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

Module 8: Range-Searching in Dictionaries for Points

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version *2026-07-02 10:52*

Outline

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
			$\uparrow i$			$\uparrow i'$			

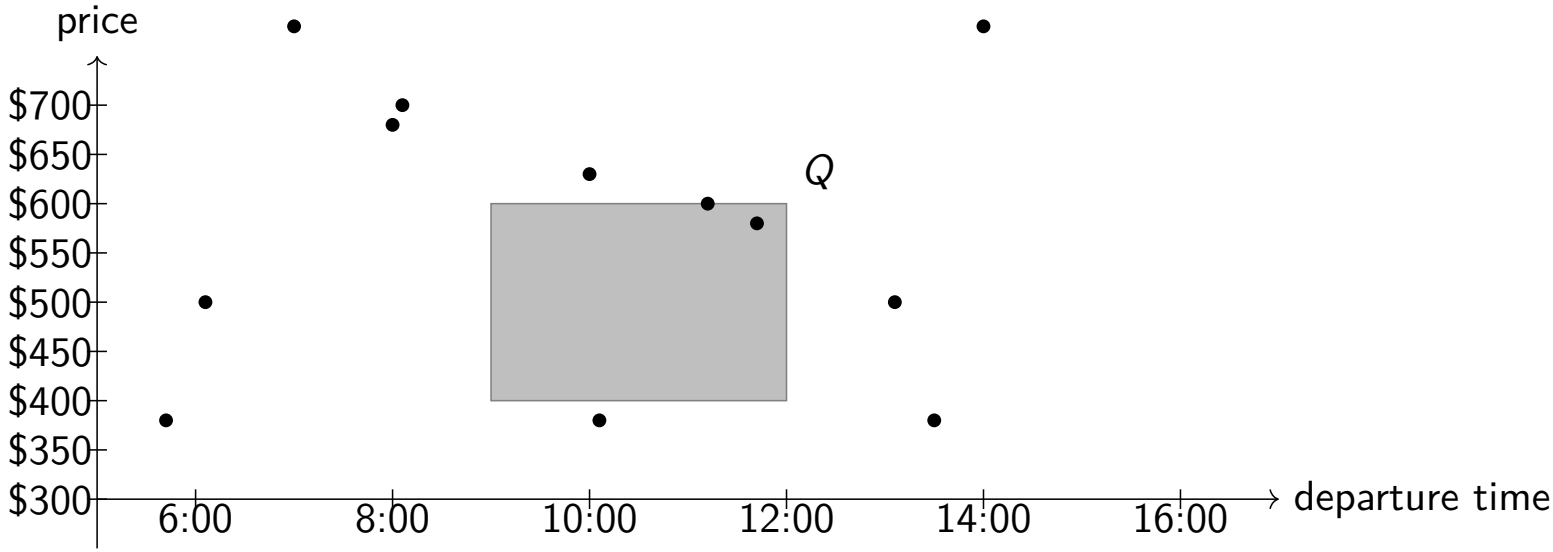
- Using binary search, find i such that x is at (or would be at) $A[i]$.
- Using binary search, find i' such that x' is at (or would be at) $A[i']$
- Report all items $A[i{+}1 \ldots i'{-}1]$
- Report $A[i]$ and $A[i']$ if they are in range

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

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:
  - 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.

Quadtrees

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

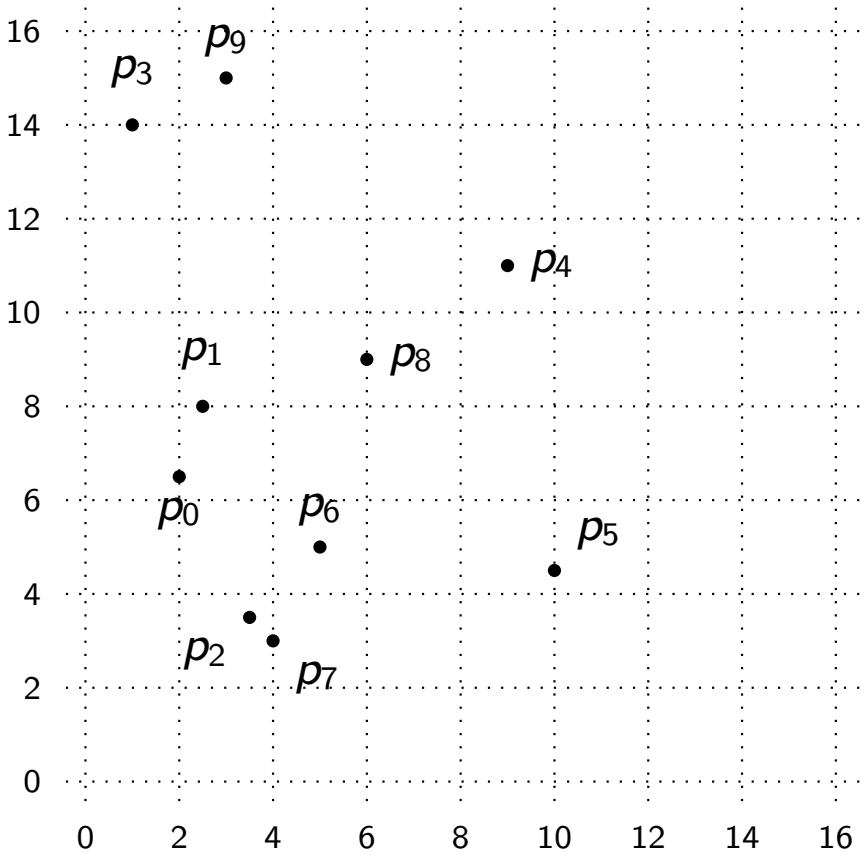
Find a **bounding box** $R = [0, 2^k) \times [0, 2^k)$: a square containing all points.

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

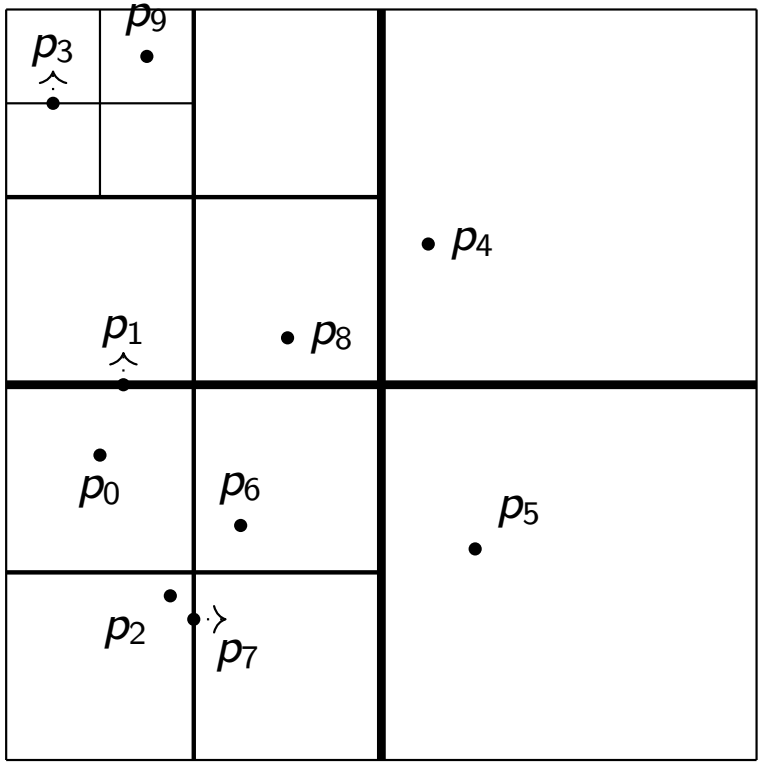
Structure (and also how to *build* the quadtree that stores P):

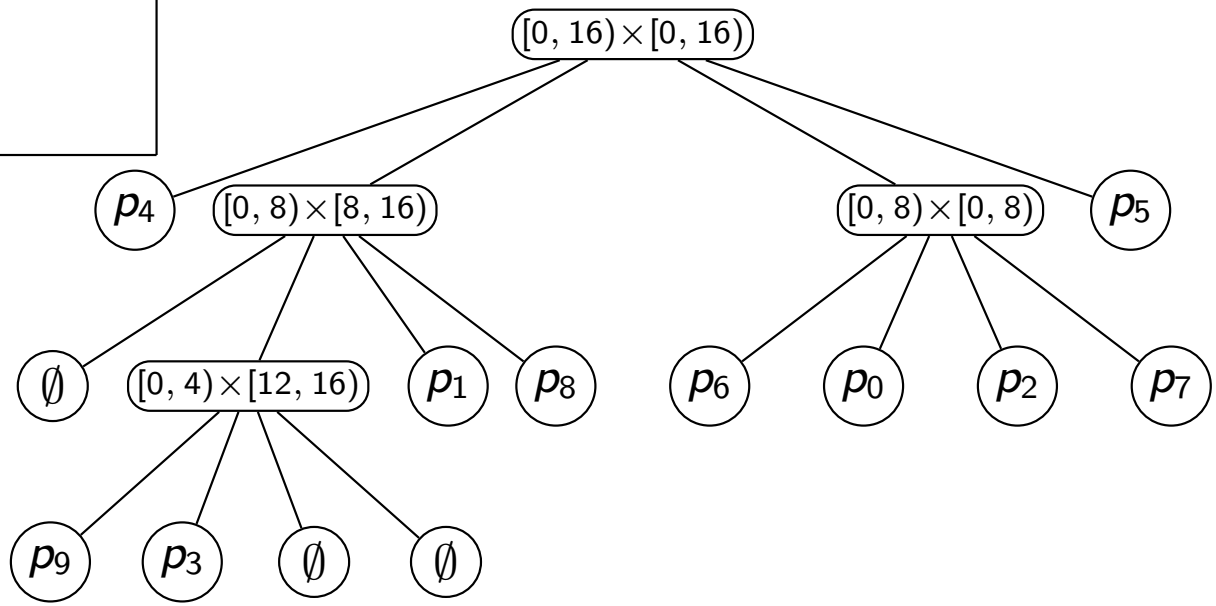
- Root r of the quadtree is associated with region R
- If R contains 0 or 1 points, then root r is a leaf that stores point.
- Else *split*: Partition R into four equal subsquares (**quadrants**) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
- Partition P into sets $P_{NE}, P_{NW}, P_{SW}, P_{SE}$ of points in these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

Quadtrees: Example



Quadtrees: Example

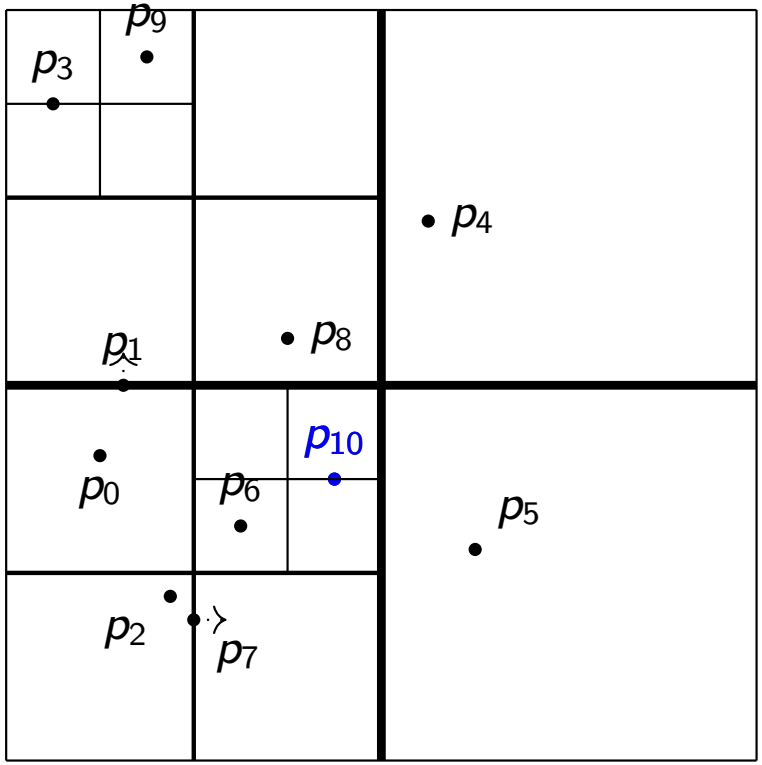




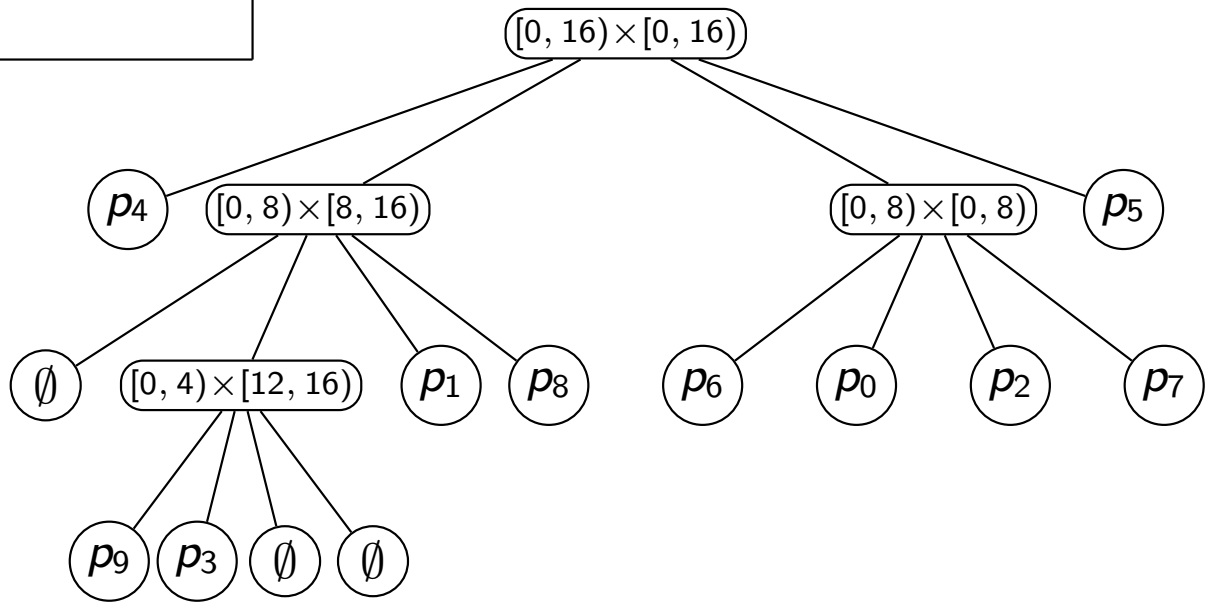
Quadtrees: Dictionary operations

- *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)

Quadtrees insertion: Example



insert(p_{10})



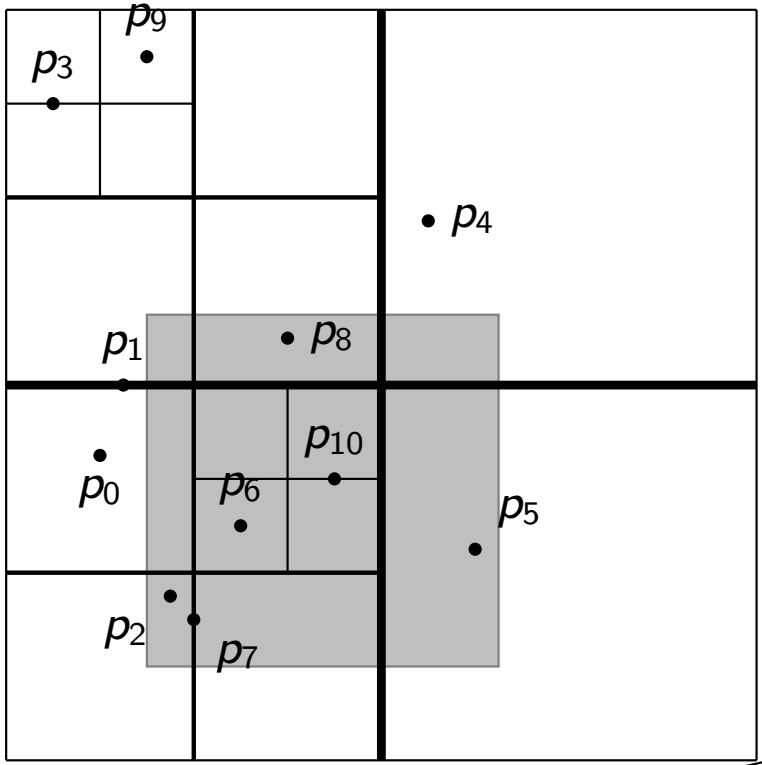
Quadtrees: Range search

QTree::range-search($r \leftarrow root, Q$)
r: The root of a quadtree, Q: Query-rectangle

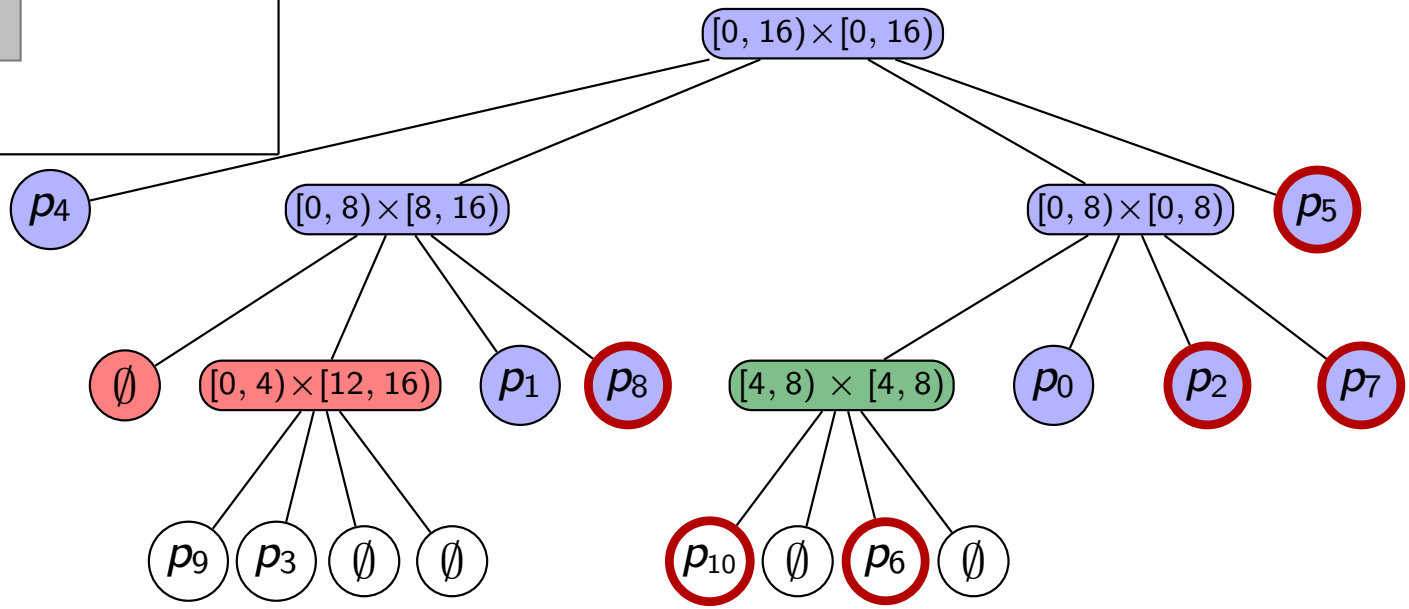
1. $R \leftarrow$ region associated with node r
2. **if** ($R \subseteq Q$) **then** // inside node, stop search
   report all points below r and **return**
3. **else if** ($R \cap Q$ is empty) **then return** // outside node, stop search
   // boundary node, recurse
4. **if** (r is a leaf) **then**
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



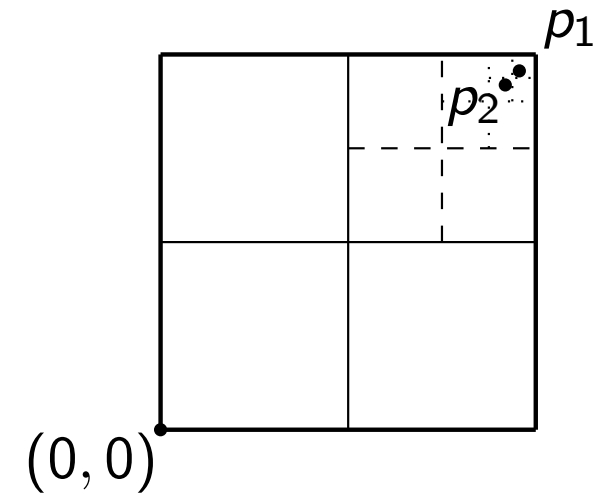
- Green: Search stopped due to $R \subseteq Q$.
- Red: Search stopped due to $R \cap Q = \emptyset$.
- Blue: Must continue search in children / evaluate.



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):

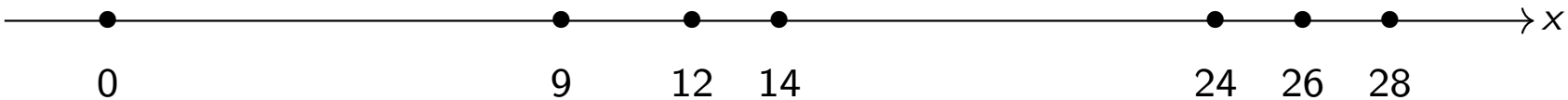
- For n randomly chosen points, the expected height is $O(\log n)$.
- The height depends on the **spread factor**:

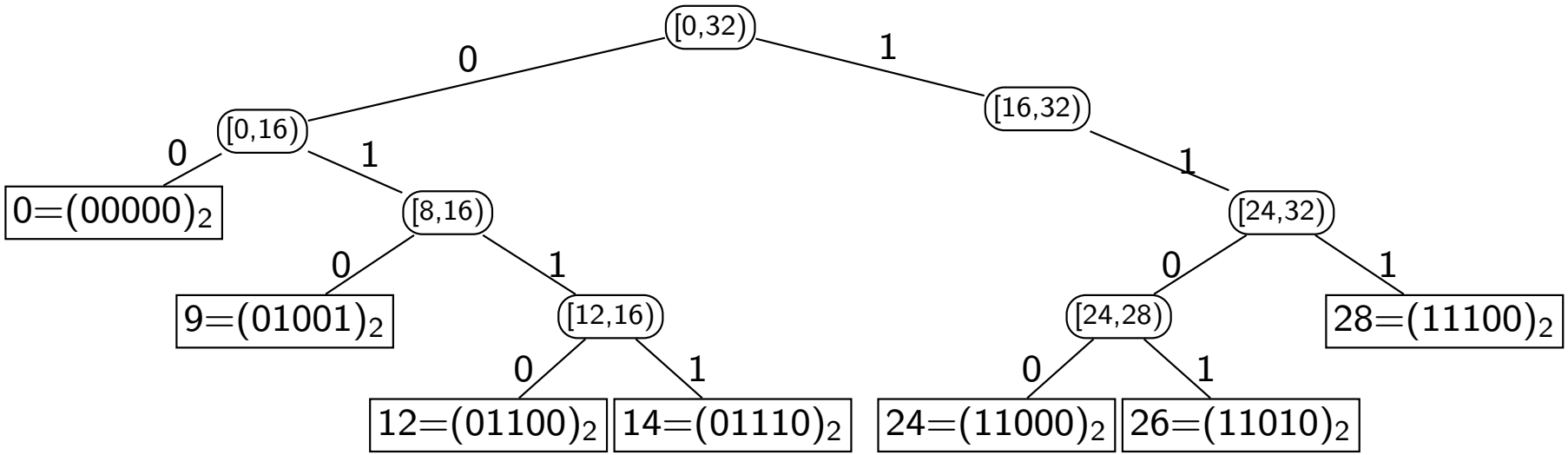
$$\frac{\text{sidelength of bounding box}}{\text{minimum distance between points in } P}$$

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

Quadtrees in other dimensions

- Quad-tree of 1-dimensional points:



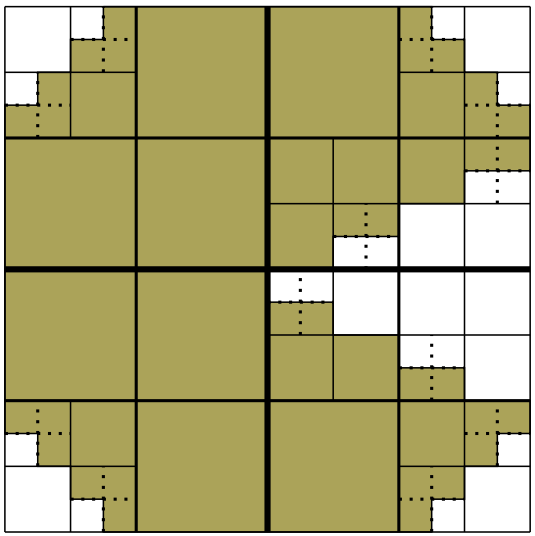


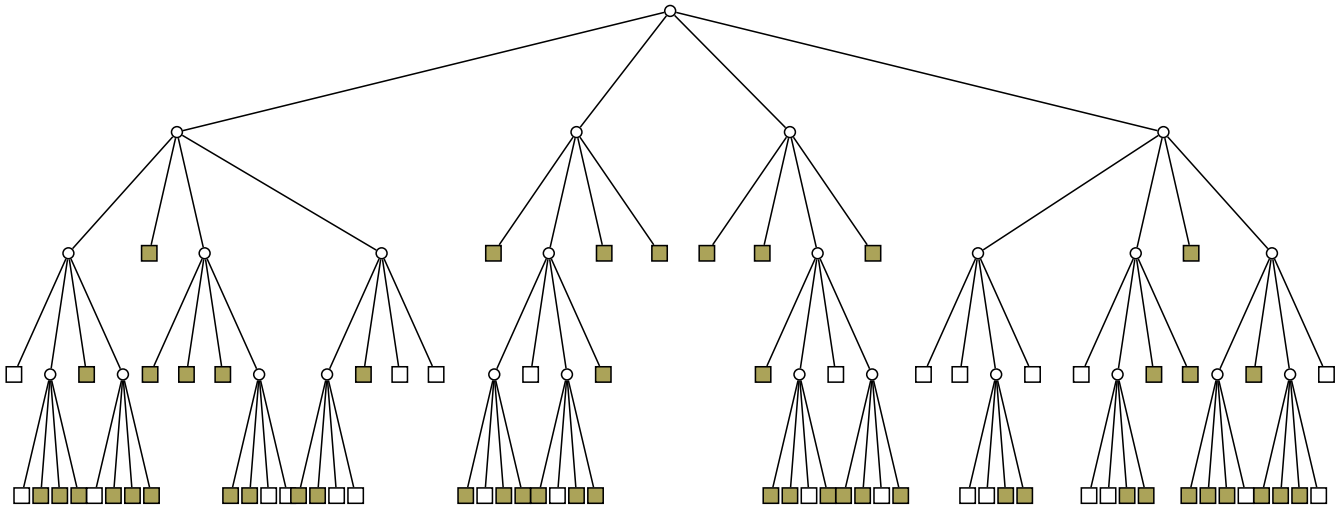
Same as a pruned trie

- Quadtrees also easily generalize to higher dimensions (split into octants $\rightarrow$ octrees, etc.) but are rarely used beyond dimension 3.

Quadtrees: 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.





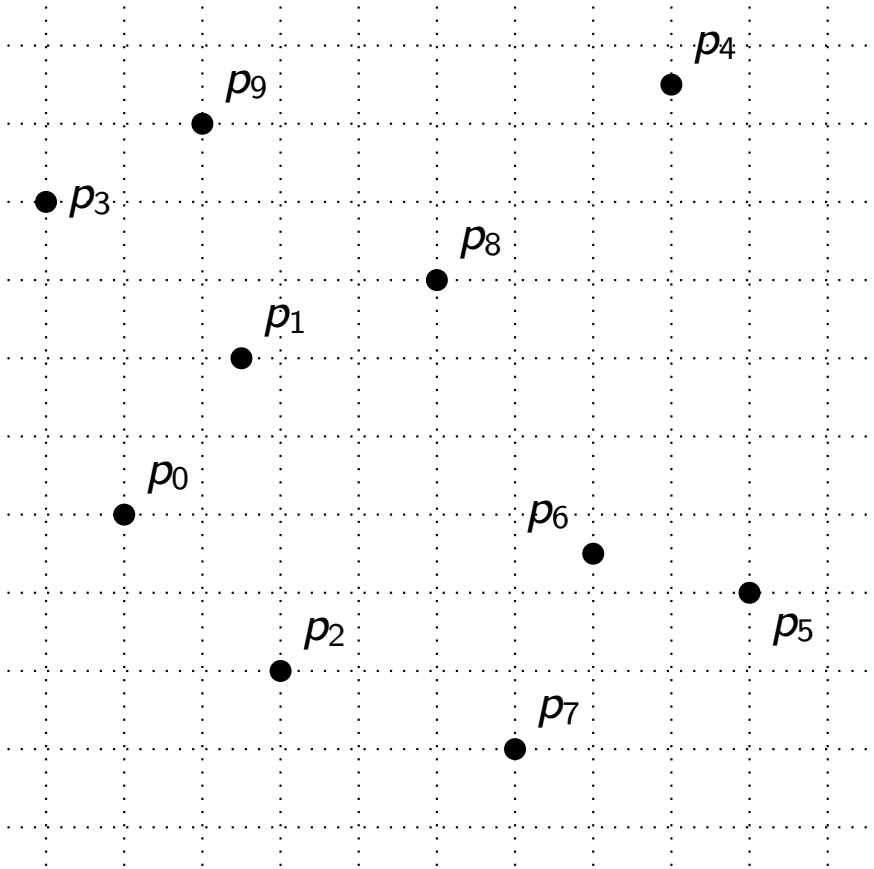
kd-trees

- 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

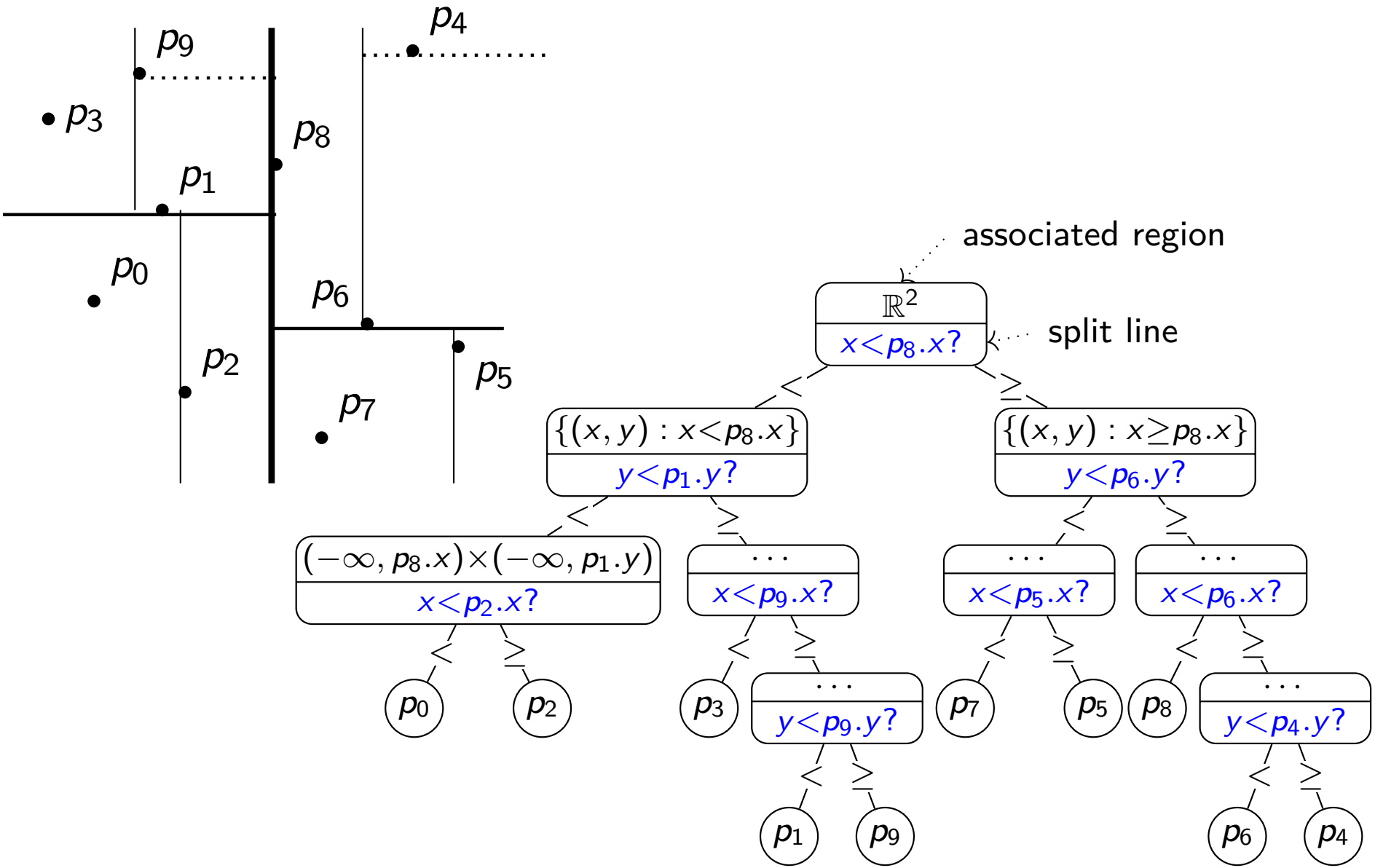
  $\rightsquigarrow$ roughly half of the point are in each subtree
- Each node of the kd-tree keeps track of a **splitting line** in one dimension (2D: either vertical or horizontal)
- **Convention:** Points on split lines belong to right/top side
- Continue splitting, switching between vertical and horizontal 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-trees: Example



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 $X :=$ *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.

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

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

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

Space: All interior nodes have exactly two children.

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

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 $\lceil \log_2 n \rceil$.

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.

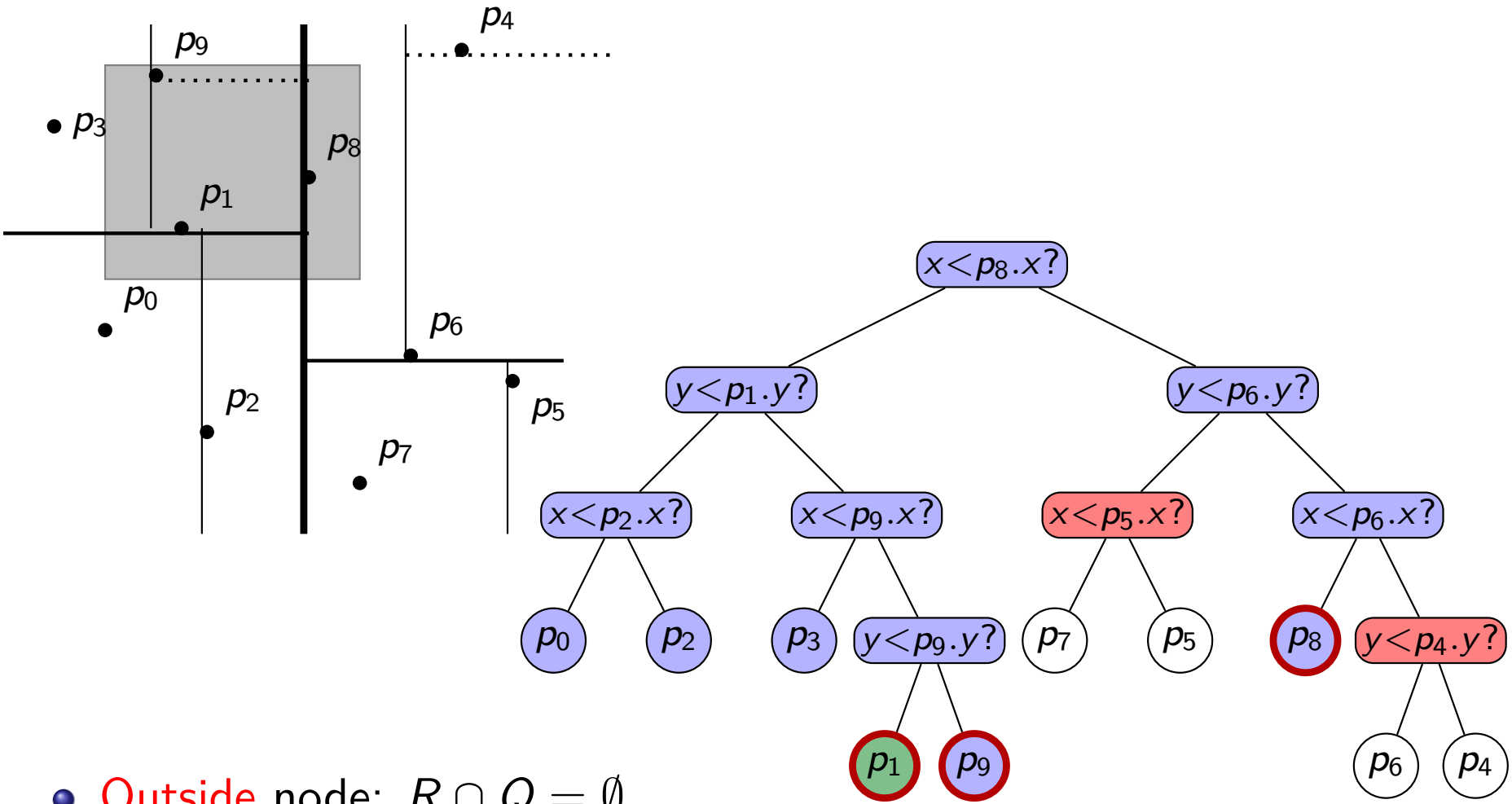
kd-trees: Range search

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

kdTree::range-search$(r \leftarrow root, Q)$

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** *QTree::range-search*(v, Q)

kd-tree range search: Example

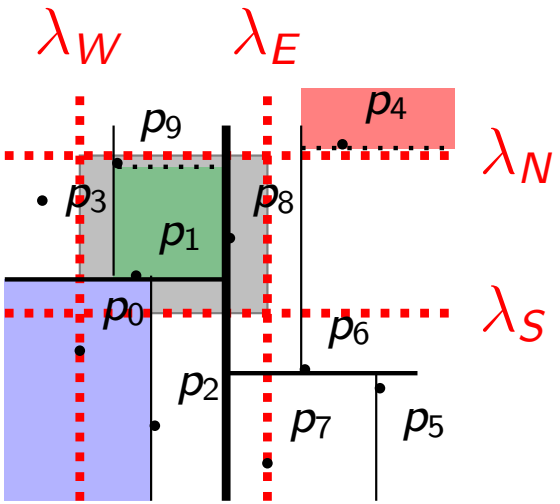


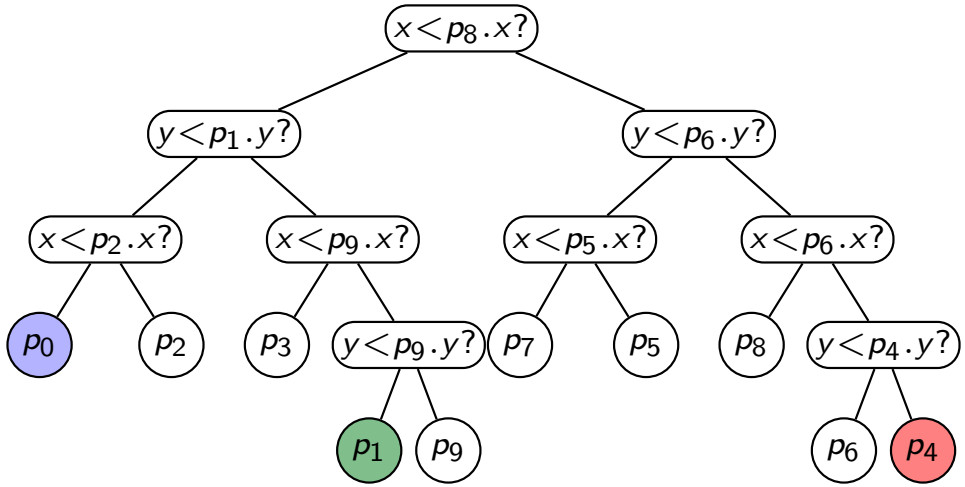
- Outside node: $R \cap Q = \emptyset$.
- Inside node: $R \subseteq Q$.
- Boundary node: Neither of the above.

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: $s =$ 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**: $\#$ 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.
- **We will show:** $\beta(n) \in O(\sqrt{n})$
- Therefore, the complexity of range search in kd-trees is $O(s + \sqrt{n})$

kd-trees range search: Boundary nodes





- Define "tic-tac-toe grid" with support-lines $\lambda_W, \lambda_N, \lambda_E, \lambda_S$ of Q
- Consider a node z with associated region R_z
- If R_z lies inside one of the grid-cells, then z is red or green.

$\Rightarrow$ For any boundary-node z, region R_z intersects at least one of $\lambda_W, \lambda_N, \lambda_E, \lambda_S$

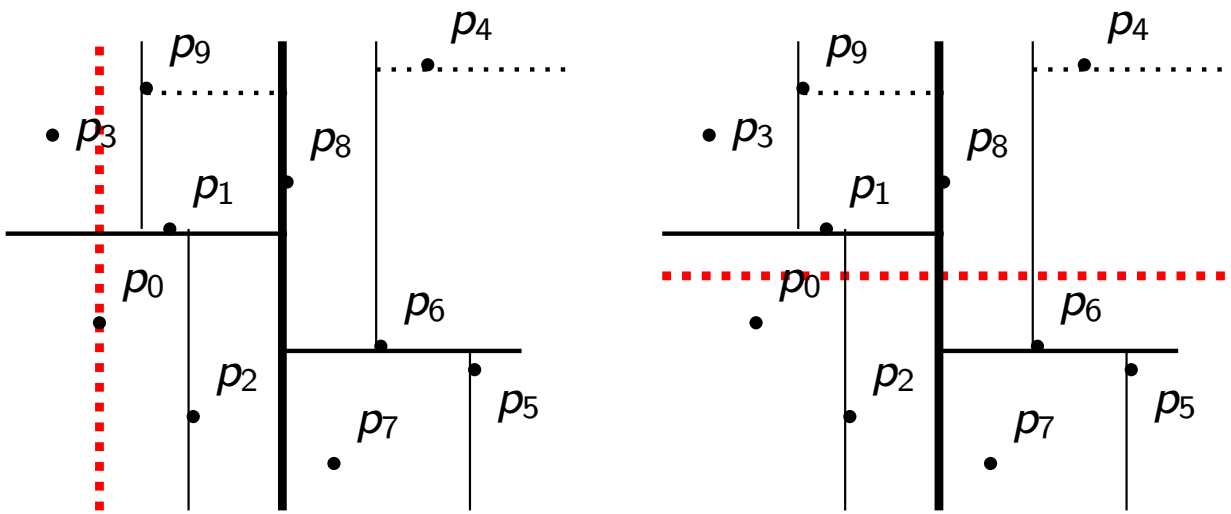
New goal: How many node-regions R_z intersect one of $\lambda_W, \lambda_N, \lambda_E, \lambda_S$?

kd-trees range search: Boundary nodes

So define $\beta(n, \lambda) := \max_{\text{kd-trees with } n \text{ points}} \left\{ \begin{matrix} \text{number of node-regions} \\ \text{that intersect a line } \lambda \end{matrix} \right\}$

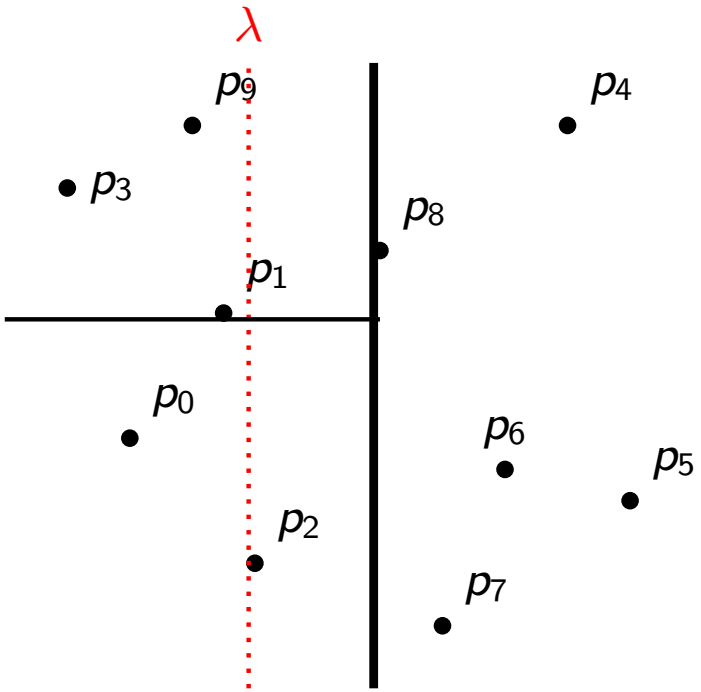
$$\begin{aligned} \text{then } \#\{\text{boundary nodes}\} &\leq \beta(n, \lambda_W) + \beta(n, \lambda_N) + \beta(n, \lambda_E) + \beta(n, \lambda_S) \\ &\leq 2\beta_{ver}(n) + 2\beta_{hor}(n) \end{aligned}$$

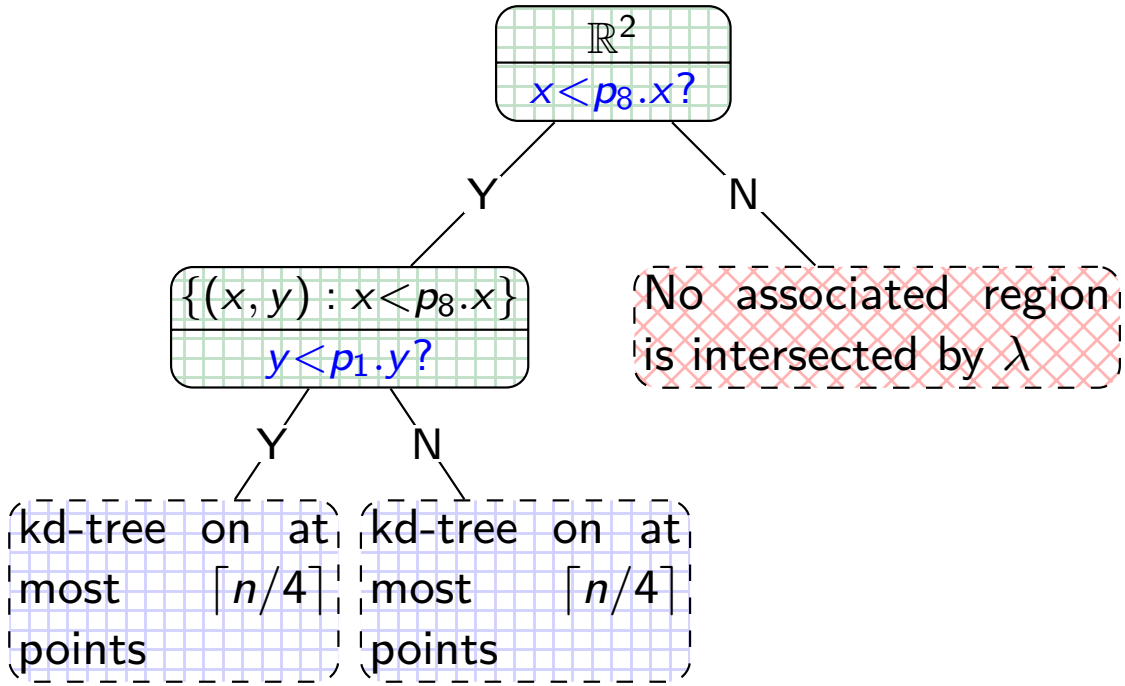
(where $\beta_{ver}(n), \beta_{hor}(n)$ take the maximum over vertical/horizontal lines)



kd-trees range search: Boundary nodes

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





kd-trees range search: Boundary nodes

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

- 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-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.

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.

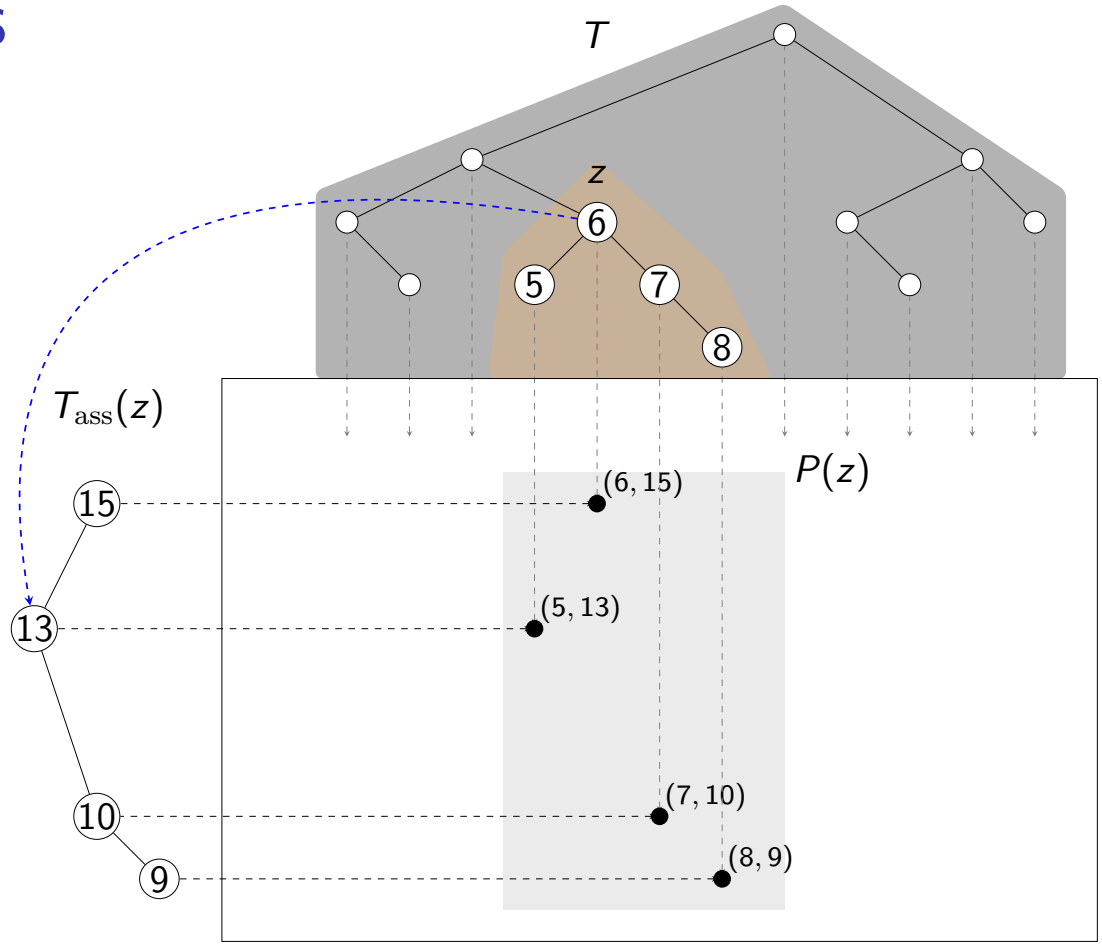
New idea: **Range trees**

- **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.

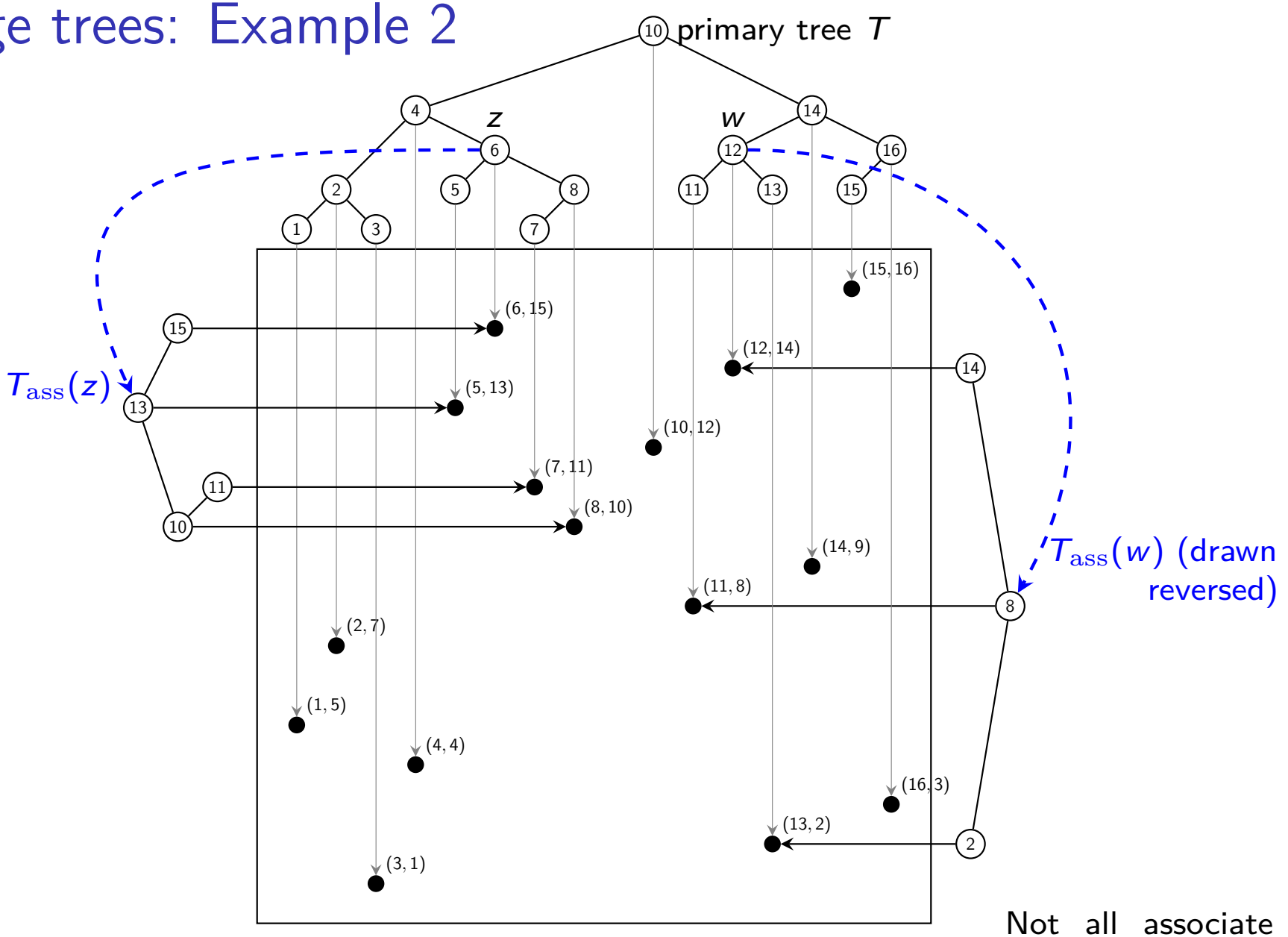
For each node z of T, let $P(z)$ be all points at descendants of z (including point at z)



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 $T_{\text{ass}}(z)$

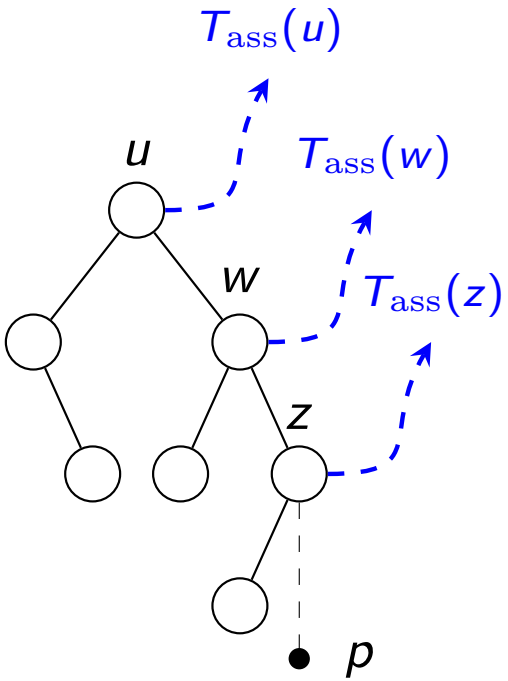
Range trees: Example 2



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?



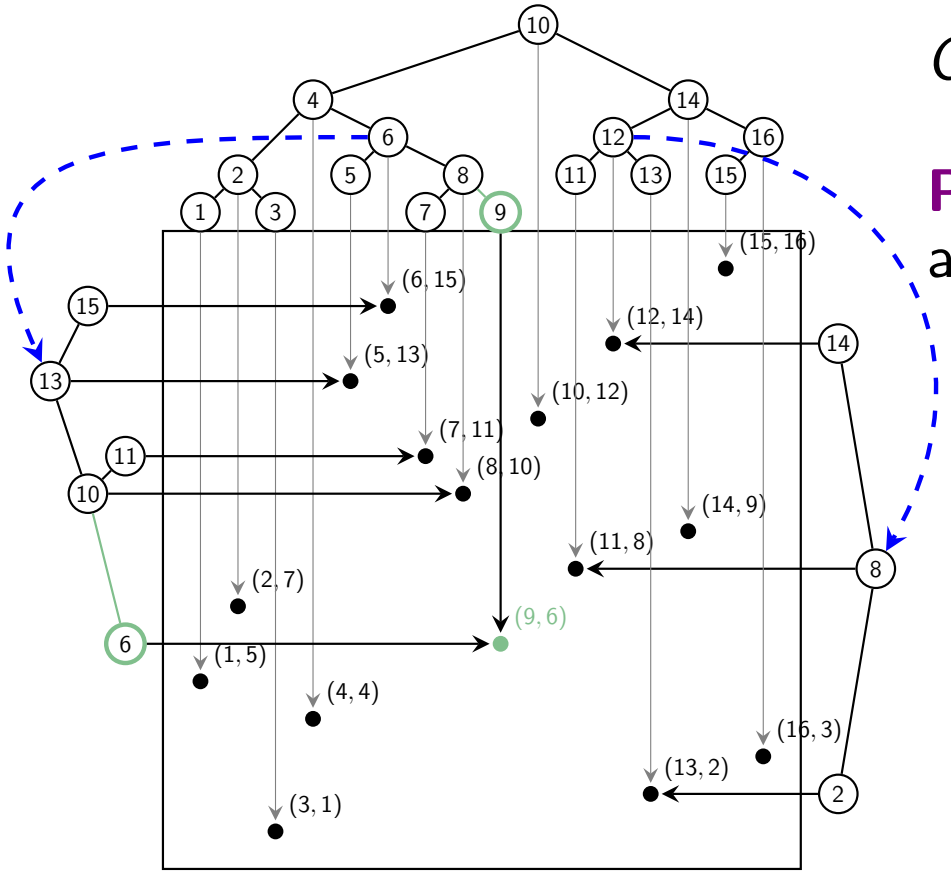
- Consider a point p stored at node z in T:
  - ★ $T_{\text{ass}}(z)$ stores p
  - ★ $T_{\text{ass}}(\mathit{parent}(z))$ stores p
  - ★ $T_{\text{ass}}(\mathit{parent}(\mathit{parent}(z)))$ stores p, etc.
  - ★ No other associate trees store p
- **Key insight**: p is in associate tree $T_{\text{ass}}(u)$ 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.

This is tight for some primary trees.

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**: Use Scapegoat trees! (No rotations.)
- Run-time for *insert*/*delete* becomes $O(\log^2 n)$ amortized.

Range trees: Range search

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

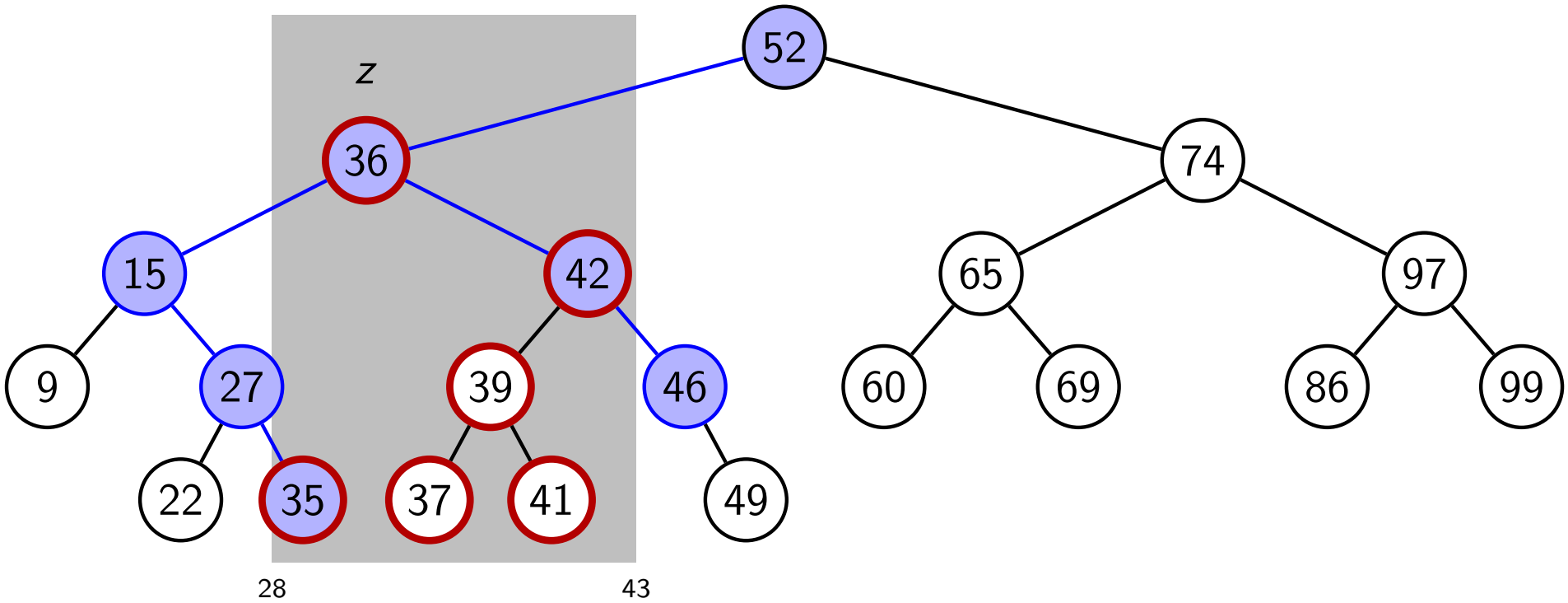
- Perform a range search (on the x-coordinates) for the interval $[x_1, x_2]$ in primary tree T but do **not** report all points.
- Instead, return $O(\log n)$ nodes of T with special properties.
- Use these to determine the points in range, using the associated trees.

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

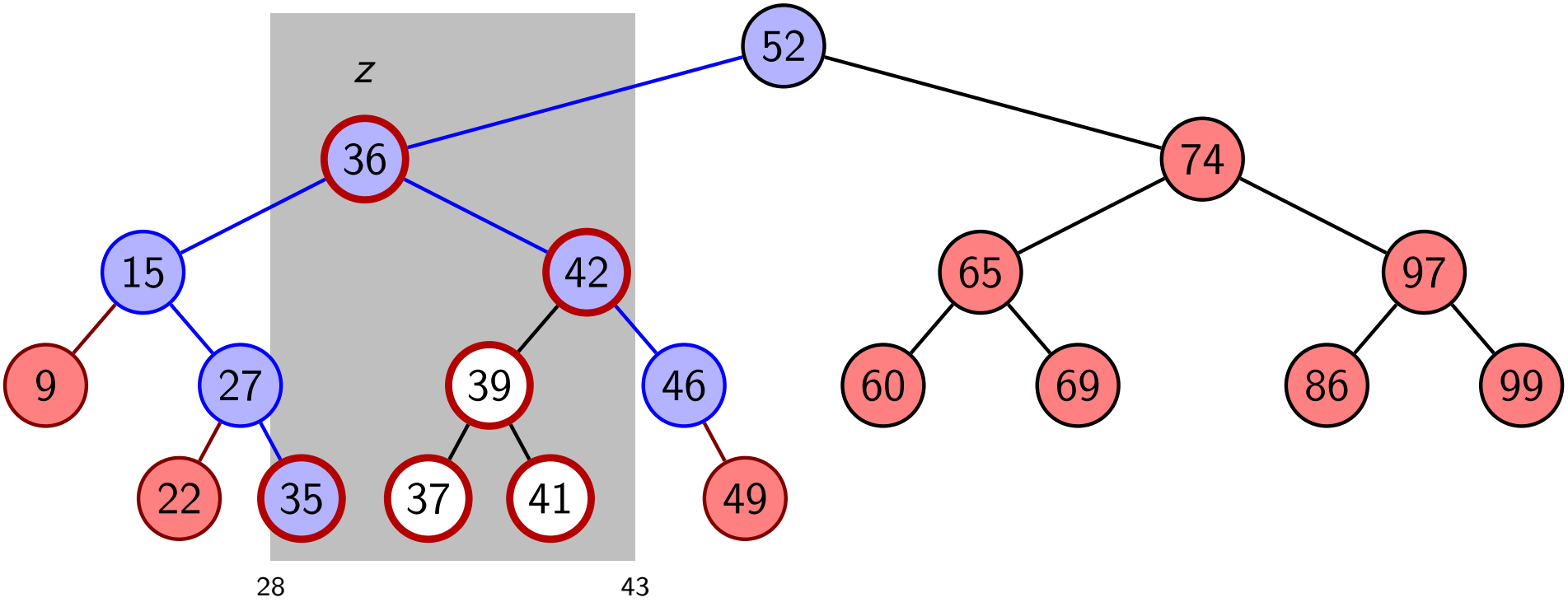
BST::range-search$(28, 43)$



- 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

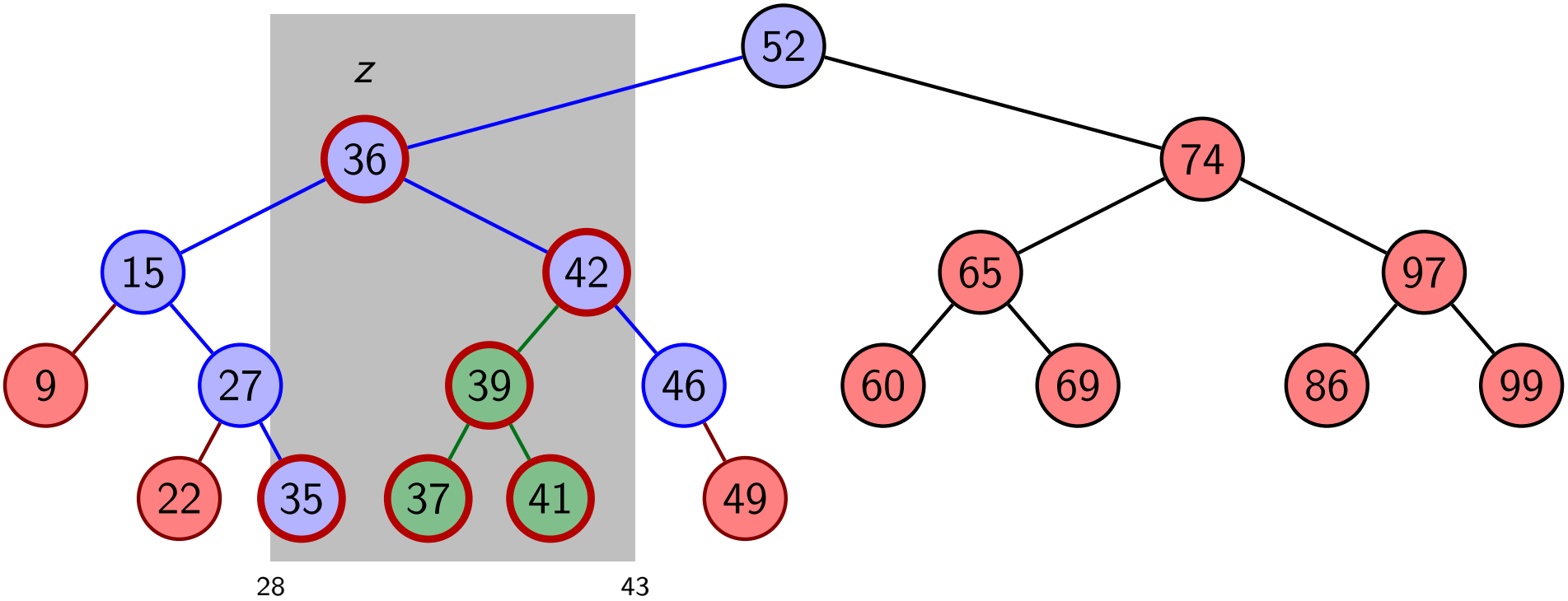
BST::range-search$(28, 43)$



- 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

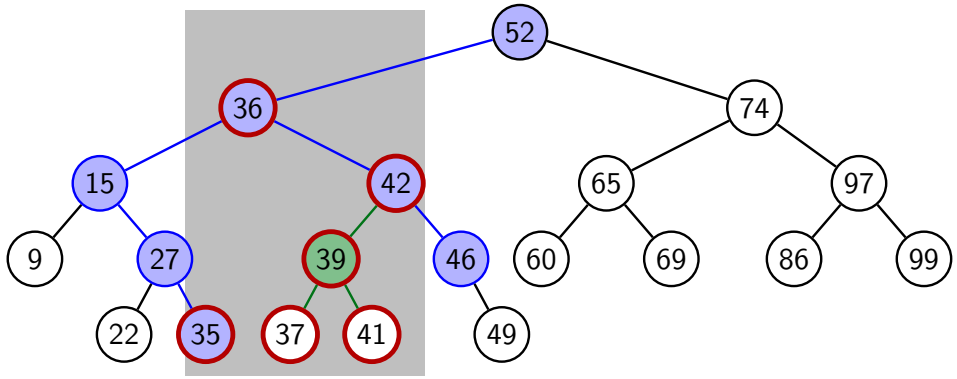
BST::range-search$(28, 43)$



- 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 $O(\mathit{height})$ 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: $O(\log n)$
- Crucial: Every descendant of a topmost inside node is in the range.
- For 1d 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 for 1d range search: $O(\log n + s)$.

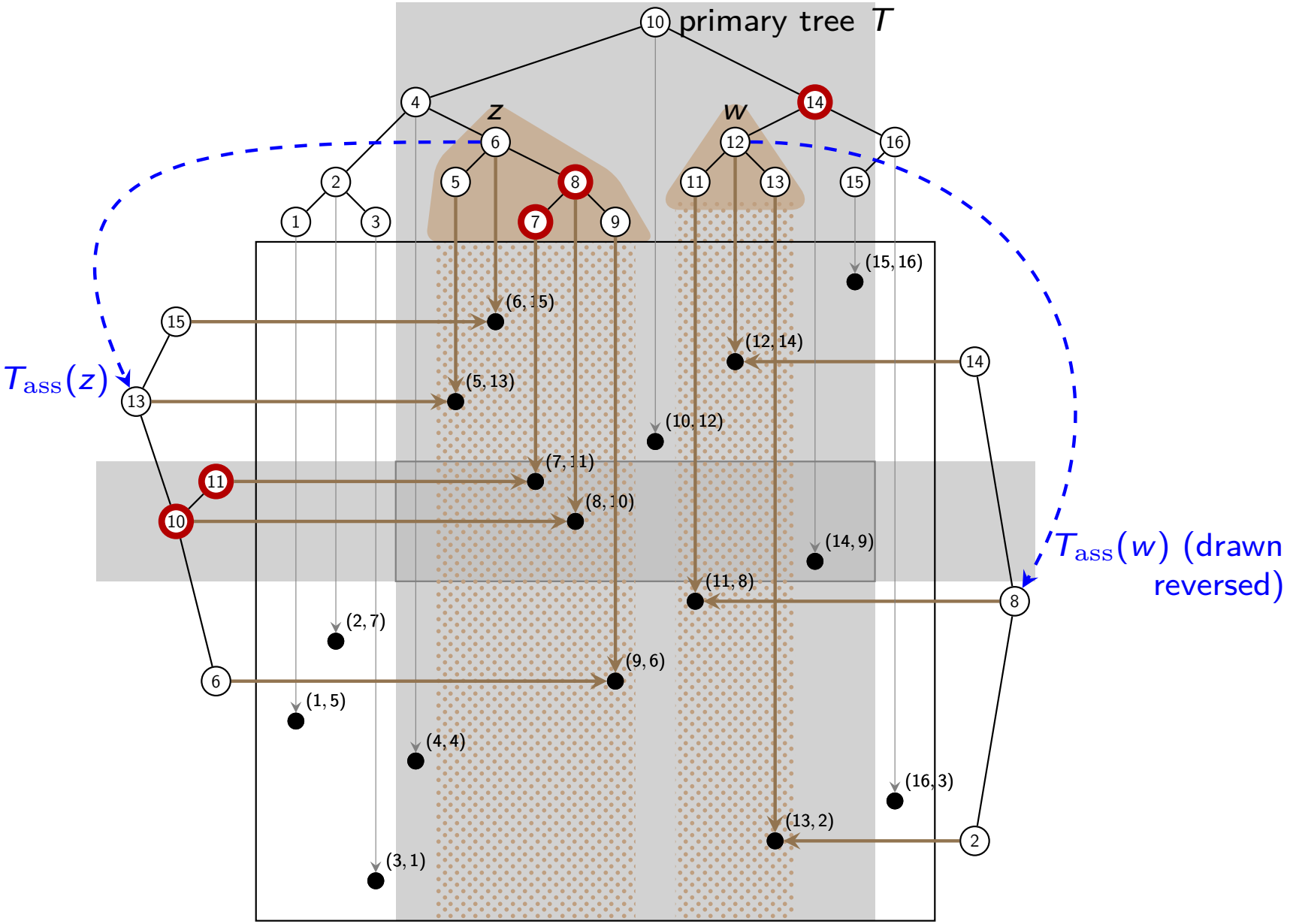
For 2d range search, we proceed slightly differently.

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))
- Do **not** report all points in range, instead 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 $T_{\text{ass}}(z)$.
  - To find points in $P(z)$ where the y-cordinates are within range as well, perform a range search in $T_{\text{ass}}(z)$: *BST::range-search*$(T_{\text{ass}}(z), y_1, y_2)$

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

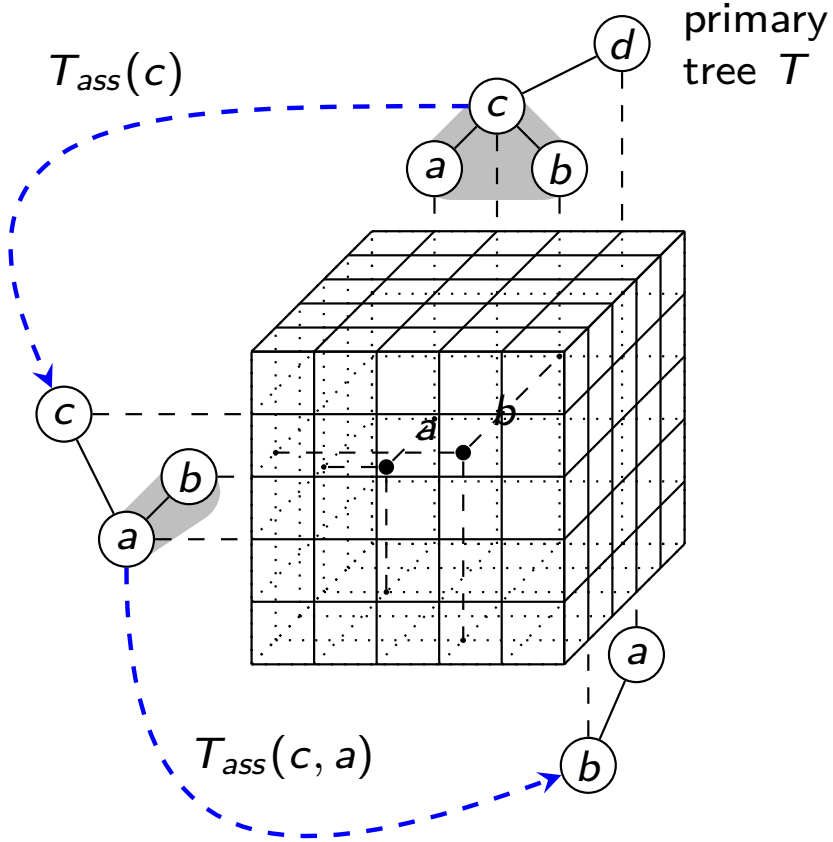
$$\sum_{z \text{ topmost inside}} (\log n + s_z) \in O(\log^2 n + s)$$

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

Range Trees: Higher Dimensions

Range trees can be generalized to d-dimensional space.

- Primary tree by x-coordinate.
- For each of its nodes: associated tree by y-coordinate.
- For each of its nodes: associated tree by z-coordinate.
- ... and so on for higher dimension.



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

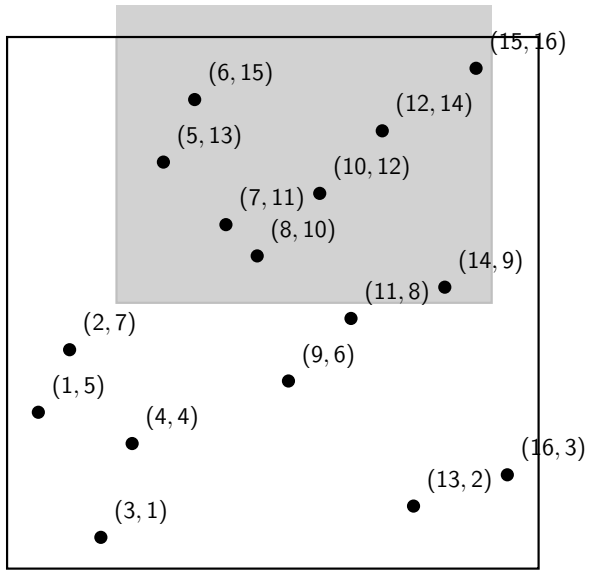
(Note: d is considered to be a constant.)

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

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* $y \geq y'$.

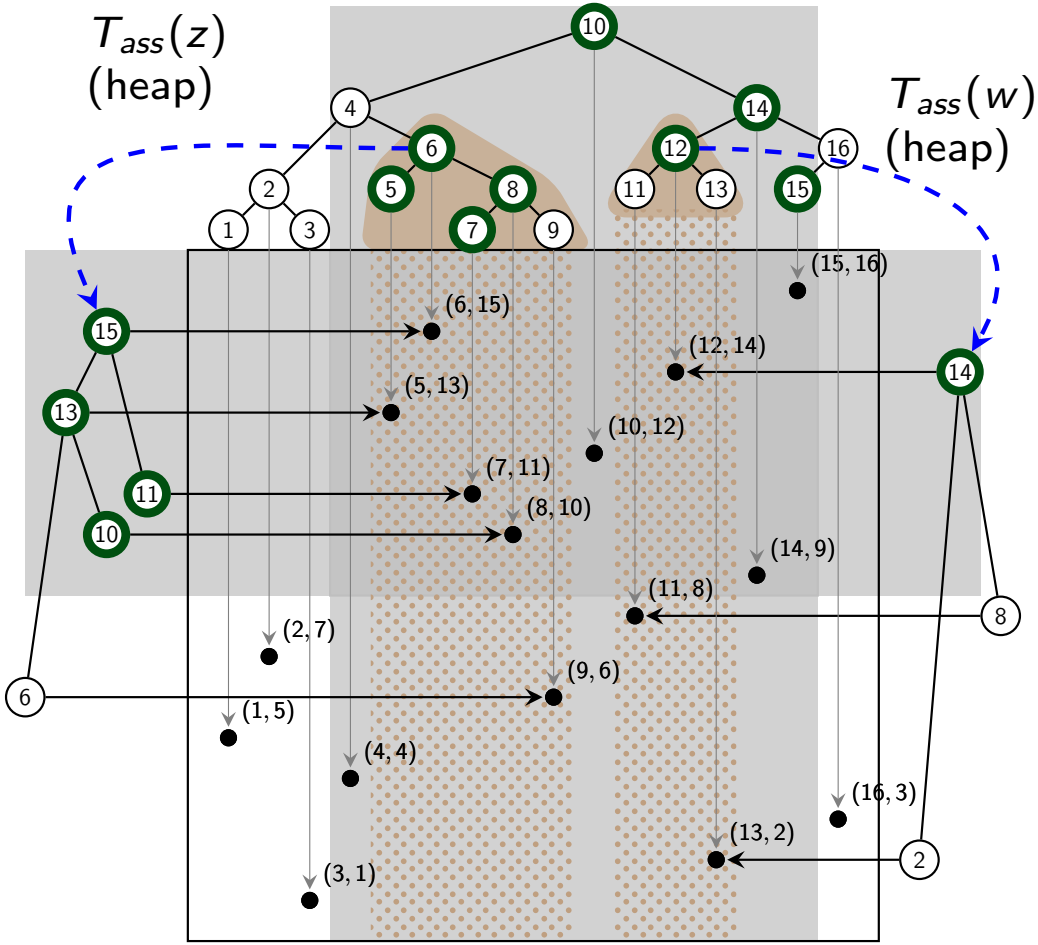


- We can do this with a range tree in $O(\log^2 n + s)$ with $\Theta(n \log n)$ space.
- Can we do this faster or using less space by adapting previous ideas to the special situation?

Idea 1: Associated heaps

Observe: In a heap, we can support 1-sided queries ("return all items bigger than y'") in $O(1+s)$ time. (Exercise.)

So to reduce the run-time, replace associated trees by associated heaps.

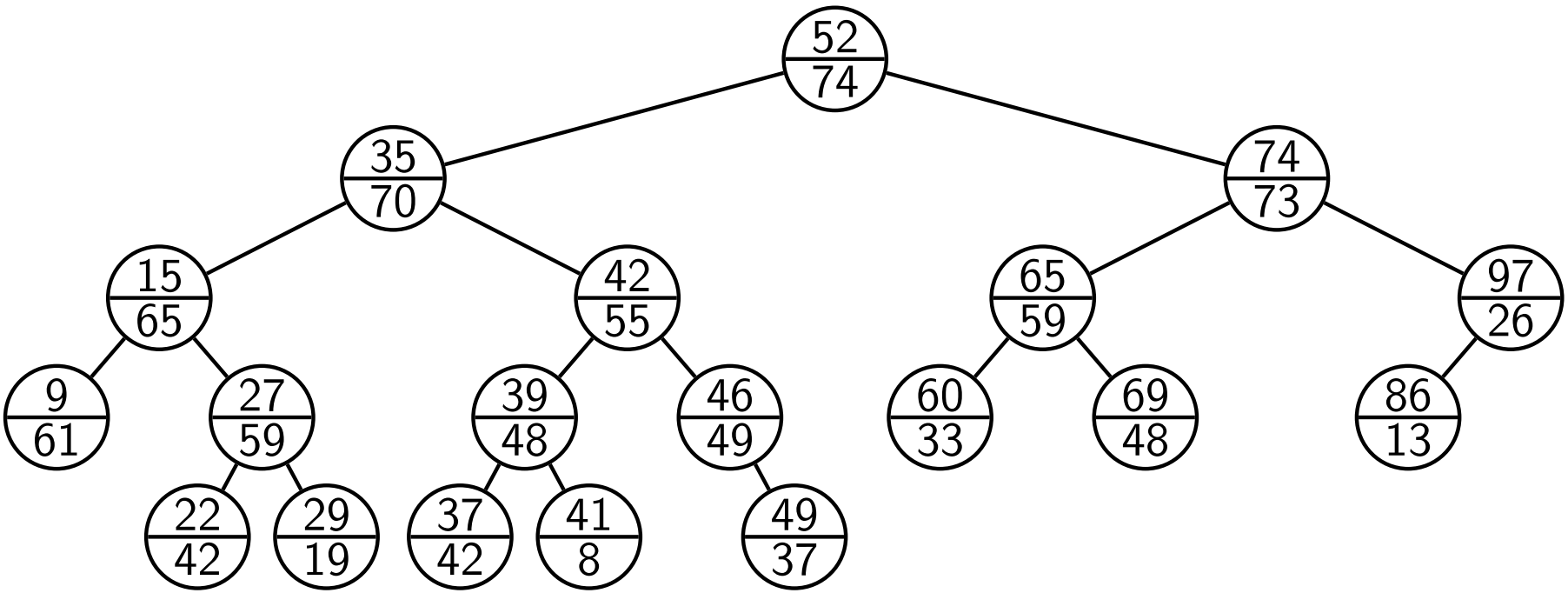


3-sided-range-search(x_1, x_2, y'):

- Range-query for $[x_1, x_2]$ in T as before to get boundary and topmost inside nodes.
- Boundary-nodes: Explicitly test whether in range.
- For each topmost inside node z: Search in $T_{\text{ass}}(z)$ for points above y' in $O(1+s_z)$ time.
- Total time: $O(\log n + s)$
- But space is $\omega(n)$

Idea 2: Cartesian trees

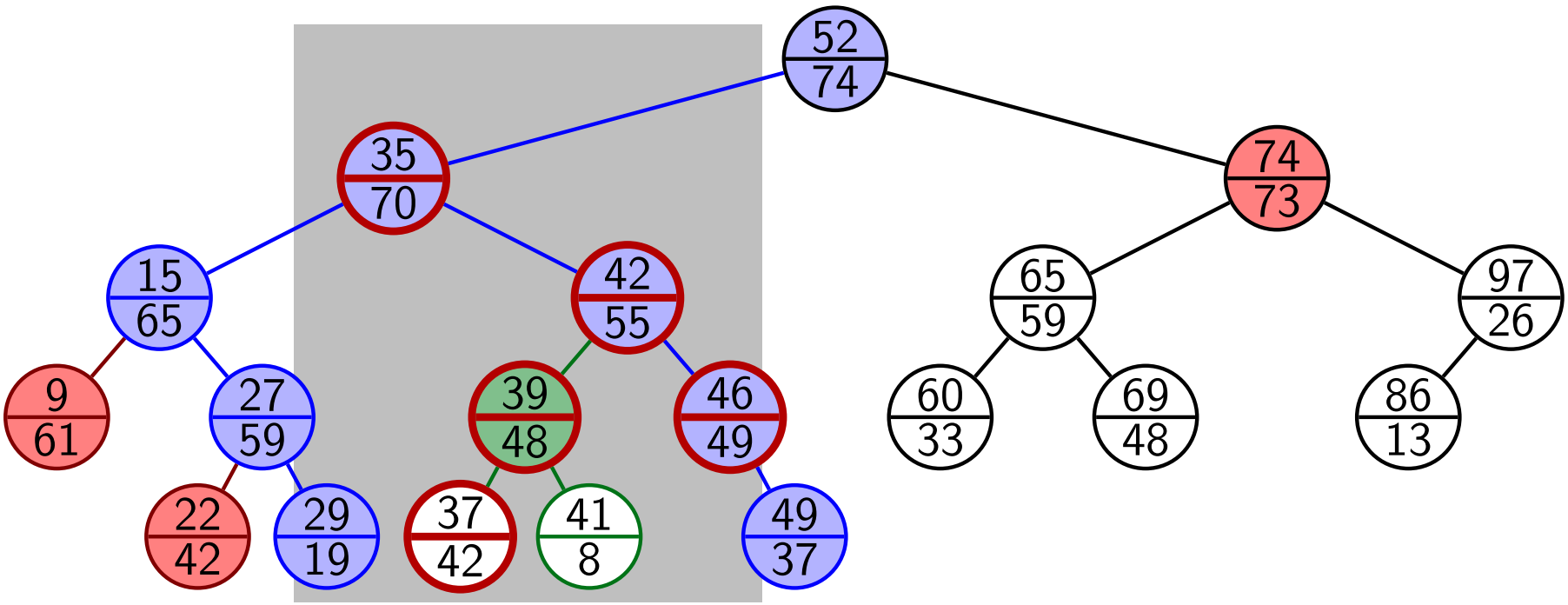
- Recall: Treap = binary search tree (with respect to keys)
  + heap (with respect to randomly chosen priorities)
- **Cartesian tree**: Use y-coordinate as priority instead.



Space: $\Theta(n)$.

Cartesian trees: 3-sided range search

CartesianTree::3-sided-range-search(T, x_1, x_2, y') :



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

Cartesian Trees: 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

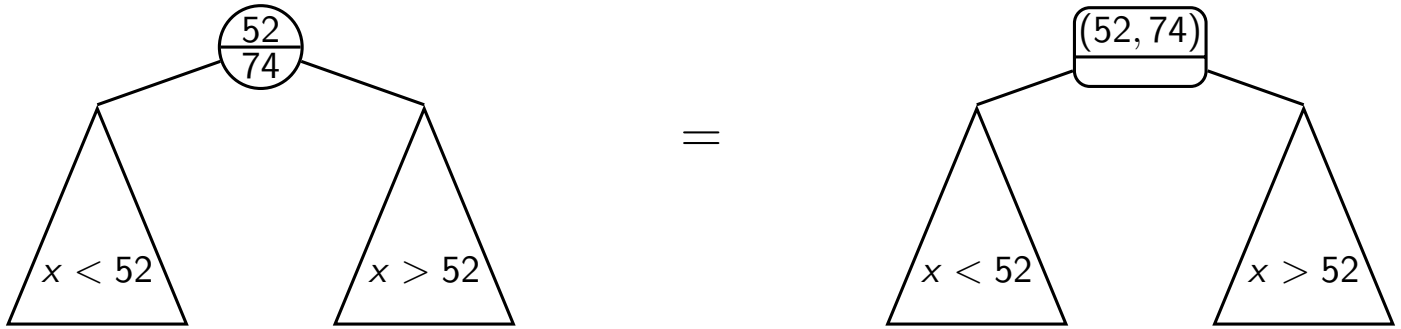
$\Rightarrow O(\textit{height} + s)$ run-time, $O(n)$ space

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

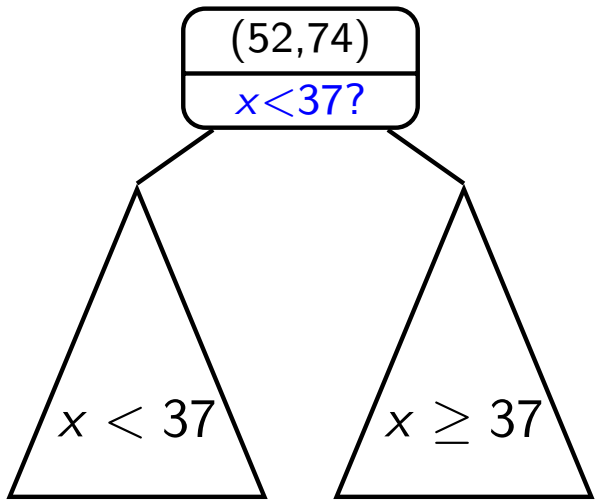
Question: Can we design a new data structure that keeps the good aspects of Cartesian tree but guarantees height $O(\log n)$?

Idea 3: Priority search trees

Idea of Cartesian trees:

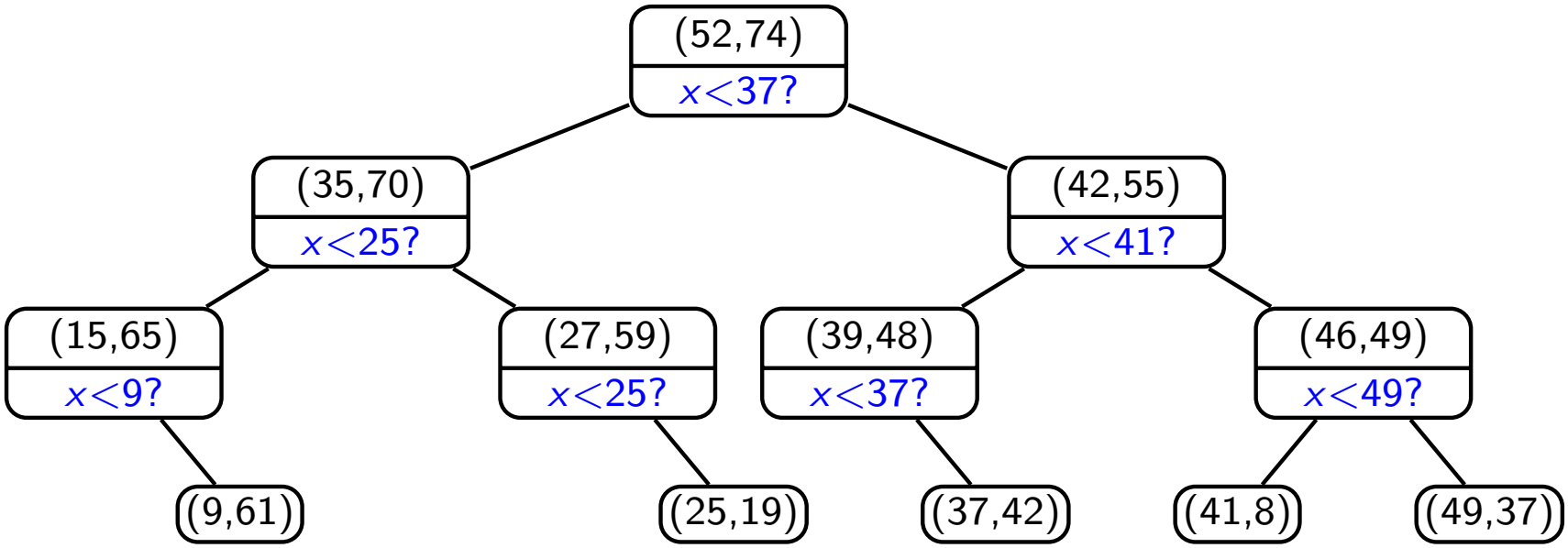


Better idea: Store a *separate* x-coordinate to use for the split.



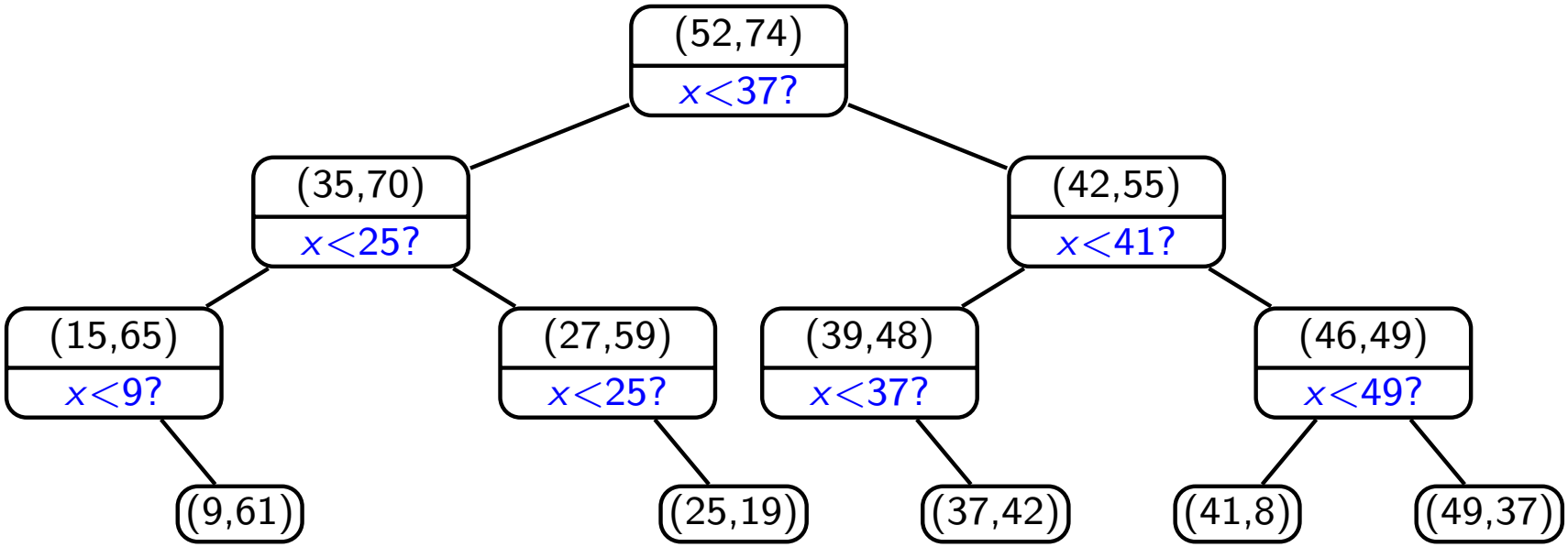
Choose this **split-line coordinate** carefully to ensure height $O(\log n)$.

Priority search trees: Definition and construction



- The root stores the point p with maximum y-coordinate in P.
- The root stores a split-line coordinate x'.
  - Normally use upper median among x-coordinates of $P \setminus p$.
  - But *insert*/*delete* may shift this a bit later.
- P_L: Points in $P \setminus p$ with x-coordinates less than x'.
  P_R: Points in $P \setminus p$ with x-coordinates at least x'.
- Left and right subtree are built recursively with P_L and P_R.

Idea 3: Priority search trees



- 3-sided range search: As for Cartesian trees, but height now $O(\log n)$. Run-time $O(\log n + s)$
- *insert* is easy if we do not care about height-balance (exercise).
- We can perform one rotation (+ required updates) in $O(\log n)$ time

  (We need a *fix-down*-like procedure to restore heap-order—no details.)
- *AVL::insert* needs only *one* rotation, so we can insert in $O(\log n)$ time.

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.