CS 240E – Data Structures and Data Management (Enriched)

Module 8: Range-Searching in Dictionaries for Points

Therese Biedl

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025−03−03 11:52

Outline

8. Range-Searching in Dictionaries for Points
   - Range Searches
   - Multi-Dimensional Data
   - Quadtrees
   - kd-Trees
   - Range Trees
   - 3-sided range search

Outline

8. Range-Searching in Dictionaries for Points
   - Range Searches
   - Multi-Dimensional Data
   - Quadtrees
   - kd-Trees
   - Range Trees
   - 3-sided range search

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 $\{19, 33, 45\}$

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: $O(\log n + s)$.

Range searches in existing dictionary realizations

Unsorted list/array/hash table: Range search requires $\Omega(n)$ time: We have to check for each item explicitly whether it is in the range.

Sorted array: Range search in A can be done in $O(\log n + s)$ time:

range-search((18,45])

5	10	11	17	19	33	45	51	55	59
			$\uparrow i$			$\uparrow i'$			

- Using binary search, find i such that x is at (or would be at) $A[i]$.
- Using binary search, find i' such that x' is at (or would be at) $A[i']$
- Report all items $A[i{+}1...i'{-}1]$
- Report $A[i]$ and $A[i']$ if they are in range

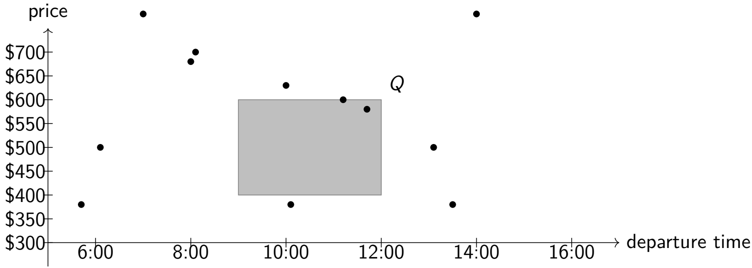
BST: Range searches can similarly be done in time $O(\text{height}{+}s)$ time. We will see this in detail later.

Outline

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): $(x_0, x_1, \cdots, x_{d-1})$ 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** $Q = [x_1, x_1'] \times \cdots \times [x_d, x_d']$, 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:
  - Could store a 1-dimensional dictionary (where the key is some combination of the aspects.)
    Problem: Range search on one aspect is not straightforward
  - Could use one dictionary for each aspect
    Problem: inefficient, wastes space
- **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
   - 3-sided range search

Quadtrees

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

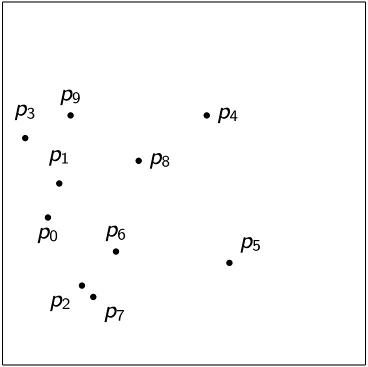
Find a **bounding box** $R = [0, 2^k) \times [0, 2^k)$: a square containing all points.

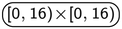
- Assume (after translation) that all coordinates are non-negative.
- Find max-coordinate in P, use the smallest k such that it is $< 2^k$.

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

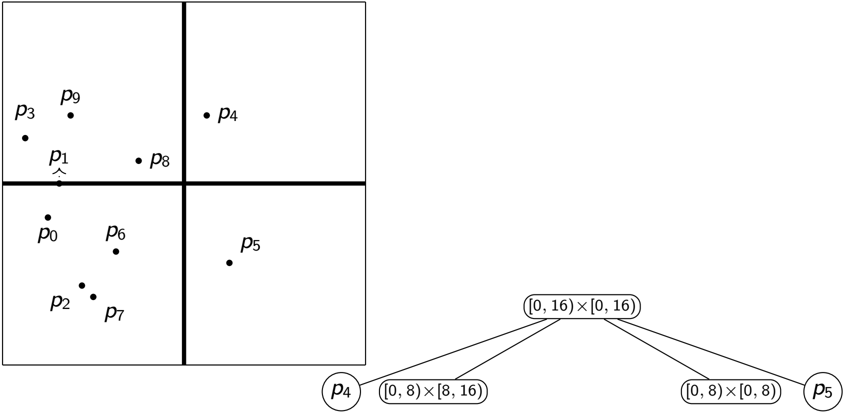
- Root r of the quadtree is associated with region R
- If R contains 0 or 1 points, then root r is a leaf that stores point.
- Else *split*: Partition R into four equal subsquares (**quadrants**) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Partition P into sets $P_{NE}, P_{NW}, P_{SW}, P_{SE}$ of points in these 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.

Quadtree example

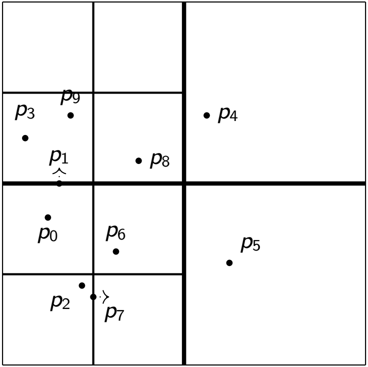


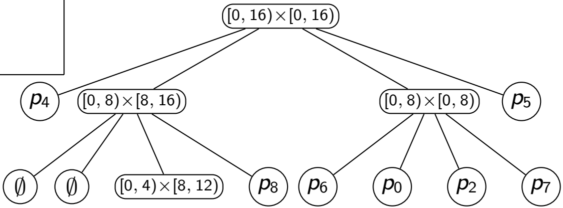


Quadtree example

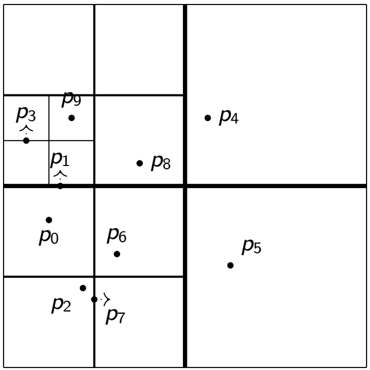


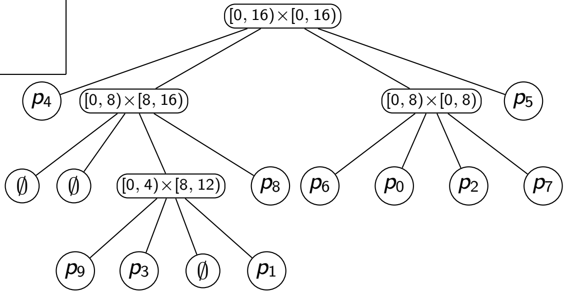
Quadtree example





Quadtree example

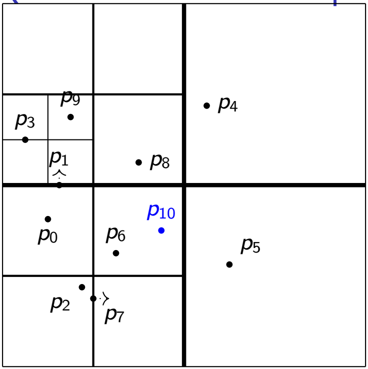




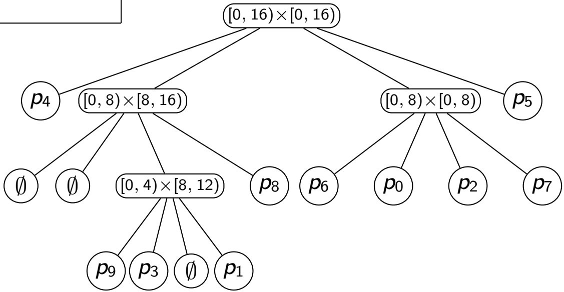
Quadtree Dictionary Operations

- *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)

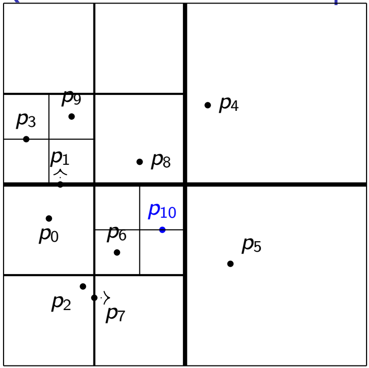
Quadtree Insert example



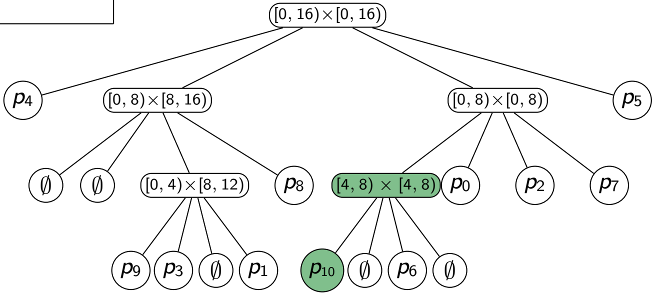
insert(p_{10})



Quadtree Insert example



insert(p_{10})



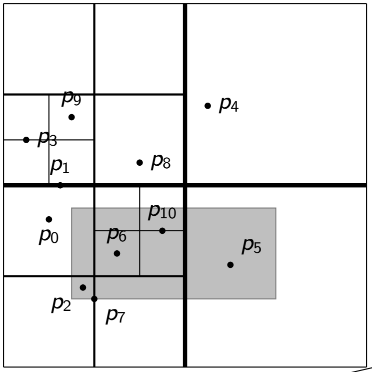
Quadtree Range Search

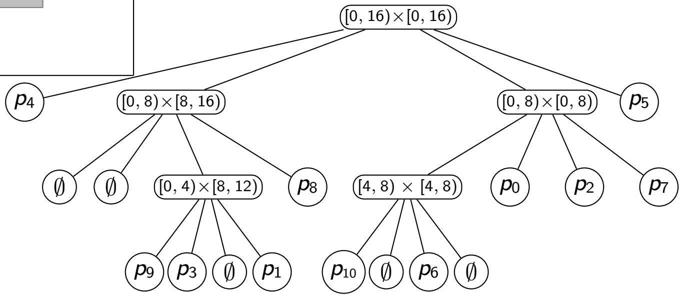
QTree::range-search($r \leftarrow root, Q$)
r: The root of a quadtree, Q: Query-rectangle
1. $R \leftarrow$ region associated with node r
2. **if** ($R \subseteq Q$) **then** // inside node, stop searching
    report all points below r and **return**
3. **else if** ($R \cap Q$ is empty) **then return** // outside node, stop searching

    // 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 and **return**
7. **else return**
8. **for** each child v of r **do** *QTree::range-search*(v, Q)

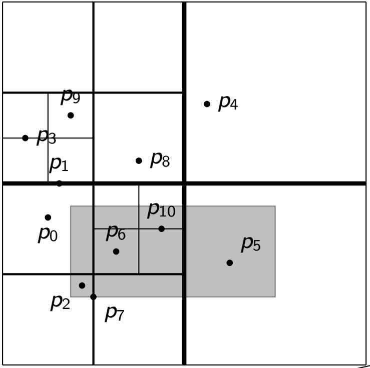
Note: We assume here that each node of the quadtree stores the associated square. Alternatively, these could be re-computed during the search (space-time tradeoff).

Quadtree range search example

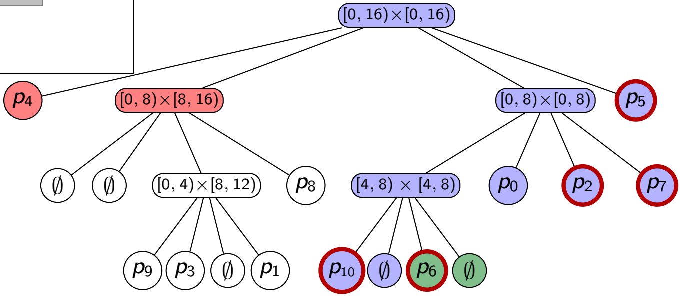




Quadtree range search example



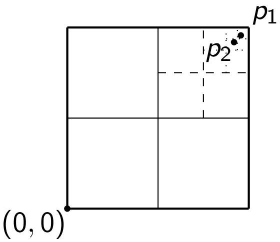
- Green: Search stopped due to $R \subseteq Q$.
- Red: Search stopped due to $R \cap Q = \emptyset$.
- Blue: Must continue search in children / evaluate.



Quadtree 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.

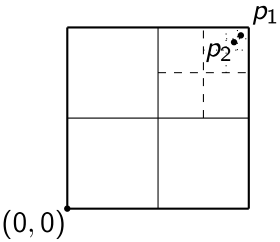


(Even with $n = 3$ points, the height can be arbitrarily large.)

Quadtree 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**:

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

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

Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

"Points:" 0 9 12 14 24 26 28

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

Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

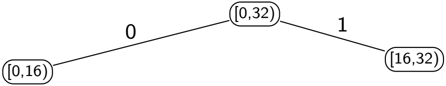
"Points:"	0	9	12	14	24	26	28
(in base-2)	00000	01001	01100	01110	11000	11010	11100

[0,32)

Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:

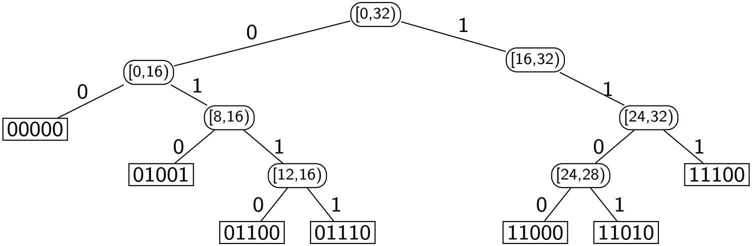
"Points:"	0	9	12	14	24	26	28
(in base-2)	00000	01001	01100	01110	11000	11010	11100



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

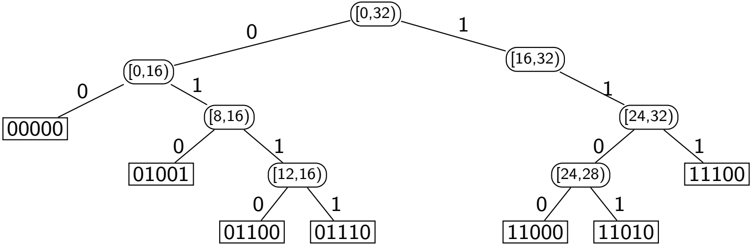


Same as a pruned trie

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

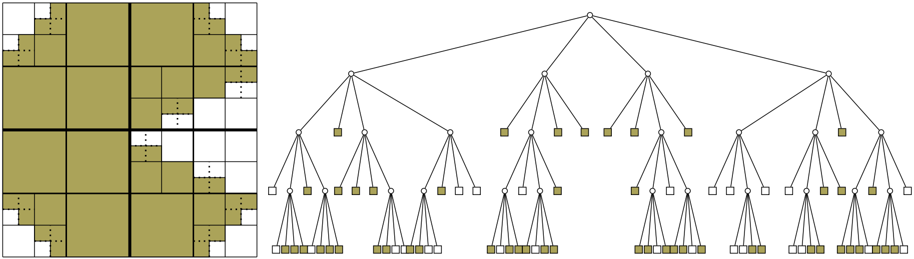


Same as a pruned trie

- Quadtrees also easily generalize to higher dimensions (split into octants $\rightarrow$ 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 bounding box 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 K points in a leaf (for some fixed bound K).
- Variation: Use quad-tree to store pixelated images.



Outline

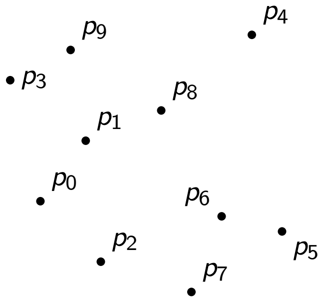
8 Range-Searching in Dictionaries for Points
- Range Searches
- Multi-Dimensional Data
- Quadtrees
- kd-Trees
- Range Trees
- 3-sided range search

kd-trees

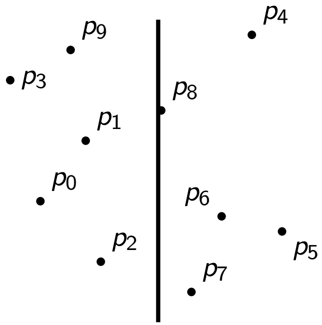
- We have n points $P = \{(x_0, y_0), (x_1, y_1), \cdots, (x_{n-1}, y_{n-1})\}$
- Quadtrees: Split region into quadrants regardless of where points are
- kd-tree idea: Split region based on where points are.
  - We split at upper median of coordinates
  - $\rightsquigarrow$ roughly half of the point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

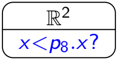
  (There are alternatives, e.g., split by the dimension that has better aspect ratios for the resulting regions.)

kd-tree example

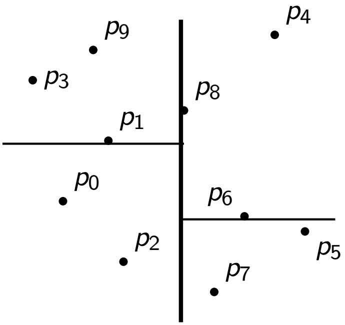


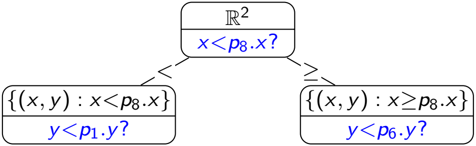
kd-tree example



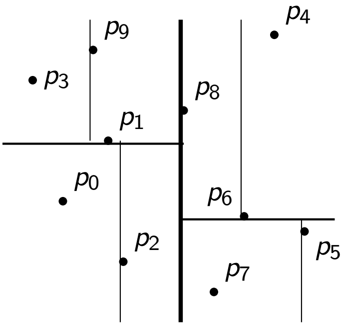


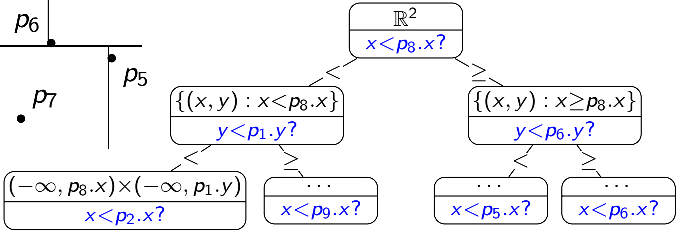
kd-tree example





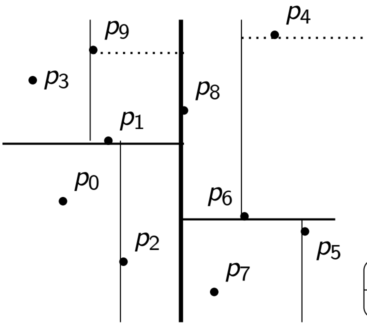
kd-tree example

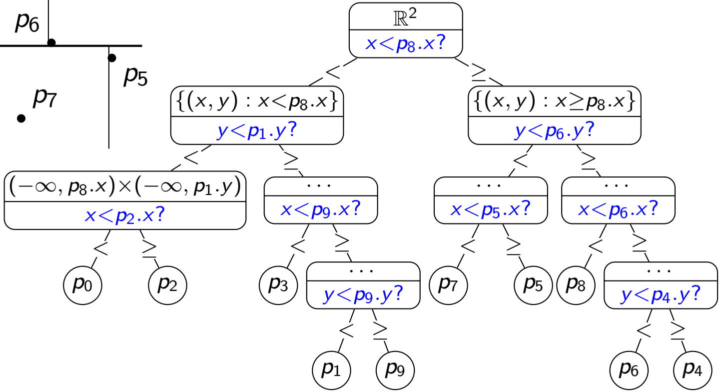




For ease of drawing, we will usually not list the associated regions of nodes.

kd-tree example





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

Constructing kd-trees

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 $X :=$ *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.

Constructing kd-trees

Run-time:

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

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

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

- This can be reduced to $\Theta(n \log n)$ *worst-case* time by pre-sorting.

Height: $h(1) = 0$, $h(n) \le h(\lceil n/2 \rceil) + 1$.

- This resolves to $O(\log n)$ (specifically $\lceil \log n \rceil$).
- This is tight (binary tree with n leaves)

Space: All interior nodes have exactly two children.

- Therefore have $n - 1$ interior nodes.
- Space is $\Theta(n)$.

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.

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

We can maintain $O(\log n)$ height by occasionally re-building entire subtrees. (No details.) But *range-search* will be slower.

kd-trees do not handle insertion/deletion well.

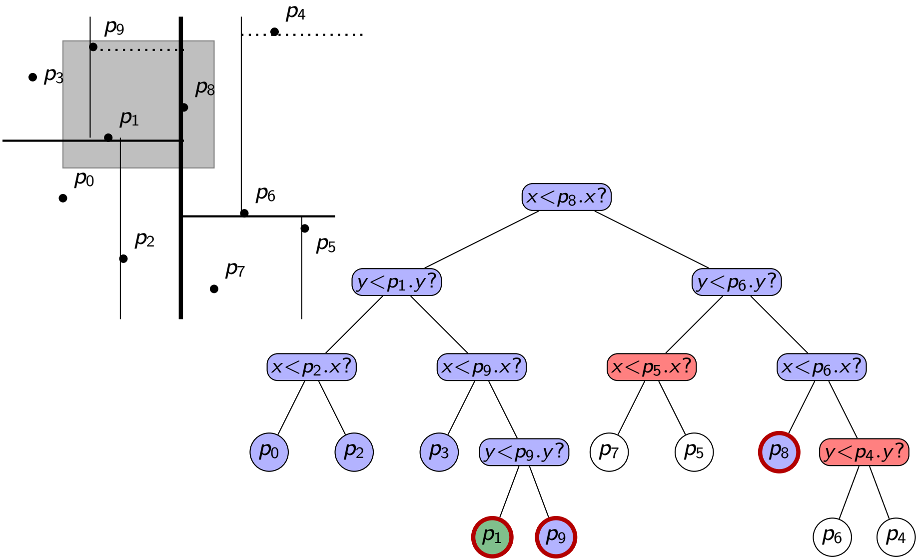
kd-tree Range Search

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

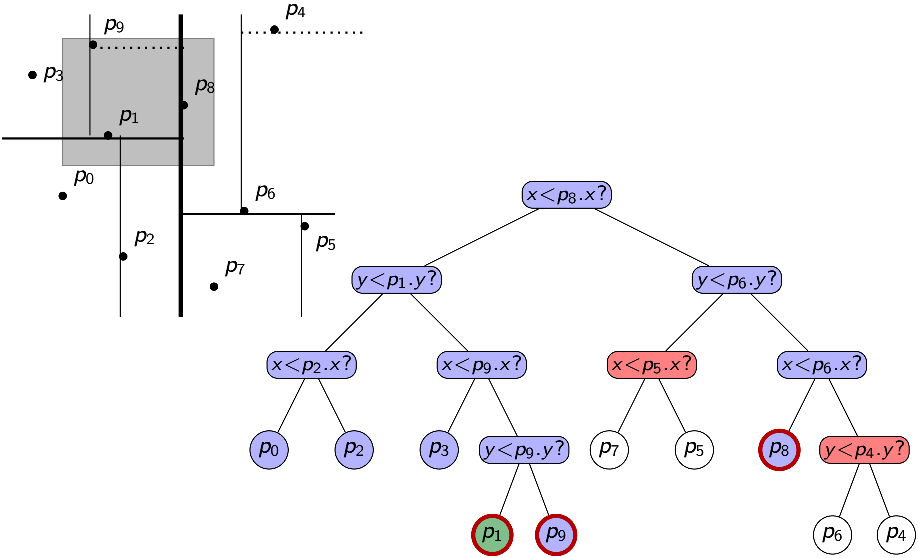
```
kdTree::range-search(r ← root, Q)
r: The root of a kd-tree, Q: Query-rectangle
    R ← region associated with node r
    if (R ⊆ Q) then report all points below r; return
    if (R ∩ Q is empty) then return
    if (r is a leaf) then
        p ← point stored at r
        if p is in Q return p
        else return
    for each child v of r do kdTree::range-search(v, Q)
```

- We assume again that each node stores its associated region.
- To save space, we could instead pass the region as a parameter and compute the region for each child using the splitting line.

kd-tree: Range Search Example



kd-tree: Range Search Example



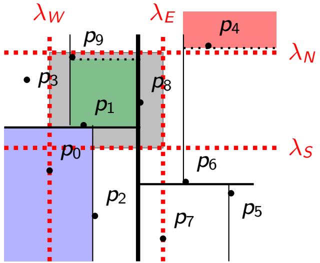
Red: Search stopped due to $R \cap Q = \emptyset$. Green: Search stopped due to $R \subseteq Q$.

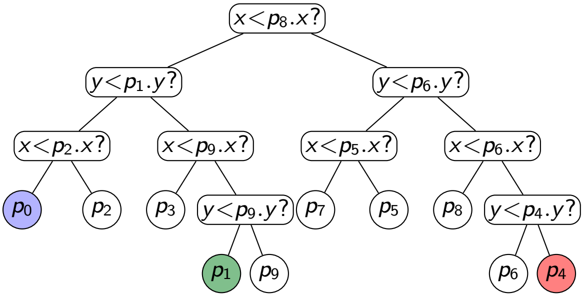
kd-tree: Range Search Complexity

- 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 is the output-size)
- **Observe**: $\#$ visited nodes is $O(\beta(n))$
  where $\beta(n)$ is the number of "boundary" nodes (blue):
  - *kdTree::range-search* was called.
  - Neither $R \subseteq Q$ nor $R \cap Q = \emptyset$
- **We will show:** $\beta(n) \in O(\sqrt{n})$
- Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$

Boundary nodes in kd-trees

Goal: The number of boundary-nodes satisfies $\beta(n) \in O(\sqrt{n})$.

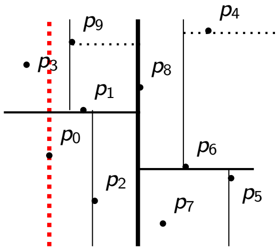




Observation: If z is a boundary-node, then its associated region intersects one of the lines $\lambda_W, \lambda_N, \lambda_E, \lambda_S$ that support the query-rectangle.

Boundary nodes in kd-trees

$$\beta(n,\lambda) := \max_{\text{kd-trees with } n \text{ points}} \left\{ \begin{array}{l} \text{number of associated regions} \\ \text{that intersect a given line } \lambda \end{array} \right\}$$

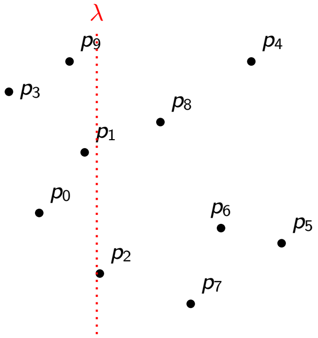


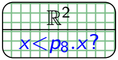
$$\beta_{ver}(n) := \max_{\text{vertical lines } \lambda} \beta_{ver}(n,\lambda) \qquad \beta_{hor}(n) := \max_{\text{horizontal lines } \lambda} \beta_{hor}(n,\lambda)$$

$$\begin{aligned} \beta(n) &\leq \beta(n,\lambda_W) + \beta(n,\lambda_N) + \beta(n,\lambda_E) + \beta(n,\lambda_S) \\ &\leq 2\beta_{ver}(n) + 2\beta_{hor}(n) \end{aligned}$$

Boundary nodes in kd-trees

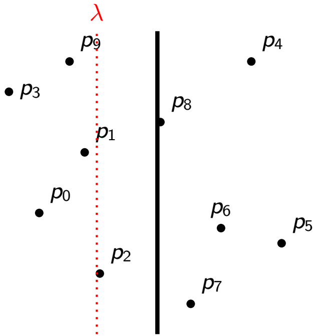
Goal: Recursive formula for $\beta_{ver}(n)$.

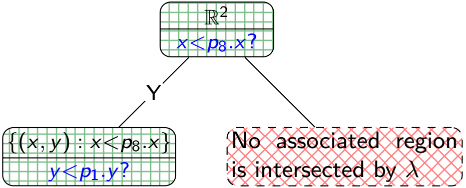




Boundary nodes in kd-trees

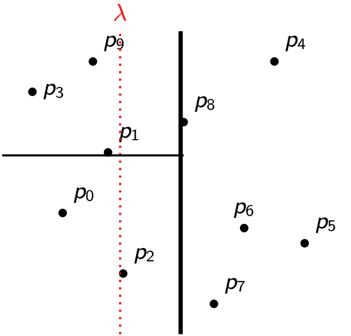
Goal: Recursive formula for $\beta_{ver}(n)$.

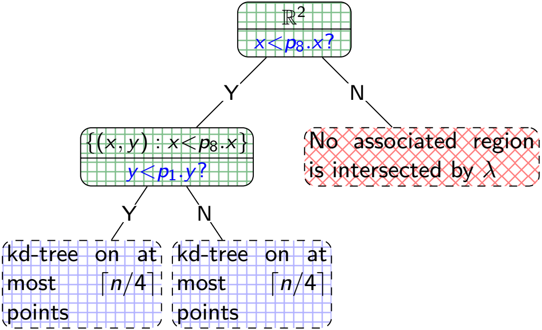




Boundary nodes in kd-trees

Goal: Recursive formula for $\beta_{ver}(n)$.





Boundary nodes in kd-trees

- $\beta_{ver}(n) \leq 2\beta_{ver}(n/4) + 2$ $\Rightarrow \beta_{ver}(n) \in O(\sqrt{n})$

Boundary nodes in kd-trees

- $\beta_{ver}(n) \leq 2\beta_{ver}(n/4) + 2 \quad \Rightarrow \beta_{ver}(n) \in O(\sqrt{n})$
- Similarly: $\beta_{hor}(n) \leq 2\beta_{hor}(n/4) + 3 \quad \Rightarrow \beta_{hor}(n) \in O(\sqrt{n})$

Boundary nodes in kd-trees

- $\beta_{ver}(n) \leq 2\beta_{ver}(n/4) + 2 \quad \Rightarrow \beta_{ver}(n) \in O(\sqrt{n})$
- Similarly: $\beta_{hor}(n) \leq 2\beta_{hor}(n/4) + 3 \quad \Rightarrow \beta_{hor}(n) \in O(\sqrt{n})$

- $\beta(n) \leq 2\beta_{ver}(n) + 2\beta_{hor}(n) \in O(\sqrt{n})$

Theorem: In a range-query in a kd-tree (of points in general position) there are $O(\sqrt{n})$ boundary-nodes.

Boundary nodes in kd-trees

- $\beta_{ver}(n) \le 2\beta_{ver}(n/4) + 2 \quad \Rightarrow \beta_{ver}(n) \in O(\sqrt{n})$
- Similarly: $\beta_{hor}(n) \le 2\beta_{hor}(n/4) + 3 \quad \Rightarrow \beta_{hor}(n) \in O(\sqrt{n})$

- $\beta(n) \le 2\beta_{ver}(n) + 2\beta_{hor}(n) \in O(\sqrt{n})$

Theorem: In a range-query in a kd-tree (of points in general position) there are $O(\sqrt{n})$ boundary-nodes.

- So range-search takes $O(\sqrt{n} + s)$ time.
- Note: It is *crucial* that we have $\approx n/4$ points in each grand-child of the root.

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

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.

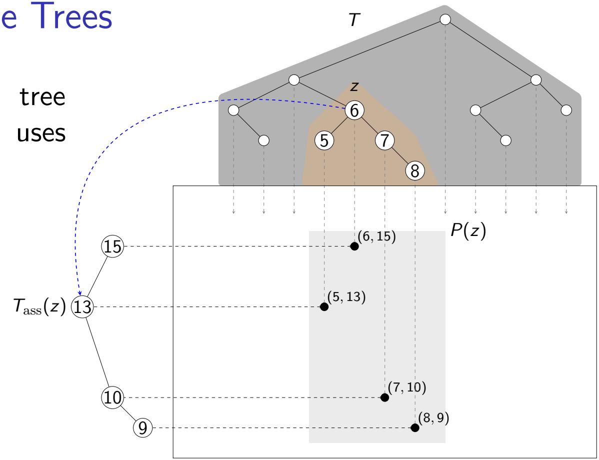
New idea: **Range trees**

- **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:

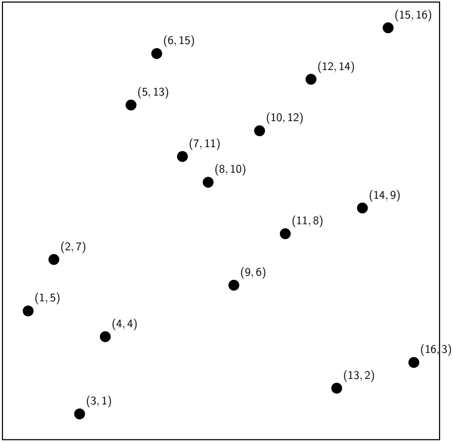
Balanced binary search tree T that stores P and uses *x-coordinates* as keys.



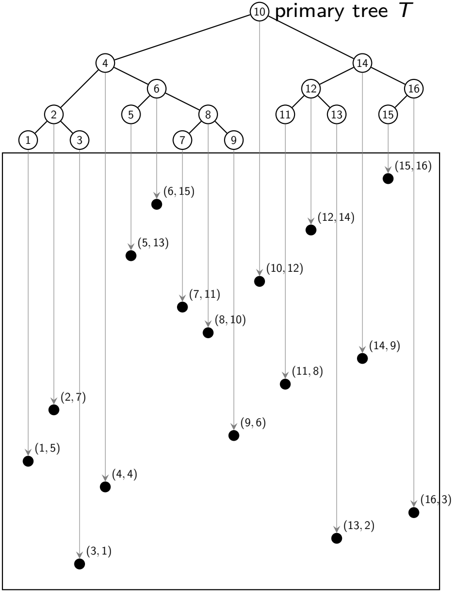
Every node z of T stores an **associate structure** $T_{\mathrm{ass}}(z)$:

- Let $P(z)$ be all points in subtree of z in T (including point at z)
- $T_{\mathrm{ass}}(z)$ 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 $T_{\mathrm{ass}}(z)$

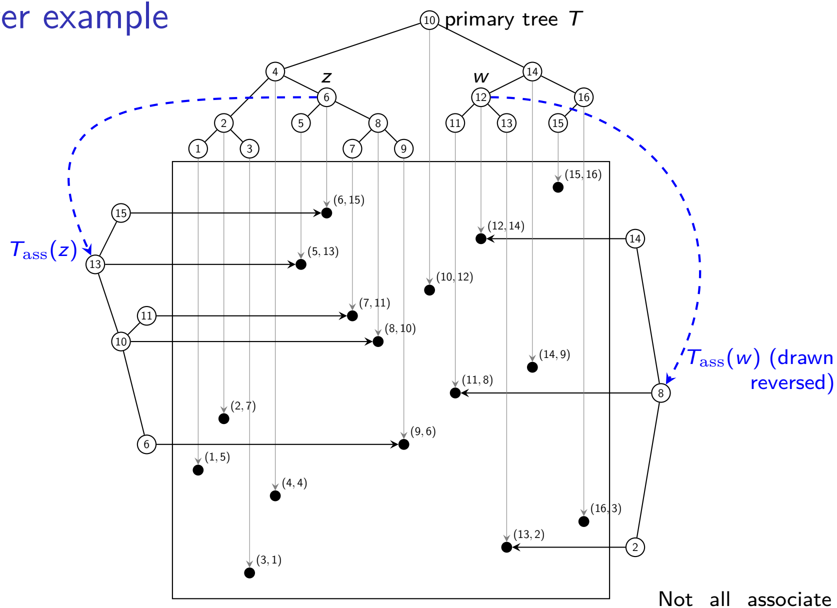
Bigger example



Bigger example



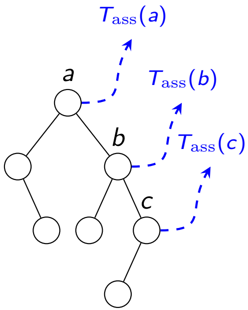
Bigger example



Not all associate trees are shown.

Range Tree Space Analysis

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



  - point of a is only in associate tree $T_{\text{ass}}(a)$
  - point of b is in associate trees $T_{\text{ass}}(a)$, $T_{\text{ass}}(b)$
  - point of c is in associate trees $T_{\text{ass}}(a)$, $T_{\text{ass}}(b)$, $T_{\text{ass}}(c)$
  - **Key insight**: point of z is in associate tree $T_{\text{ass}}(u)$ if and only if u is an ancestor of z in T
  - 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.

This is tight for some primary trees.

Range Trees 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.

  Problem: We want the binary search trees to be balanced.
  - This makes *insert*/*delete* very slow if we use AVL-trees. (A rotation at z changes $P(z)$, so requires a re-build of $T_{\text{ass}}(z)$.)
  - **Solution**: Use Scapegoat trees! (No rotations.)
  - Run-time for *insert*/*delete* becomes $O(\log^2 n)$ amortized.

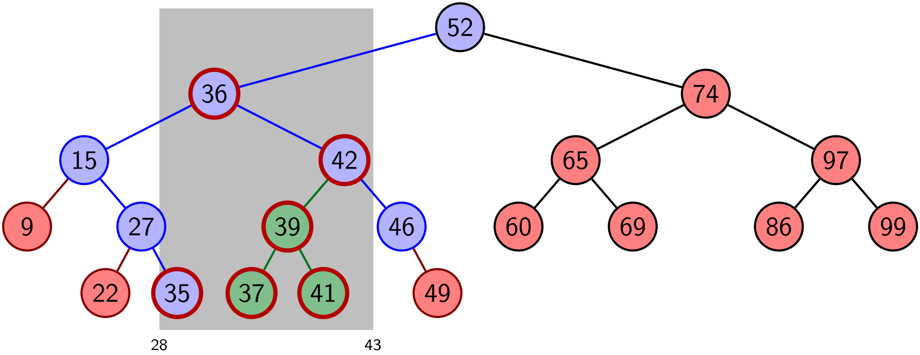
Range Trees Operations

- *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.

  Problem: We want the binary search trees to be balanced.
  - This makes *insert*/*delete* very slow if we use AVL-trees. (A rotation at z changes $P(z)$, so requires a re-build of $T_{\text{ass}}(z)$.)
  - **Solution**: Use Scapegoat trees! (No rotations.)
  - Run-time for *insert*/*delete* becomes $O(\log^2 n)$ amortized.

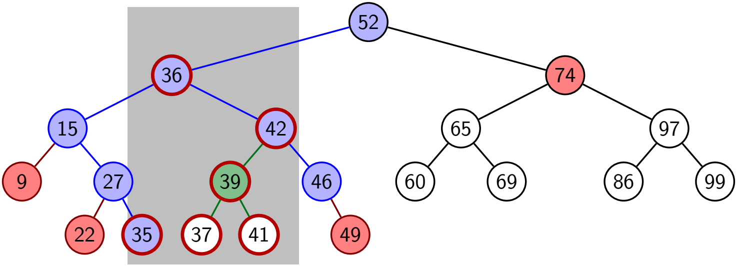
- *range-search*: search by x-range in T. Among found points, search by y-range in some associated trees.
- Must understand first: How to do (1-dimensional) range search in binary search tree?

BST Range Search



- Search for left boundary x_1: this gives path P_1
- Search for right boundary x_2: this gives path P_2
- Three types of nodes: outside, on, or between the paths.
- This classification will be crucial later!

BST Range Search re-phrased

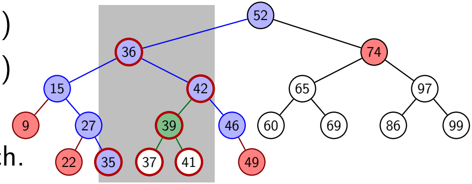


- boundary nodes: nodes in P_1 or P_2
  - For each boundary node, test whether it is in the range.
- outside nodes: nodes that are left of P_1 or right of P_2
  - These are *not* in the range, we do not search in them.
- inside nodes: nodes that are right of P_1 and left of P_2
  - We keep a list of the topmost inside nodes.
  - *All* descendants of such a node are in the range.
    For a 1d range search, report all of them.

BST Range Search analysis

Assume that the binary search tree is balanced:

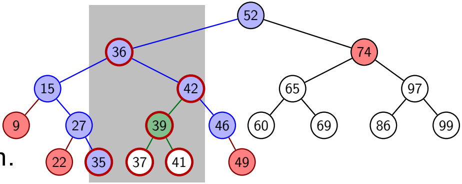
- Search for path P_1: $O(\log n)$
- Search for path P_2: $O(\log n)$
- $O(\log n)$ boundary nodes
- We spend $O(1)$ time on each.



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
- We spend $O(1)$ time on each.



- We spend $O(1)$ time per topmost inside node v.
  - They are children of boundary nodes, so this takes $O(\log n)$ time.
- For 1d range search, also report the descendants of v.
  - We have $\sum_{z \text{ topmost inside}} \#\{\text{descendants of } z\} \leq s$ since subtrees of topmost inside nodes are disjoint. So this takes time $O(s)$ overall.

Run-time for 1d range search: $O(\log n + s)$.

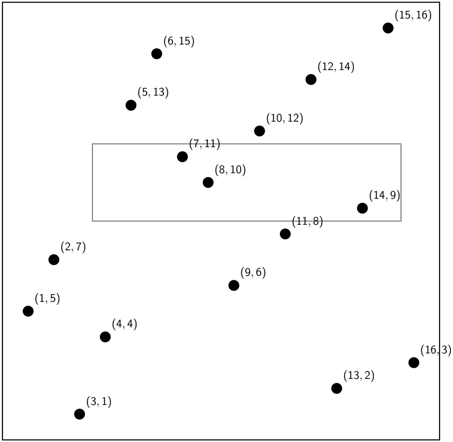
The ability to report the topmost inside nodes will be important for 2d range search.

Range Trees: Range Search

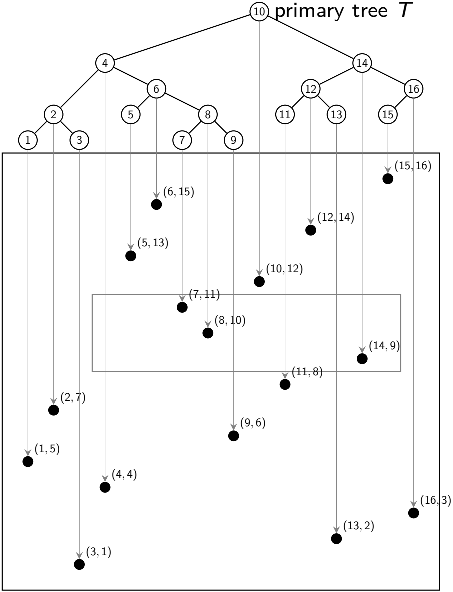
Range search for $Q = [x_1, x_2] \times [y_1, y_2]$ is a two stage process:

- 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))
- Get boundary and topmost inside nodes as before.
- For every boundary node, test to see if the corresponding point is within the region Q.
- For every topmost inside node v:
  - 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 $T_{\mathrm{ass}}(z)$.
  - To find points in $P(z)$ where the y-cordinates are within range as well, perform a range search in $T_{\mathrm{ass}}(z)$: *BST::range-search*$(T_{\mathrm{ass}}(z), y_1, y_2)$

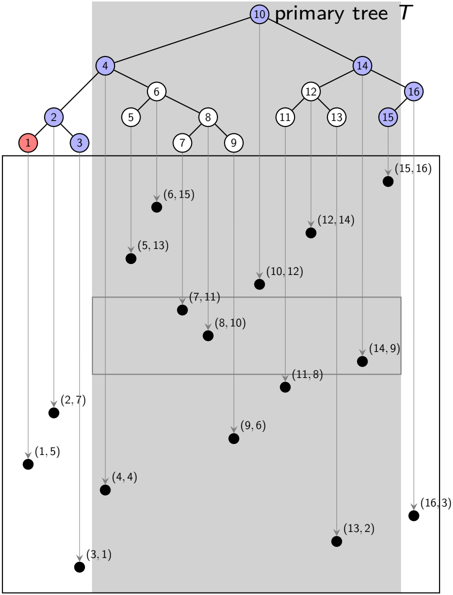
Range tree range search example



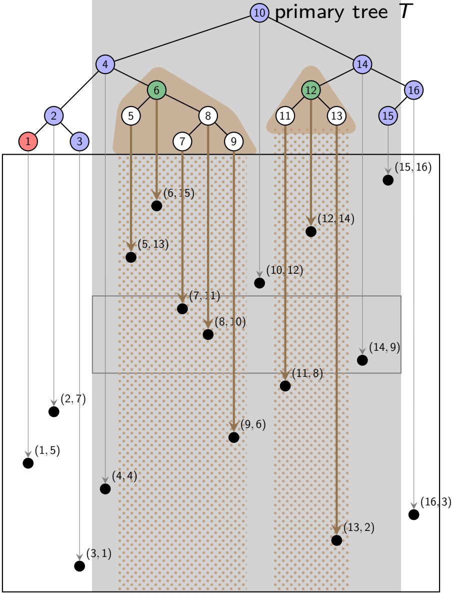
Range tree range search example



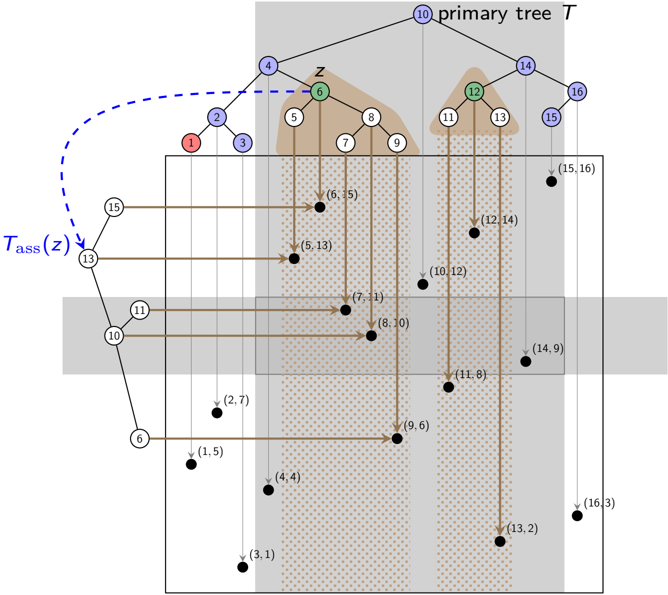
Range tree range search example



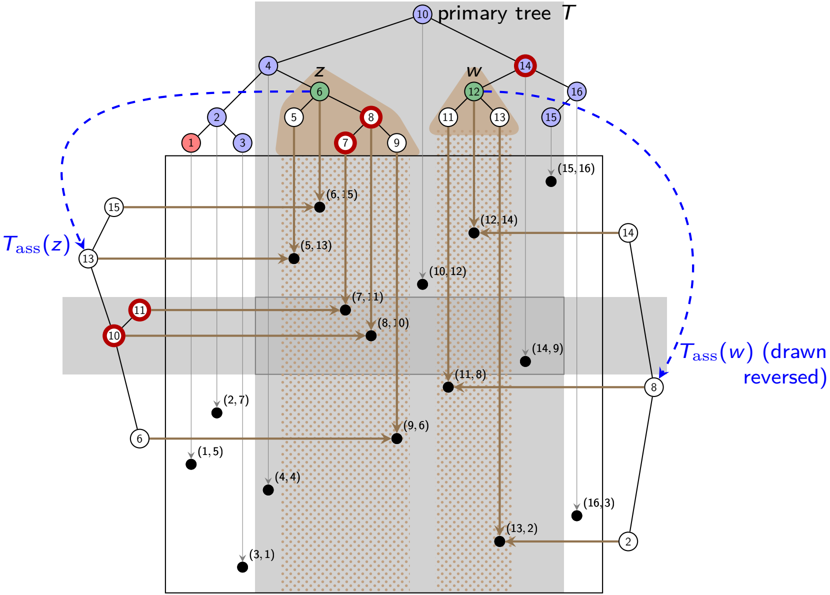
Range tree range search example



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
- Two topmost inside nodes have no common point in their trees
  $\Rightarrow$ every point is reported in at most one associate structure
  $\Rightarrow \sum_{z \text{ topmost inside}} s_z \leq s$

Time for range search in range-tree is proportional to

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

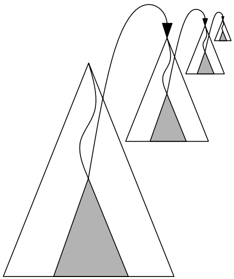
(There are ways to make this even faster. No details.)

Range Trees: Higher Dimensions

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

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

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



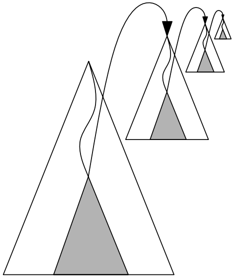
Range Trees: Higher Dimensions

- 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: $O(s + n^{1-1/d})$

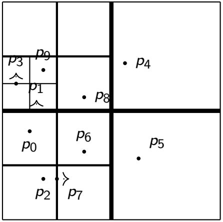
(Note: d is considered to be a constant.)

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

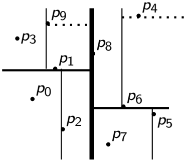


Range search data structures summary

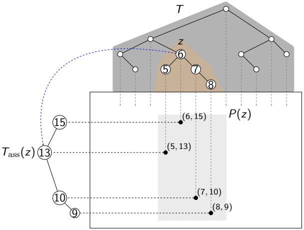
- Quadtrees
  - simple (also for dynamic set of points)
  - work well only if points evenly distributed
  - wastes space for higher dimensions



- 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.

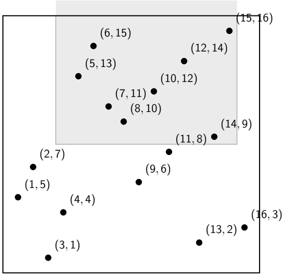
Outline

8 Range-Searching in Dictionaries for Points
- Range Searches
- Multi-Dimensional Data
- Quadtrees
- kd-Trees
- Range Trees
- 3-sided range search

3-sided range search

Consider a special kind of range-search:

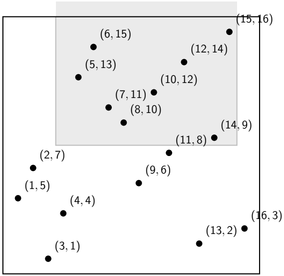
3sidedRangeSearch(x_1, x_2, y'): *return* (x, y) *with* $x_1 \leq x \leq x_2$ *and* $y \geq y'$.



3-sided range search

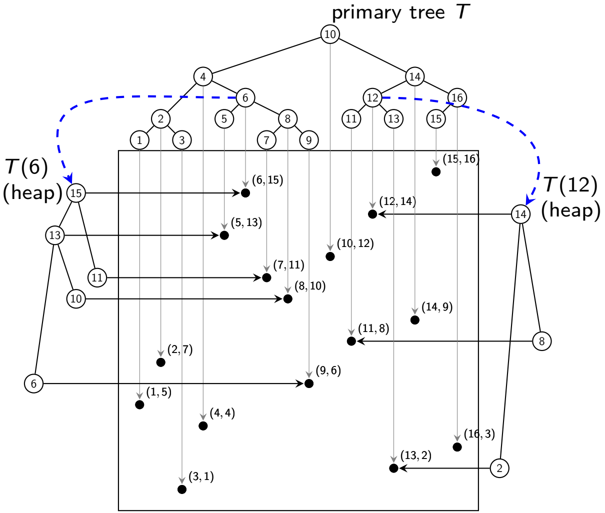
Consider a special kind of range-search:

3sidedRangeSearch(x_1, x_2, y'): *return* (x, y) *with* $x_1 \le x \le x_2$ *and* $y \ge y'$.



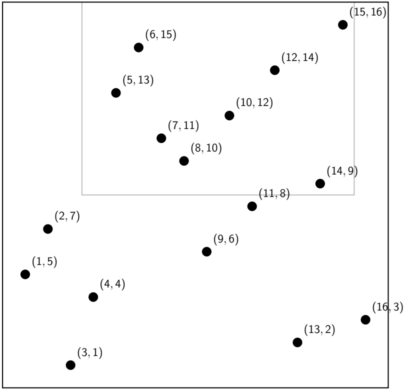
- We can do this with a range tree in $O(\log^2 n + s)$ with $\Theta(n \log n)$ space.
- Can we do this faster or using less space by adapting previous ideas to the special situation?

Idea 1: Associated heaps

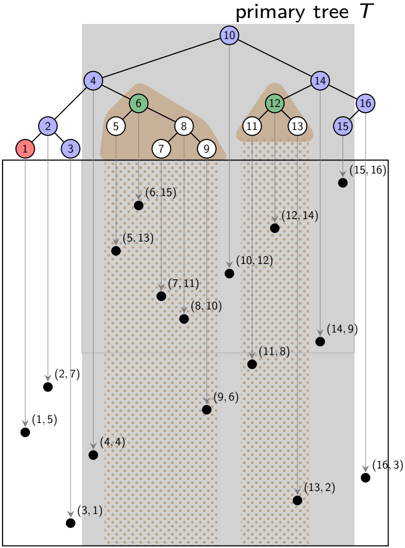


- Primary tree: balanced binary search tree.
- Associated tree: binary heap.
- Space: $\Theta(n \log n)$.
- Range-search time?

Idea 1: Associated heaps - 3-sided range search

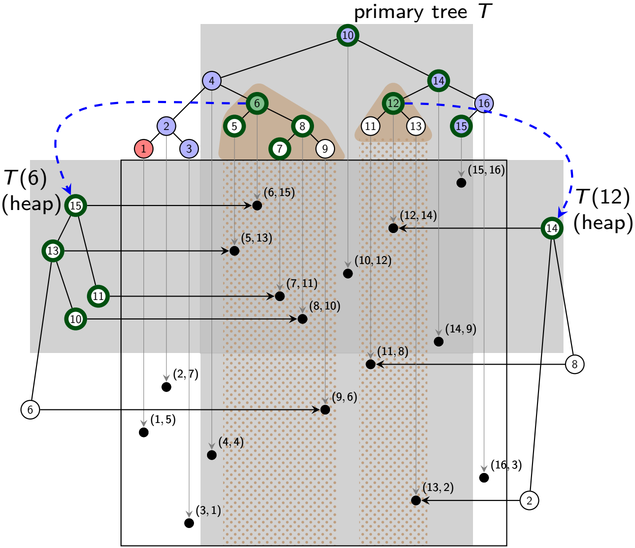


Idea 1: Associated heaps - 3-sided range search



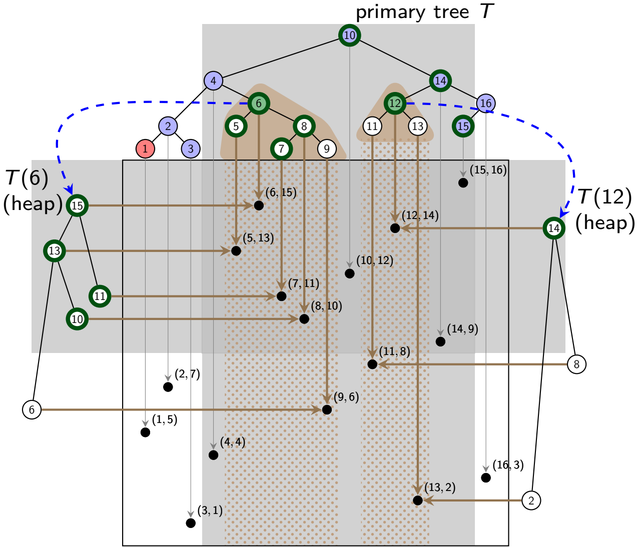
- Search in primary as before.

Idea 1: Associated heaps - 3-sided range search



- Search in primary as before.
- In associated heap: Search by y-coordinate in $O(1 + s)$ time. (Exercise.)

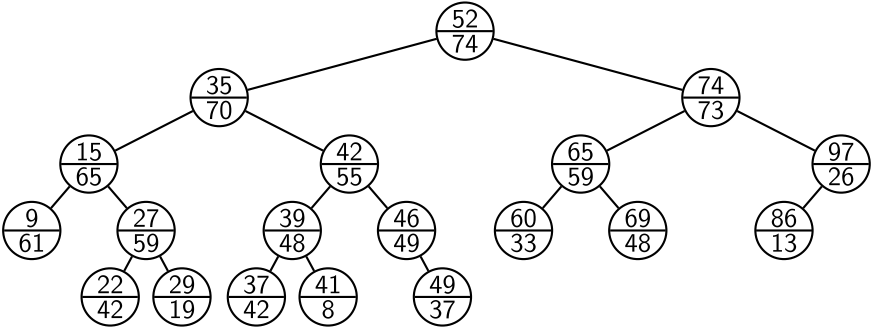
Idea 1: Associated heaps - 3-sided range search



- Search in primary as before.
- In associated heap: Search by y-coordinate in $O(1 + s)$ time. (Exercise.)
- Total time: $O(\log n + s)$
- But space is $\omega(n)$

Idea 2: Cartesian Trees

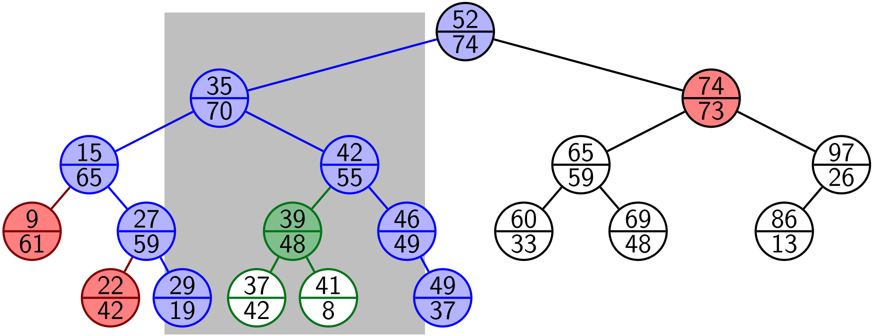
Recall: Treap = binary search tree (with respect to keys)
+ heap (with respect to priorities)



Cartesian tree: Use x-coordinate as key, y-coordinate as priority.
Space: $\Theta(n)$.

Idea 2: Cartesian Tree - 3-sided range search

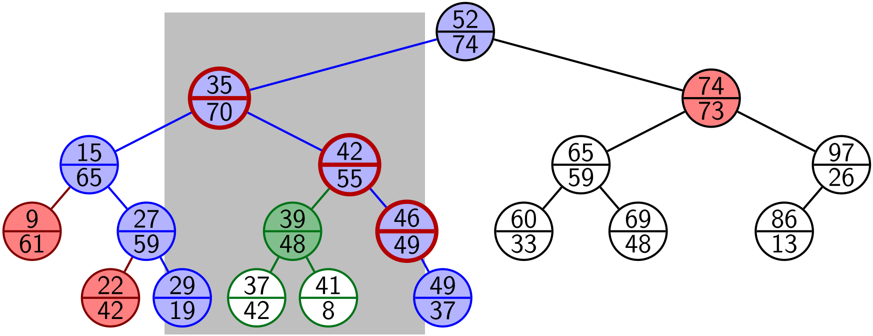
CartesianTree::3-sided-range-search$(T, 28, 47, 36)$:



- BST::range-search(x_1, x_2) to get boundary and topmost inside nodes.

Idea 2: Cartesian Tree - 3-sided range search

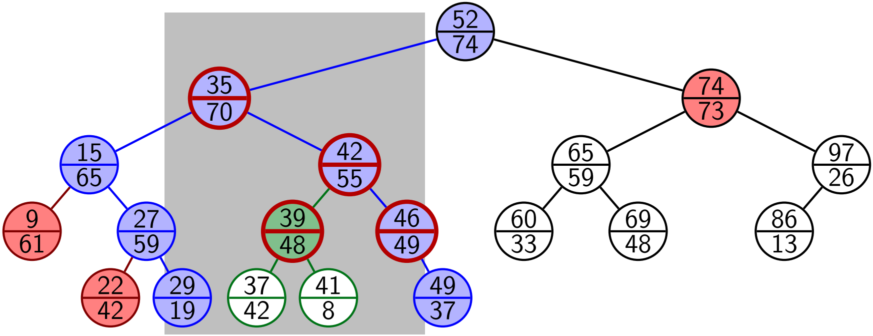
CartesianTree::3-sided-range-search$(T, 28, 47, 36)$:



- BST::range-search(x_1, x_2) to get boundary and topmost inside nodes.
- Boundary-nodes: Explicitly test whether in x-range and y-range.

Idea 2: Cartesian Tree - 3-sided range search

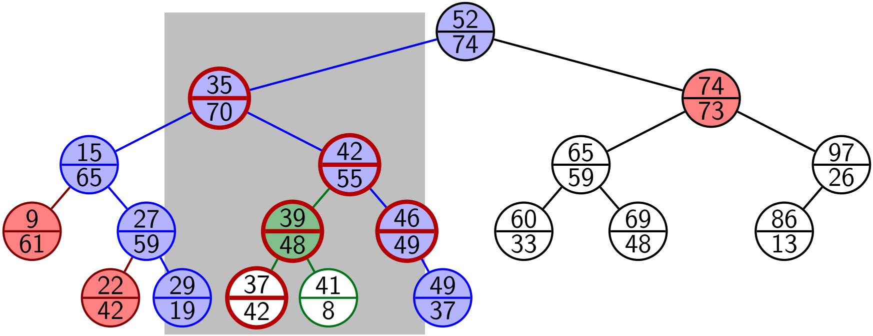
CartesianTree::3-sided-range-search$(T, 28, 47, 36)$:



- BST::range-search(x_1, x_2) to get boundary and topmost inside nodes.
- Boundary-nodes: Explicitly test whether in x-range and y-range.
- Topmost inside-nodes: If $y \geq y_1$, report and recurse in children.

Idea 2: Cartesian Tree - 3-sided range search

CartesianTree::3-sided-range-search$(T, 28, 47, 36)$:



- BST::range-search(x_1, x_2) to get boundary and topmost inside nodes.
- Boundary-nodes: Explicitly test whether in x-range and y-range.
- Topmost inside-nodes: If $y \geq y_1$, report and recurse in children.

Idea 2: Cartesian Tree - 3-sided range search

Run-time for 3-sided range search:

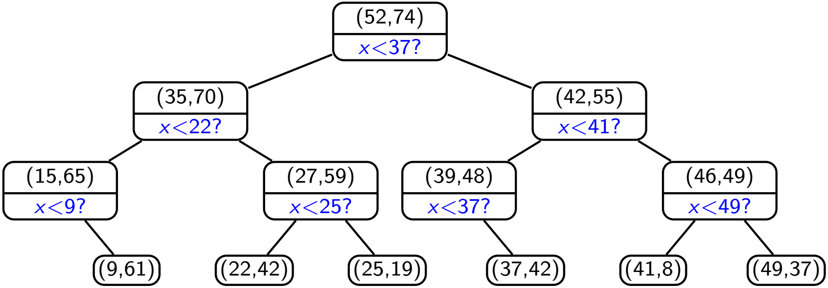
- BST::range-search(x_1, x_2) — $O(\textit{height})$ since we do not report points.
- Testing boundary-nodes: $O(\textit{height})$
- Testing heap: $O(1 + s_z)$ per topmost inside-node z

$\Rightarrow O(\textit{height} + s)$ run-time, $O(n)$ space

But: No guarantees on the height (not even in expectation) since we cannot choose priorities.

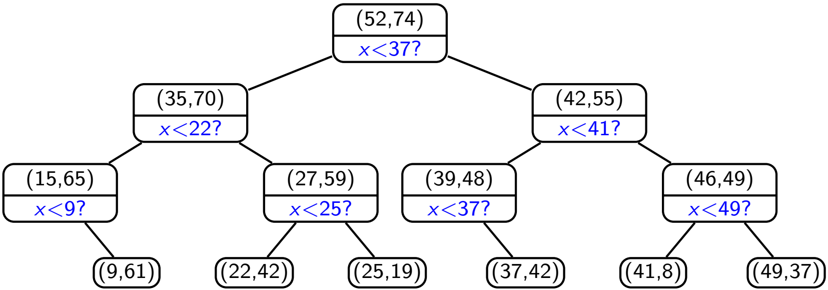
Idea 3: Priority search trees

- Design a new data structure
- Keep good aspects of Cartesian trees (store y-coordinates in heap-order)
- Keep good aspects of kd-tree (split in half by x-coordinate)



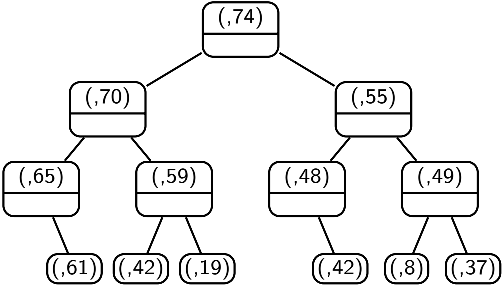
Key idea: The x-coordinate stored for splitting can be *different* from the x-coordinate of the stored point.

Idea 3: Priority search trees



- Every node z stores a point $p_z = (x_z, y_z)$,
  - y_z is the maximum y-coordinate in subtree
- Every non-leaf z stores an x-coordinate x'_z (split-line)
  - Every point p in left subtree has $p.x < x'_z$
  - Every point p in right subtree has $p.x \geq x'_z$
- x'_z is chosen so that tree is balanced $\Rightarrow$ height $O(\log n)$.

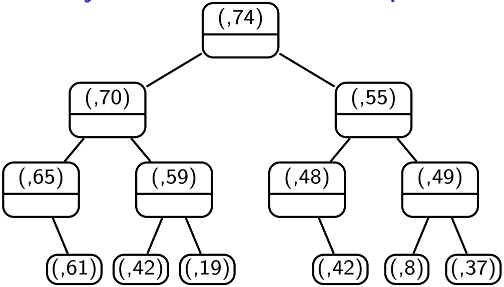
Priority search tree closeup



Looking only at *y*-coordinates:

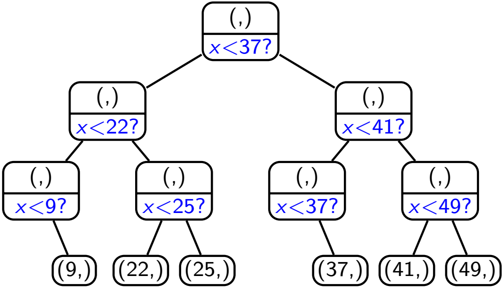
- Heap-order property
- But not heap-structure

Priority search tree closeup



Looking only at y-coordinates:

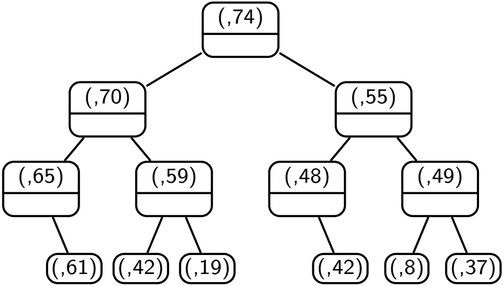
- Heap-order property
- But not heap-structure



Looking only at x-coordinates:

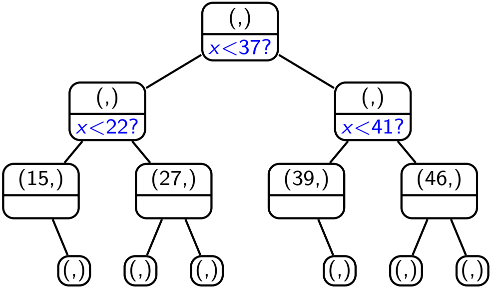
- Points at leaves ≈ kd-tree (1-dimensional)

Priority search tree closeup



Looking only at y-coordinates:

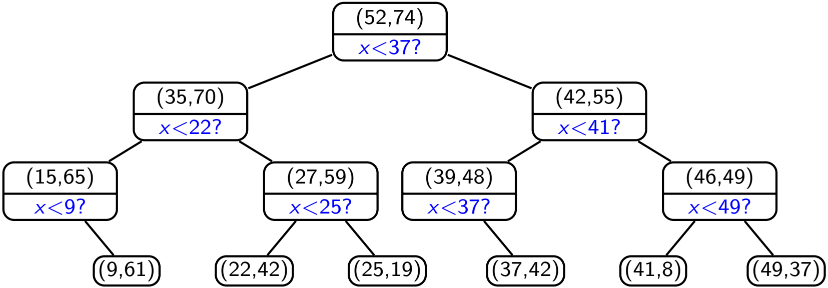
- Heap-order property
- But not heap-structure



Looking only at x-coordinates:

- Points at leaves $\approx$ kd-tree (1-dimensional)
- Points at level $\ell \approx$ kd-tree if we ignore points below.

Idea 3: Priority search trees



- Construction: $O(n \log n)$ time (exercise)
- *search*: $O(\log n)$ time
  - Get search-path by following split-lines, check all nodes on path
- *insert*, *delete*: Re-balancing is difficult, but can be done (no details).
- 3-sided range search: As for Cartesian trees, but height now $O(\log n)$.

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

3-sided range search summary

- Idea 1: Scapegoat tree + associated heaps
  $O(\log n + s)$ time for range search, but $\omega(n)$ space.
- Idea 2: Cartesian Tree
  $O(n)$ space, but range search takes $O(\mathit{height} + s)$, could be slow
- Idea 3: Priority search tree
  $O(n)$ space, $O(\log n + s)$ time for range search.

Sometimes it pays to design purpose-built data structures.