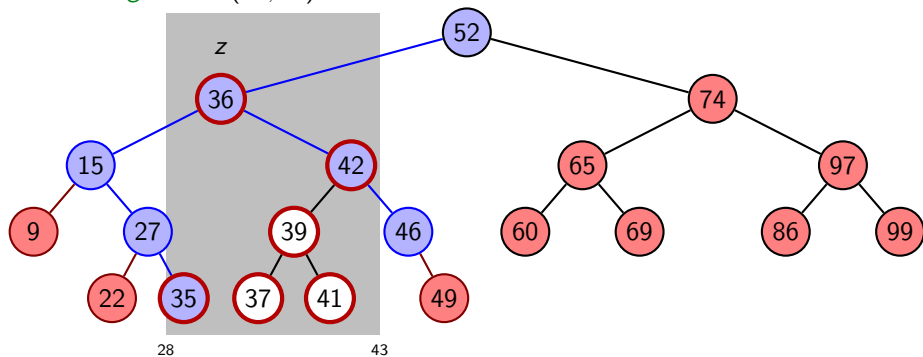
BST::range-search(28, 43)



- Search for left boundary x_1 : this gives path P_1
- Search for right boundary x_2 : this gives path P_2
- **boundary nodes**: nodes in P_1 or P_2
- Also important: node z where P_1 and P_2 diverge

BST range search

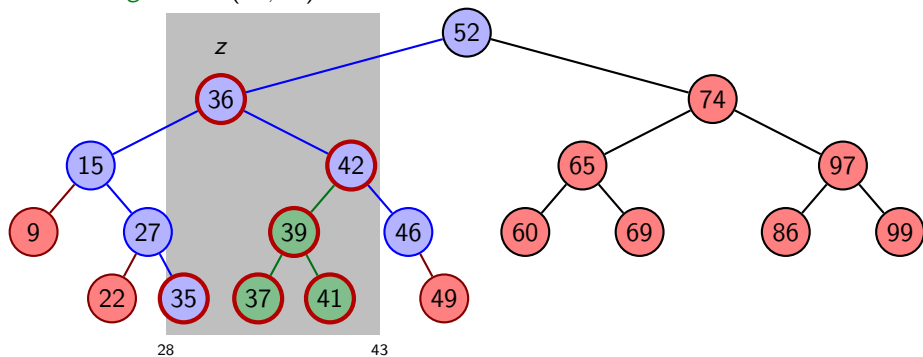
BST::range-search(28, 43)



- **Outside nodes:** left of P_1 or right of P_2
- These can never be in the range.
- We do not do anything at them.

BST range search

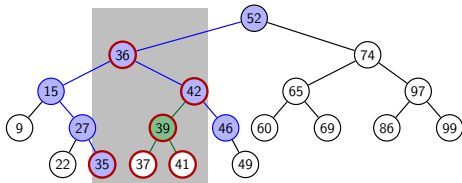
BST::range-search(28, 43)



- **Inside nodes**: right of P_1 and left of P_2 . All are in the range.
- A node u is a **topmost inside node** if
 - ▶ Parent p is a boundary-node below the node where the paths diverge.
 - ▶ If $p \in P_1$ then u is right child. If $p \in P_2$ then u is left child.
- We compute and report topmost inside nodes, in $O(\text{height})$ time.

BST range search: Analysis

Assume that the binary search tree is balanced (so has height $O(\log n)$):



- Search for boundary nodes: $O(\log n)$
- Search for topmost inside nodes: $O(\log n)$
- Every descendant of a topmost inside node is in the range:
- Range search:
 - ▶ Check for each boundary node whether it is in the range.
 - ▶ Report all descendants of topmost inside nodes.
 - ▶ We have $\sum_z \text{topmost inside } \#\{\text{descendants of } z\} \leq s$ since subtrees of topmost inside nodes are disjoint.

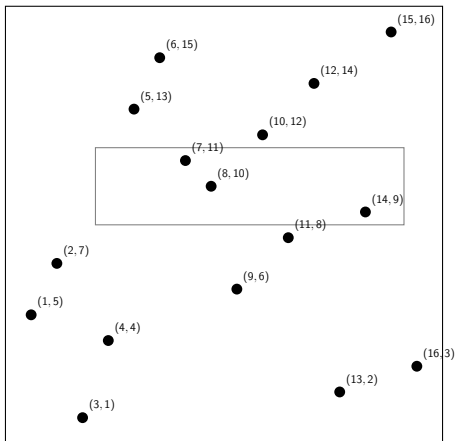
Run-time: $O(\log n + s)$.

Range Trees: Range Search

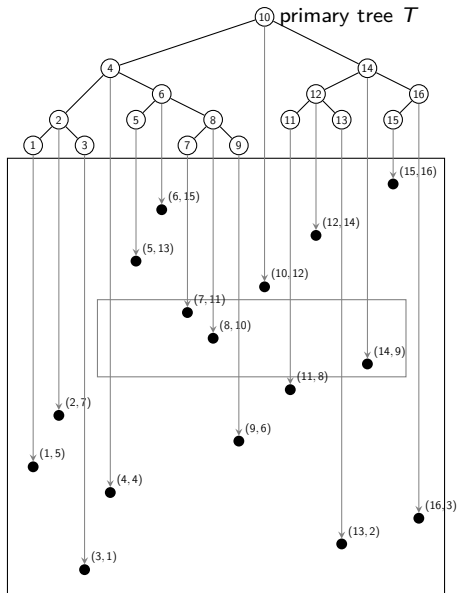
Range search for $Q = [x_1, x_2] \times [y_1, y_2]$ is a two stage process:

- Perform a range search (on the x -coordinates) for the interval $[x_1, x_2]$ in primary tree T ($BST::range-search(T, x_1, x_2)$)
- Just get **boundary** and **topmost inside** nodes.
- For every **boundary node**, test to see if the corresponding point is within the region Q .
- For every **topmost inside node** z :
 - ▶ Let $P(z)$ be the points in the subtree of z in T .
 - ▶ We know that all x -coordinates of points in $P(z)$ are within range.
 - ▶ Recall: $P(z)$ is stored in $T_{ass}(z)$.
 - ▶ To find points in $P(z)$ where the y -coordinates are within range as well, perform a range search in $T_{ass}(z)$: $BST::range-search(T_{ass}(z), y_1, y_2)$

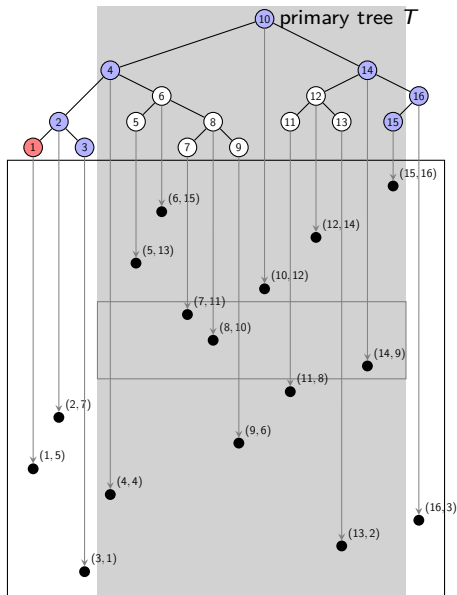
Range tree range search: Example



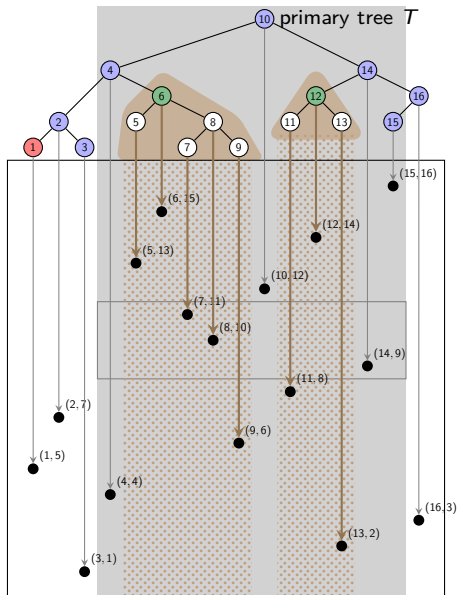
Range tree range search: Example



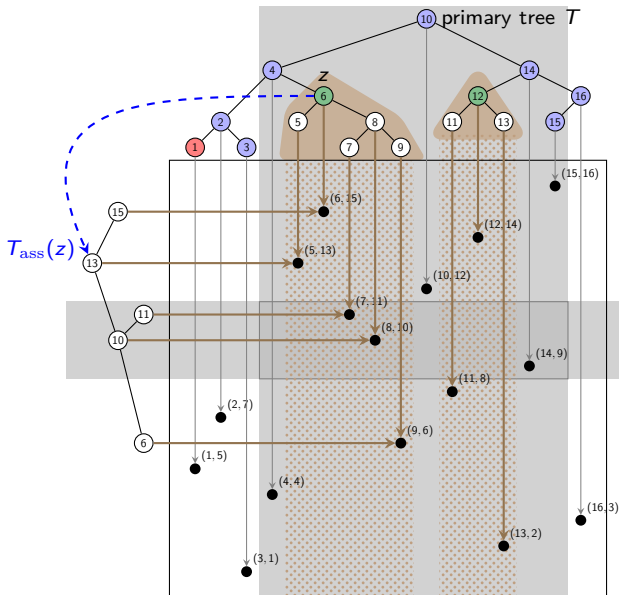
Range tree range search: Example



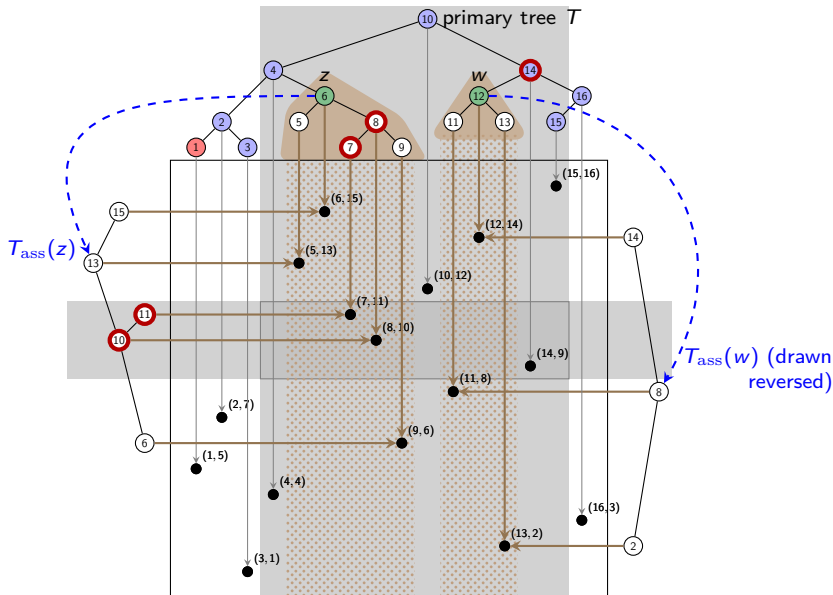
Range tree range search: Example



Range tree range search: Example



Range tree range search: Example



Range trees range search: Run-time

- $O(\log n)$ time to find boundary and topmost inside nodes in primary tree.
- There are $O(\log n)$ such nodes.
- $O(\log n + s_z)$ time for each topmost inside node z , where s_z is the number of points in $T_{\text{ass}}(z)$ that are reported
- As before $\sum_z \text{topmost inside } s_z \leq s$, so time for range search in range-tree is proportional to

$$\sum_{z \text{ topmost inside}} (\log n + s_z) \in O(\log^2 n + s)$$

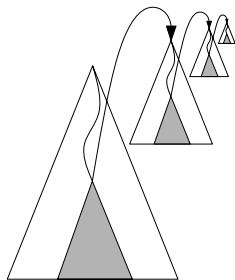
Range Trees: Higher Dimensions

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

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

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

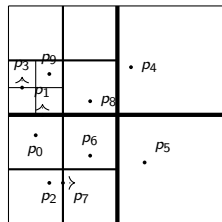
- Space/time trade-off compared to kd-trees.



Range search data structures: Summary

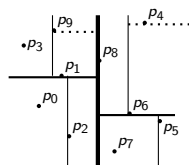
• Quadtrees

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



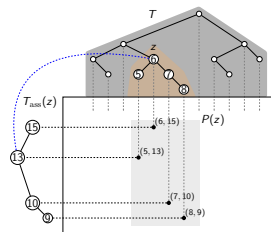
• kd-trees

- ▶ linear space
- ▶ range search time $O(\sqrt{n} + s)$
- ▶ inserts/deletes destroy balance and range search time (no simple fix)



• range-trees

- ▶ range search time $O(\log^2 n + s)$
- ▶ wastes some space
- ▶ inserts/deletes destroy balance (can fix this with occasional rebuild)



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