CS 240 – Data Structures and Data Management

Module 8: Range-Searching in Dictionaries for Points

O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2024

Outline

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

Outline

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

Range Searches

- *search*(k) looks for *one* specific item
- New operation *RangeSearch* (x, x')
  - look for *all* items that fall within given range (interval) $Q = (x, x')$
    - Q may have open or closed ends
  - report all KVPs in the dictionary with $k \in Q$
  - example

$s = 3, n = 10$

5	**10**	**11**	**17**	**18**	**33**	**45**	**51**	**55**	**77**

RangeSearch $(17,45]$ should return $\{18, 33, 45\}$

- As usual, n is the number of input items
- Let s be the output-size, i.e. the number of items in the range
- Need $\Omega(s)$ time just to report the items in the range
  - s can be anything between 0 and n (it depends on input interval Q)
- Therefore, running time depends both on s and n
  - so keep s as a parameter when analyzing runtime
  - getting $O(n)$ time is trivial
    - can we get $O(\log n + s)$?

Range Search in Existing Dictionary Realizations

- *Unsorted list/array/hash table*
  - range search requires $\Omega(n)$ time
    - must check for each item explicitly if it is in the range
- *Sorted array*

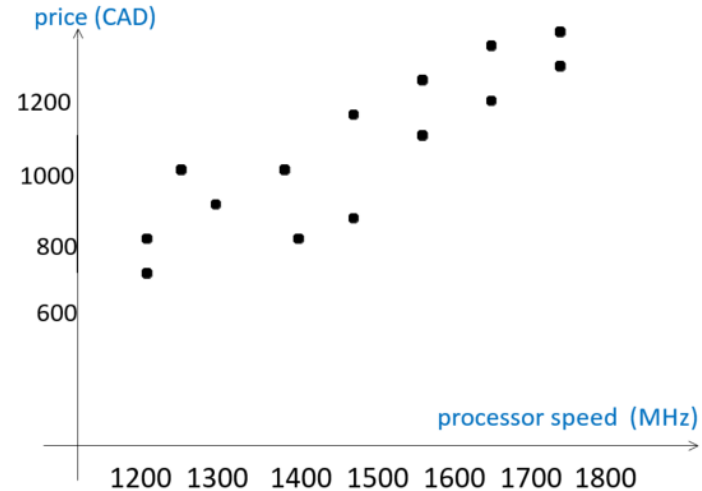
5	**10**	**11**	**17**	**18**	**33**	**45**	**51**	**55**	**77**
			i				i'		

  - *RangeSearch* $(16,50)$
  - $O(\log n)$ use binary search to find i s.t. x is at (or would be at) $A[i]$
  - $O(\log n)$ use binary search to find i' s.t. x' is at (or would be at) $A[i']$
  - $O(s)$ report all items in $A[i+1 \ldots i'-1]$
  - $O(1)$ report $A[i]$ and $A[i']$ if they are in the range
  - range search can be done in $O(\log n + s)$ time
- *BST*
  - can do range search in $O(height + s)$ time
    - details later

Outline

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

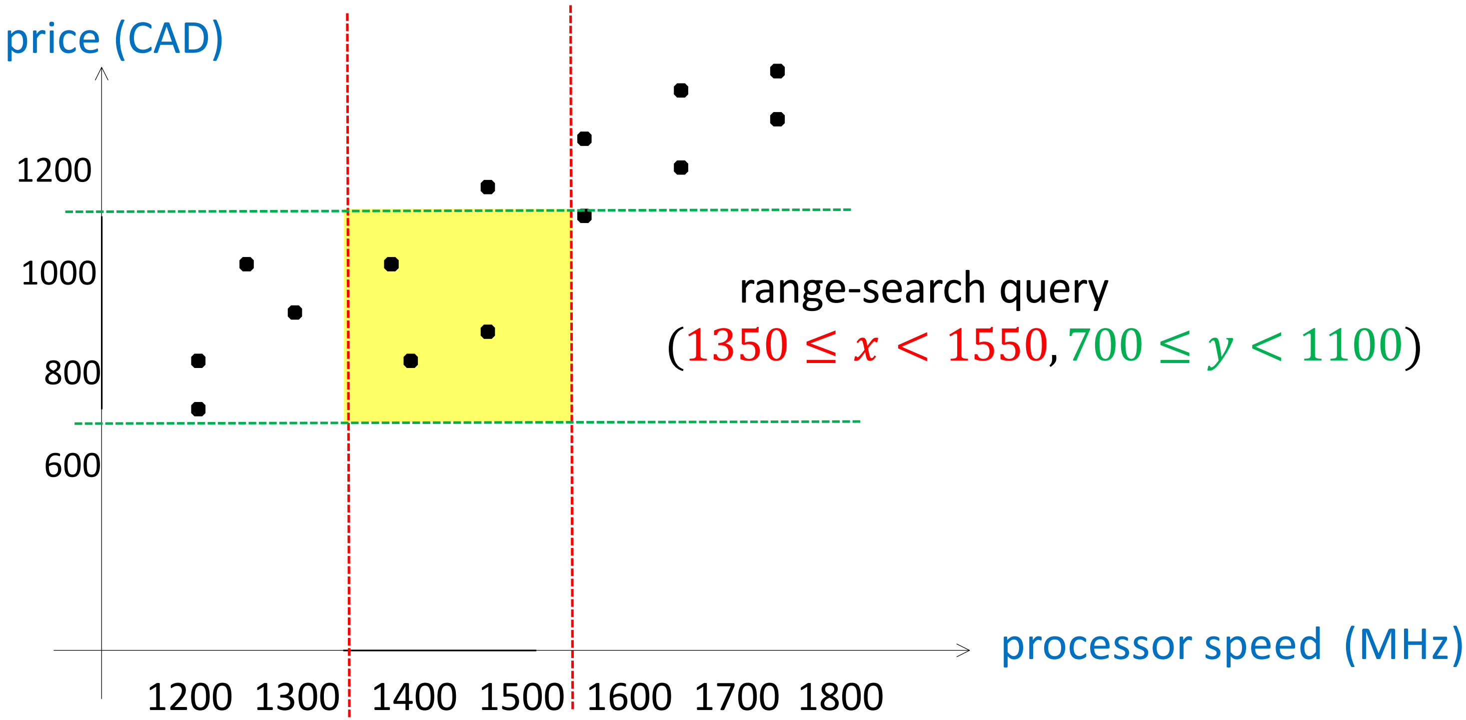
Multi-dimensional Data



- Data with multiple aspects of interest
  - laptops: price, screen size, processor speed, ...
  - employees: name, age, salary, ...
- Range searches are of special interest for multidimensional data
  - flights that leave between 9am and noon, and cost between $400 and $600
- Dictionary for multi-dimensional data
  - collection of d-dimensional items (or points)
  - each item has d aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
  - need usual dictionary operations: *insert*, *delete*, *search*
  - also need *RangeSearch*
- We focus on $d = 2$, i.e. points in Euclidean plane

Multi-Dimensional Range Search

- (Orthogonal) d-dimensional range search
  - given a *query rectangle* Q, find all points that lie within Q

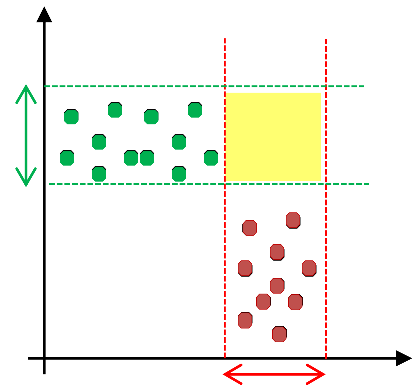


d-Dimensional Dictionary via 1-Dimensional Dictionary

- Option 1: Reduce to one-dimensional dictionary
    - combine d-dimensional key into one dimensional key
        - i.e. $(x, y) \rightarrow x + y \cdot n^2$
        - $(price, screenSize) \rightarrow price + screenSize \cdot n^2$
        - two distinct (x, y) map to a distinct one dimensional key
    - can search for a specific key (x, y)
    - but no efficient range search

d-Dimensional Dictionary via 1-Dimensional Dictionary

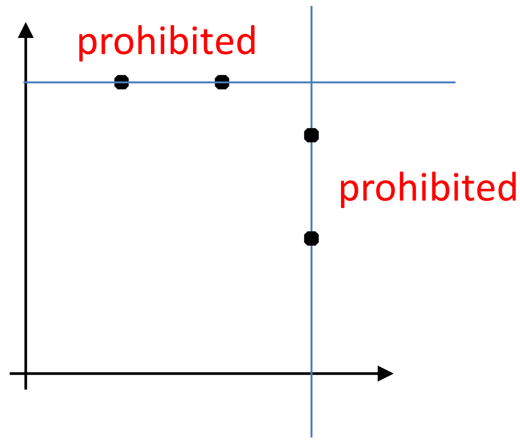
- Option2: Use several dictionaries, one for each dimension
    - problem: wastes space, inefficient search
  - Worst Case Example
    - insert all n points in horizontal dictionary
      - key is x coordinate
    - insert all n points in vertical dictionary
      - key is y coordinate



    - 1D range search in horizontal dictionary returns $n/2$ points
    - 1Drange search in vertical dictionary returns $n/2$ points
    - For 2D range search result, need to find points which are both in the red and the green clouds
      - insert $n/2$ red points in AVL tree
      - for each of $n/2$ green point, check if it is in the AVL Tree
    - total time to find points in both clouds is $O(n \log n)$
      - worse than exhaustive search!
      - far from $O(s + \log n)$, especially since $s = 0$

Multi-Dimensional Range Search

- Better idea
  - design new data structures specifically for points
- Assumption: points are in *general position*: no two x-coordinates or y-coordinates are the same
  - i.e. no two points on a horizontal lines, no two points on a vertical line



  - simplifies presentation, data structures can be generalized to arbitrary points

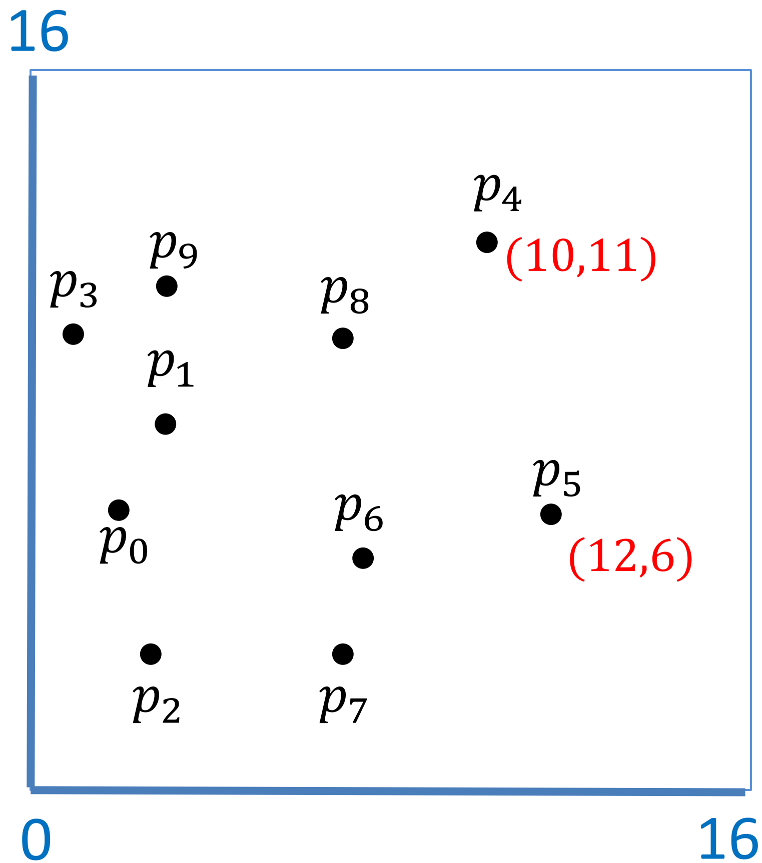
Multi-Dimensional Range Search

- Partition trees
  - organize space to facilitate efficient multidimensional search
    - internal nodes are associated with spatial regions
    - actual dictionary points stored only at leaves
  - We study 2 types of partition trees
    1. quadtrees
       - does not use general points position assumption
    2. kd-trees
       - uses general points position assumption
- Multi-dimensional range trees
  - a tree that generalizes BST to support multidimensional search
  - both internal and leaf nodes store points, similar to one dimensional BST
  - uses general points position assumption

Outline

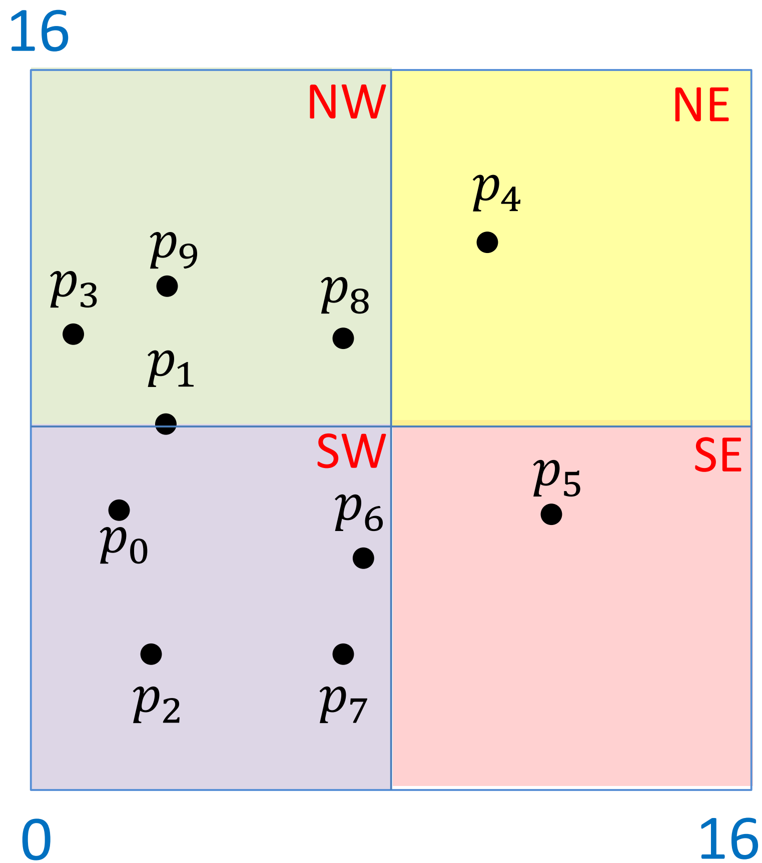
- Range-Searching in Dictionaries for Points
  - Range Search Query
  - Multi-Dimensional Data
  - Quadtrees
  - kd-Trees
  - Range Trees
  - Conclusion

Quadtrees

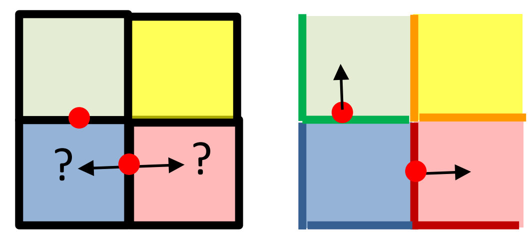


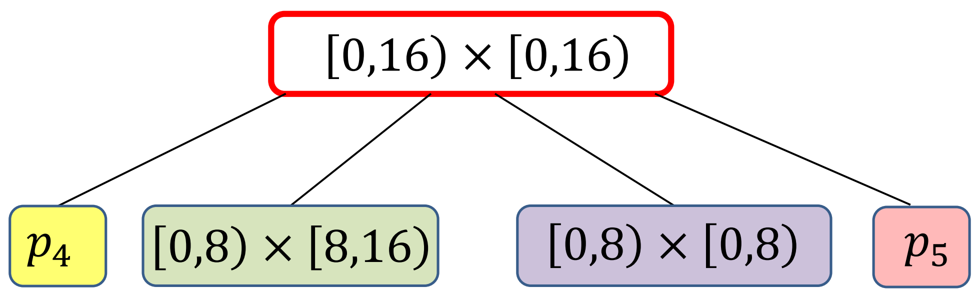
- Have a set S of n points in the plane
- Find *bounding box* $R = [0, 2^k) \times [0, 2^k)$
  - translate points so coordinates are non-negative
  - smallest $2^k \times 2^k$ square containing all points
    - find smallest k s.t. max-coordinate in S is less than 2^k
- Quadtree is a tree
- Each node corresponds to a region
- Higher levels responsible for larger regions
- Leaves responsible for regions small enough to store one point

Quadtree Construction Example

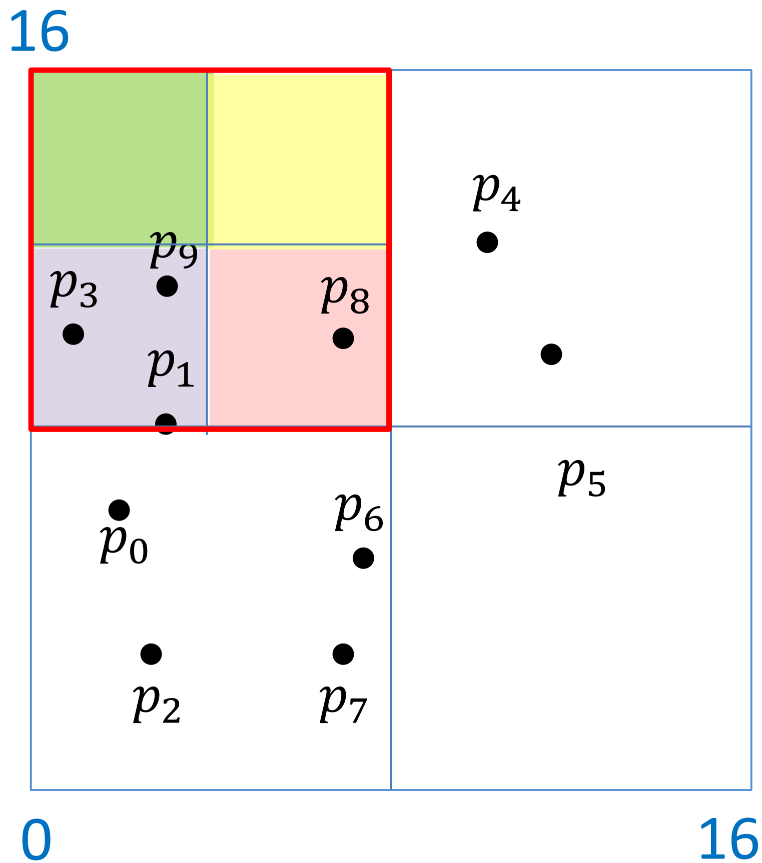


- Root corresponds to the whole square
- Split the square into 4 equal regions
- Convention: points on split lines belong to region on the right (or top)

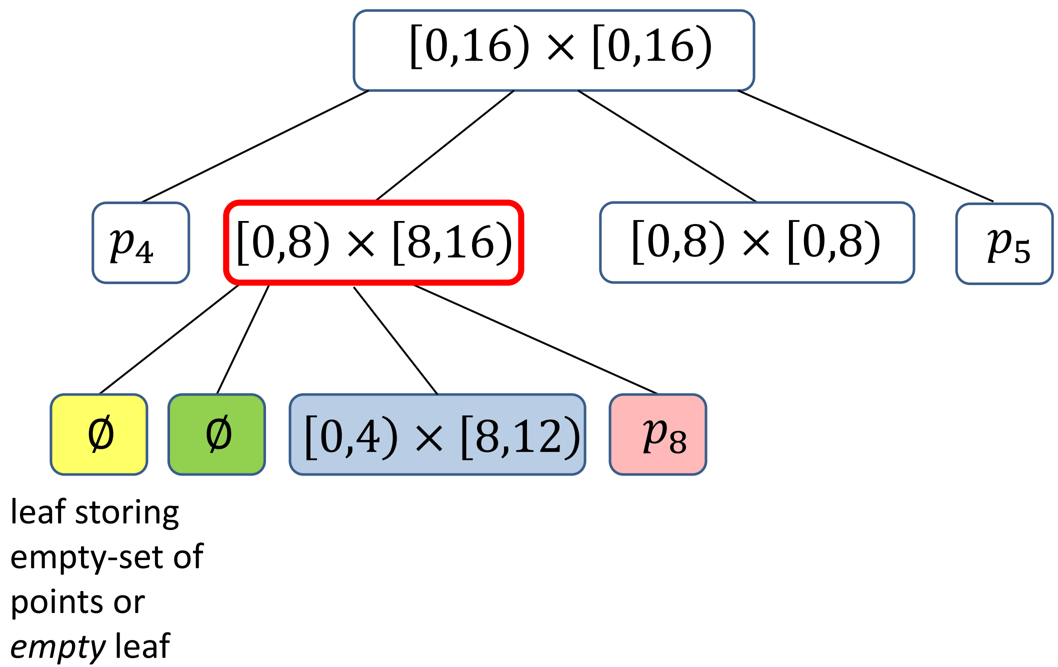




Quadtree Construction Example

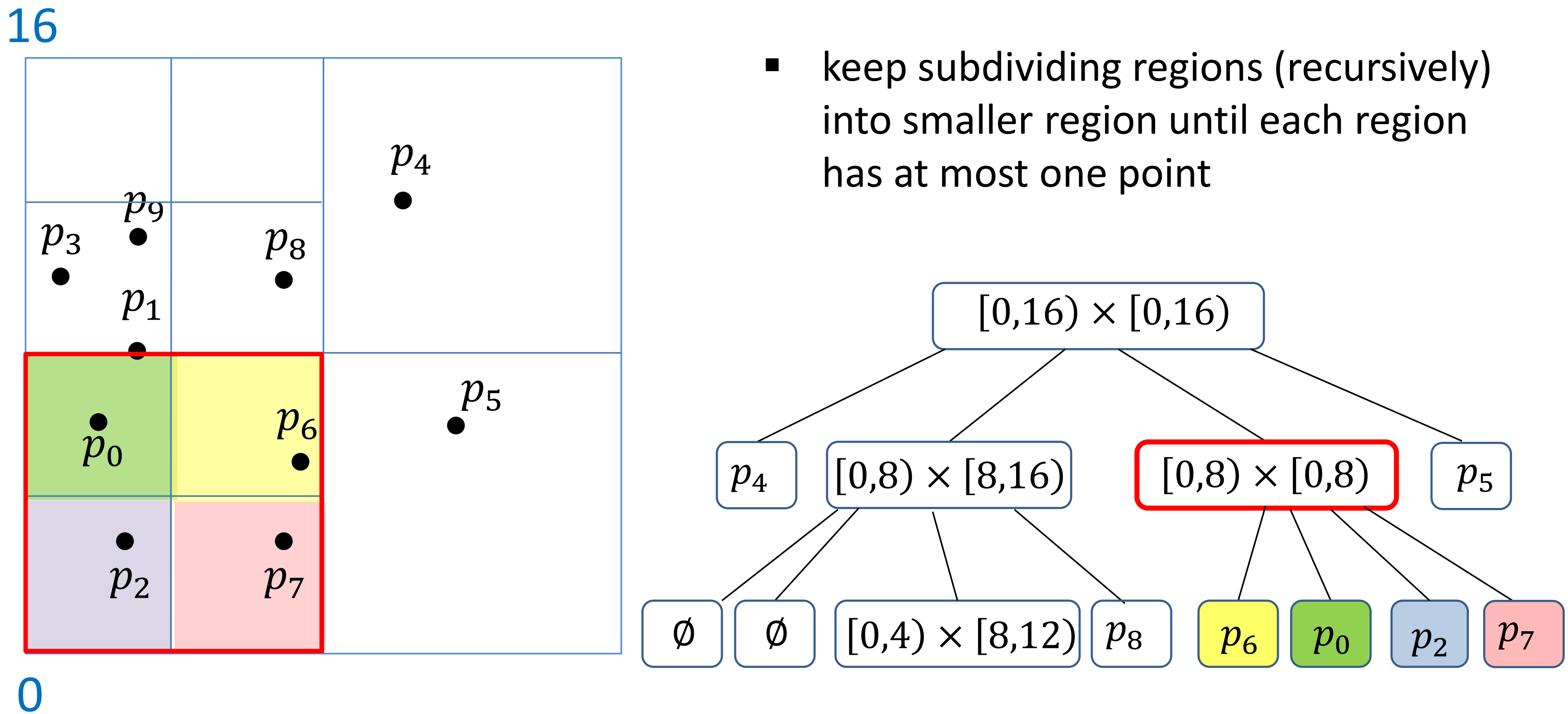


- keep subdividing regions (recursively) into smaller region until each region has at most one point

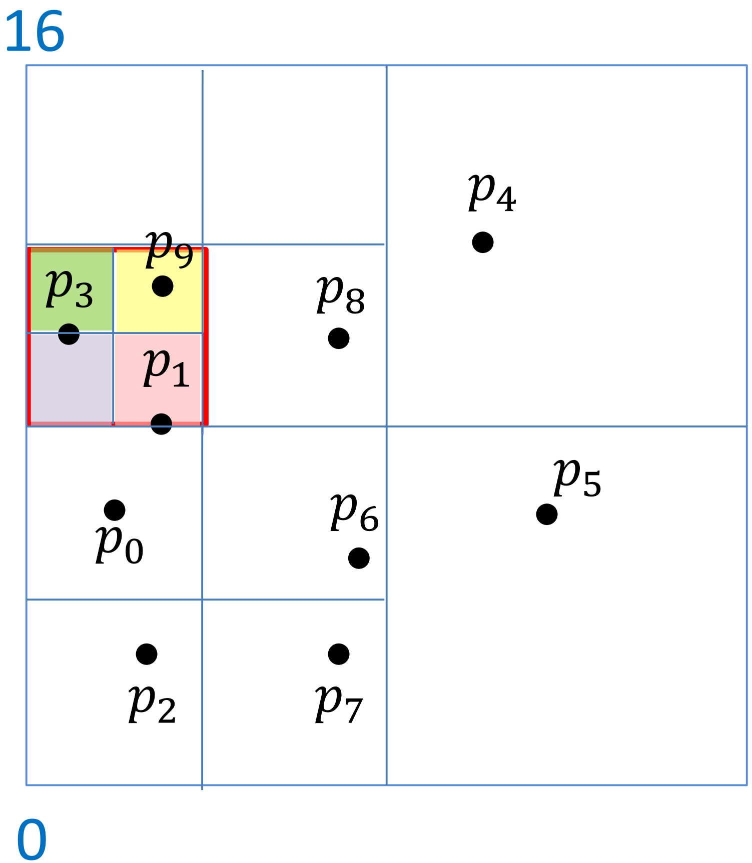


Quadtree Construction Example

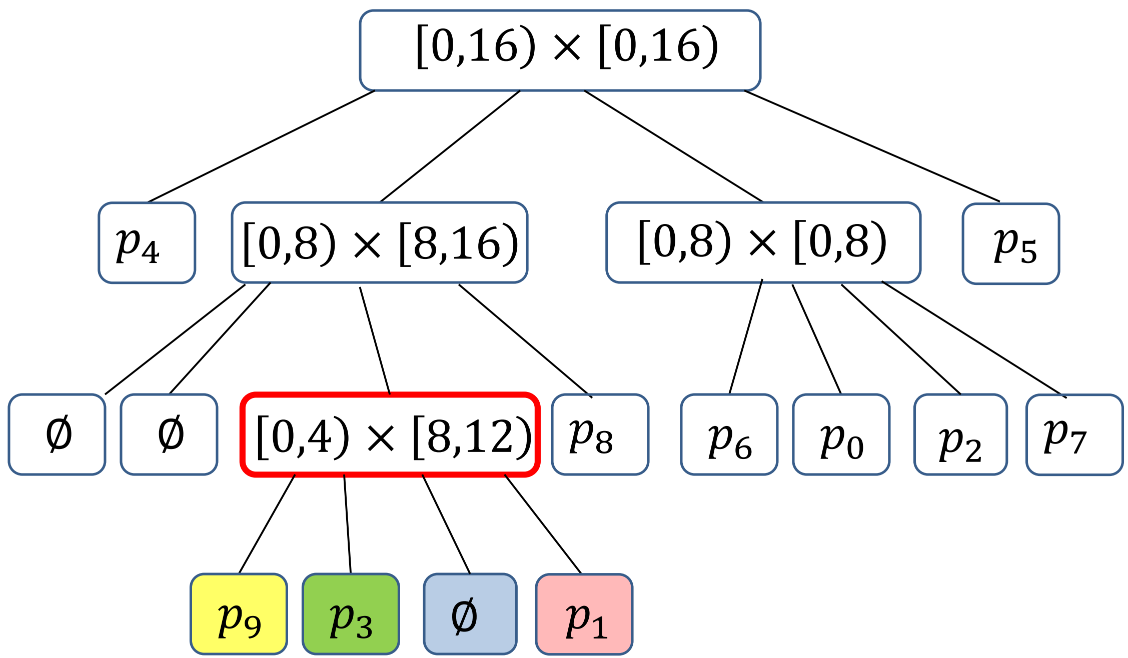
- keep subdividing regions (recursively) into smaller region until each region has at most one point



Quadtree Construction Example



- keep subdividing regions (recursively) into smaller region until each region has at most one point



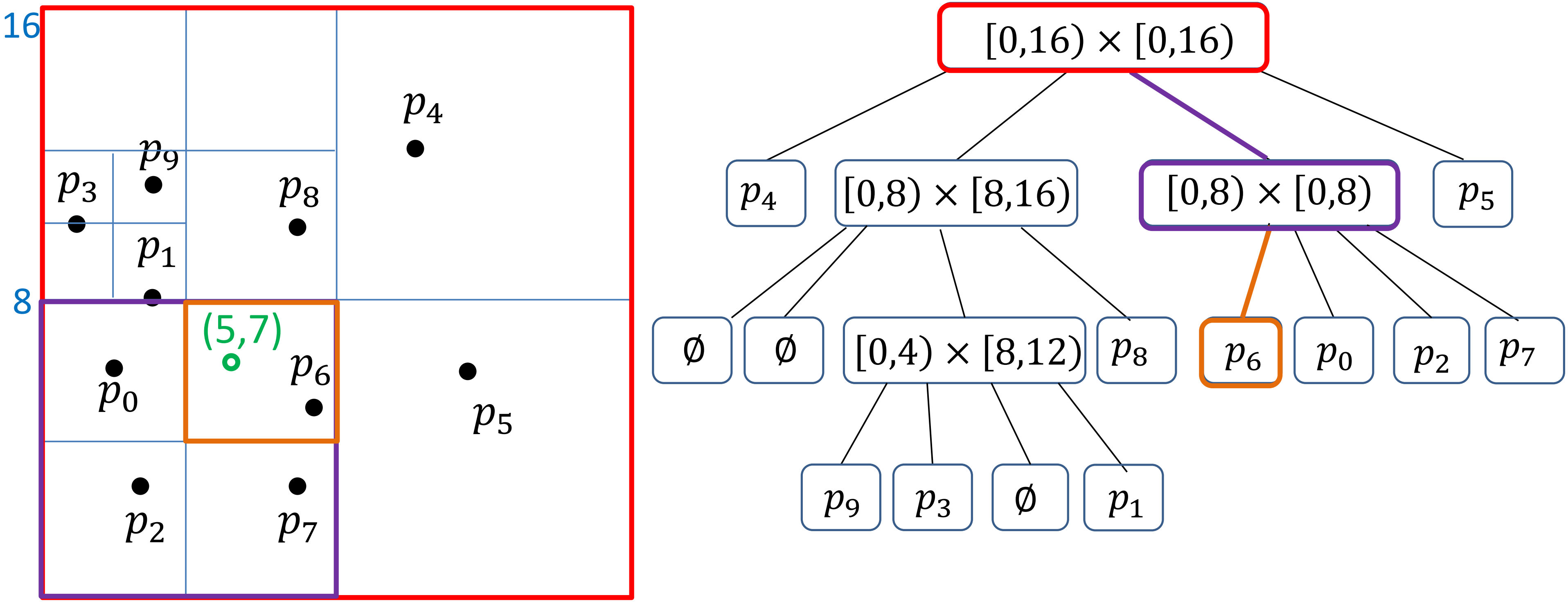
Quadtree Building Summary

- Have n points $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$
  - all points are within a square R
- To build quadtree on S
  - root r corresponds to R
  - if R contains 0 (or 1) point
    - then root r is an empty leaf (or a leaf that stores 1 point)
  - else
    - partition R into four equal subsquares (quadrants) $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
    - partition S into sets $S_{NE}, S_{NW}, S_{SW}, S_{SE}$
      - convention: points on split lines belong to region on the right (or top)
    - recursively build tree T_i for points S_i in R_i and make them children of root

Quadtree Search

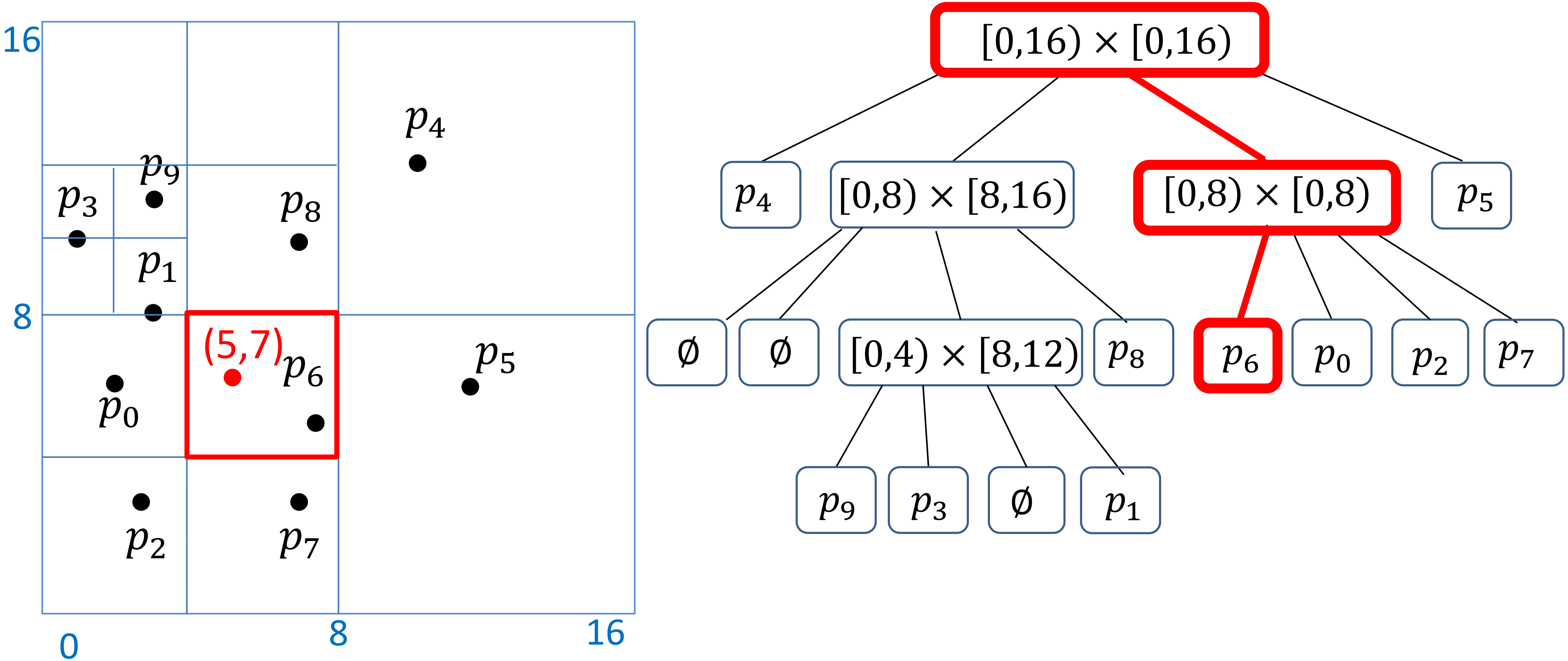
- Whenever possible, search rules out regions at higher level of hierarchy, achieving efficiency

Quadtree Search



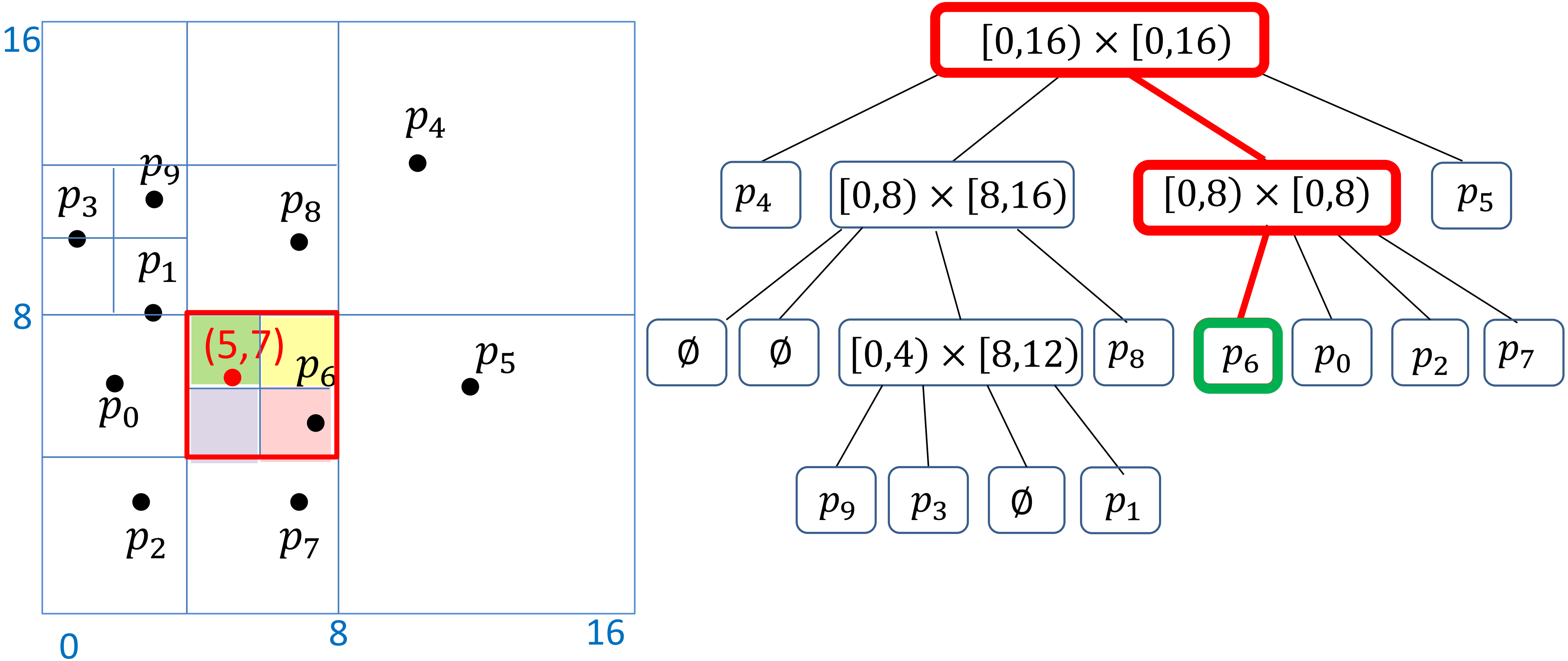
- Analogous to trie or BST
- Three possibilities for where search ends
  1. leaf storing point we search for (found)
  2. leaf storing point different from search point (not found)
  3. empty leaf (not found)
- Example: search(5,7) (not found)
- Search is efficient if quadtree has small height

Quadtree Insert



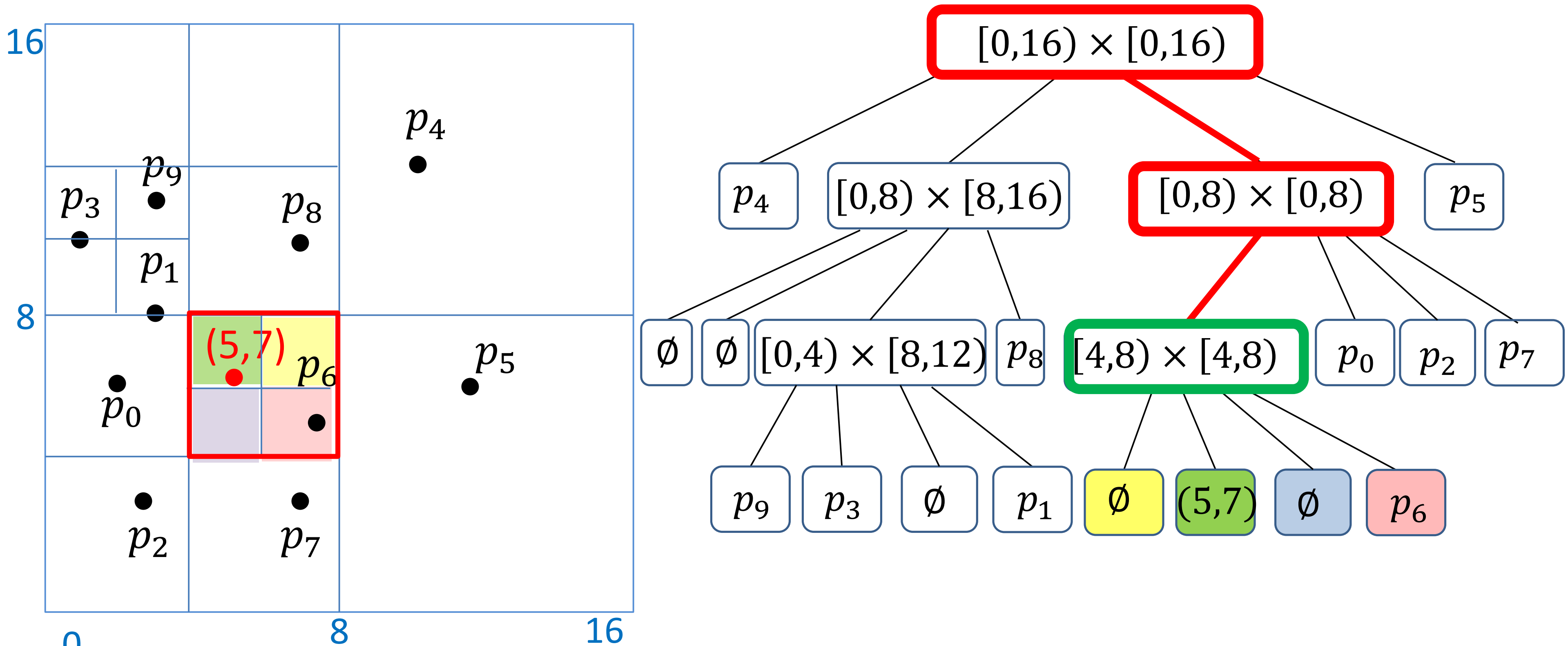
- First perform search
- Two cases
    1. search finds a leaf storing one point
        - example: insert(5,7)

Quadtree Insert



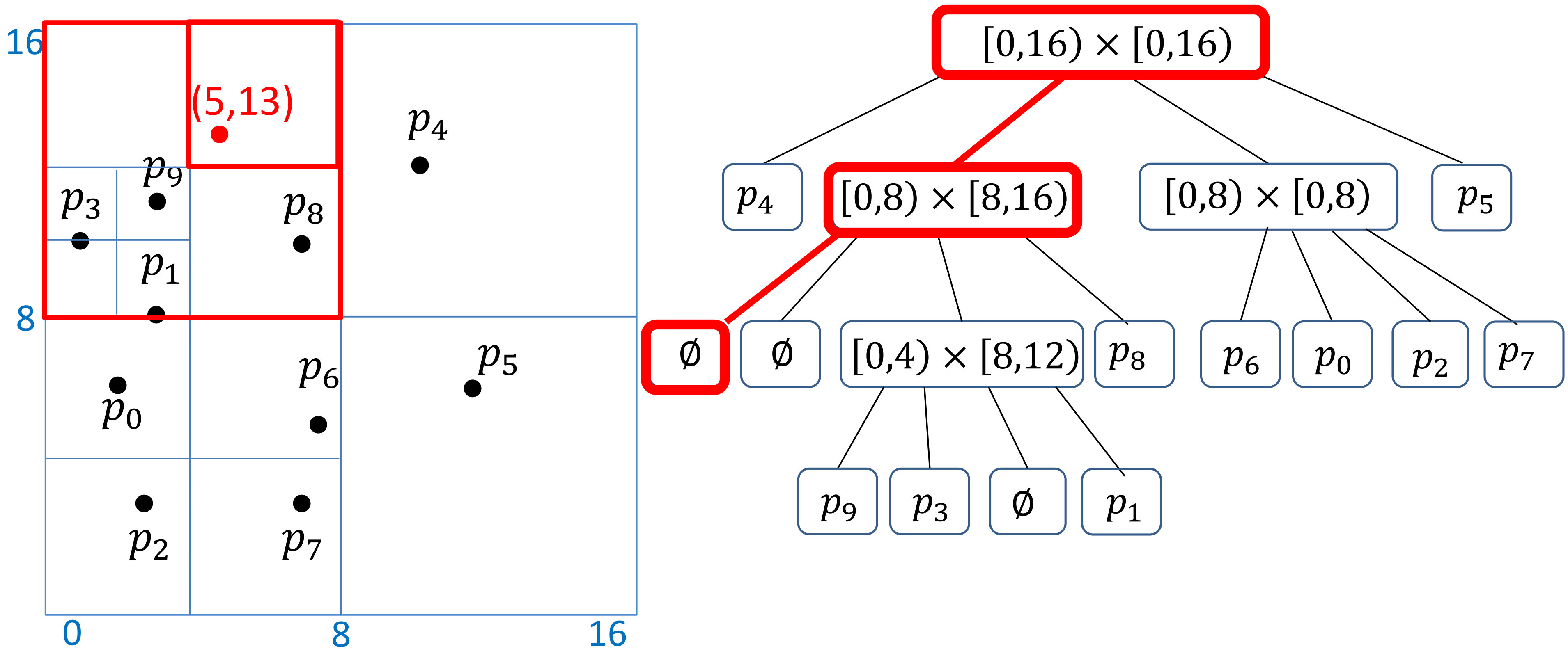
- First perform search
- Two cases
    1. search finds a leaf storing one point
        - example: insert(5,7)
        - repeatedly split the leaf **while** there are two points in one region

Quadtree Insert



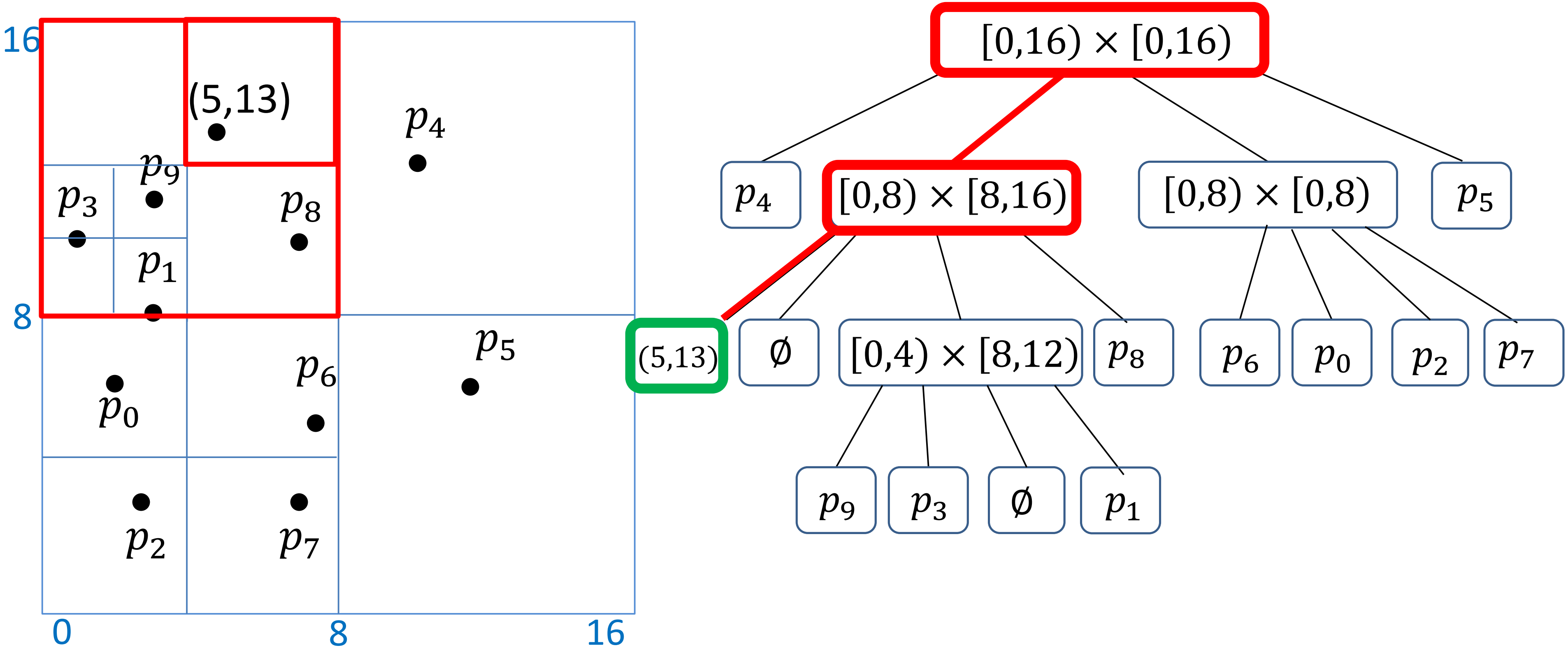
- First perform search
- Two cases
    1. search finds a leaf storing one point
        - example: insert(5,7)
        - repeatedly split the leaf **while** there are two points in one region

Quadtree Insert



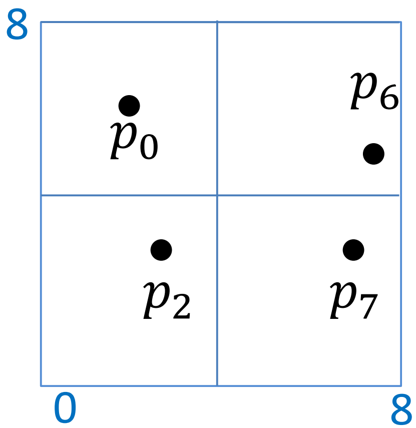
- First perform search
- Two cases
    1. search finds a leaf storing one point
    2. search finds an empty leaf
        - example: insert (5,13)

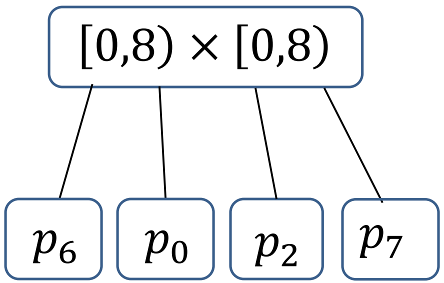
Quadtree Insert



- First perform search
- Two cases
  1. search finds a leaf storing one point
  2. search finds an empty leaf
     - example: insert(5,13)
     - insert the point into leaf

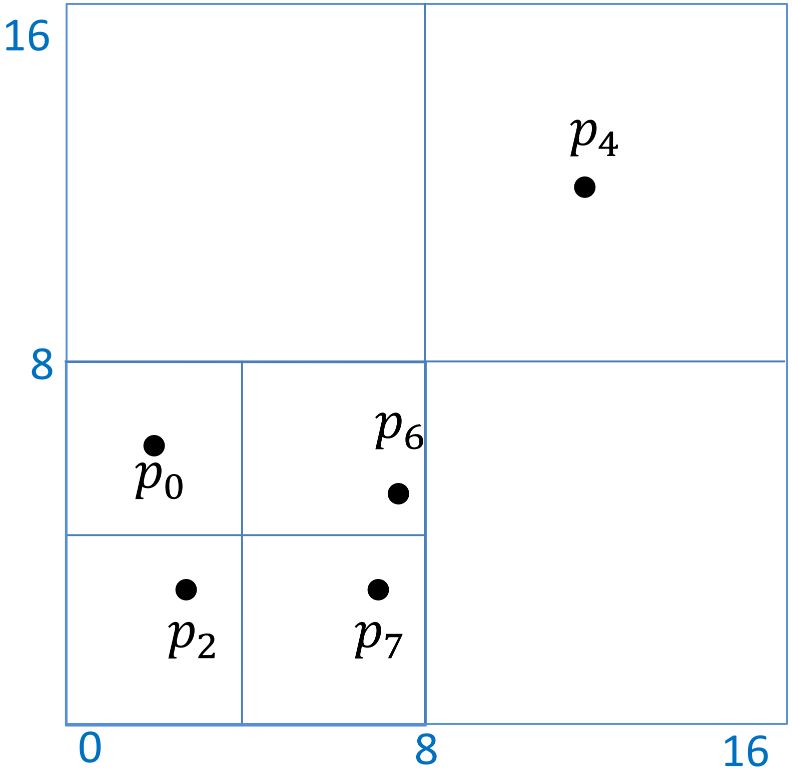
Quadtree Insert

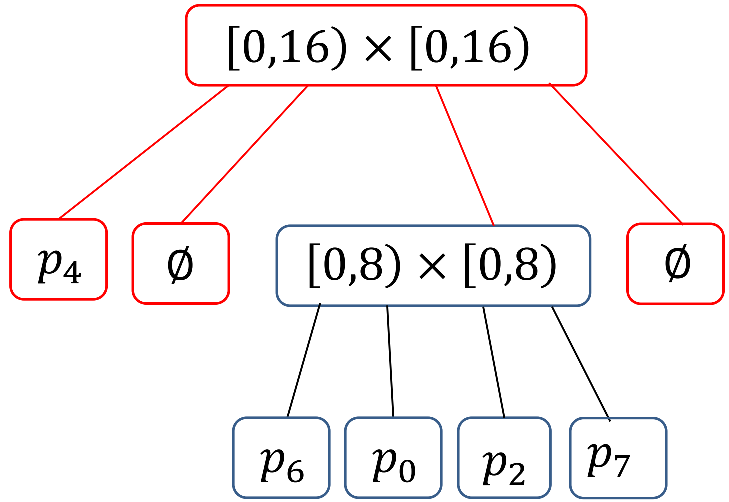




- If we insert point outside the bounding box, no need to rebuild the tree due to bounding box being $[0, 2^k) \times [0, 2^k)$

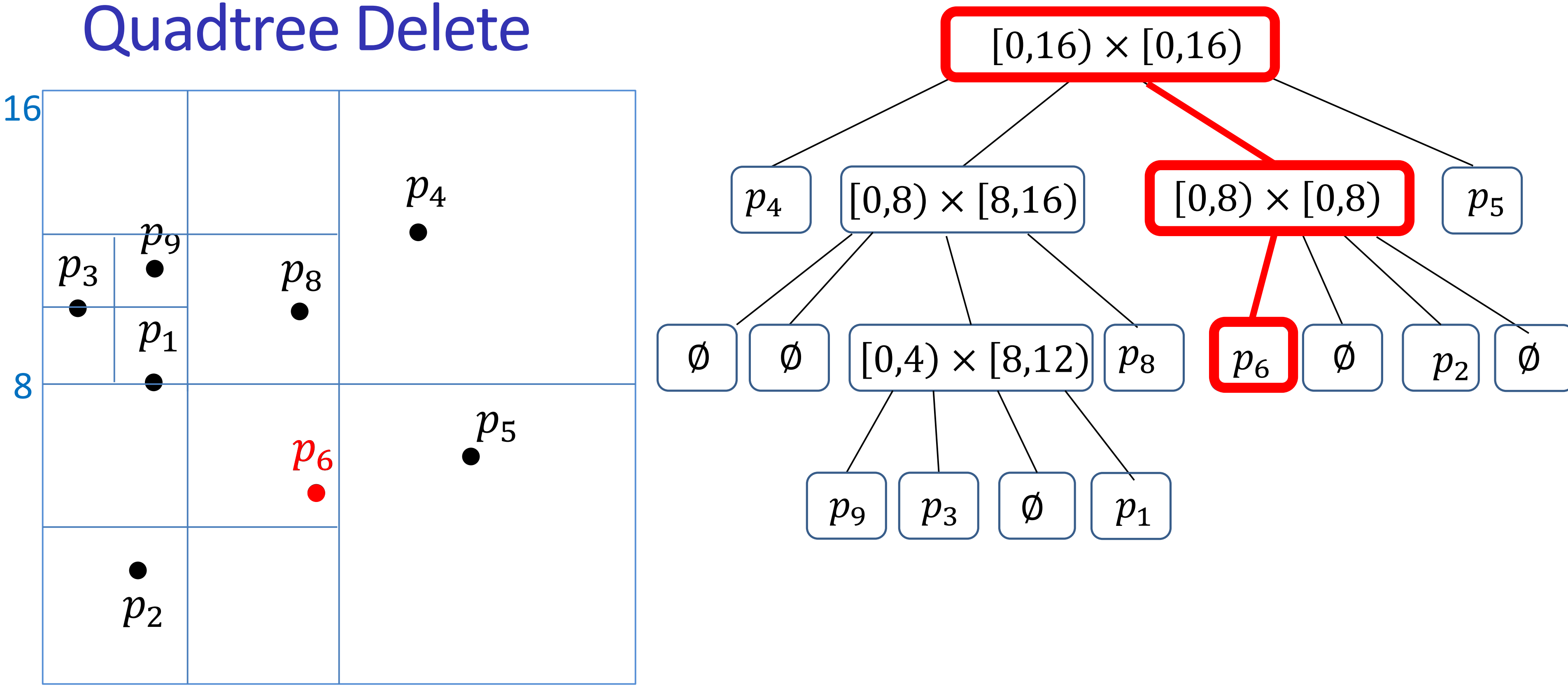
Quadtree Insert





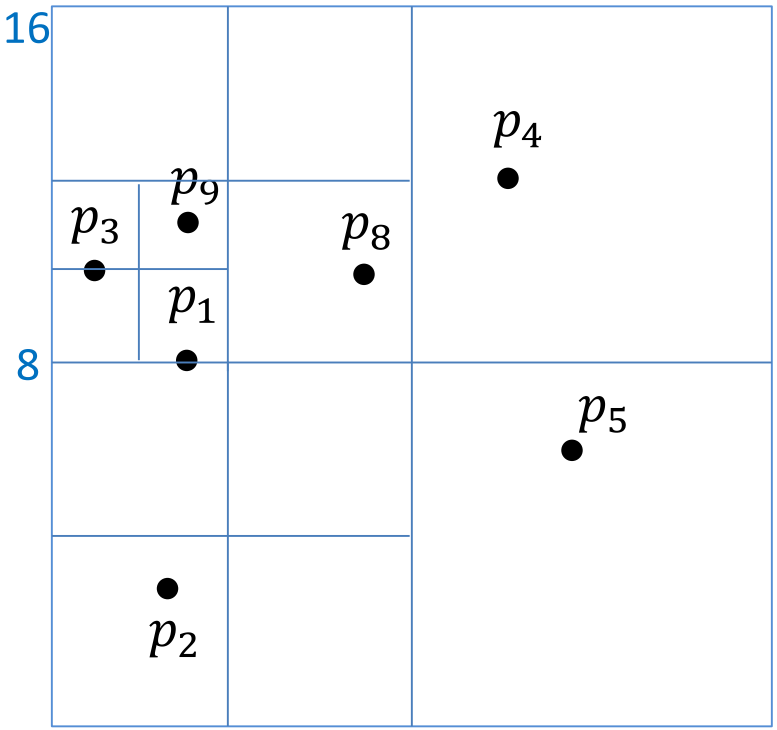
- If we insert point outside the bounding box, no need to rebuild the tree due to bounding box being $[0, 2^k) \times [0, 2^k)$

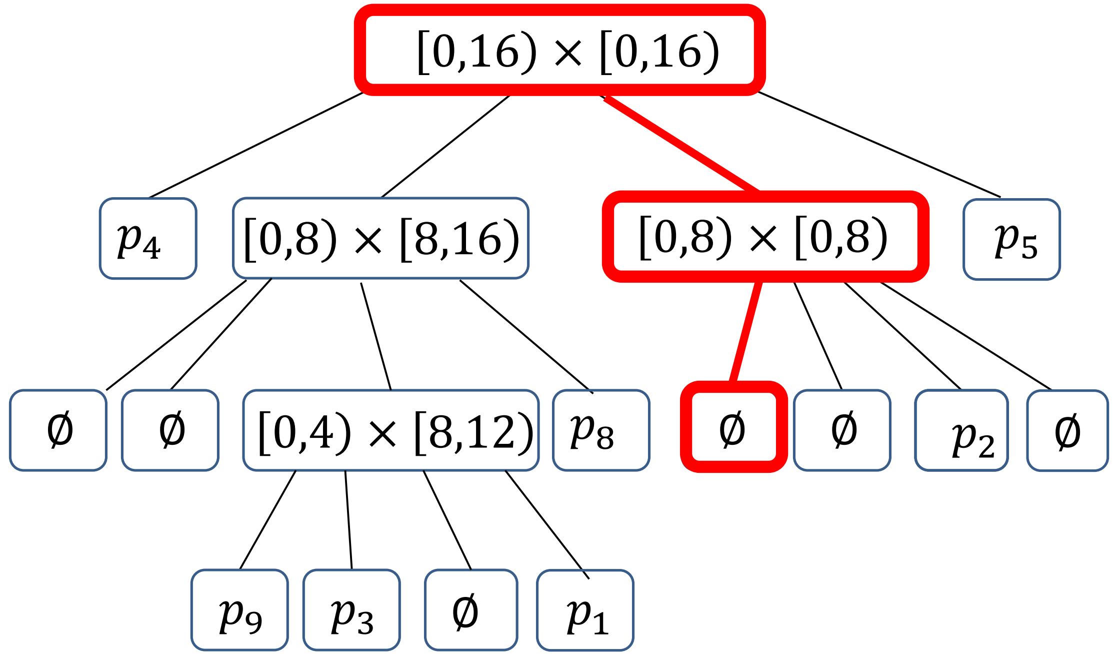
Quadtree Delete



- search will find a leaf containing the point
    - example: delete(p_6)
- remove the point leaving the leaf empty

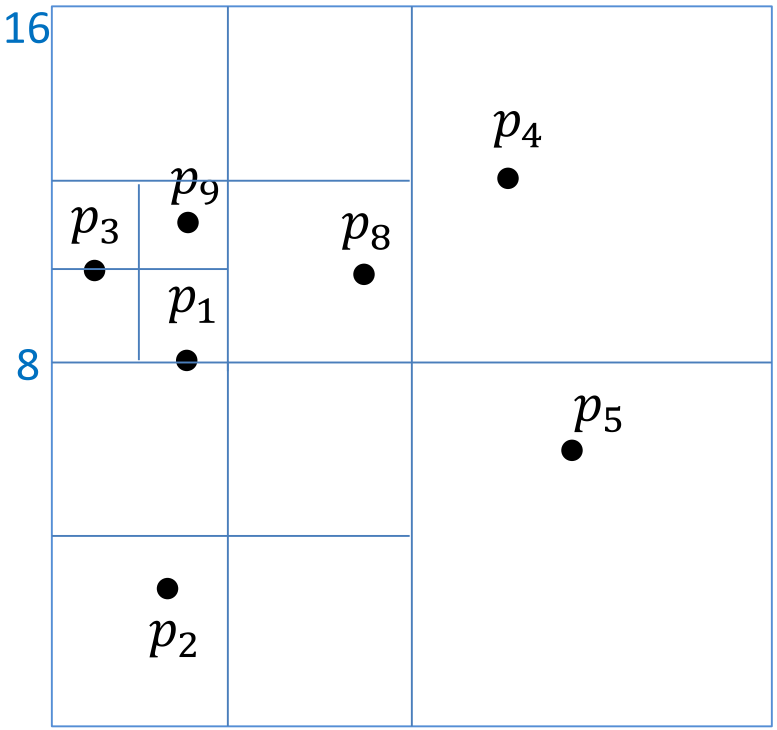
Quadtree Delete

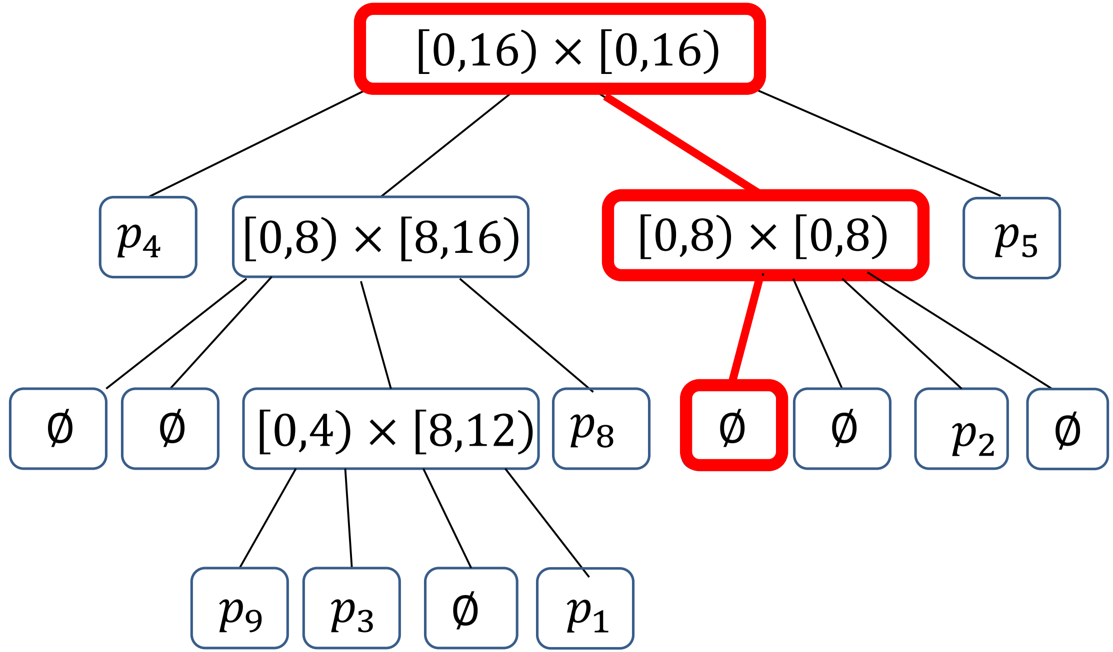




- search will find a leaf containing the point
  - example: delete(p_6)
- remove the point leaving the leaf empty

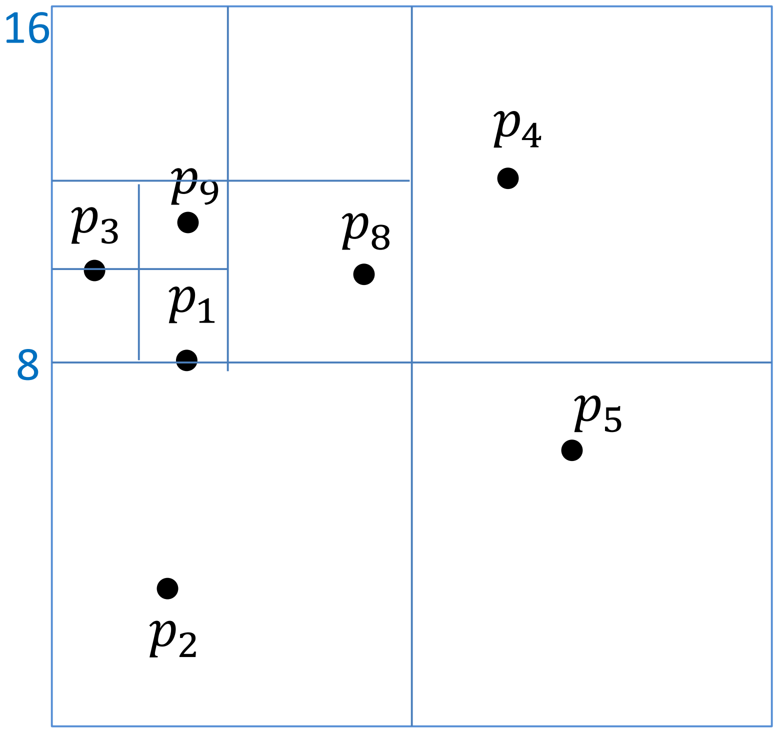
Quadtree Delete

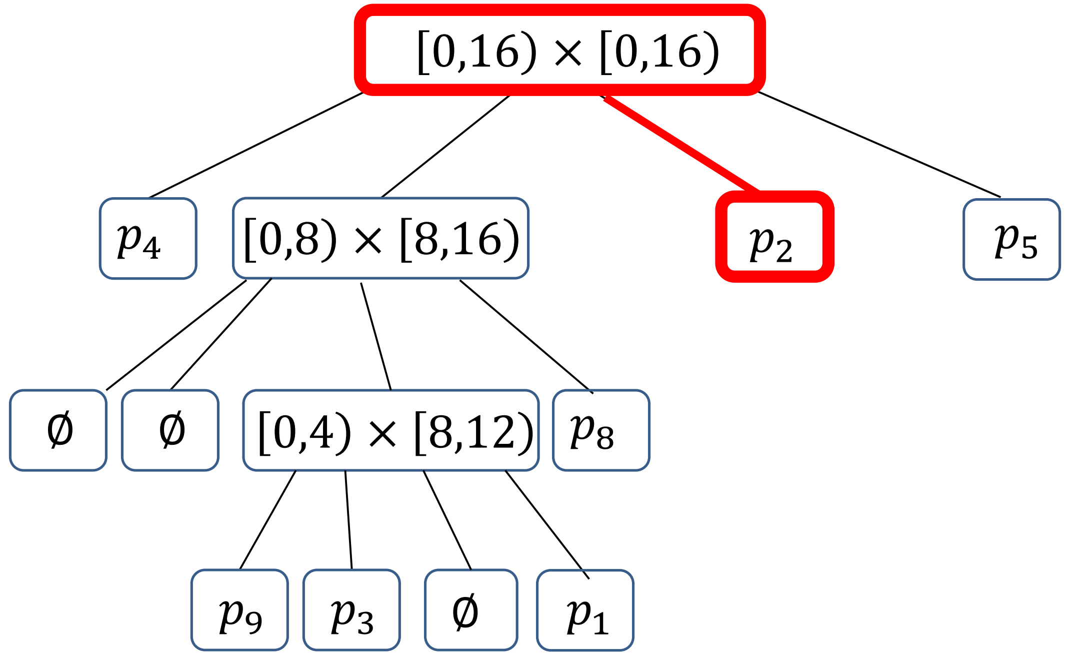




- search will find a leaf containing the point
    - example: delete(p_6)
- remove the point leaving the leaf empty
- if parent now stores only one point in its region
    - make parent node into a leaf storing its only child

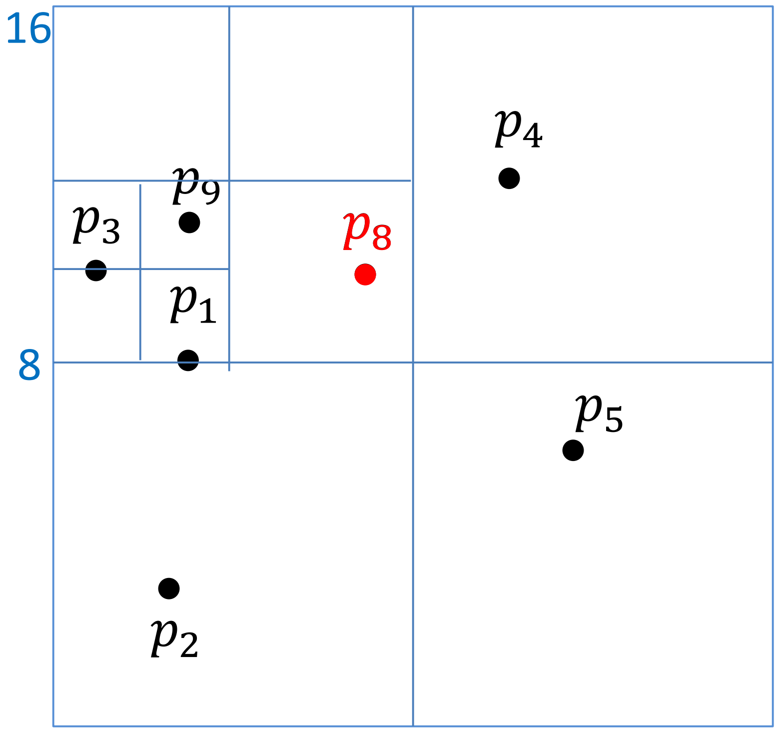
Quadtree Delete

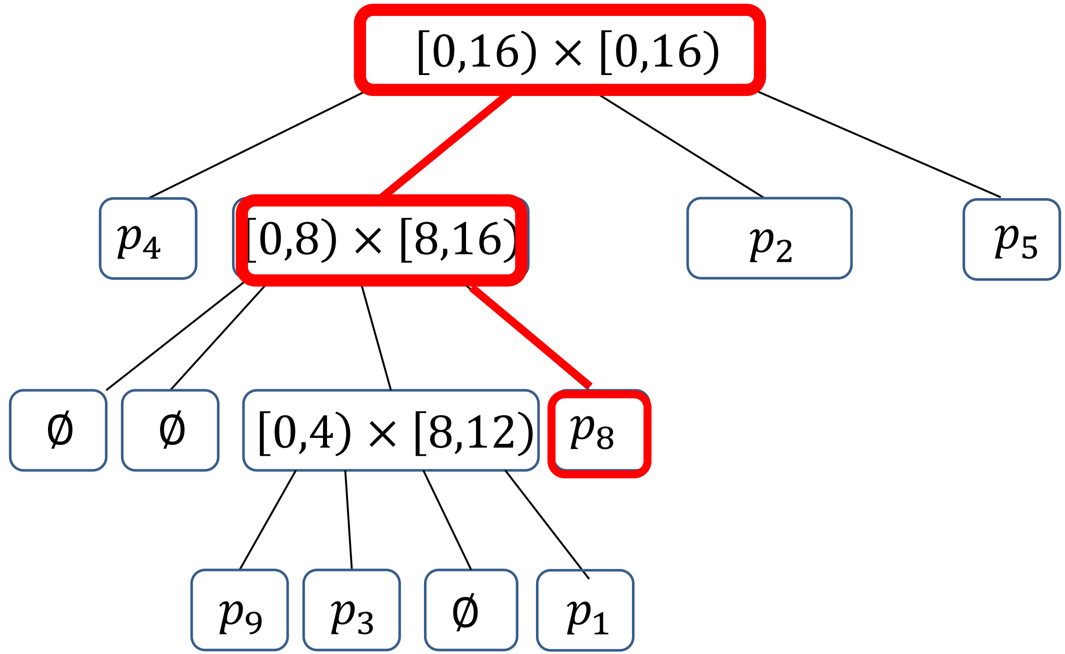




- search will find a leaf containing the point
  - example: delete(p_6)
- remove the point leaving the leaf empty
- if parent now stores only one point in its region
  - make parent node into a leaf
  - check up the tree, repeating making any parent with only 1 point into a leaf

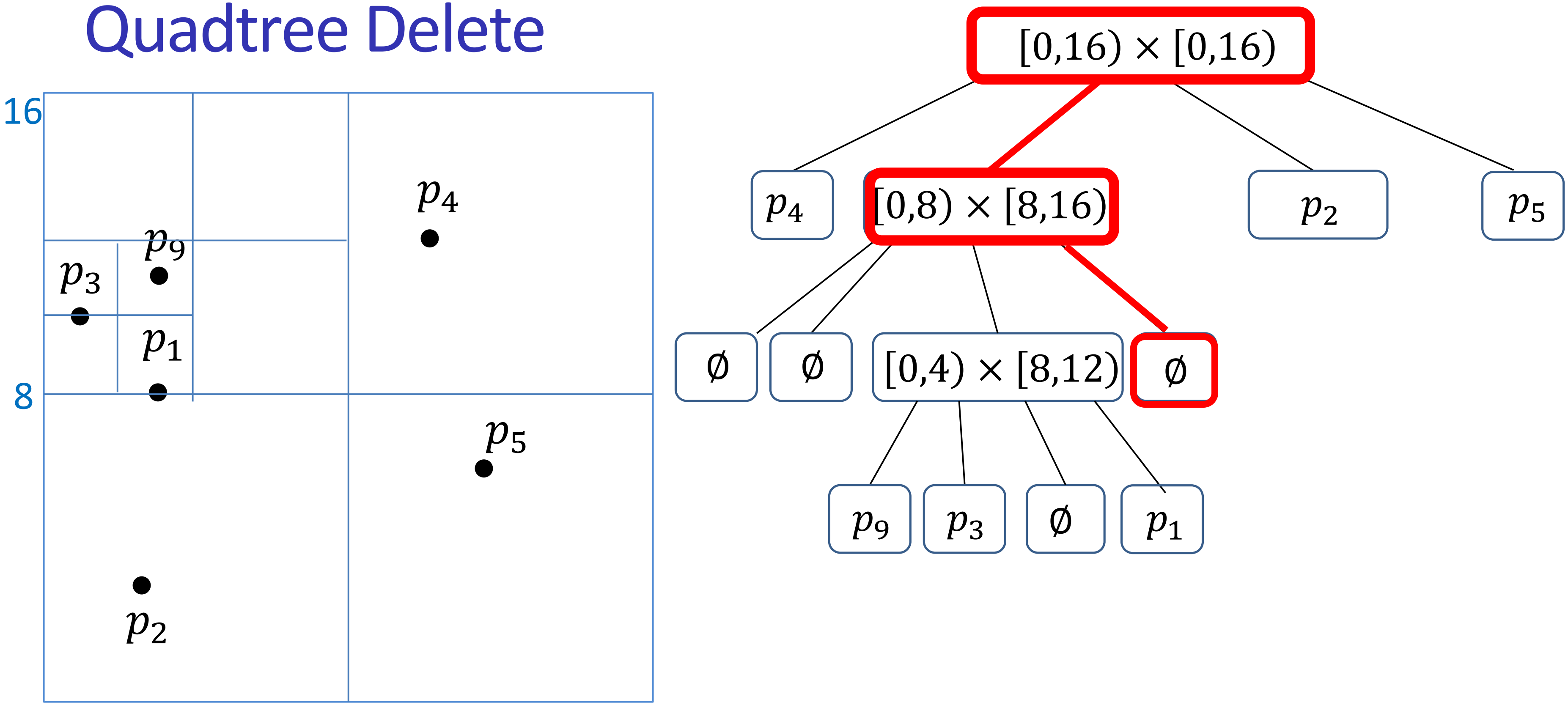
Quadtree Delete





- Another example: delete(p_8)

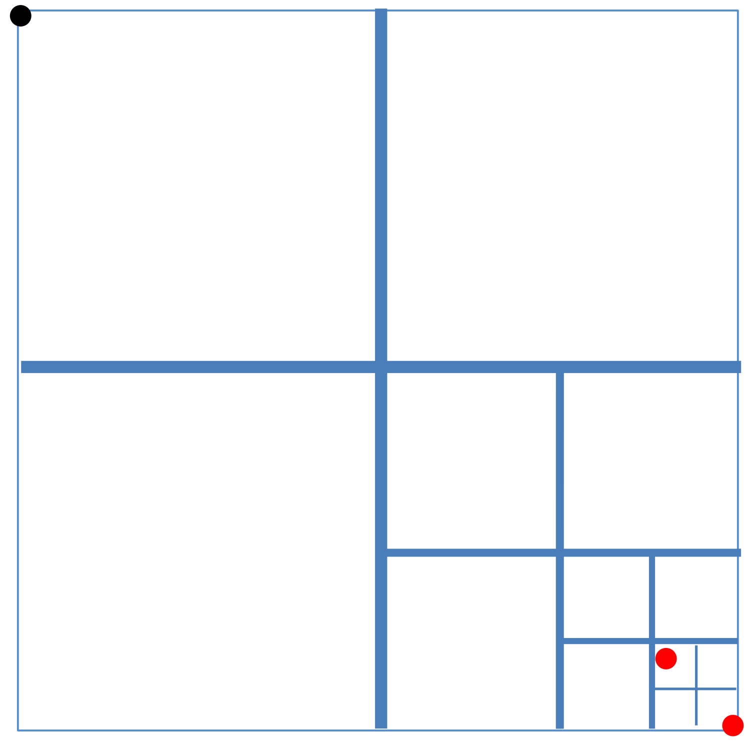
Quadtree Delete



- Do not make parent into a leaf as it stores multiple points

Quadtree Analysis

$$\text{height} = 4$$



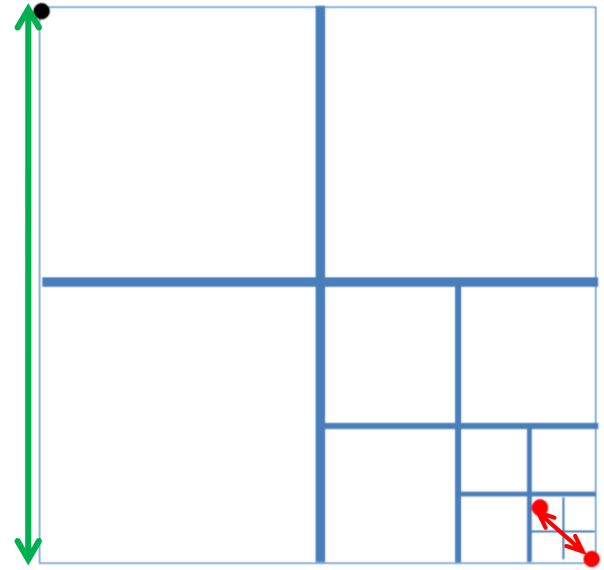
- Search, insert, delete depend on quadtree height
- What is the height of a quadtree?
  - can have very large height for bad distributions of points
  - example with just three points
  - can make height arbitrarily large by moving red points closer together

Quadtree Analysis

- spread factor of points S

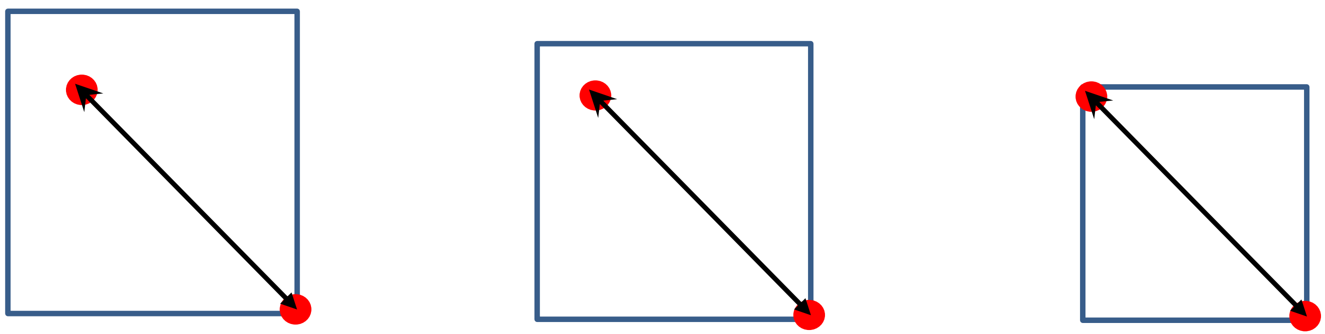
$$\rho(S) = \frac{L}{d_{min}}$$

  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S



- Worst case: height $h \in \Omega(\log \rho(S))$

red points are at at distance d_{min} from each other



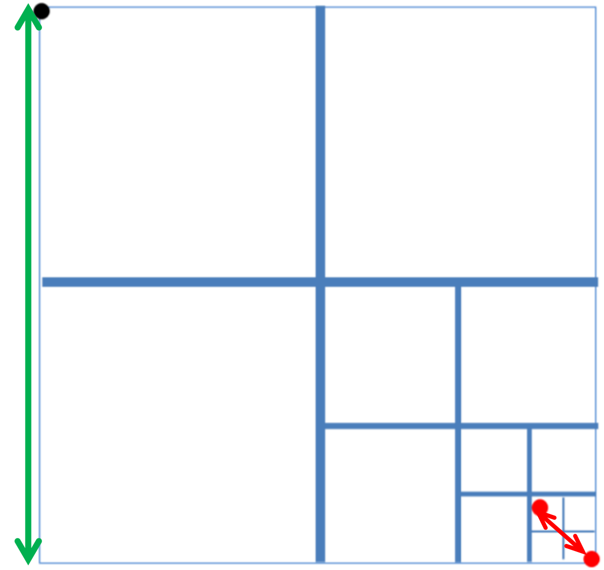
- While smallest region diagonal is $\geq d_{min}$, 2 red points are in same region

Quadtree Analysis

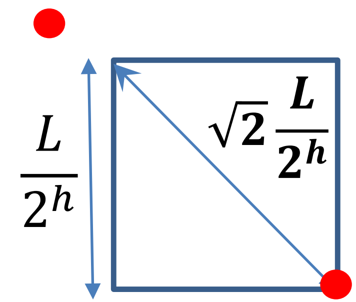
- spread factor of points S

$$\rho(S) = \frac{L}{d_{min}}$$

  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S



- Worst case: height $h \in \Omega(\log \rho(S))$
  - while smallest region diagonal is $\geq d_{min}$, 2 red points are in same region
  - if height is h, then we do h rounds of subdivisions
  - after h subdivisions, smallest regions have side length $\frac{L}{2^h}$
  - diagonal in smallest region is $\sqrt{2}\frac{L}{2^h}$
  - smallest region contains one red point $\Rightarrow \sqrt{2}\frac{L}{2^h} < d_{min}$
  - rearrange: $\sqrt{2}\frac{L}{d_{min}} < 2^h$
  - take log of both sides: $h > \log\left(\sqrt{2}\frac{L}{d_{min}}\right) = \log(\sqrt{2}\rho(S))$

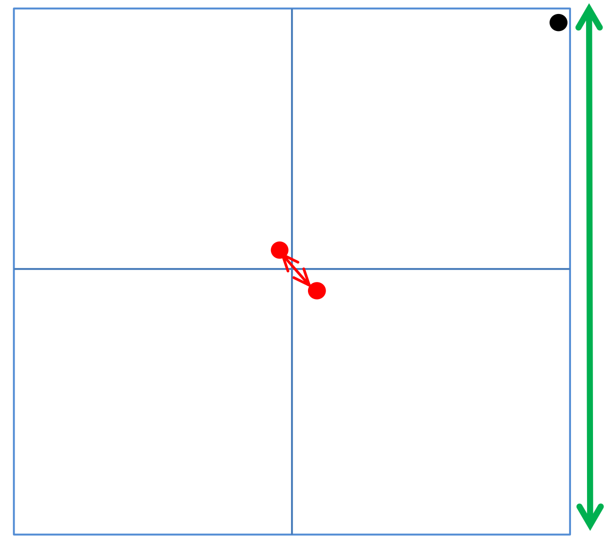


Quadtree Analysis

- spread factor of points S

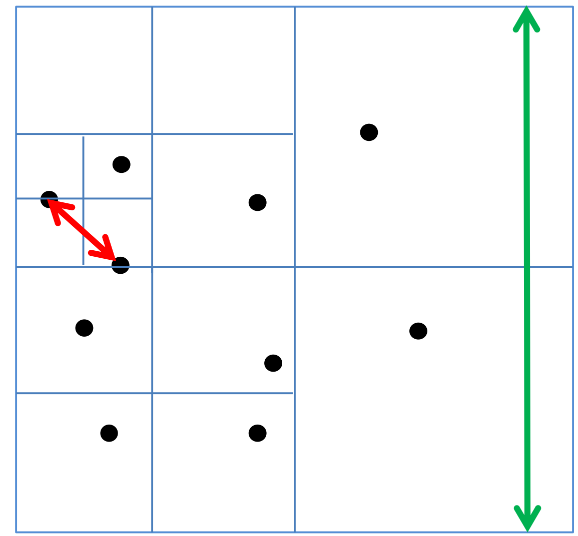
$$\rho(S) = \frac{L}{d_{min}}$$

  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S



- In the ***worst*** case, height $h \in \Omega(\log \rho(S))$
- However, height can be much better even if the spread is arbitrarily large

Quadtree Analysis

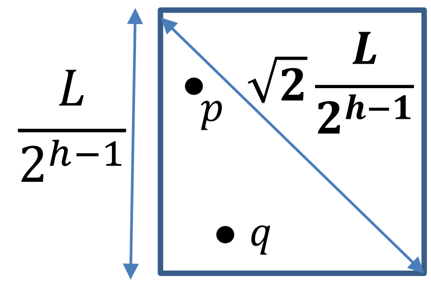


- spread factor of points S

$$\rho(S) = \frac{L}{d_{min}}$$

  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S

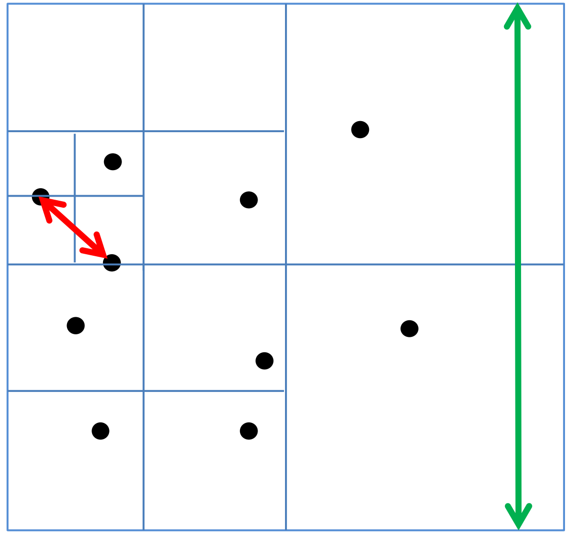
- In the worst case, height $h \in \Omega(\log \rho(S))$
- In **any case**, height $h \in O(\log \rho(S))$
  - let v be an internal node at depth $h-1$
    - there are at lest 2 points p, q inside its region
      - $d_{min} \le d(p,q)$
    - the corresponding region has side length $\frac{L}{2^{h-1}}$
    - maximum distance between 2 points in such region is $\sqrt{2}\frac{L}{2^{h-1}}$



$$d_{min} \le d(p,q) \le \sqrt{2}\frac{L}{2^{h-1}}$$

$$2^{h-1} \le \sqrt{2}\frac{L}{d_{min}} = \sqrt{2}\,\rho(S) \quad \Rightarrow h \le 1 + \log(\sqrt{2}\rho(S))$$

Quadtree Analysis



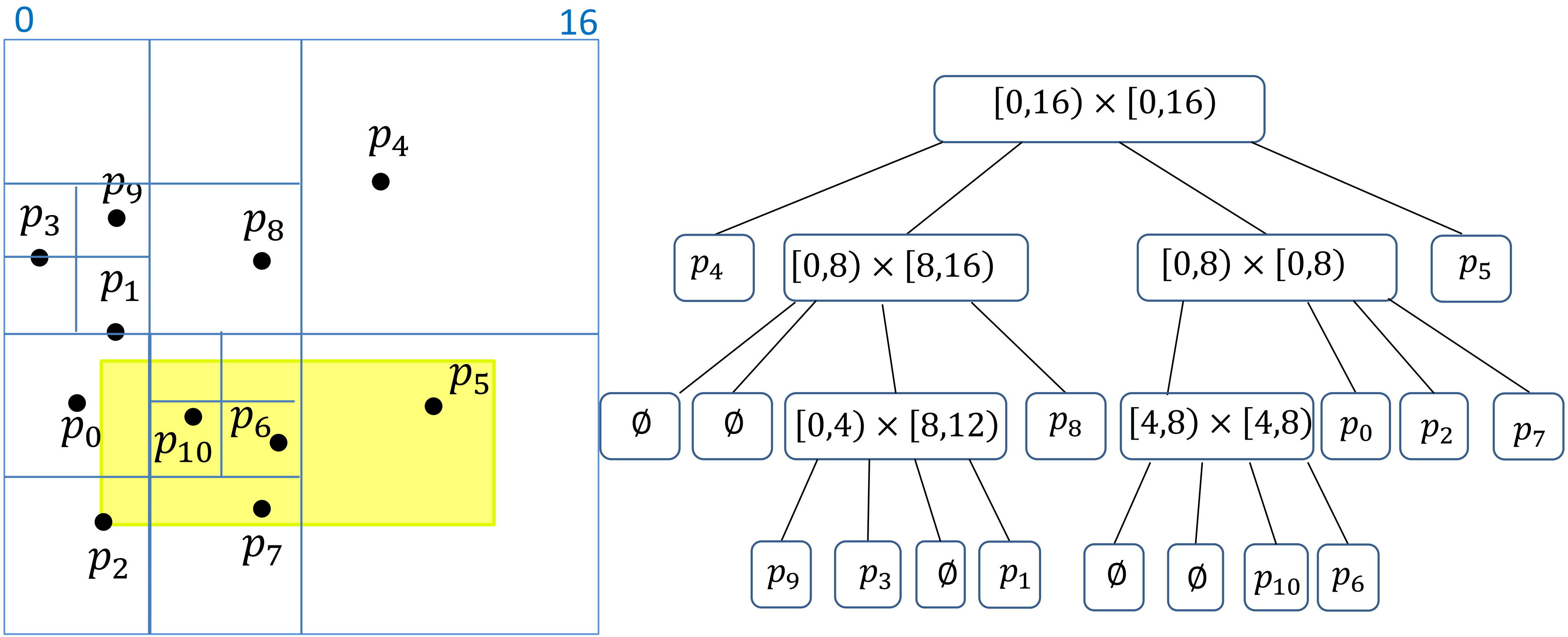
- spread factor of points S

$$\rho(S) = \frac{L}{d_{min}}$$

  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S

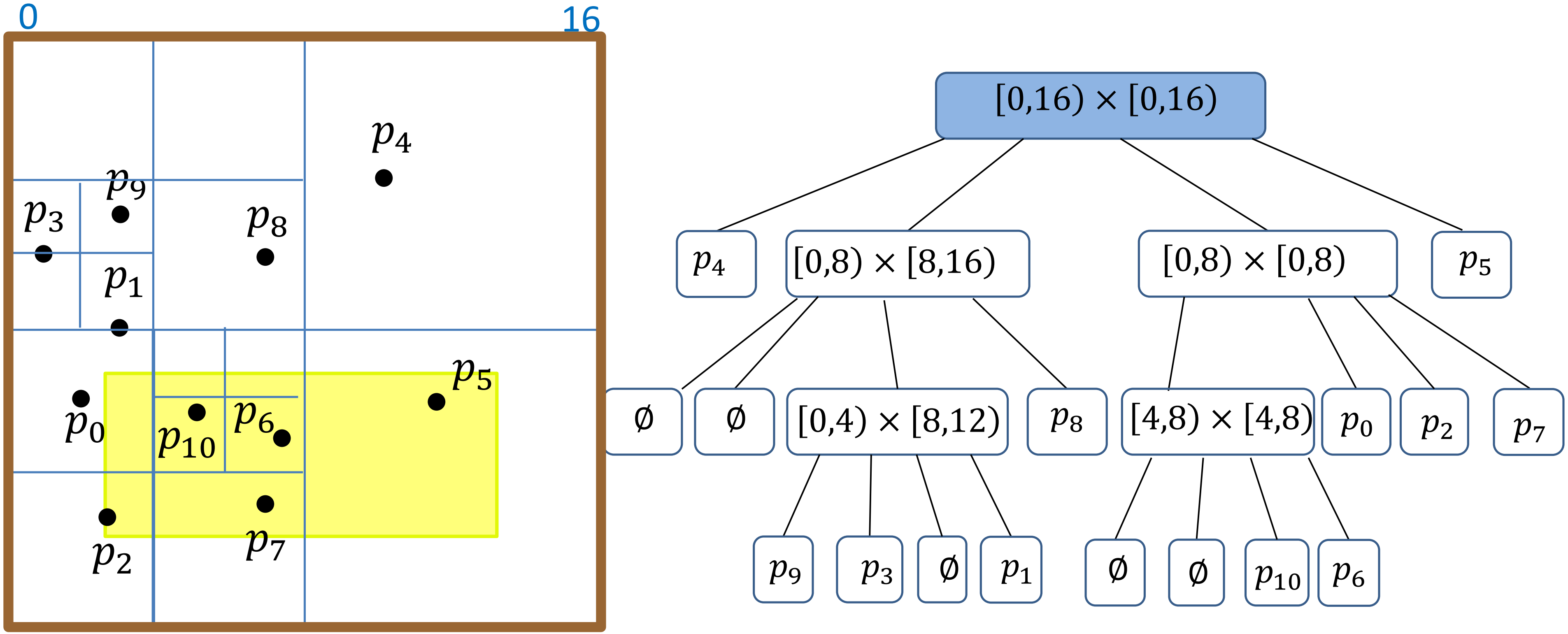
- In the worst case, height $h \in \Omega(\log \rho(S))$
- In any case, height $h \in O(\log \rho(S))$
    - to guarantee good performance, $\log \rho(S)$ should be much smaller than n
- Complexity to build initial tree: $\Theta(nh)$ worst-case
    - expensive if large height (as compared to the number of points)

Quadtree Range Search Example



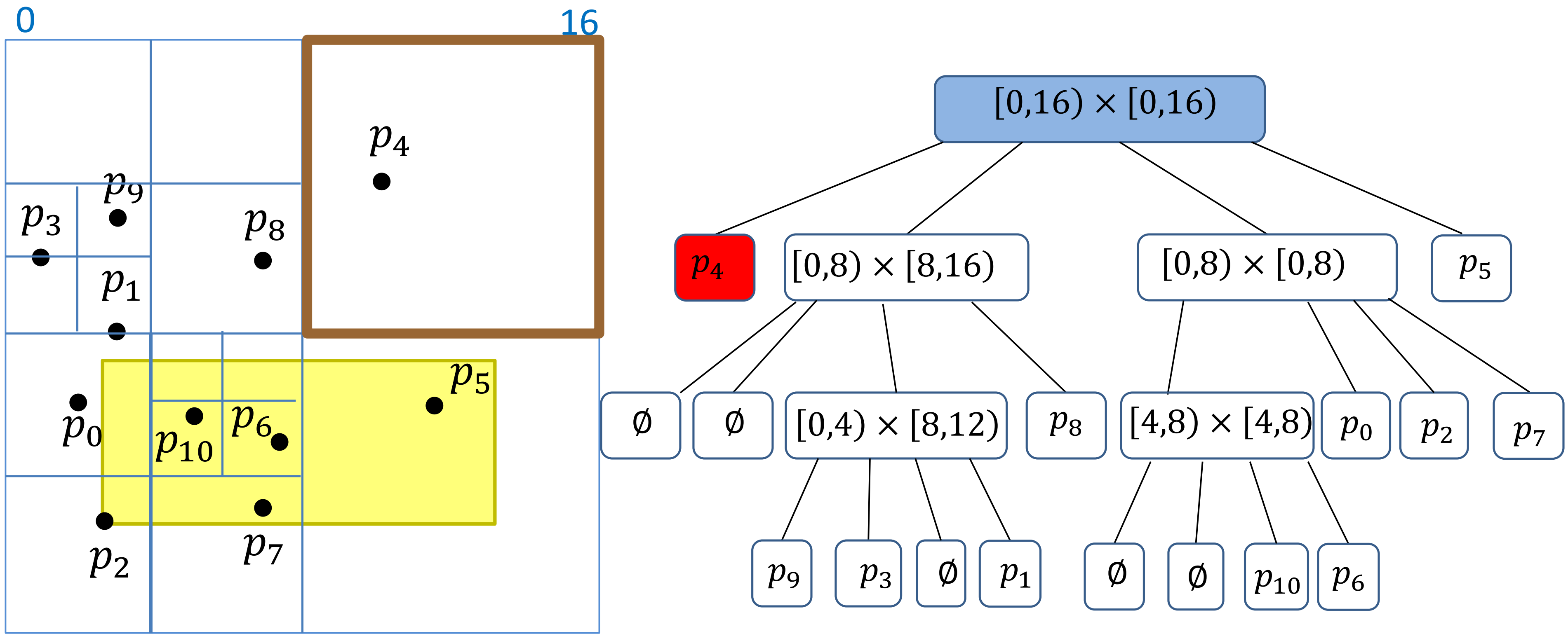
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



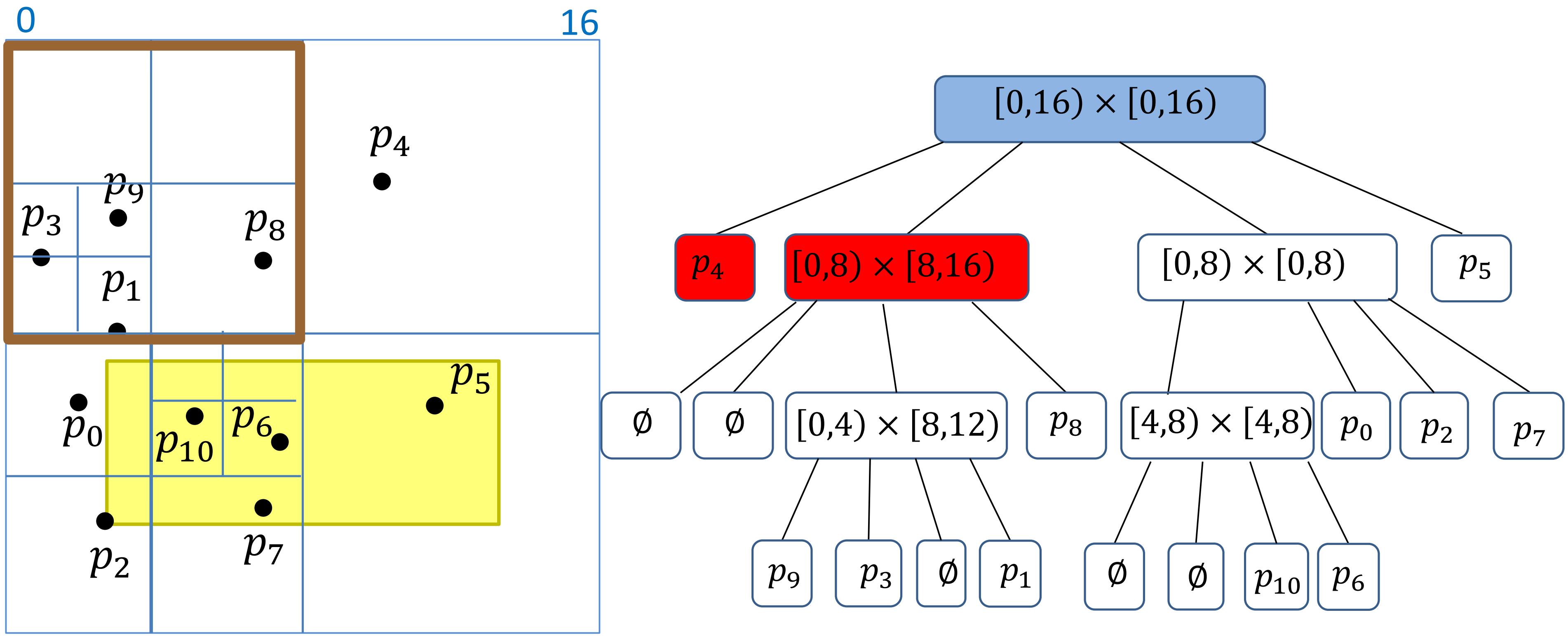
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



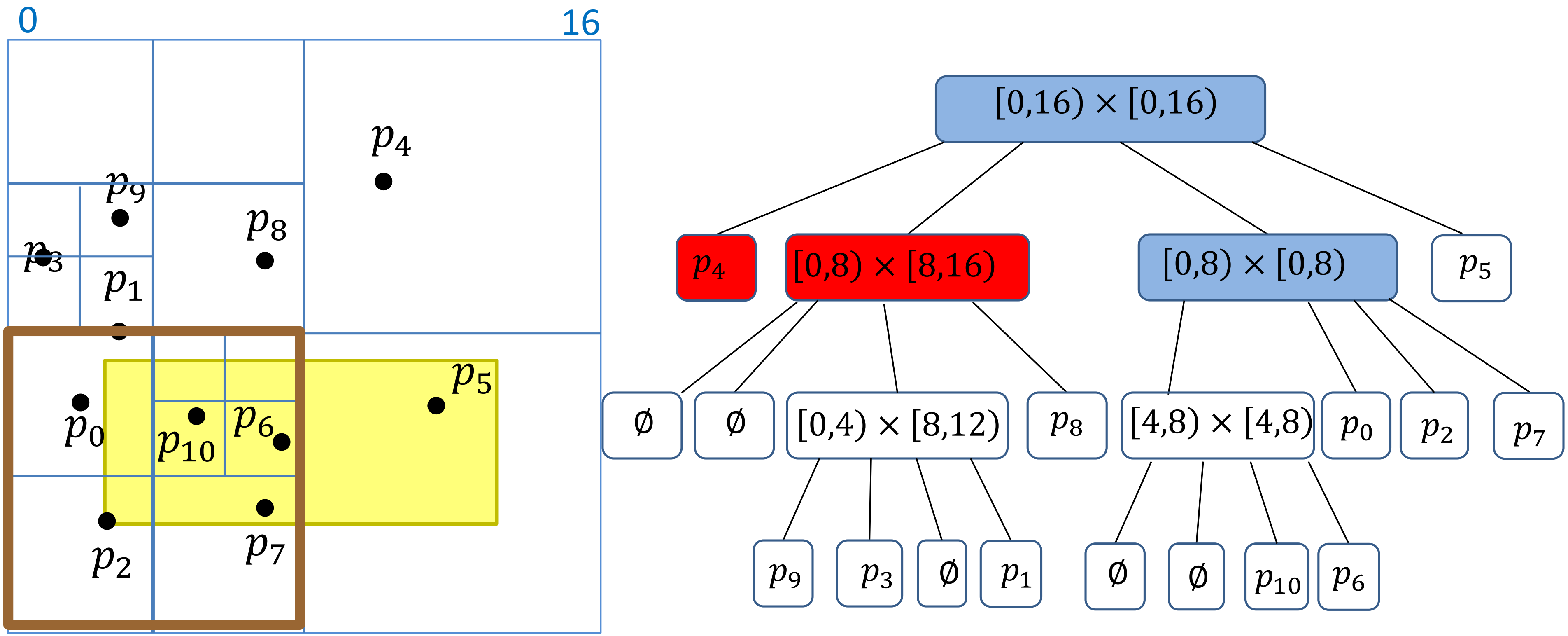
- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



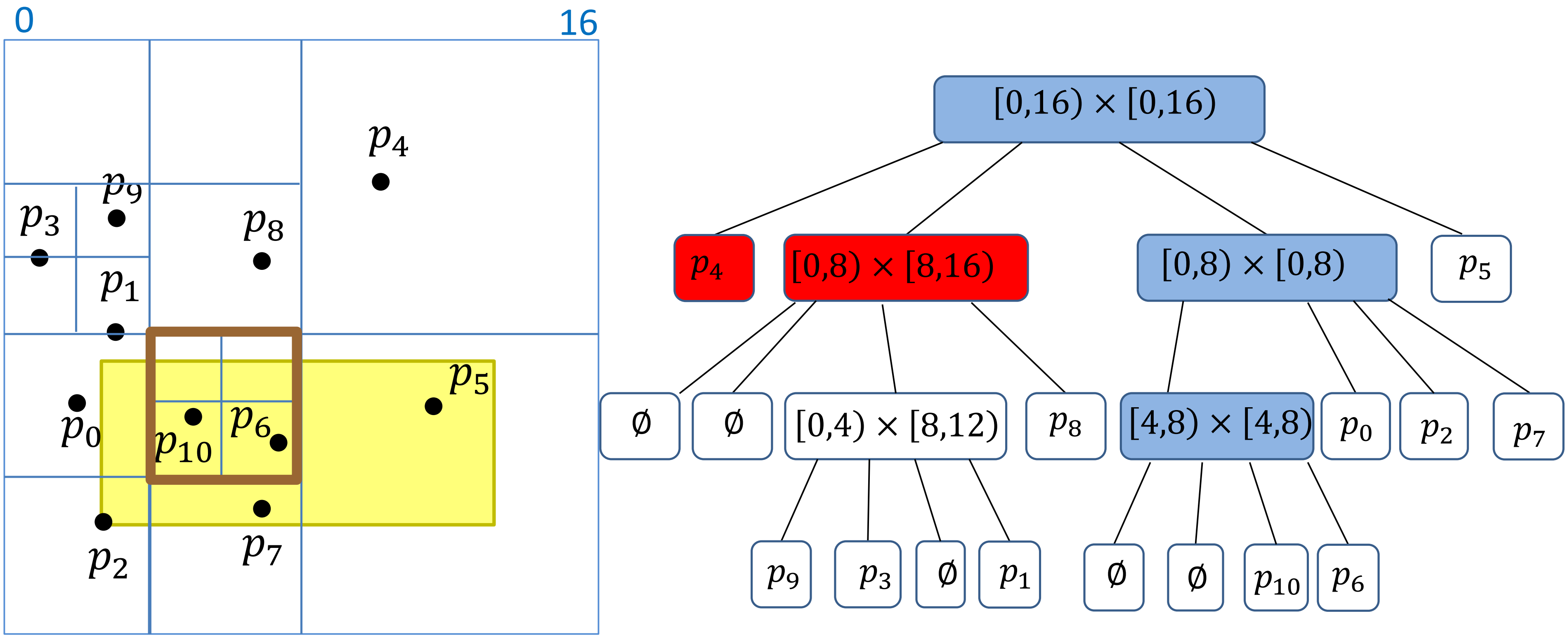
- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



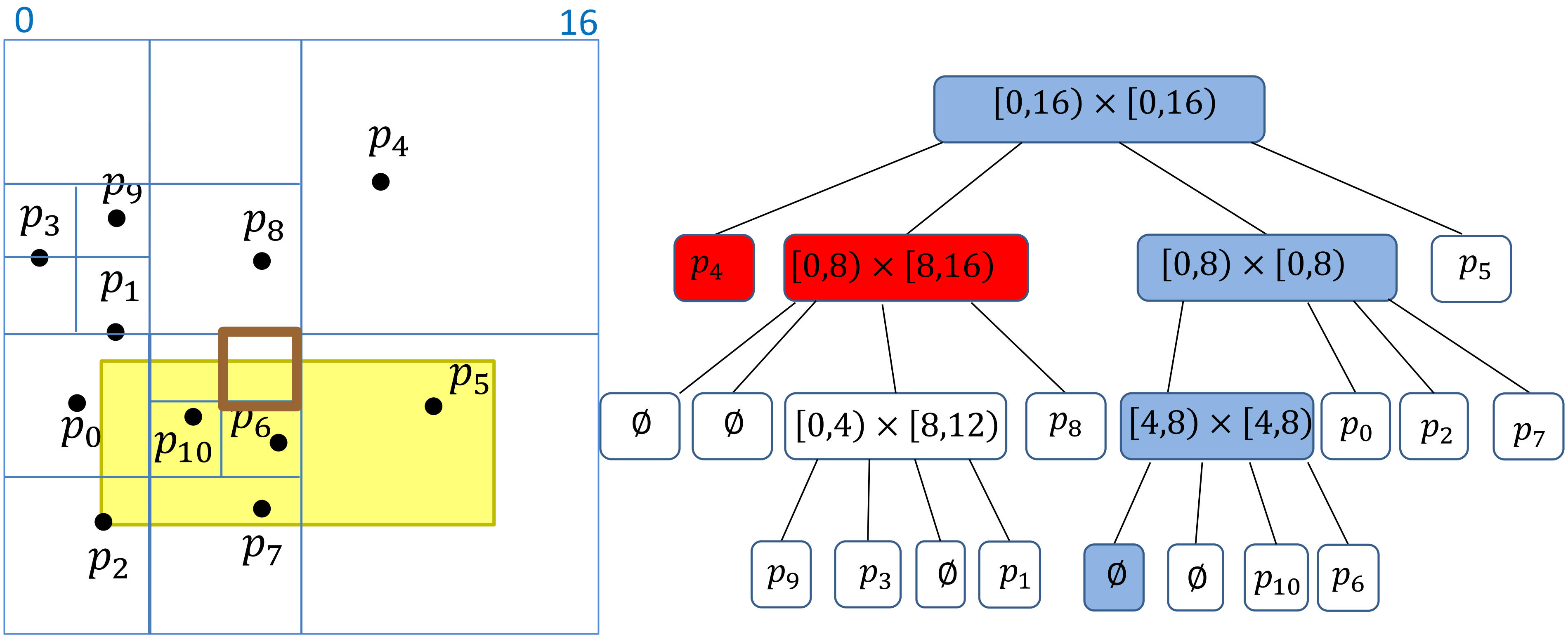
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



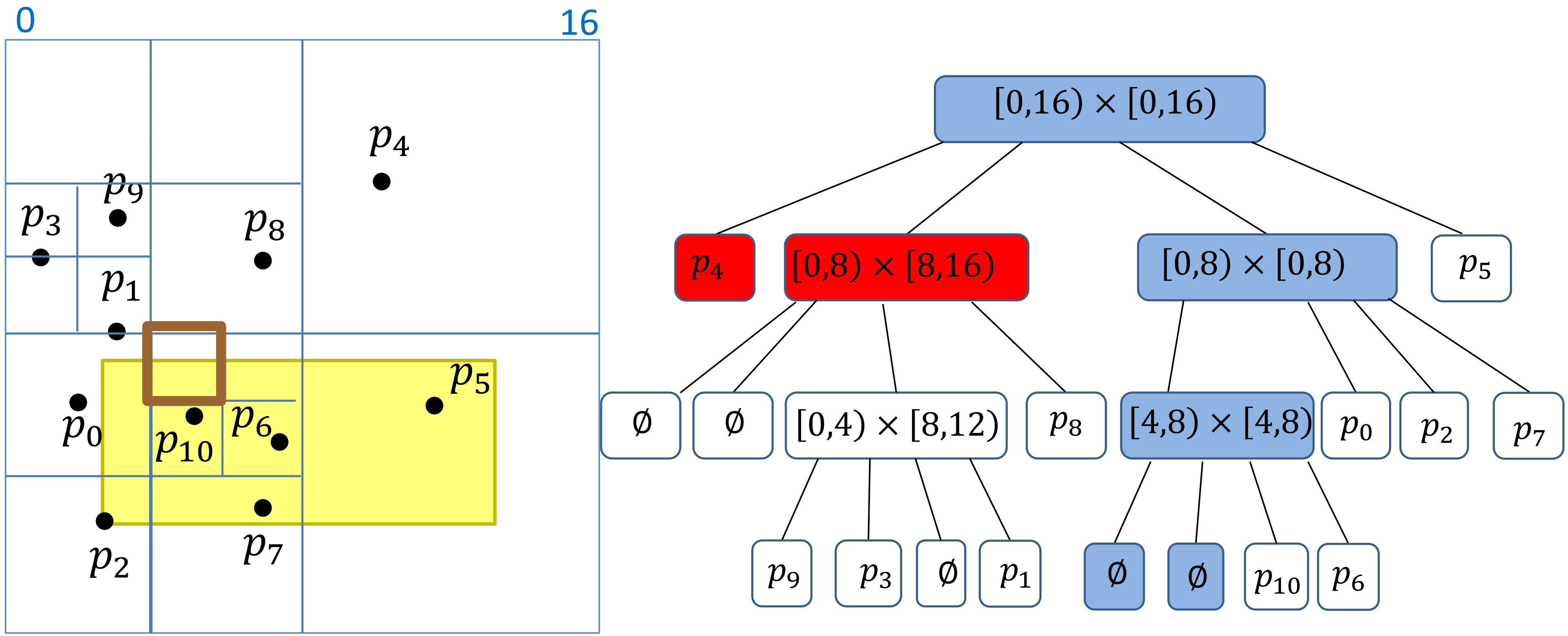
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



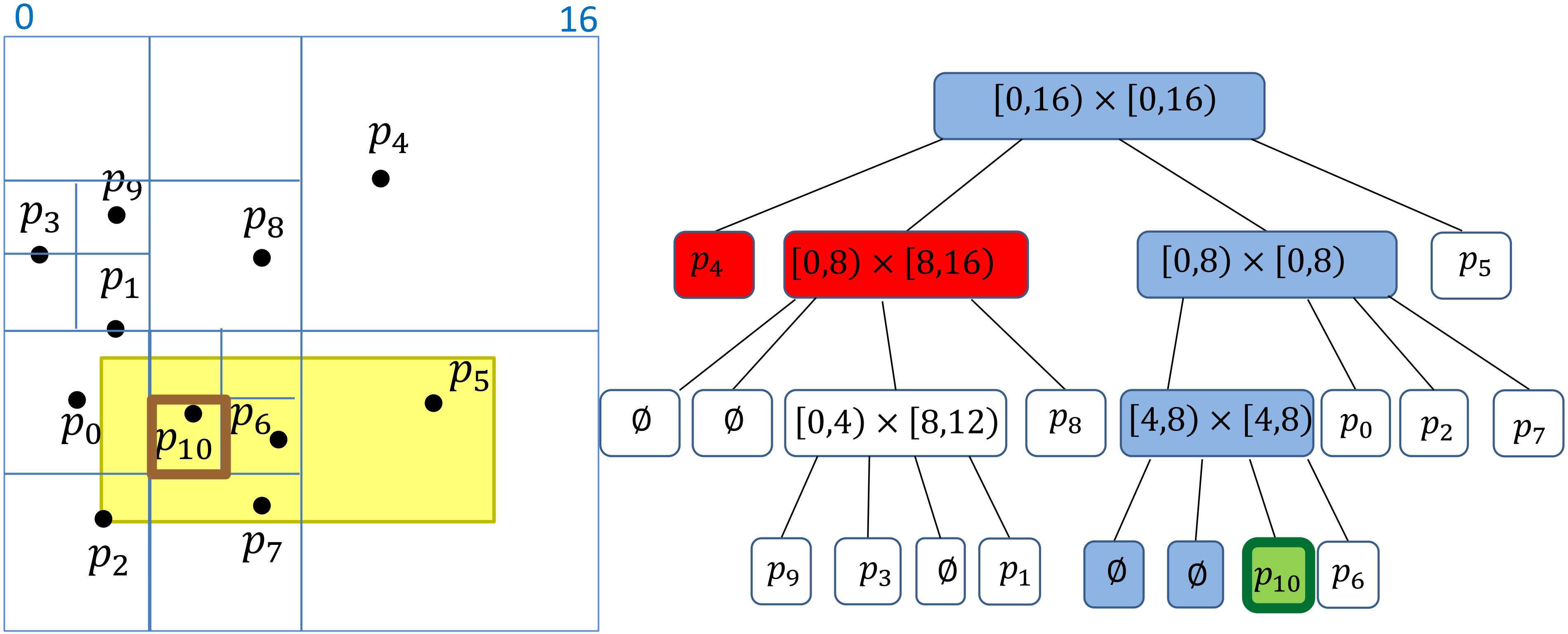
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



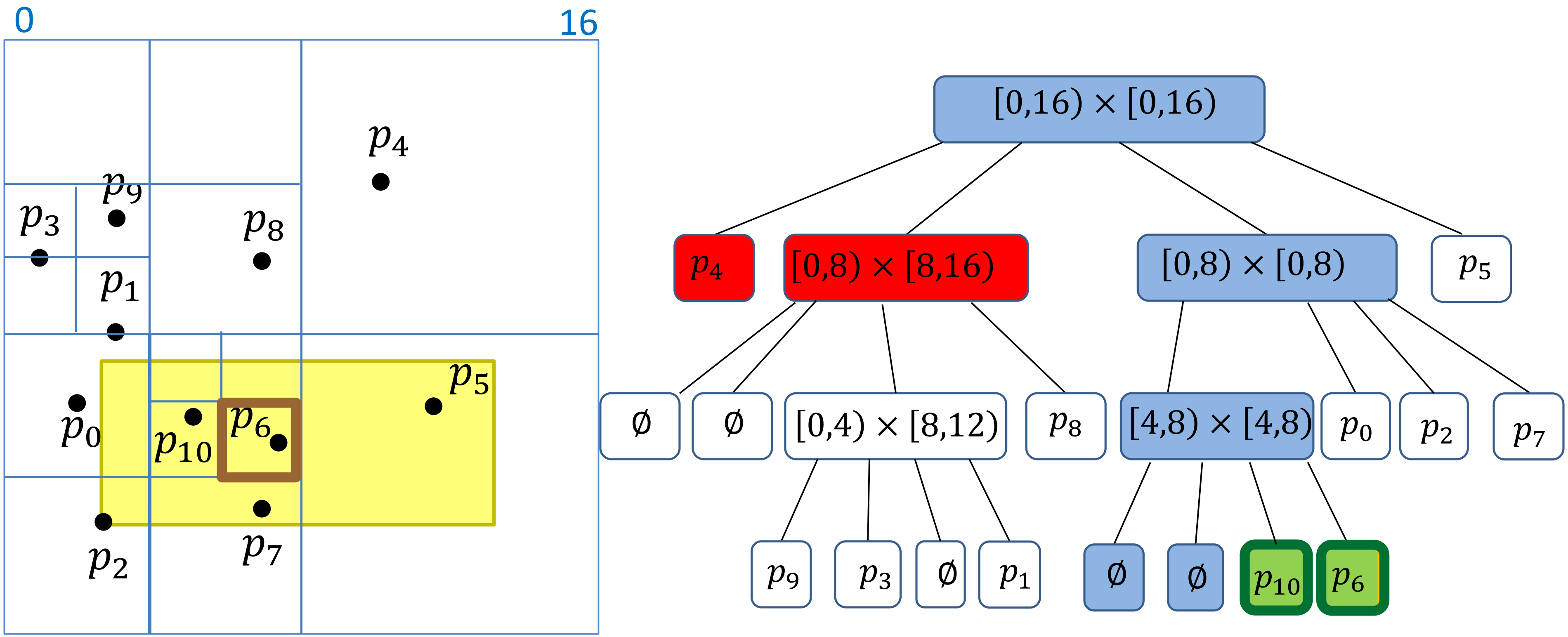
- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



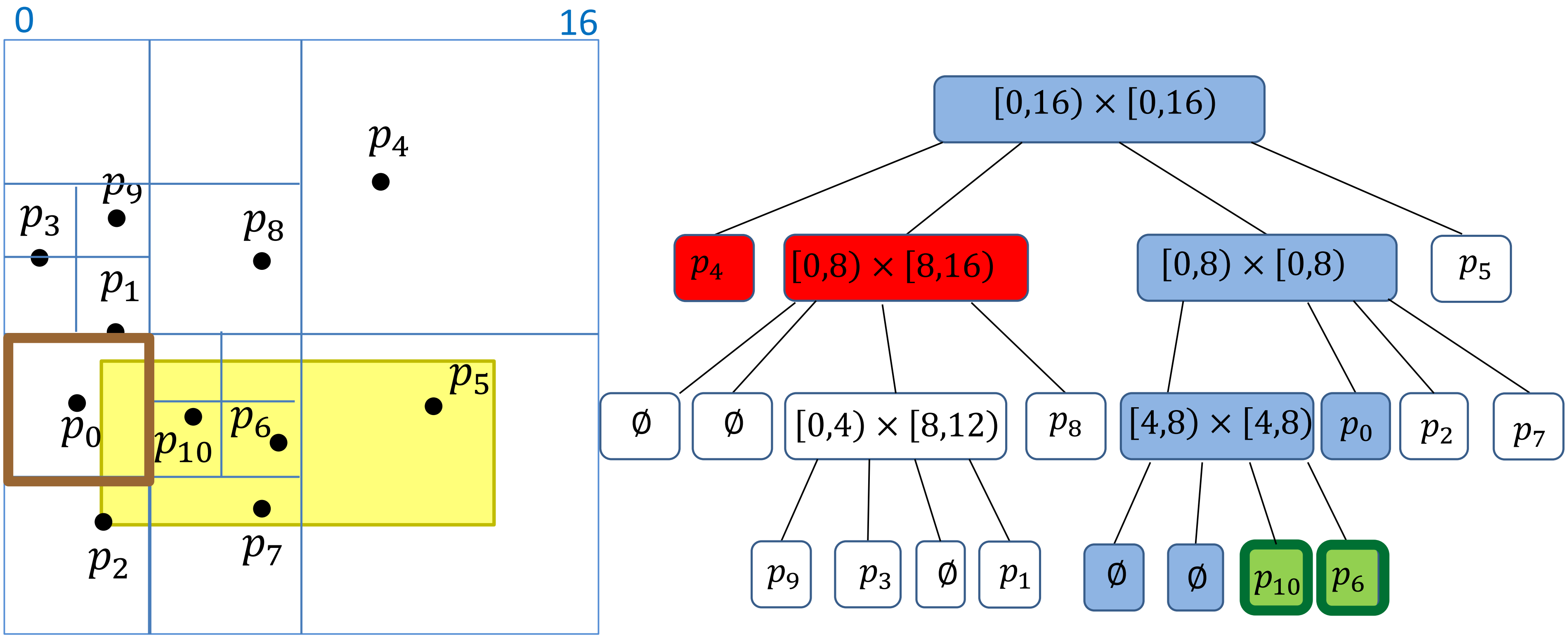
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



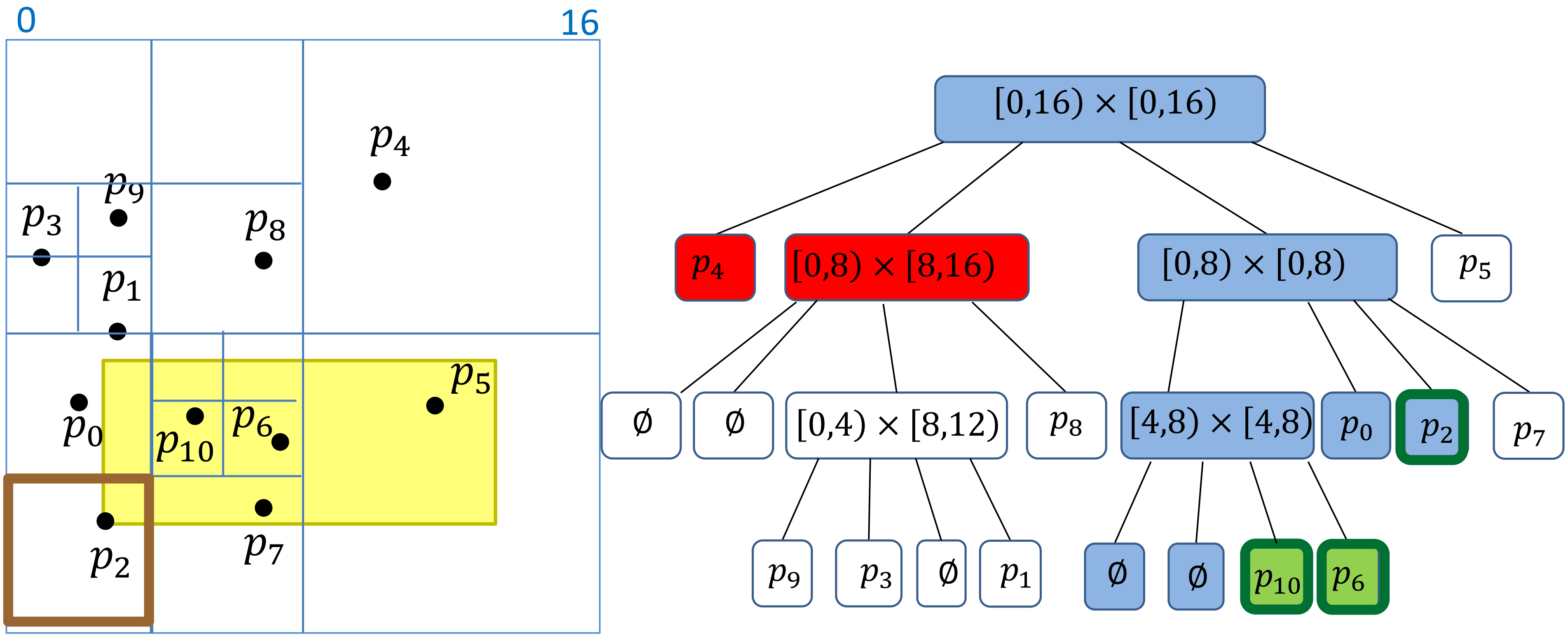
- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



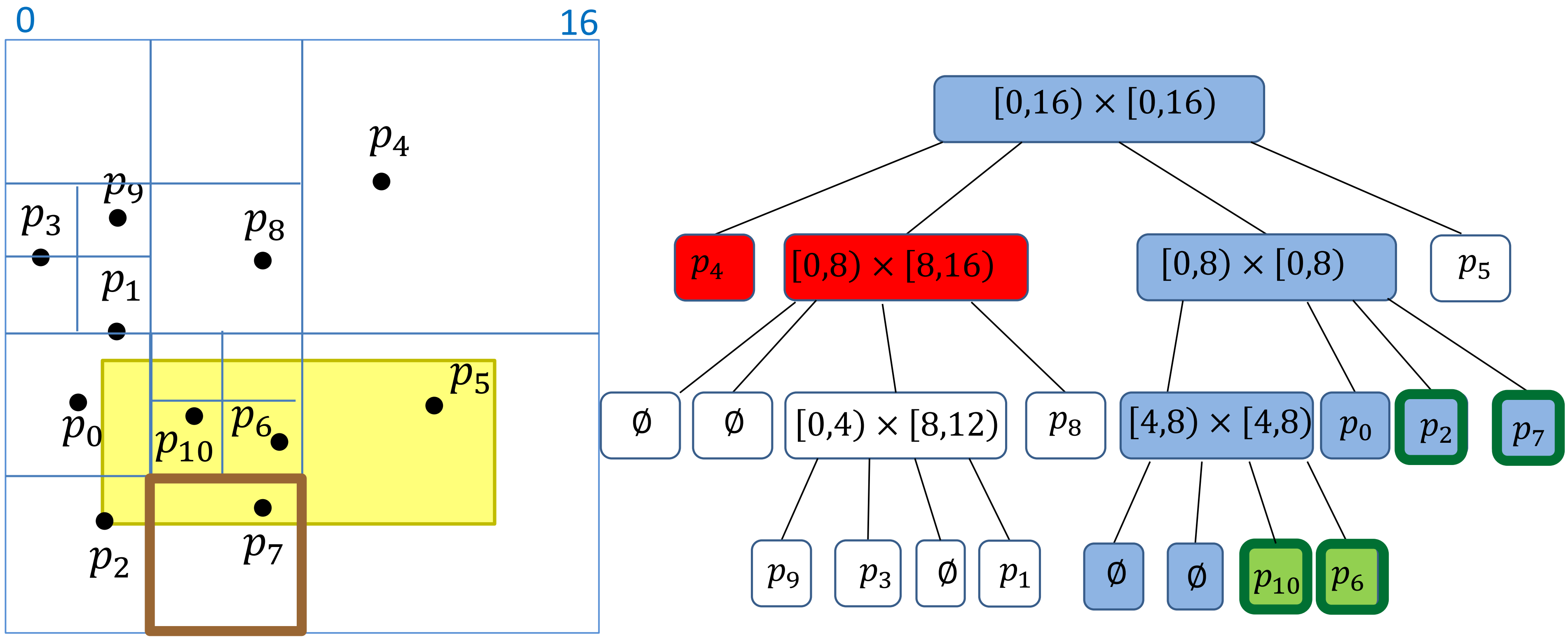
- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



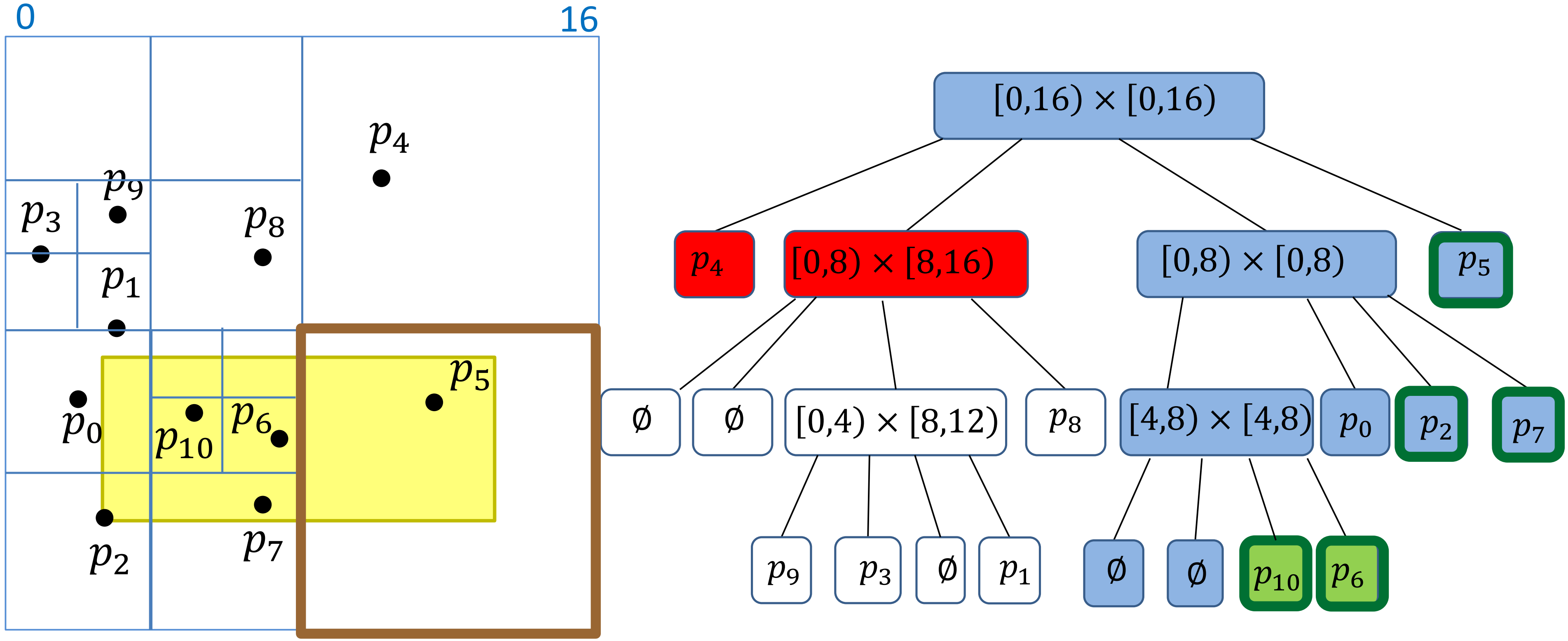
- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



- Query rectangle $Q = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search Example



- Query rectangle $Q = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

Quadtree Range Search

```
Qtree::RangeSearch(r ← root, Q)
r : quadtree root, Q: query rectangle
        let R be the region associated with r
        if R ⊆ Q then      //inside node, stop search
            report all points below r
            return
        if R ∩ Q = ∅ then //outside node, stop search
            return
        // boundary node, recurse if not a leaf
        if r is a leaf  then // leaf, do not recurse
            p ← point stored at r
            if p is not NULL and in Q return p
            else return
        for each child v of r do
            QTree-RangeSearch(v, Q)
```

- $R \subseteq Q, R \cap Q = \emptyset$ computed in constant time from coordinates of R, Q
- Code assumes each quadtree node stores the associated square
- Alternatively, these could be re-computed during search
    - space-time tradeoff

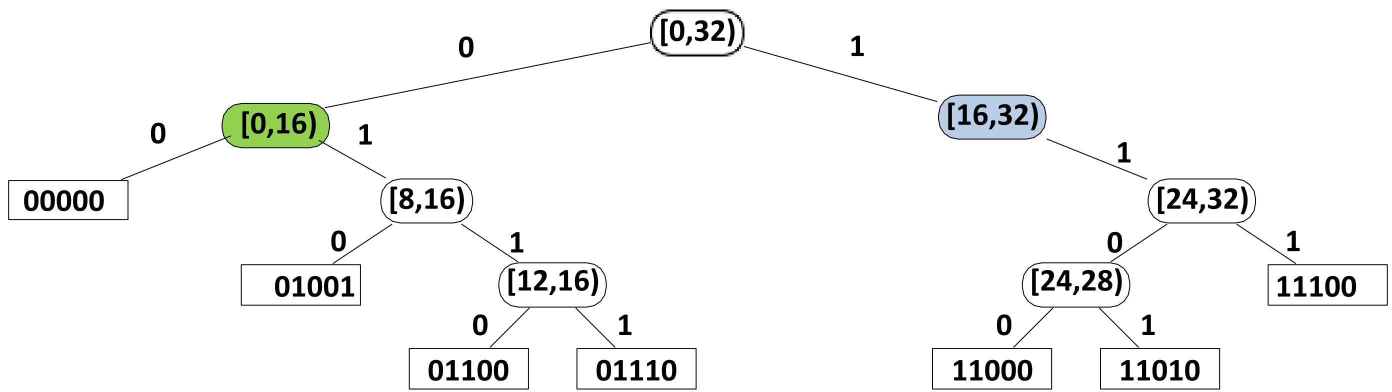
RangeSearch Analysis

- Running time is number of visited nodes + output size
- No good bound on number of visited nodes
    - may have to visit nearly all nodes in the worst case
    - $\Theta(nh)$ worst-case
        - this is worse than exhaustive search
        - even if the range search returns empty result
        - but in practice usually much faster

Quadtrees in other dimensions

points	0	9	12	14	24	26	28
base 2	00000	01001	01100	01110	11000	11010	11100

- Quad-tree of 1-dimensional points



- Same as a pruned trie
  - with splitting stopped once key is unique

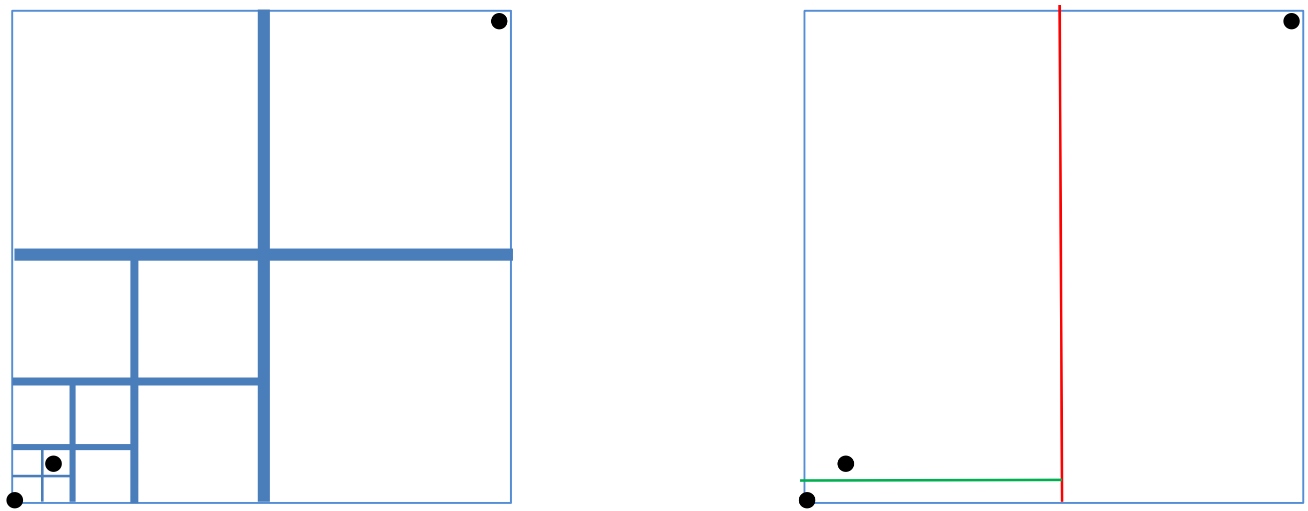
Quadtree summary

- Quadtrees easily generalize to higher dimensions
  - octrees, *etc.*
  - but rarely used beyond dimension 3
- Easy to compute and handle
- No complicated arithmetic, only divisions by 2
  - bit-shift if the width/height of R is a power of 2
- Space potentially wasteful, but good if points are well-distributed
- Variation
  - stop splitting earlier and allow up to k points in a leaf for some fixed k

Outline

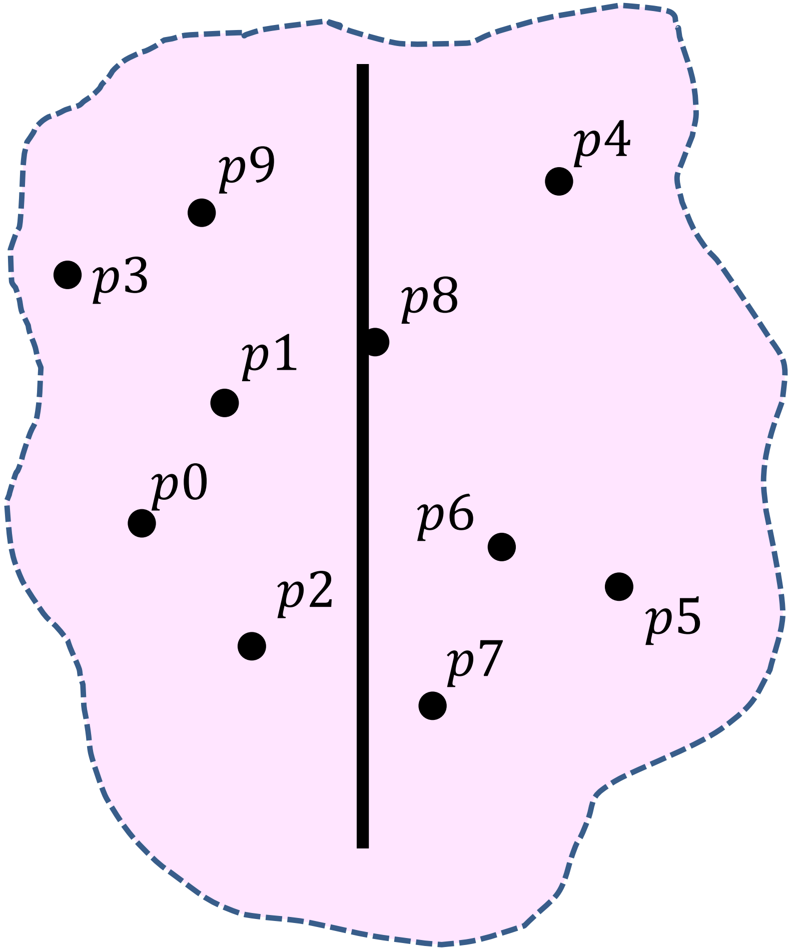
- Range-Searching in Dictionaries for Points
  - Range Search Query
  - Multi-Dimensional Data
  - Quadtrees
  - kd-Trees
  - Range Trees
  - Conclusion

kd-tree motivation



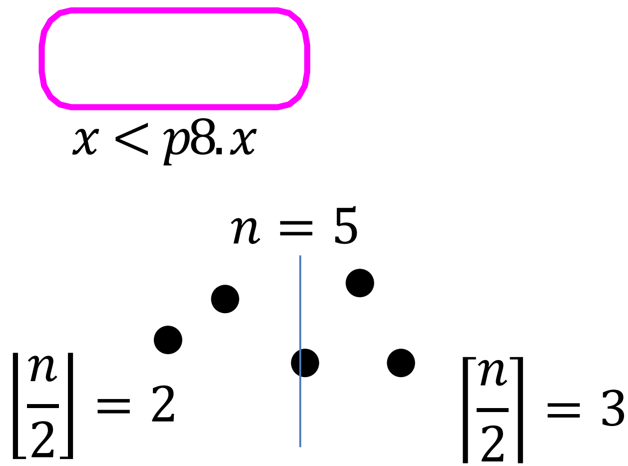
- Quadtree can be very unbalanced
- kd-tree idea
  - split into regions with equal number of points
  - easier to split into two regions with equal number of points (rather than four regions)
  - can split either vertically or horizontally
  - alternating vertical and horizontal splits gives range search efficiency

kd-tree example



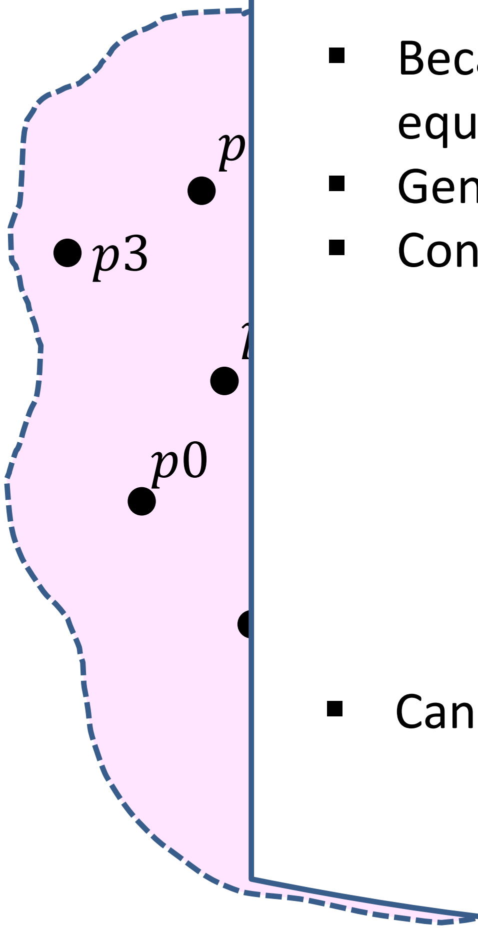
$\mathcal{R}^2$ is split into two half regions

- No need for bounding box
- Root corresponds to the whole $\mathcal{R}^2$
- First find the best vertical split
- $\left\lfloor \frac{n}{2} \right\rfloor$ on one side and $\left\lceil \frac{n}{2} \right\rceil$ and points on the other

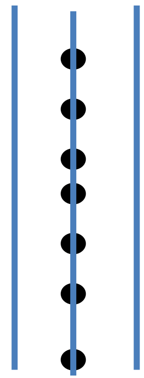


- $m = \left\lfloor \frac{n}{2} \right\rfloor$ in sorted list of x -coordinates
- partition S into $S_{x<m}$ and $S_{x\geq m}$

kd-tree example



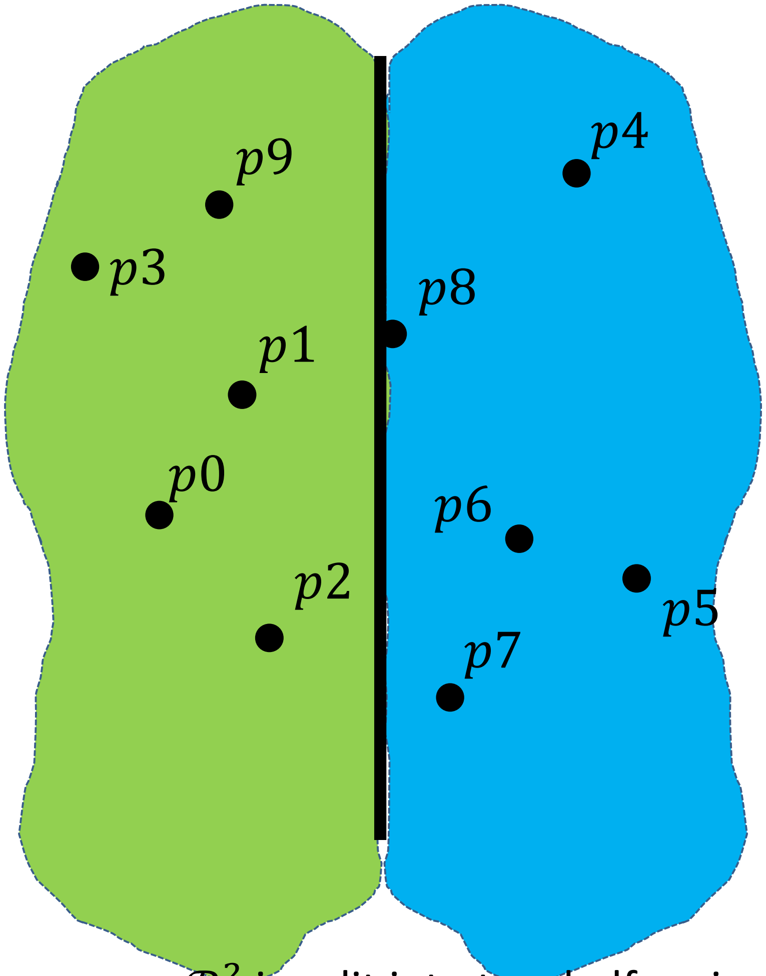
- Because points are in general position, always can split in two equal (or almost equal subsets)
- General position means no two x or y coordinates are the same
- Consider the points below **not** in general position



- Cannot divide them in two equal subsets by a vertical line

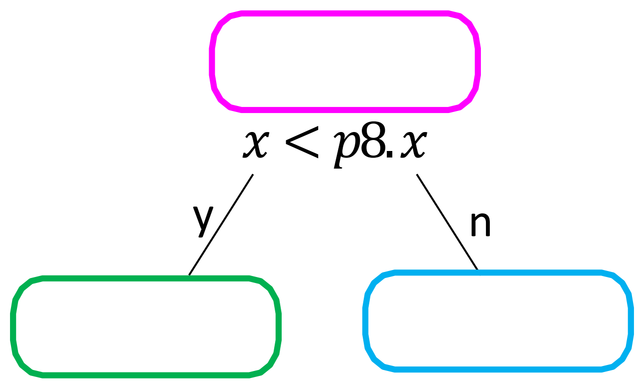
$\mathcal{R}^2$ is split into two half regions

kd-tree example

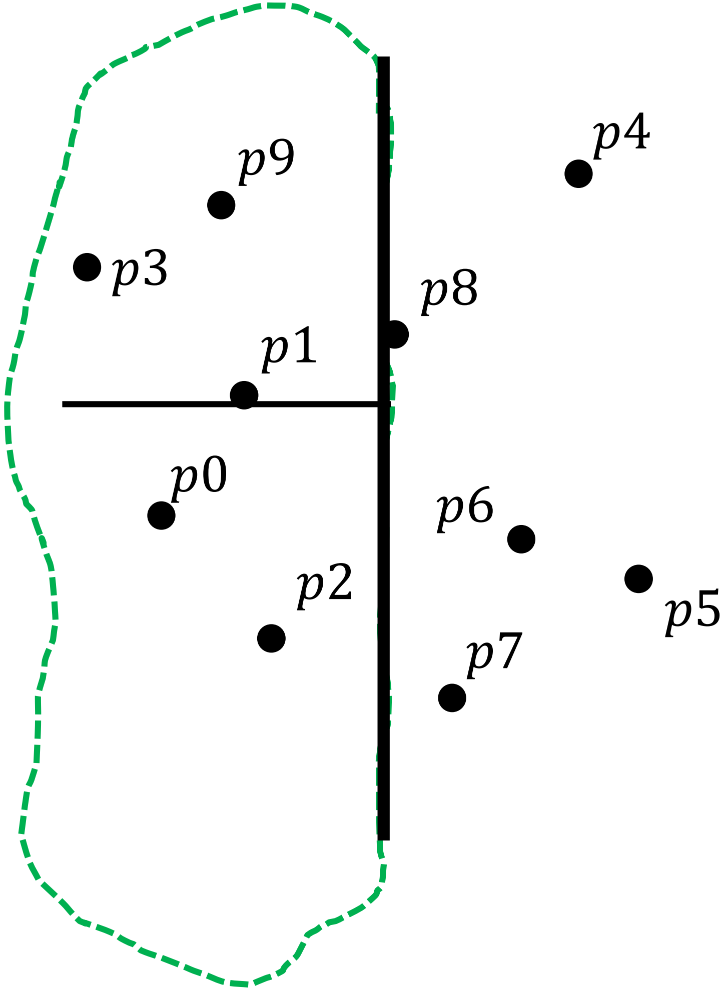


$\mathcal{R}^2$ is split into two half regions

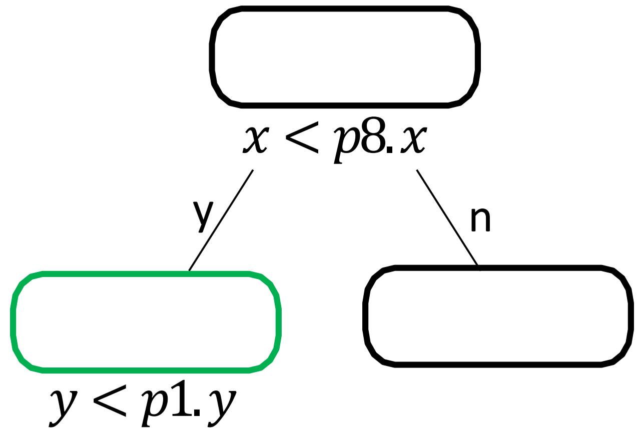
- No need for bounding box
- Root corresponds to the whole $\mathcal{R}^2$
- First find the best vertical split
- $\left\lfloor \frac{n}{2} \right\rfloor$ on one side and $\left\lceil \frac{n}{2} \right\rceil$ and points on the other



kd-tree example

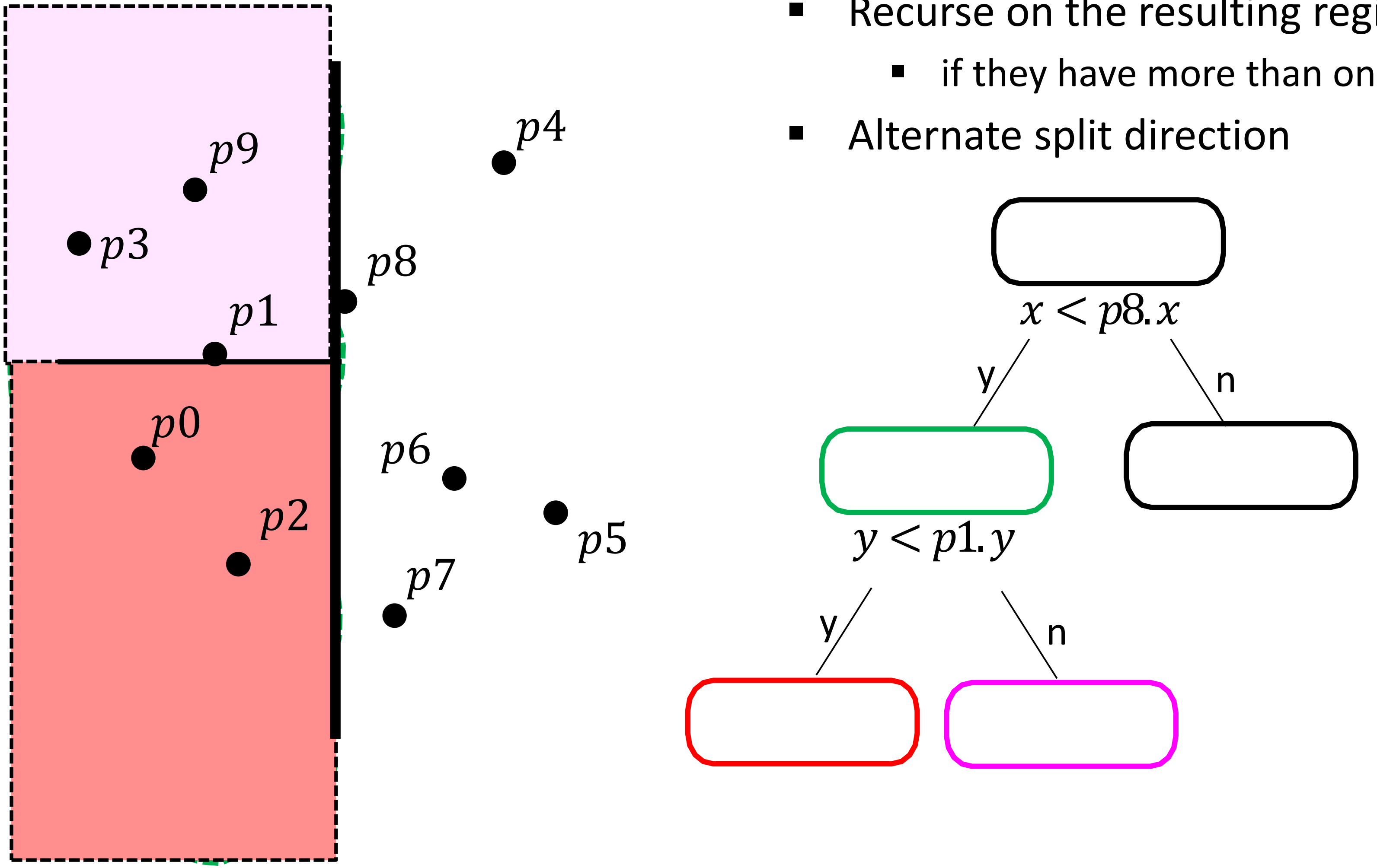


- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



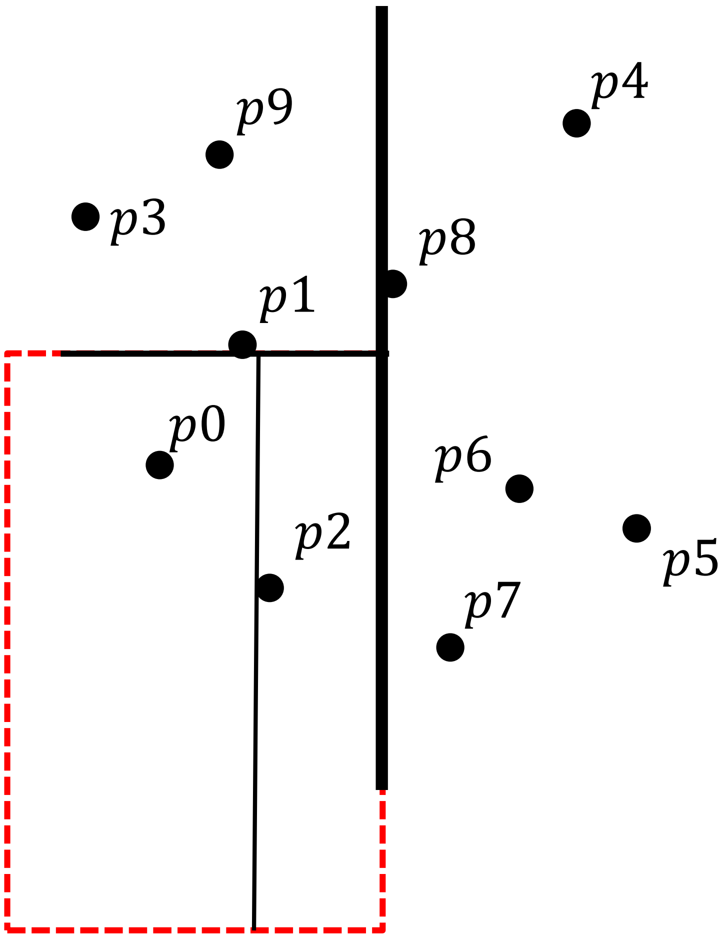
kd-tree example

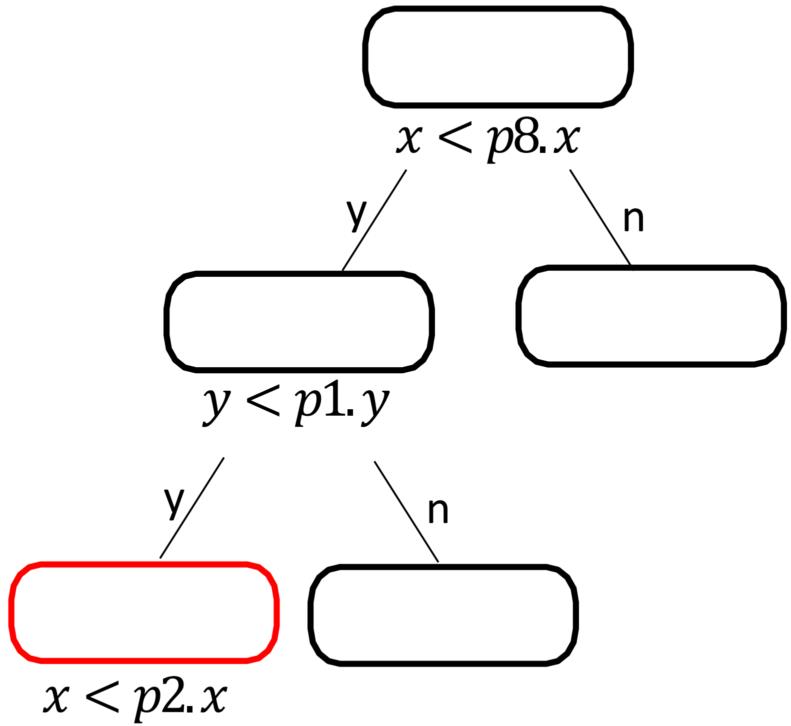
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



kd-tree example

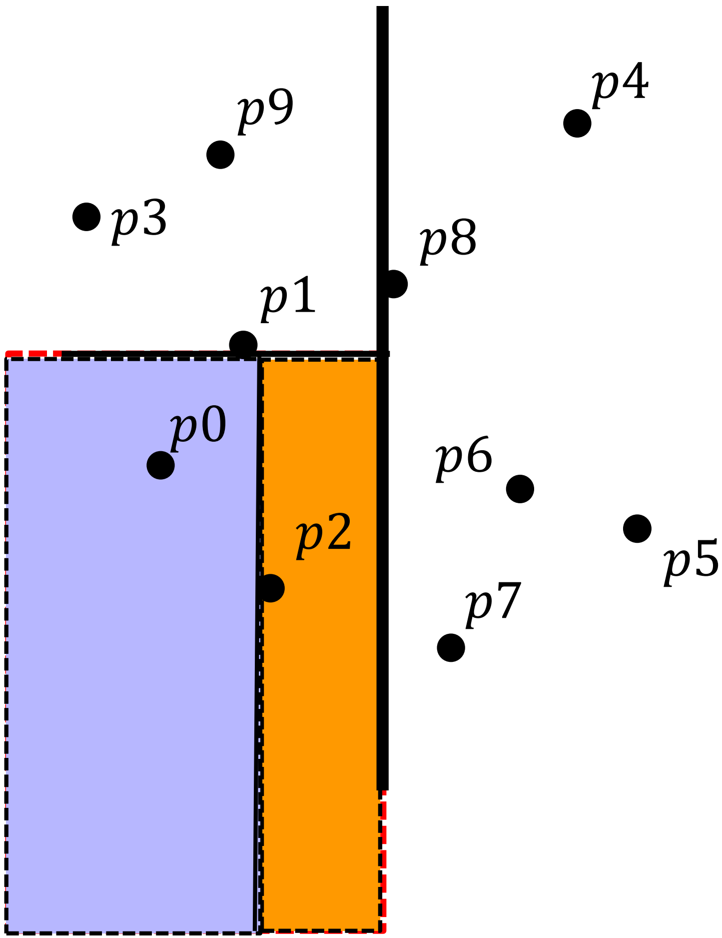
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction

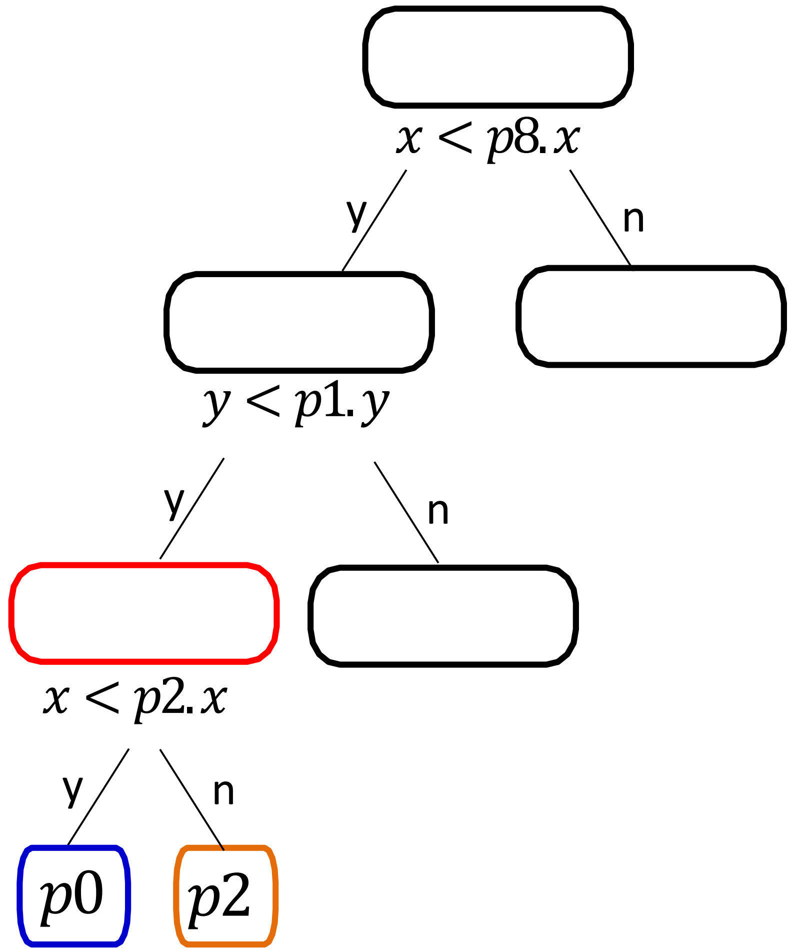




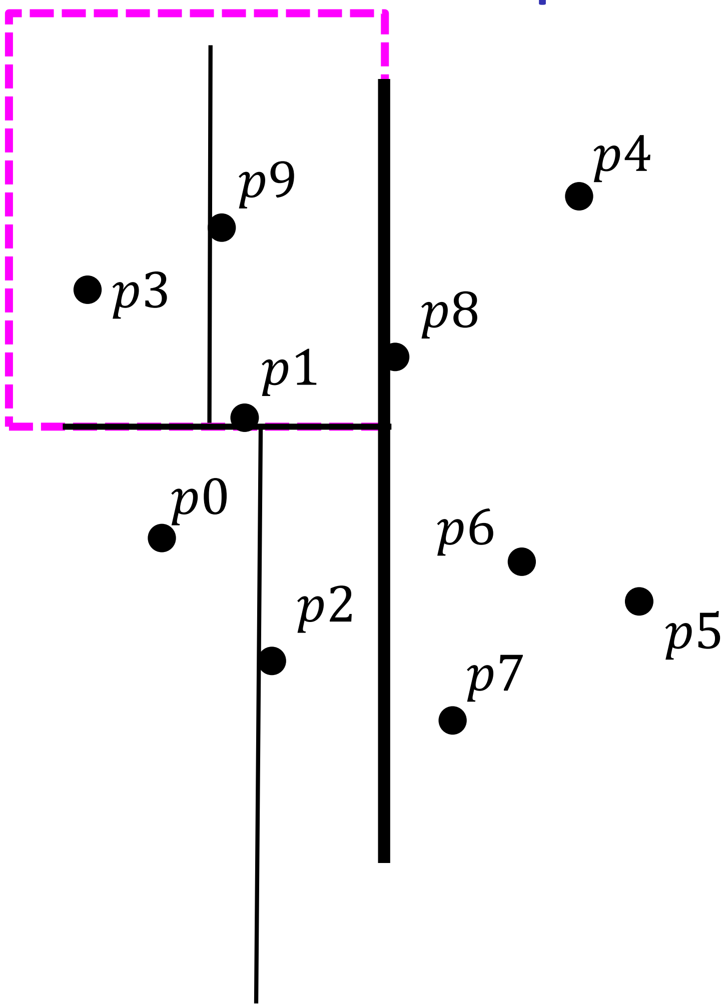
kd-tree example

- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction

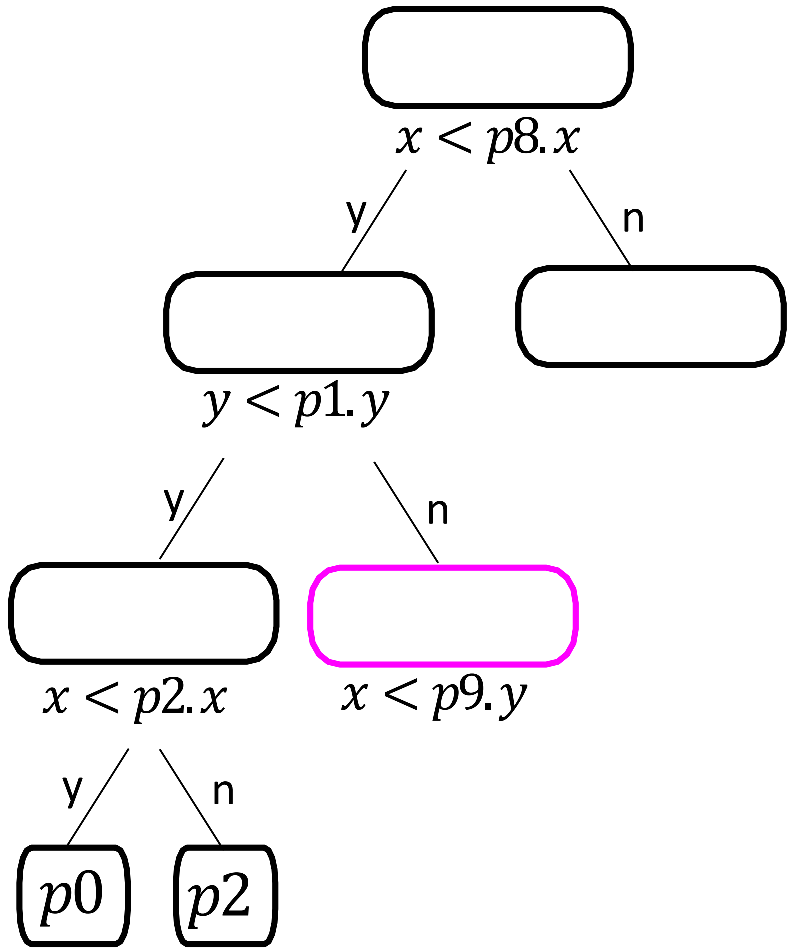




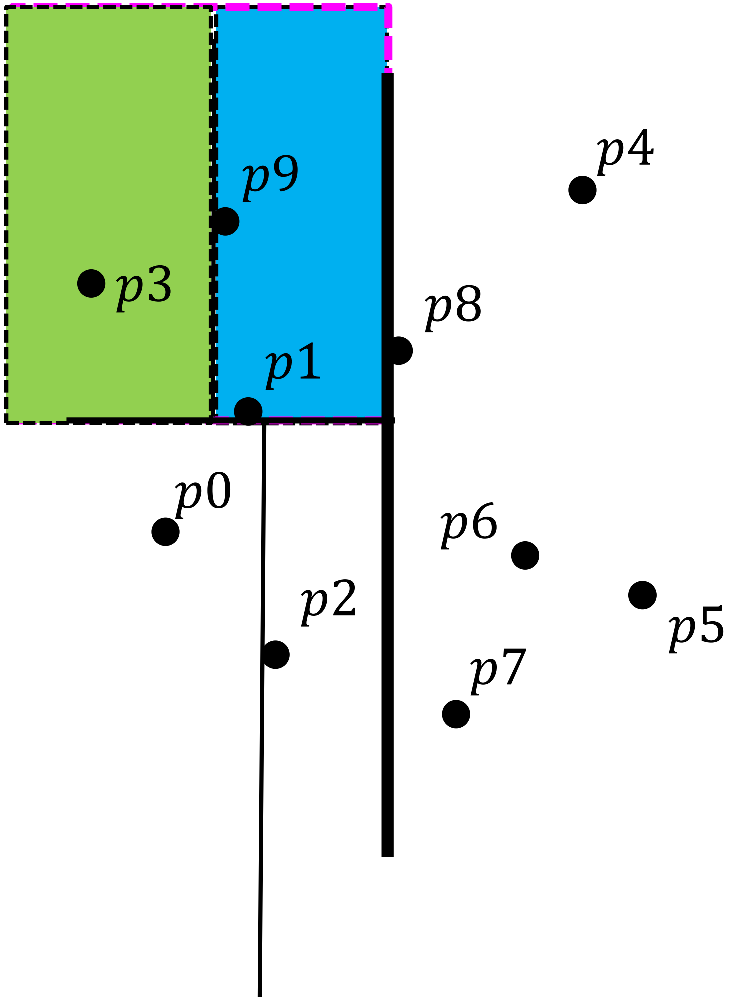
kd-tree example



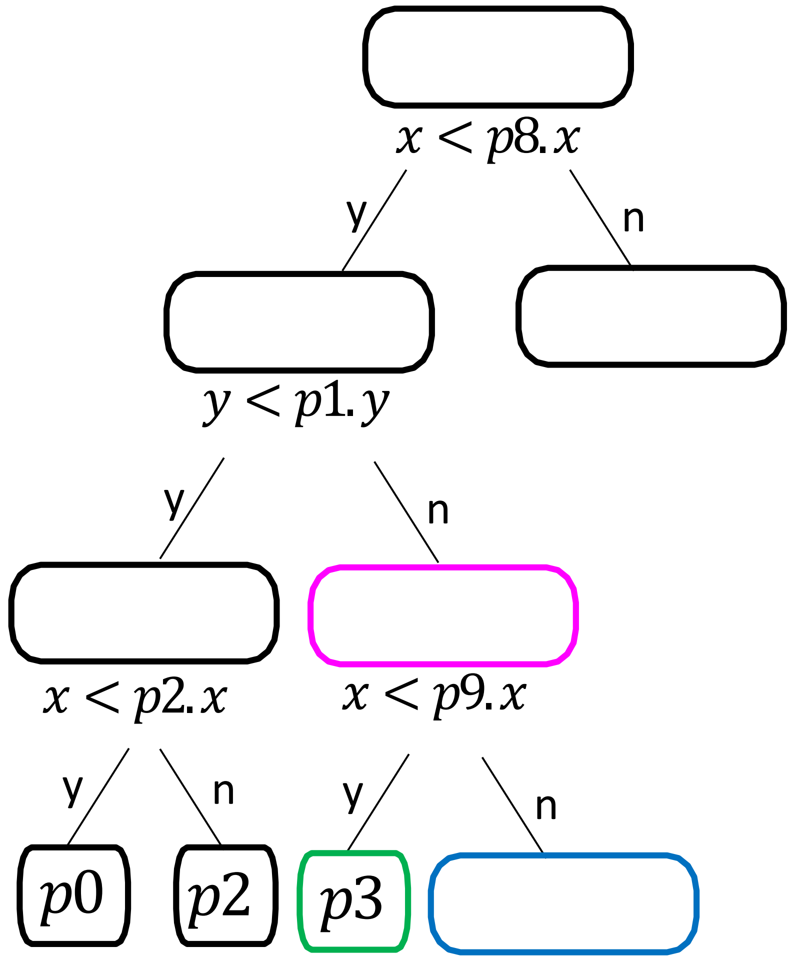
- Recurse on the resulting regions
    - if they have more than one point
- Alternate split direction



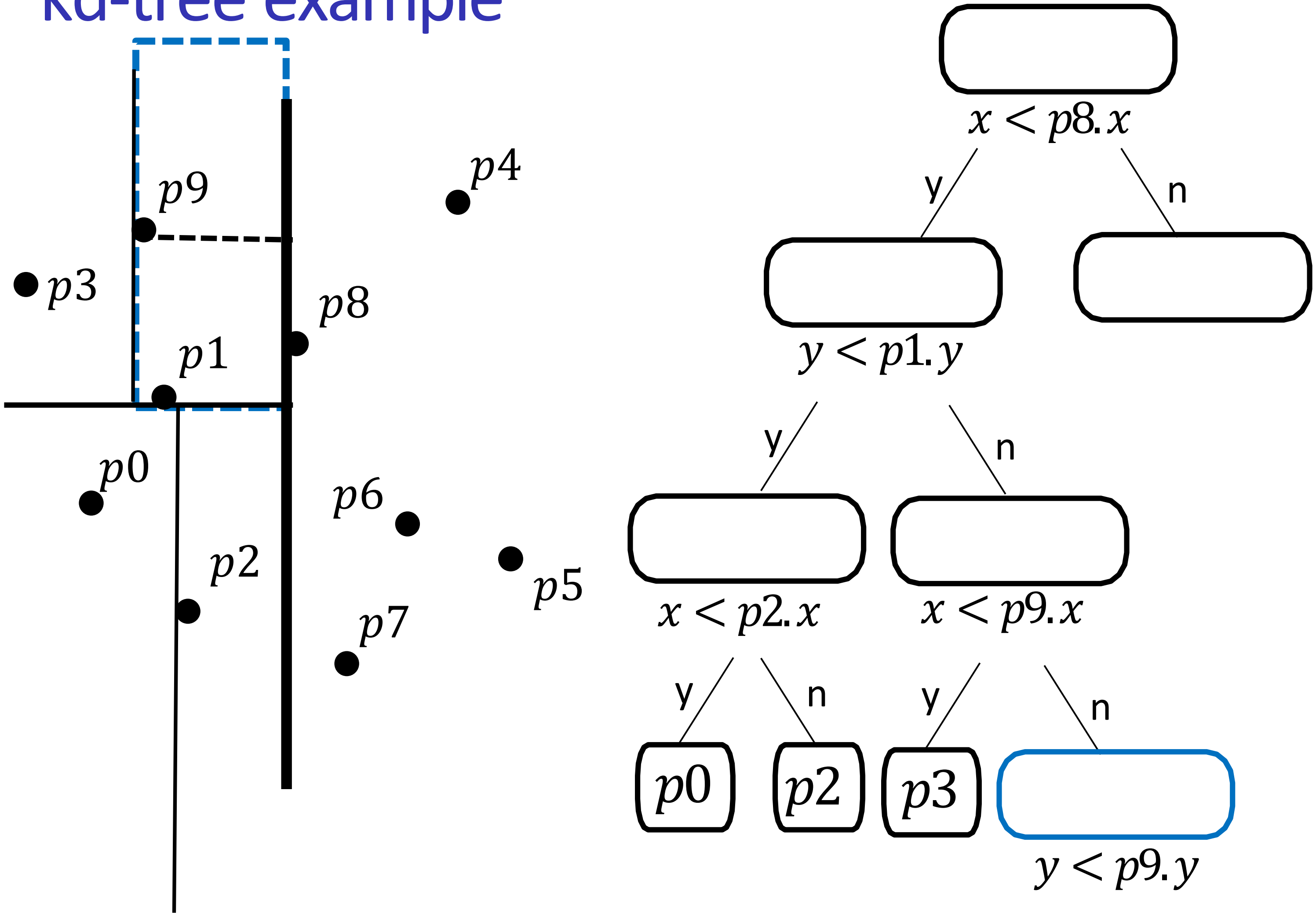
kd-tree example



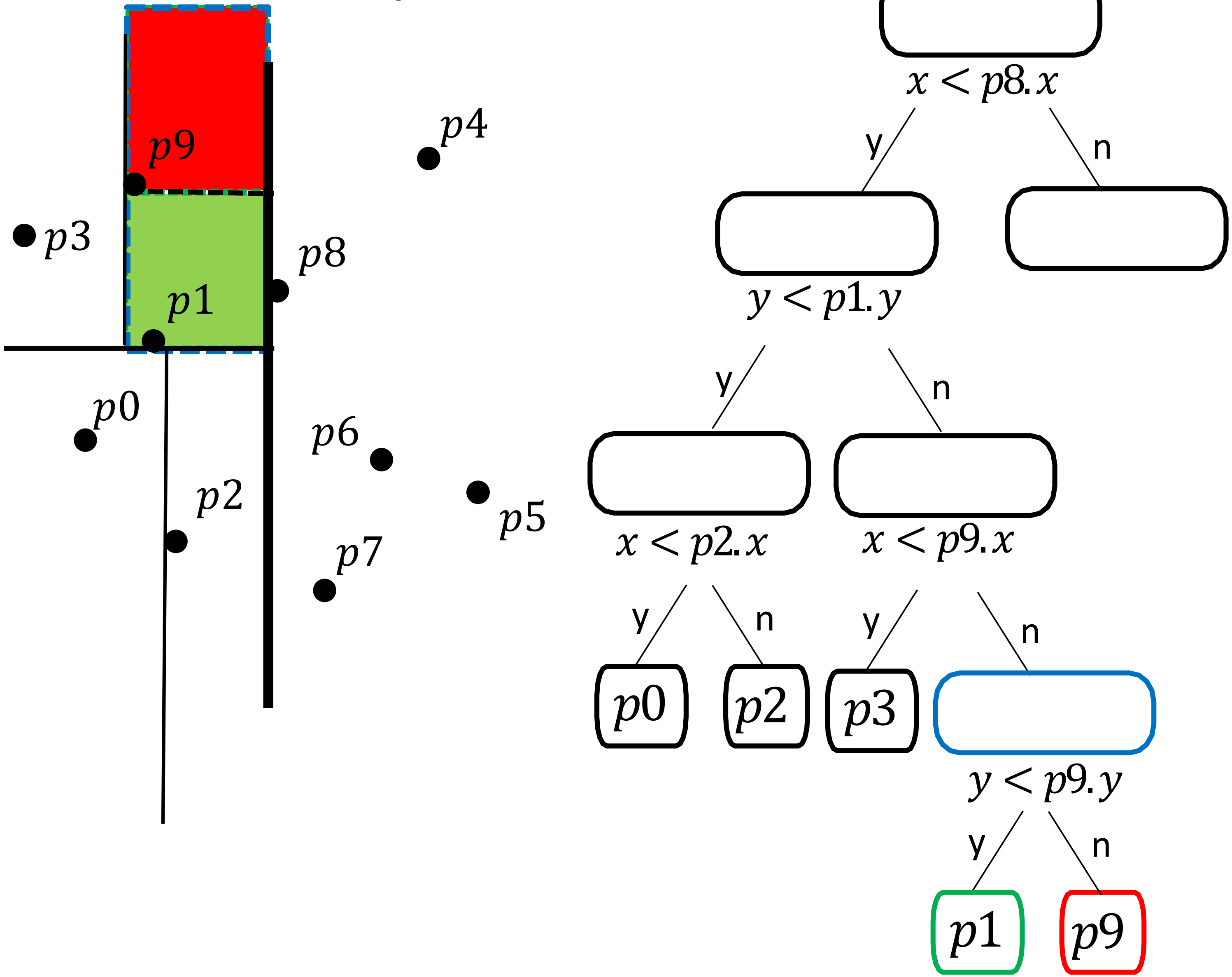
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



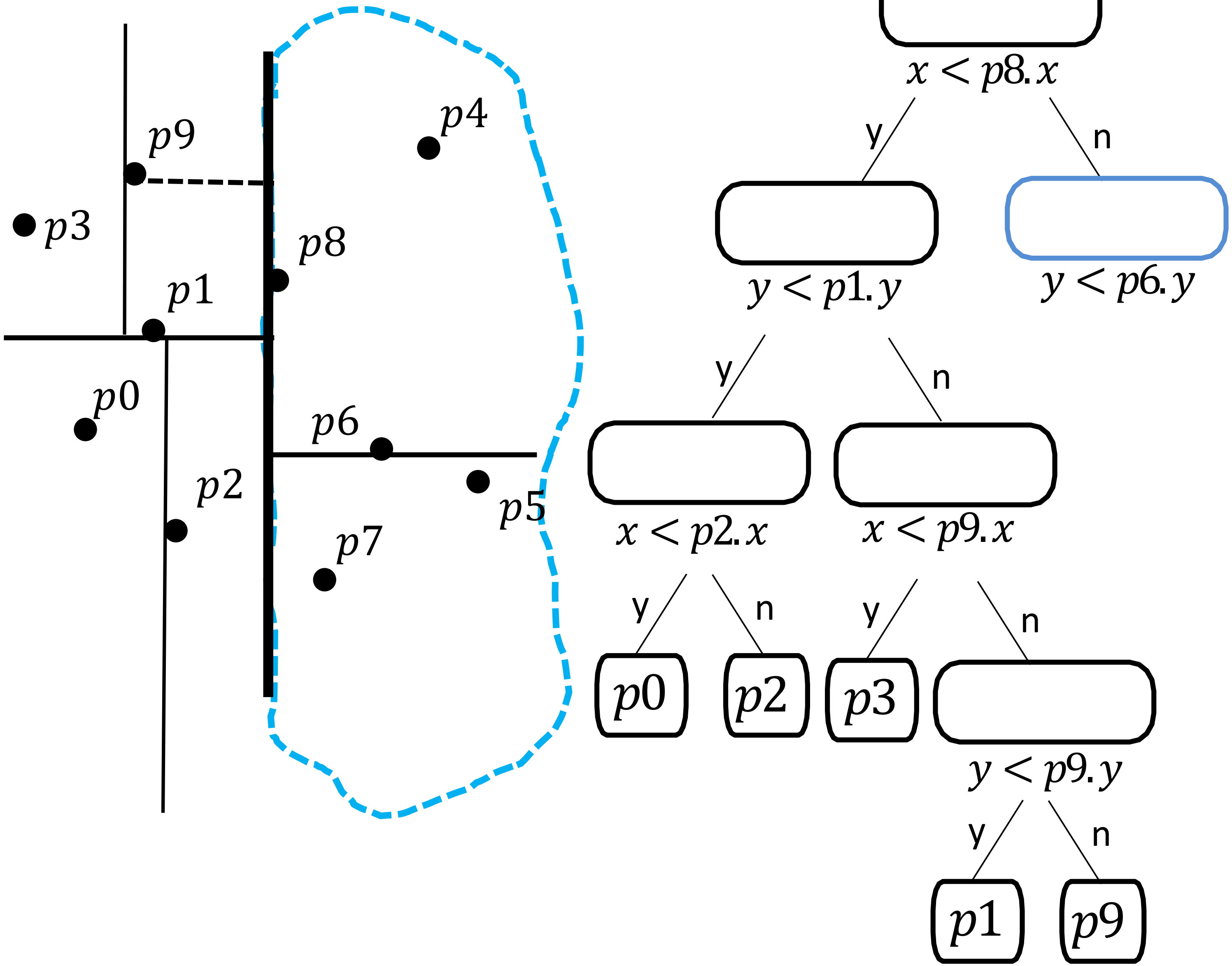
kd-tree example



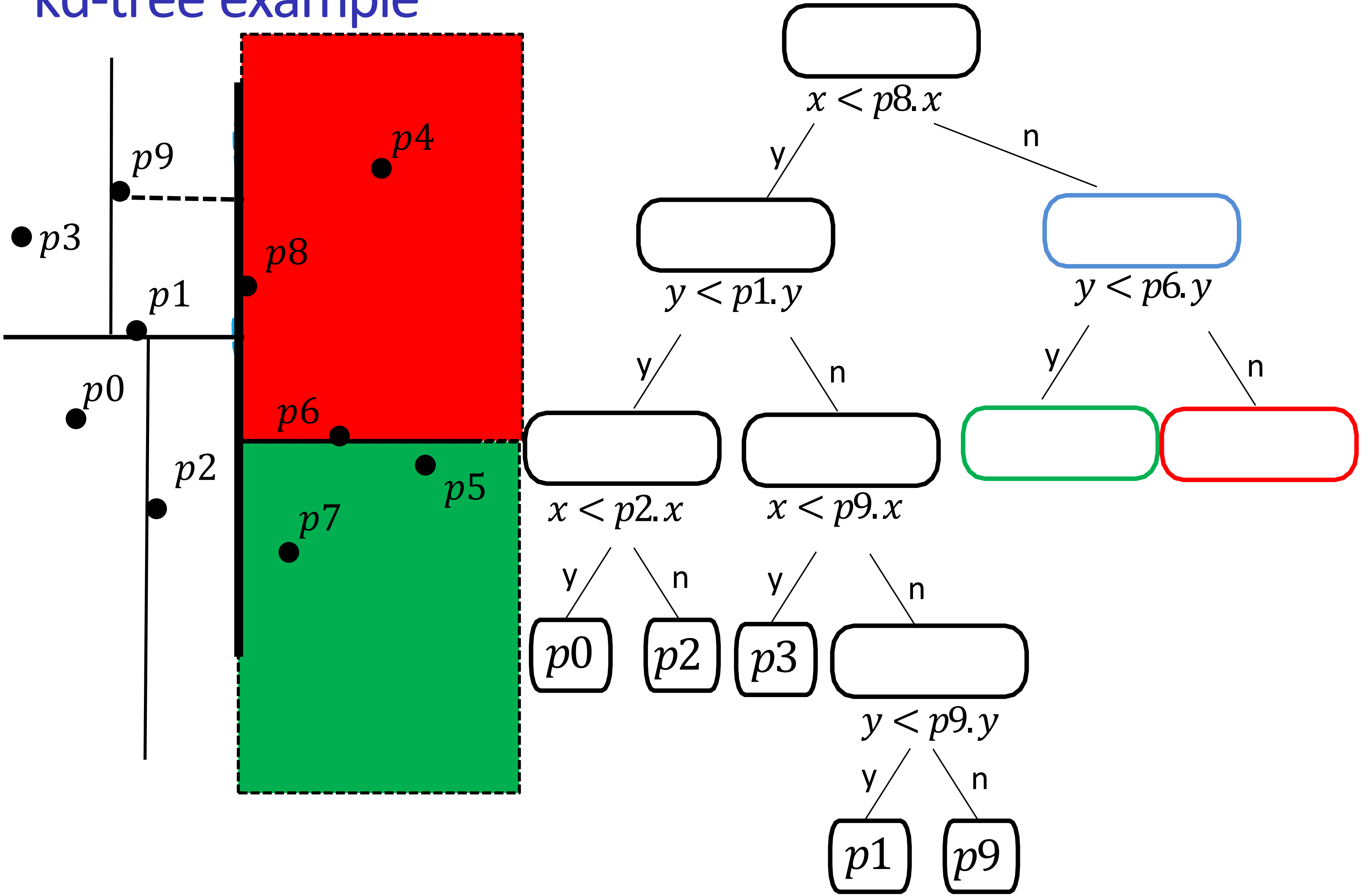
kd-tree example



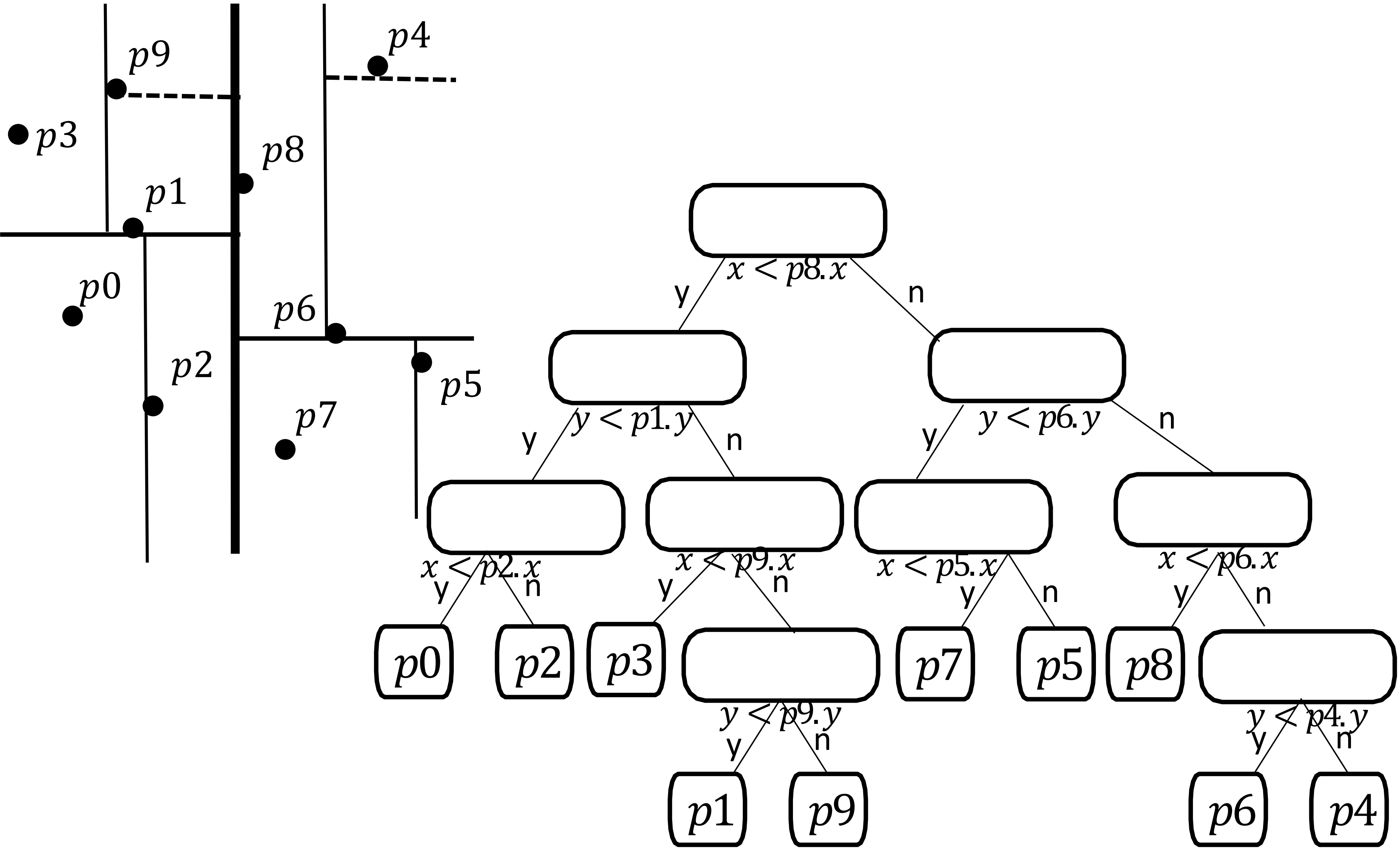
kd-tree example



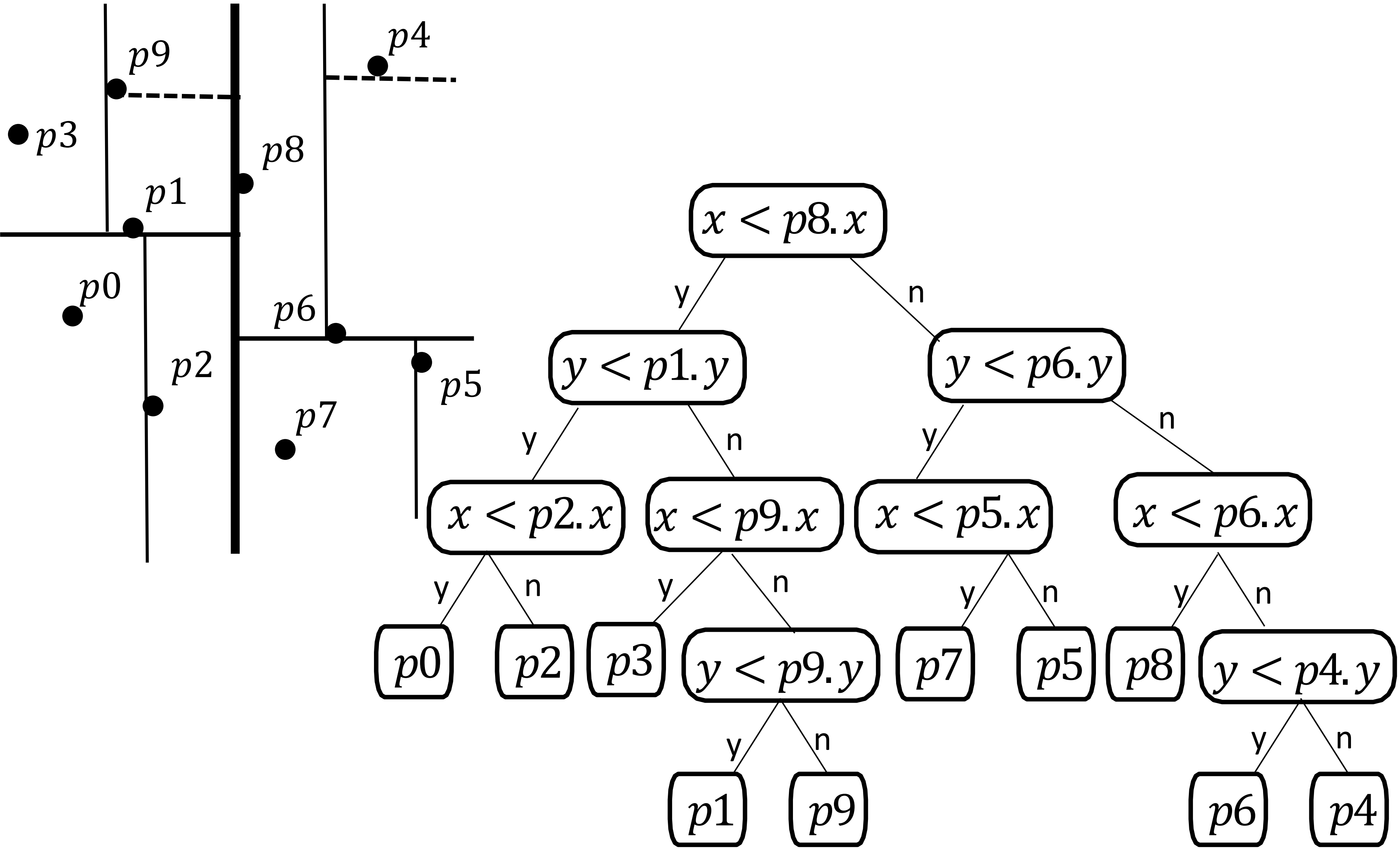
kd-tree example



kd-tree example



kd-tree example



Building kd-trees

- Points $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$
- To build kd-tree with initial x-split
    - if $|S| \le 1$ create a leaf and return
    - else find x-coordinate in position $m = \left\lfloor \frac{n}{2} \right\rfloor$ in sorted list of x -coordinates

      or partition by calling *quickSelect*$(S, \left\lfloor \frac{n}{2} \right\rfloor)$
        - partition S into $S_{x<m}$ and $S_{x \ge m}$ by comparing the x coordinate of a point with m
            - $\left\lfloor \frac{n}{2} \right\rfloor$ goes to one side and $\left\lceil \frac{n}{2} \right\rceil$ to the other
        - create left subtree recursively (splitting on y) for points $S_{x<m}$
        - create right subtree recursively (splitting on y) for points $S_{x \ge m}$
        - each node keeps track of the splitting line
- Building with initial y-split symmetric
- Points on split lines belong to right/top side

kd-tree Construction Running Time and Space

- Partition S in $\Theta(n)$ expected time with *QuickSelect*
- Both subtrees have $\approx n/2$ points
- Sloppy recurrence
    - $T^{exp}(n) = 2T^{exp}\left(\frac{n}{2}\right) + O(n)$
    - resolves to $\Theta(n \log n)$ expected time
- Can improve to $\Theta(n \log n)$ worst-case runtime by pre-sorting coordinates
- Recurrence inequality for height

$$h(1) = 0$$
$$h(n) \leq h\left(\left\lceil\frac{n}{2}\right\rceil\right) + 1$$

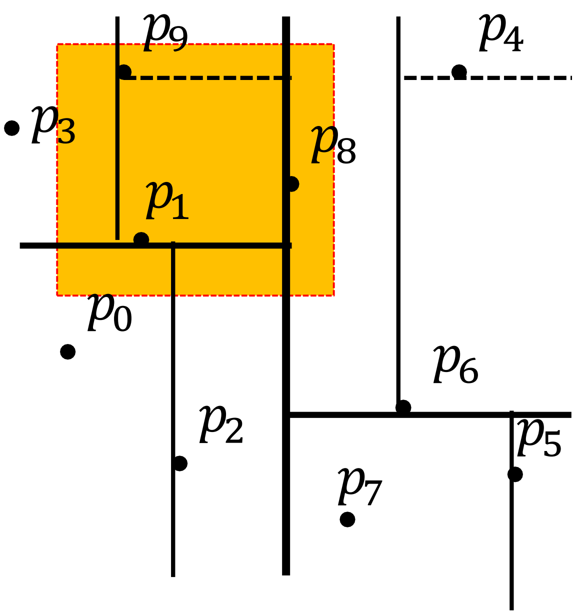
- resolves to $O(\log n)$, specifically $\lceil \log n \rceil$
- this is tight (binary tree with n leaves)

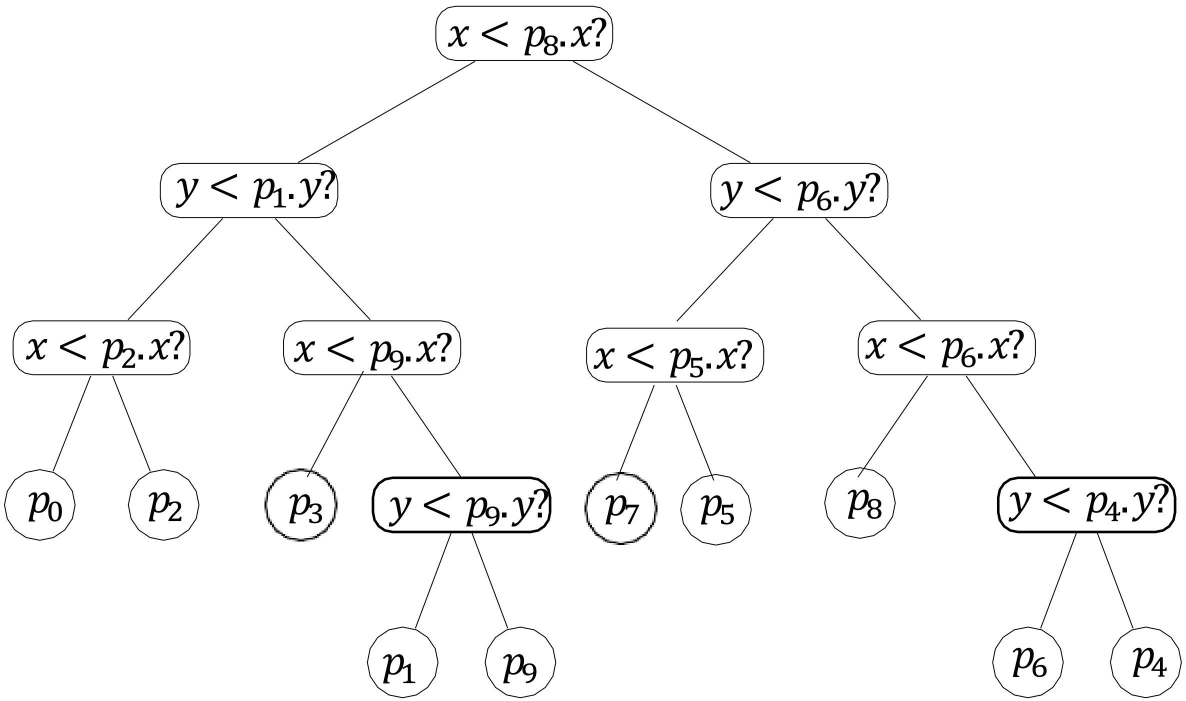
- Space
    - all interior nodes have exactly 2 children, therefore $n-1$ interior nodes
    - total number of nodes is $2n-1$
    - space is $\Theta(n)$

kd-tree Dictionary Operations

- *search* as in binary search tree using indicated coordinate
- *insert* first search, insert as new leaf
- *delete* first search, remove leaf and any parent with one child
- **Problem**
  - after insert or delete, split might no longer be at exact median
  - height is no longer guaranteed to be $O(\log n)$
  - kd-tree do not handle insertion/delection well
  - remedy
    - allow a certain imbalance
    - re-building the entire tree when it becomes too unbalanced
    - no details
    - but *rangeSearch* will be slower

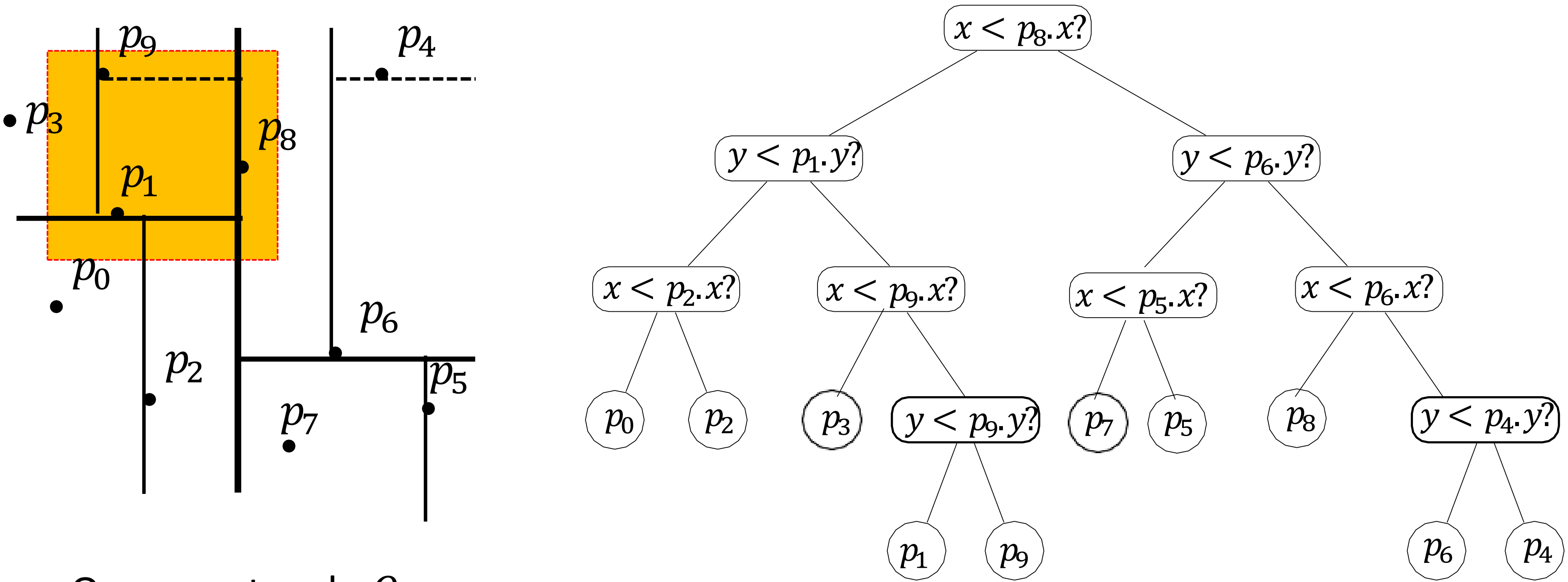
kd-tree: Range Search Example





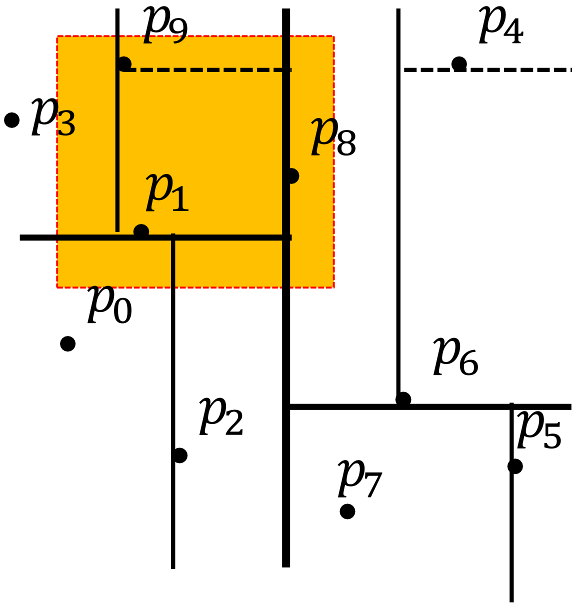
- Every node is associated with a region
  - range search is exactly as for quadtrees, except there are only two children and leaves always store points

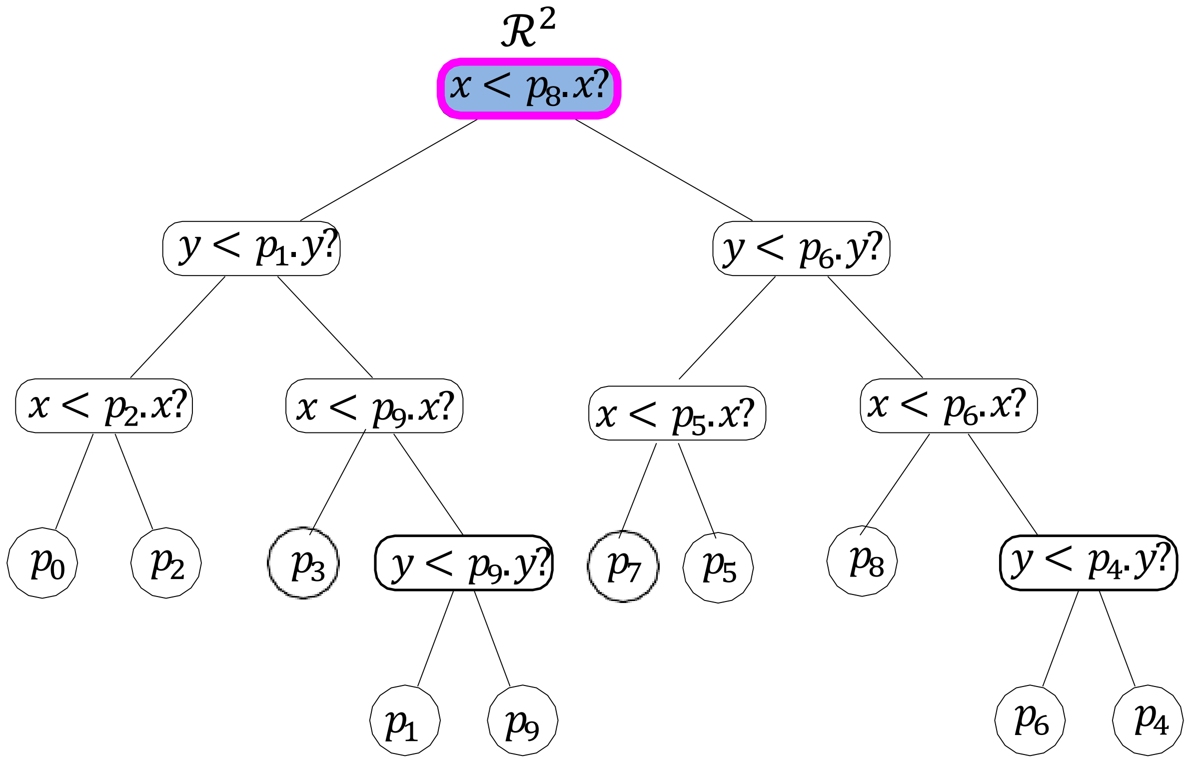
kd-tree: Range Search Example



- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

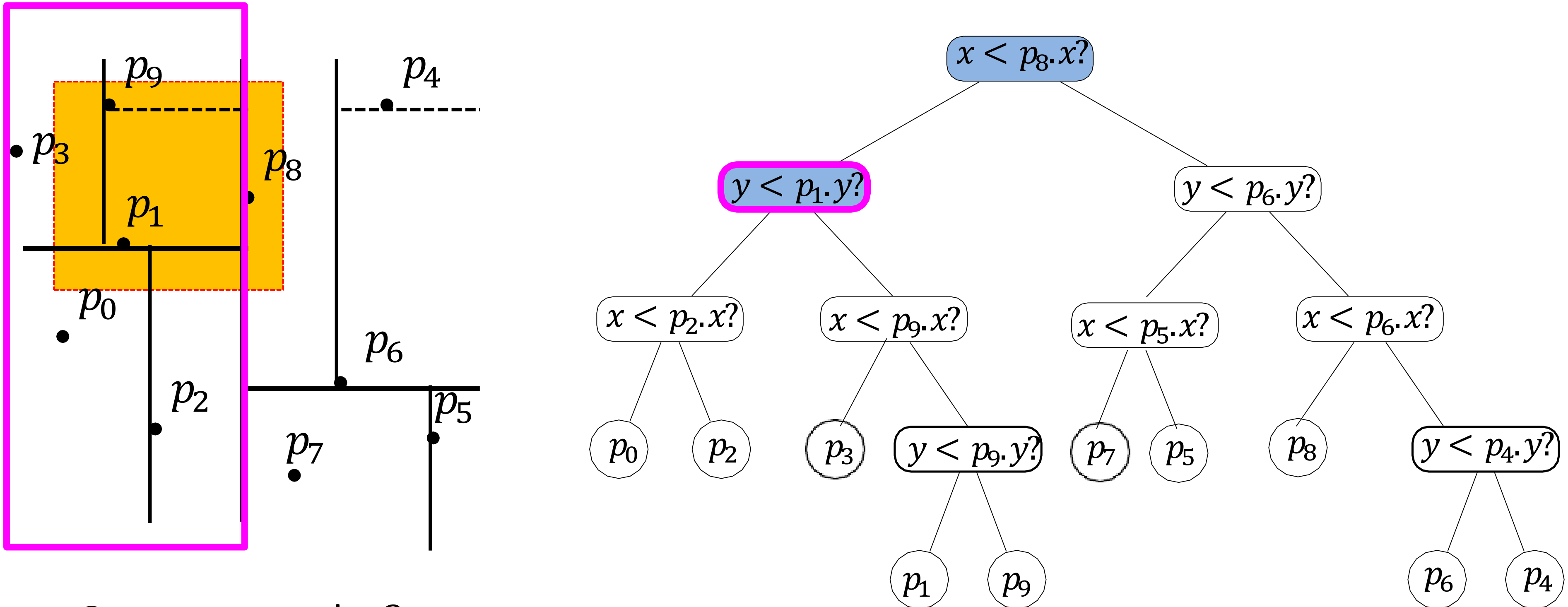
kd-tree: Range Search Example





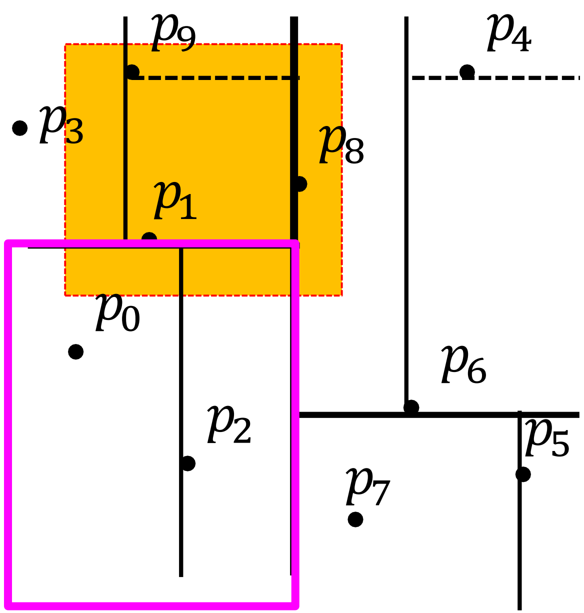
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

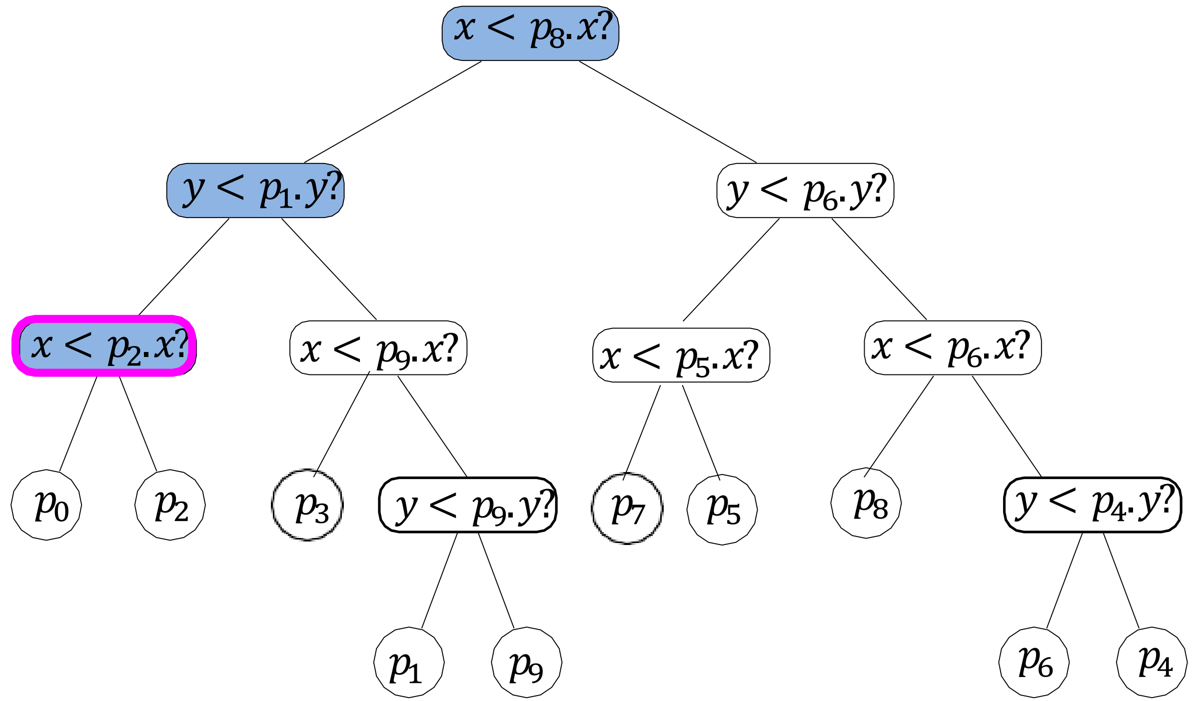
kd-tree: Range Search Example



- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

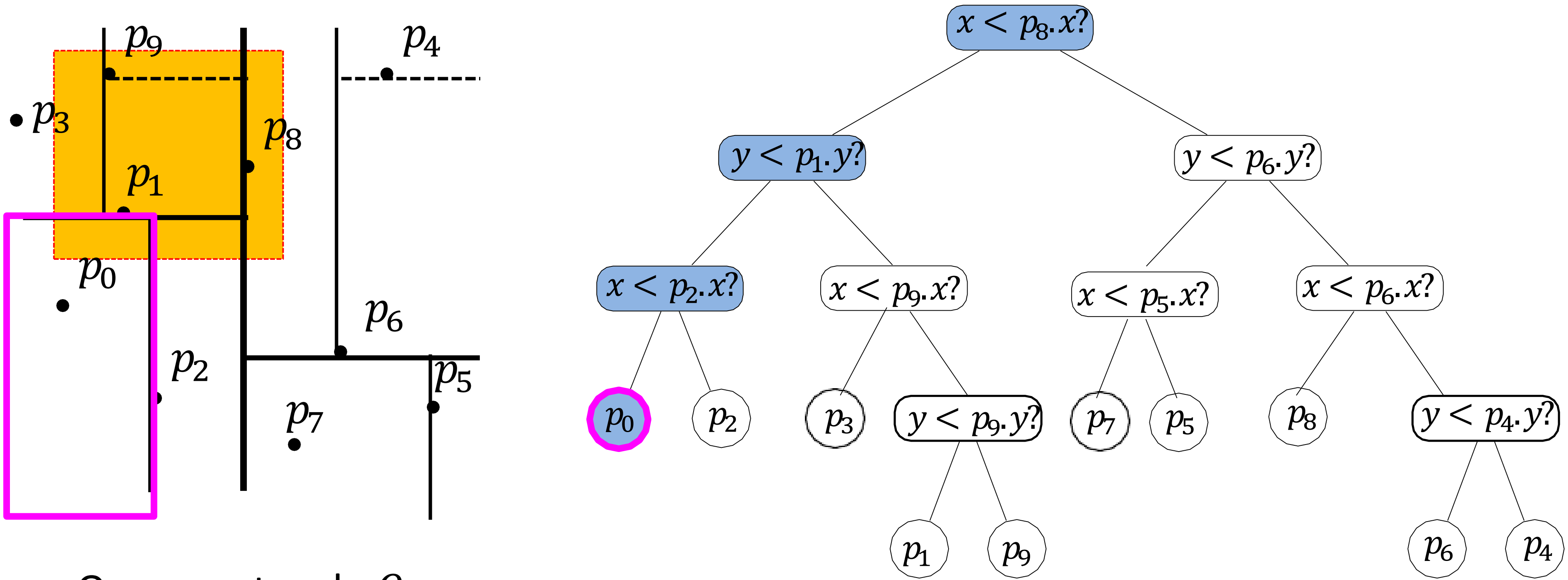
kd-tree: Range Search Example





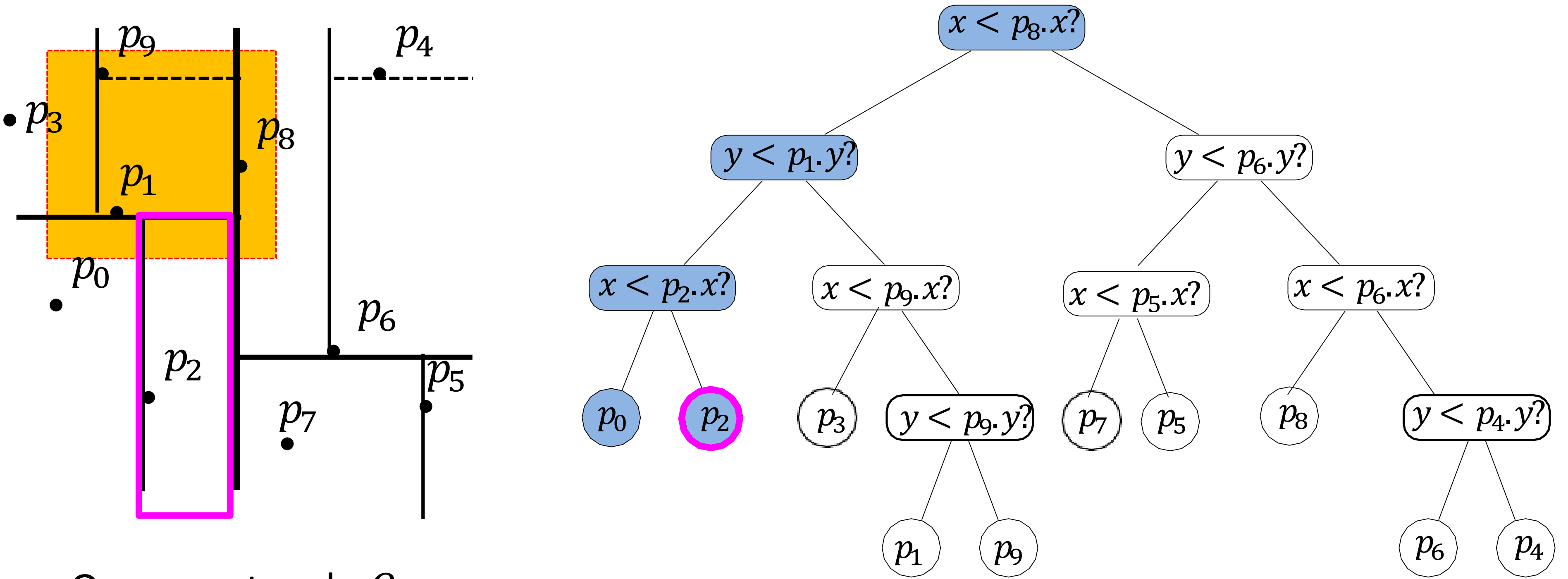
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



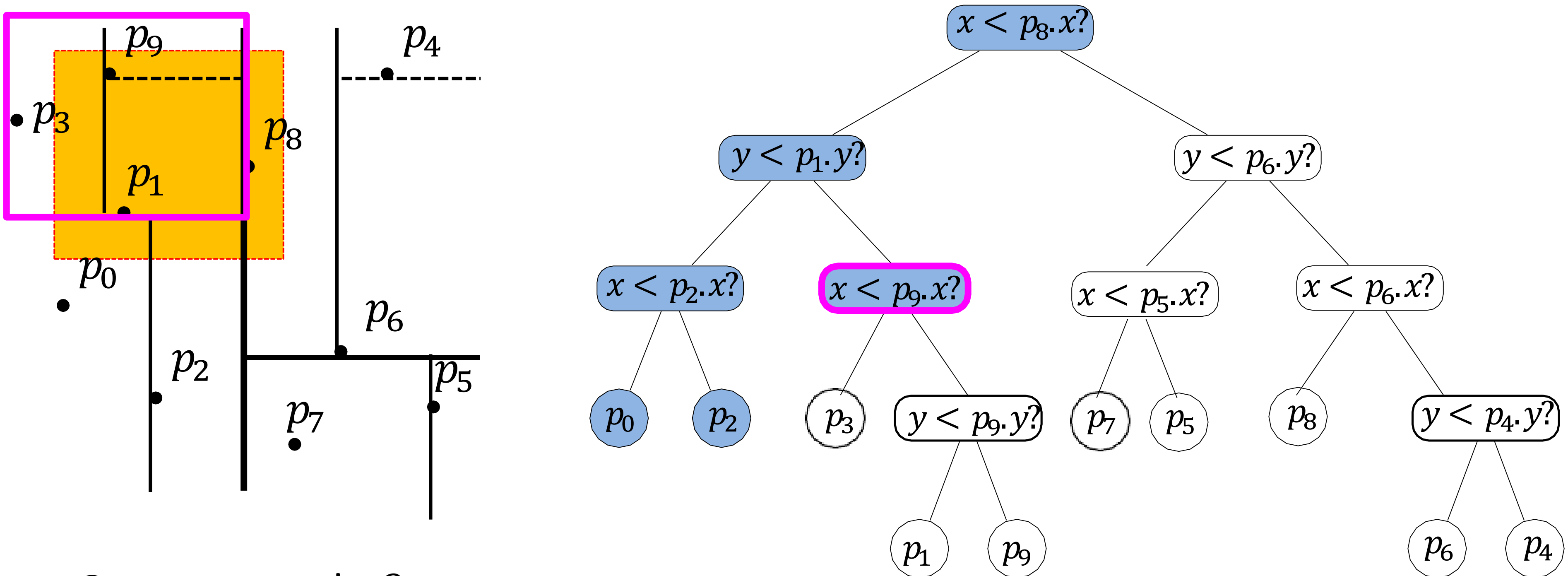
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



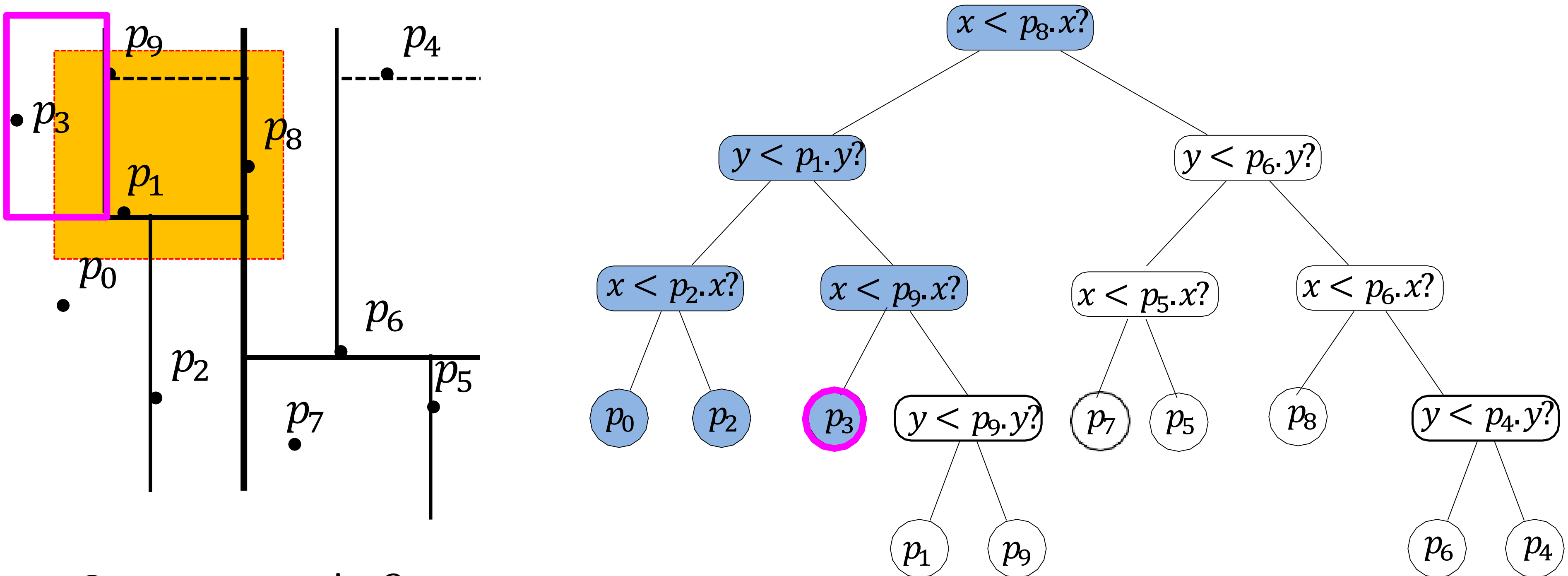
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



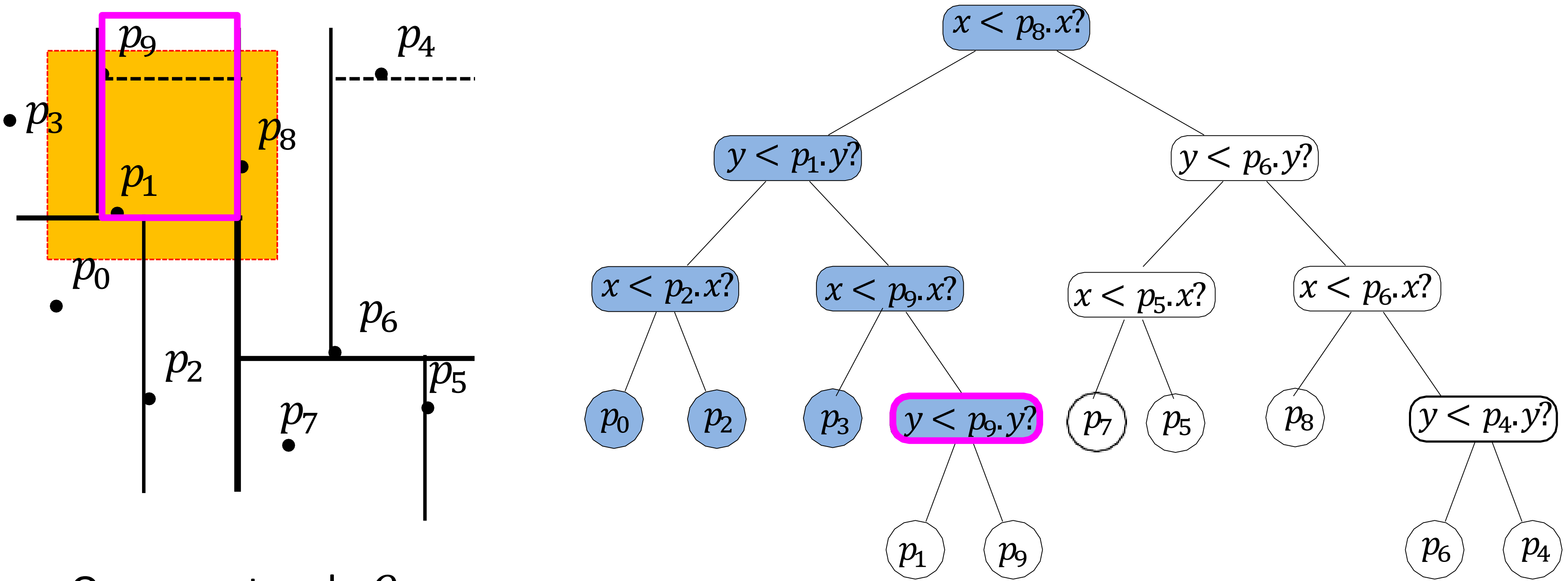
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



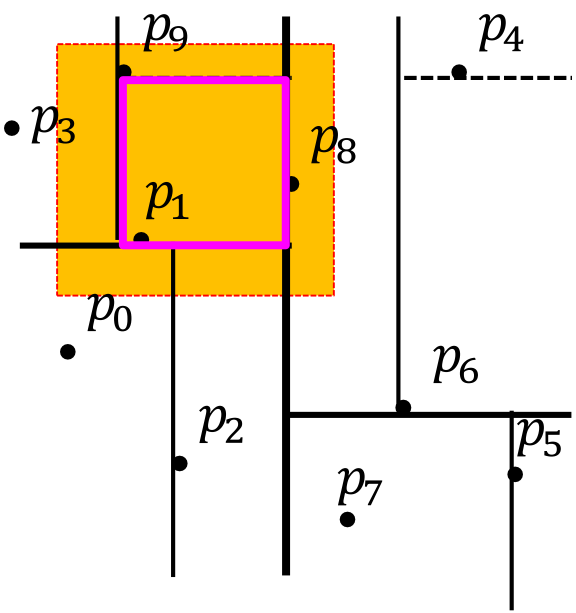
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

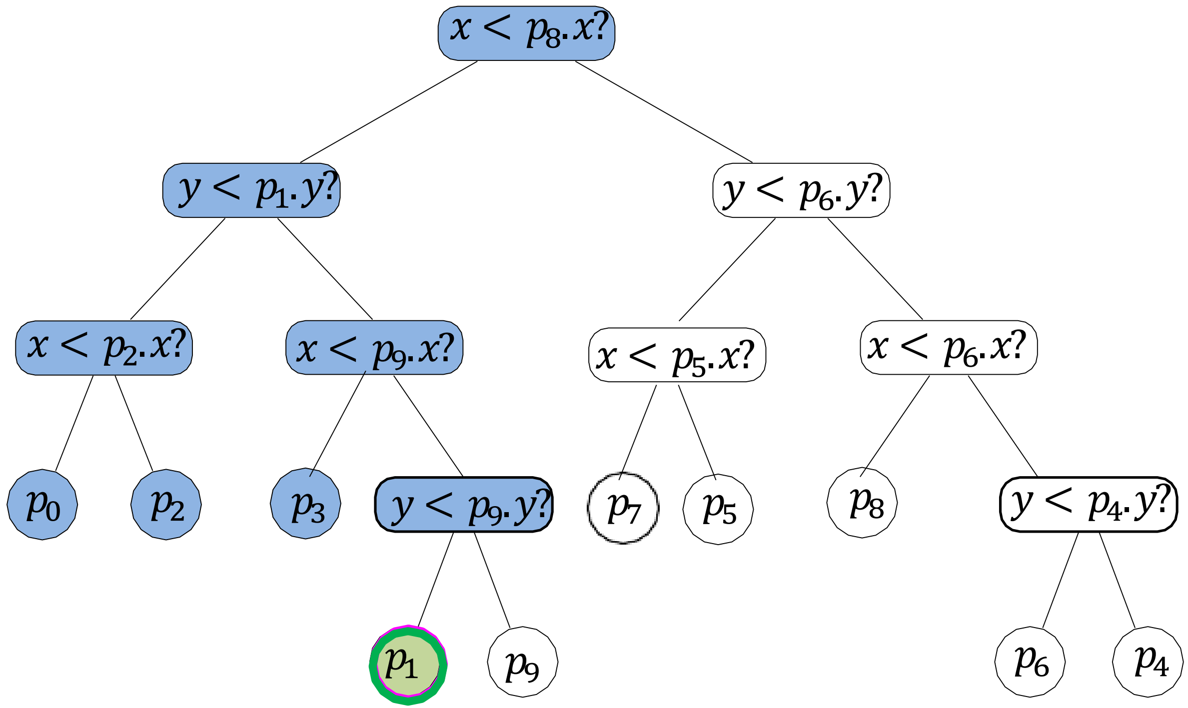
kd-tree: Range Search Example



- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

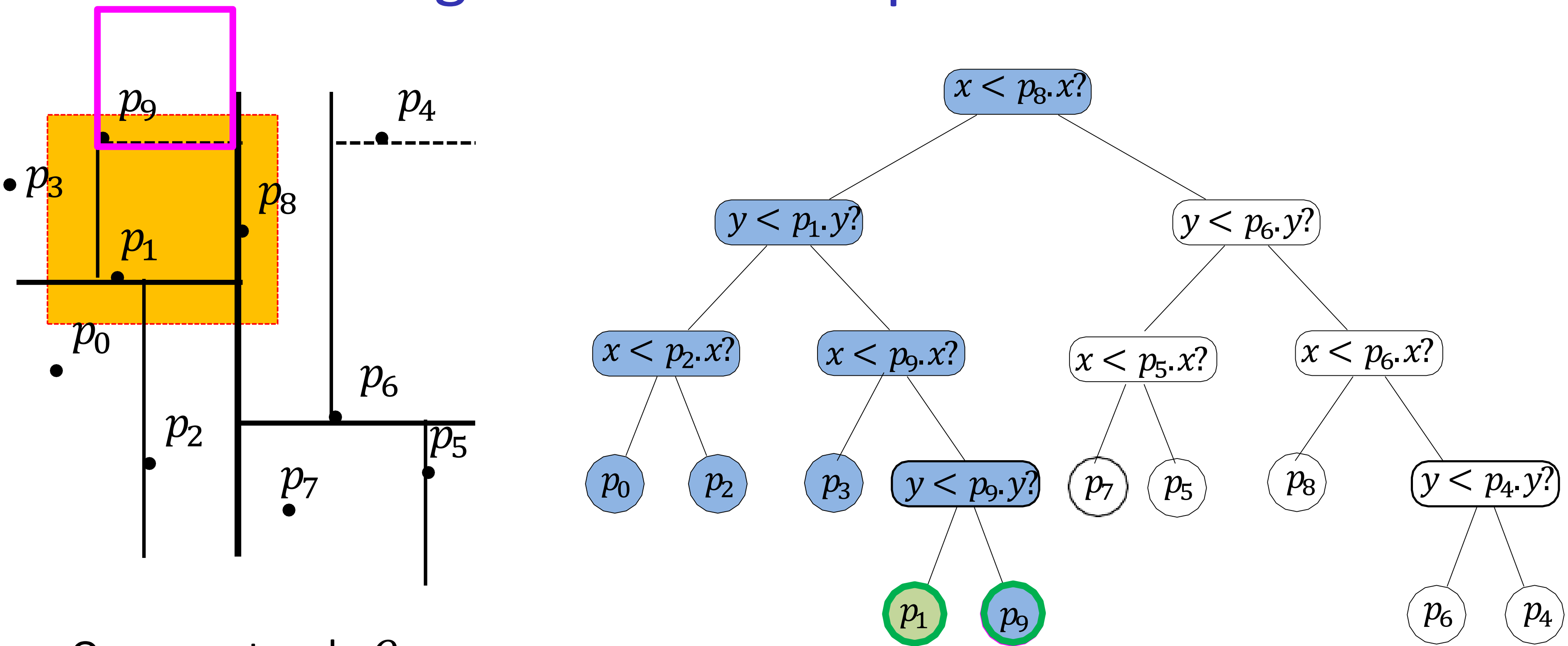
kd-tree: Range Search Example





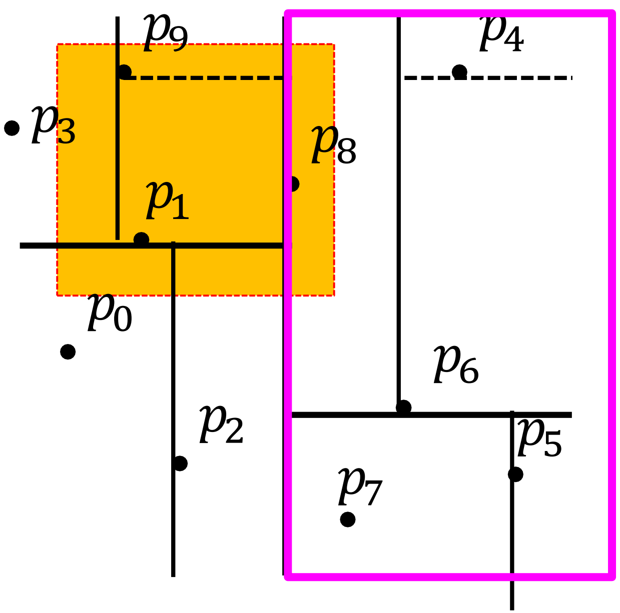
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

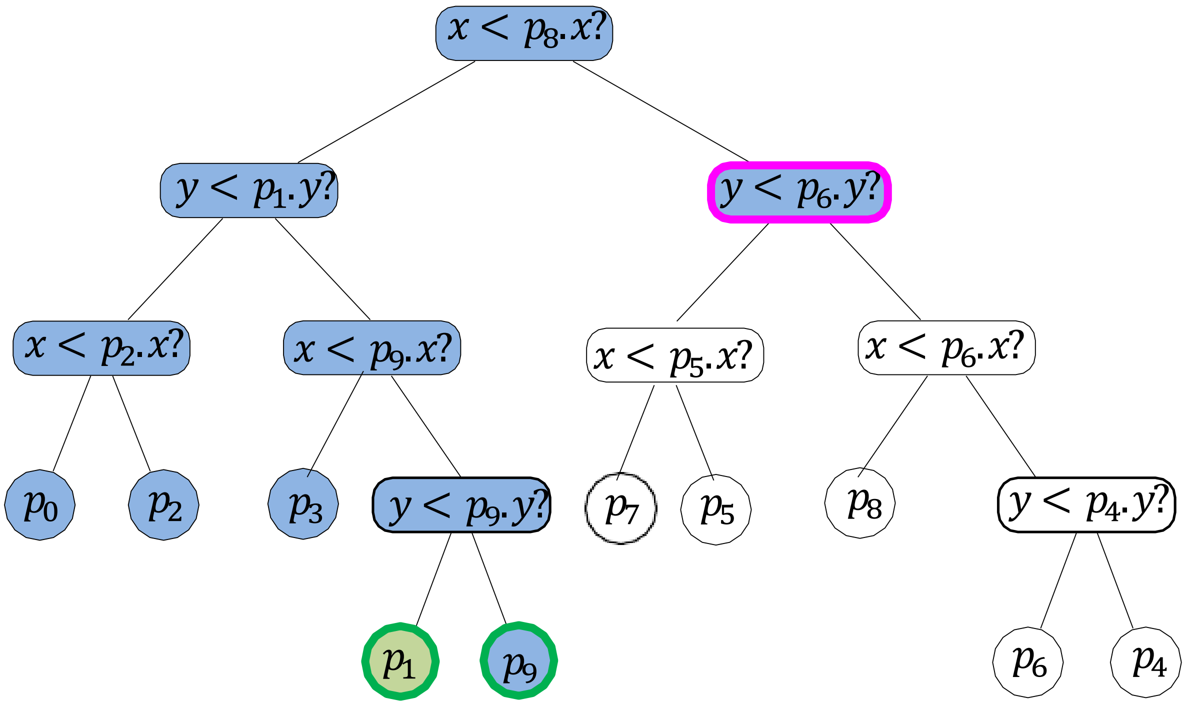
kd-tree: Range Search Example



- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

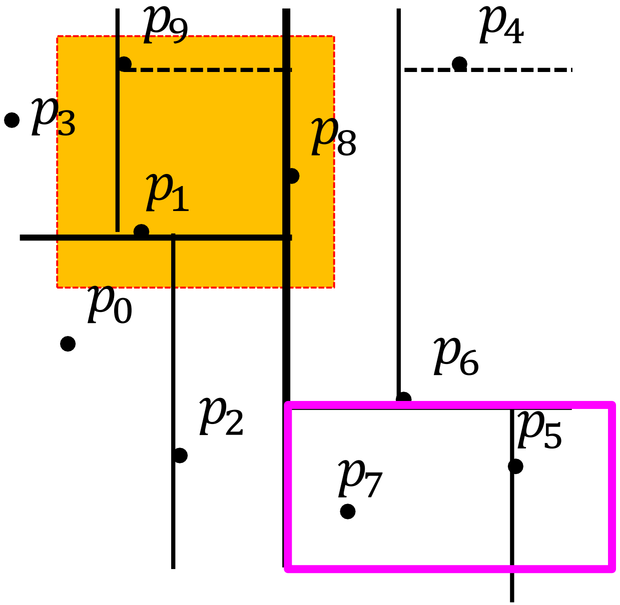
kd-tree: Range Search Example

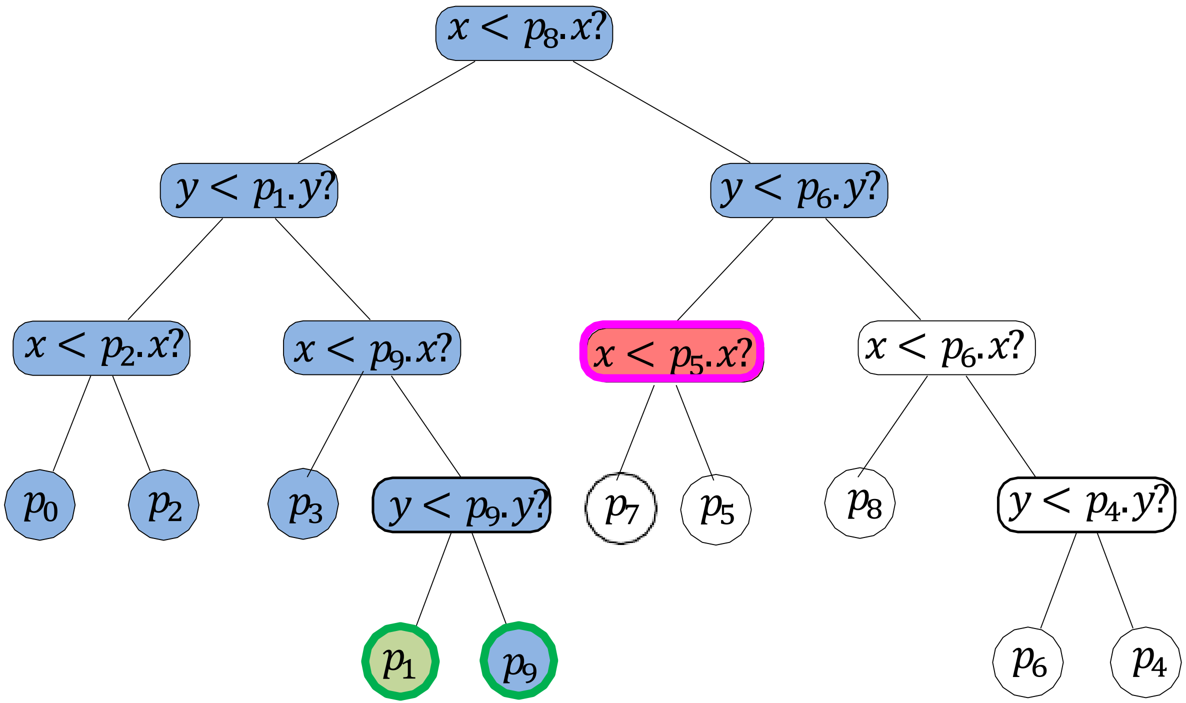




- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

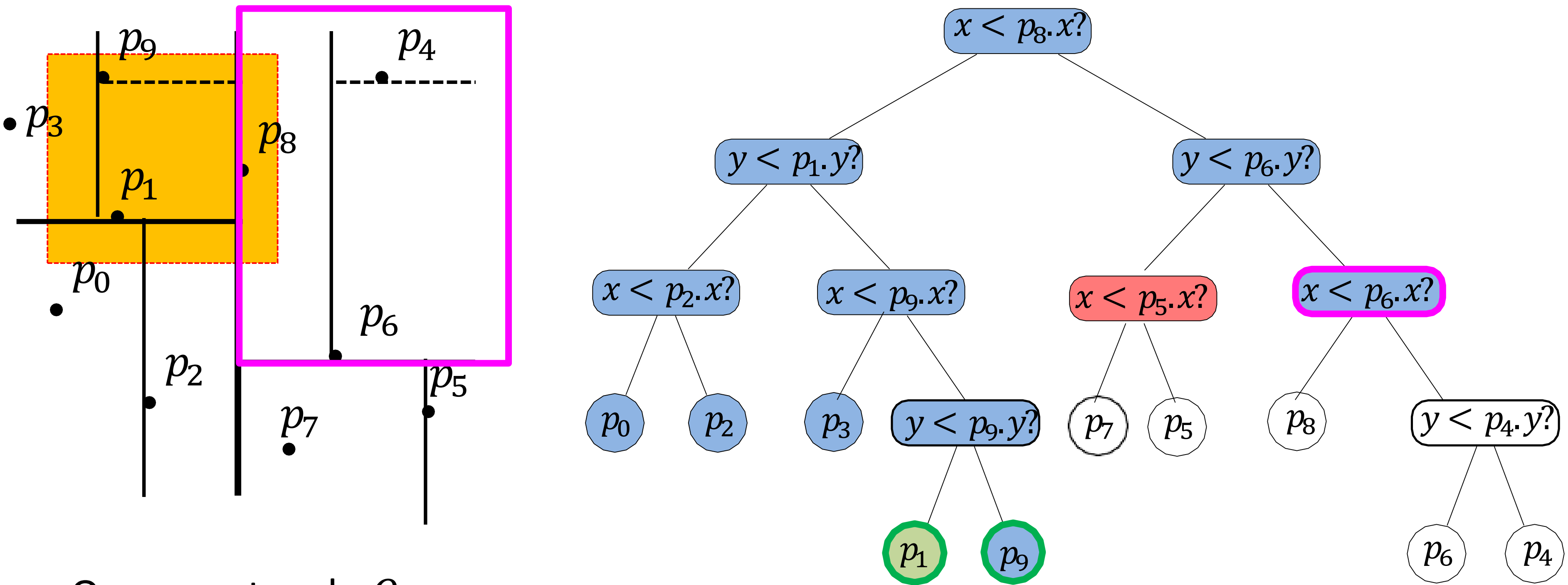
kd-tree: Range Search Example





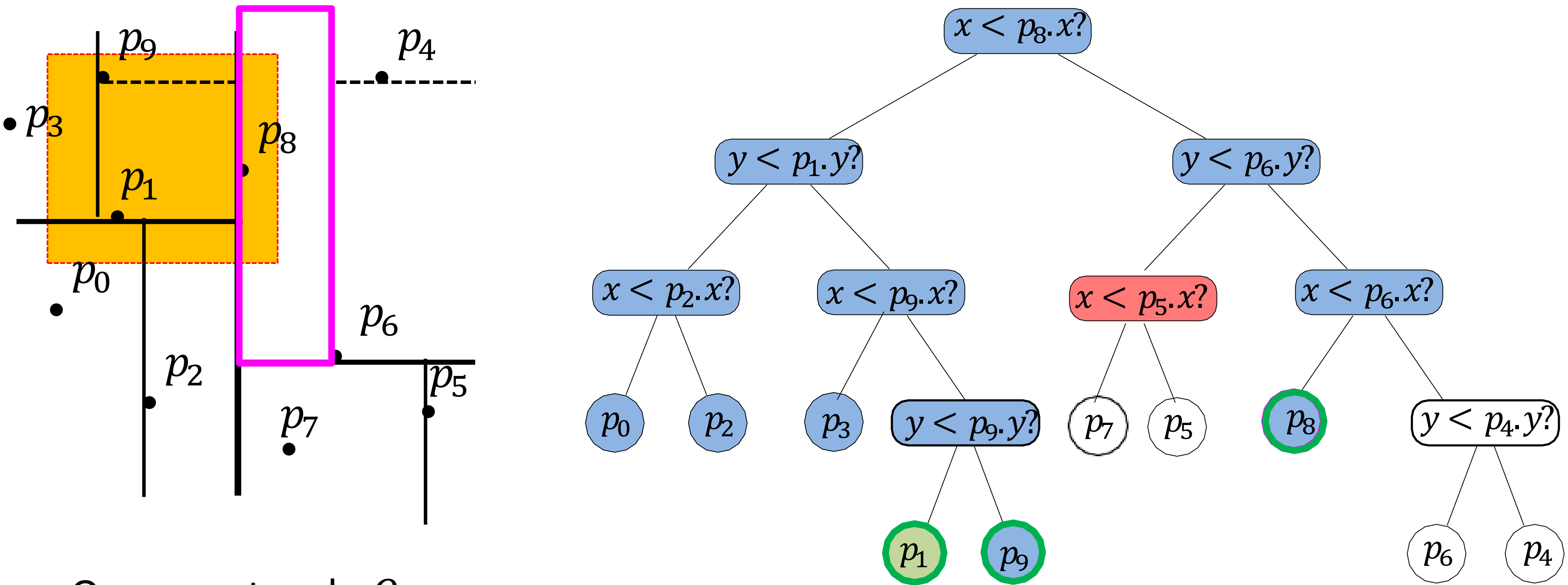
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



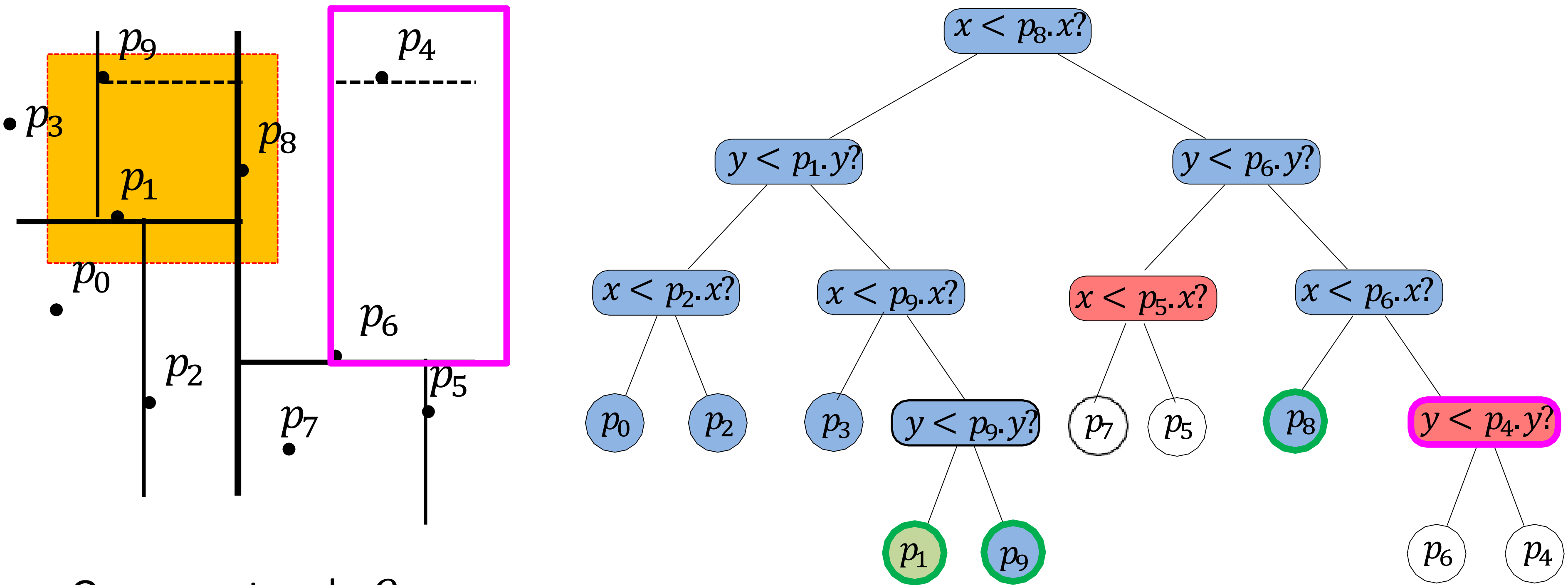
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
  1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
  3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
     - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



- Query rectangle Q
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

kd-tree: Range Search Example



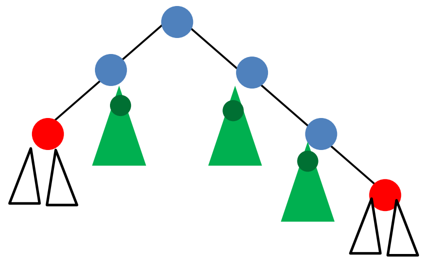
- Query rectangle Q
- Let R be region associated with current node, have 3 cases
    1. $R \cap Q = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq Q$: green (inside) node, no need to search children, report all points in R
    3. $R \cap Q \neq \emptyset$: blue (boundary) node, search its children (if any)
        - if R is a leaf, if it stores point inside Q, report it

kd-tree Range Search

```
kdTree::RangeSearch(r ← root, Q)
r : root of kd-tree, Q: query rectangle
        R ← region associated with node r
        if R ⊆ Q then
                report all points below r
                return
        if R ∩ Q = ∅ then return
        if r is a leaf then
              p ← point stored at r
              if p ∈ Q return p
              else return
        for each child v of r do
              kdTree::RangeSearch(v, Q)
```

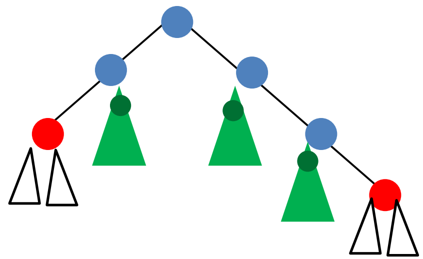
- We assume that each node stores its associated region
- To save space, we could instead pass the region as a parameter and compute the region for each child using the splitting line

kd-tree: Range Search Running Time



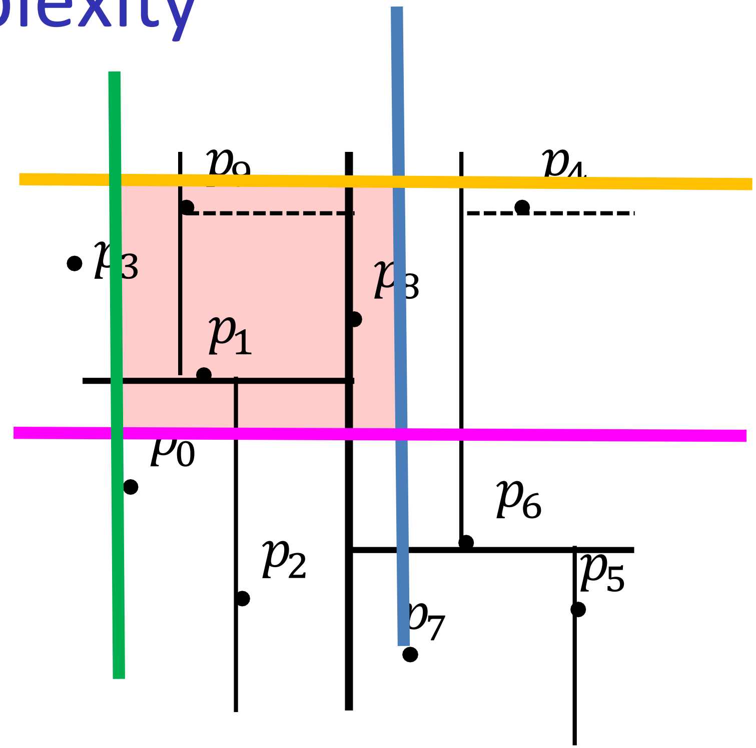
- Visit blue, red, and green nodes, constant work at each node
  - runtime is proportional to the number of blue, red, green nodes
- Green nodes form green subtrees
  - subtree root is the *topmost* green node
  - let v be the topmost green node
    - recall that s is the number of nodes in the output of range search
    - subtree of v is a kd-tree itself
      - number of internal nodes is 1 less than the number of leaves
    - at most s leaves over all green subtrees, and, therefore, at most $2s$ nodes over all green subtrees
  - number of green nodes is $O(s)$

kd-tree: Range Search Running Time



- Visit blue, red, and green nodes, constant time at each node
  - $O(s)$ of green nodes
- red nodes $\leq 2 \cdot$ blue nodes
  - each red node has a blue parent
  - for asymptotic runtime, enough to count blue nodes and add $O(s)$
- Let $B(n)$ is the number of blue nodes
  - if R corresponds to a blue node, neither $R \cap Q = \emptyset$ nor $R \subseteq Q$
  - regions that intersect Q but not completely inside Q
- Can show that $B(n)$ satisfies $B(n) \leq 2B\left(\frac{n}{4}\right) + O(1)$
  - resolves to $B(n) \in O(\sqrt{n})$
- Therefore, running time of range search is $O(s + \sqrt{n})$

kd-tree: Range Search Complexity

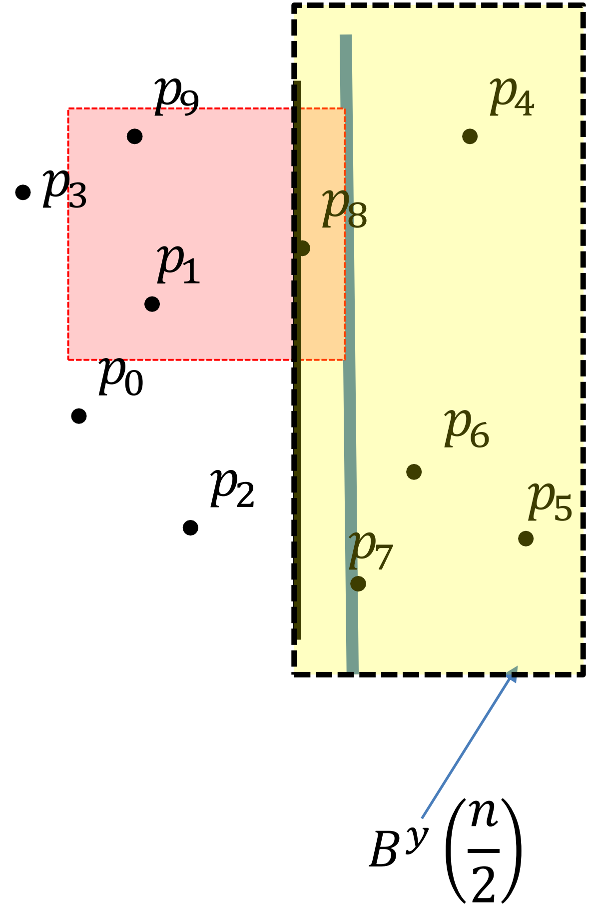


- search rectangle Q
- $B(n) = \#$ regions intersecting Q but not completely inside Q
- $B(n) \le \#$ regions intersecting [blue line]
  $+ \#$ regions intersecting [green line]
  $+ \#$ regions intersecting [magenta line]
  $+ \#$ regions intersecting [orange line]
- Will look at # regions intersecting [blue line]
- Other cases are handled similarly

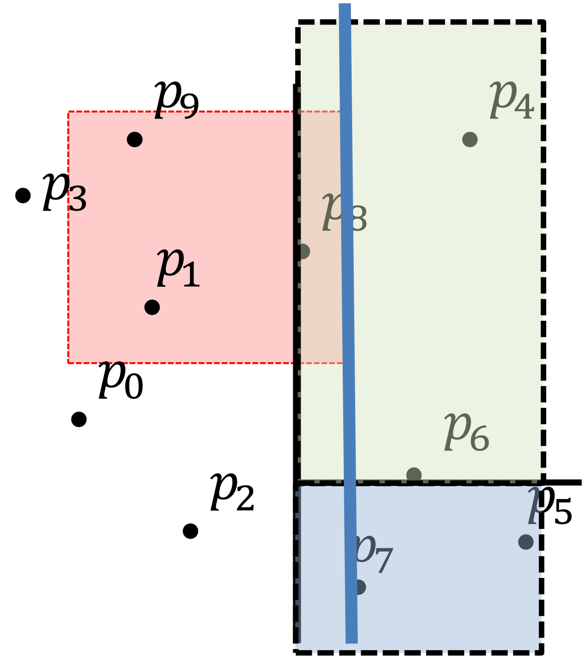
kd-tree: Range Search Complexity

- $B^x(n) =$ # regions intersected by ▌, if tree root split by x coordinate

- $B^x(n) = 1 + B^y\left(\frac{n}{2}\right)$
  - 1 for the root region R
  - root region is split in 2 by vertical line
  - ▌ can intersect only one of these regions



kd-tree: Range Search Complexity



- $B^x(n) =$ # regions intersected by ▌, if tree root split by x coordinate
- $B^x(n) = 1 + B^y\left(\frac{n}{2}\right)$
  - 1 for the root region
  - root region is split in 2 by vertical line
  - ▌ can intersect only one of these regions
- Next, $B^y\left(\frac{n}{2}\right) = 1 + 2B^x\left(\frac{n}{4}\right)$
  - 1 for the root region
  - root region is split in 2 by horizontal line
  - ▌ can intersect both of these regions
- Combining, get recurrence $Q^x(n) = 2 + 2B^x\left(\frac{n}{4}\right)$
- Resolves to $B^x(n) \in O(\sqrt{n})$

kd-tree: Higher Dimensions

- kd-trees for d-dimensional space
  - at depth 0 (the root) partition is based on the 1st coordinate
  - at depth 1 partition is based on the 2nd coordinate
  - …
  - at depth $d - 1$ the partition is based on the last coordinate
  - at depth d start all over again, partitioning on 1st coordinate
- Storage $O(n)$
- Height $O(\log n)$
- Construction time $O(n \log n)$
- Range query time $O(s + n^{1-\frac{1}{d}})$
  - assumes that d is a constant

Outline

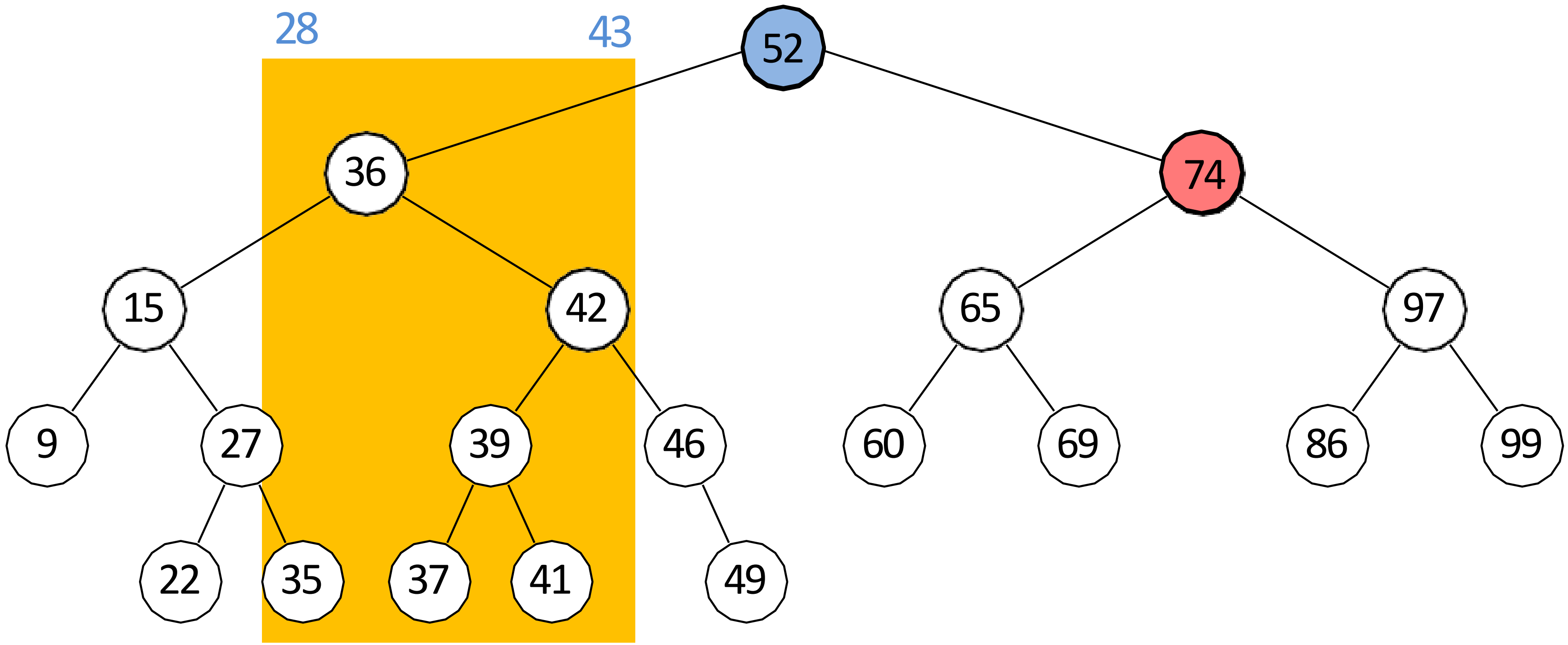
- Range-Searching in Dictionaries for Points
  - Range Search
  - Multi-Dimensional Data
  - Quadtrees
  - kd-Trees
  - Range Trees
  - Conclusion

Towards Range Trees

- Quadtrees and kd-trees
  - intuitive and simple
  - but both may be slow for range searches
  - quadtrees are also potentially wasteful in space
- Consider BST/AVL trees
  - efficient for one-dimensional dictionaries, if balanced
    - range search is also efficient
  - can we use ideas from BST/AVL trees for multi dimensional dictionaries?
- First let us consider range search in BST
  - all searches will be inclusive of the boundaries
  - *BST::RangeSearch-recursive*$(T, 28, 43)$
    - search includes both 28 and 43
      - easy to modify when one or both endpoints are excluded

BST Range Search example

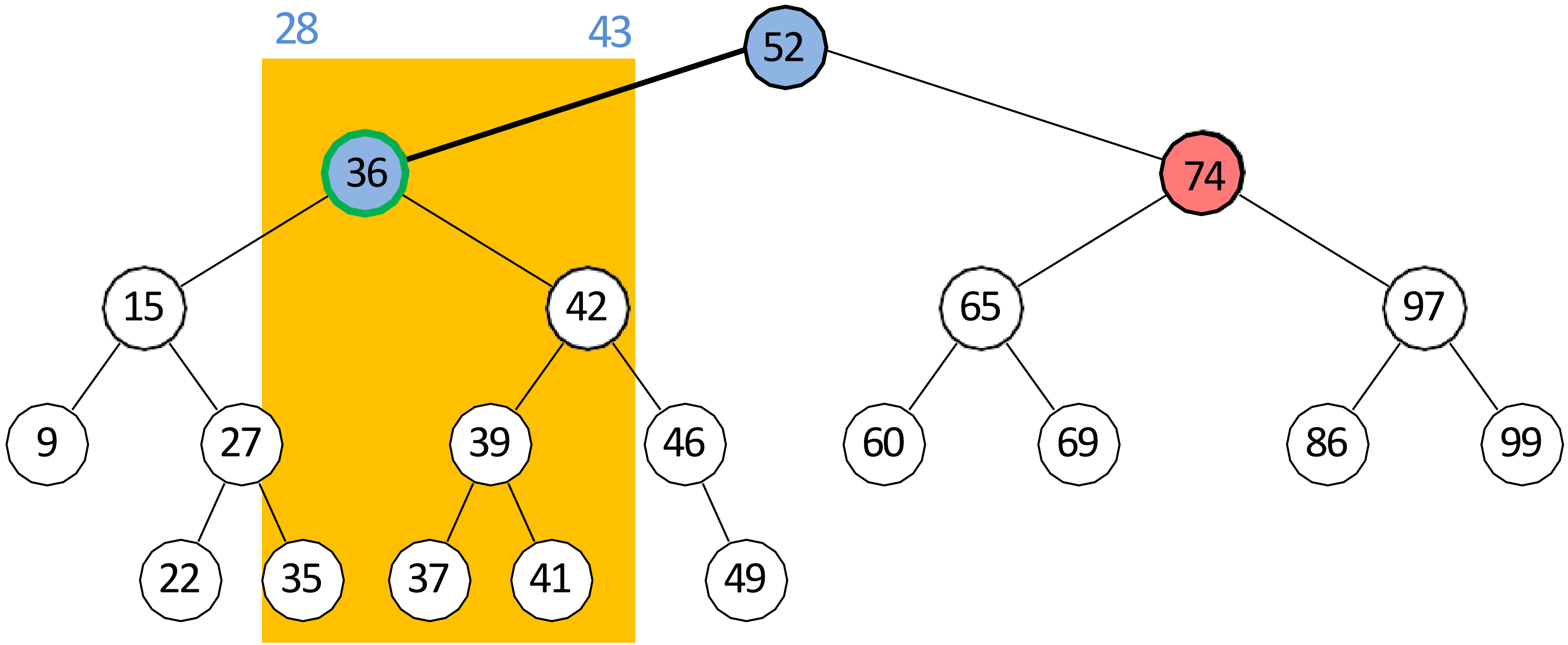
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
    - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
    - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
    - inside node, subtree completely inside range query

BST Range Search example

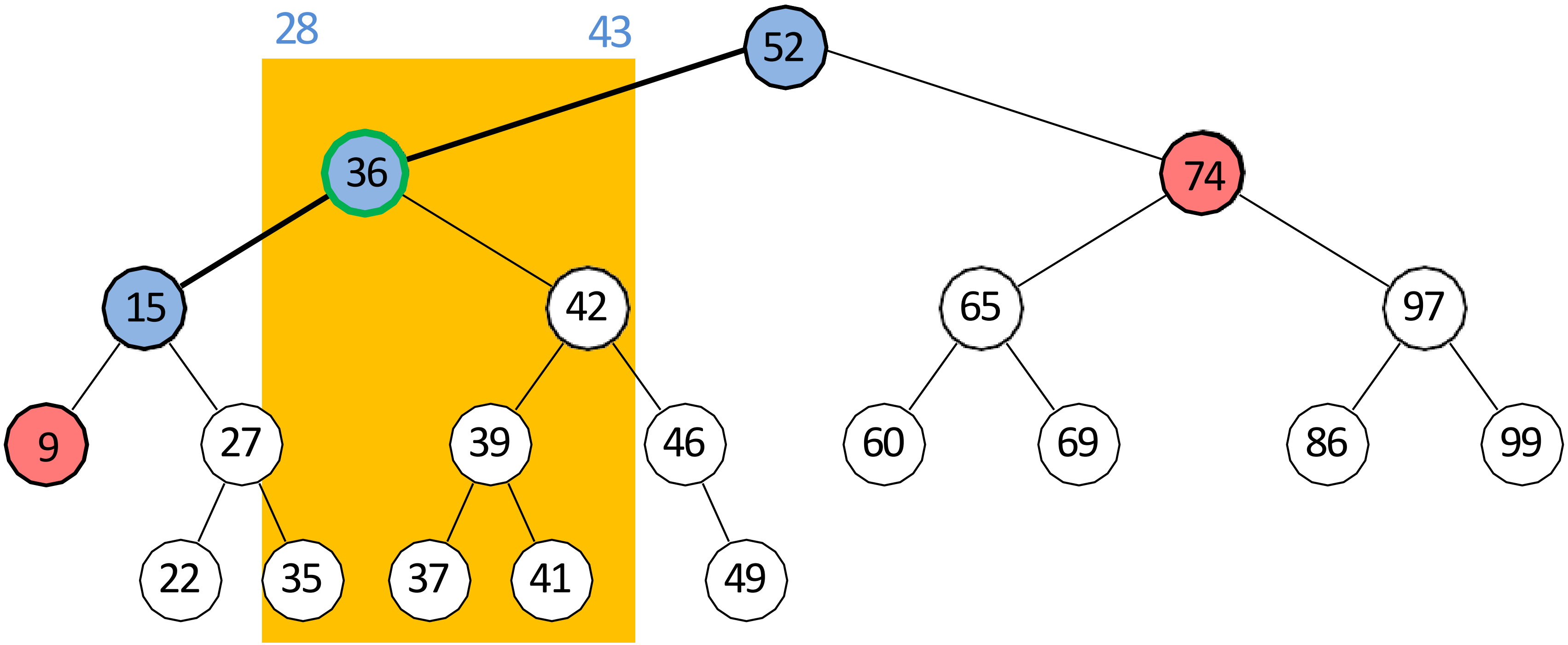
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

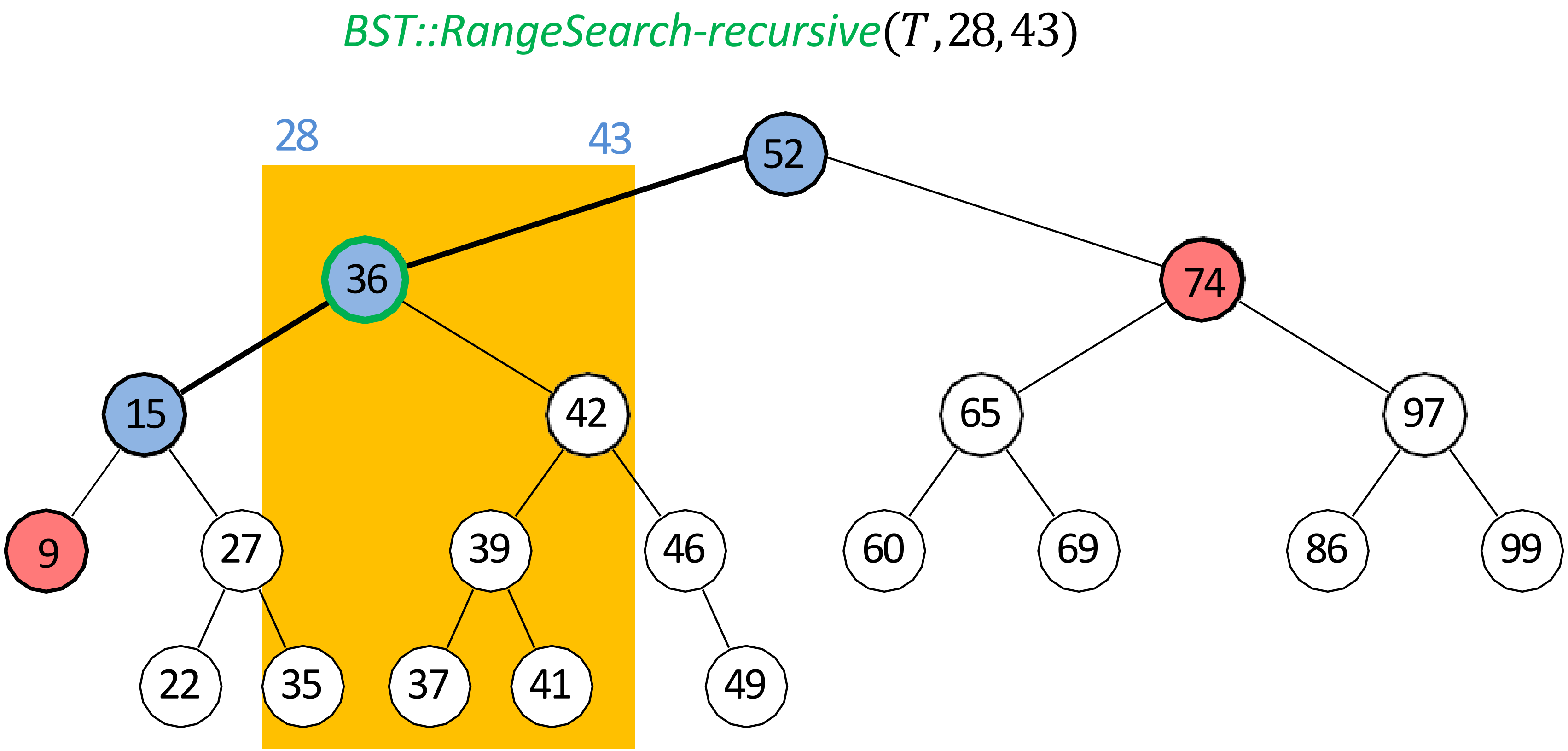
BST Range Search example

BST::RangeSearch-recursive$(T, 28, 43)$



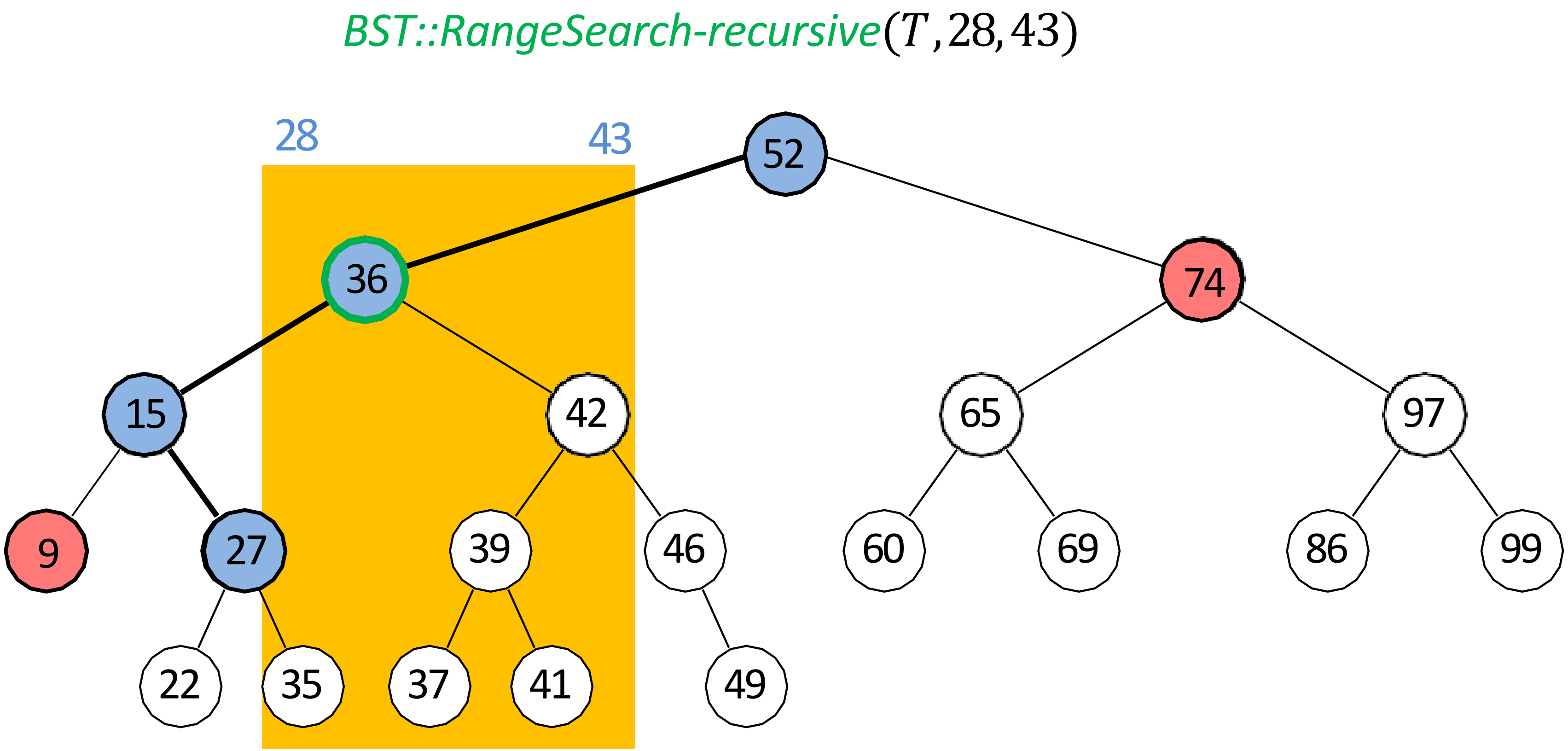
- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

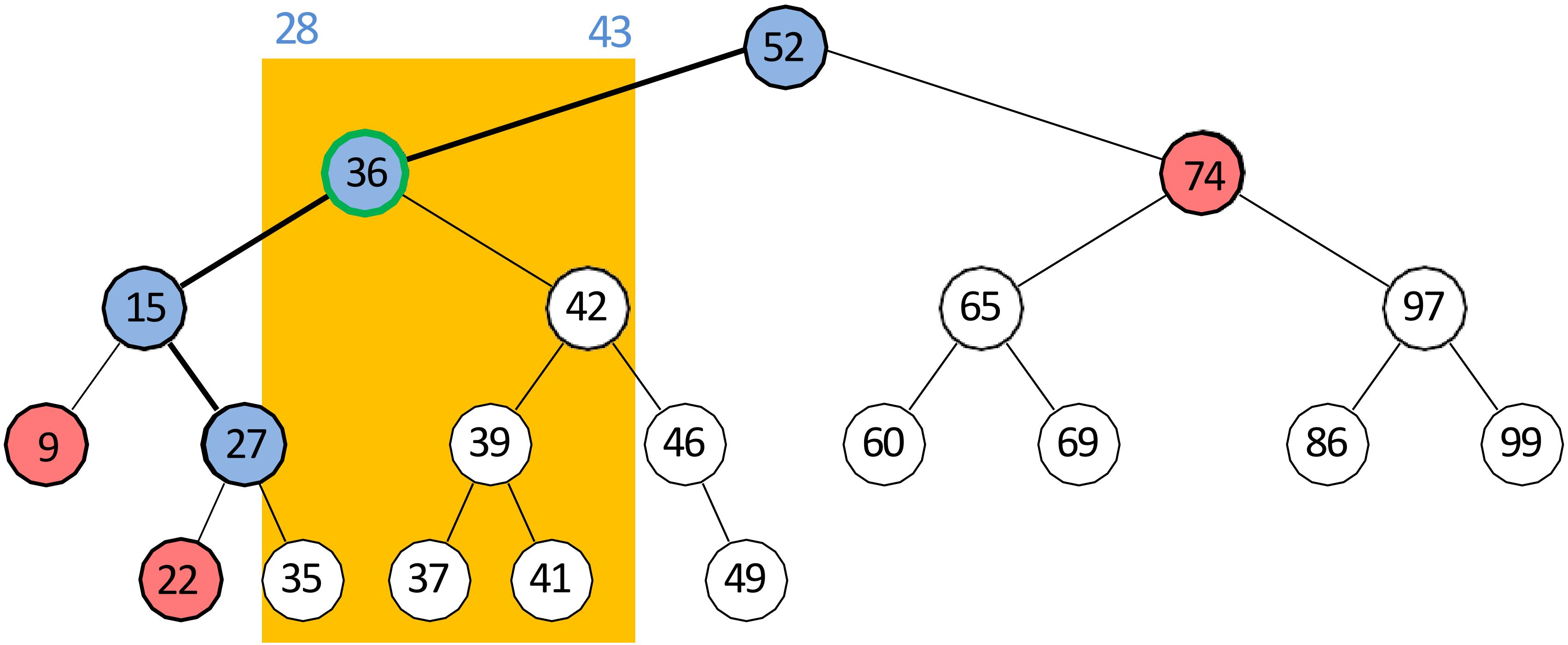
BST Range Search example



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example

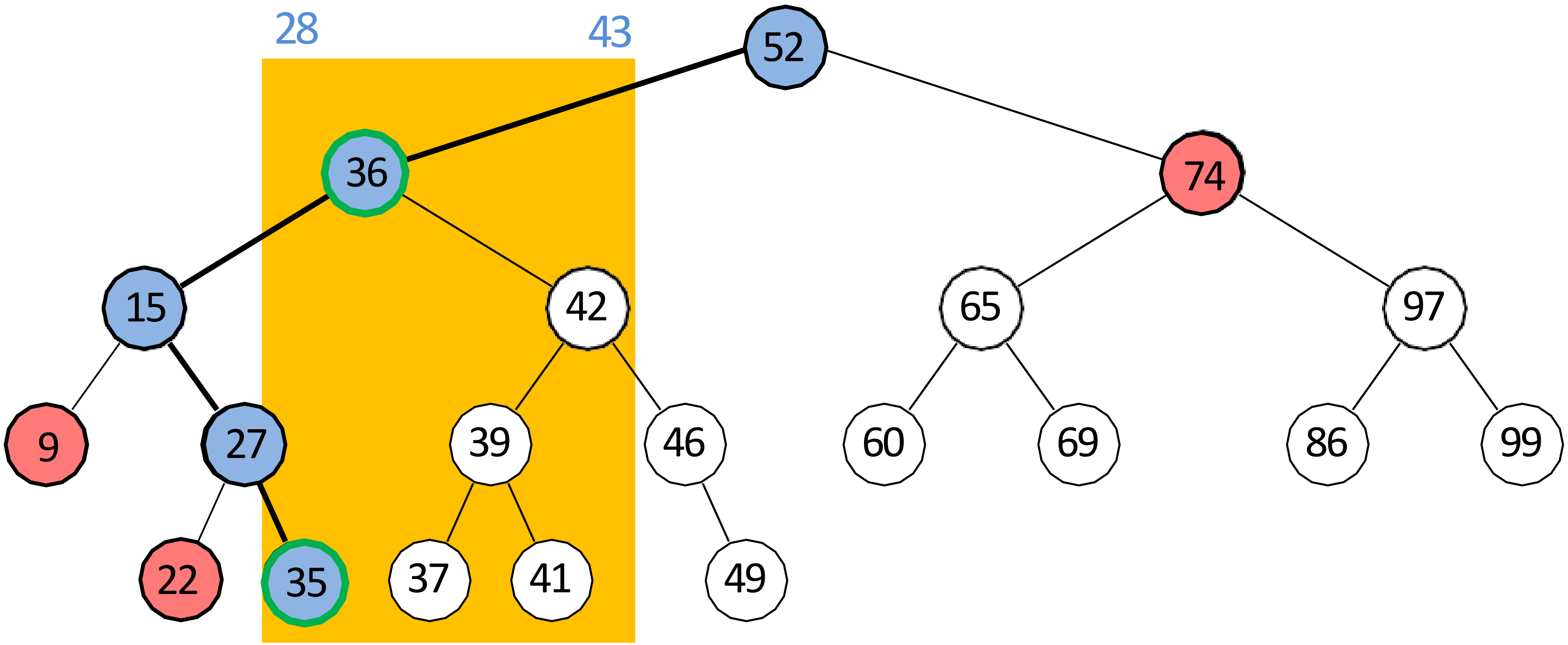
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example

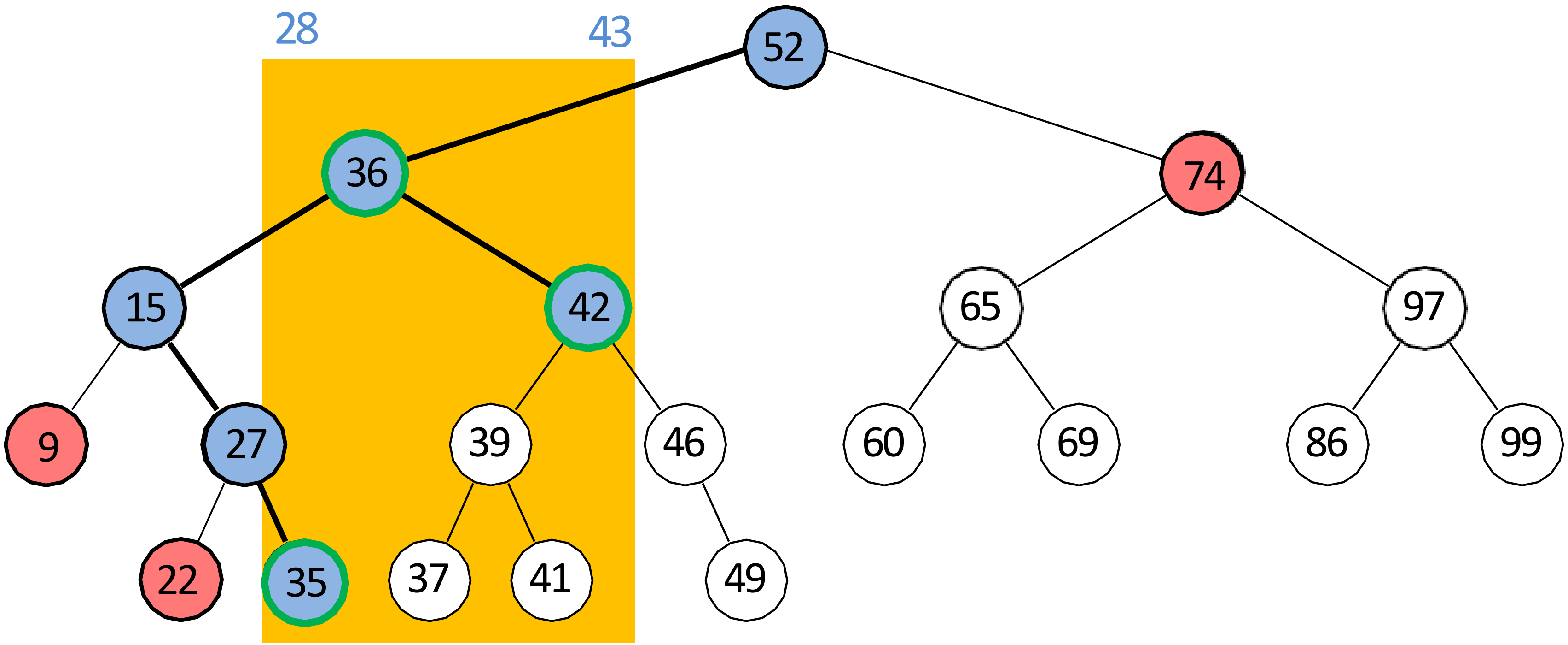
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example

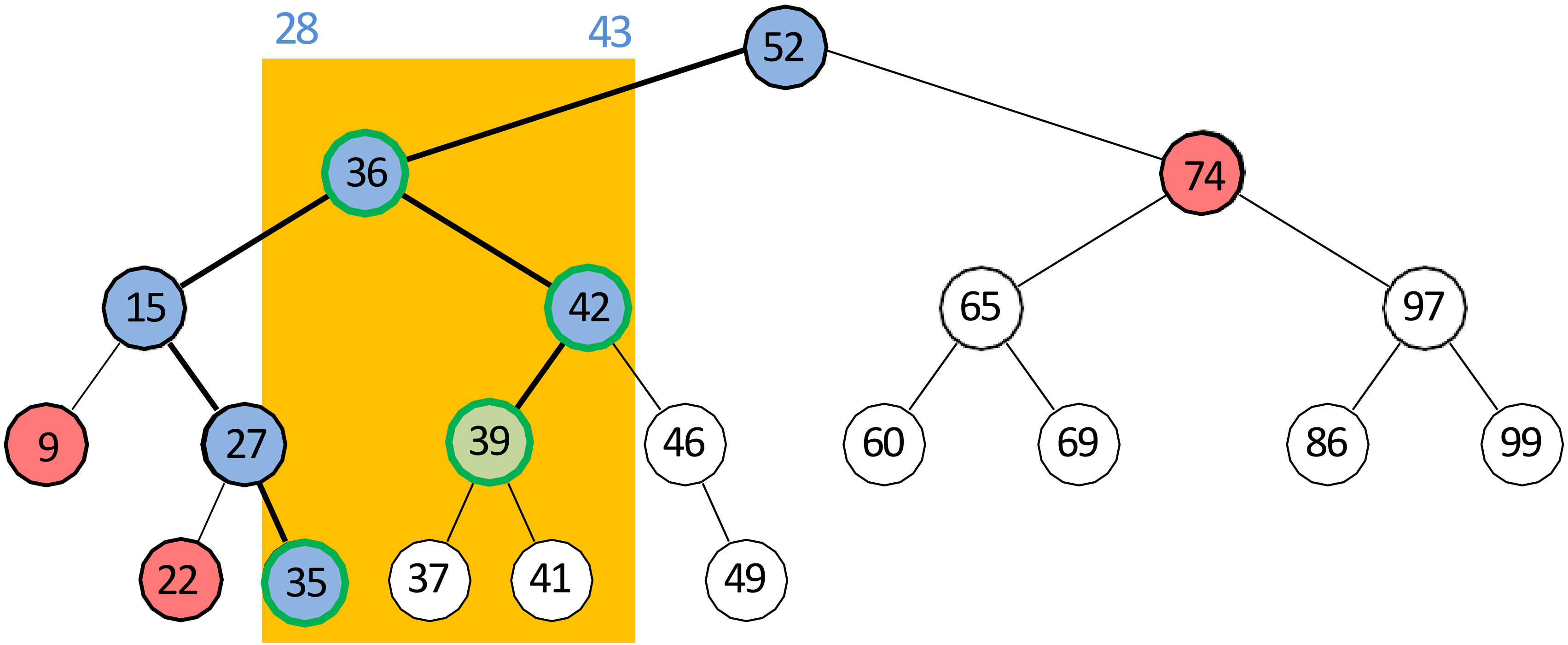
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example

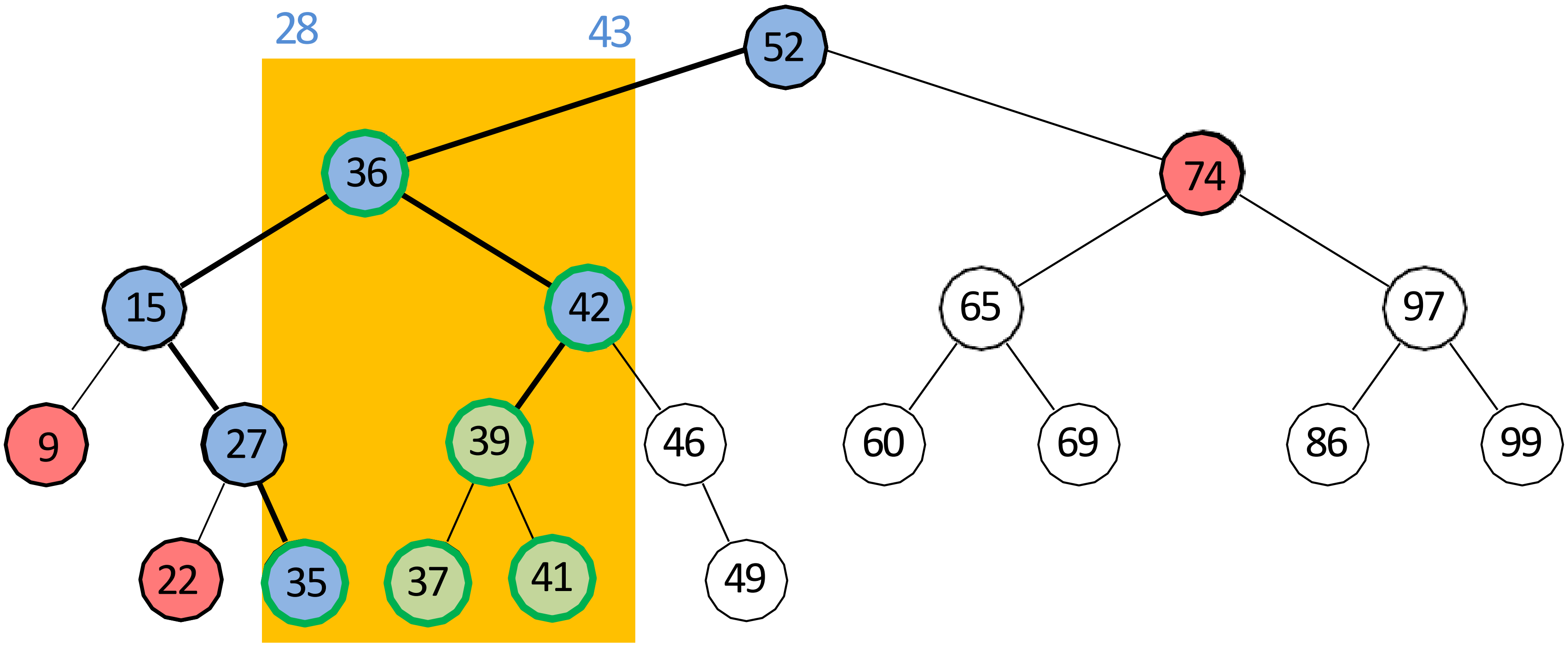
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
    - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
    - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
    - inside node, subtree completely inside range query

BST Range Search example

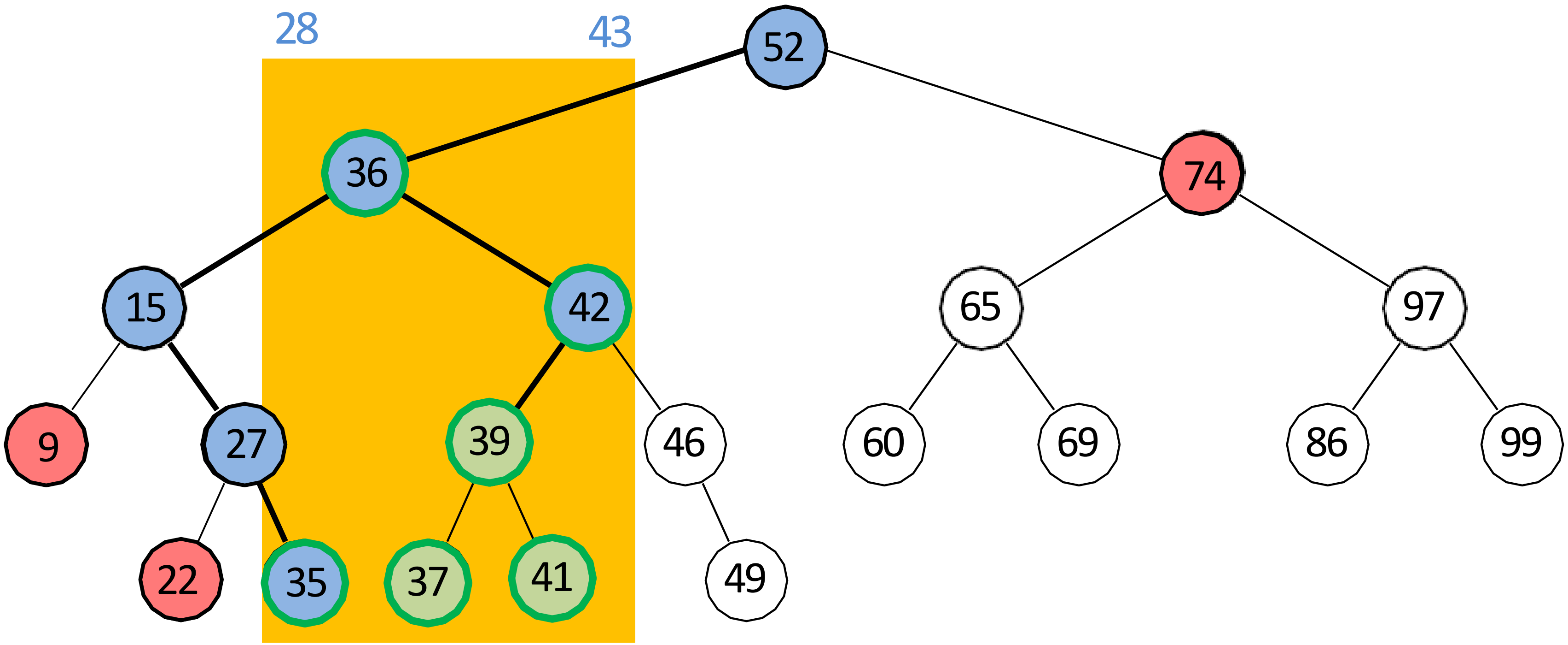
BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

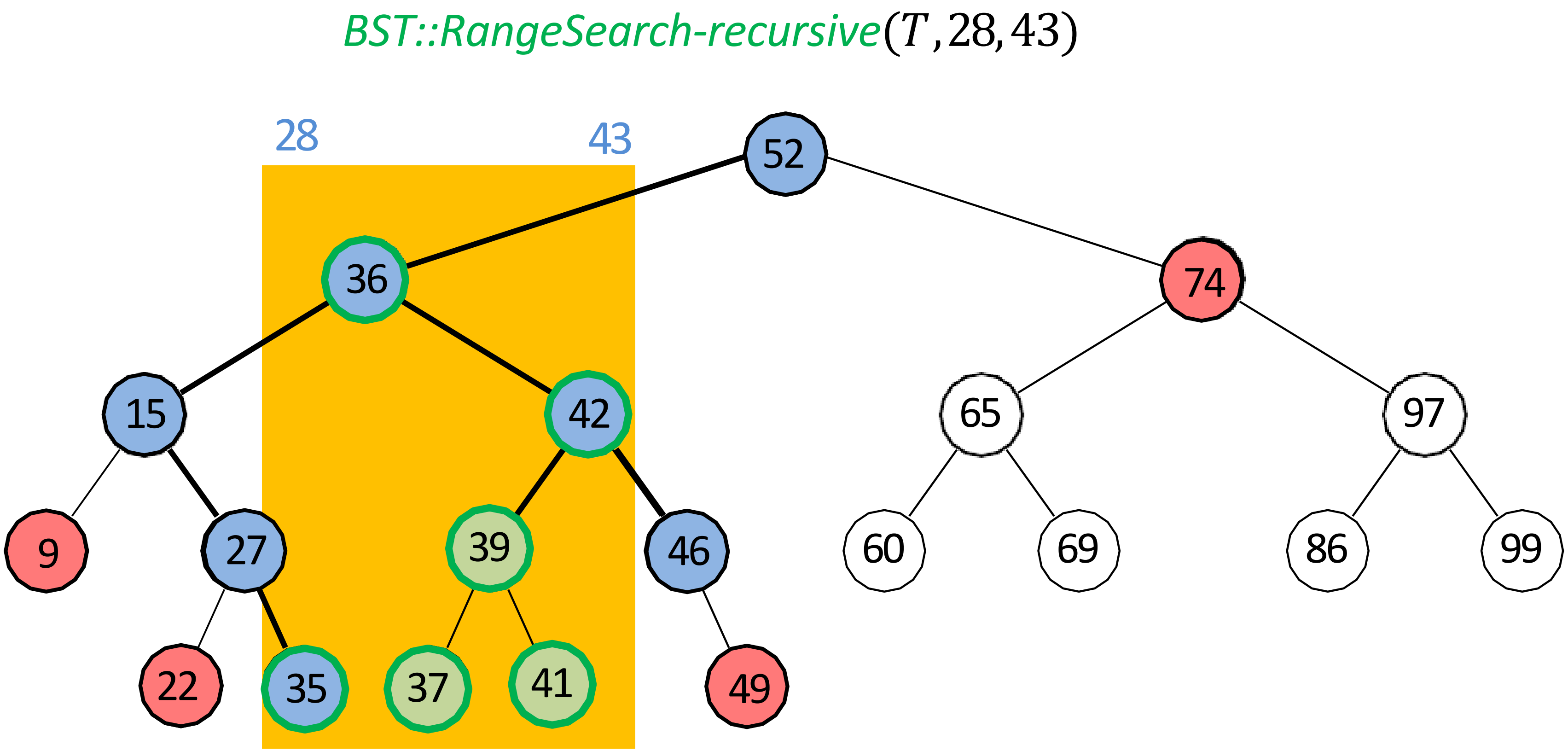
BST Range Search example

BST::RangeSearch-recursive$(T, 28, 43)$



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search example



- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
- green node : all the keys in the subtree are in the range
  - inside node, subtree completely inside range query

BST Range Search

BST::RangeSearch-recursive$(r \leftarrow root, k_1, k_2)$

r: root of a binary search tree, k_1, k_2: search keys

Returns keys in subtree at r that are in range $[k_1, k_2]$

if $r = NULL$ **then return** $\emptyset$

$L \leftarrow \emptyset, R \leftarrow \emptyset$

if $r.key < k_1$ **then**

  $R \leftarrow$ *BST::RangeSearch-recusive*$(r.right, k_1, k_2)$

if $r.key > k_2$ **then**

  $L \leftarrow$ *BST-RangeSearch-recursive*$(r.left, k_1, k_2)$

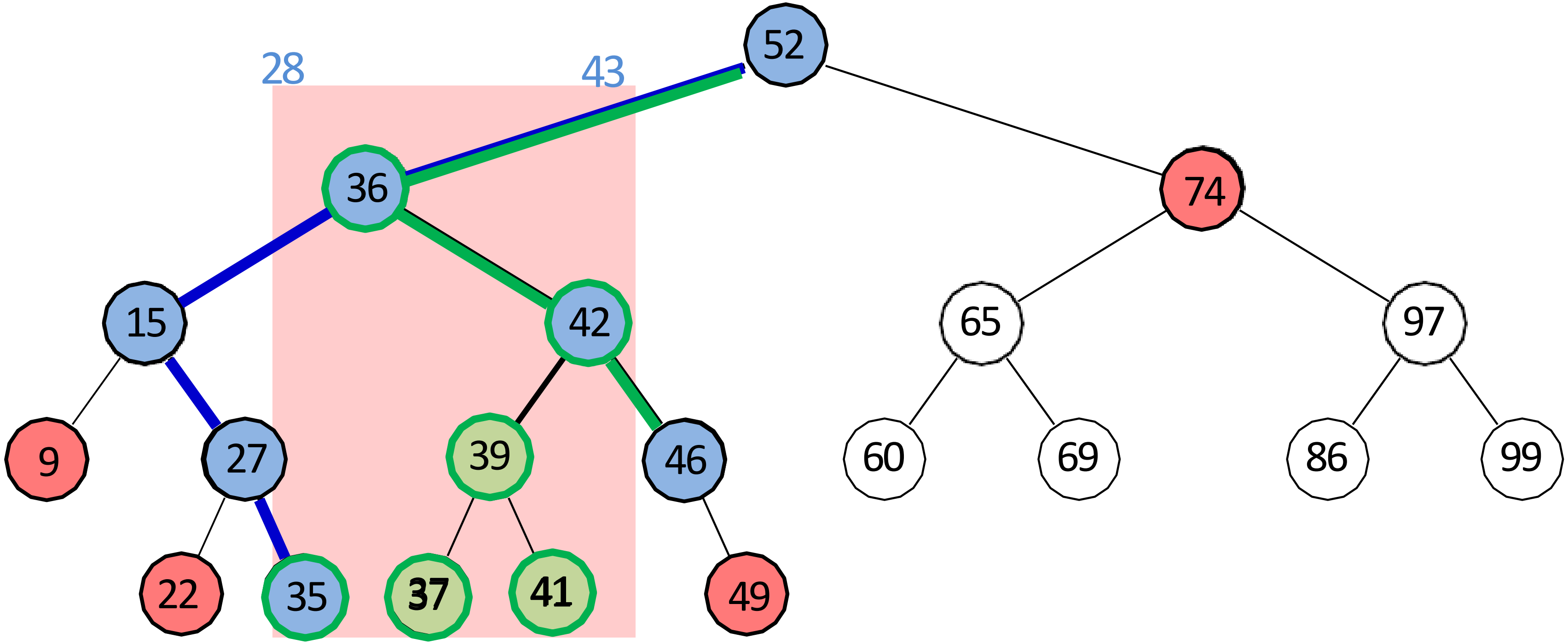
if $k_1 \leq r.key \leq k_2$ **then**

  return $L \cup \{r.key\} \cup R$

else return $L \cup R$

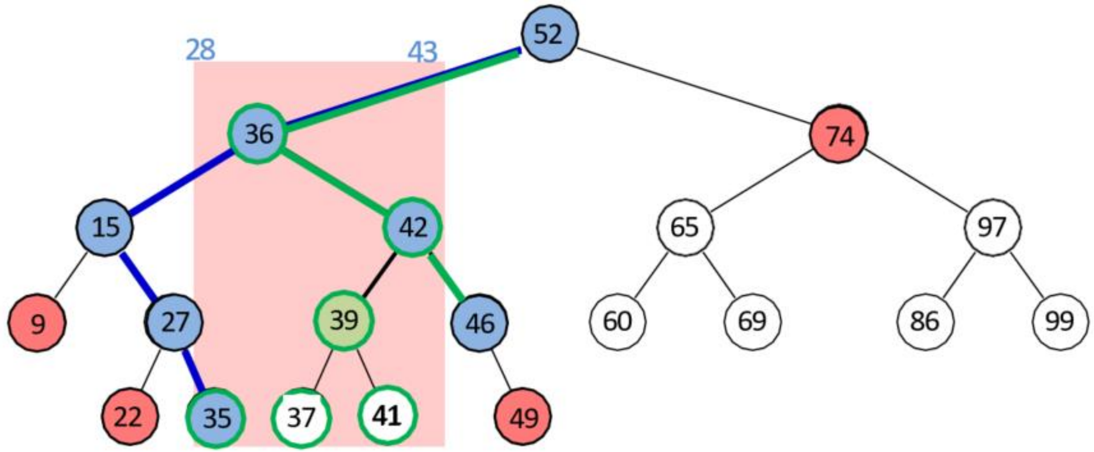
- Keys returned in sorted order

Modified BST Range Search



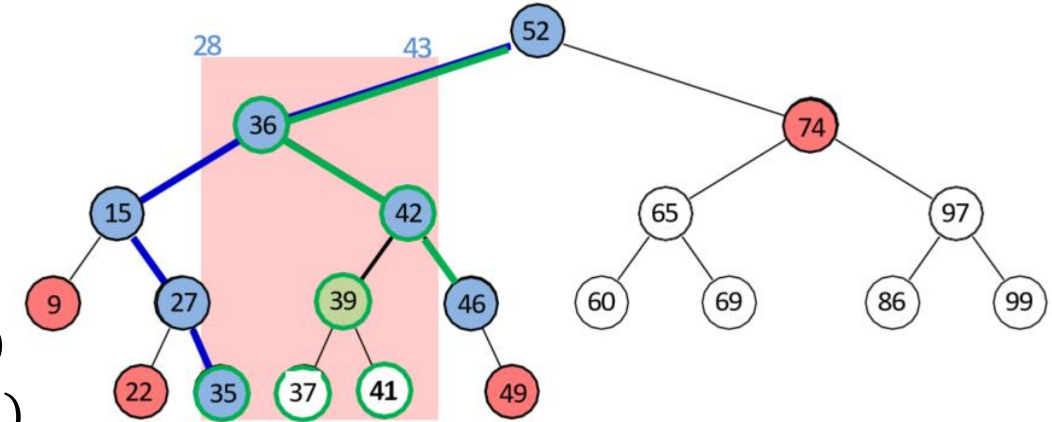
- Search for left boundary k_1 : this gives path ***P*₁**
- Search for right boundary k_2 : this gives path ***P*₂**
- Boundary (blue nodes) are exactly all the nodes on paths P_1 and P_2
- Nodes are partitioned into three groups: boundary, outside, inside

Modified BST Range Search



- Boundary nodes: nodes in P_1 and P_2
  - check if boundary nodes are in the search range
- Outside nodes: nodes that are left of P_1 or right of P_2
  - outside nodes are not in the search range
  - range search is never called on an outside node
- Inside nodes: nodes that are right of P_1 and left of P_2
  - **we will stop the search at the topmost inside node**
  - all descendants of such node are in the range, just report them without search
  - this is not more efficient for BST range search, but useful when we develop 2D search in ***range trees***

Modified BST Range Search Analysis



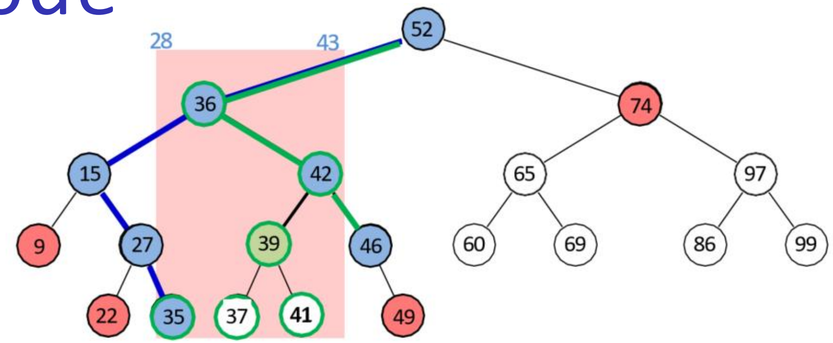
- Assume balanced BST
- Running time consists of
  1. search for path ***P*₁** is $O(\log n)$
  2. search for path ***P*₂** is $O(\log n)$
  3. check if boundary nodes in the range
     - $O(1)$ at each boundary node, there are $O(\log n)$ of them, $O(\log n)$ total time
  4. spend $O(1)$ at each topmost inside node
     - since each topmost inside node is a child of boundary node, there are at most $O(\log n)$ topmost inside nodes, so total time $O(\log n)$
  5. report descendants in subtrees of all topmost inside nodes
     - topmost nodes are disjoint, so #descendants for inside topmost nodes is at most s, output size

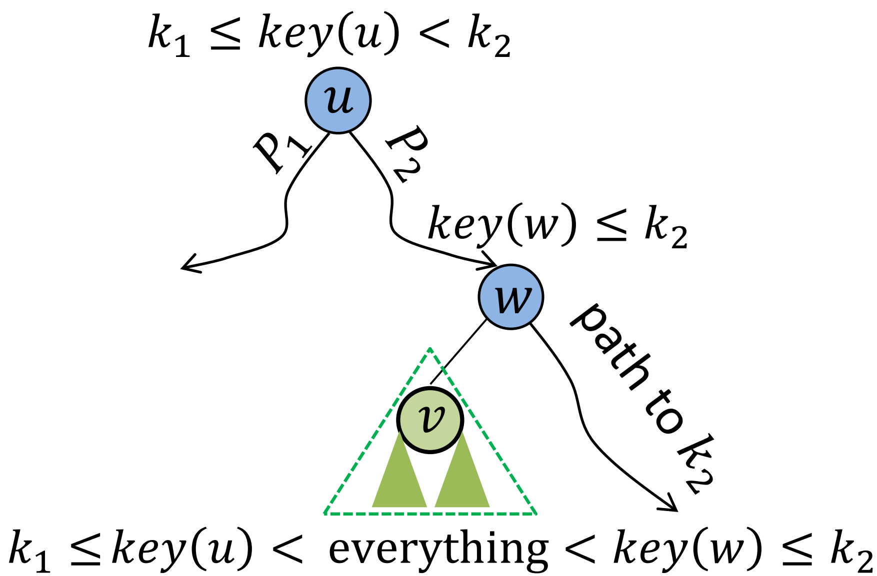
$$\sum_{\text{topmost inside node } v} \#\text{descendants of } v \le s$$

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

How to Find Top Inside Node

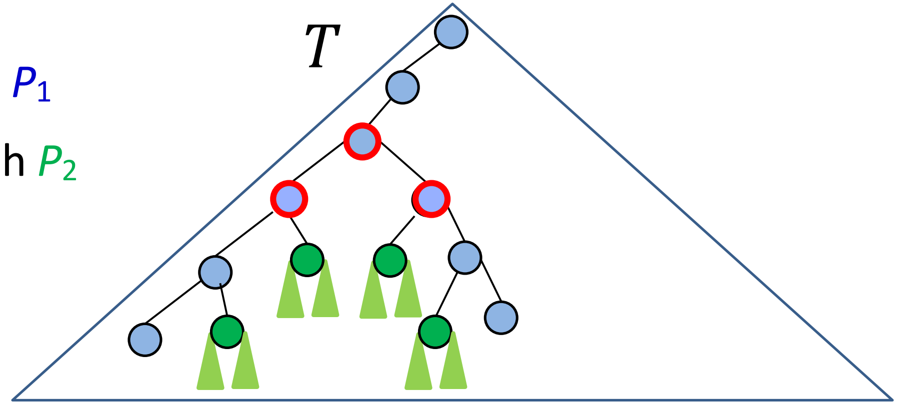
- v is a top inside node if
  - v is not is in P_1 or P_2
  - parent of v is in P_1 or P_2 (but not both)
  - if parent is in P_1, then v is right child
  - if parent is in P_2, then v is left child





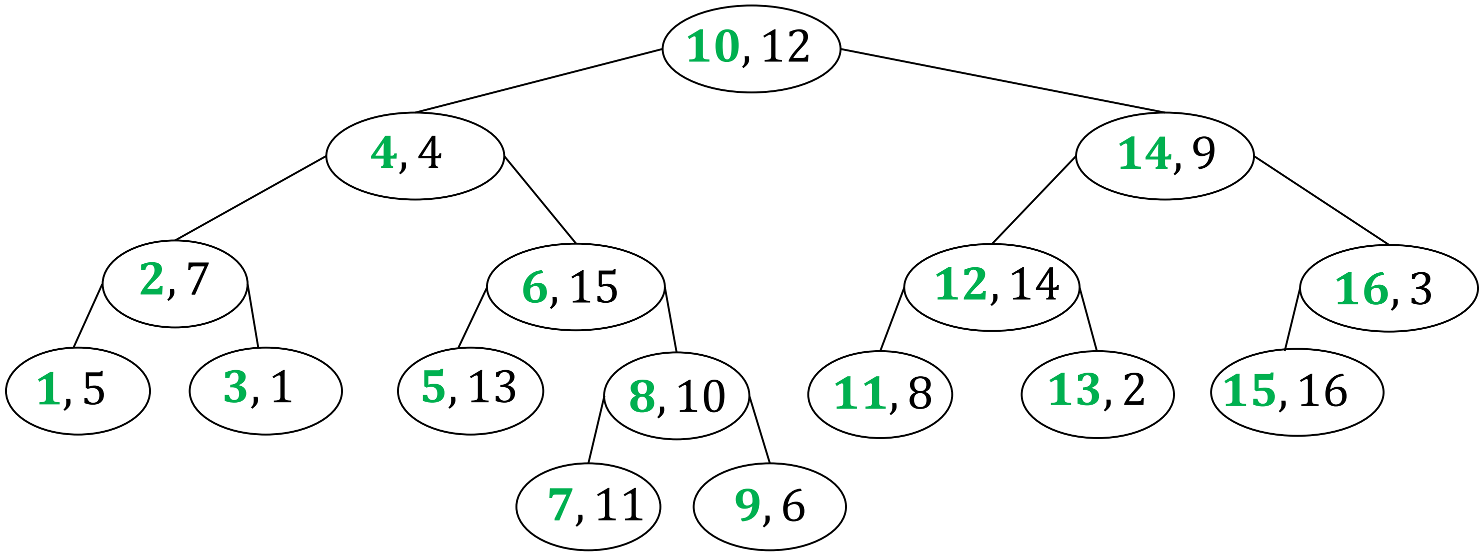
- Thus for each top inside node can report all descendants, no need for search
  - BST range search does not become not faster overall, but top inside nodes are important for $2D$ range search efficiency
  - also important if need to just count the number of points in the search range

Modified BST Range Search Summary



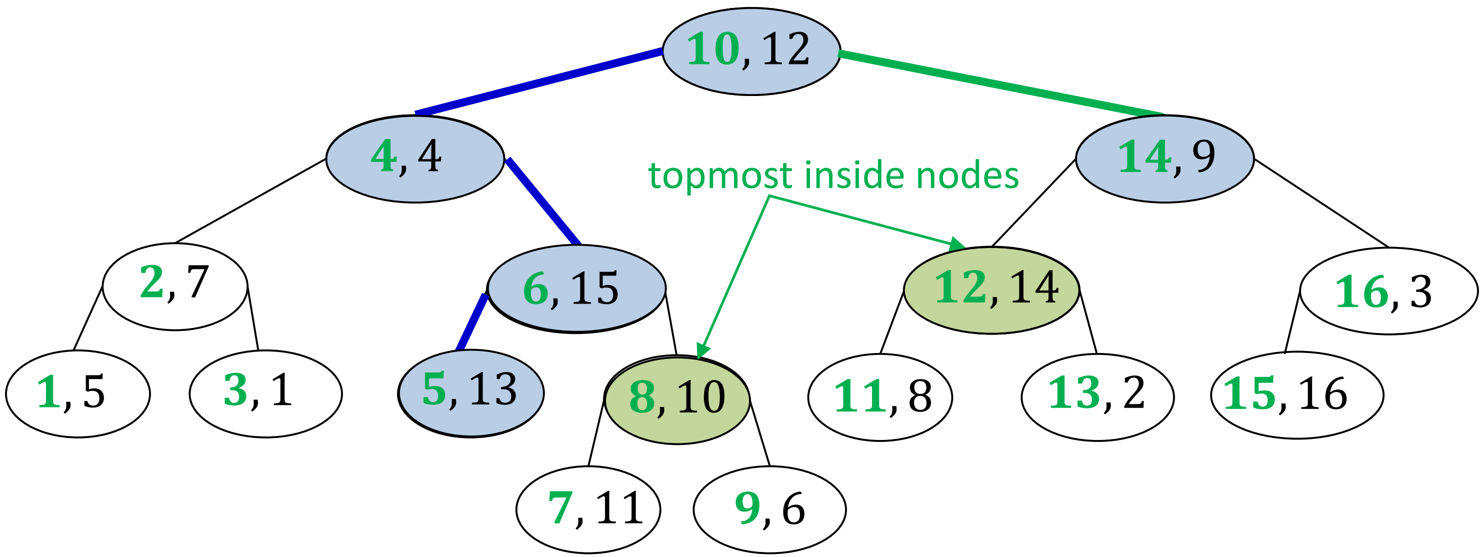
- Search for k_1: this gives left boundary path P_1
- Search for k_2: this gives right boundary path P_2
- Find all topmost inside nodes
  - not in P_1 or P_2
  - left children of nodes in P_2
  - right children of nodes in P_1
- Inside node (which is not a topmost inside) is in a subtree of some topmost inside node
- Set of inside nodes = union disjoint subtrees rooted at topmost inside nodes
- To output nodes in the search range
  - test each node in P_1 , P_2 and report if in range
  - go over all topmost inside nodes and report all nodes in their subtree

2D Range Tree Motivation



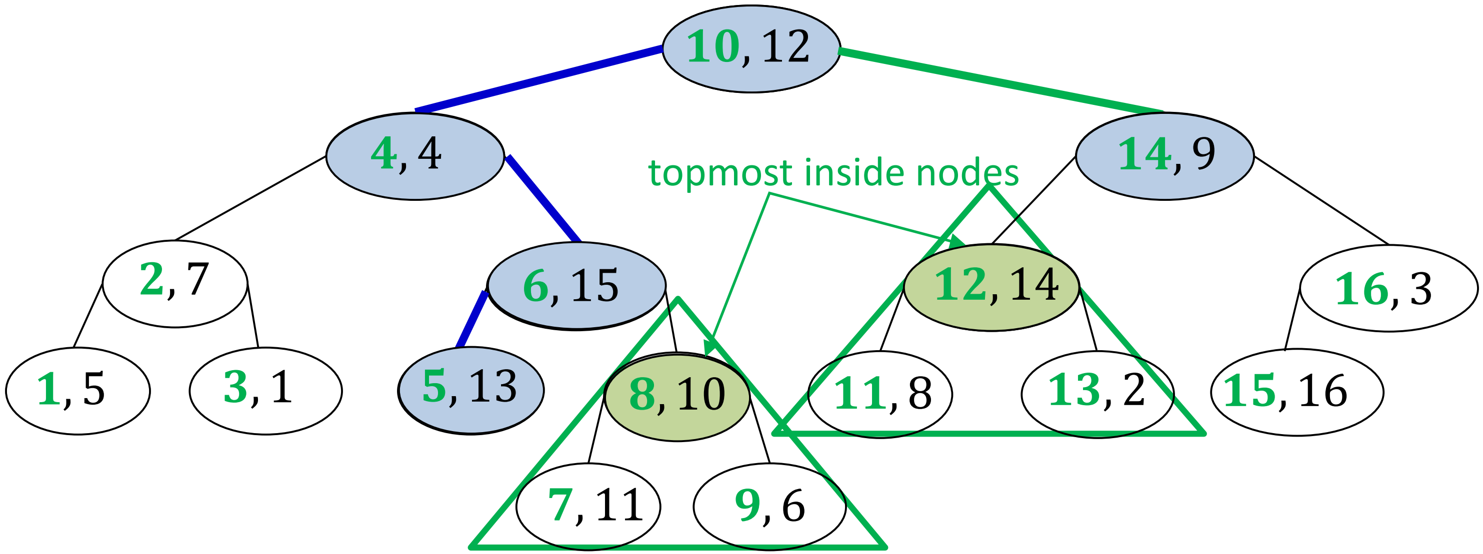
- Have a set of 2D points
  - $S = \{(1{,}5), (2{,}7), (3{,}1), (4{,}4), (5{,}13), (6{,}15)(7{,}11), (8{,}10), (9{,}6), (10{,}12), (11{,}8), (12{,}14), (13{,}2), (14{,}9), (15{,}16), (16{,}3)\}$
- Example of 2D range search
- *BST-RangeSearch*$(T, 5, 14, 5, 9)$
  - find all points with $5 \le x \le 14$ and $5 \le y \le 9$
- Construct BST with x-coordinate key
  - recall that points are in general positon, so all x-keys are distinct
    - for any (x_1, y_1) and (x_2, y_2) in our set of points, $x_1 \neq x_2$
  - can search efficiently based only on x-coordinate

2D Range Tree Motivation



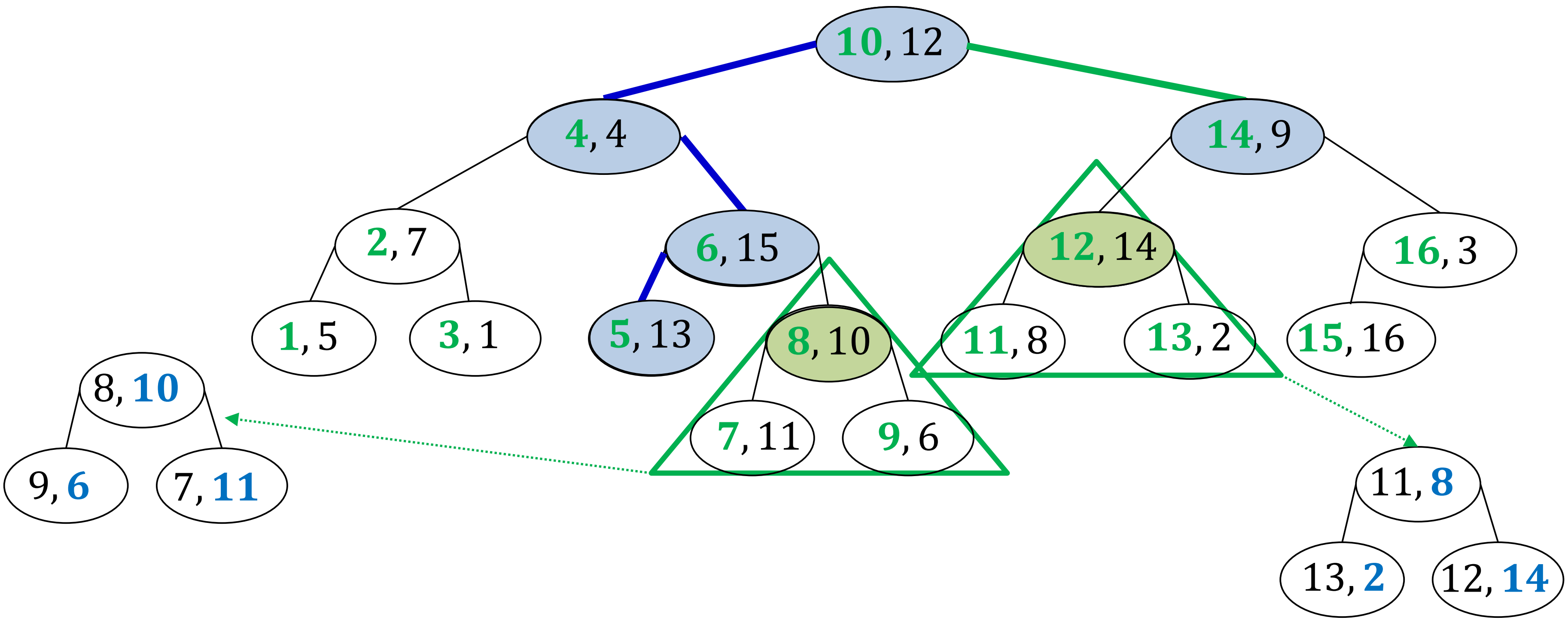
- Consider $2D$ range search *BST-RangeSearch*$(T, 5, 14, 5, 9)$
- Could first perform *BST-RangeSearch*$(T, 5, 14)$
  - let A be the set of nodes *BST-RangeSearch*$(T, 5, 14)$ returns
    - $A = \{(10{,}12), (6{,}15), (5{,}13), (14{,}9), (8{,}10), (7{,}11), (9{,}6), (12{,}14), (11{,}8), (13{,}2)\}$
  - let B be the set of nodes *BST-RangeSearch*$(T, 5, 14, 5, 9)$ should return
    - $B \subseteq A$
  - Need to go over all nodes in A and check if their y-coordinate is in valid range, $O(|A|)$
    - could be very inefficient
    - for example, $|A|$ can be, say $\Theta(n)$ and $|B|$ could be $O(1)$
      - $O(n)$, as bad as exhaustive search and worse than kd-trees search, $O(|B| + \sqrt{n})$

2D Range Tree Motivation



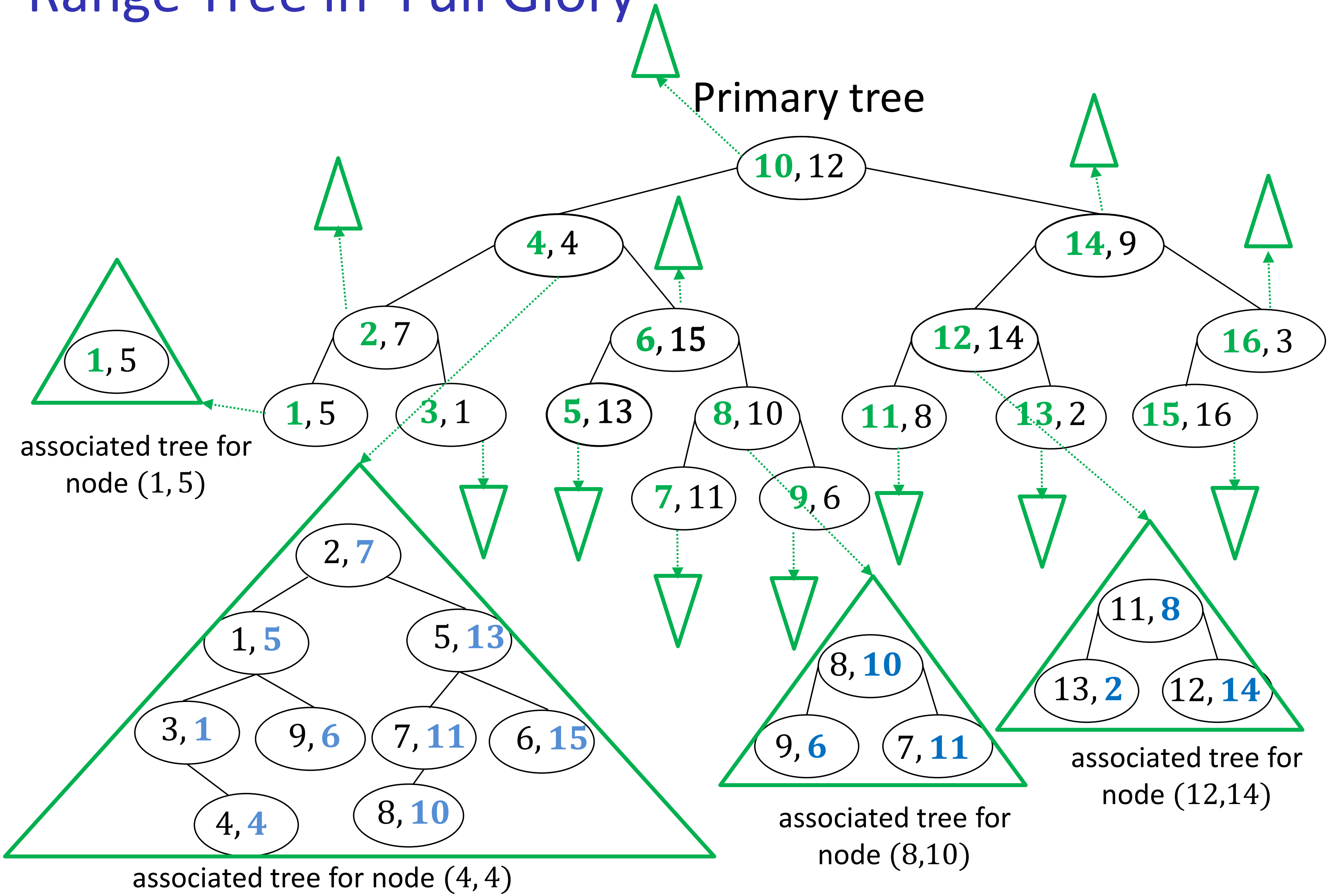
- Consider $2D$ range search *BST-RangeSearch*$(T, 5, 14, 5, 9)$
- First perform only **partial** *BST-RangeSearch*$(T, 5, 14)$
    - find boundary and topmost inside nodes, takes $O(\log n)$ time
- Next
    - for boundary nodes, check if **both** x and y coordinates are in the range, takes $O(\log n)$ time as there are $O(\log n)$ boundary nodes
    - inside nodes are stored in $O(\log n)$ subtrees, with a topmost inside node as a root of each subtree
        - if we could search these subtrees, time would be very efficient
        - however these subtrees do not support efficient search by y coordinate

2D Range Tree Motivation

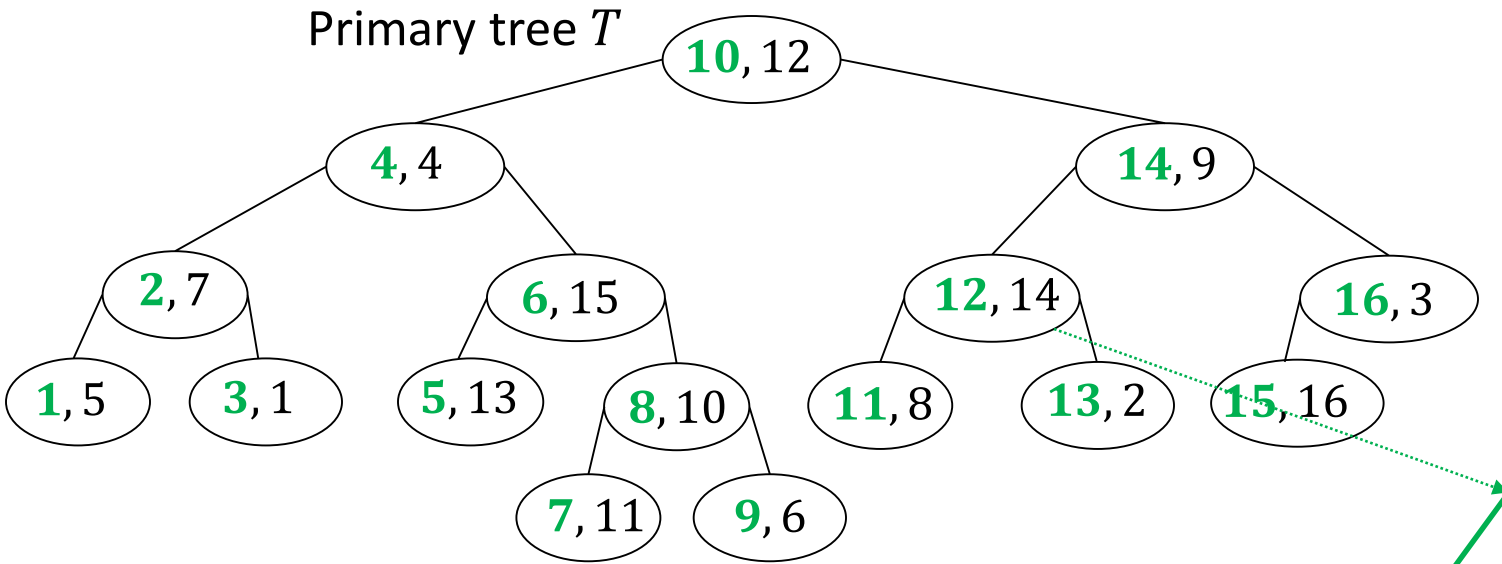


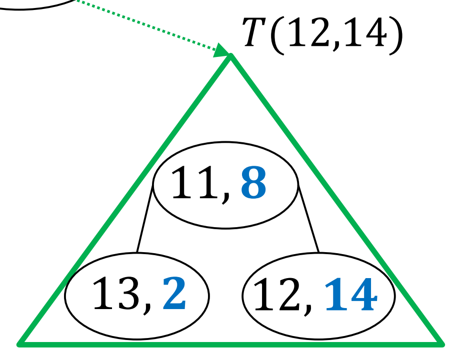
- Need to search subtrees by y-coordinate, but they are x-coordinate based
- Brute-force solution
  - need an associate balanced BST tree for each node v
    - stores same items as the main (primary) subtree rooted at node v
    - but key is y-coordinate

Range Tree in ‘Full Glory’



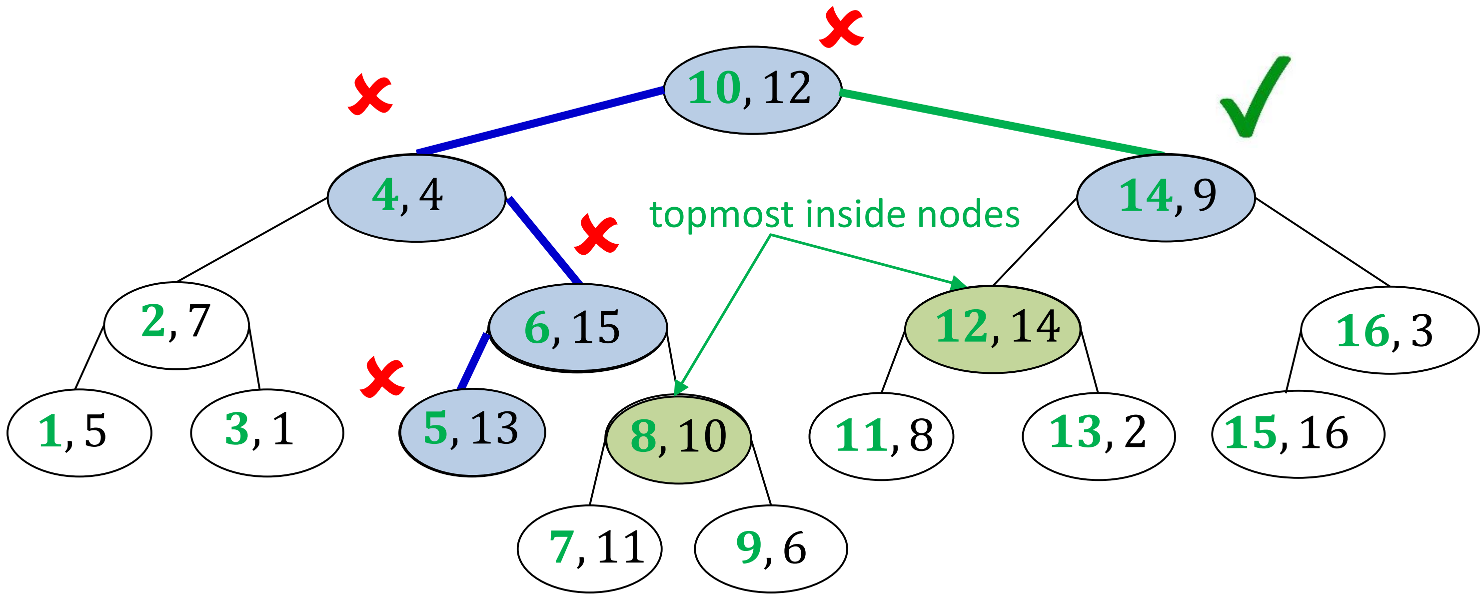
2-dimensional Range Trees Full Definition





- Points $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$
- Range tree is a tree of trees (a *multi-level* data structure)
  - Primary structure
    - balanced BST T storing S and uses x-coordinates as keys
    - assume T is balanced, so height is $O(\log n)$
  - Each node v of T stores an associated tree $T(v)$, which is a balanced BST
    - let $S(v)$ be all descendants of v in T, including v
    - $T(v)$ stores $S(v)$ in BST, using y-coordinates as key
      - note that v is not necessarily the root of $T(v)$

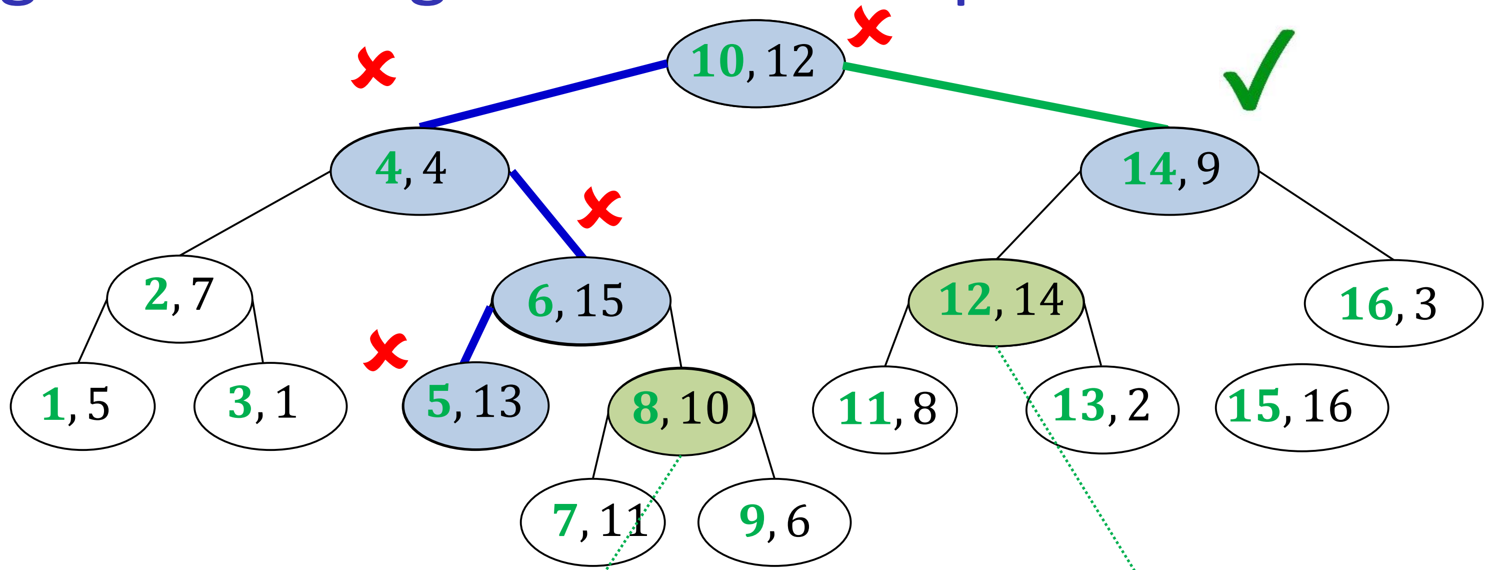
Range search in 2D Range Tree Overview



- *RangeTree::RangeSearch*(T, x_1, x_2, y_1, y_2)
  - *RangeTree::RangeSearch*$(T, 5, 14, 5, 9)$

1. Perform *modified BST-RangeSearch*$(T, 5, 14)$
   - find boundary and topmost inside nodes, but do not go through the inside subtrees
   - modified version takes $O(\log n)$ time
     - does not visit all the nodes in valid range for *BST-RangeSearch*$(T, 5, 14)$
2. Check if boundary nodes have valid x-coordinate **and** valid y-coordinate
3. For every topmost inside node v, search in associated tree *BST::RangeSearch*$(T(v), 5, 9)$

Range Tree Range Search Example Finished

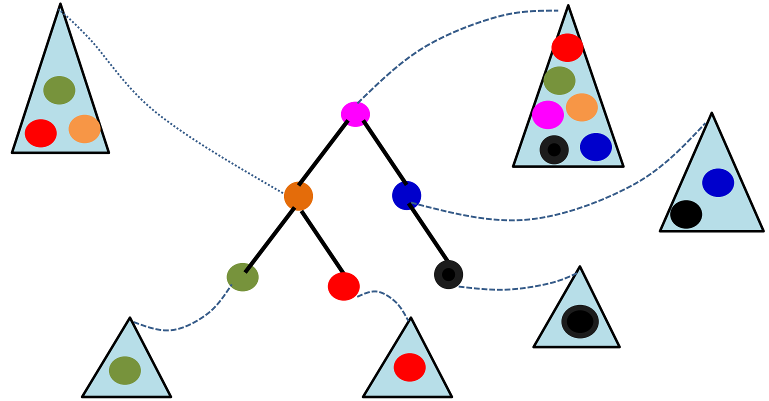


- *RangeTree::RangeSearch*$(T, 5, 14, 5, 9)$
- For every topmost inside node v, search in associated tree *BST-RangeSearch*$(T(v), 5, 9)$

BST-rangeSearch$(T(8{,}10), 5{,}9)$

BST-RangeSearch$(T(12{,}14), 5{,}9)$

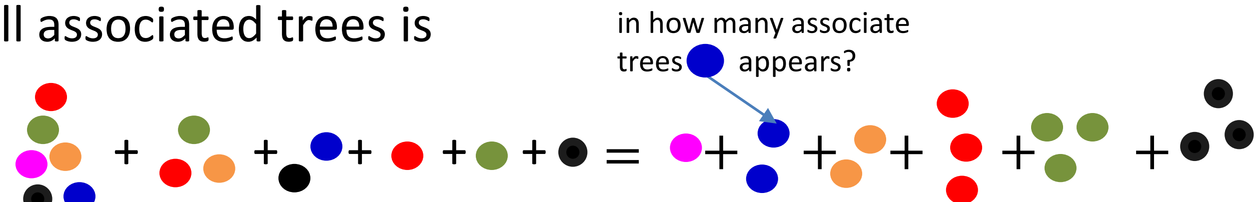
Range Tree Space Analysis



- Primary tree T uses $O(n)$ space
- For each v, associated tree $T(v)$ uses $O(|T(v)|)$ space

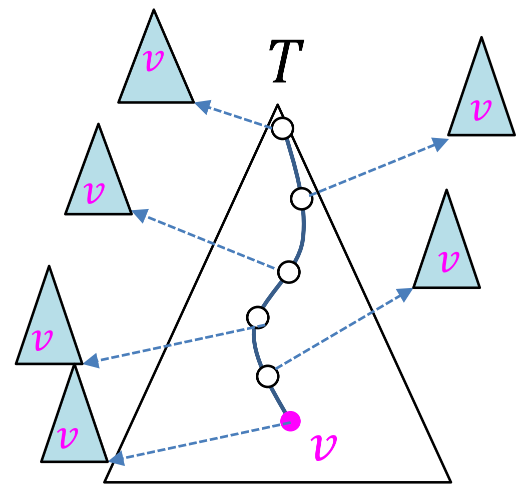
- Space for all associated trees is

$$\sum_{v\in T}|T(v)| =$$



$$= \sum_{v\in T} \text{\#of ancestors of } v \quad \text{(} \le c\log n \text{)}$$

$$\le \sum_{v\in T} c\log n = cn\log n$$



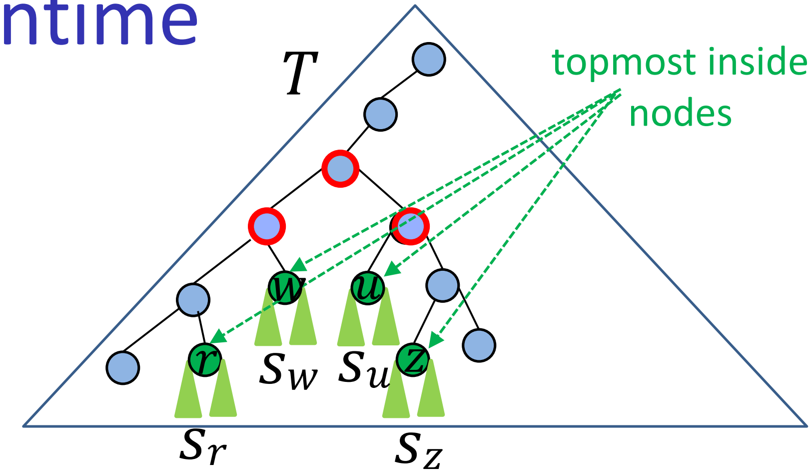
#of ancestors of v

- Space is $O(n\log n)$
  - in the worst case, have $n/2$ leaves at the last level, and space needed is $\Theta(n\log n)$

Range Trees: Dictionary Operations

- Search(x, y)
  - search by x coordinate in the primary tree T
- Insert(x, y)
  - first, insert point by x-coordinate into the primary tree T
  - then walk up to root and insert point by y-coordinate in *all* $T(v)$ of nodes v on path to root
- Delete
  - analogous to insertion
- **Problem**
  - want binary search trees to be balanced
  - if we use AVL-trees, it makes insert/delete very slow
    - rotations change primary tree structure and require rebuilding of associate trees
  - instead of rotations, can allow certain imbalance, rebuild entire subtree if imbalance becomes too large
    - no details

Range Trees: Range Search Runtime



inside subtrees do not have any nodes in common

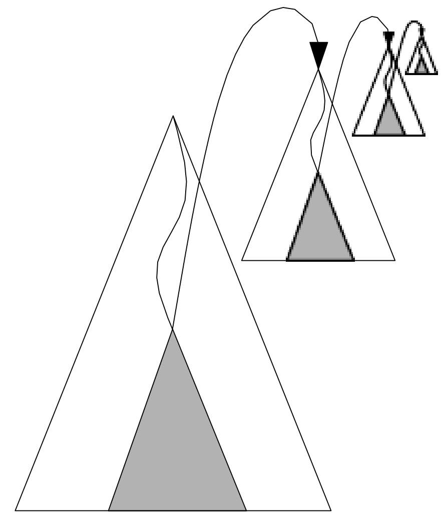
- Find boundary nodes in the primary tree and check if keys are in the range
  - $O(\log n)$
- Find topmost inside nodes in primary tree
  - $O(\log n)$
- For each topmost inside node v, perform range search for y-range in associate tree
  - $O(\log n)$ topmost inside nodes
  - let s_v be #items returned for the subtree of topmost node v
  - running time for one search is $O(\log n + s_v)$

$$\sum_{\text{topmost inside node } v} c(\log n + s_v) = \underbrace{\sum_{\text{topmost inside node } v} c\log n}_{O(\log^2 n)} + \underbrace{\sum_{\text{topmost inside node } v} cs_v}_{\leq cs}$$

- Time for range search in range tree: $O(s + \log^2 n)$
  - can make this even more efficient, but this is beyond the scope of the course

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
- Space-time tradeoff compared to kd trees



Outline

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

Range Search Data Structures Summary

- Quadtrees
  - simple, easy to implement insert/delete (i.e. dynamic set of points)
  - work well only if points evenly distributed
  - wastes space, especially for higher than two dimensions
- kd-trees
  - linear space
  - range search is $O(s+\sqrt{n})$
  - inserts/deletes destroy balance and range search time
    - fix with occasional rebuilt
- Range trees
  - fastest range search $O(s+\log^2 n)$
  - wastes some space
  - insert and delete destroy balance, but can fix this with occasional rebuilt