CS 240 – Data Structures and Data Management

Module 8: Range-Searching in Dictionaries for Points

A. Hunt and O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2023

Outline

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
  - input a range, i.e. interval $l = (x, x')$
    - may have open or closed ends
  - want to report all KVPs in the dictionary with $k \in l$
  - example

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

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

- 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
- Running time depends both on s and n
  - keep s as a parameter when analyzing runtime
  - $O(\log n + s)$ time would be the best possible with comparison based search

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*
  - range search can be done in $O(\log n + s)$ time

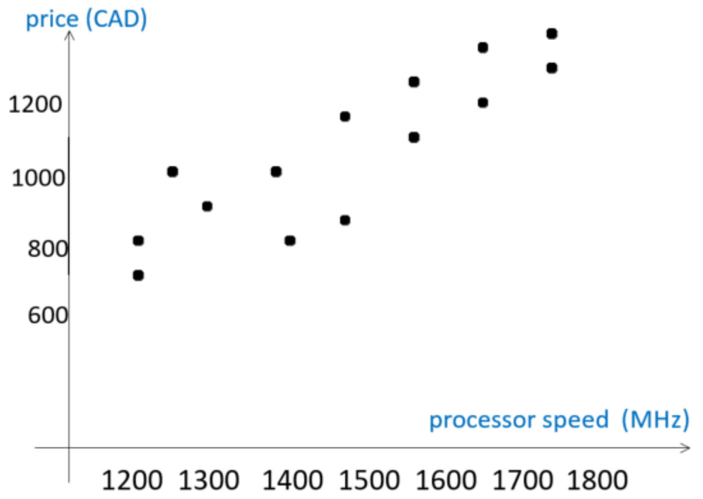
5	**10**	**11**	**17**	**18**	**33**	**45**	**51**	**55**	**77**
			i				i'		

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

Outline

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

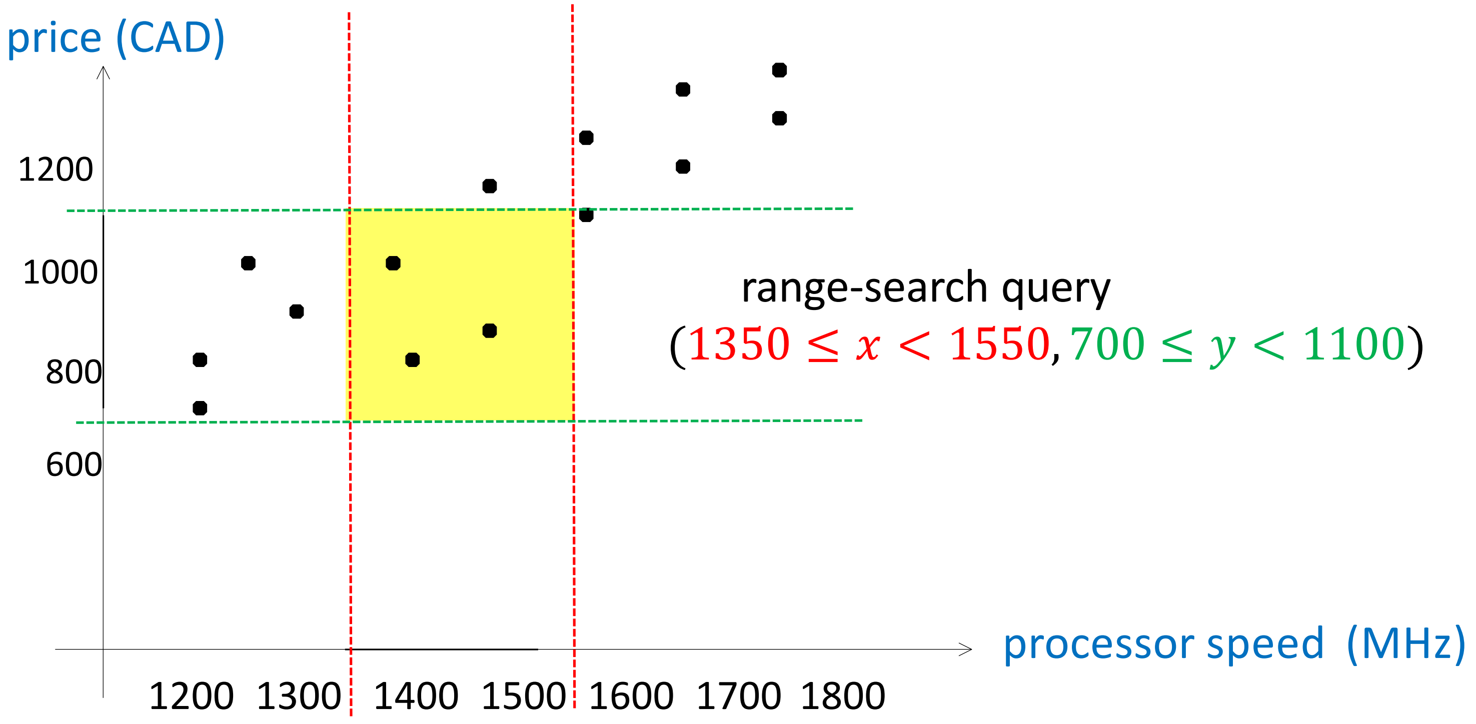
Multi-dimensional Data



- Data with multiple aspects of interest
  - laptops: price, screen size, processor speed, etc.
  - employees: name, age, salary, …
- Dictionary for multi-dimensional data
  - collection of d-dimensional items (or points)
  - each item has d aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
  - operations: insert, delete, search, range search
  - Range search
    - example: laptops with
      1) 11 inches < screen size < 13 inches
      2) 8GB < RAM < 16 GB
      3) 1,500 CAD < price < 2,000 CAD
- We focus on $d = 2$, i.e. points in Euclidean plane

Multi-Dimensional Range Search

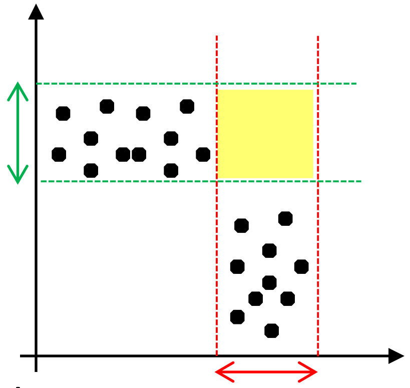
- (Orthogonal) d-dimensional range search
  - given a *query rectangle* A, find all points that lie within A



$(1350 \leq x < 1550, 700 \leq y < 1100)$

Multi-Dimensional Range Search

- Options for implementing d dimensional dictionaries
  - Reduce to one-dimensional dictionary
    - combine d-dimensional key into one dimensional key
      - i.e. $(x, y) \rightarrow x + y \cdot n^2$
    - problem: range search on one aspect is not straightforward
  - Use several dictionaries, one for each dimension
    - problem: wastes space, inefficient search
  - Example
    - $n/2$ points retrieved in each direction
    - $O(n \log n)$ time to consolidate results of both searches, with AVL trees
    - Total time is $O(n \log n)$, worse than exhaustive search
      - far from $O(s + \log n)$, especially since $s = 0$



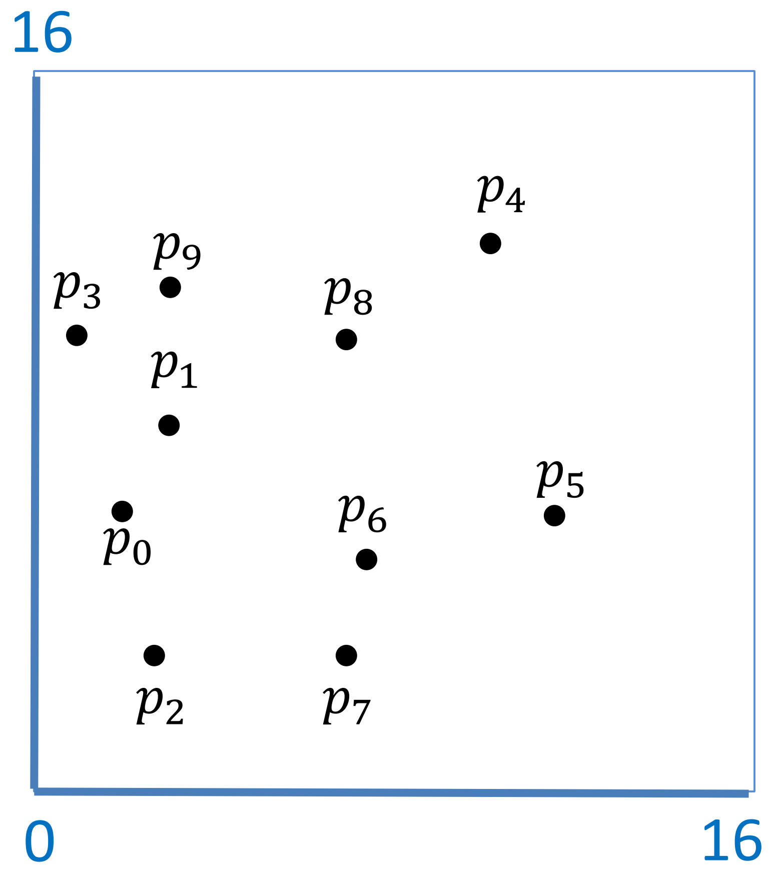
- Better idea
  - design new data structures specifically for points
  - will assume points are in *general position*: no two x-coordinates or y-coordinates are the same
    - simplifies presentation, data structures can be generalized

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
  - quadtrees, kd-trees
- Multi-dimensional range trees
  - organize dictionary points to support efficient nD search with a variant of BST search
  - both internal and leaf nodes store points, similar to one dimensional BST

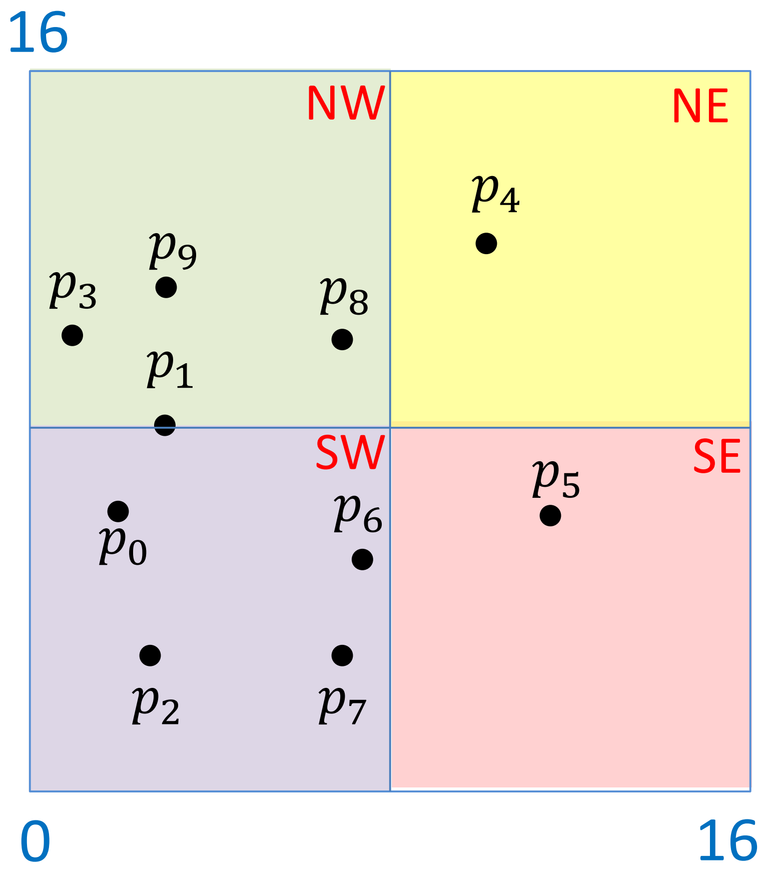
Outline

Quadtrees

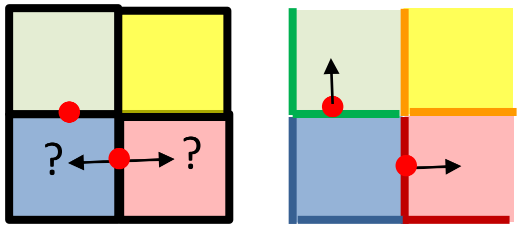


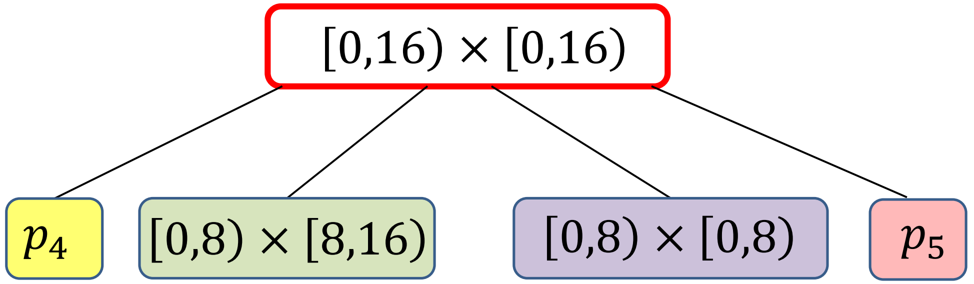
- Have a set S of n points in the plane
- Need *bounding box* R
    - width/height of R is a power of 2
    - smallest square $[0, 2^k) \times [0, 2^k)$ containing all points
    - this also simplifies insert/delete
    - find R by computing the maximum x and y values in set S
- Quadtree is a hierarchy (tree) of regions
- Higher levels responsible for larger regions
- Lower levels responsible for smaller regions
- Leaves responsible for regions small enough to store one point
- Whenever possible, search rules out regions at higher level of hierarchy, achieving efficiency

Quadtree Construction Example

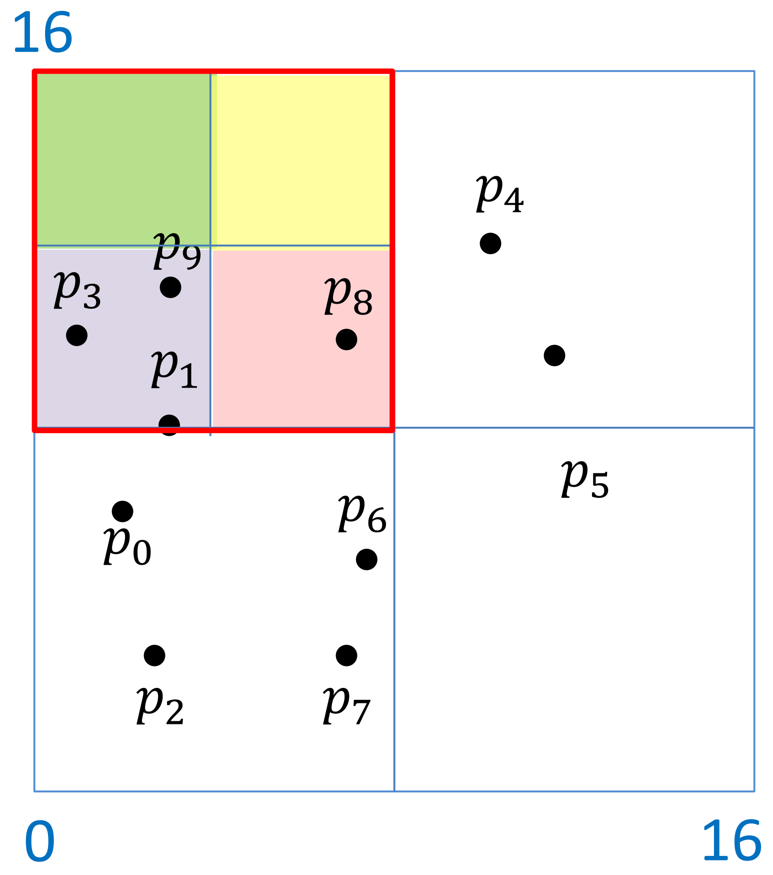


- The root corresponds to the whole square
- Split the square into 4 equal regions
- Convention: points on split lines belong to region on the right (or top)

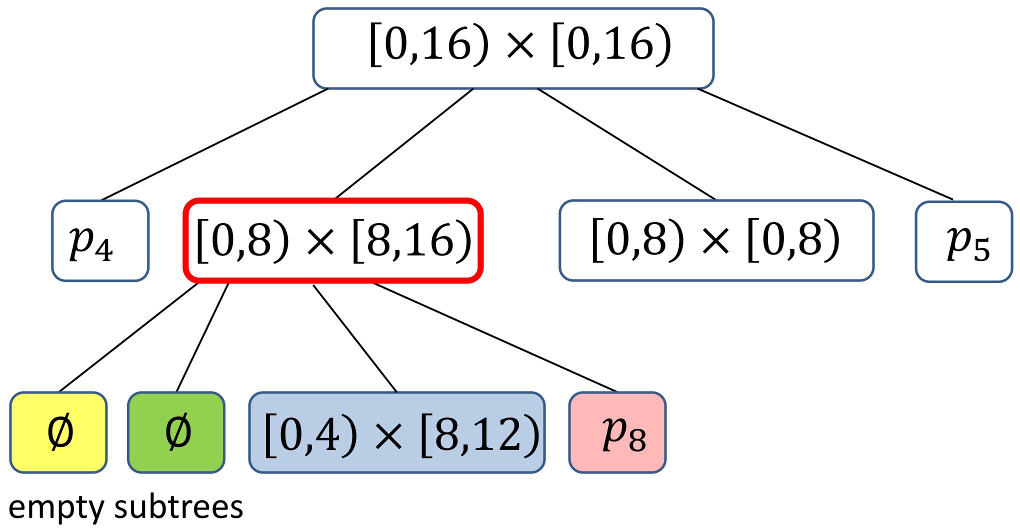




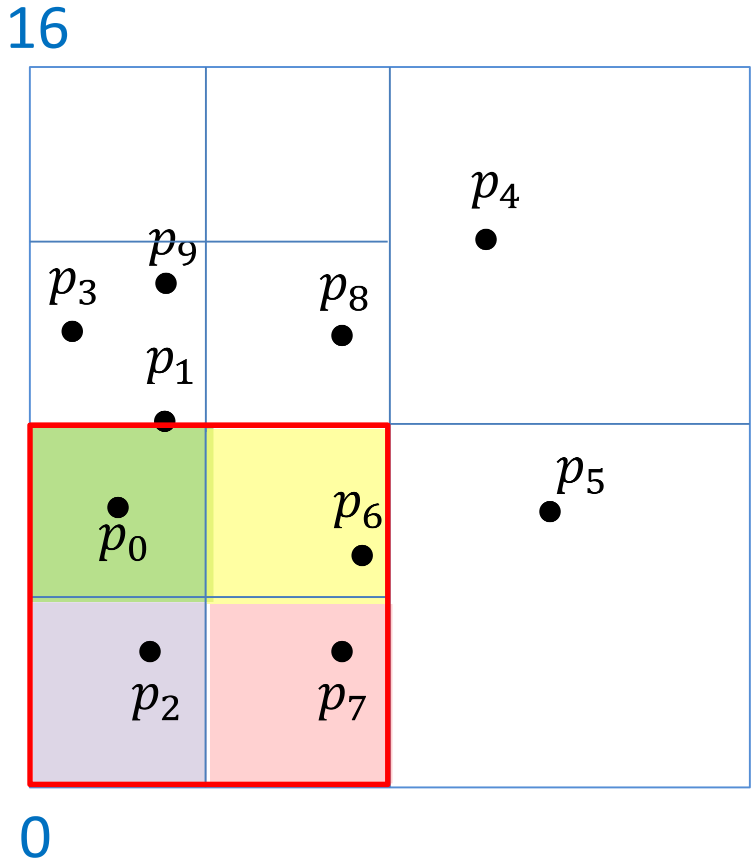
Quadtree Construction Example



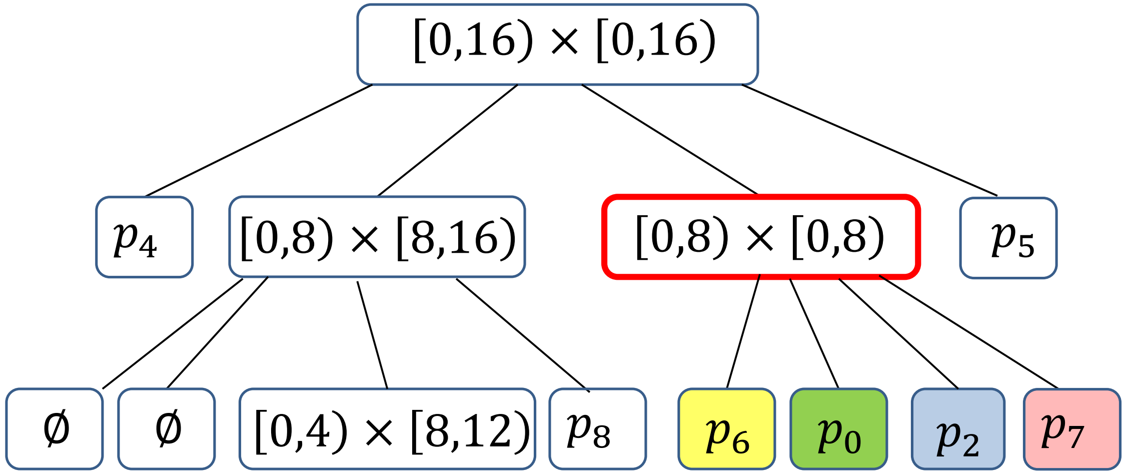
- keep subdividing regions (recursively) into smaller region until each region has one point



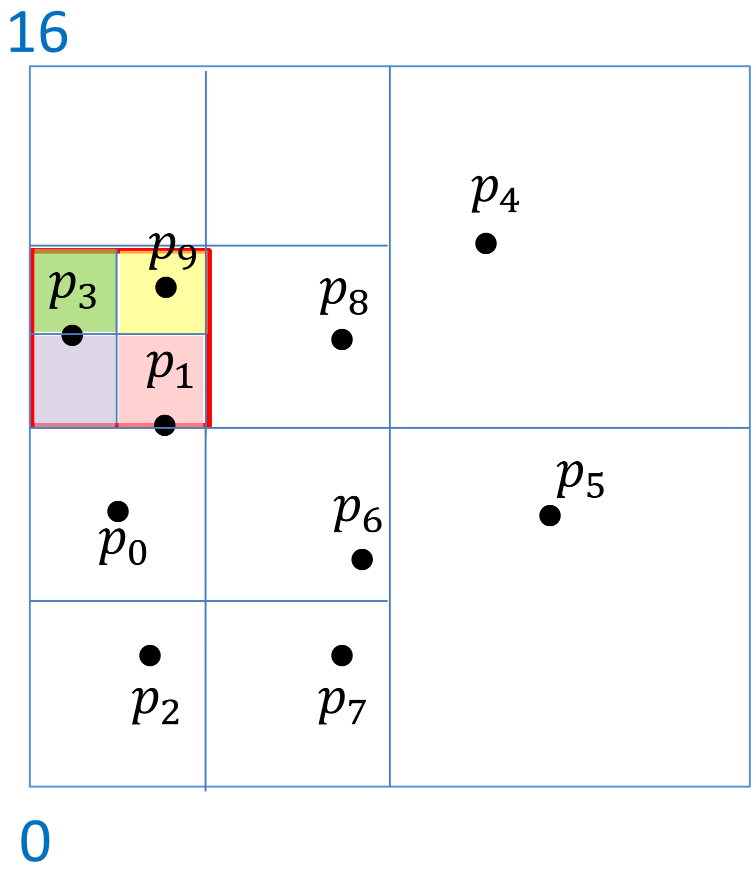
Quadtree Construction Example



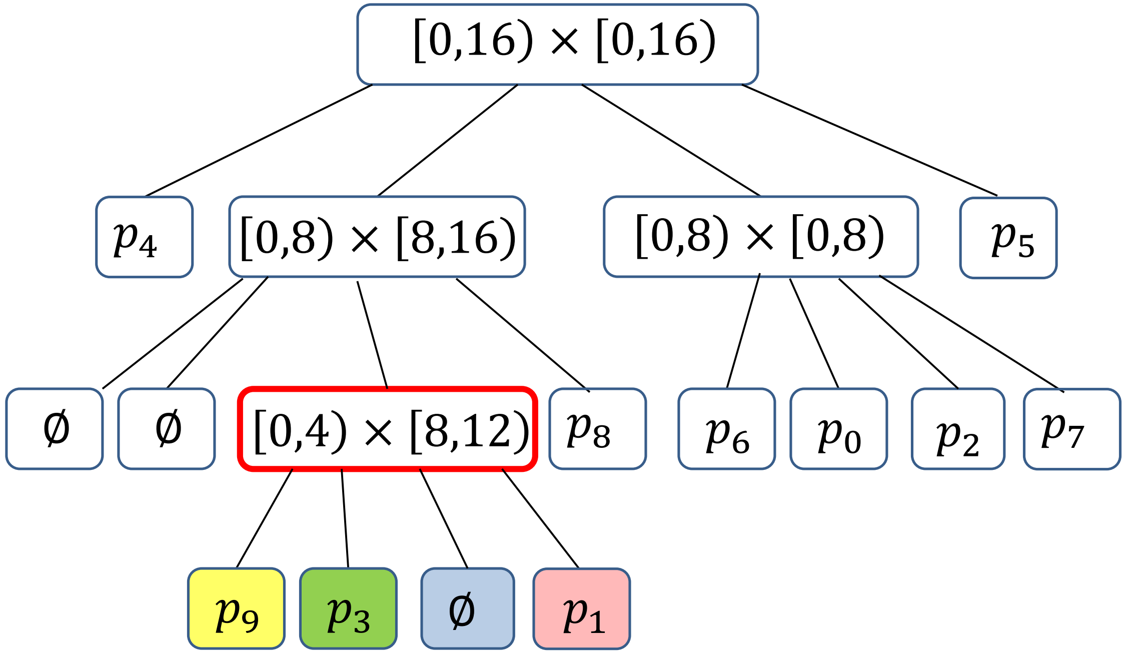
- keep subdividing regions (recursively) into smaller region until each region has one point



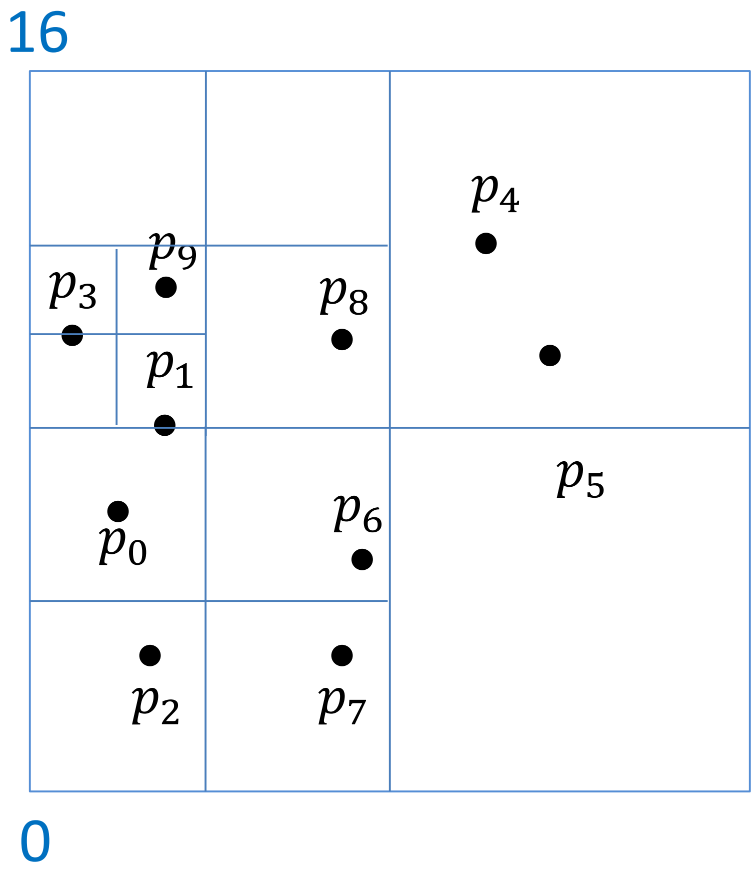
Quadtree Construction Example



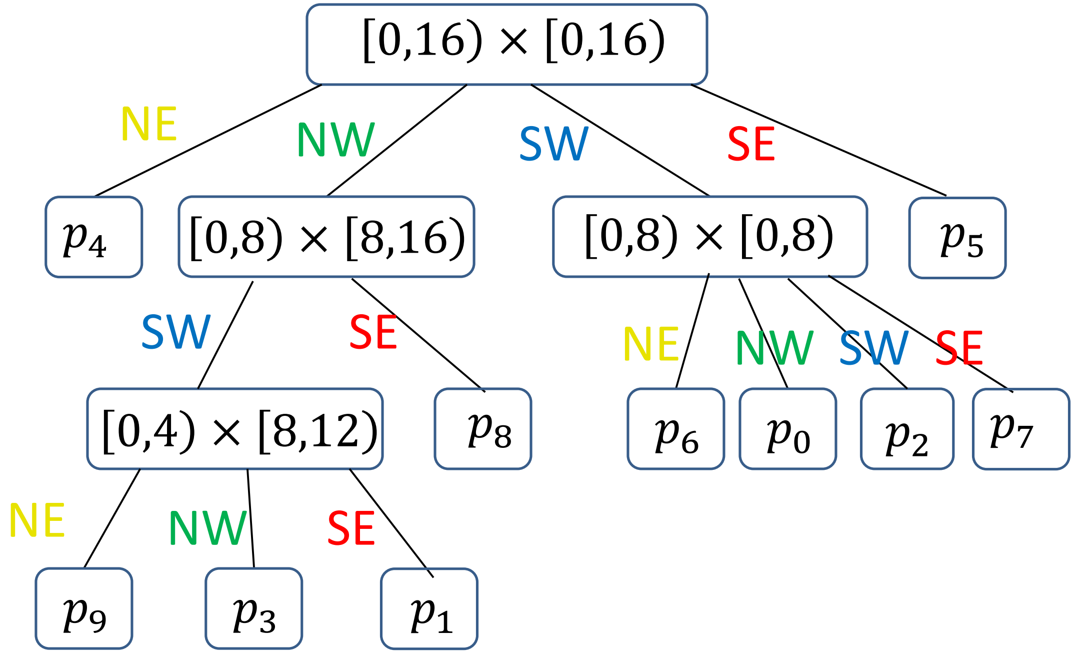
- keep subdividing regions (recursively) into smaller region until each region has one point



Quadtree: Omit Empty Subtrees



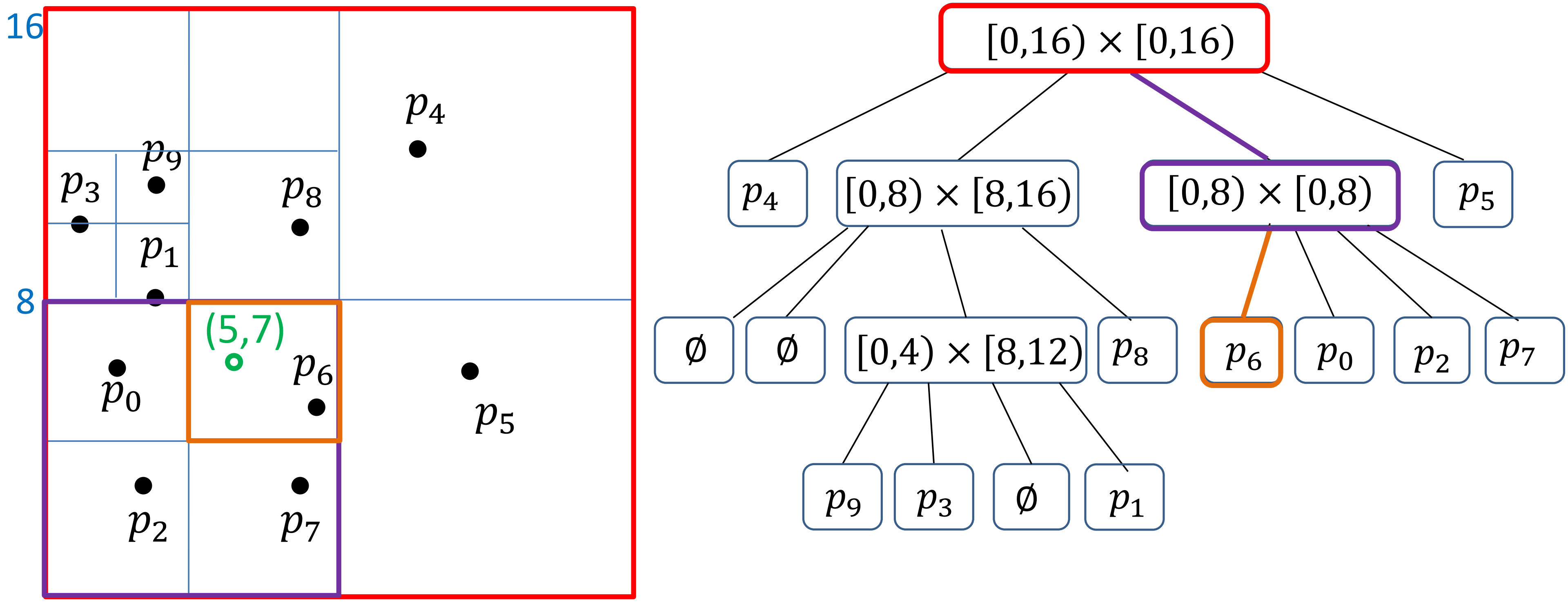
- Easier for humans
  - omit empty subtrees, label edges



Quadtree Building Summary

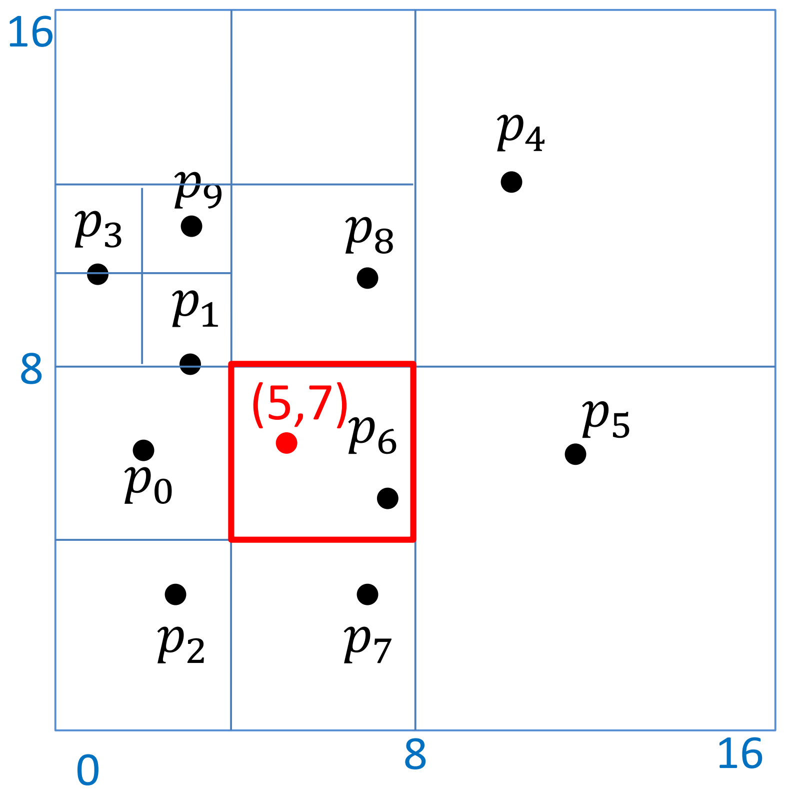
- Have n points $S = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$
    - all points are within a square R
- To build quadtree on S
    - root r corresponds to R
    - if R contains 0 (or 1) point
        - then root r is empty (or a leaf that stores 1 point)
        - else partition R into four equal subsquares (quadrants)
            - $R_{NE}, R_{NW}, R_{SW}, R_{SE}$
            - for each region, root has four subtrees $v_{NE}, v_{NW}, v_{SW}, v_{SE}$
    - recursively repeat this process at each nonleaf child
    - convention: points on split lines belong to region on the right (or top)

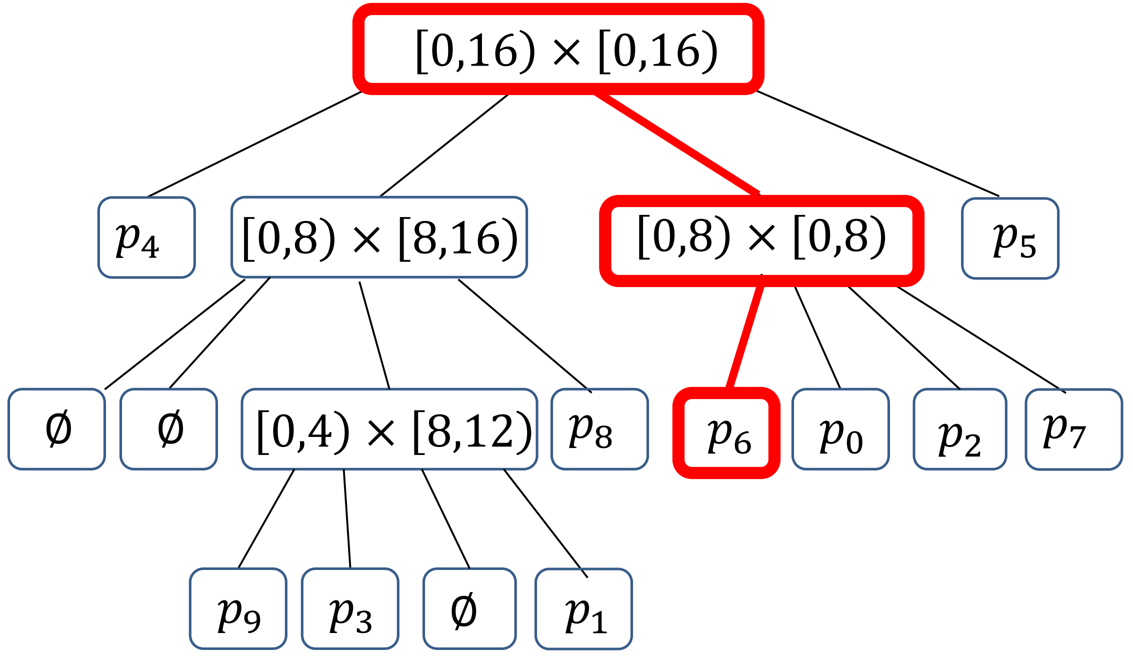
Quadtree Search



- 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 subtree (not found)
- Example: search(5,7) (not found)
- Search is efficient if quadtree has small height

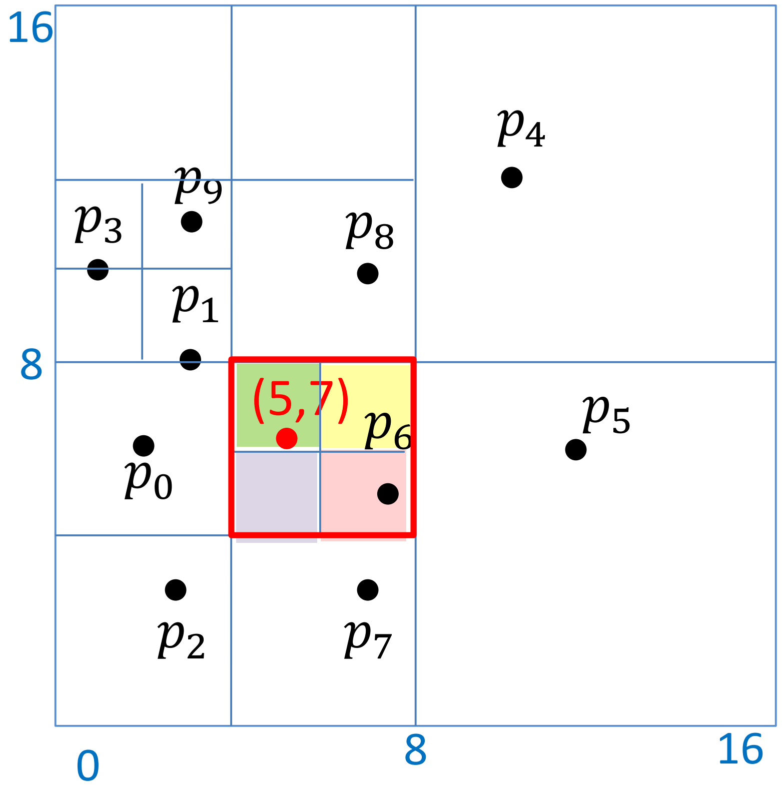
Quadtree Insert

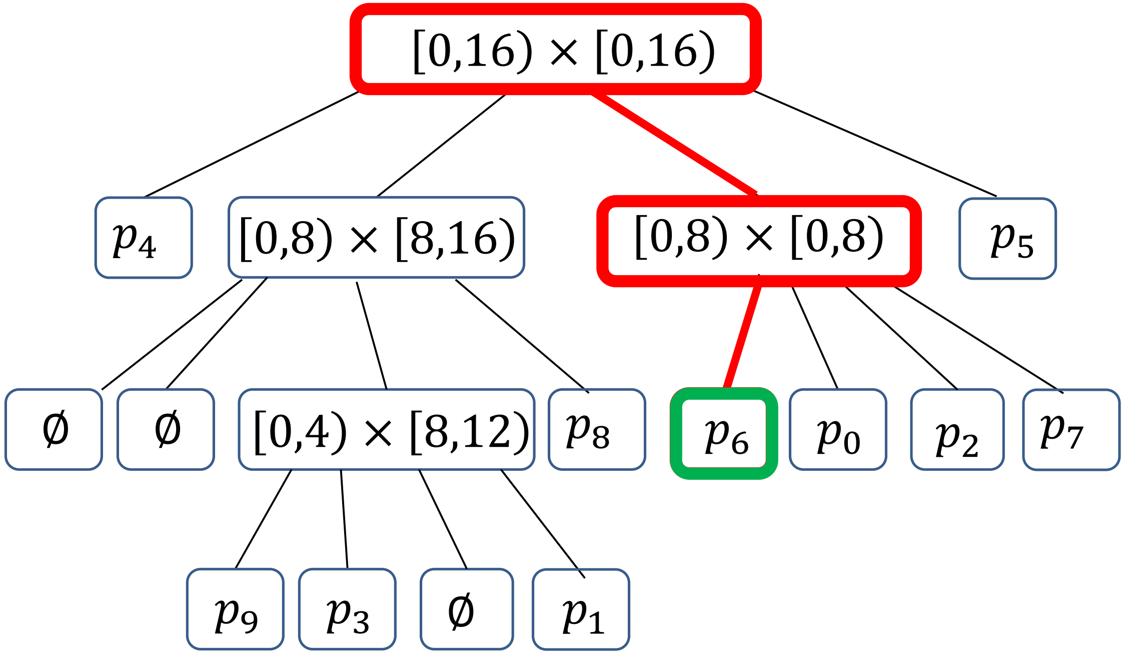




- First perform search
- Two cases
    1. search finds a leaf storing one point
        - example: insert(5,7)

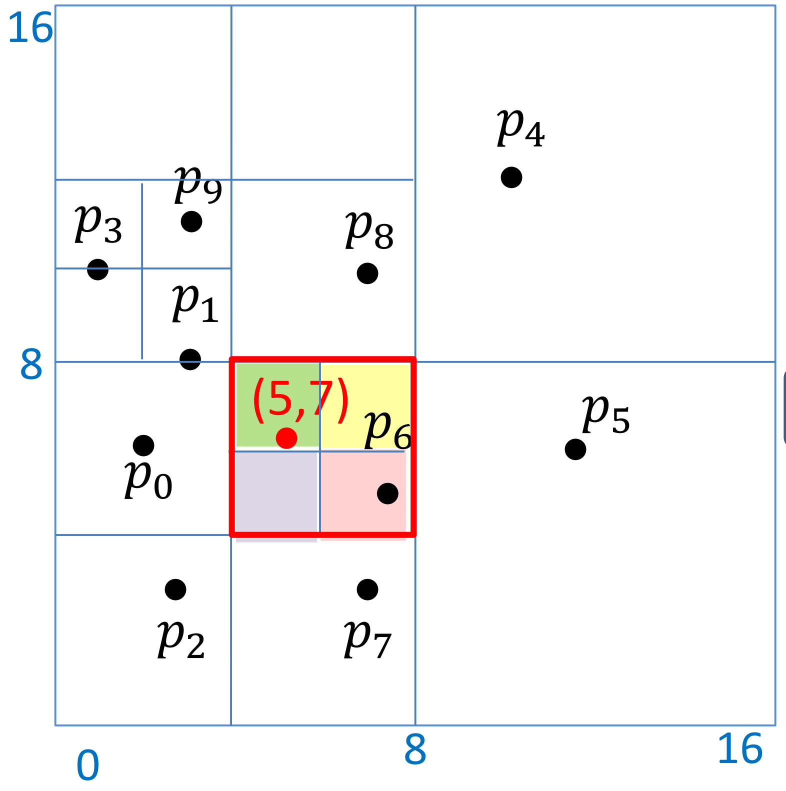
Quadtree Insert

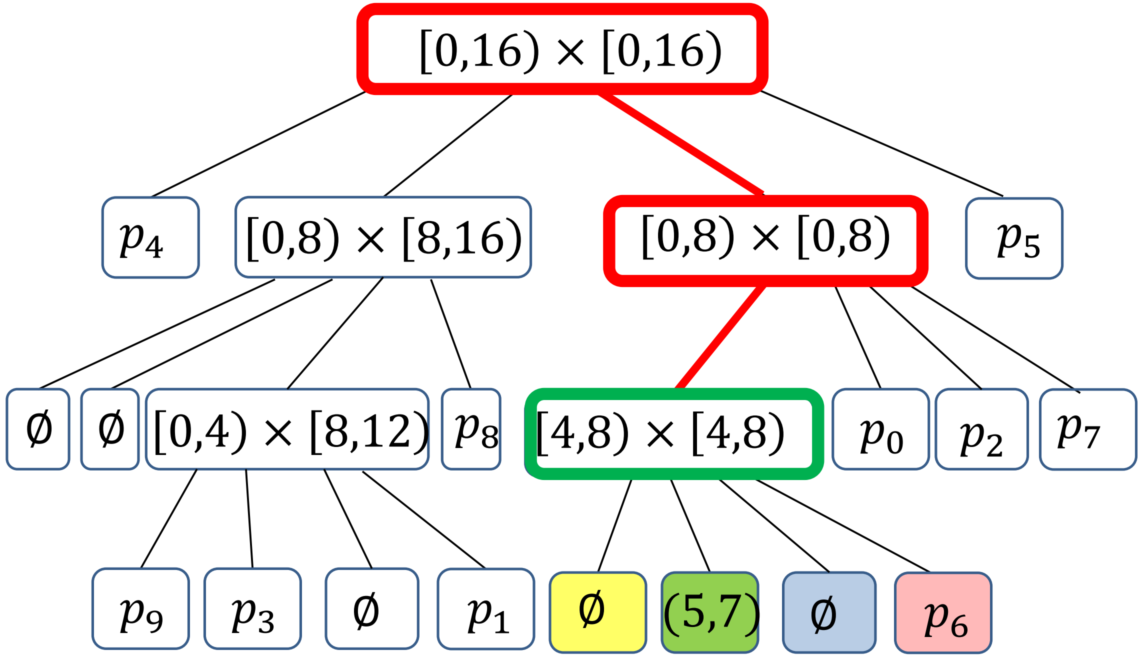




- First perform search
- Two cases
  1. search finds a leaf storing one point
     - example: insert(5,7)
     - repeatedly split the leaf **while** there are two points in one region

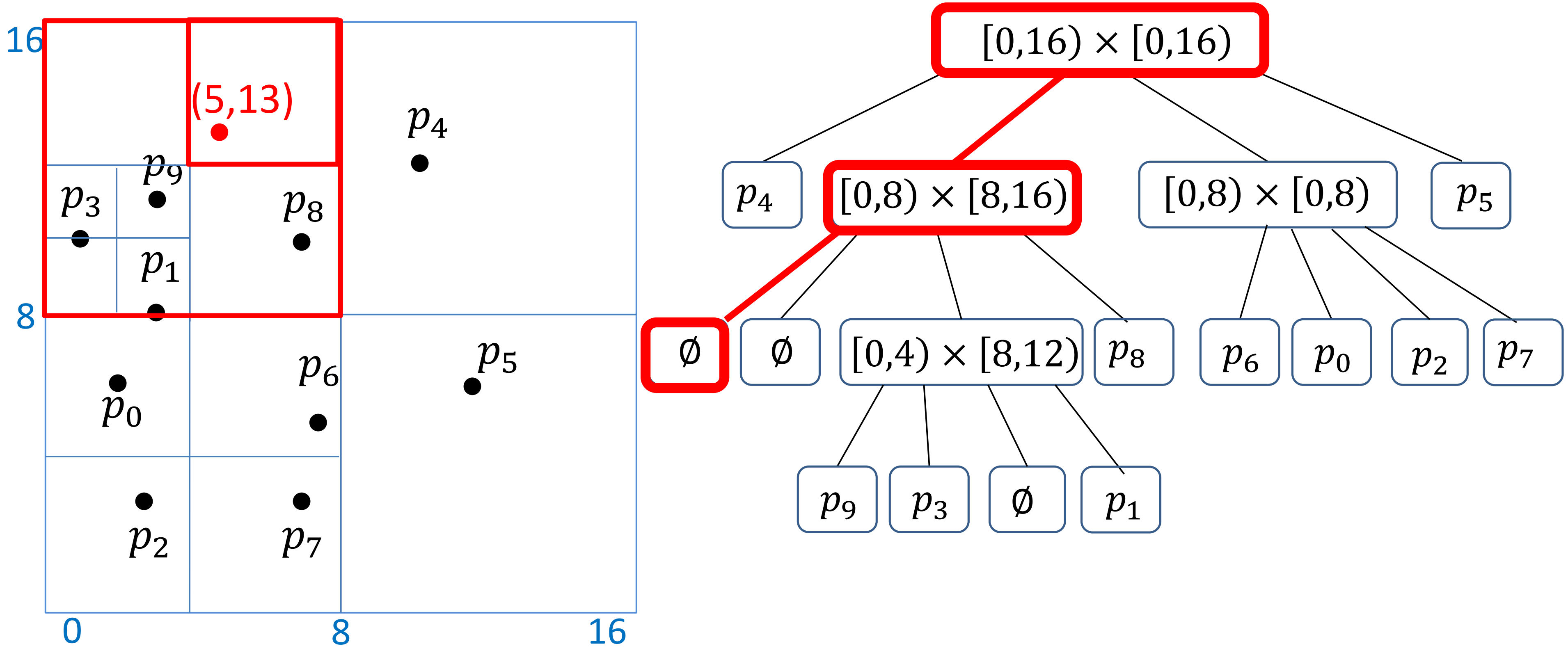
Quadtree Insert





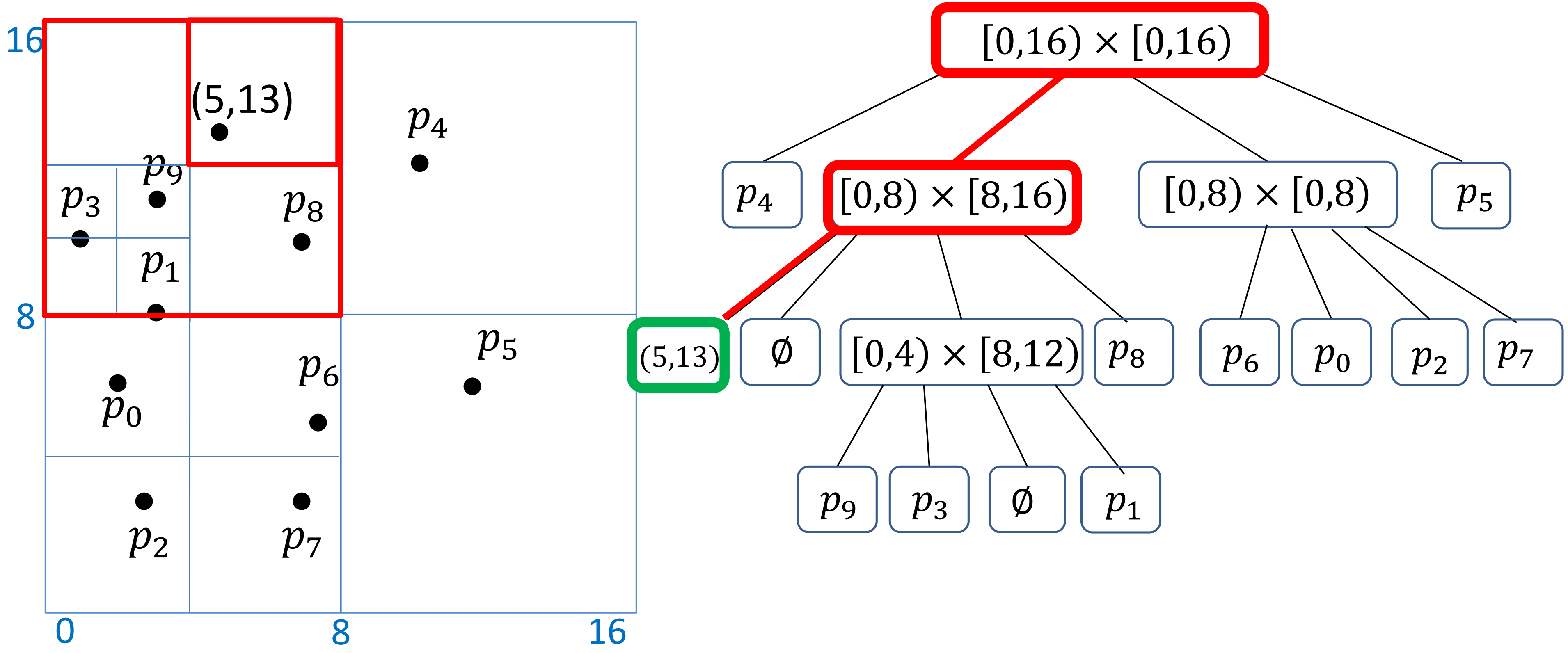
- First perform search
- Two cases
  1. search finds a leaf storing one point
     - example: insert(5,7)
     - repeatedly split the leaf **while** there are two points in one region

Quadtree Insert



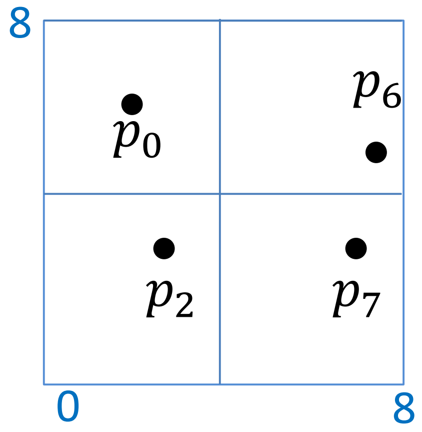
- First perform search
- Two cases
  1. search finds a leaf storing one point
  2. search finds an empty subtree
     - example: insert (5,13)

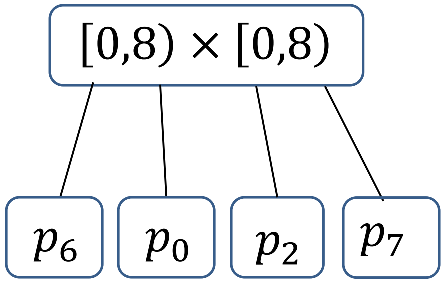
Quadtree Insert



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

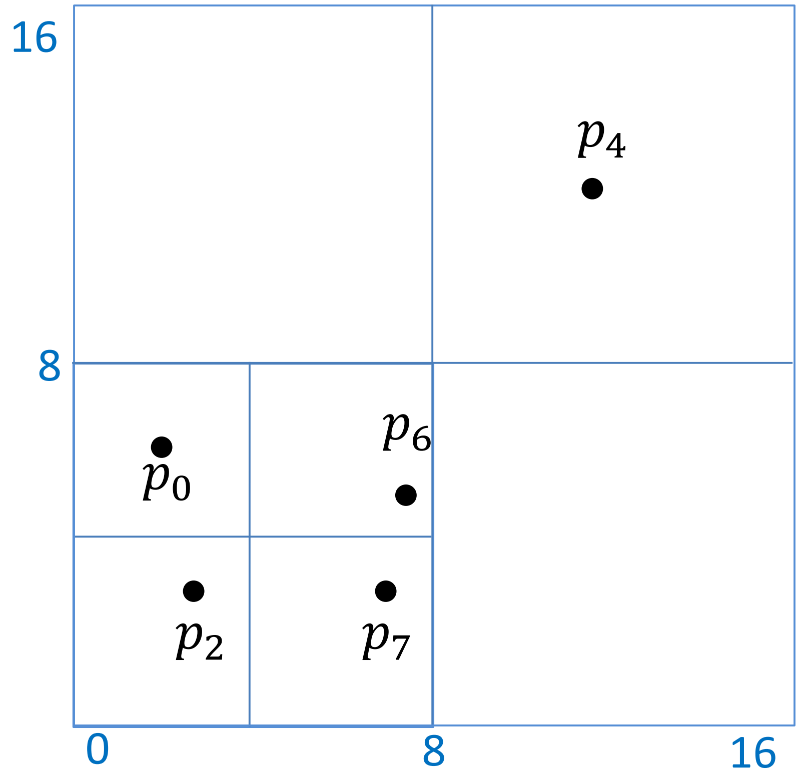
Quadtree Insert

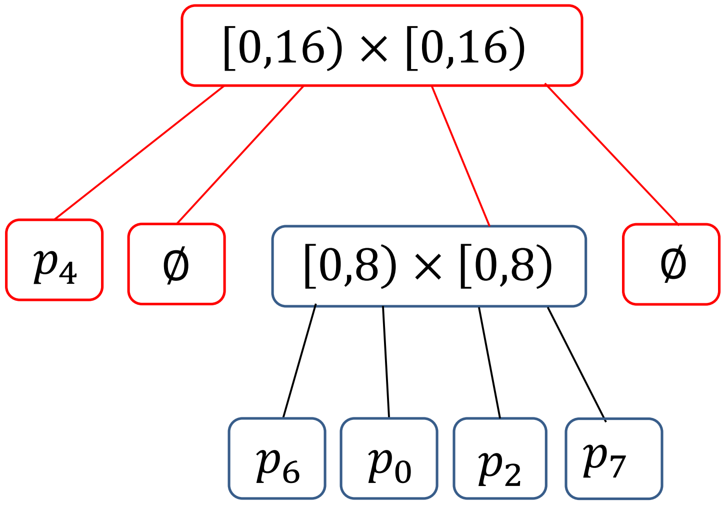




- If we insert point outside the bounding box, no need to rebuild the tree due to bounding box being $[0, 2^k) \times [0, 2^k)$

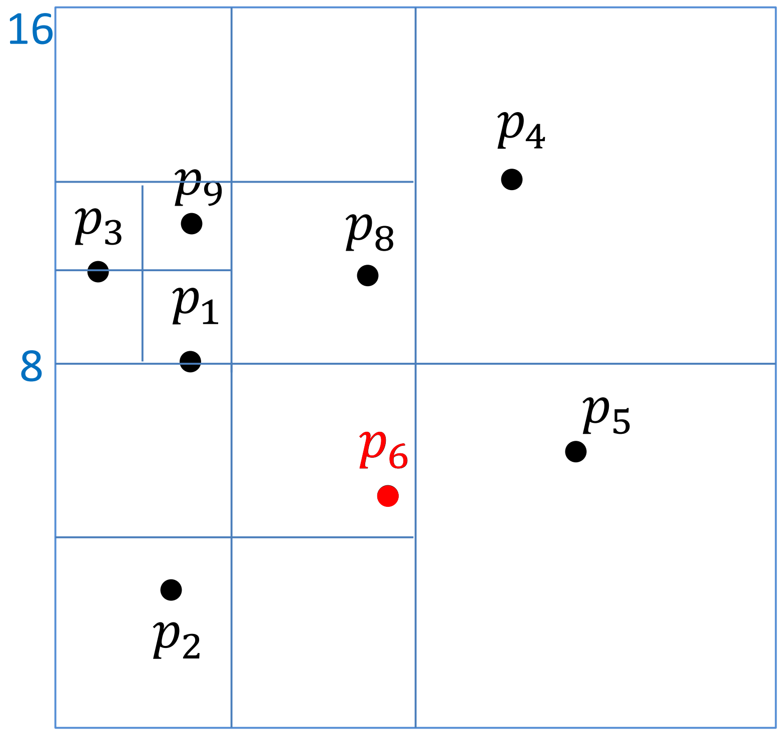
Quadtree Insert

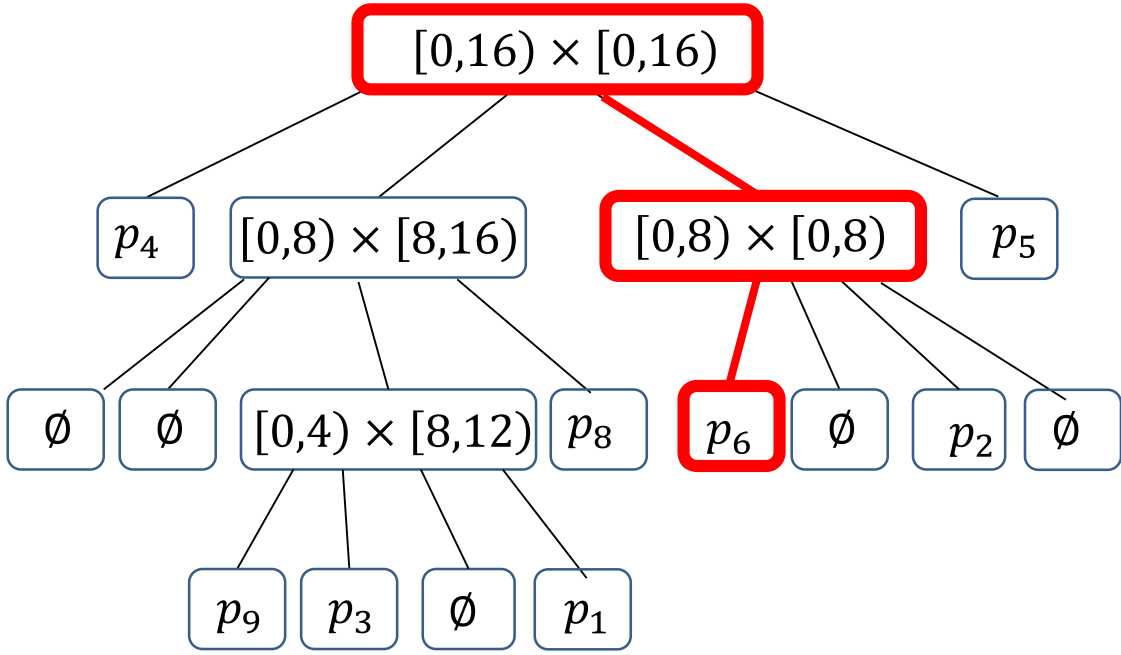




- If we insert point outside the bounding box, no need to rebuild the tree due to bounding box being $[0, 2^k) \times [0, 2^k)$

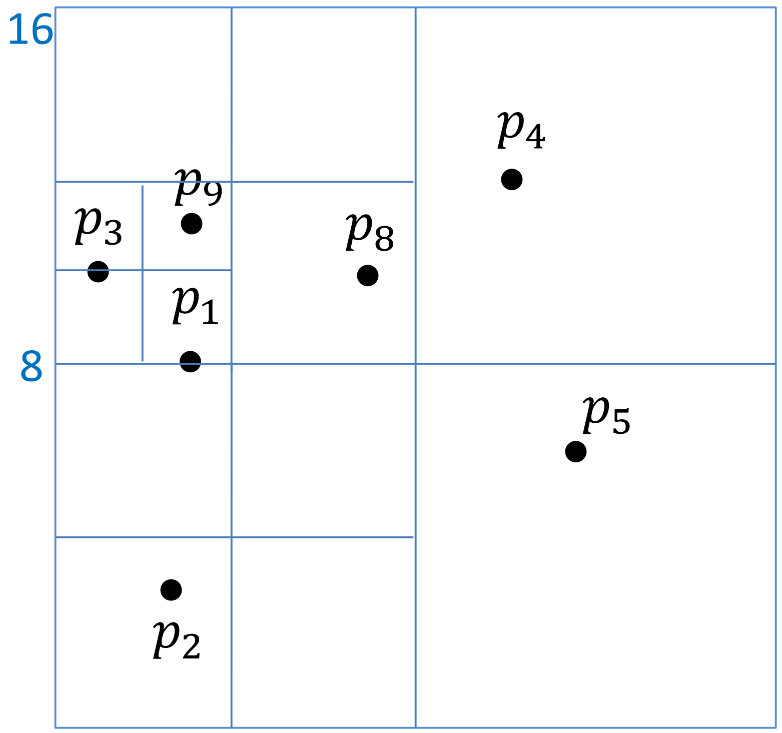
Quadtree Delete

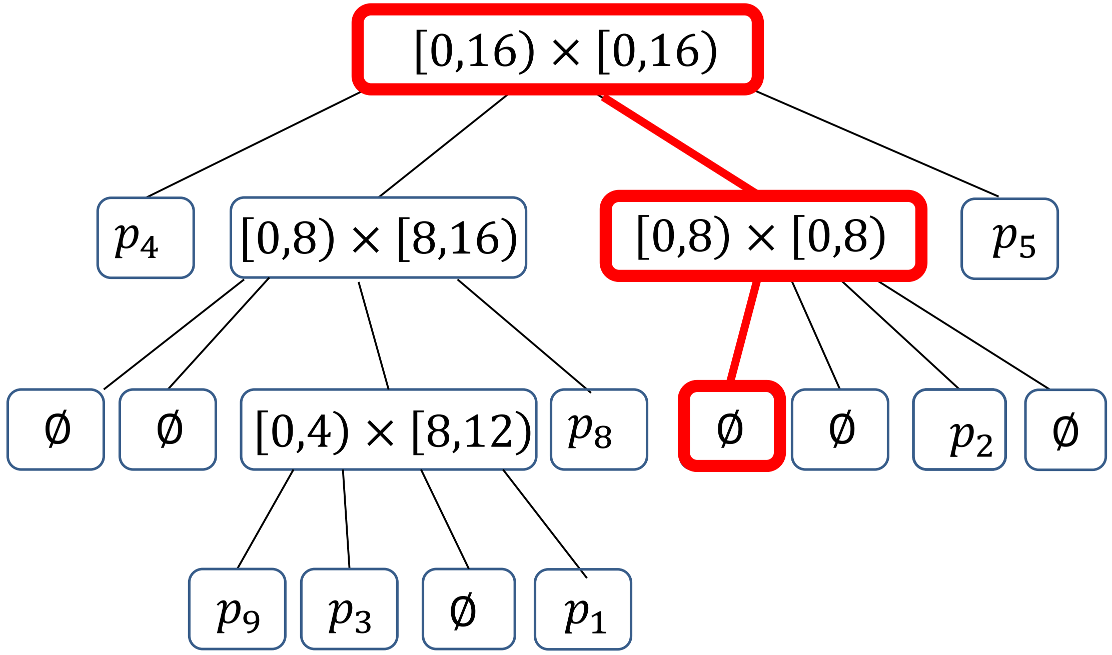




- search for leaf containing the point
    - example: delete(p_6)
- replace the leaf by empty subtree

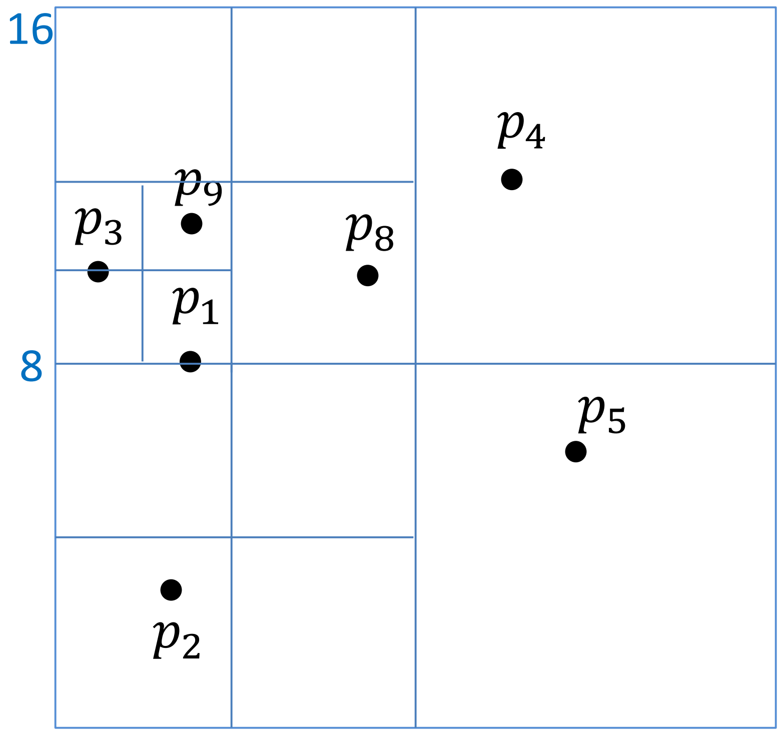
Quadtree Delete

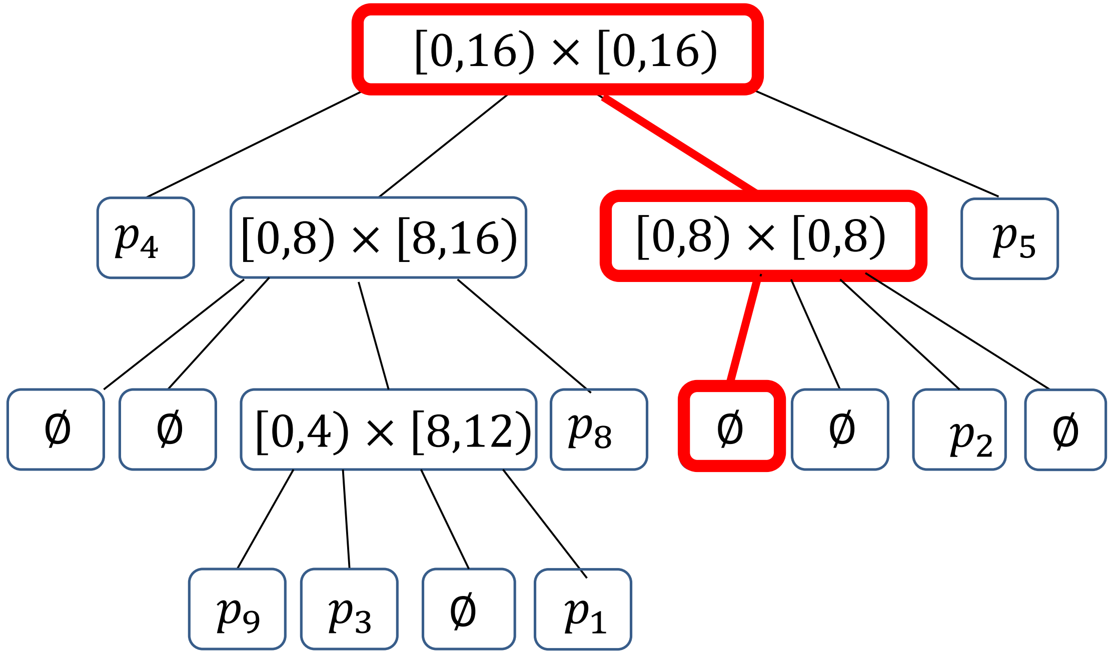




- search for leaf containing the point
    - example: delete(p_6)
- replace the leaf by empty subtree

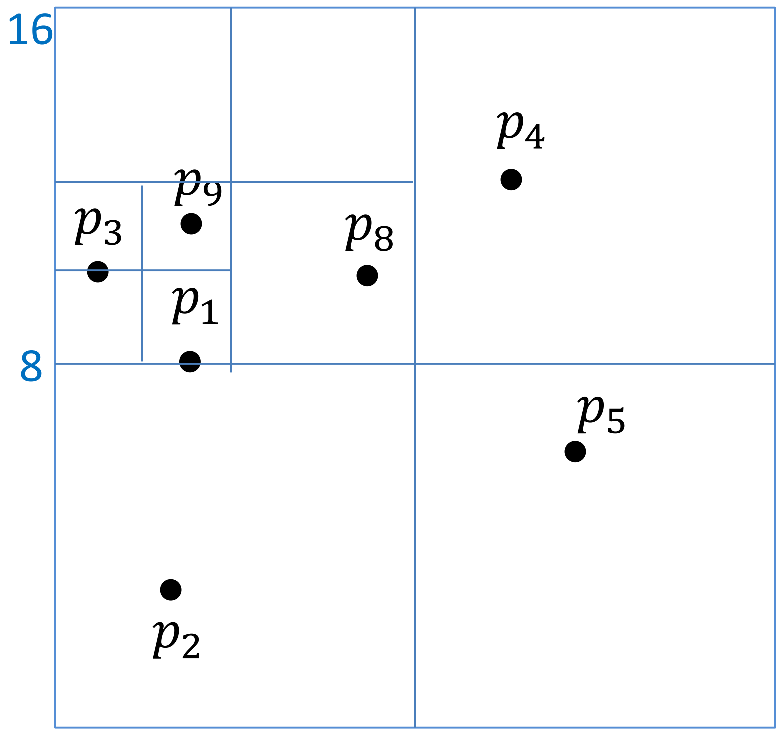
Quadtree Delete

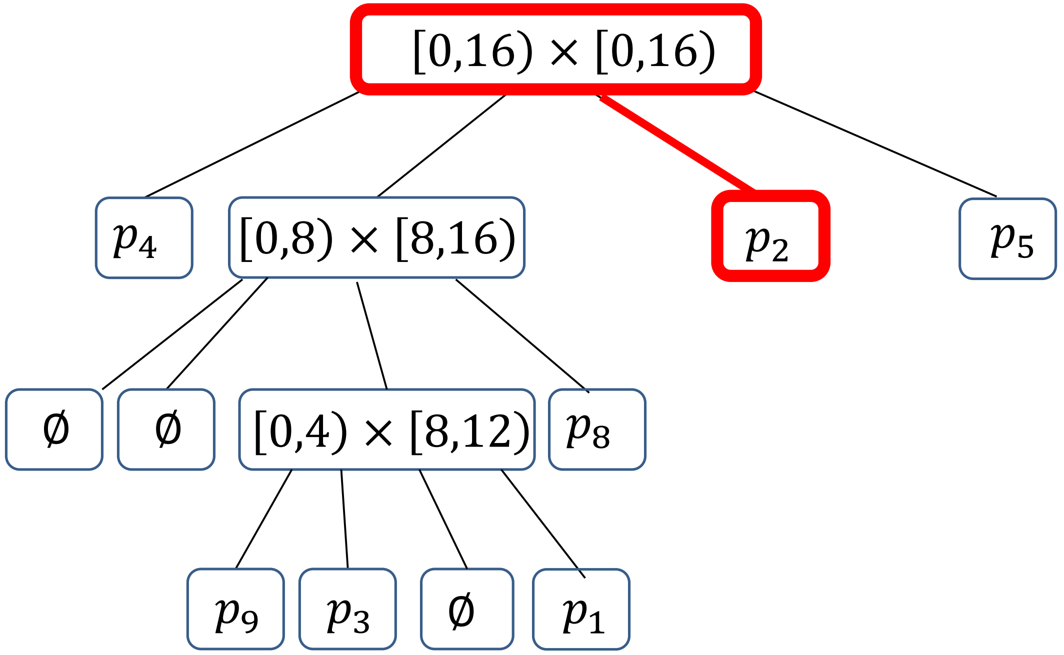




- search for leaf containing the point
    - example: delete(p_6)
- replace the leaf by empty subtree
- if parent has only one child, and the child is a *leaf*
    - delete parent, make the only leaf child to be a child of its grandparent
        - or quadtree root if no grandparent

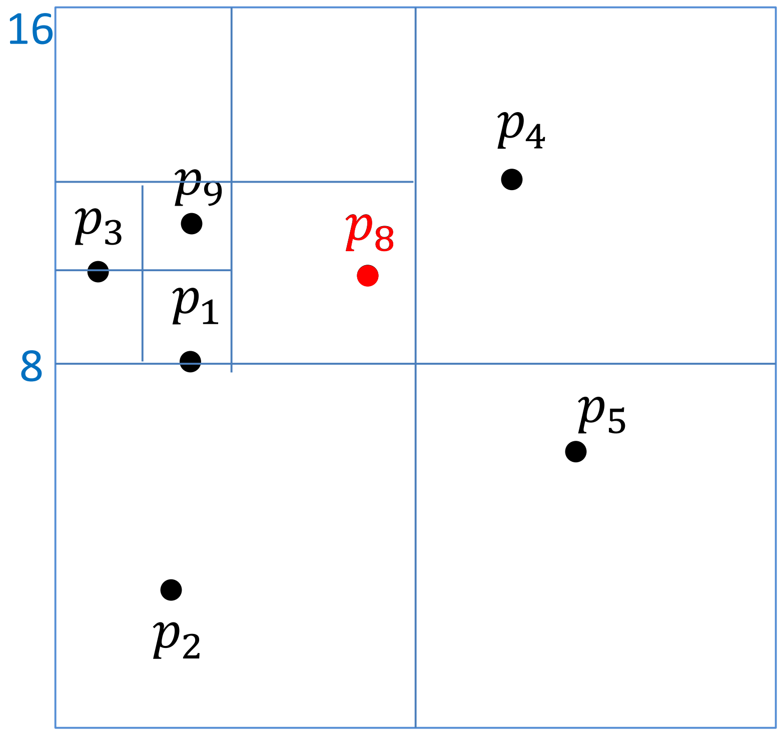
Quadtree Delete

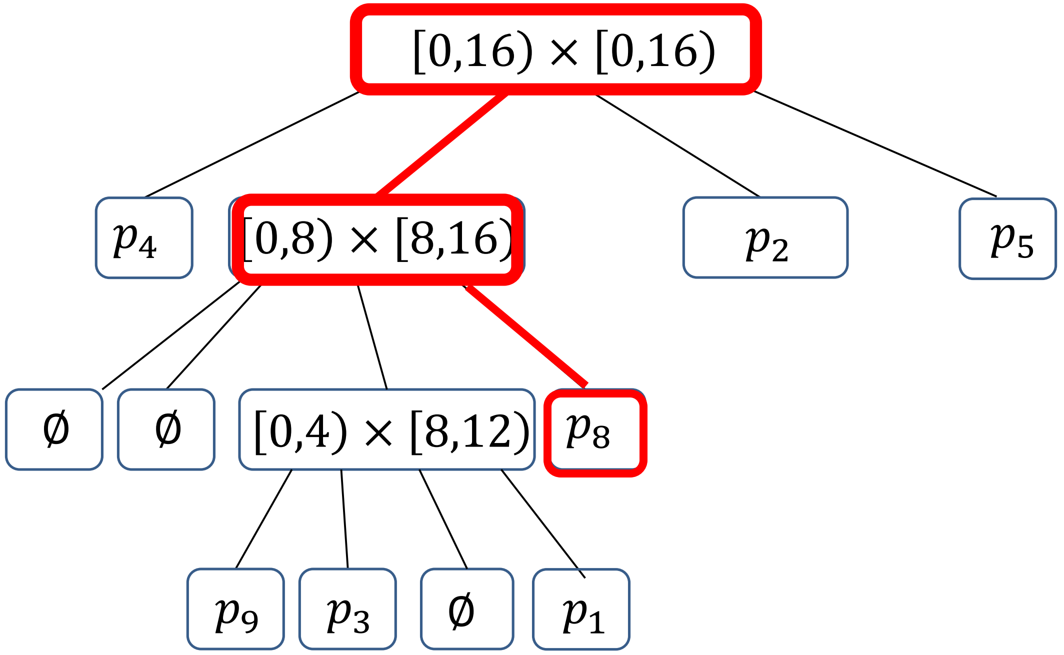




- search for leaf containing the point
    - example: delete(p_6)
- replace the leaf by empty subtree
- if parent has only one child, and the child is a *leaf*
    - delete parent, make the only leaf child to be a child of its grandparent
    - if deleted parent, may cause a series of deletes going up the tree

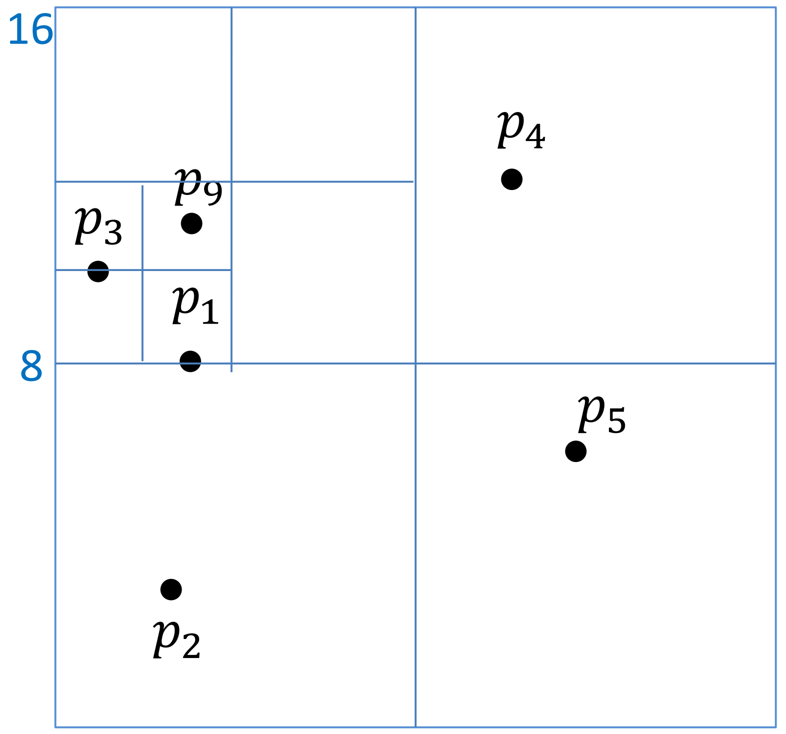
Quadtree Delete

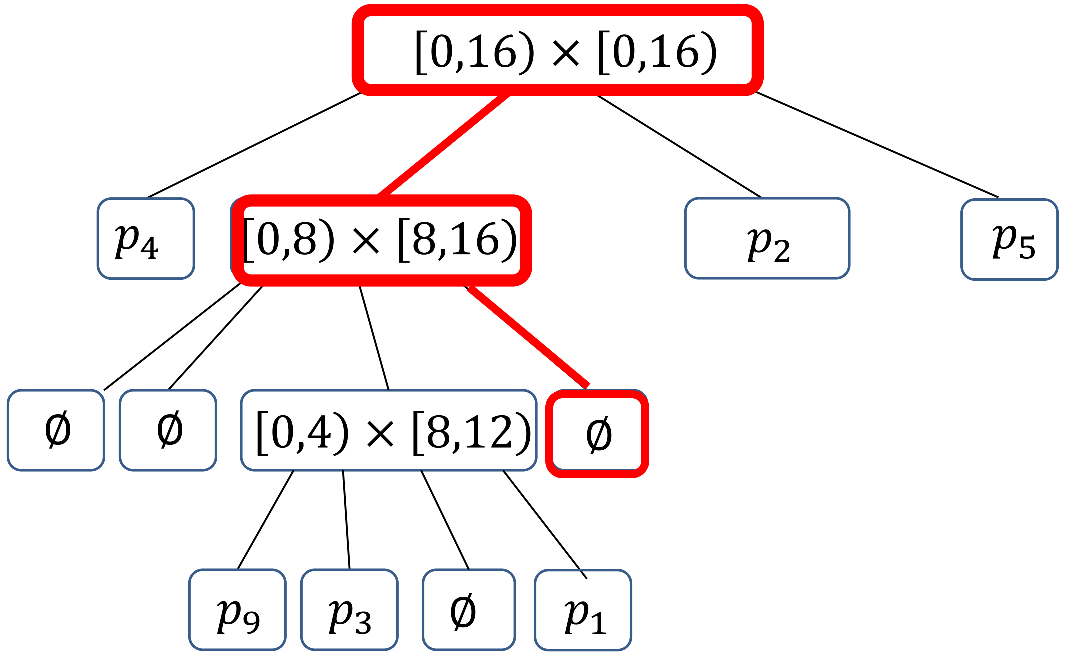




- Cannot delete if parent has a single child, but *non-leaf* child
    - example: delete(p_8)

Quadtree Delete

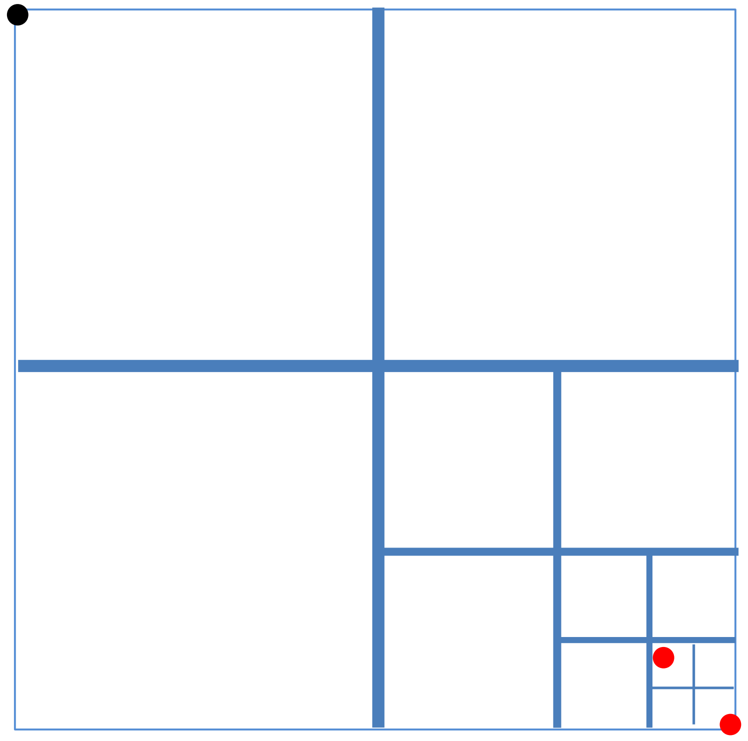




- Cannot delete if parent has a single child, but *non-leaf* child
    - non-leaf child corresponds to a region storing more than 1 point
    - example: delete(p_8)

Quadtree Analysis

$$\text{height} = 4$$



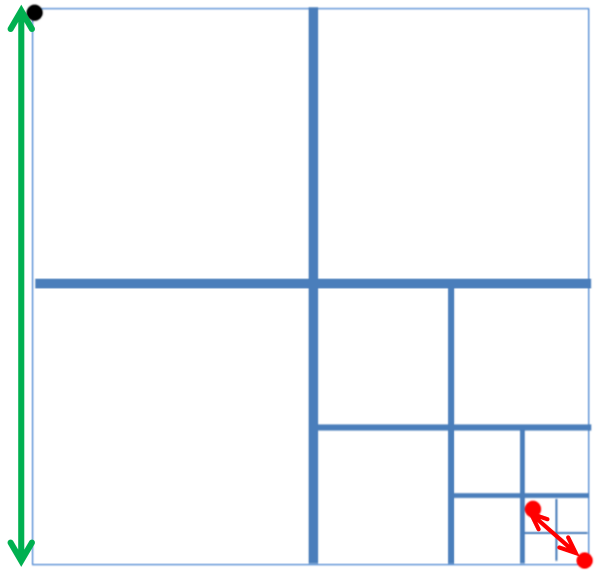
- Search, insert, delete depend on quadtree height
- What is the height of a quadtree?
    - can have very large height for bad distributions of points
    - example with just three points
    - can make height arbitrarily large by moving red points closer together

Quadtree Analysis

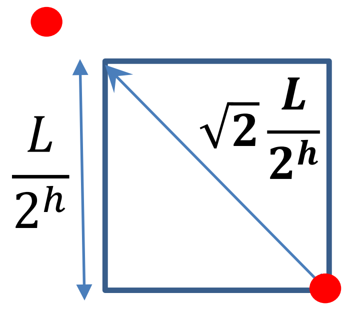
- spread factor of points S

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

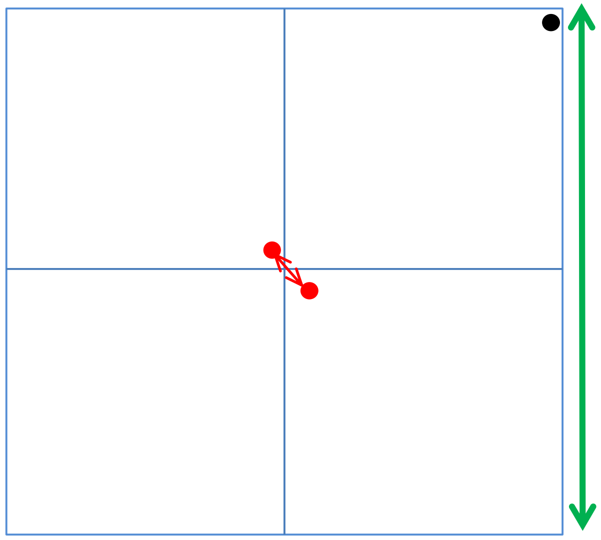
  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S



- Worst case: height $h \in \Omega(\log \beta(S))$
  - until smallest region diagonal is $< d_{min}$, 2 red points are in the 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: $h > \log\left(\sqrt{2}\frac{L}{d_{min}}\right) = \log(\sqrt{2}\beta(S))$



Quadtree Analysis



- spread factor of points S

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

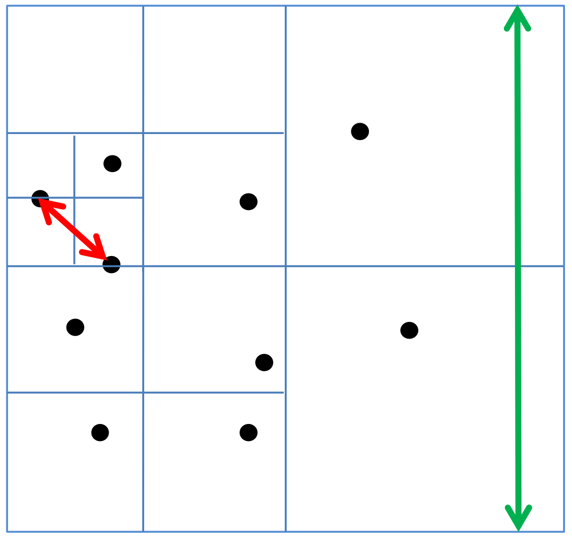
  - $L =$ side length of R
  - d_{min} is smallest distance between two points in S
- In the worst case, height $h \in \Omega(\log \beta(S))$
- However, height can be much better even if the spread is large

Quadtree Analysis

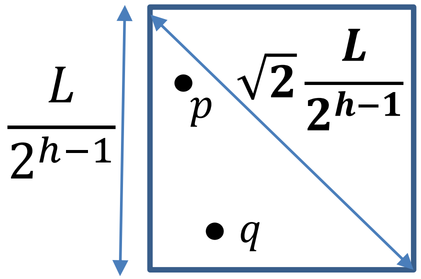
- spread factor of points S

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

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



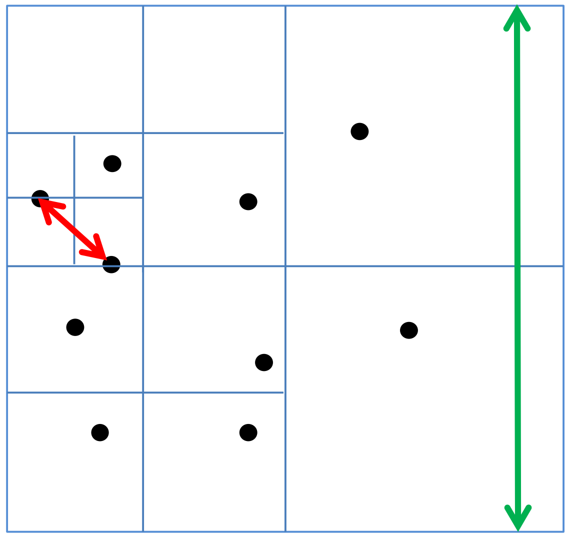
- In the worst case, height $h \in \Omega(\log \beta(S))$
- In **any case**, height $h \in O(\log \beta(S))$
  - let v be an internal node at depth $h-1$
    - there are at lest 2 points p, q inside its region
      - $d_{min} \le d(p,q)$
    - the corresponding region has side length $\frac{L}{2^{h-1}}$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



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

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

Quadtree Analysis



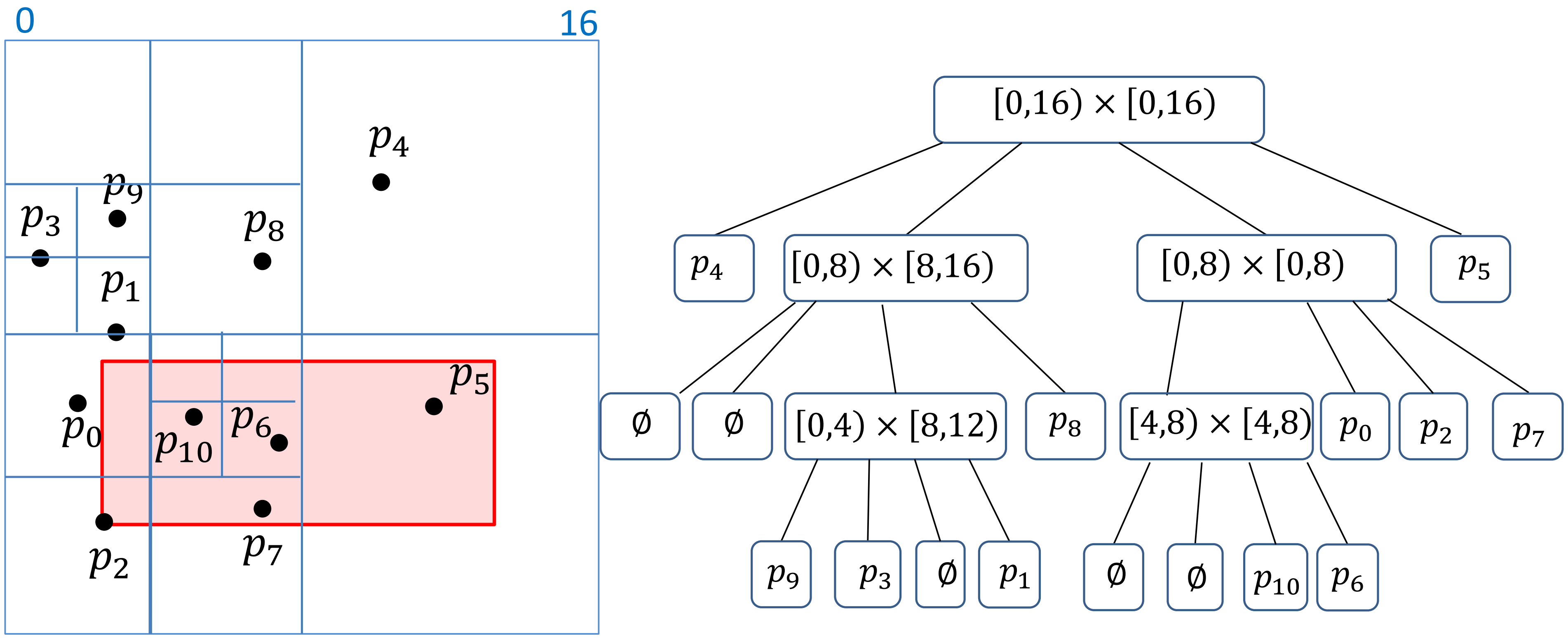
- spread factor of points S

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

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

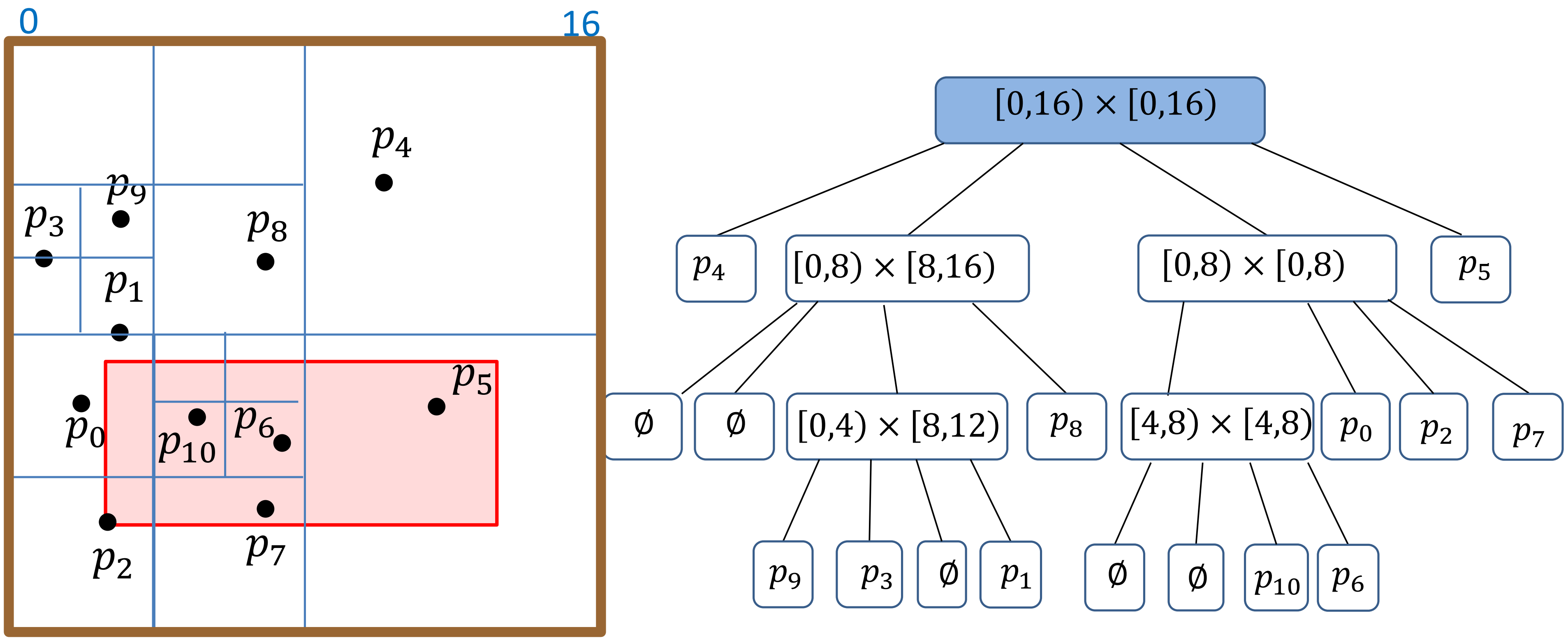
- In the worst case, height $h \in \Omega(\log \beta(S))$
- In any case, height $h \in O(\log \beta(S))$
  - to guarantee good performance, $\log \beta(S)$ should be much smaller than n
- Complexity to build initial tree: $\Theta(nh)$ worst-case
  - expensive if large height (as compared to the number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

Quadtree Range Search Example



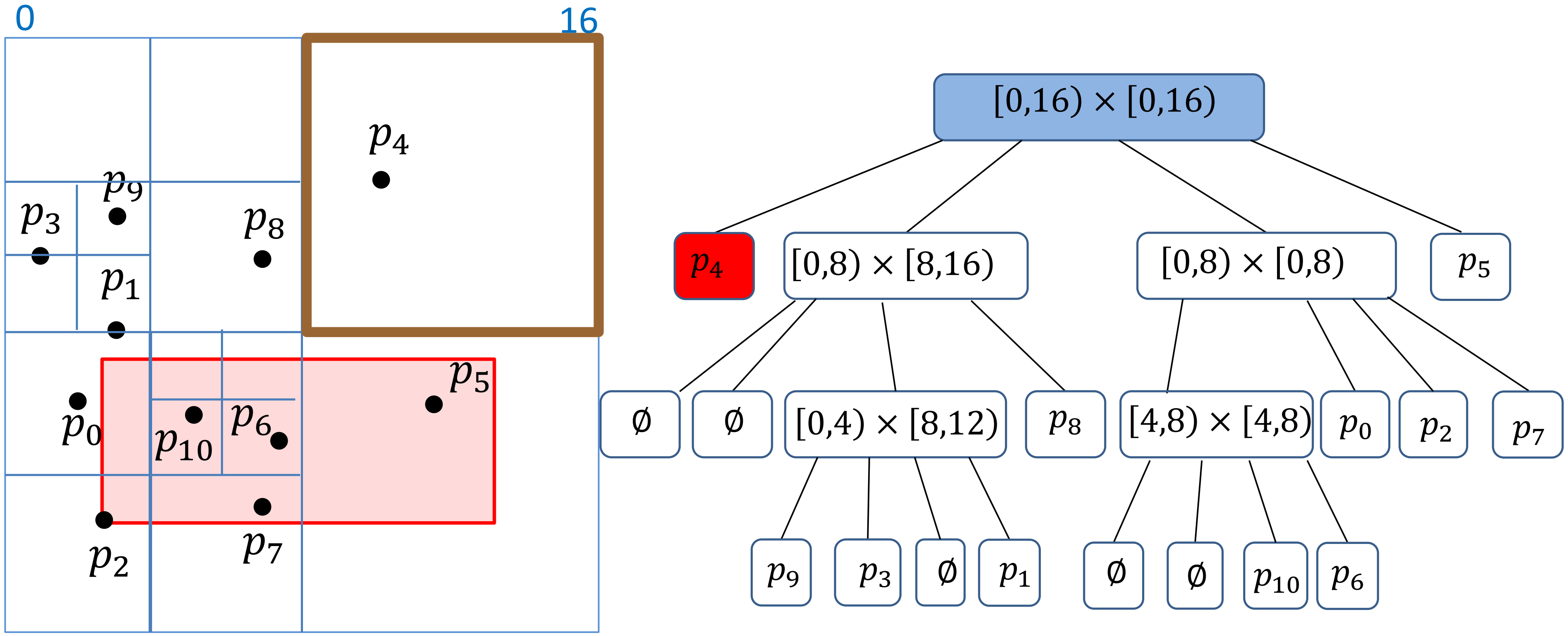
- Query rectangle A $= [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



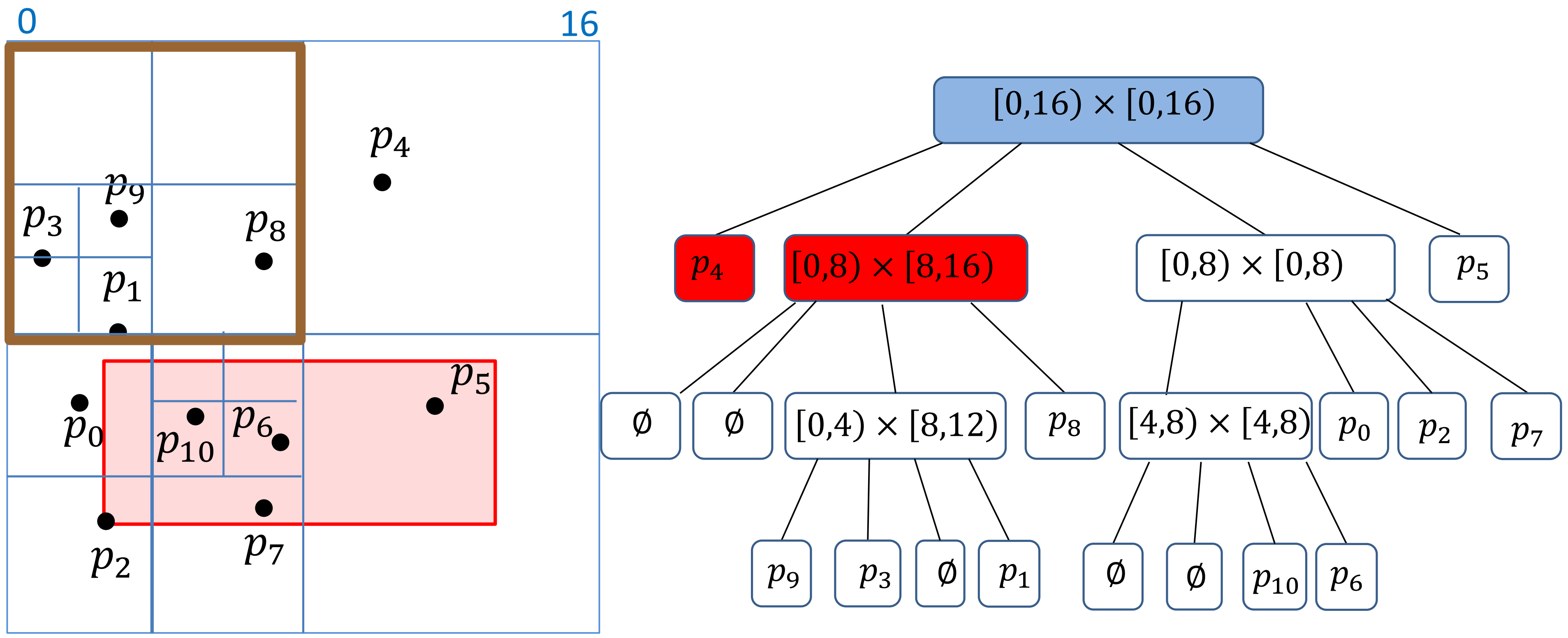
- Query rectangle $A = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



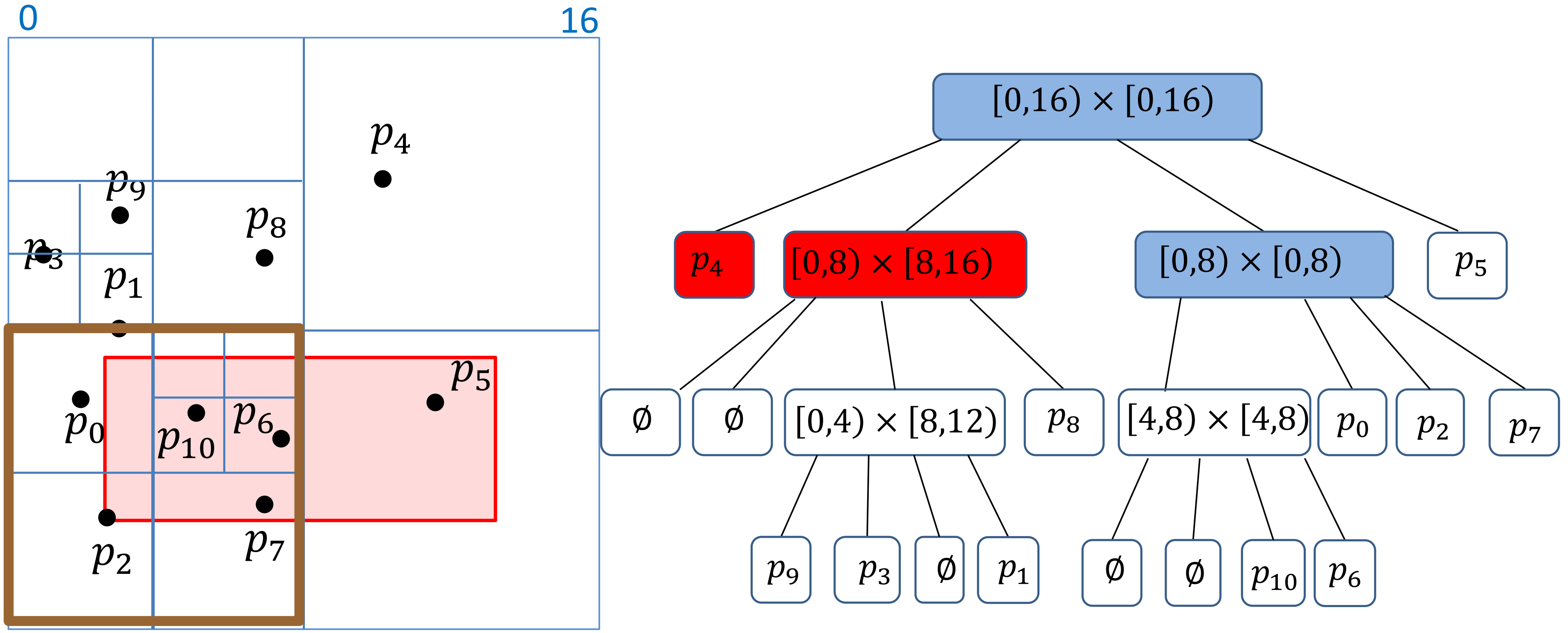
- Query rectangle A $= [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



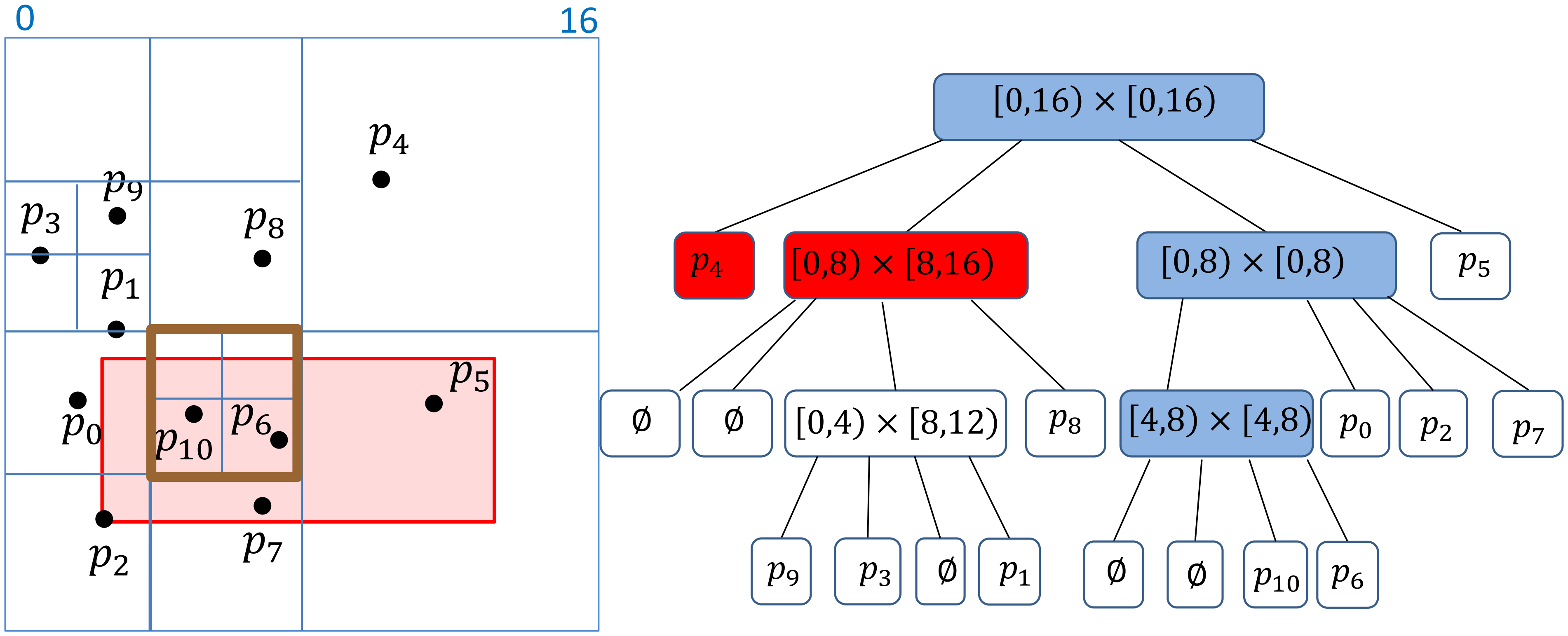
- Query rectangle $A = [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



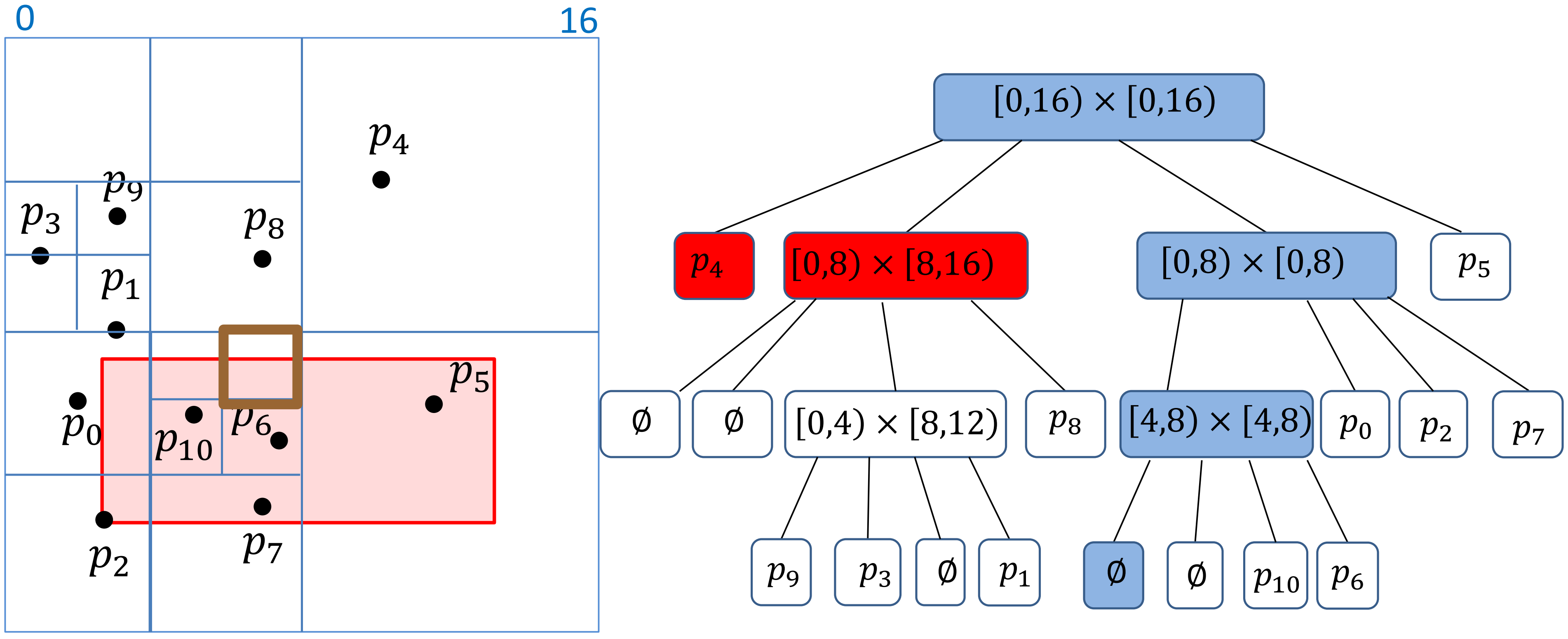
- Query rectangle A $= [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



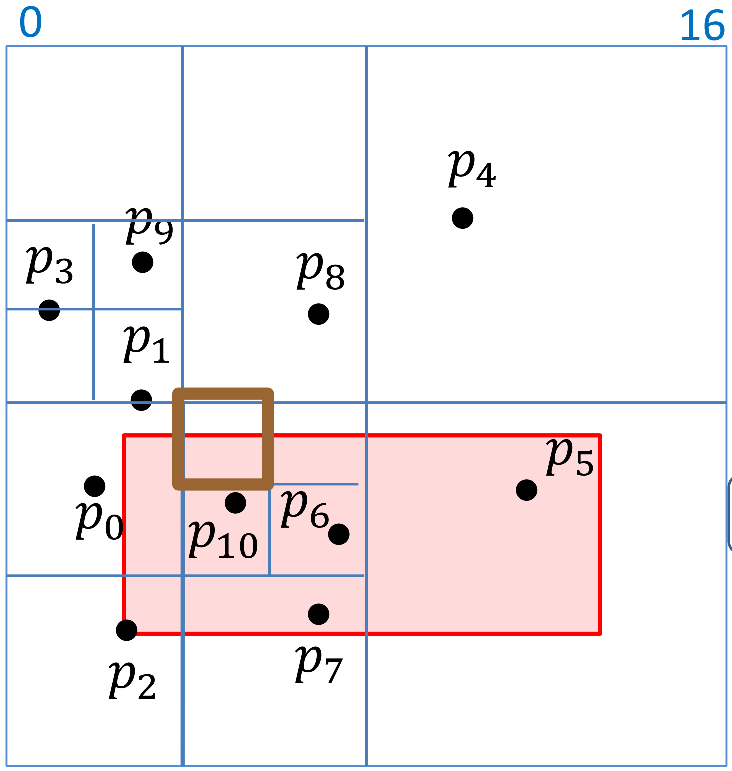
- Query rectangle A $= [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

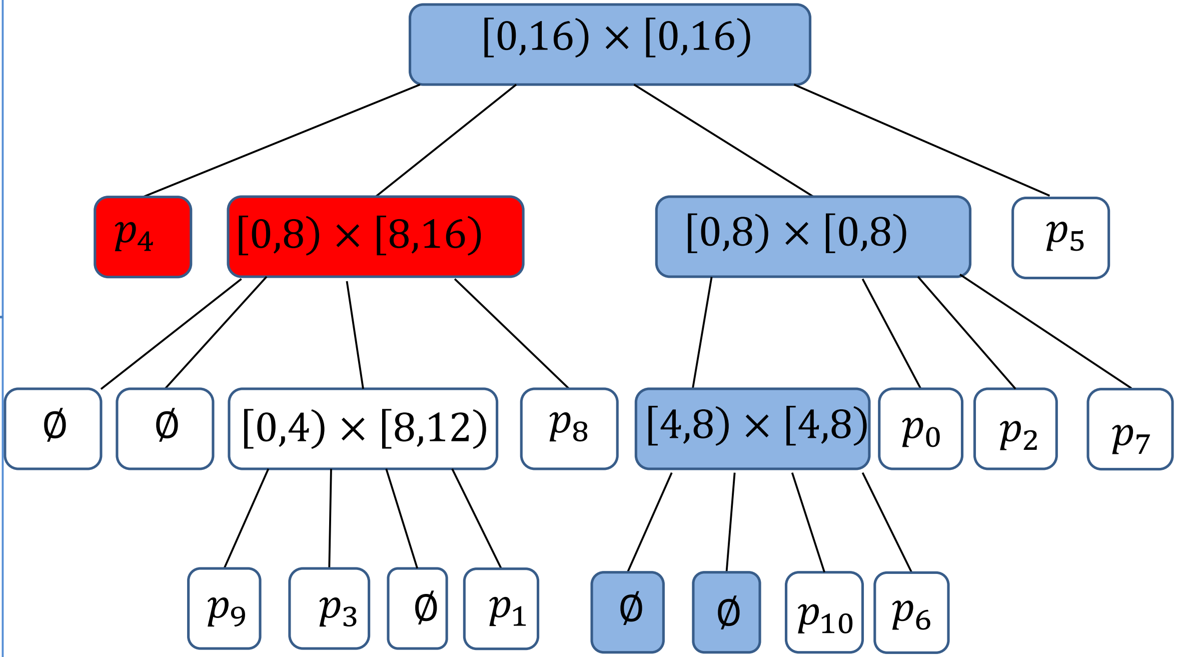
Quadtree Range Search Example



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

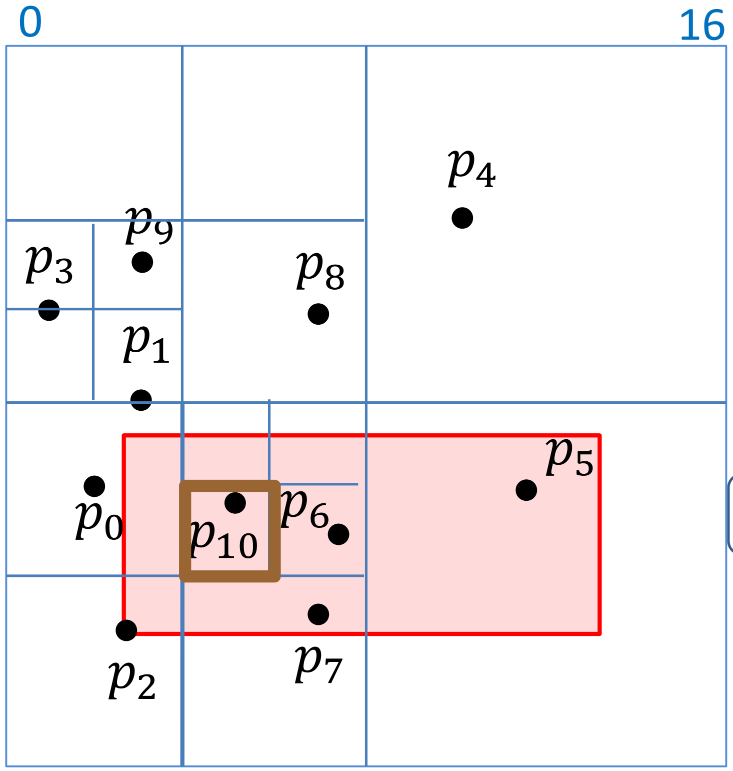
Quadtree Range Search Example

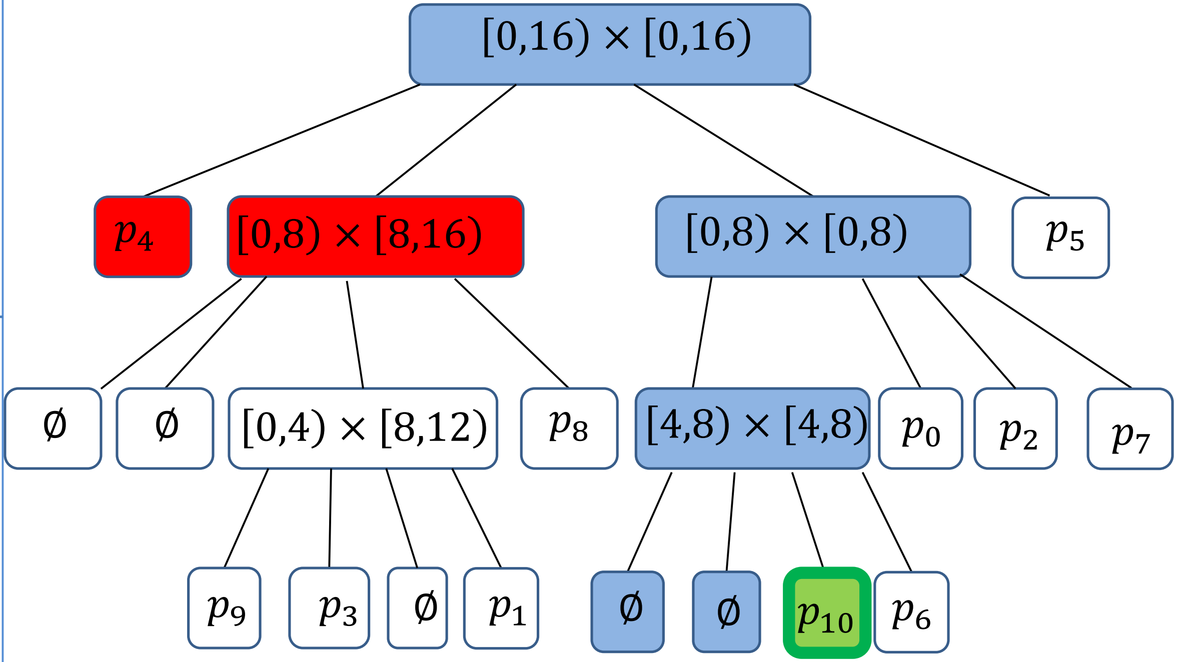




- Query rectangle $A = [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

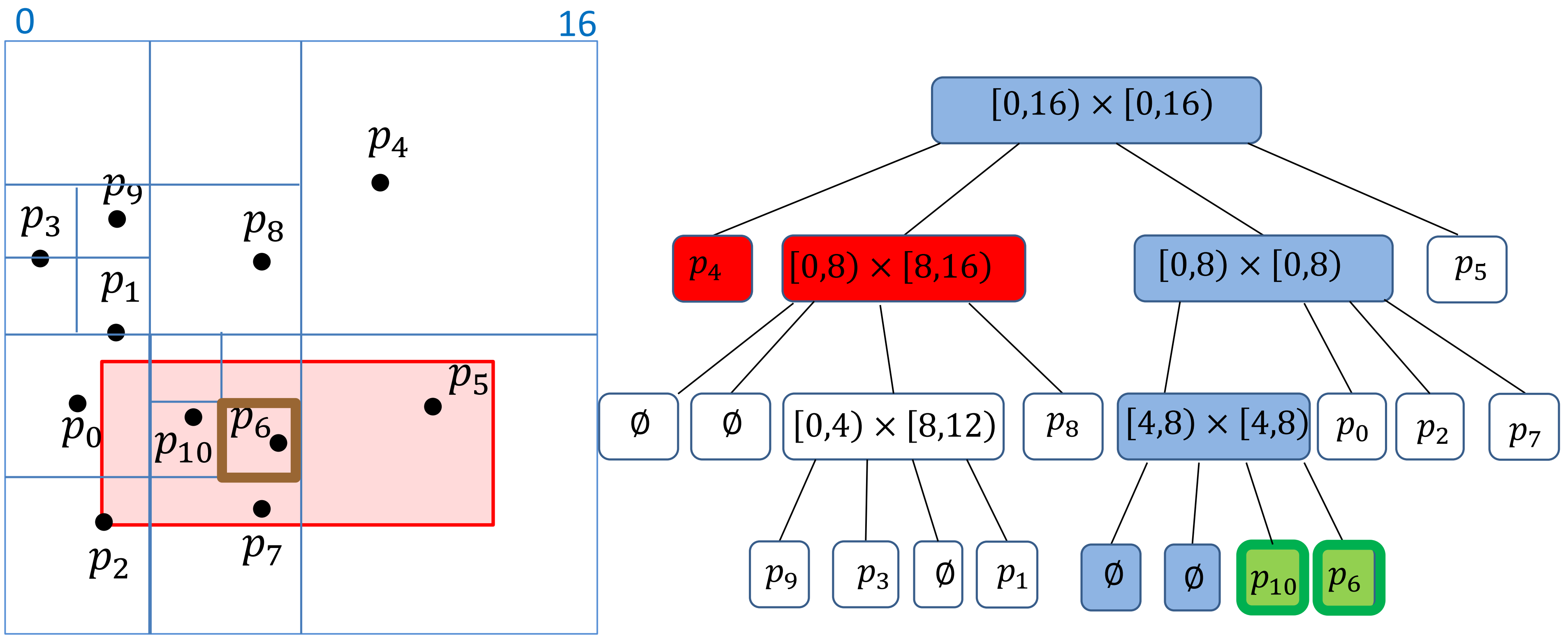
Quadtree Range Search Example





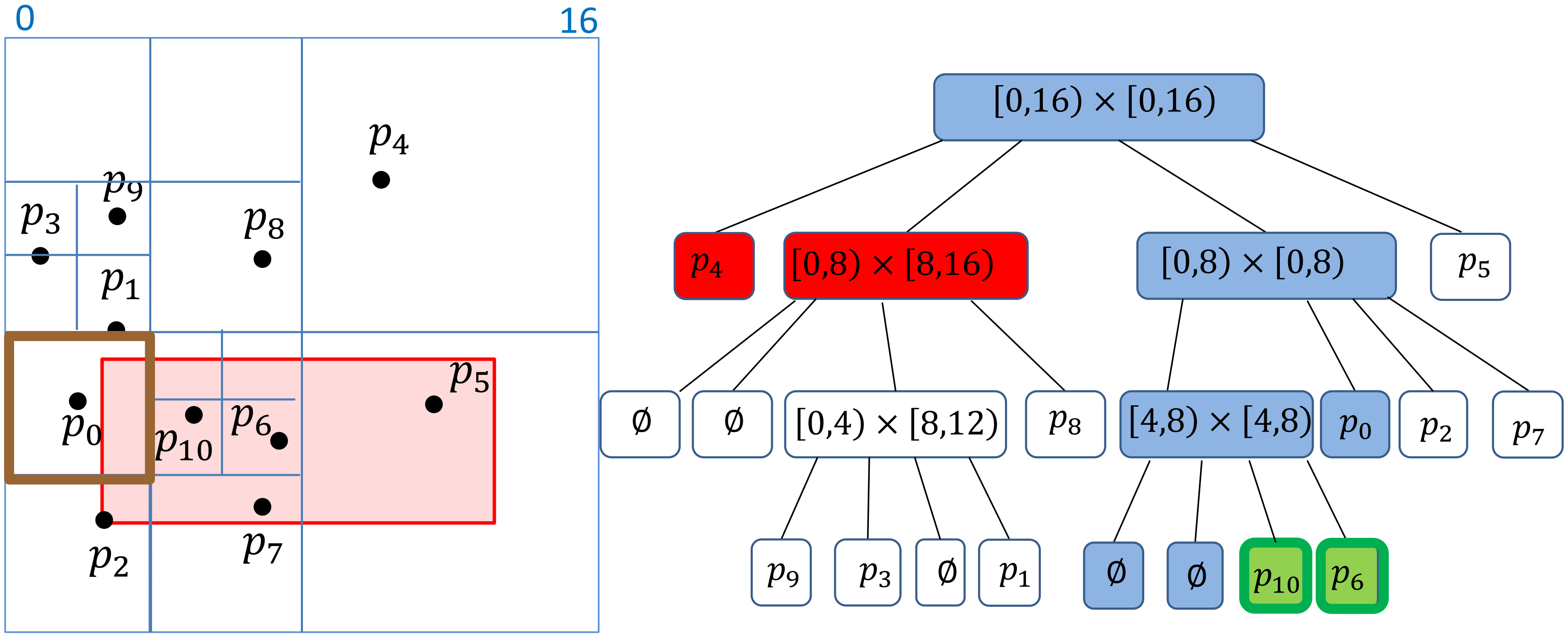
- Query rectangle A $= [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



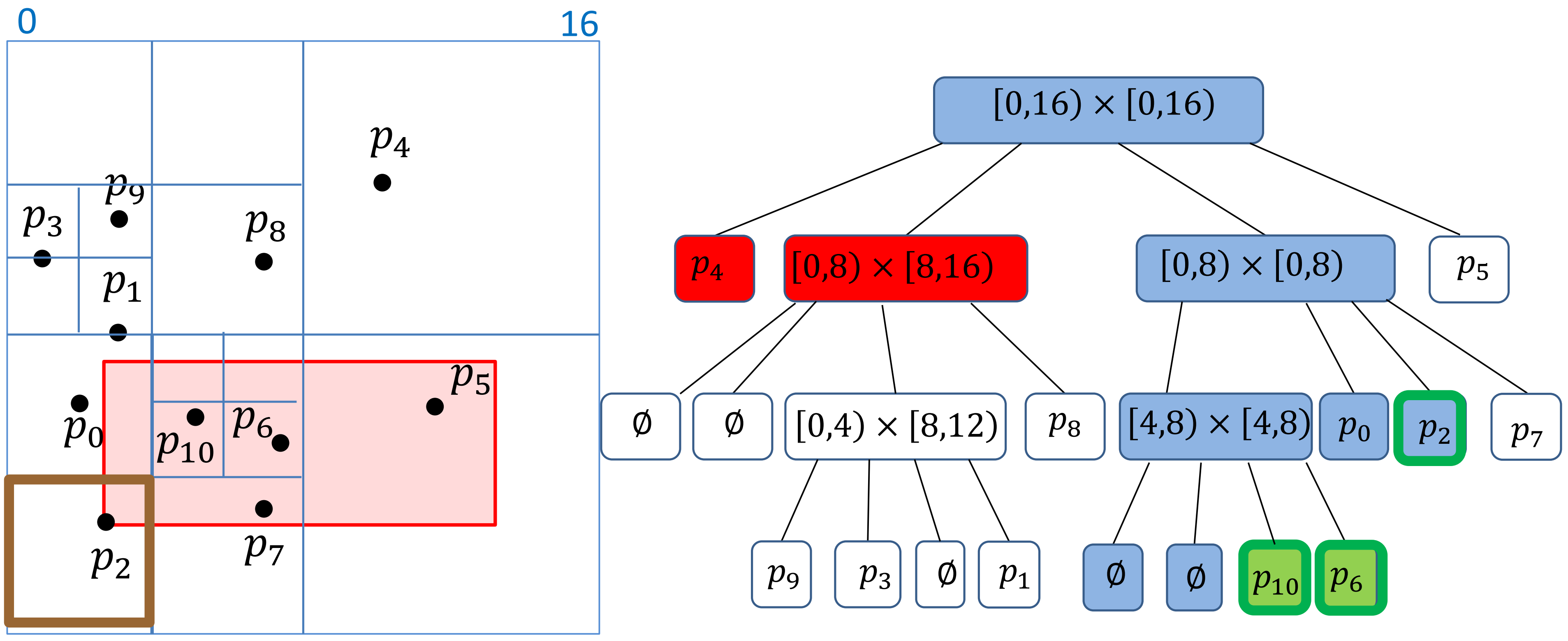
- Query rectangle A $= [3 \leq x < 13, 3 \leq y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



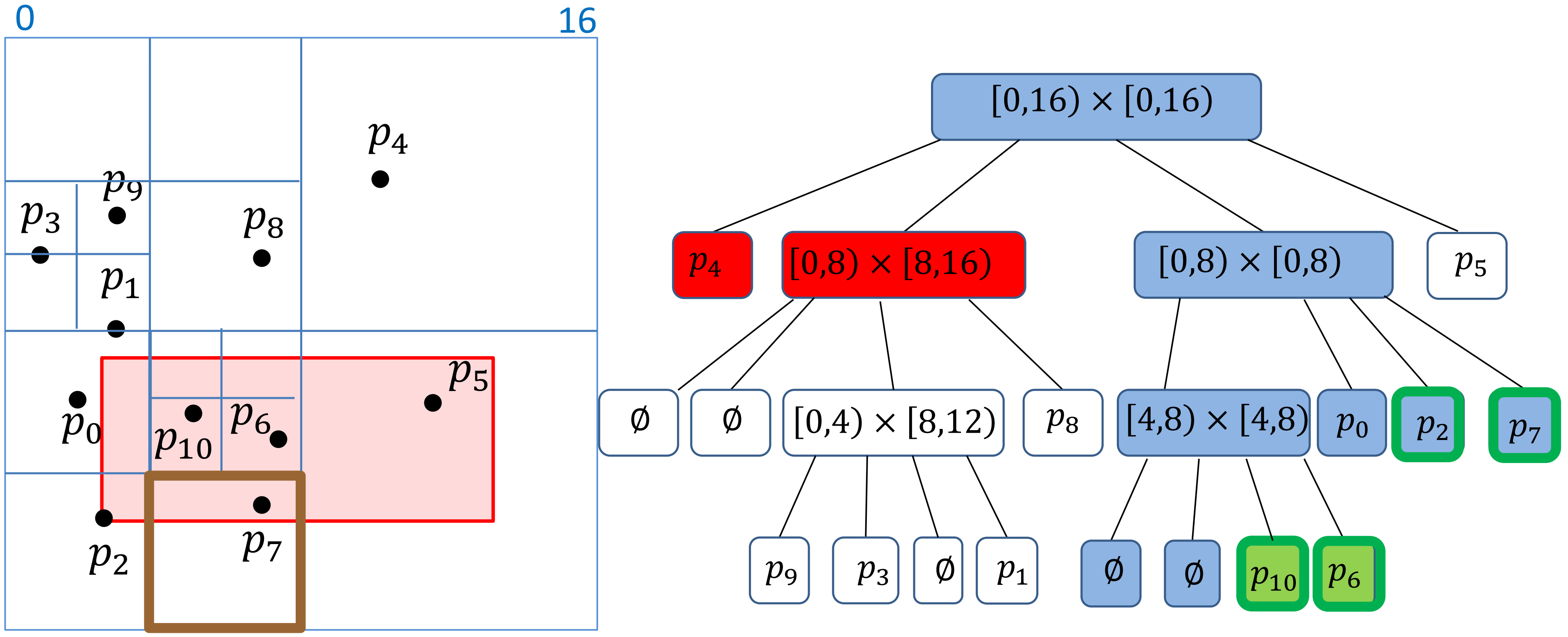
- Query rectangle A $= [3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



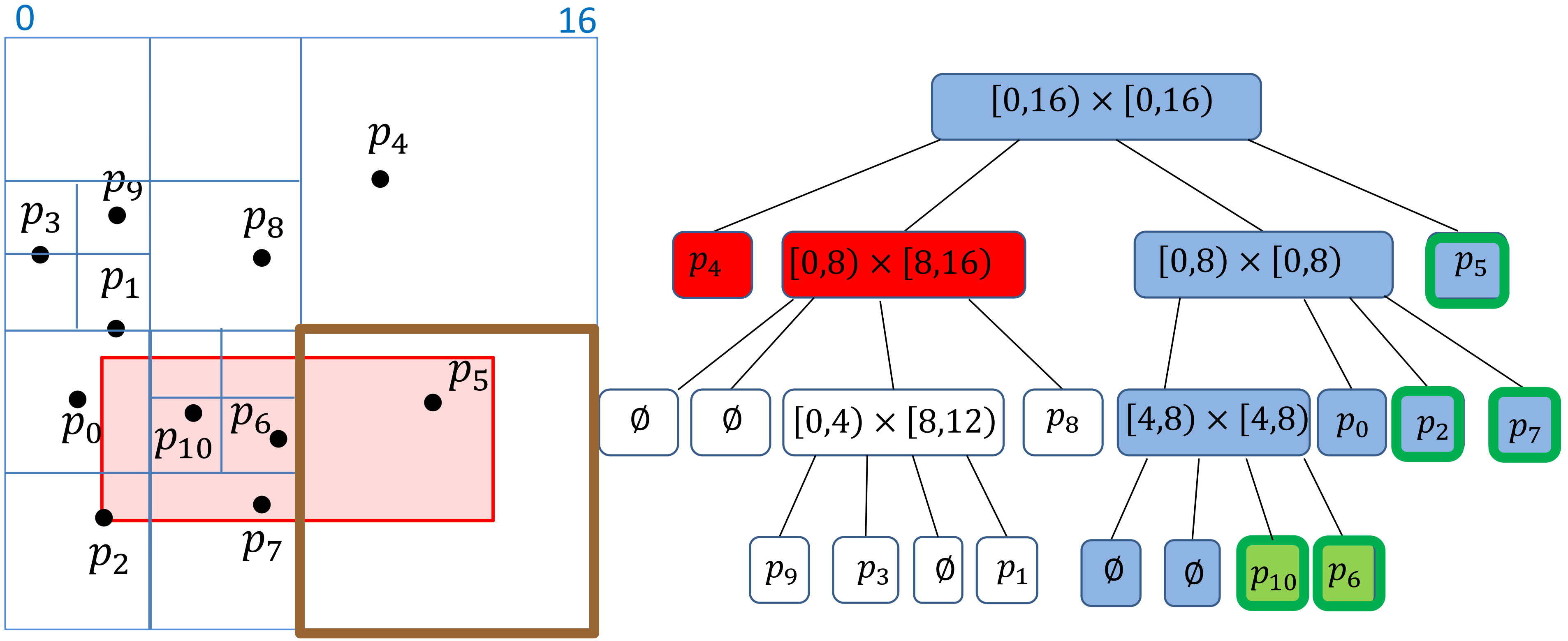
- Query rectangle A = $[3 \le x < 13, 3 \le y < 7]$
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (inside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq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

Quadtree Range Search Example



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

Quadtree Range Search Example



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

Quadtree Range Search

```
Qtree::RangeSearch($r \leftarrow root, A$)
$r$ : quadtree root, $A$: query rectangle
    let $R$ be the region associated with $r$
    if $R \subseteq A$ then      //inside node
        report all points below $r$
        return
    if $R \cap A = \emptyset$ then //outside node
        return
    // boundary node, recurse if not a leaf
    if $r$ is a leaf then // leaf, do not recurse
        $p \leftarrow$ point stored at $r$
        if $p$ is in $A$ return $p$
        else return
    for each child $v$ of $r$ do
        QTree-RangeSearch($v, A$)
```

- Code assumes each quadtree node stores the associated square
- Alternatively, these could be re-computed during search
    - space-time tradeoff

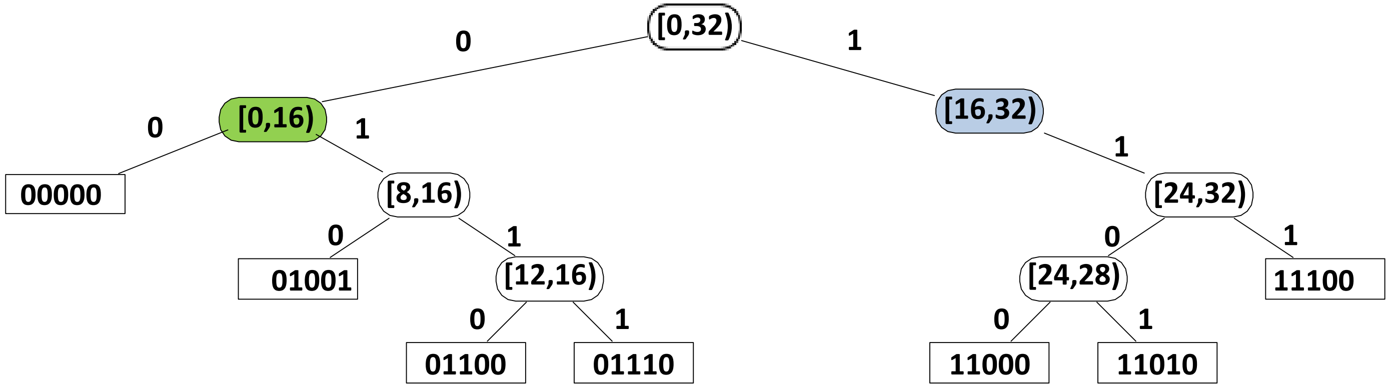
RangeSearch Analysis

- 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 trie
    - with splitting stopped once key is unique

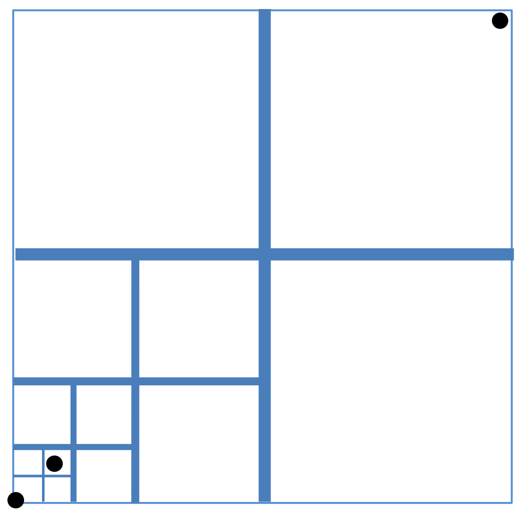
Quadtree summary

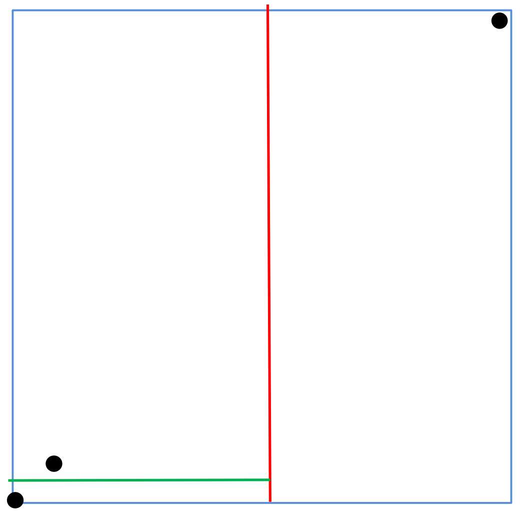
- Quadtrees easily generalize to higher dimensions
    - octrees, *etc.*
    - but rarely used beyond dimension 3
- Easy to compute and handle
- No complicated arithmetic, only divisions by 2
    - bit-shift if the width/height of R is a power of 2
- Space potentially wasteful, but good if points are 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

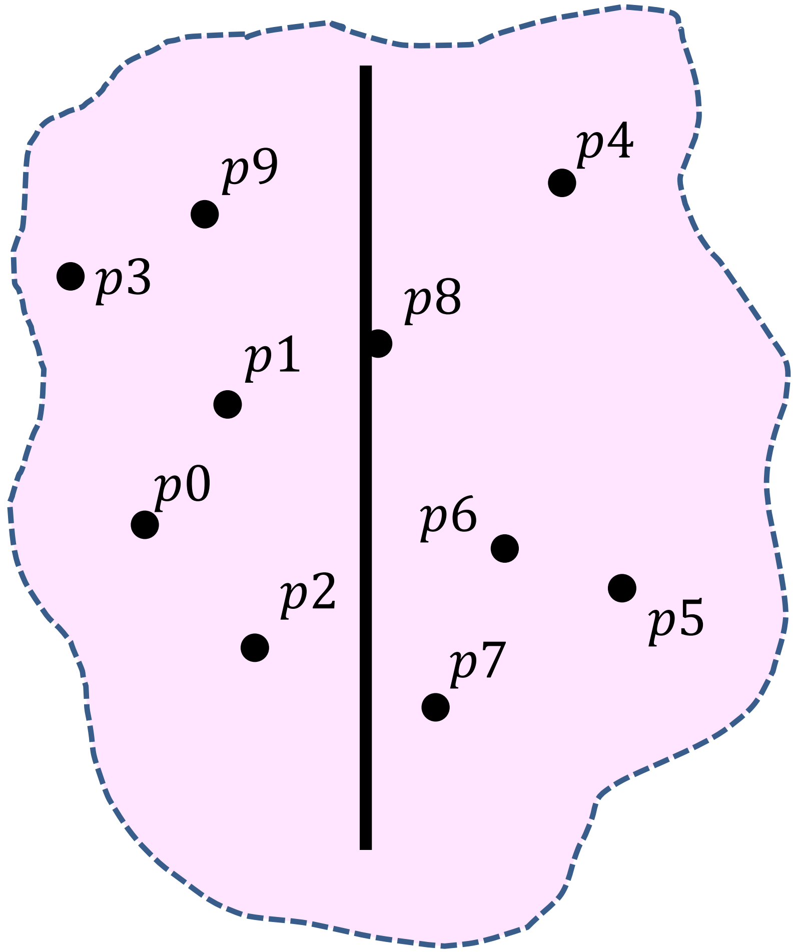
kd-tree motivation





- 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

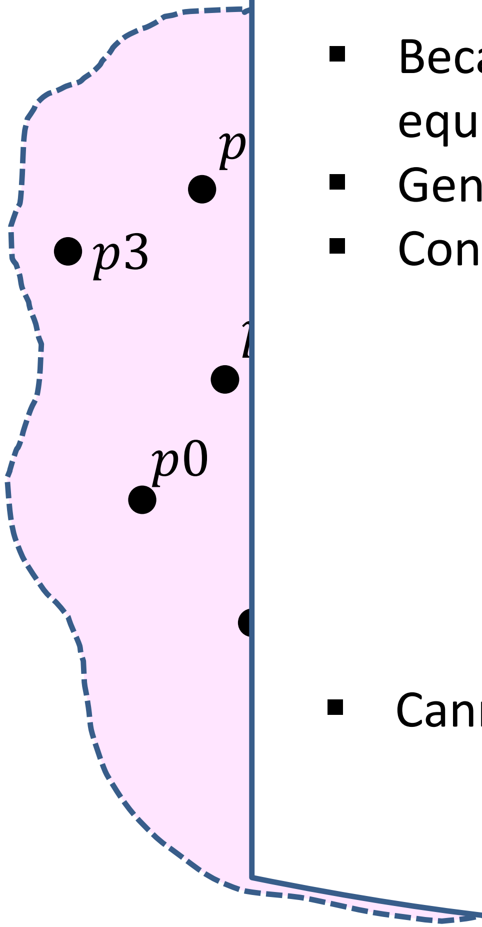


$\mathcal{R}^2$ is split into two half regions

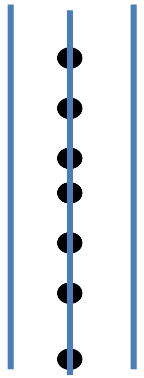
- No need for bounding box
- Root corresponds to the whole $\mathcal{R}^2$
- First find the best vertical split
- $\left\lfloor \frac{n}{2} \right\rfloor$ on one side and $\left\lceil \frac{n}{2} \right\rceil$ and points on the other

$x < p8.x$

kd-tree example



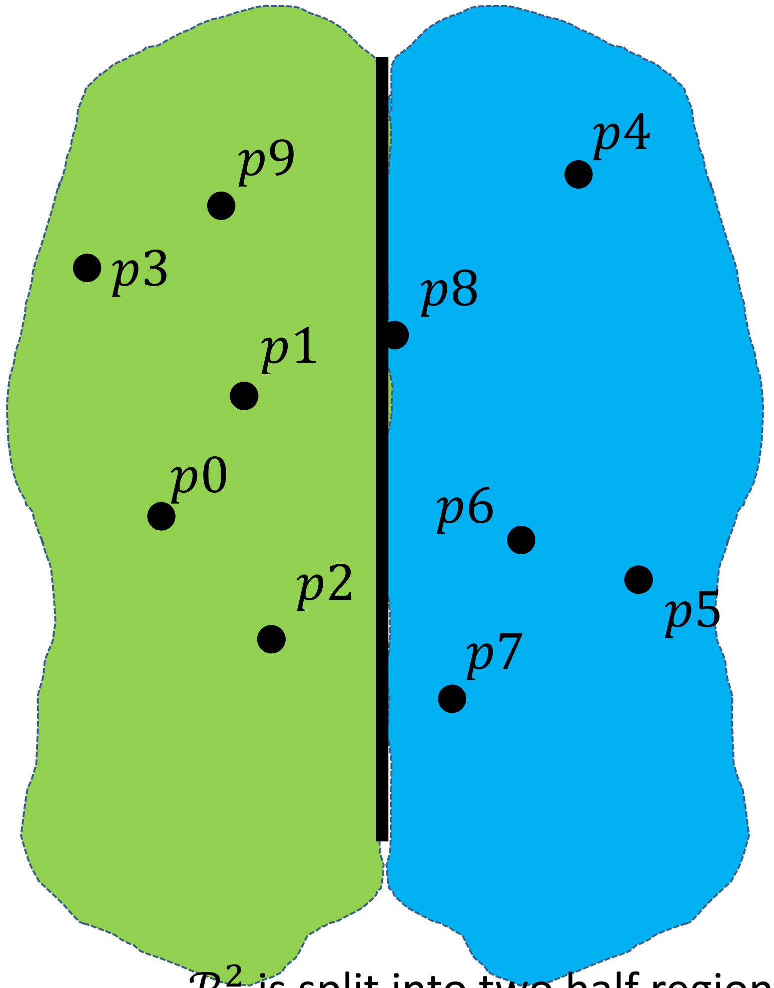
- Because points are in general position, always can split in two equal (or almost equal subsets)
- General position means no two x or y coordinates are the same
- Consider the points below **not** in general position



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

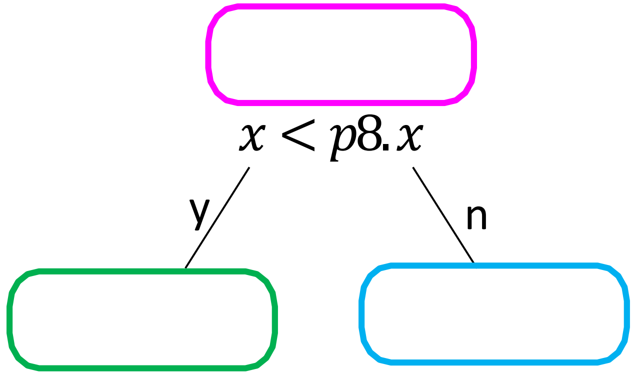
$\mathcal{R}^2$ is split into two half regions

kd-tree example

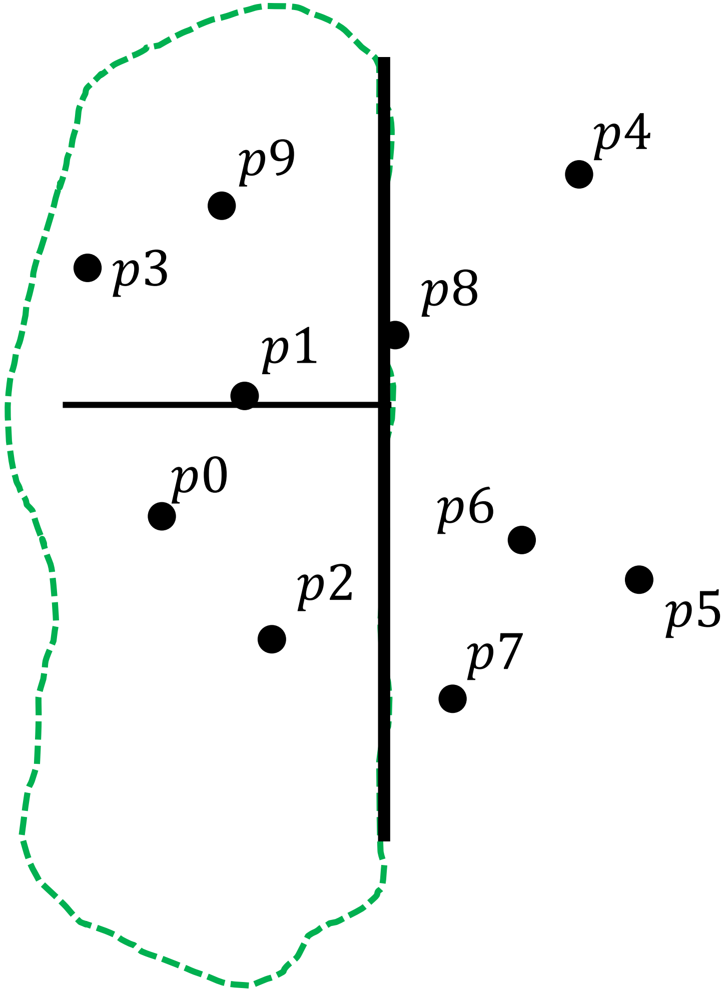


$\mathcal{R}^2$ is split into two half regions

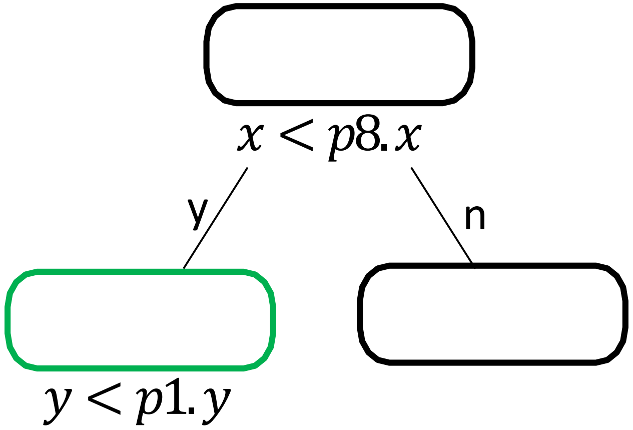
- No need for bounding box
- Root corresponds to the whole $\mathcal{R}^2$
- First find the best vertical split
- $\left\lfloor \frac{n}{2} \right\rfloor$ on one side and $\left\lceil \frac{n}{2} \right\rceil$ and points on the other



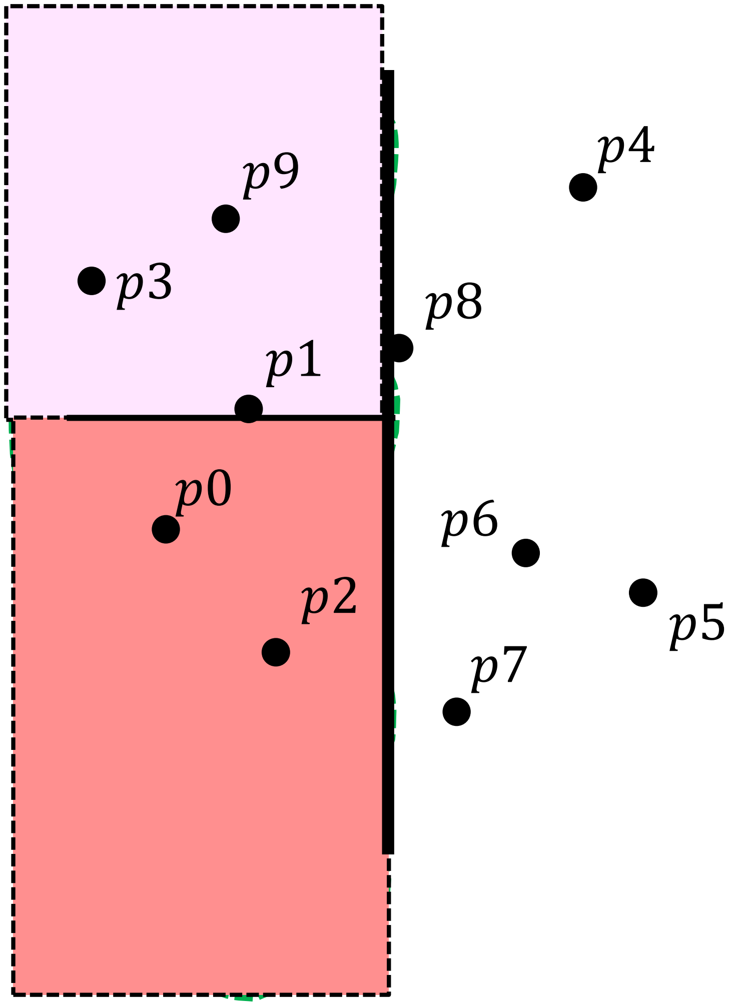
kd-tree example



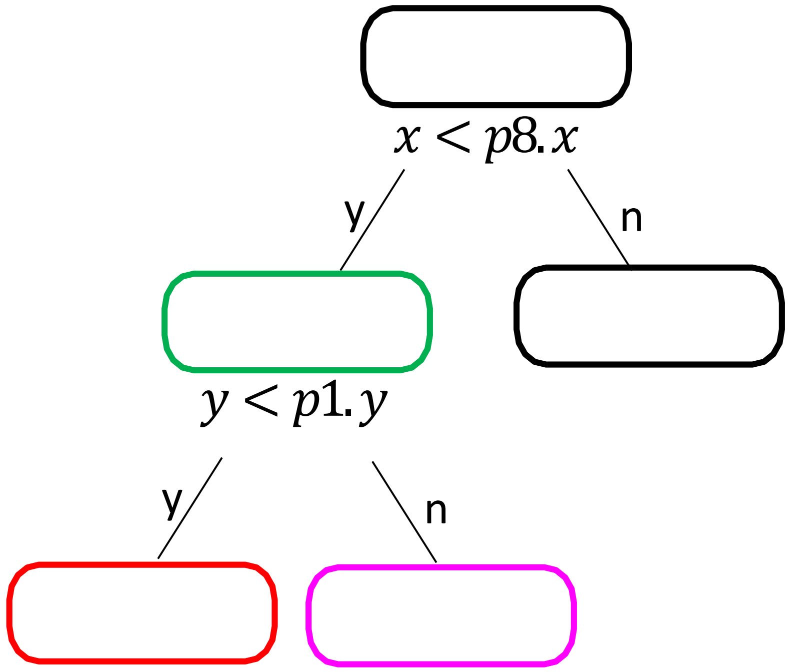
- Recurse on the resulting regions
    - if they have more than one point
- Alternate split direction



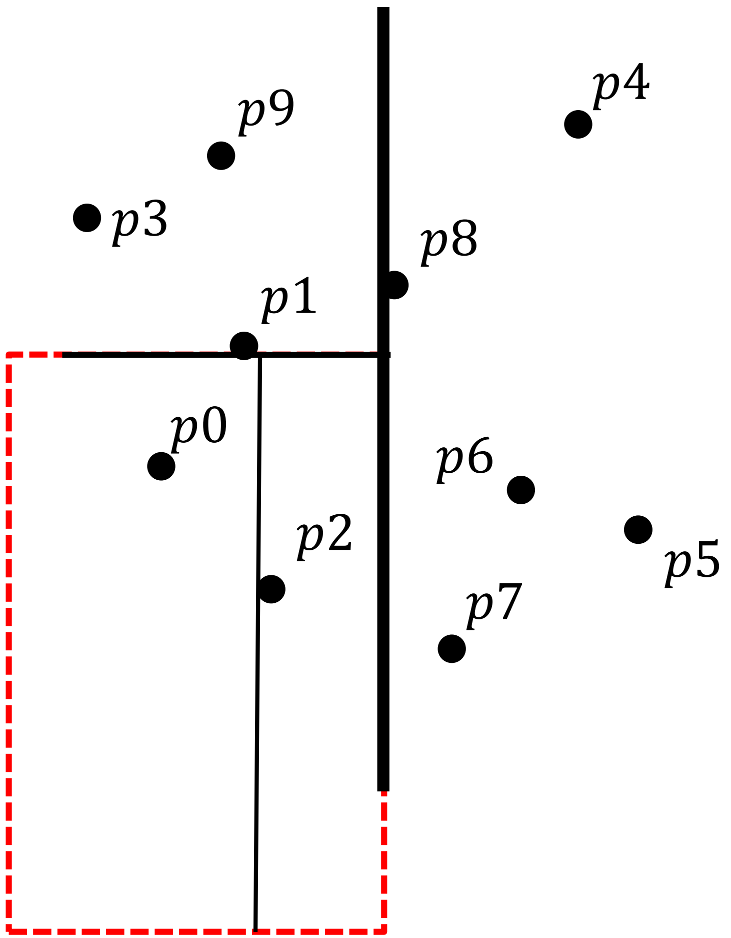
kd-tree example



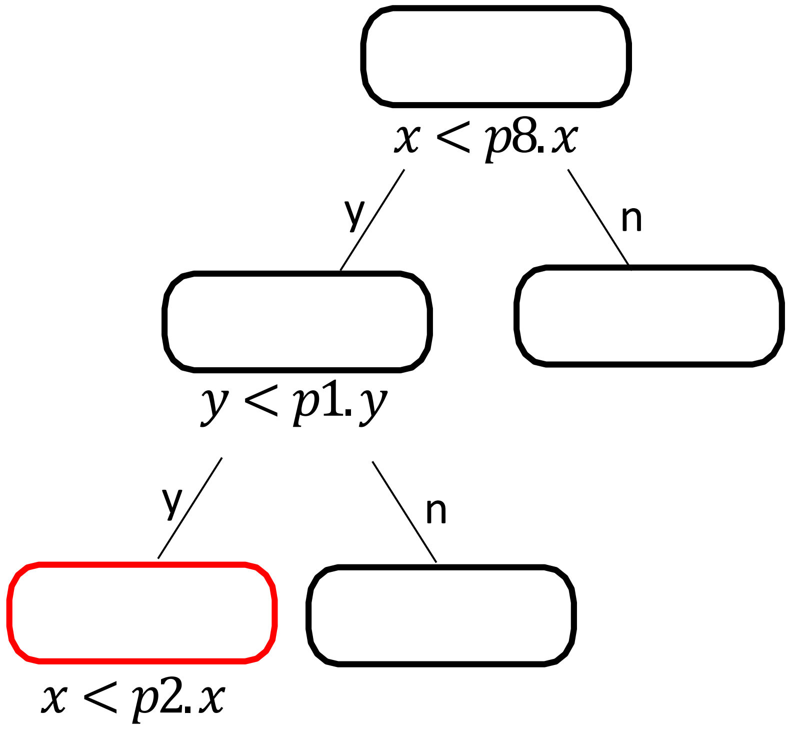
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



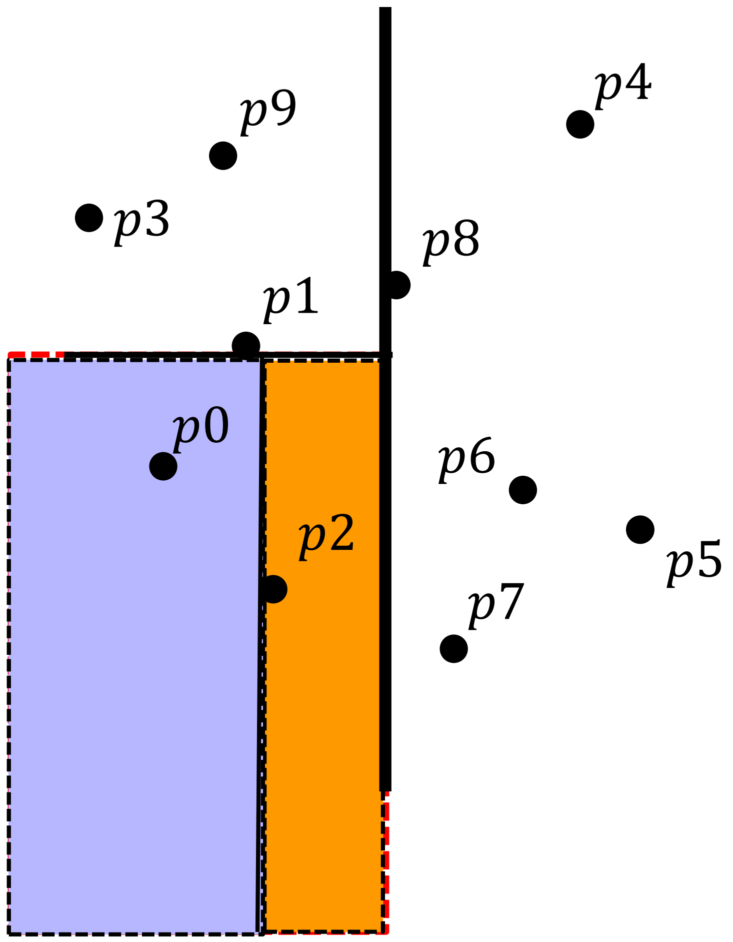
kd-tree example



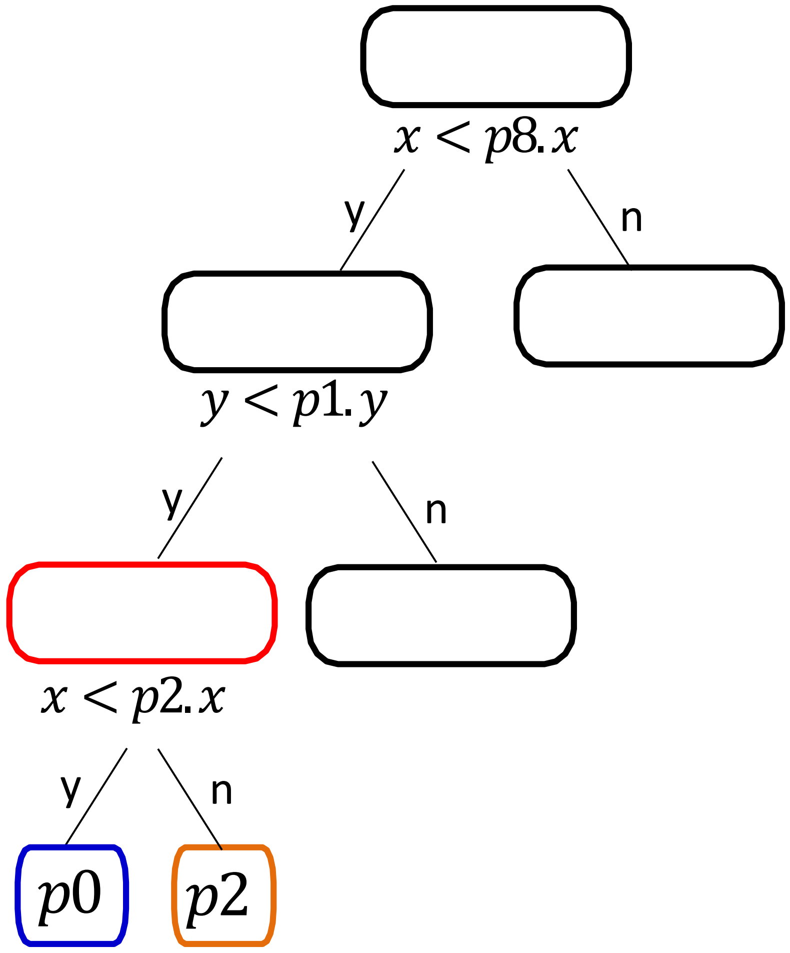
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



kd-tree example

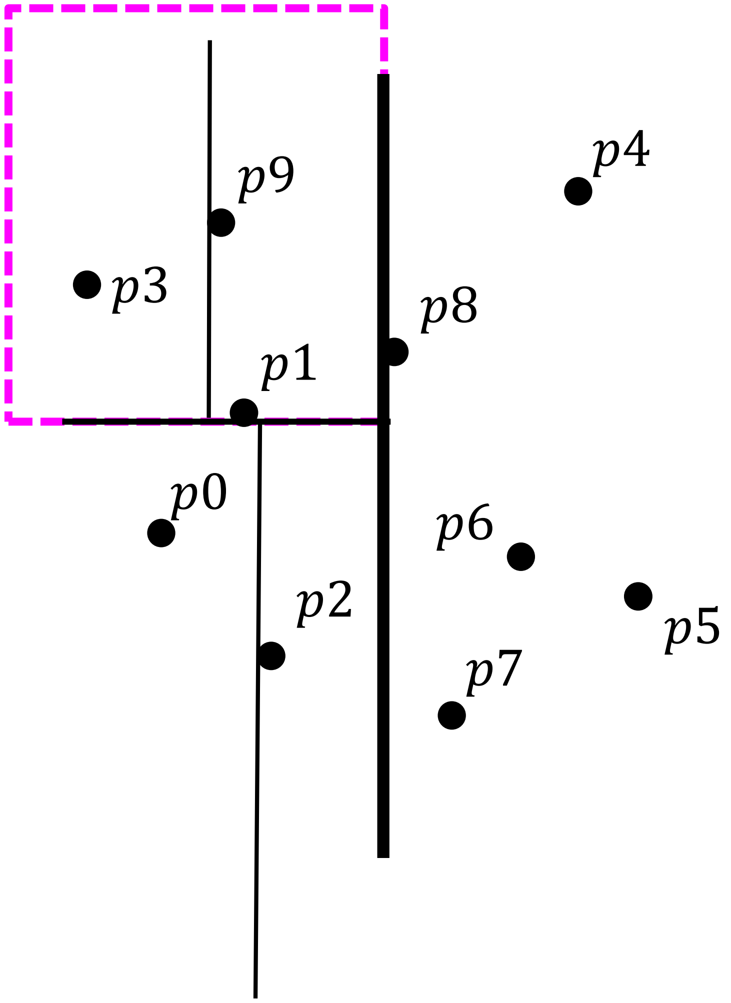


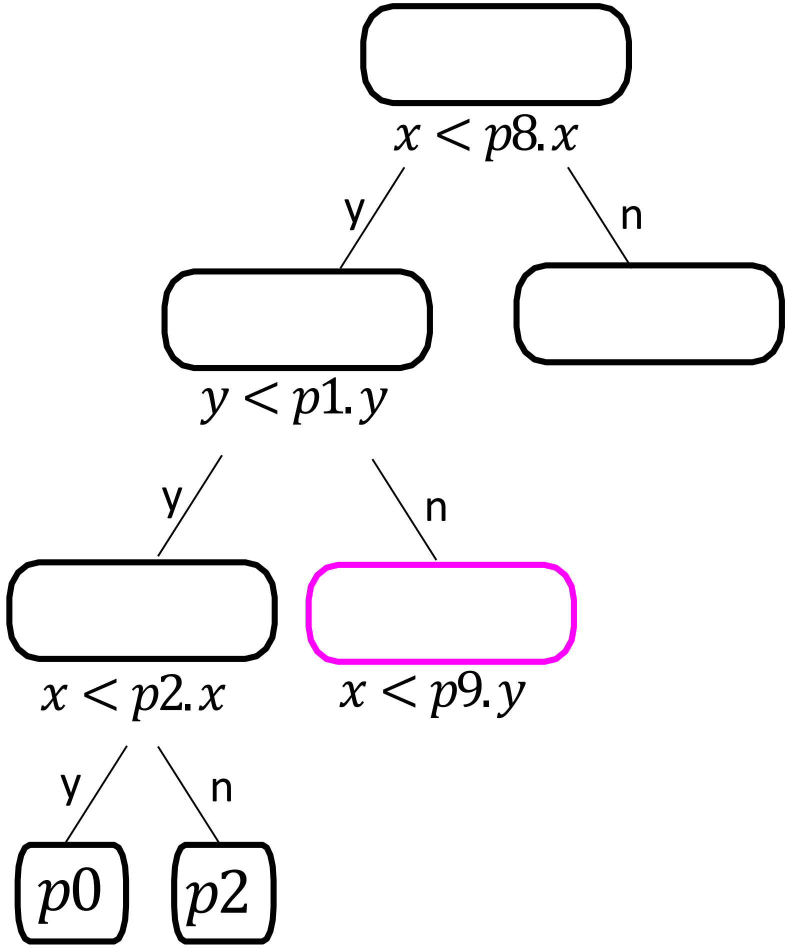
- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction



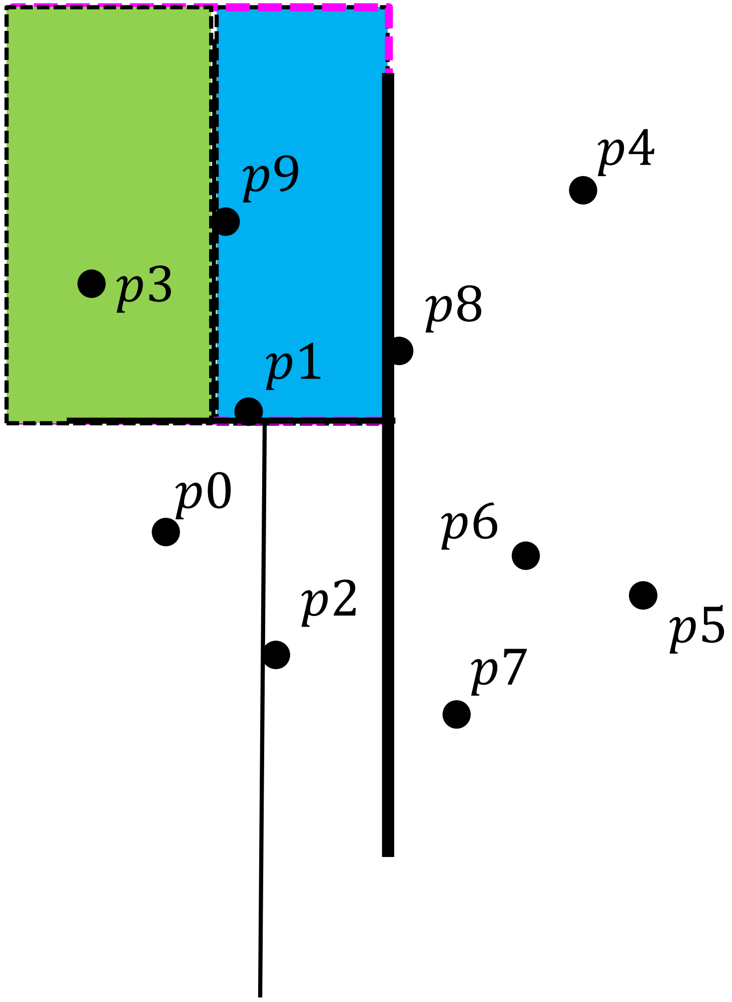
kd-tree example

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

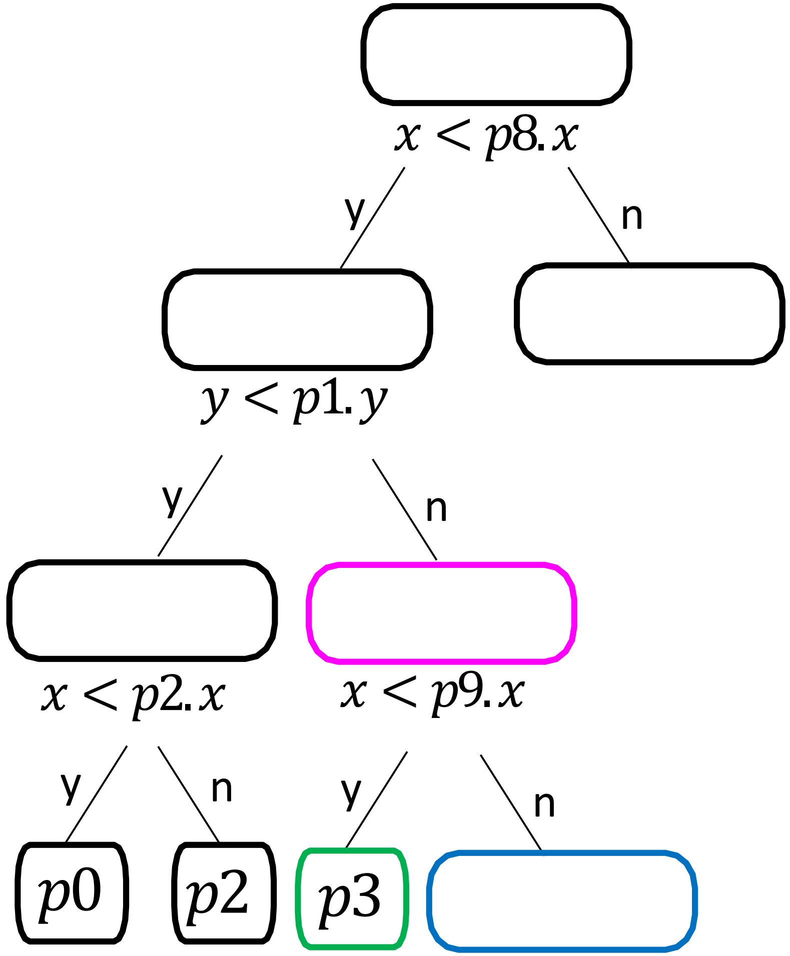




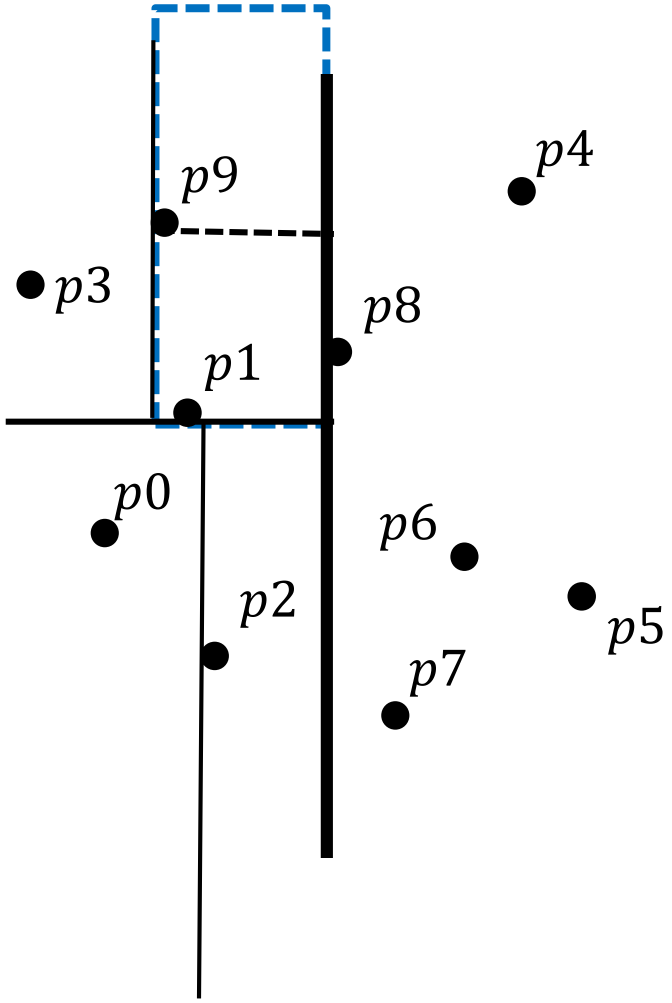
kd-tree example

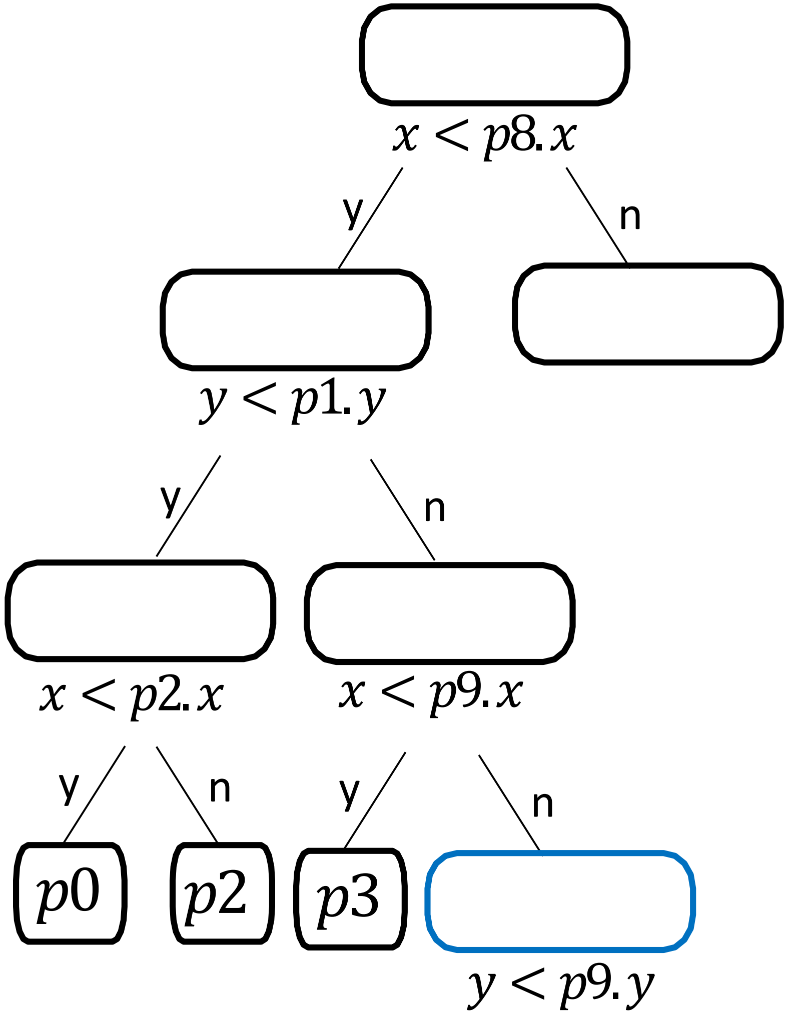


- Recurse on the resulting regions
  - if they have more than one point
- Alternate split direction

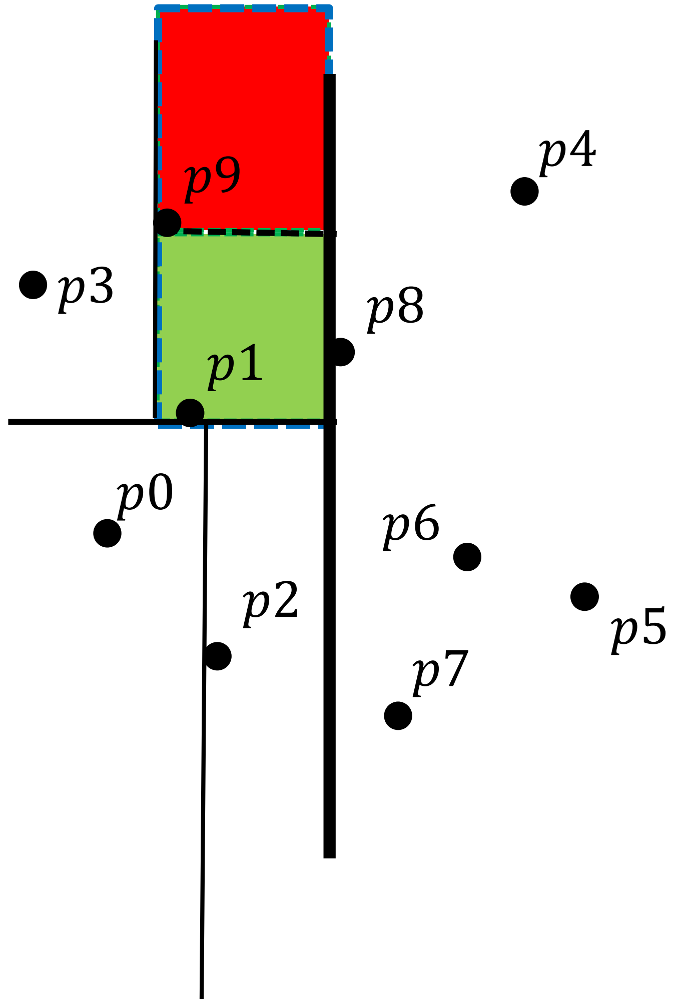


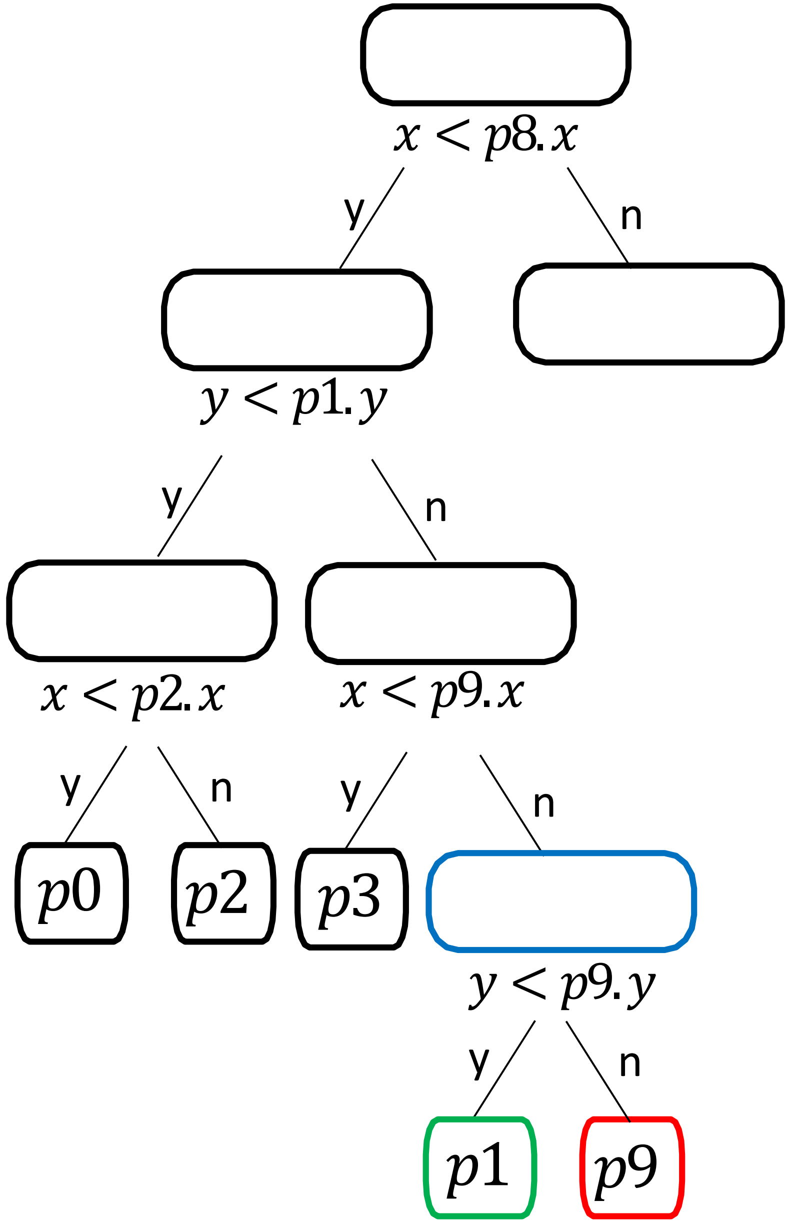
kd-tree example



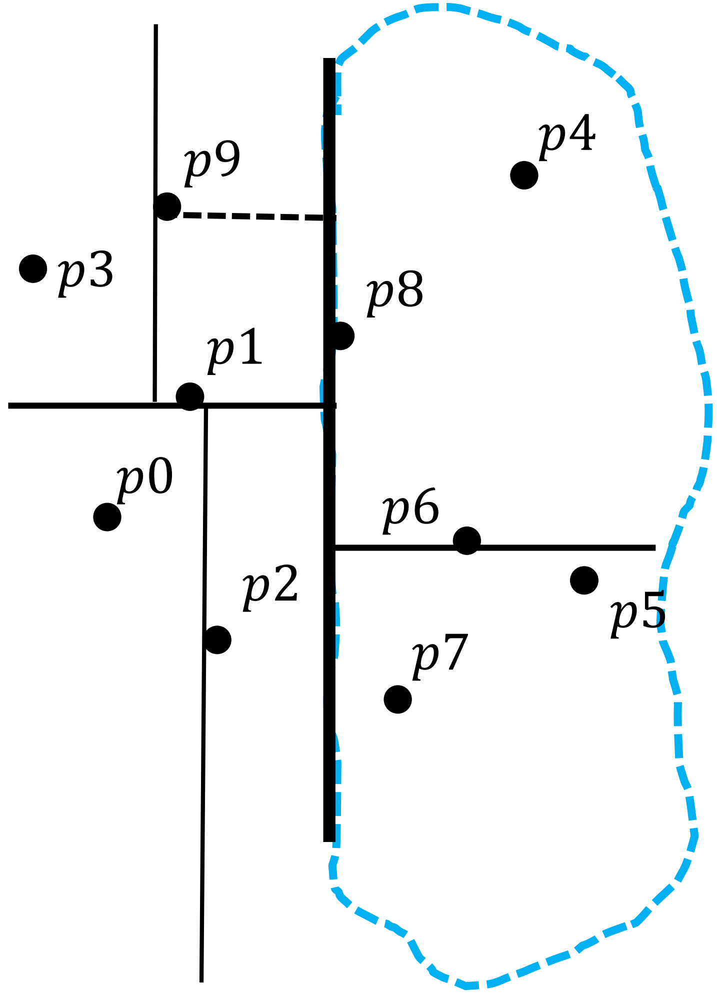


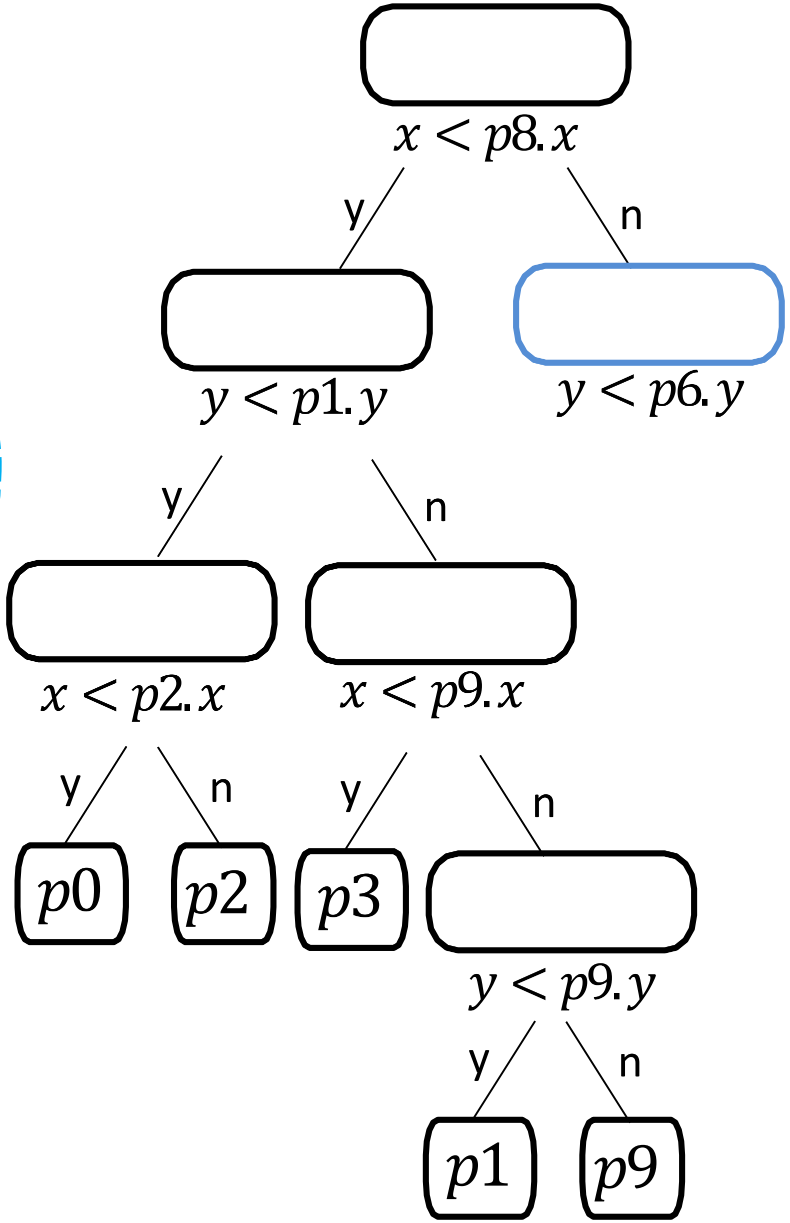
kd-tree example



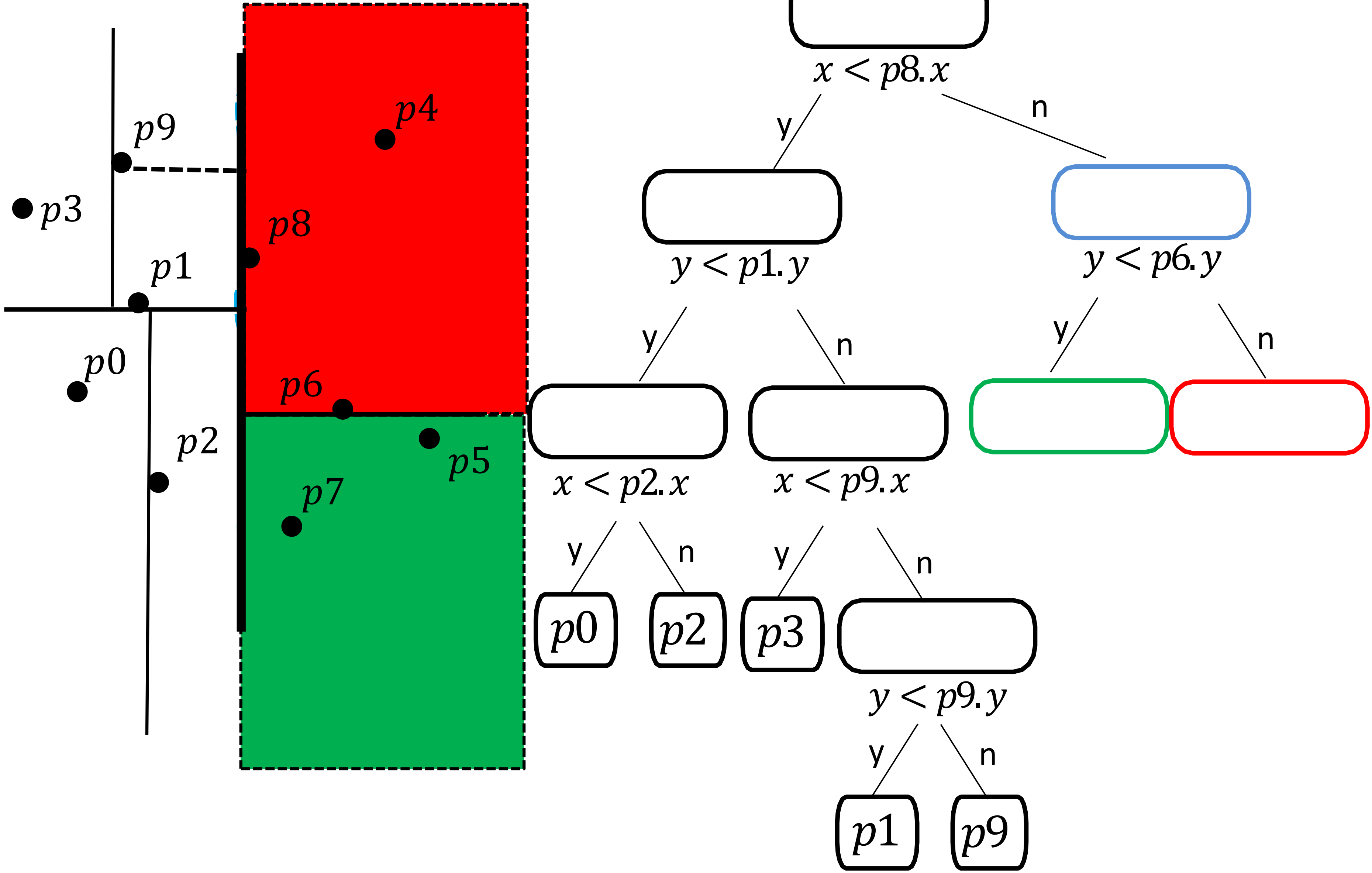


kd-tree example

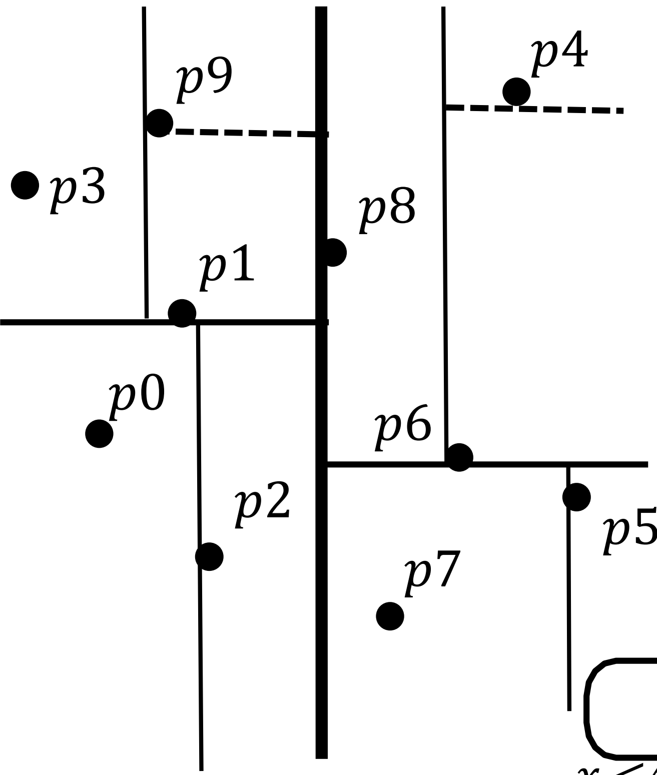


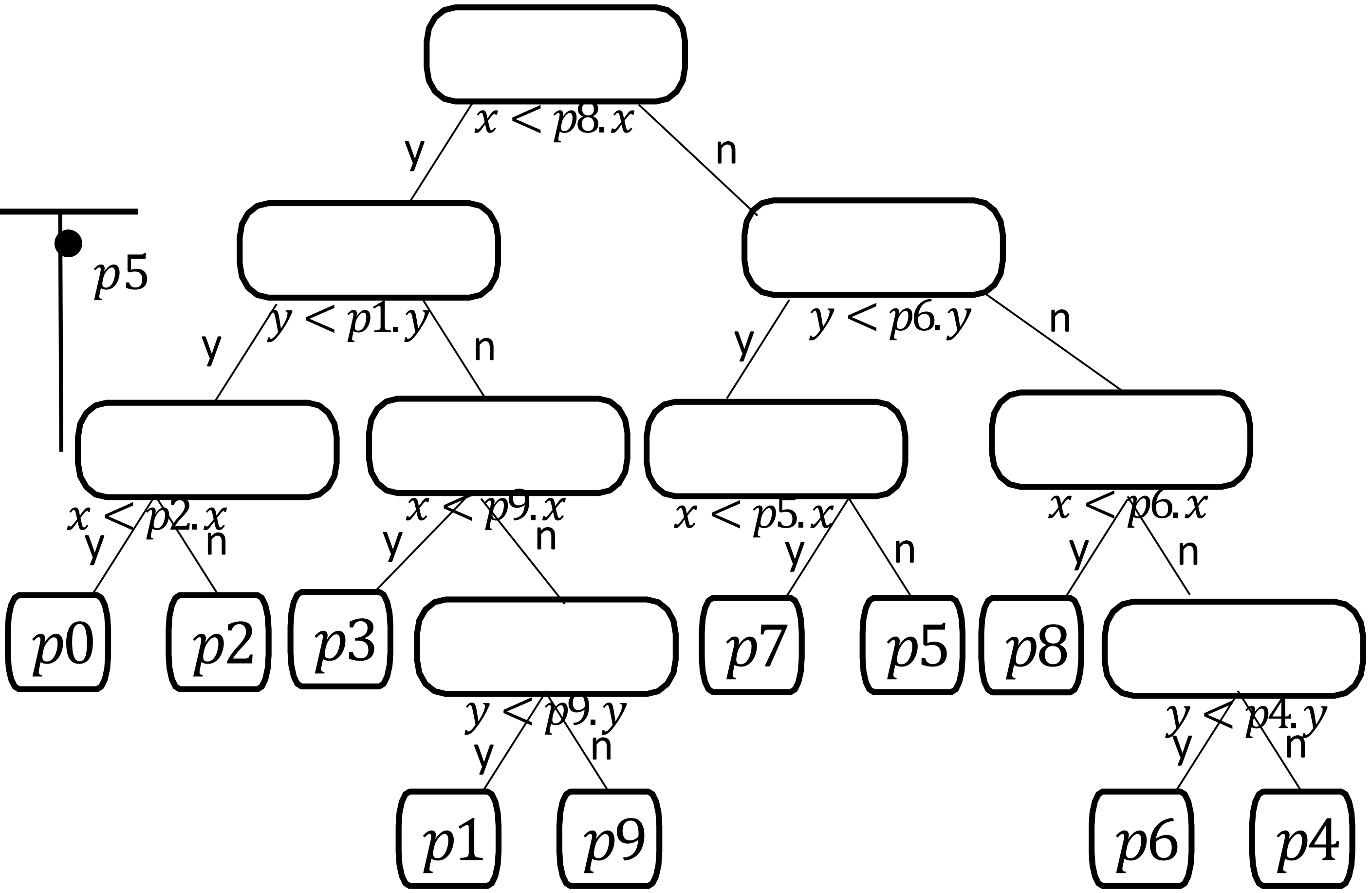


kd-tree example

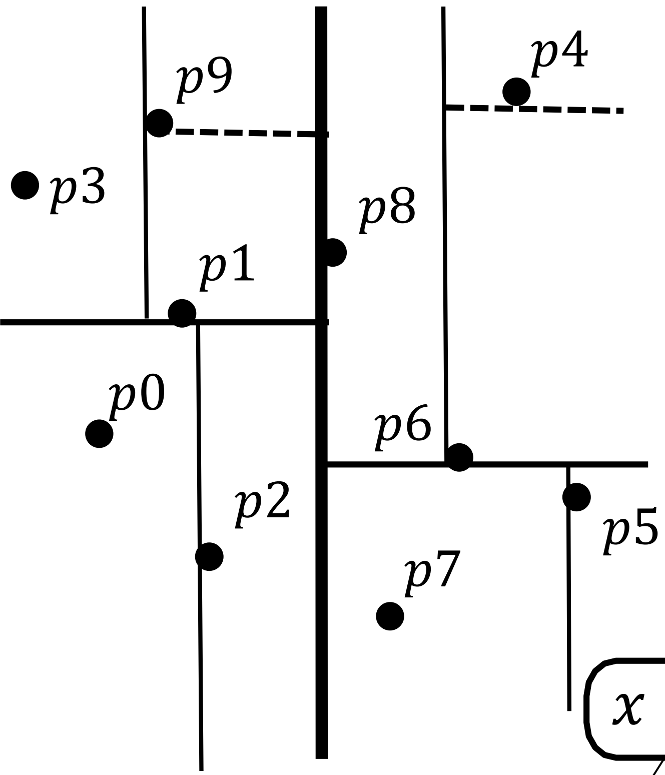


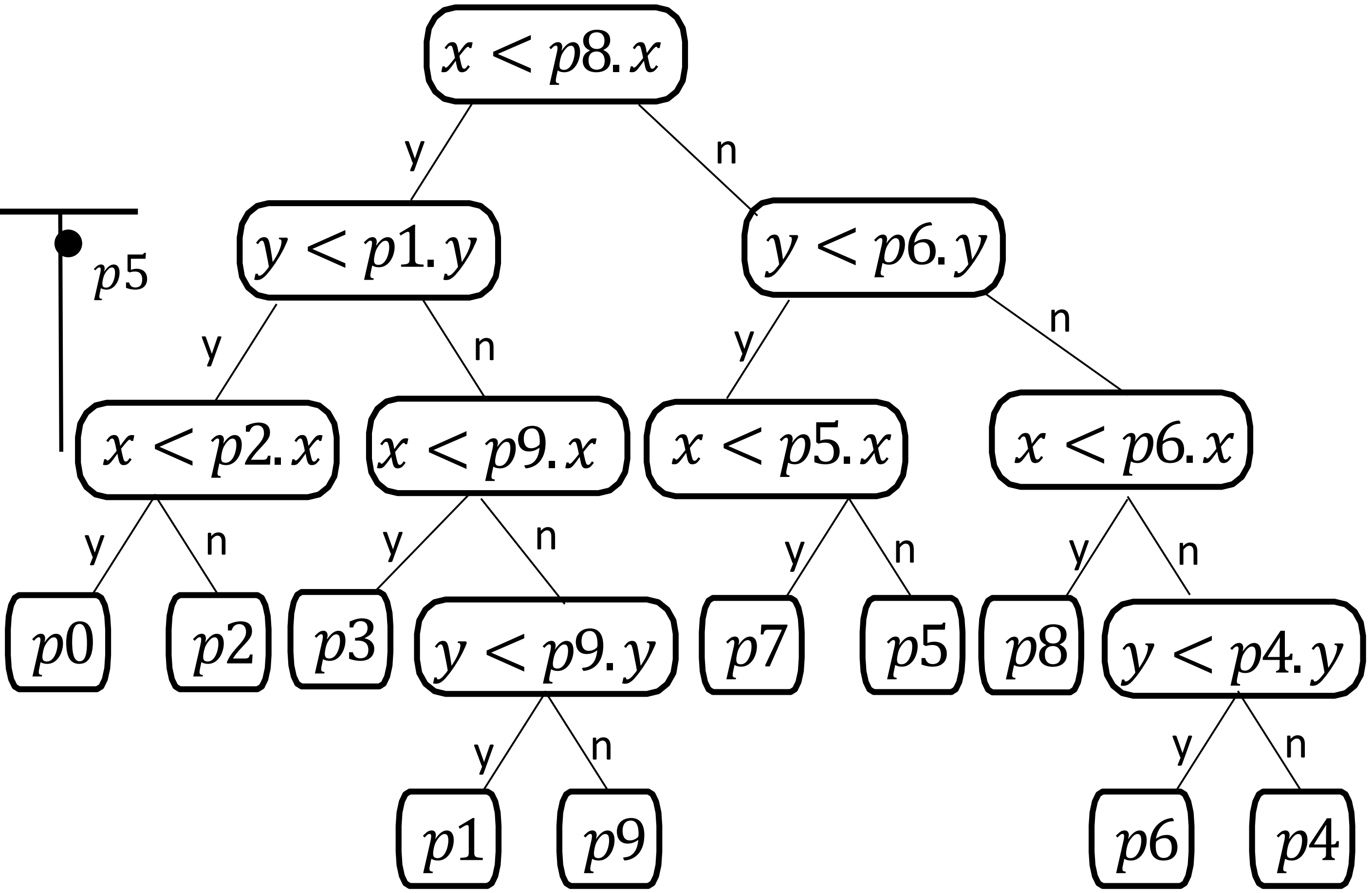
kd-tree example





kd-tree example





Building kd-trees

- Points $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$
- Build kd-tree with initial x-split
  - if $|S| \leq 1$ 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 $S_{x\geq m}$ by comparing the x coordinate of a point with m
    - create left subtree recursively (splitting on y) for points $S_{x<m}$
    - create right subtree recursively (splitting on y) for points $S_{x\geq m}$
    - 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

- 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
- Running time can be improved to $\Theta(n \log n)$ worst-case by pre-sorting coordinates
- Recurrence inequality for height

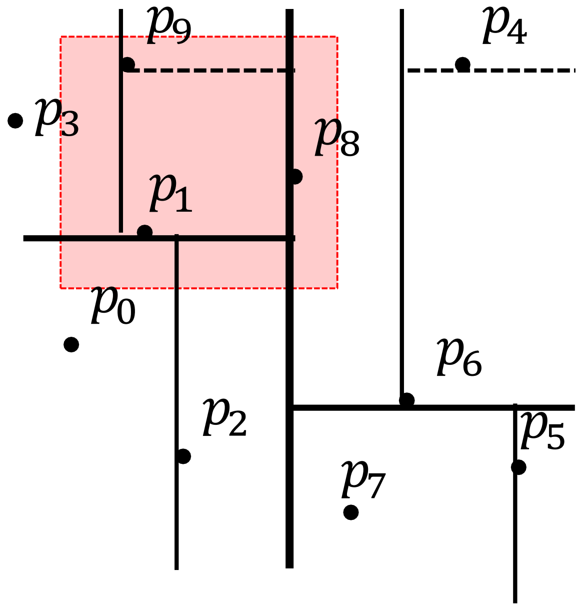
$$h(1) = 0$$
$$h(n) \leq h\left(\left\lceil\frac{n}{2}\right\rceil\right) + 1$$

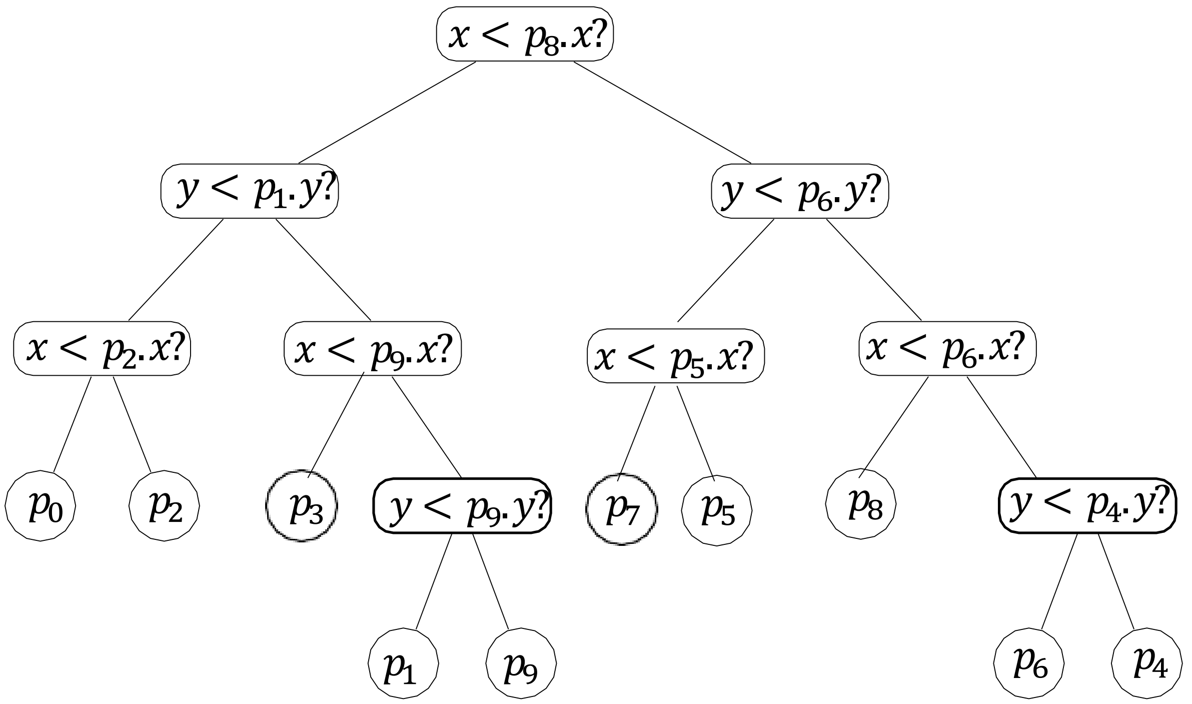
  - resolves to $O(\log n)$, specifically $\lceil \log n \rceil$

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**
    - kd-tree do not handle insertion/delection well
    - after insert or delete, split might no longer be at exact median
    - height is no longer guaranteed to be $O(\log n)$
    - remedy
        - allow a certain imbalance
        - re-building the entire tree when it becomes too unbalanced
        - no details
        - but *rangeSearch* will be slower

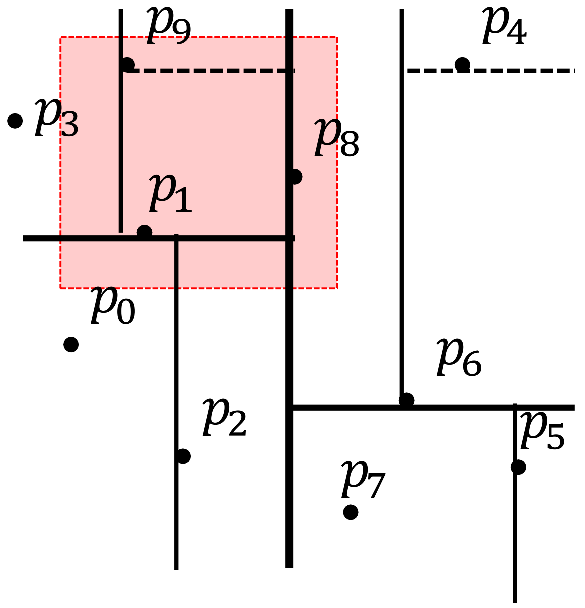
kd-tree: Range Search Example

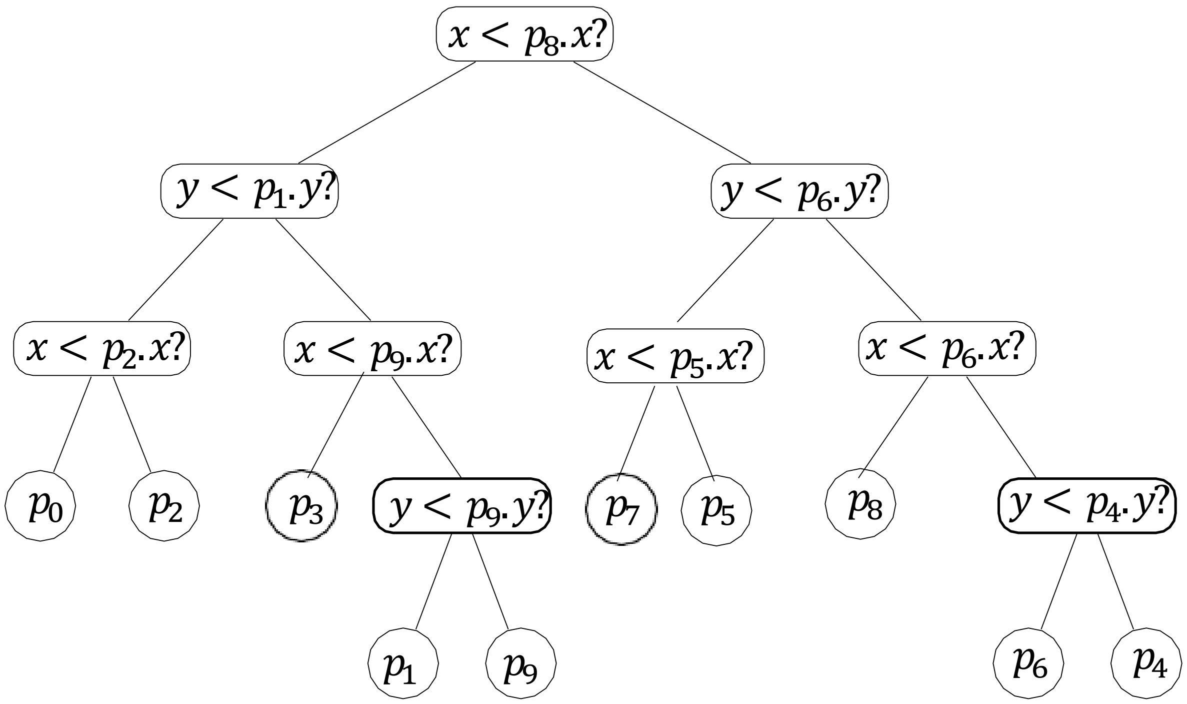




- Every node is associated with a region
    - range search is similar to quadtrees

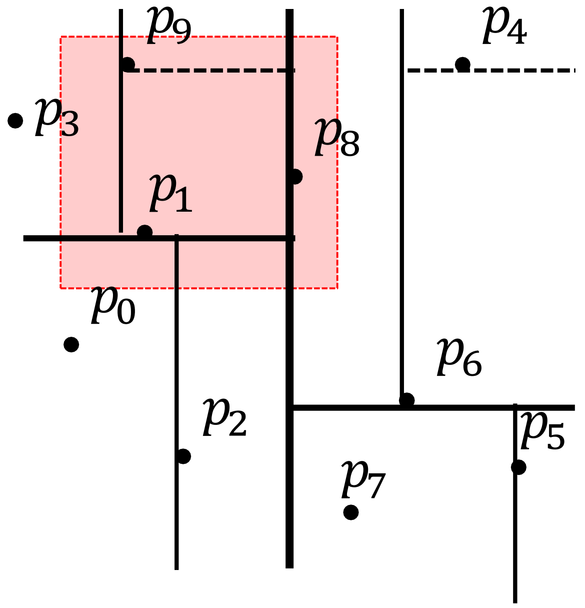
kd-tree: Range Search Example

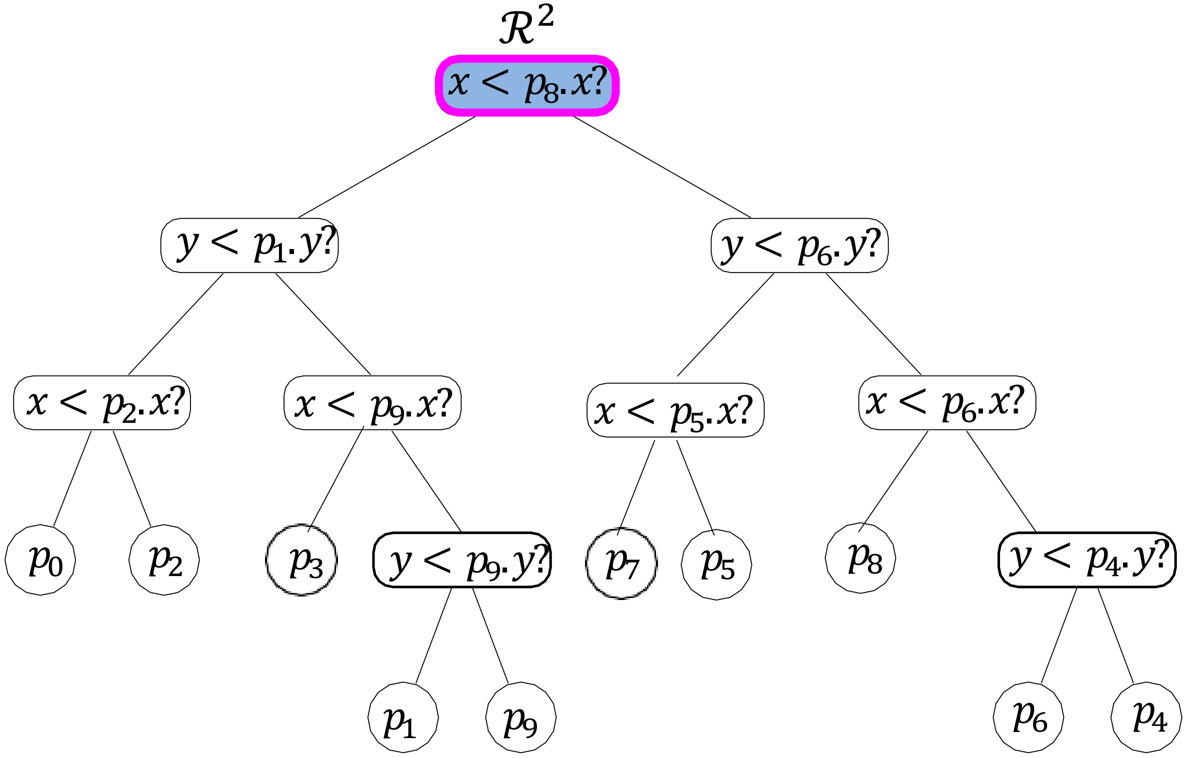




- Query rectangle A
- Let R be region associated with current node, have 3 cases
    1. $R \cap A = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
    3. $R \cap A \neq \emptyset$: blue (boundary)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

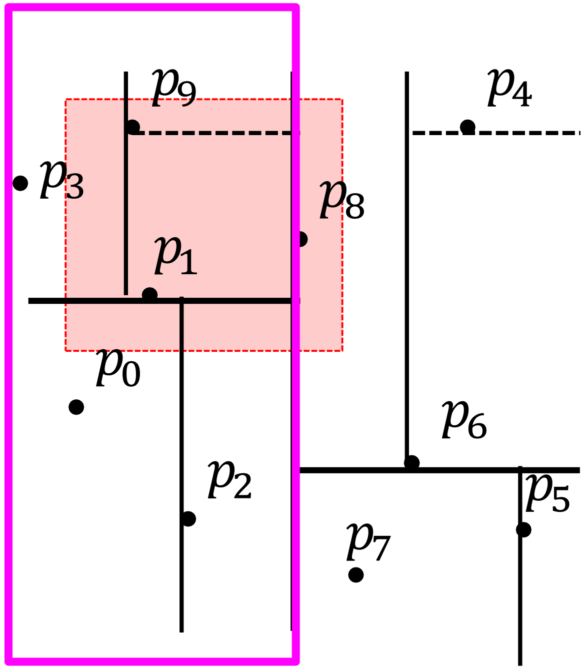
kd-tree: Range Search Example

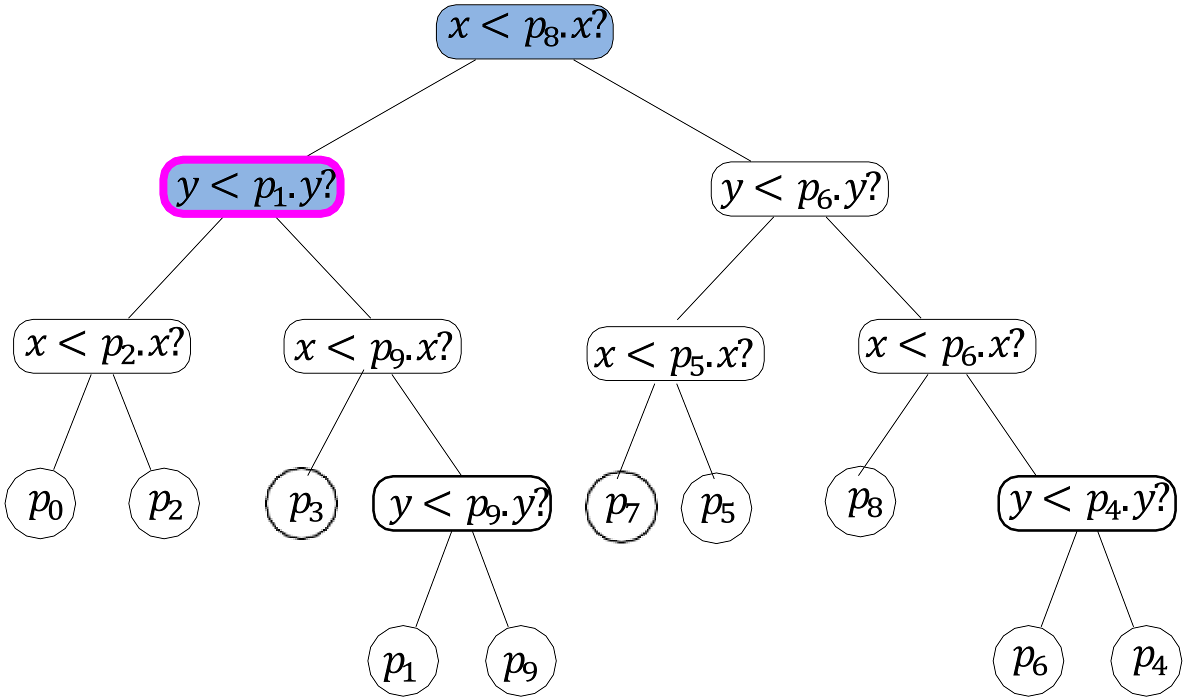




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

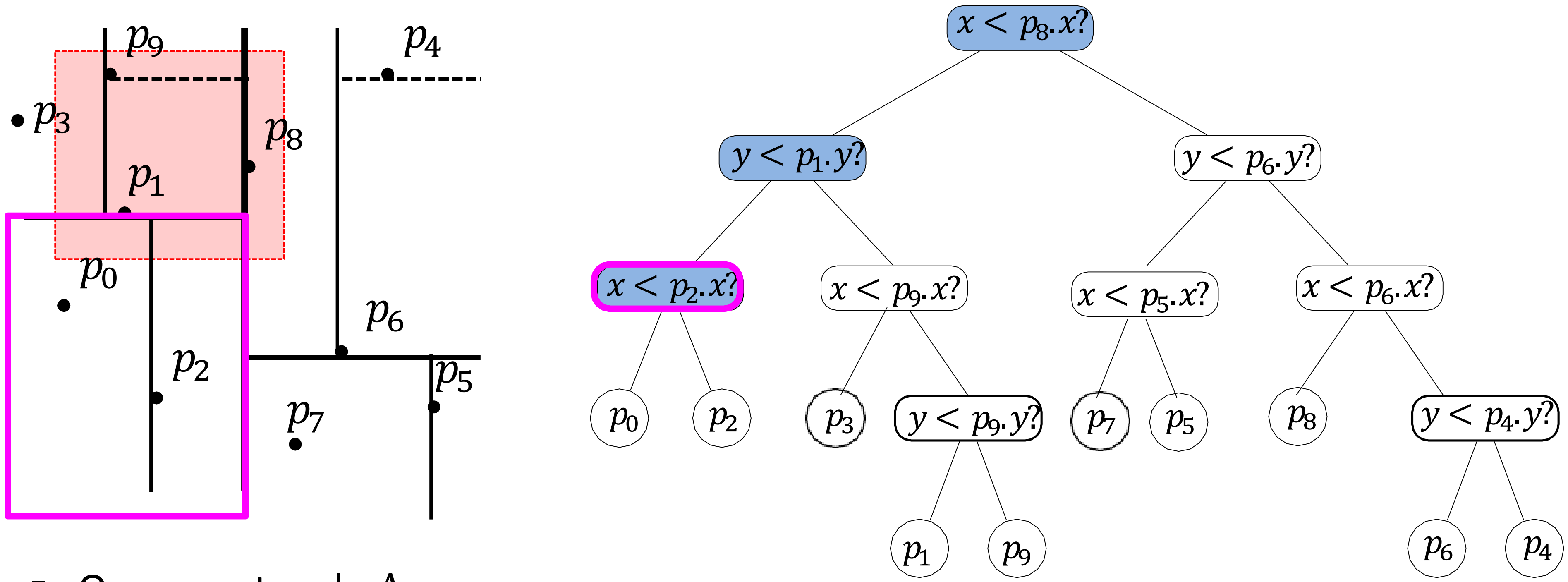
kd-tree: Range Search Example





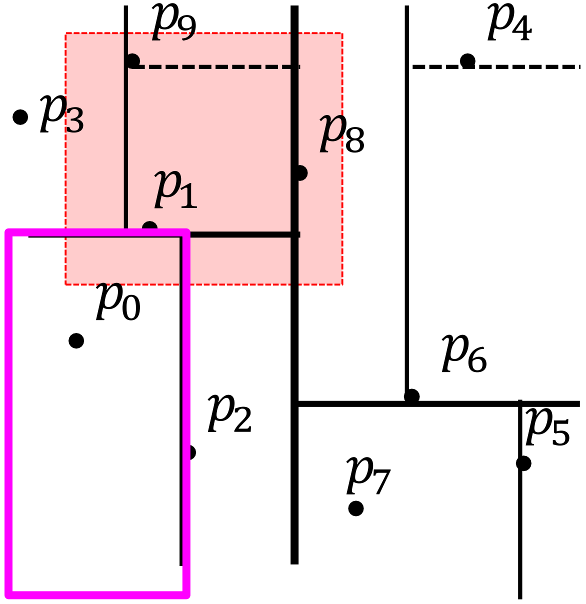
- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

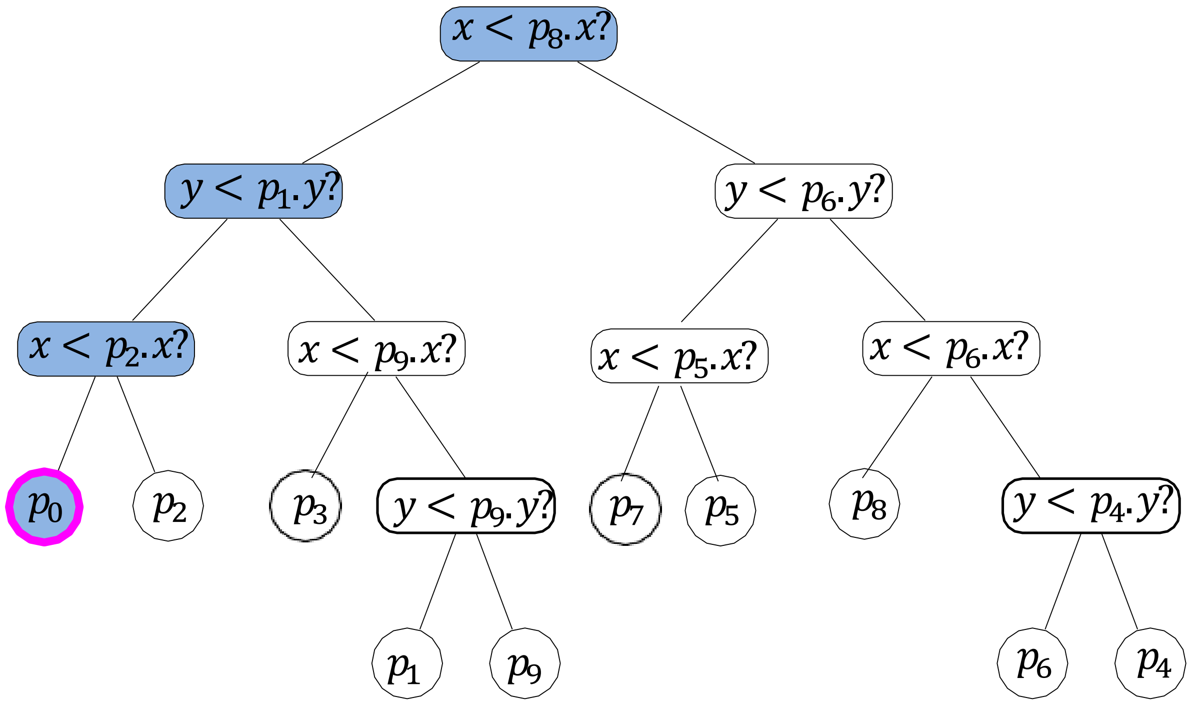
kd-tree: Range Search Example



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

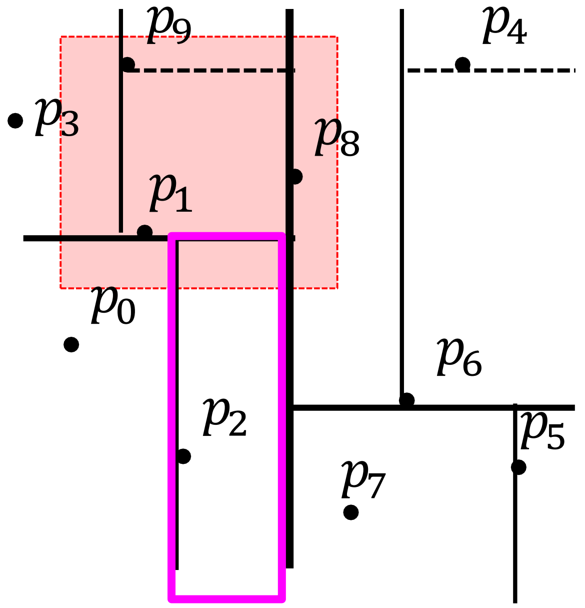
kd-tree: Range Search Example

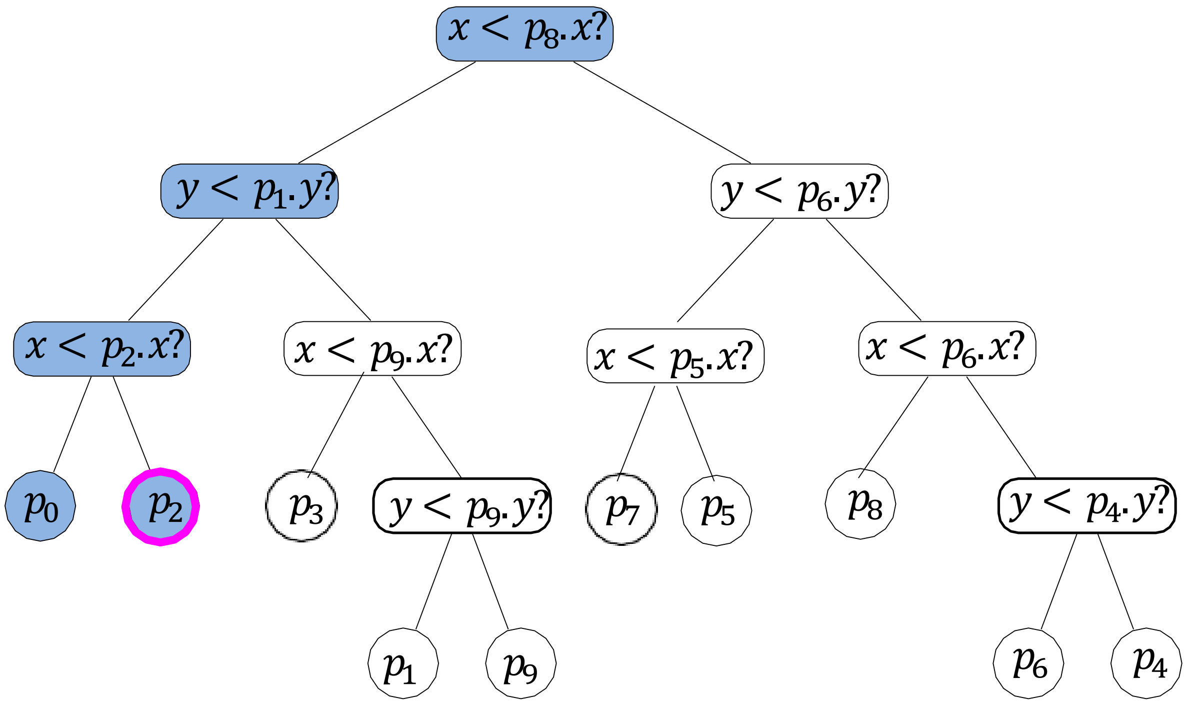




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

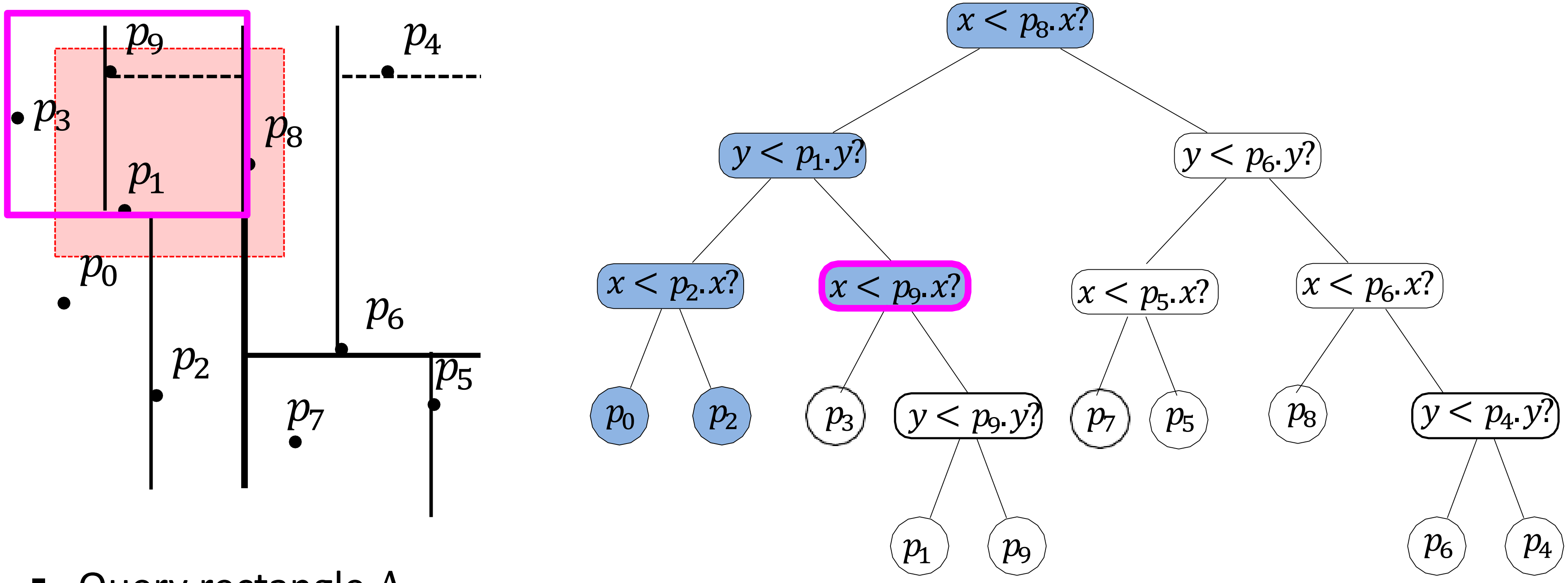
kd-tree: Range Search Example





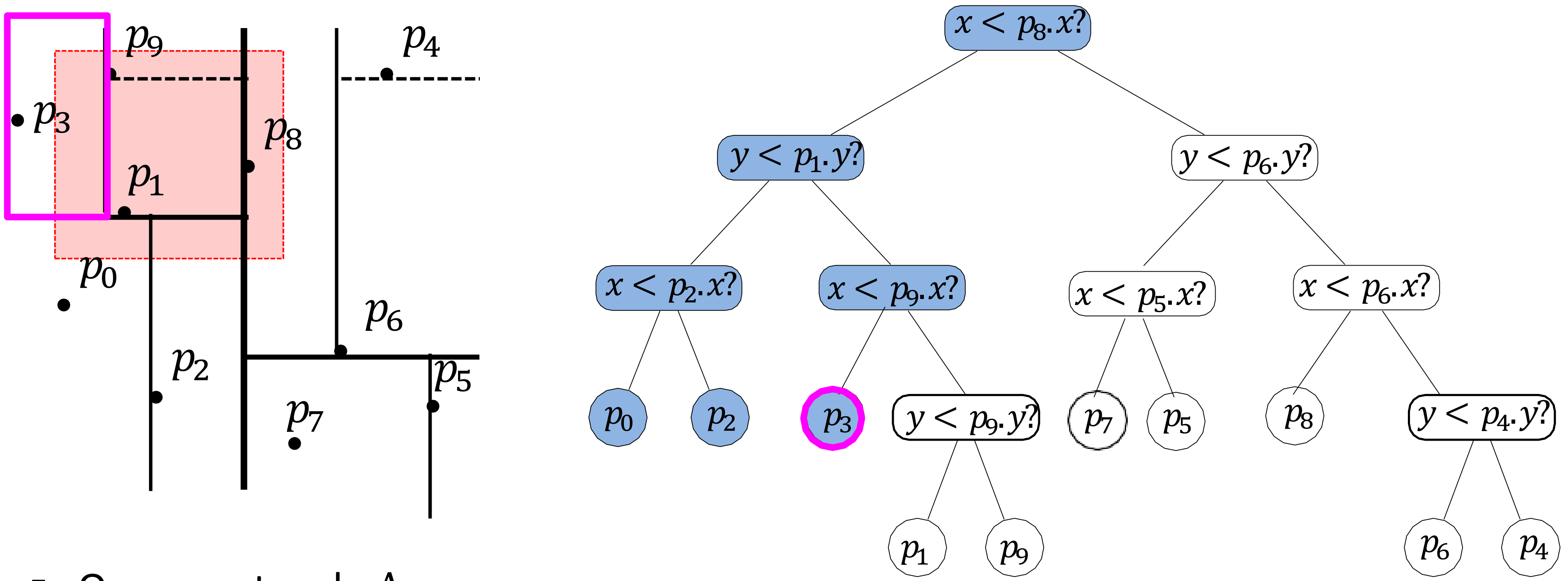
- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

kd-tree: Range Search Example



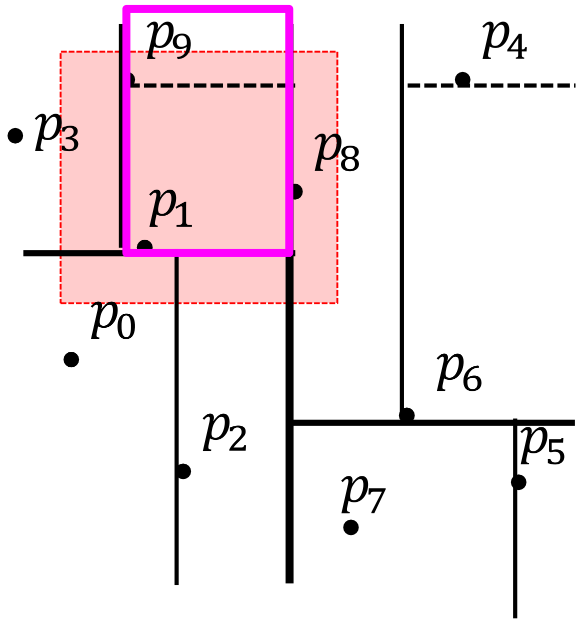
- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

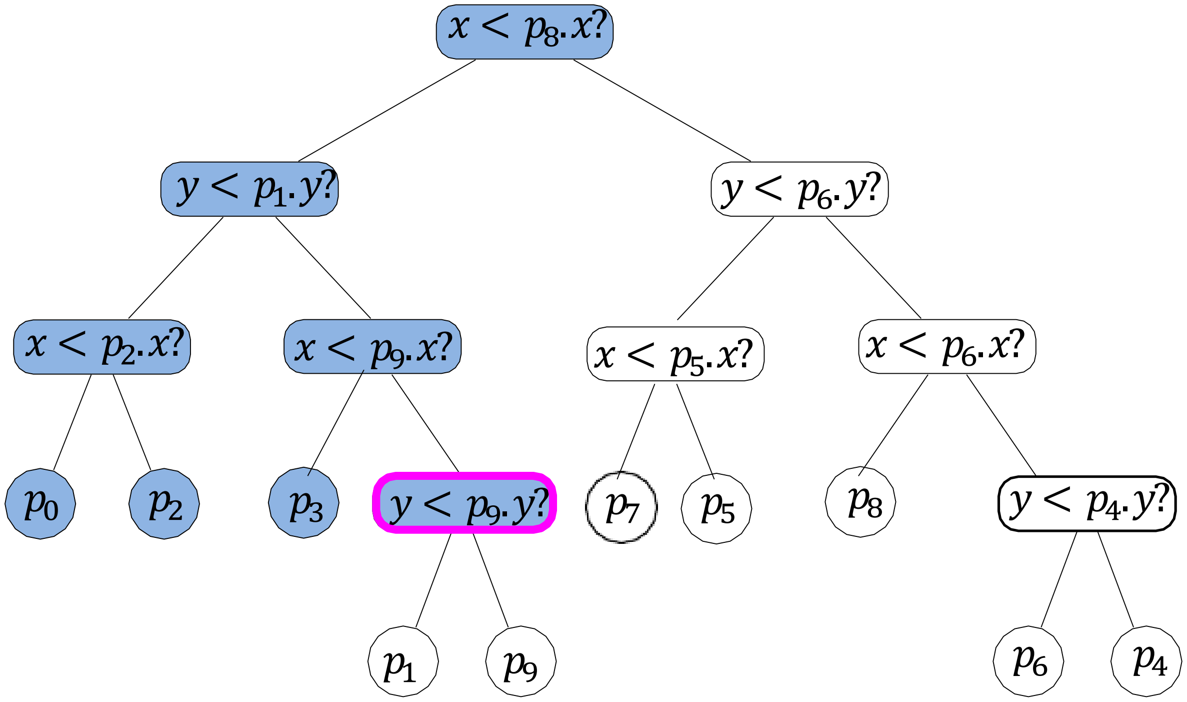
kd-tree: Range Search Example



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

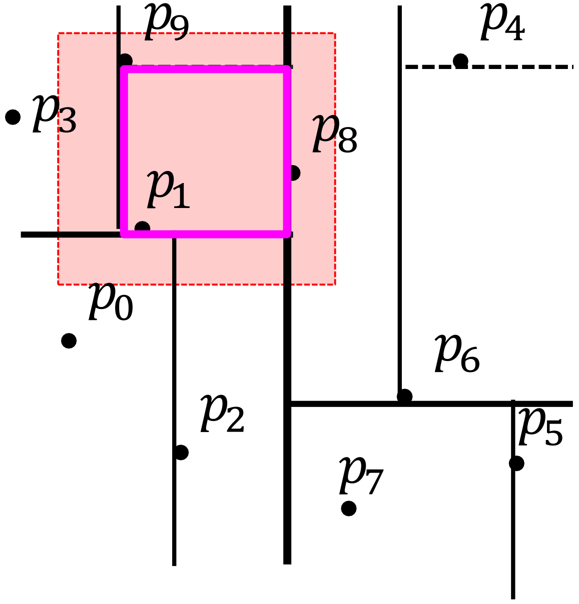
kd-tree: Range Search Example

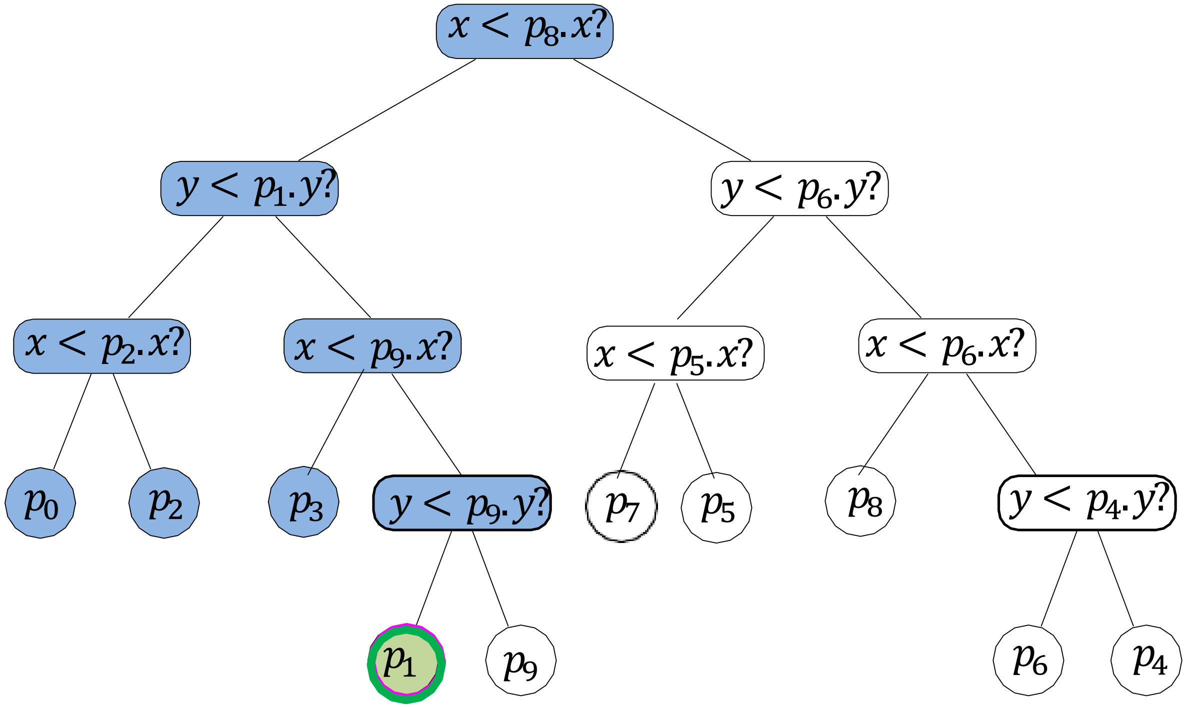




- Query rectangle A
- Let R be region associated with current node, have 3 cases
    1. $R \cap A = \emptyset$: red (outside) node, do not search its children
    2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
    3. $R \cap A \neq \emptyset$: blue (boundary)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

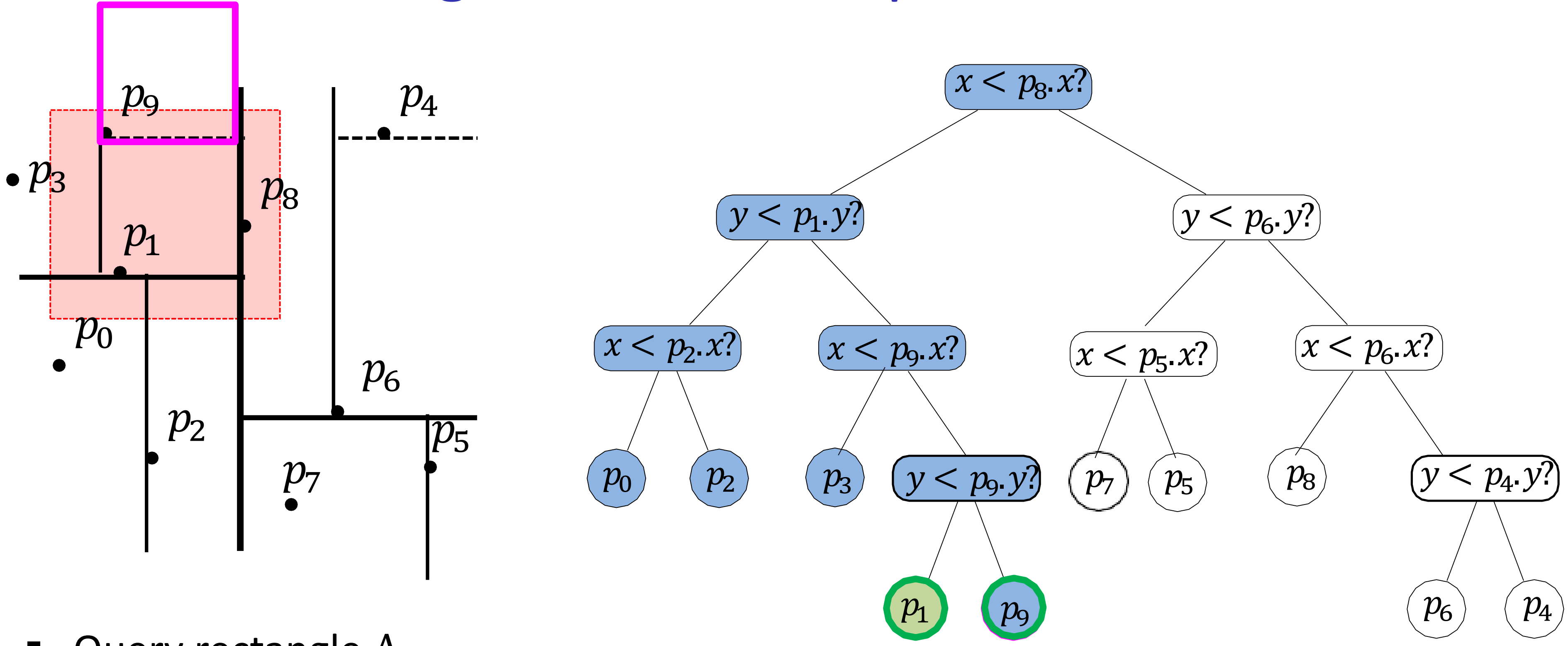
kd-tree: Range Search Example





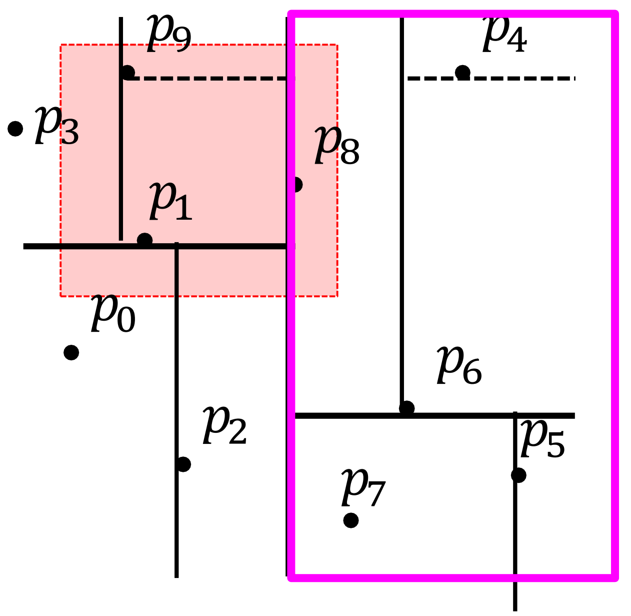
- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

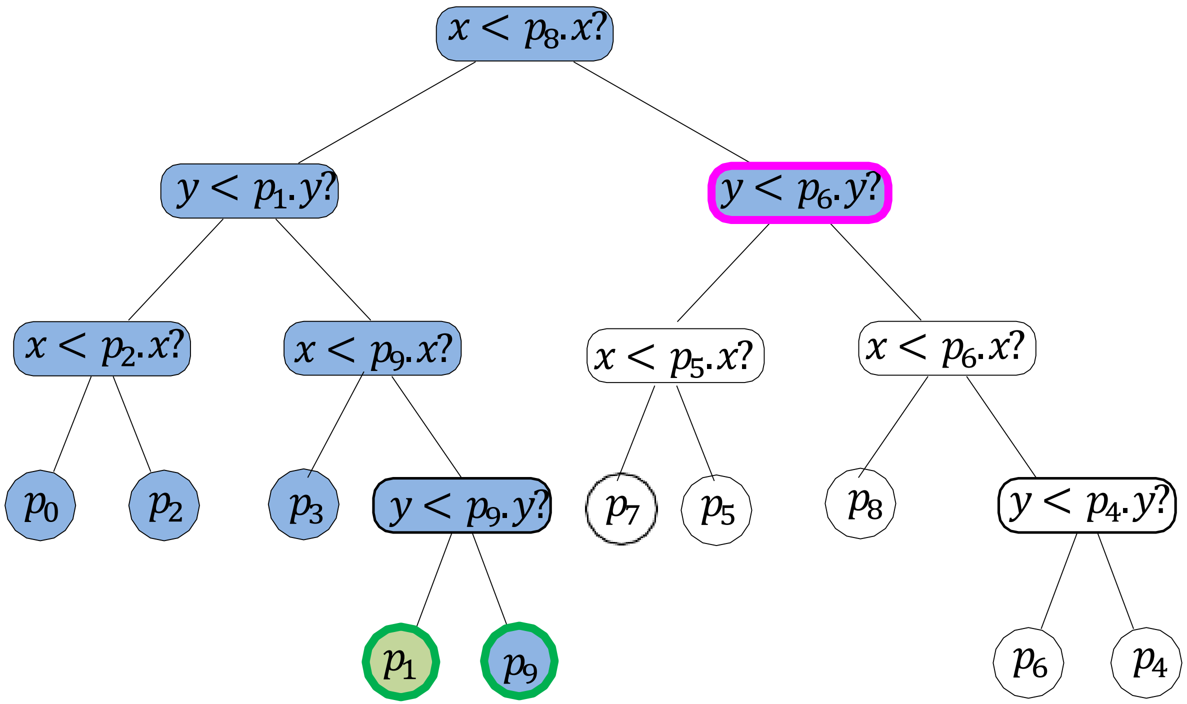
kd-tree: Range Search Example



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

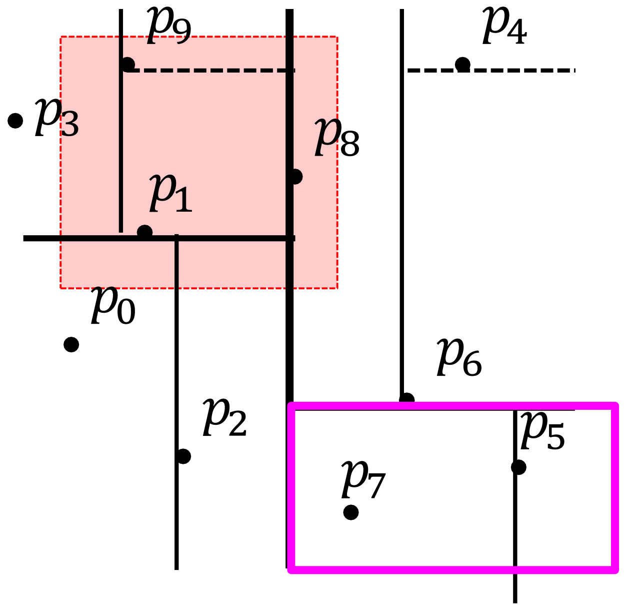
kd-tree: Range Search Example

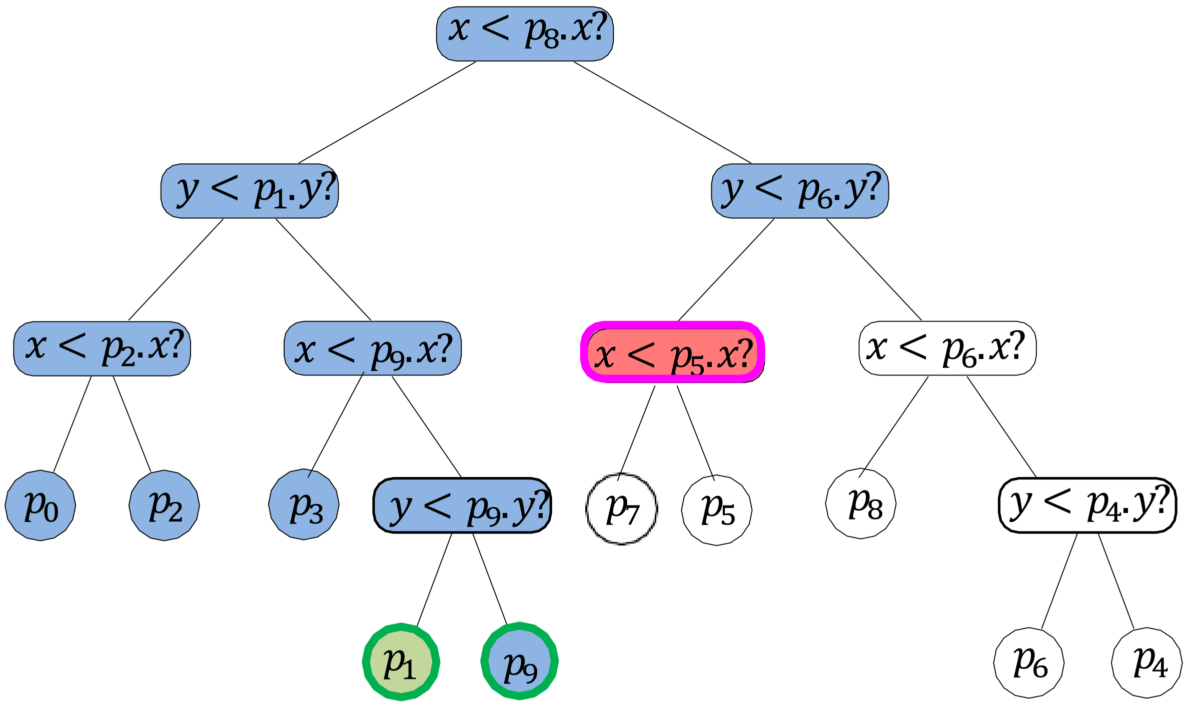




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

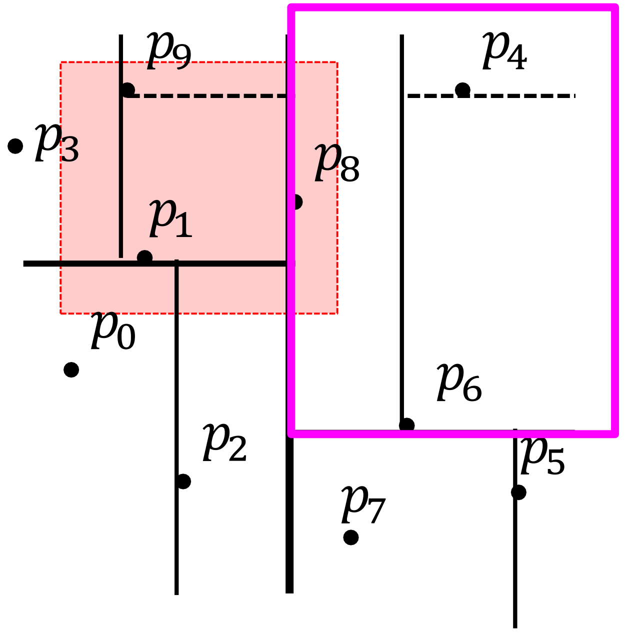
kd-tree: Range Search Example

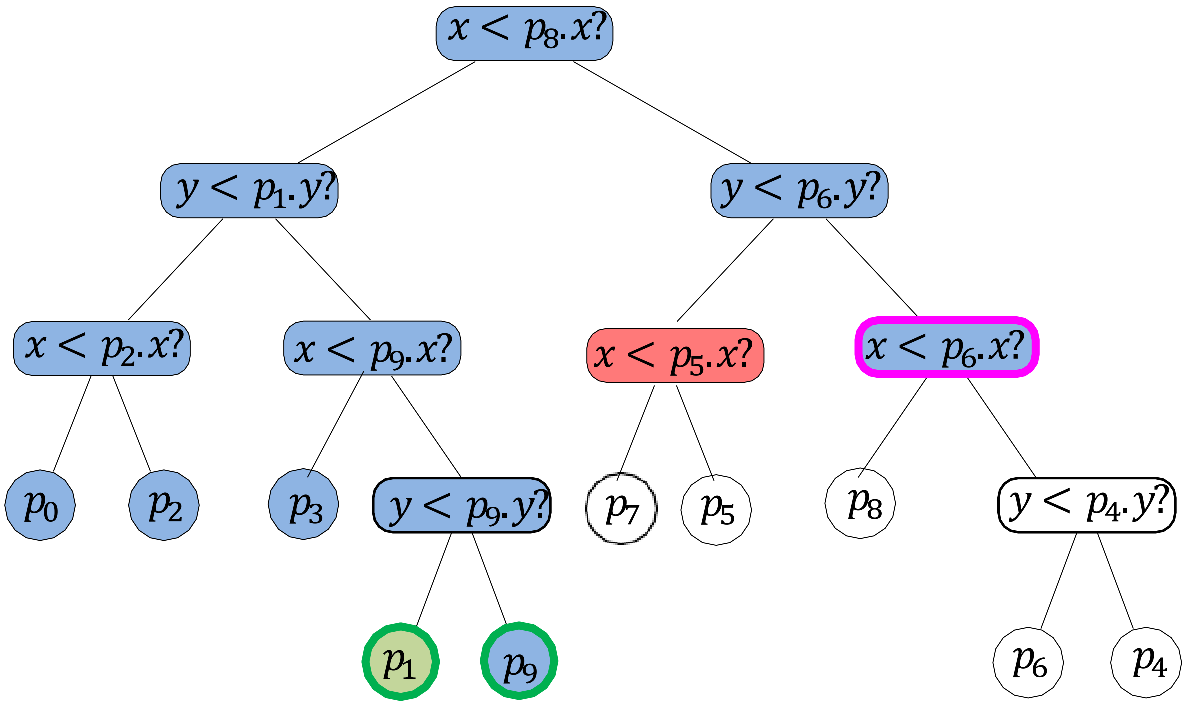




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

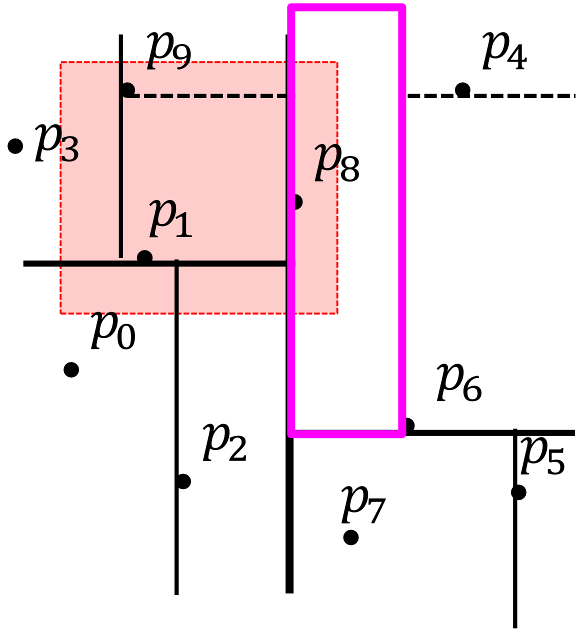
kd-tree: Range Search Example

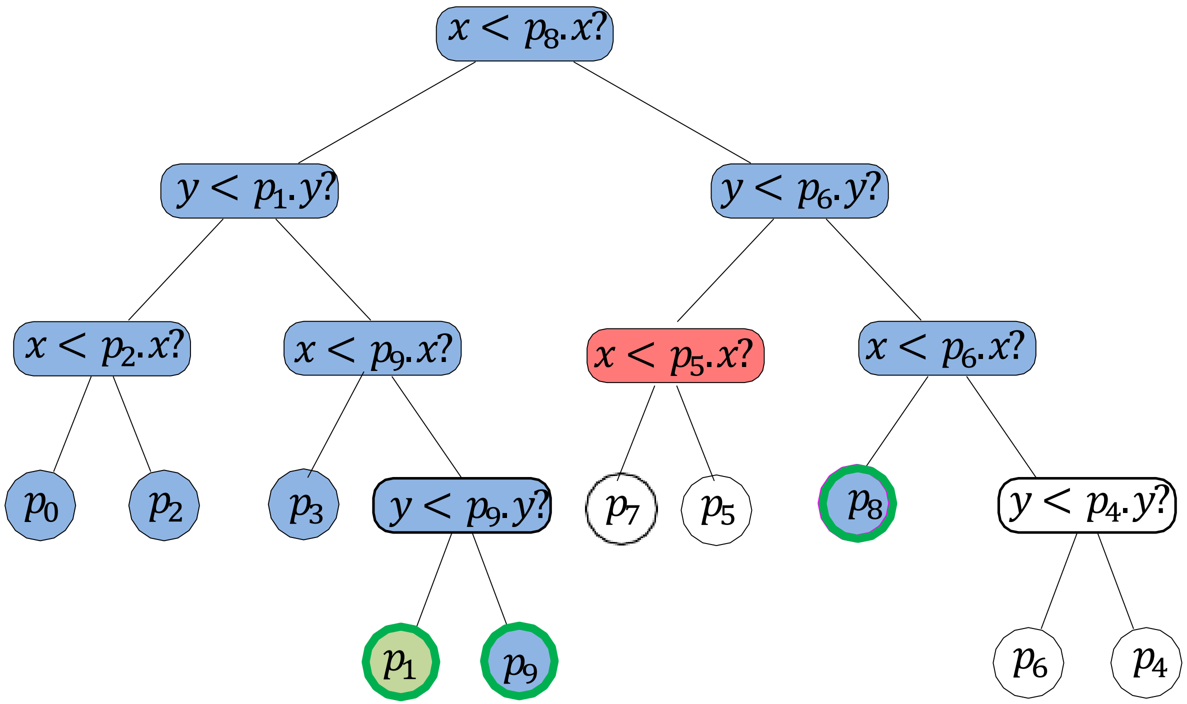




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

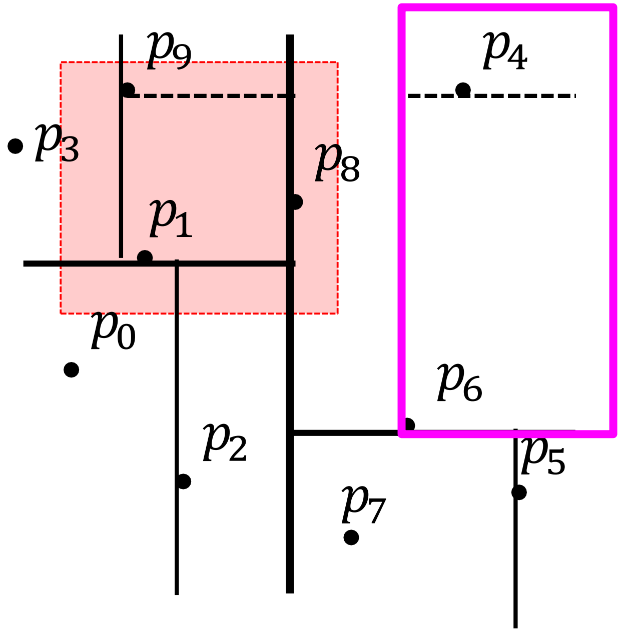
kd-tree: Range Search Example

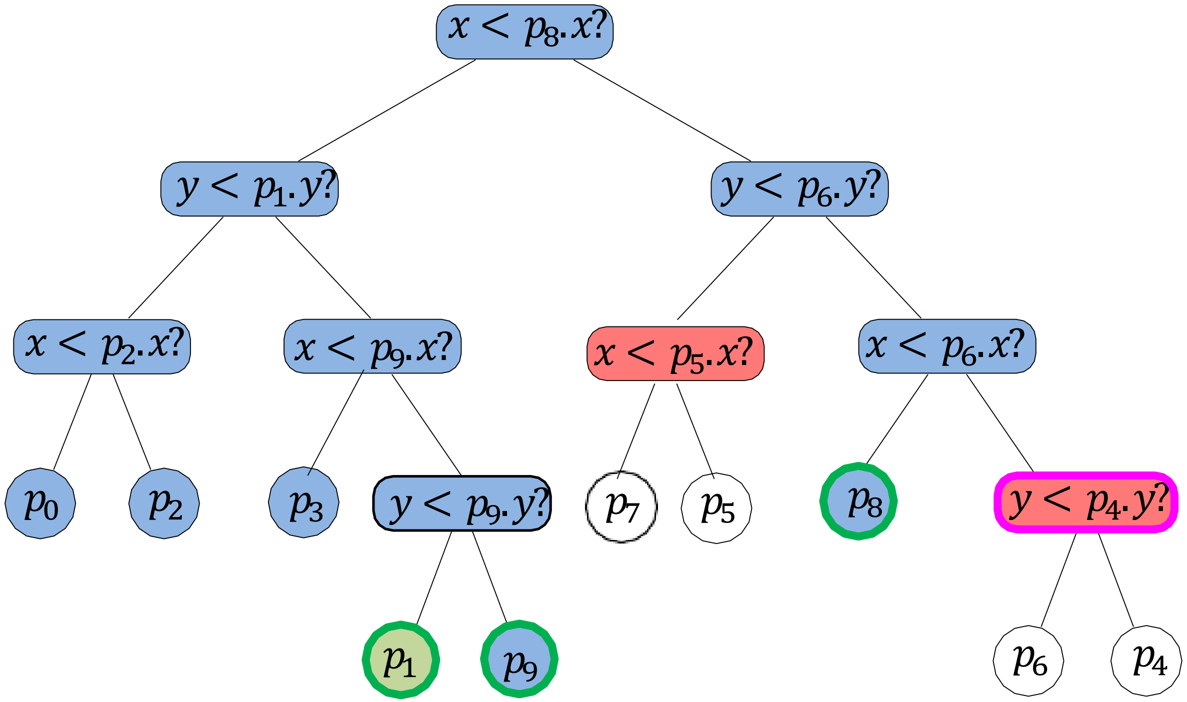




- Query rectangle A
- Let R be region associated with current node, have 3 cases
  1. $R \cap A = \emptyset$: red (outside) node, do not search its children
  2. $R \subseteq A$: green (inside) node, no need to search children, report all points in R
  3. $R \cap A \neq \emptyset$: blue (boundary) node,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

kd-tree: Range Search Example





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

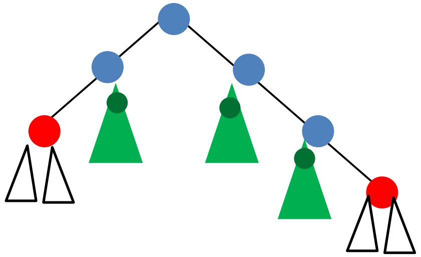
kd-tree Range Search

```
kdTree::RangeSearch($r \leftarrow$ root, $A$)
$r$ : root of kd-tree, $A$: query rectangle
        $R \leftarrow$ region associated with node $r$
        if $R \subseteq A$ then
                report all points below $r$
                return
        if $R \cap A = \emptyset$ then return
        if $r$ is a leaf then
              $p \leftarrow$ point stored at $r$
              if $p \in A$ return $p$
              else return
        for each child $v$ of $r$ do
              kdTree::RangeSearch($v$, $A$)
```

- 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

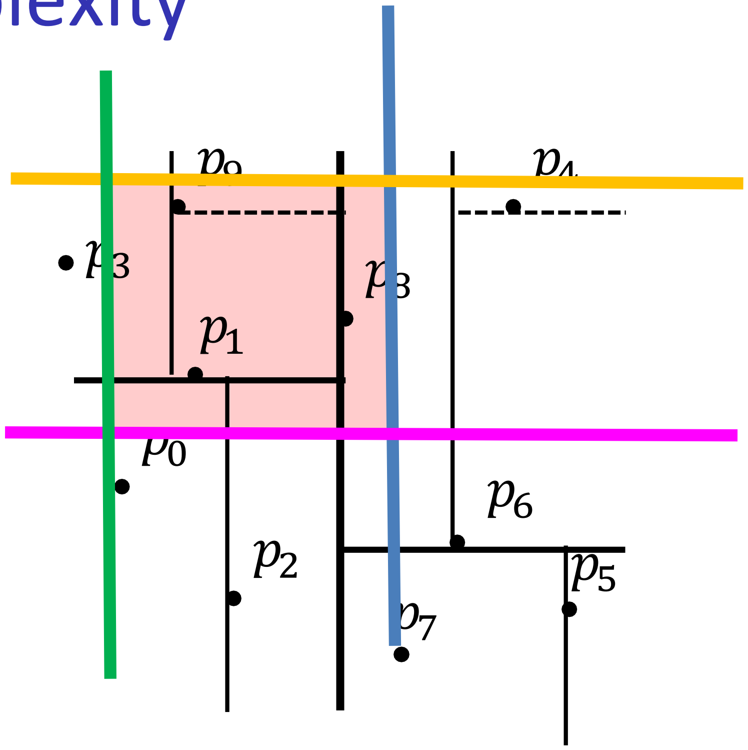
kd-tree: Range Search Complexity

- We visit blue and red nodes and also green nodes
    - at each blue, red and topmost green node do a constant amount of work
    - for each topmost green node v, report points stored at leaves in the subtree rooted at v
        - each node has 2 children → number of internal nodes is less than number of leaves for any subtree
        - at most s leaves over all green subtrees, at most $2s$ nodes over all green subtrees, $O(s)$ work to report points stored in green subtrees
    - topmost green nodes + red nodes $\le 2 \cdot$ blue nodes
        - each topmost green and red node has a blue parent
    - for running time, enough to count blue nodes and add $O(s)$



- Let $Q(n)$ is the number of blue nodes visited
    - neither $R \cap A = \emptyset$ nor $R \subseteq A$
    - these are regions that intersect A but not completely inside A
- Can show that $Q(n)$ satisfies $Q(n) \le 2Q\left(\frac{n}{4}\right) + O(1)$
    - resolves to $Q(n) \in O(\sqrt{n})$
- Therefore, running time of range search is $O(s + \sqrt{n})$

kd-tree: Range Search Complexity

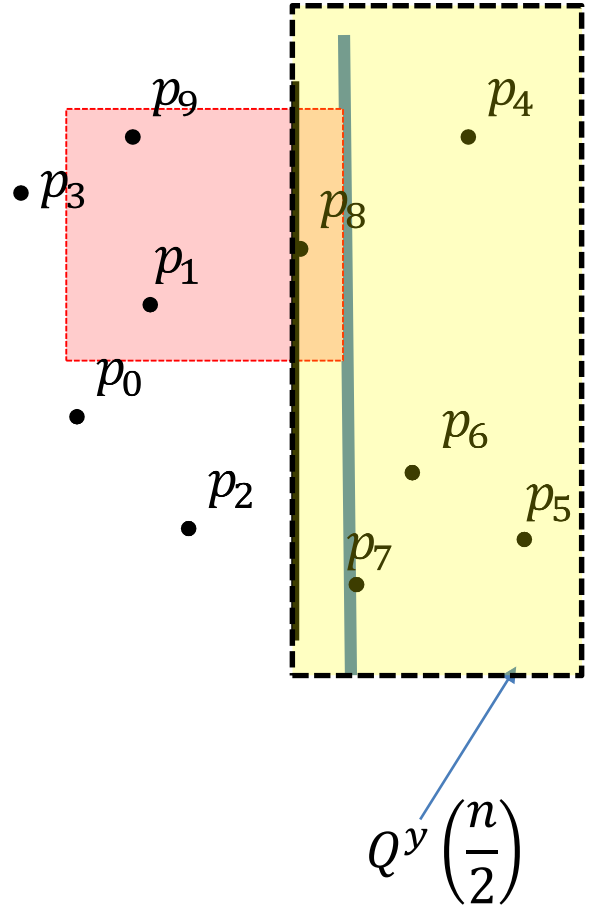


- search rectangle A
- $Q(n) =$ # regions intersecting A but not completely inside A
- $Q(n) \leq$ # regions intersecting 🟦
  $+$ # regions intersecting 🟩
  $+$ # regions intersecting 🟪
  $+$ # regions intersecting 🟨
- Will look at # regions intersecting 🟦
- Other cases are handled similarly

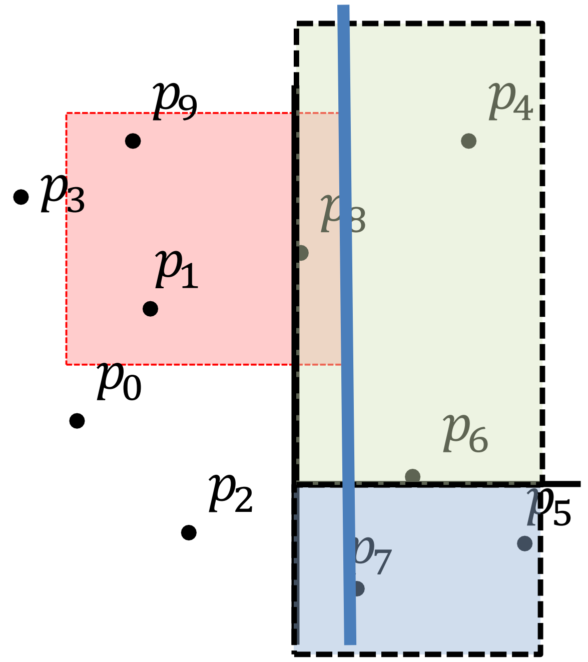
kd-tree: Range Search Complexity

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

- $Q^x(n) = 1 + Q^y\left(\frac{n}{2}\right)$
    - 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



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

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

- Next, $Q^y\left(\frac{n}{2}\right) = 1 + 2Q^x\left(\frac{n}{4}\right)$
    - 1 for the root region
    - root region is split in 2 by horizontal line
    - | can intersect both of these regions

- Combining, get recurrence $Q^x(n) = 2 + 2Q^x\left(\frac{n}{4}\right)$

- Resolves to $Q^x(n) \in O(\sqrt{n}\,)$

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

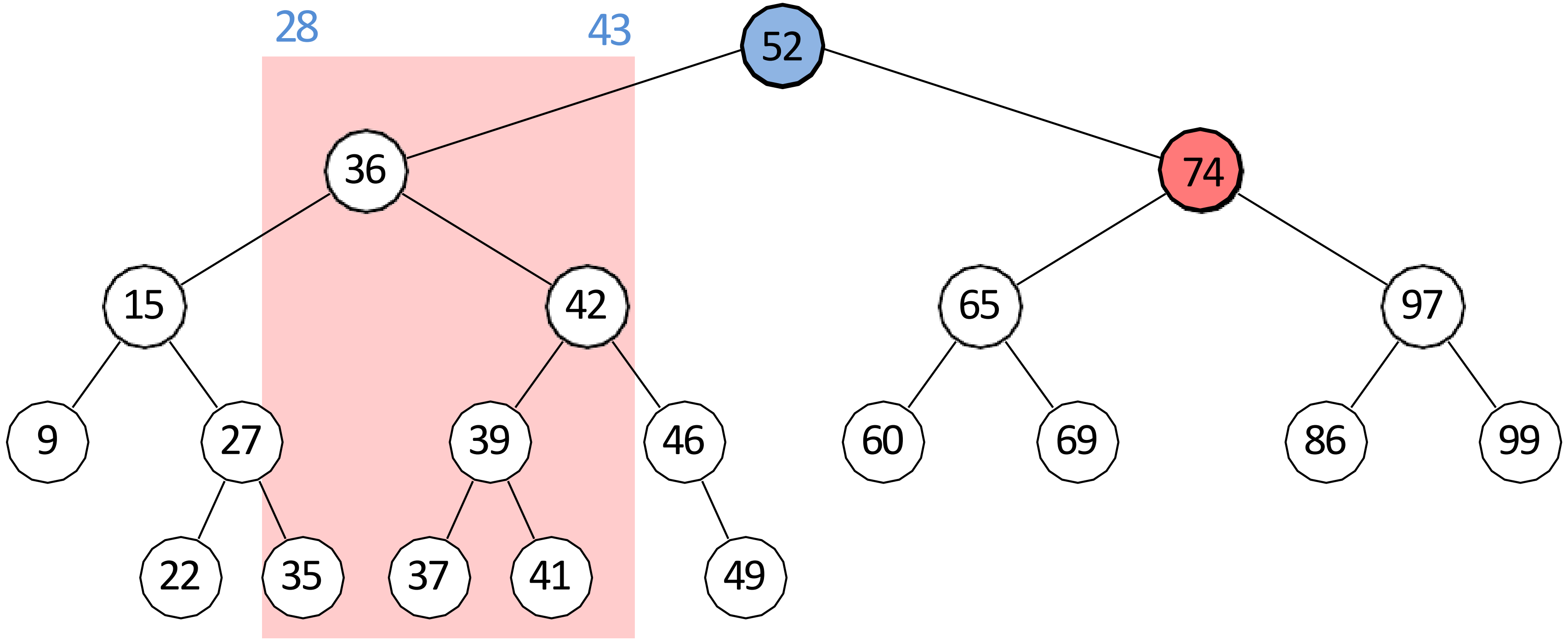
- Range-Searching in Dictionaries for Points
  - Range Search
  - Multi-Dimensional Data
  - Quadtrees
  - kd-Trees
  - Range Trees
  - Conclusion

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

BST Range Search example

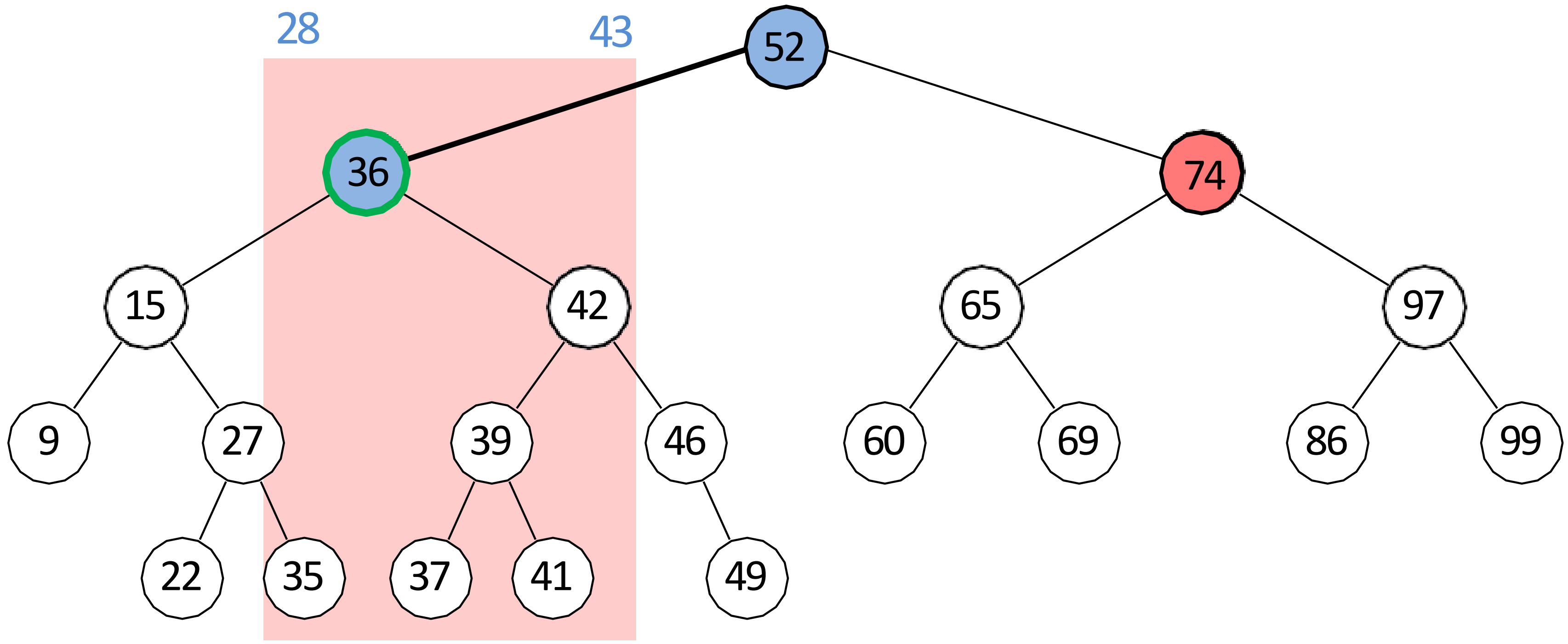
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

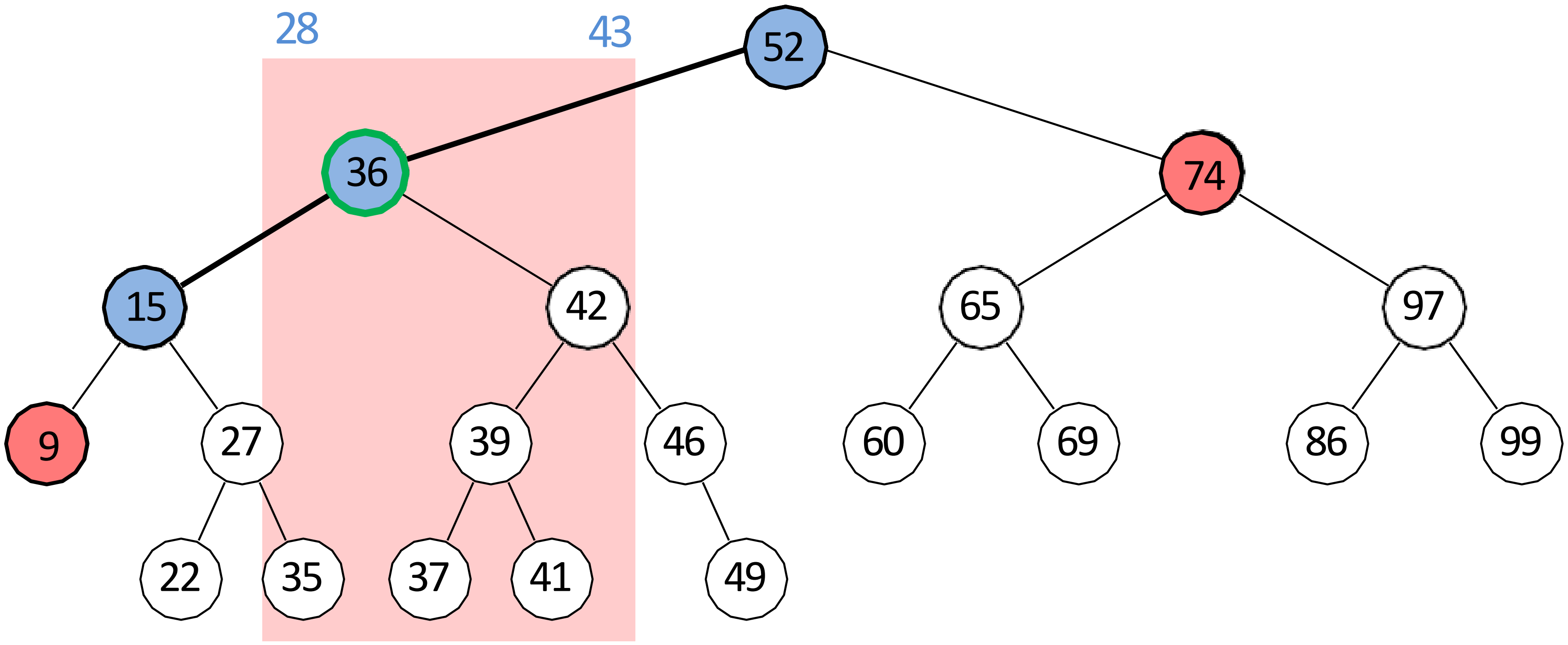
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

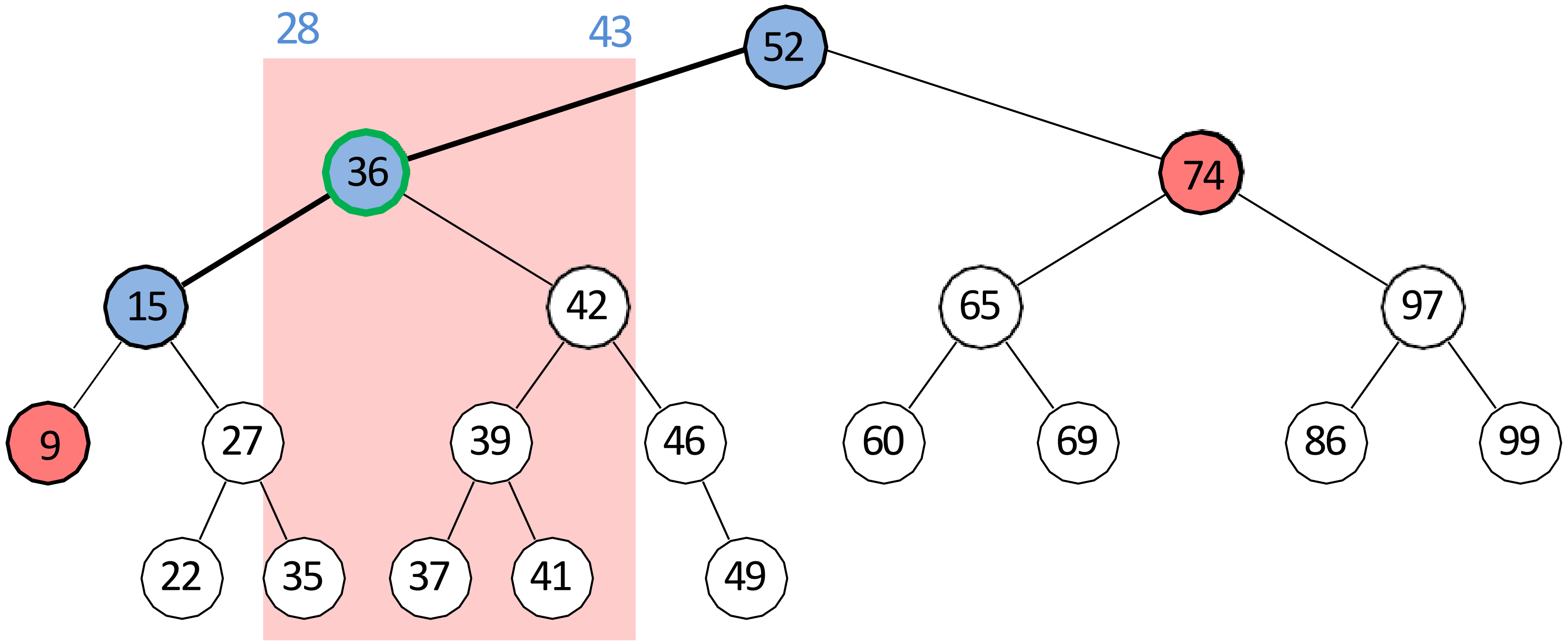
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

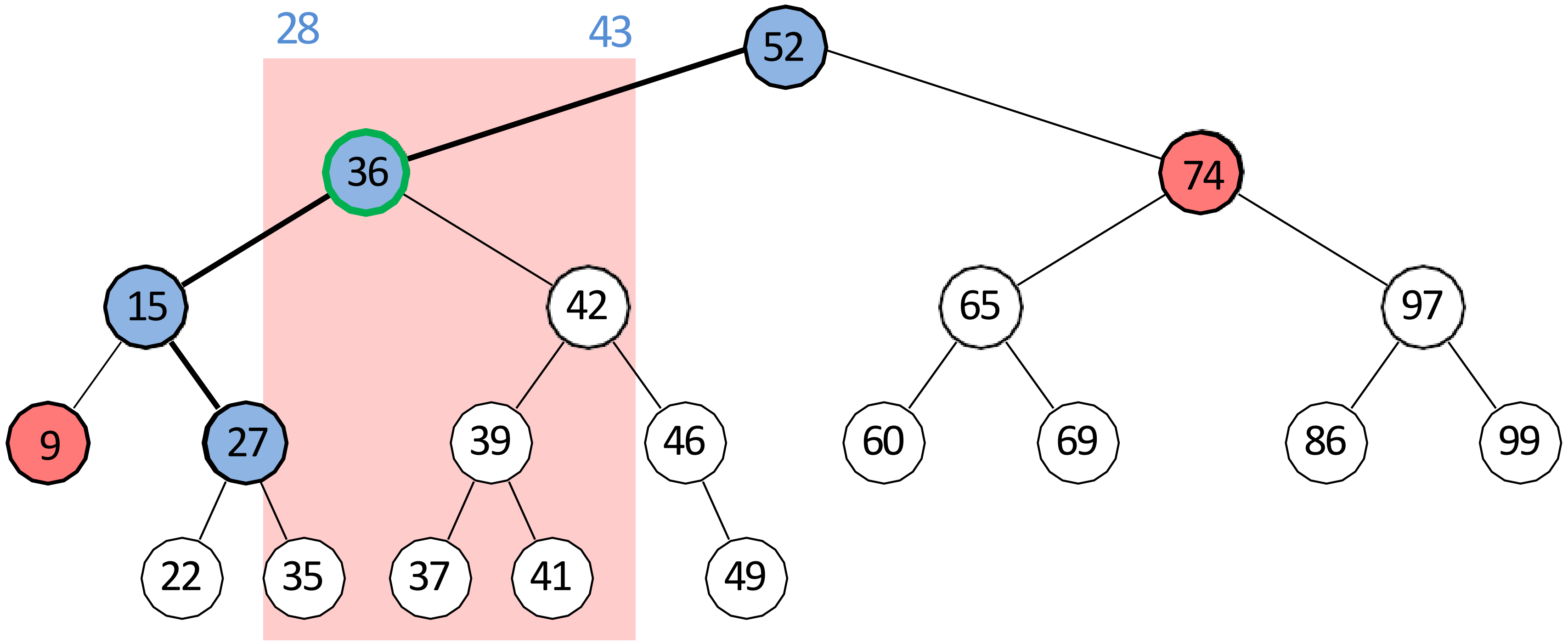
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

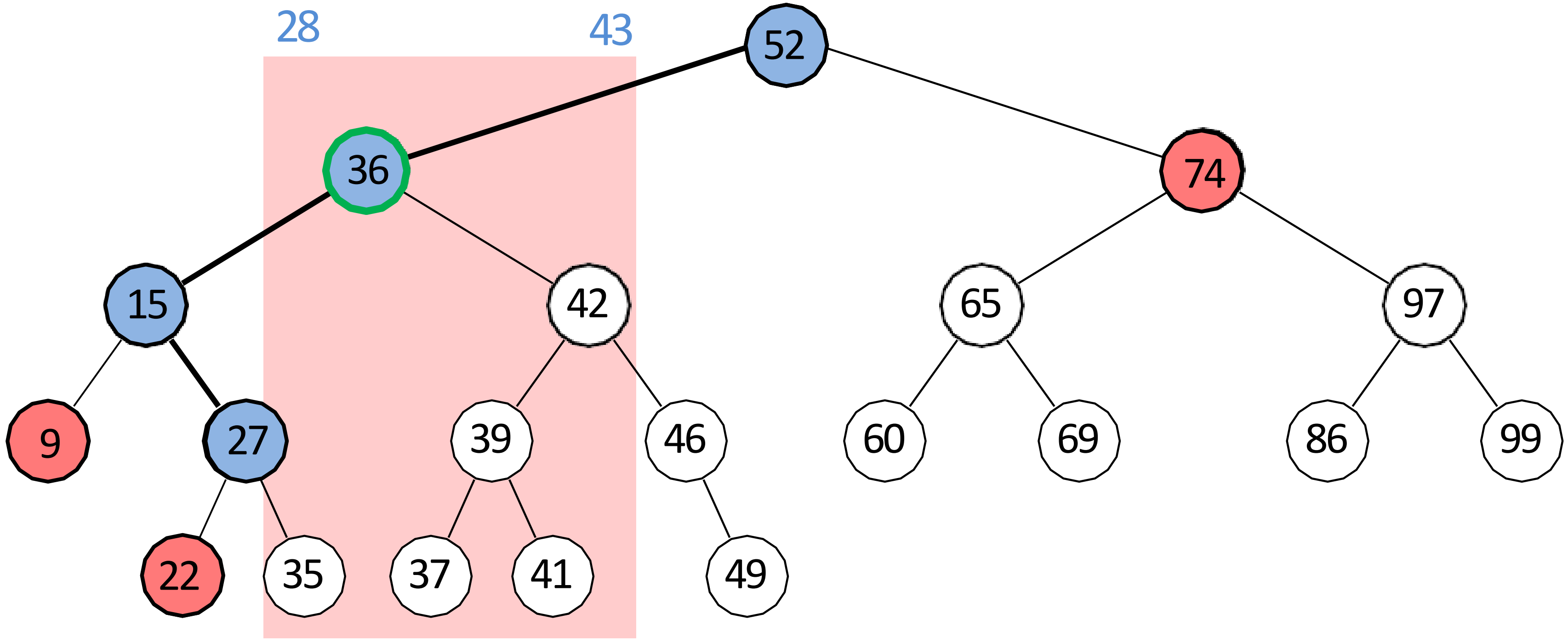
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

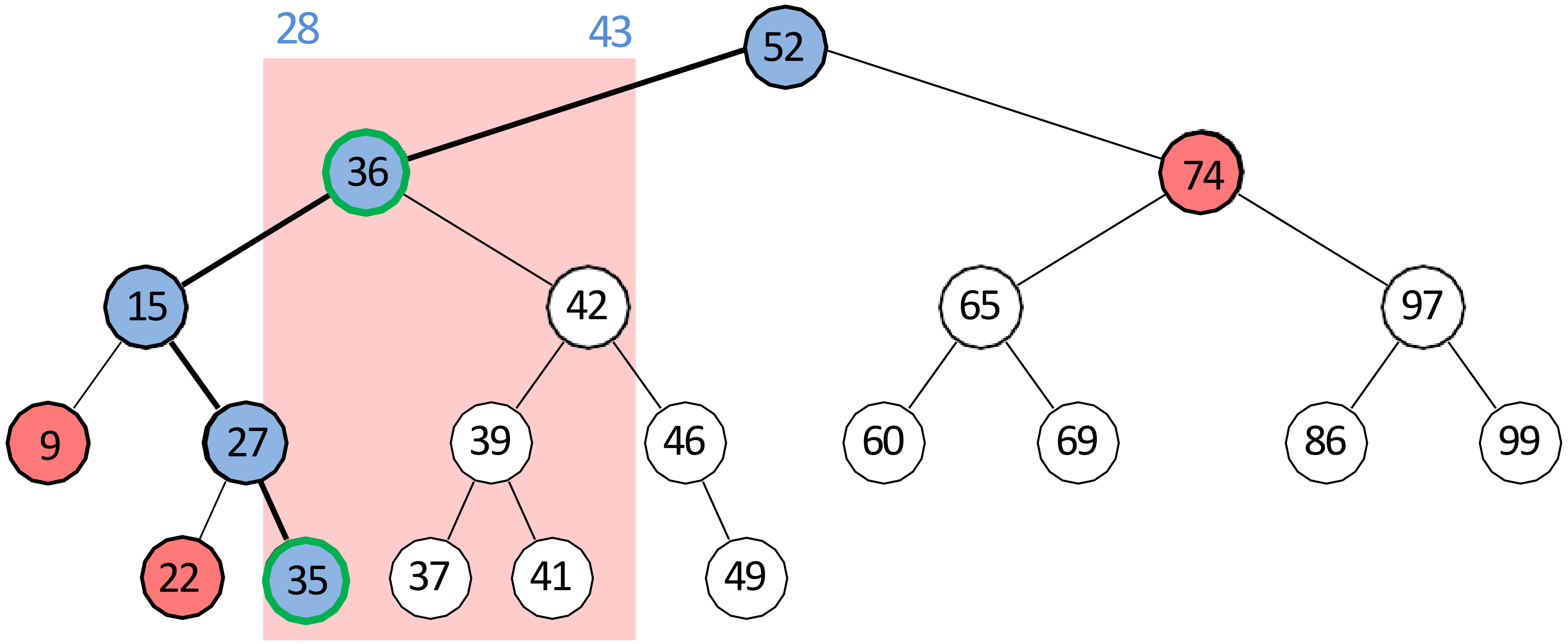
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

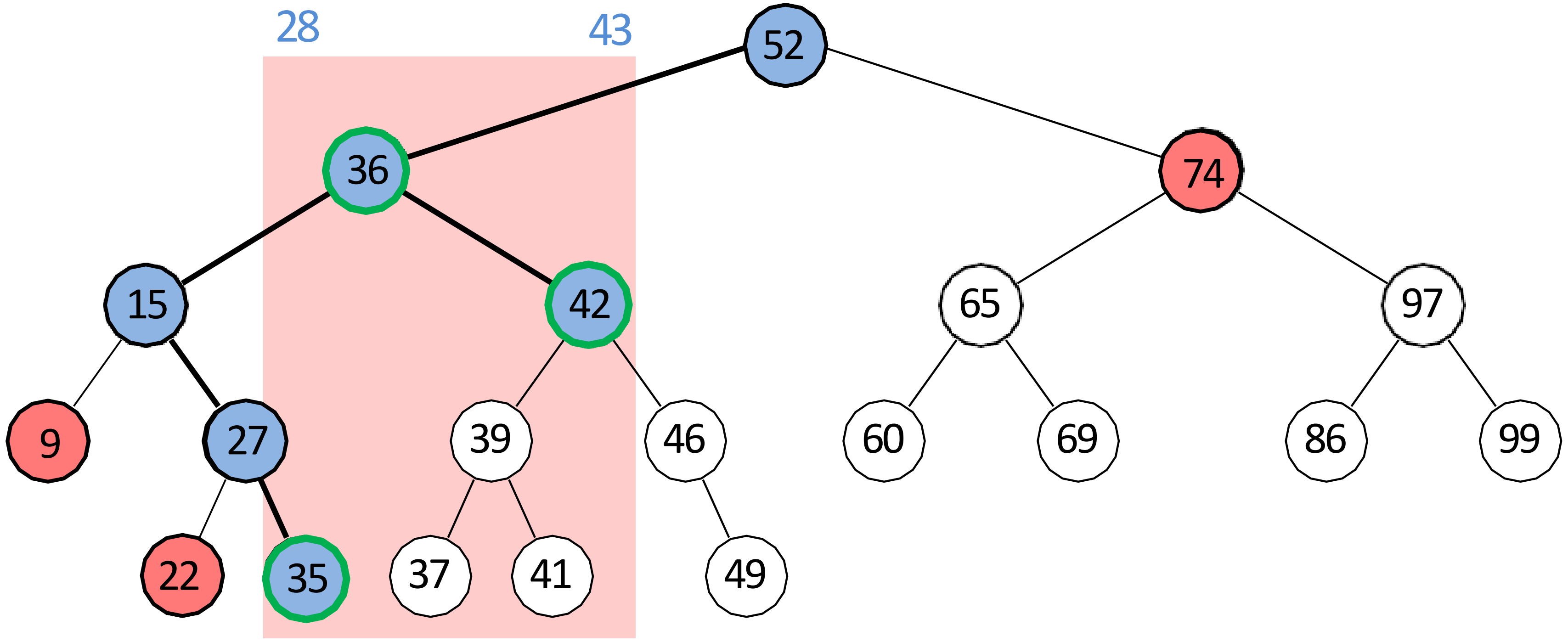
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

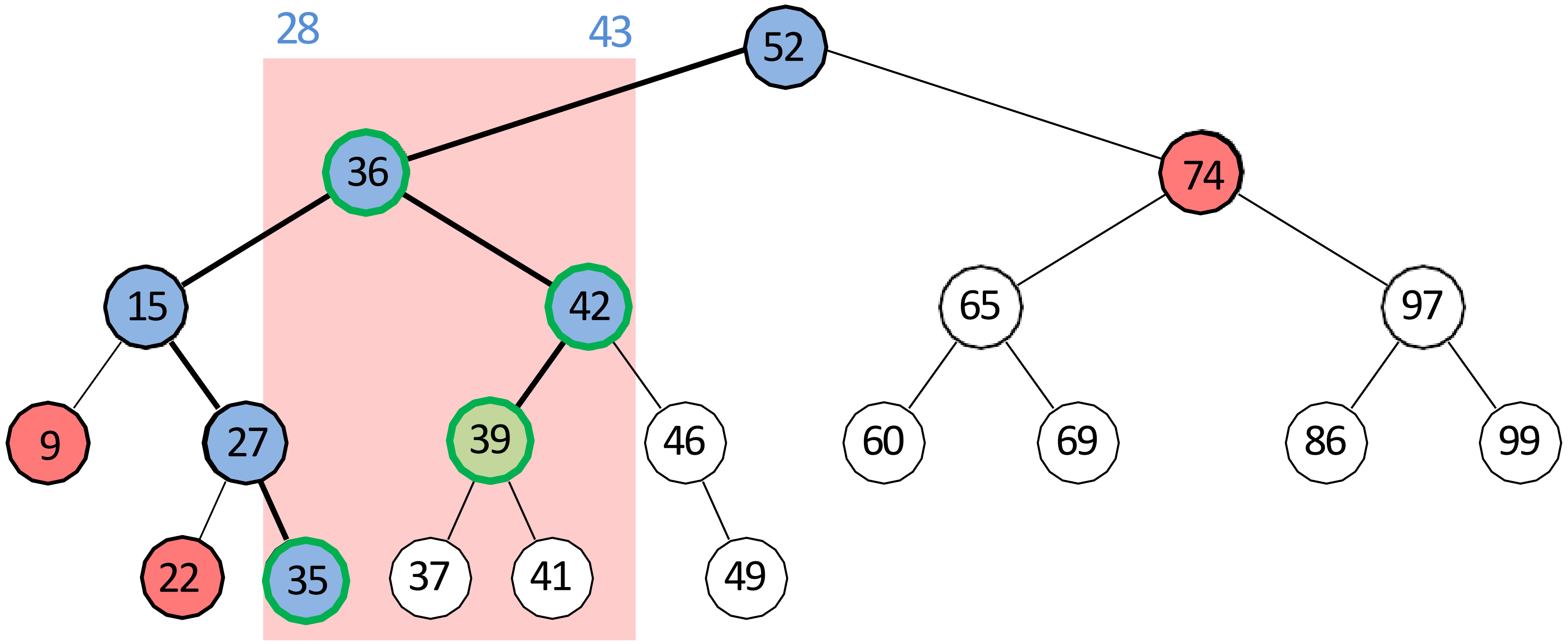
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

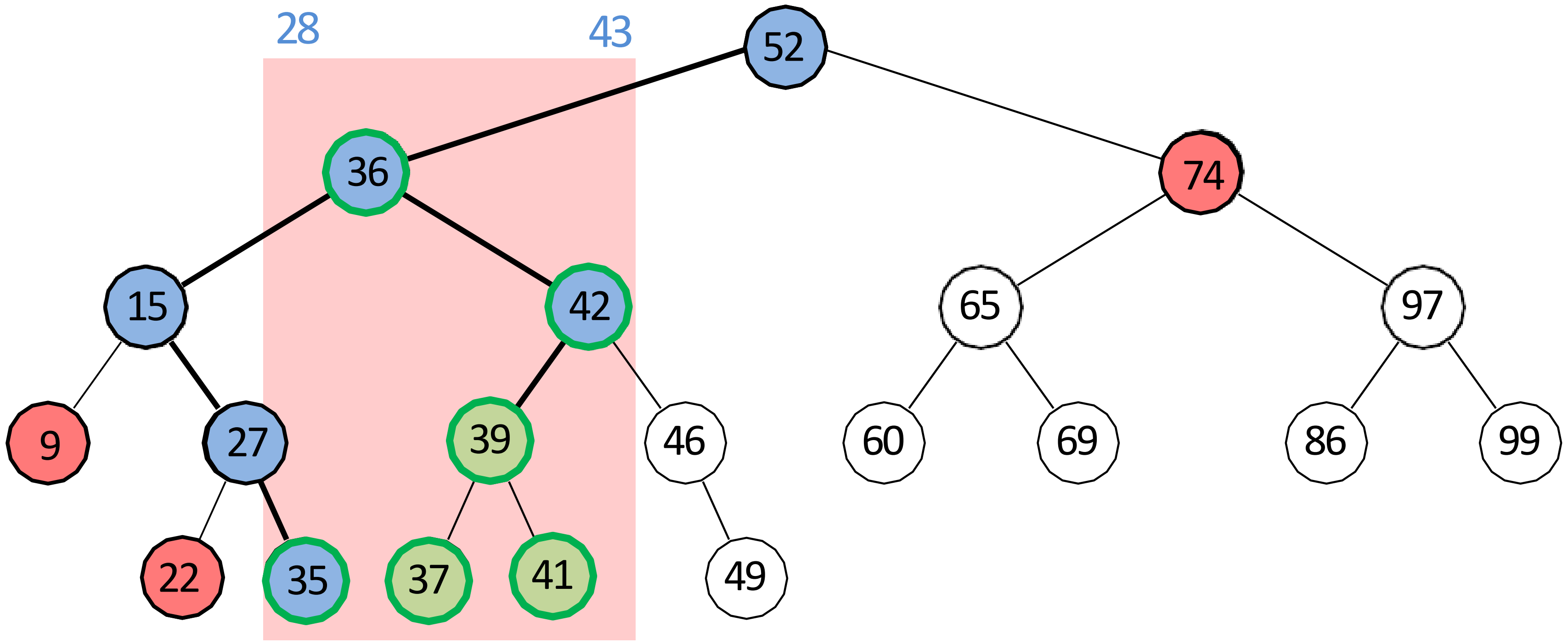
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

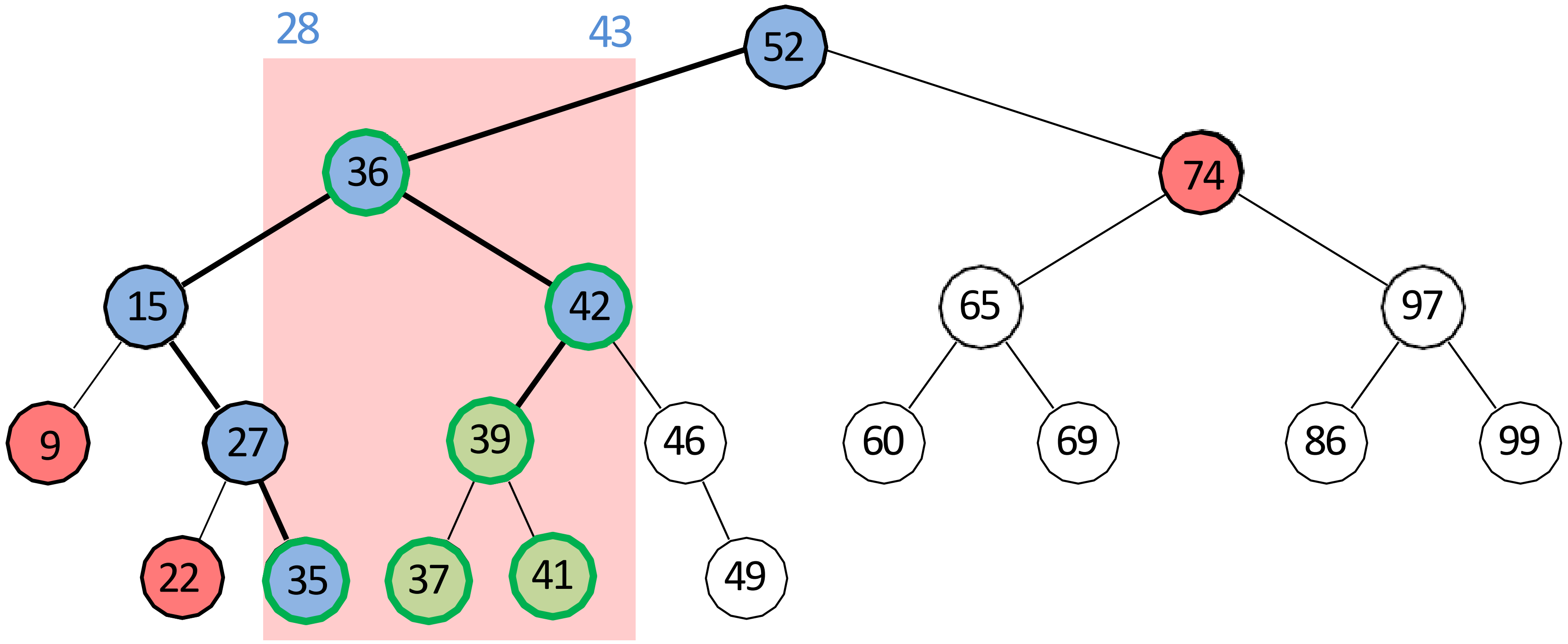
BST::RangeSearch-recursive$(T, 28, 43)$



- 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

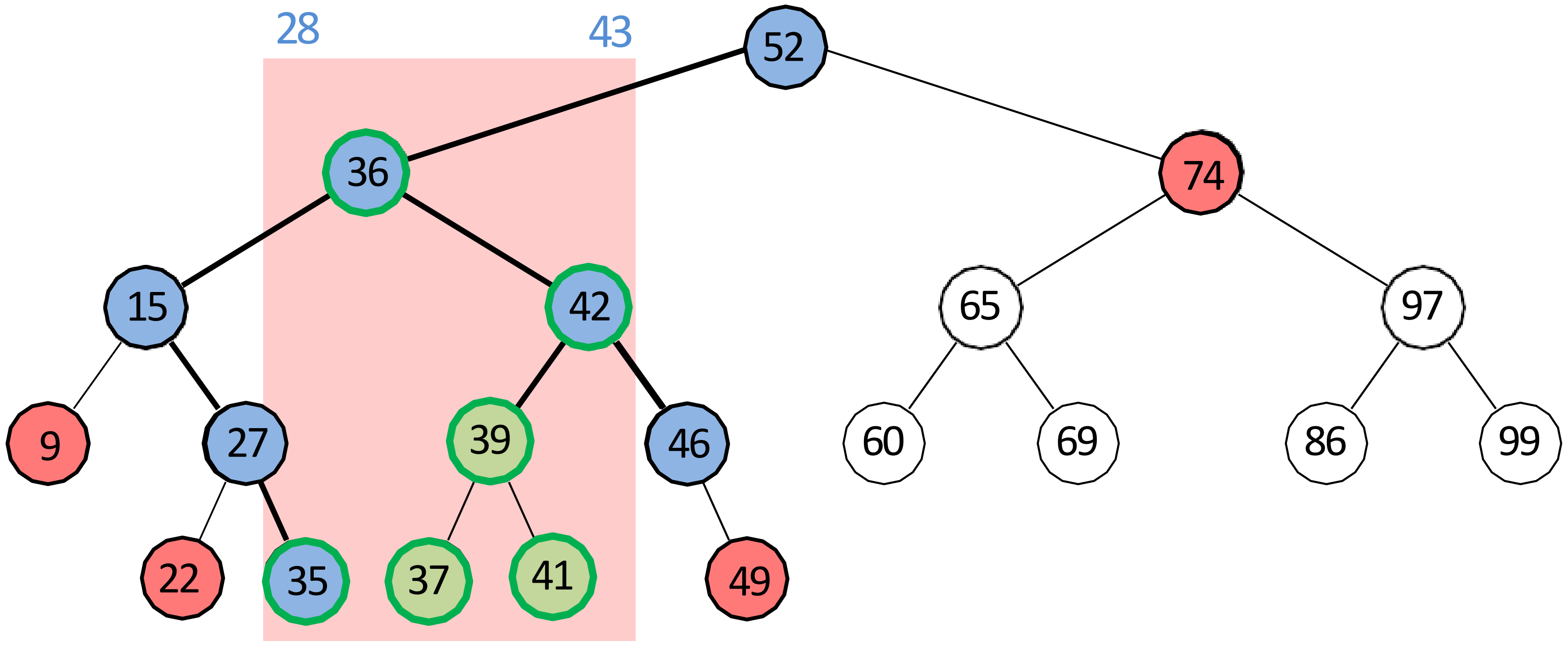
BST::RangeSearch-recursive$(T, 28, 43)$



- 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$(T, 28, 43)$



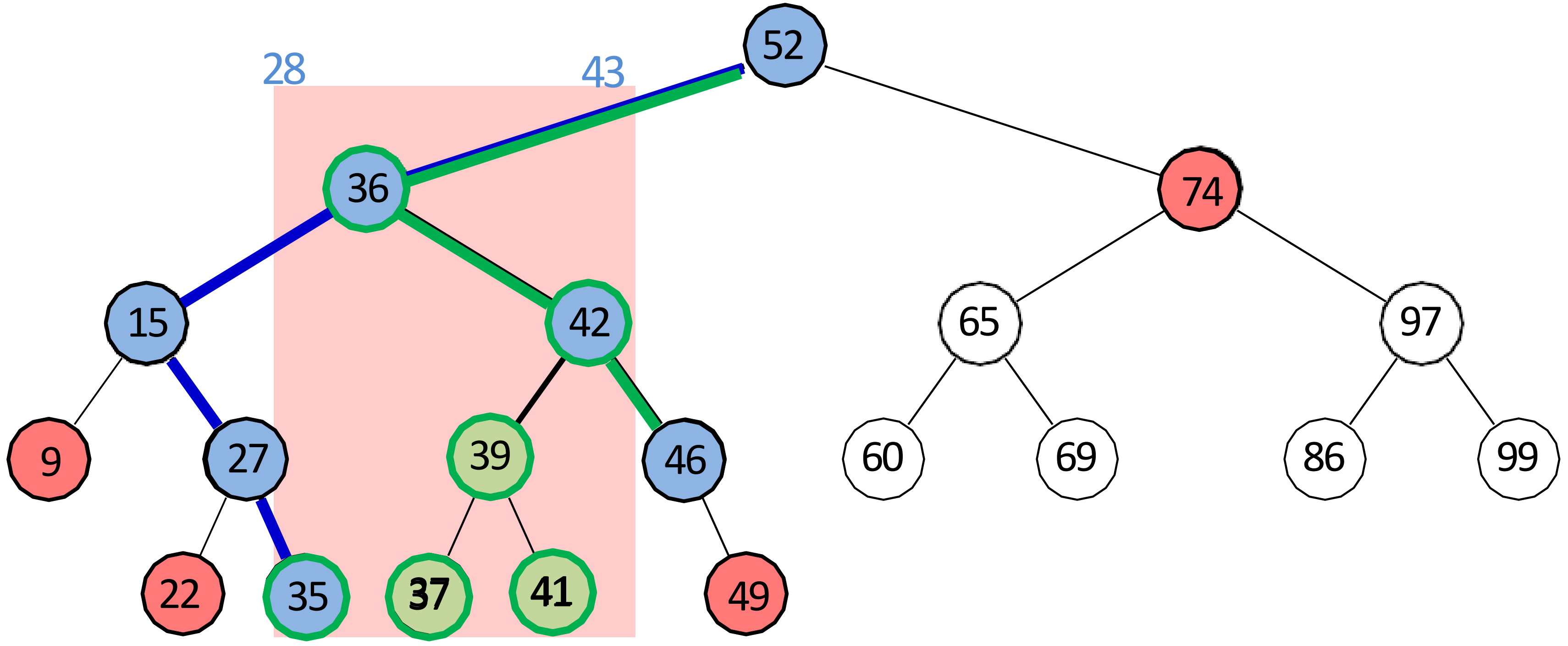
- blue node: recurse either to the left, or to the right, or both (according to the key value)
  - boundary node, one or both subtrees may intersect range query
- red node: range search was not called on red node, but was called on its parent
  - outside node, subtree does not intersect range query
-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

BST Range Search

```
*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** $r = NIL$  **then return**
 **if** $k_1 \leq r.key \leq k_2$ **then**
        $L \leftarrow$ *BST::RangeSearch-recursive*$(r.left, k_1, k_2)$
        $R \leftarrow$ *BST::RangeSearch-recursive*$(l.right, k_1, k_2)$
        **return** $L \cup \{r.key\} \cup R$
 **if** $r.key < k_1$ **then**
        return *BST::RangeSearch-recusive*$(r.right, k_1, k_2)$
 **if** $r.key > k_2$ **then**
        return *BST-RangeSearch-recursive*$(r.left, k_1, k_2)$
```

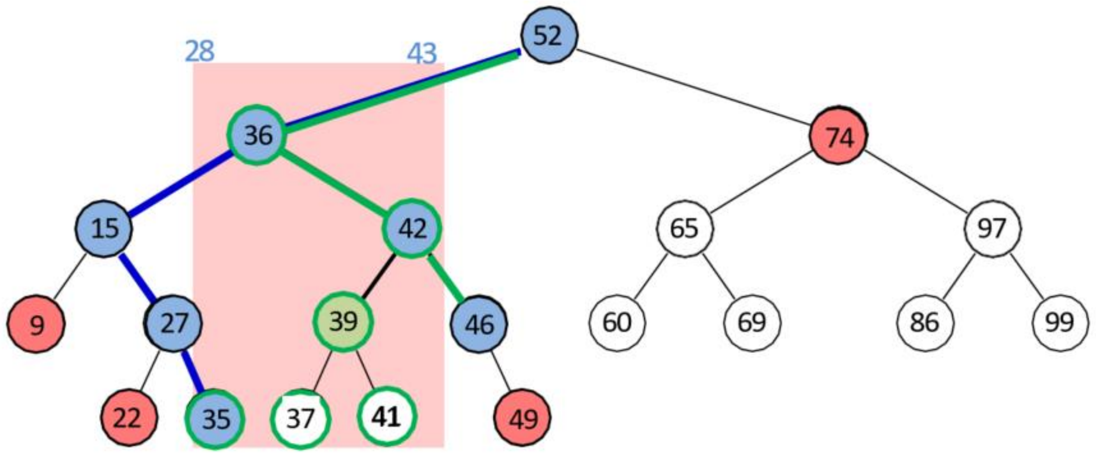
- Keys returned in sorted order

Modified BST Range Search



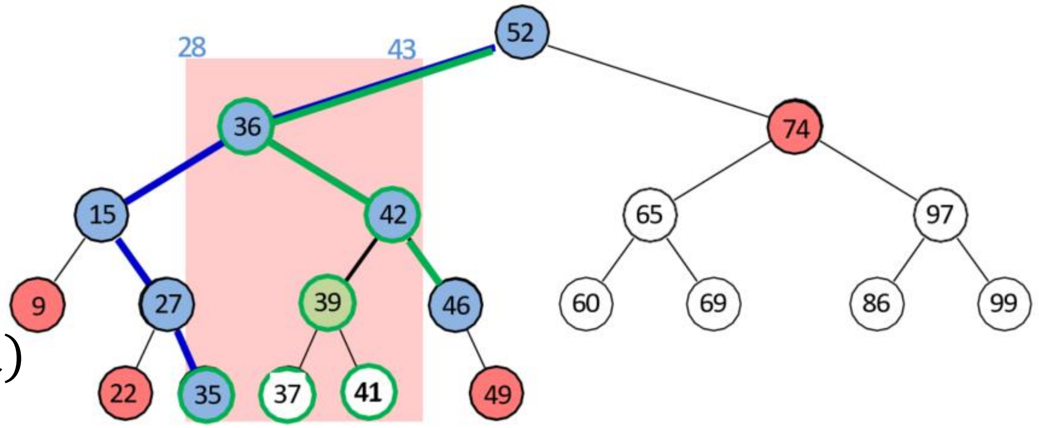
- Search for left boundary k_1 : this gives path ***P*₁**
- Search for right boundary k_2 : this gives path ***P*₂**
- Boundary (blue nodes) are exactly all the nodes on paths ***P*₁** and ***P*₂**
- Nodes are partitioned into three groups: boundary, outside, inside

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 will be efficient when we move to 2D search in ***range trees***

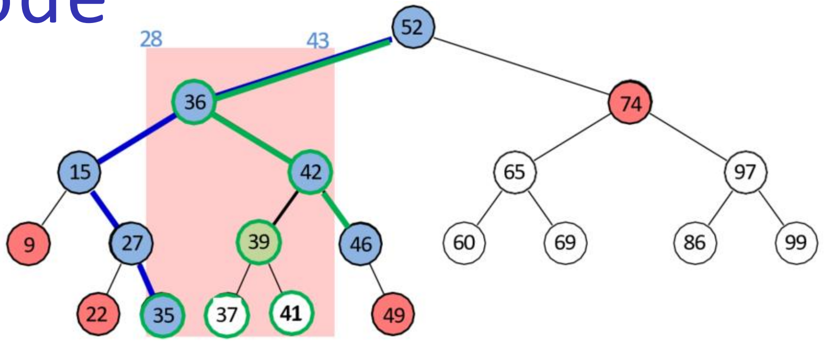
Modified BST Range Search Analysis

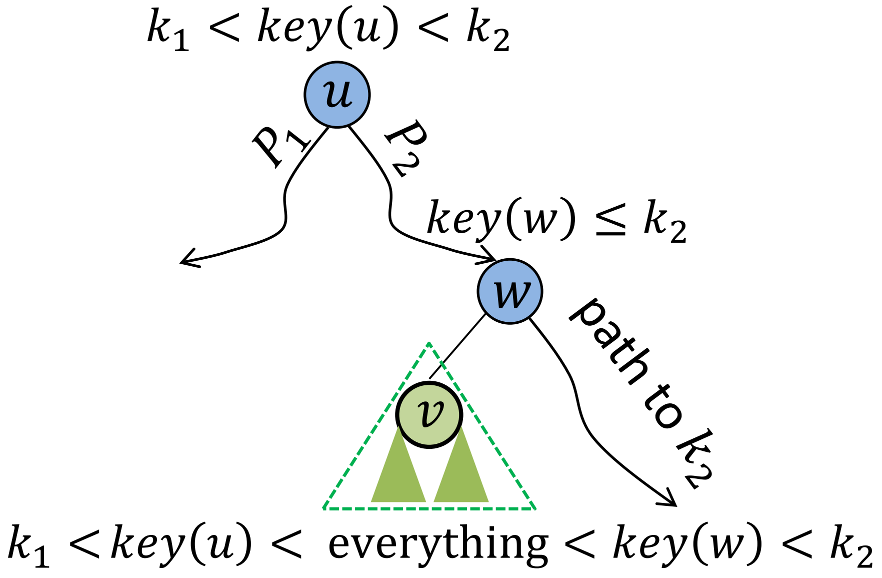


- Assume balanced BST
- Running time consists of
  1. search for path ***P*₁**
     - $O(\log n)$
  2. search for path ***P*₂** is $O(\log n)$
     - $O(\log n)$
  3. check if boundary nodes in the range
     - $O(1)$ at each boundary node, there are $O(\log n)$ of them, $O(\log n)$ total time
  4. spend $O(1)$ 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

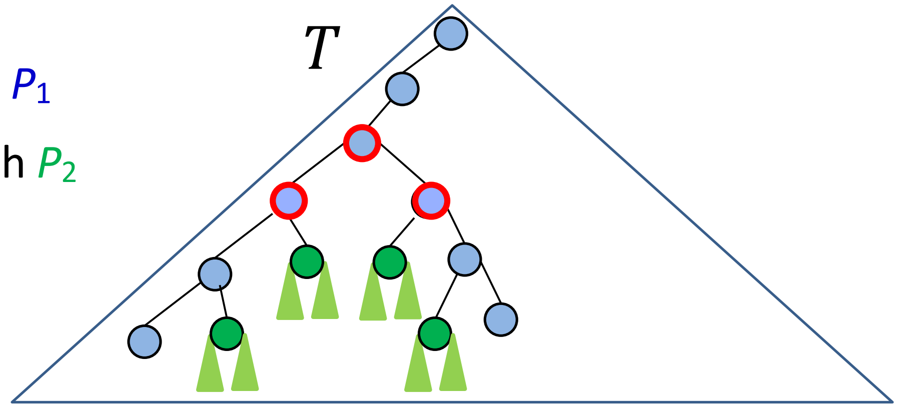
- v is a top inside node if
  - v is not is in P_1 or P_2
  - parent of v is in P_1 or P_2 (but not both)
  - if parent is in P_1, then v is right child
  - if parent is in P_2, then v is left child





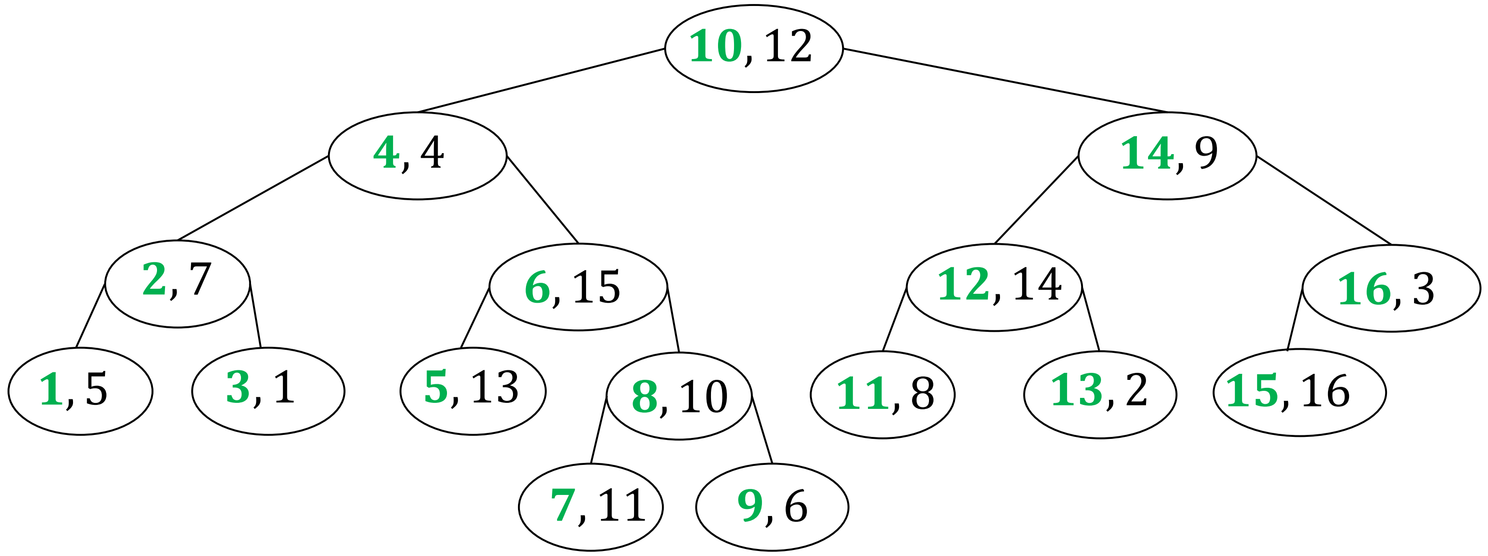
- Thus for each top inside node can report all descendants, no need for search
  - BST range search does not become not faster overall, but top inside nodes are important for $2d$ range search efficiency
  - also important if need to just count the number of points in the search range

Modified BST Range Search Summary



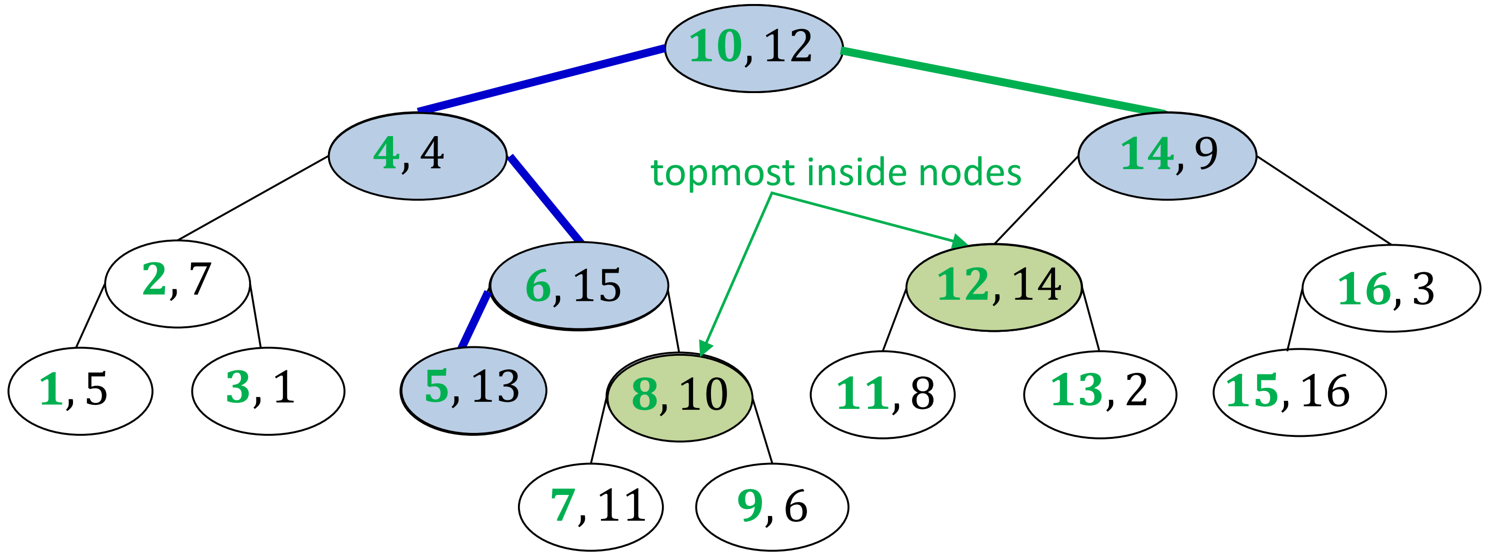
- Search for k_1: this gives left boundary path P_1
- Search for k_2: this gives right boundary path P_2
- Find all topmost inside nodes
  - not in P_1 or P_2
  - left children of nodes in P_2
  - right children of nodes in P_1
- Inside node (which is not a topmost inside) is in a subtree of some topmost 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



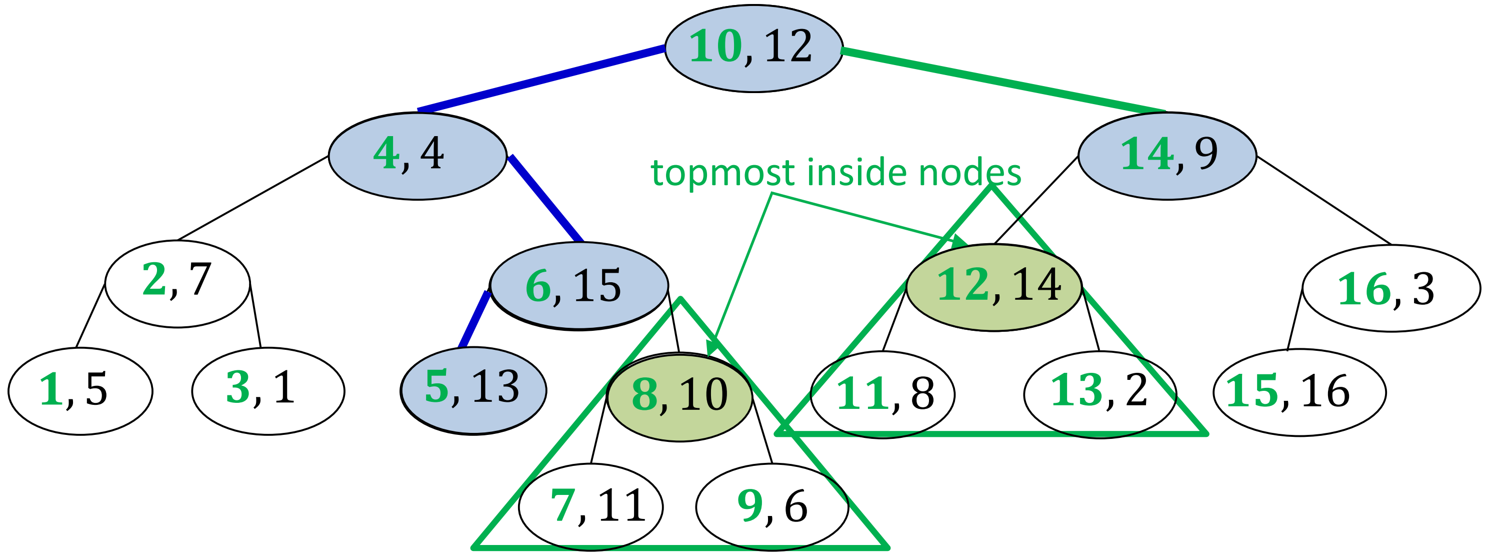
- Have a set of 2D points
  - $S = \{(1,5), (2,7), (3,1), (4,4), (5,13), (6,15)(7,11), (8,10), (9,6), (10,12), (11,8), (12,14), (13,2), (14,9), (15,16), (16,3)\}$
- Example of 2D range search
- *BST-RangeSearch*$(T, 5, 14, 5, 9)$
  - find all points with $5 \leq x \leq 14$ and $5 \leq y \leq 9$
- Construct 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



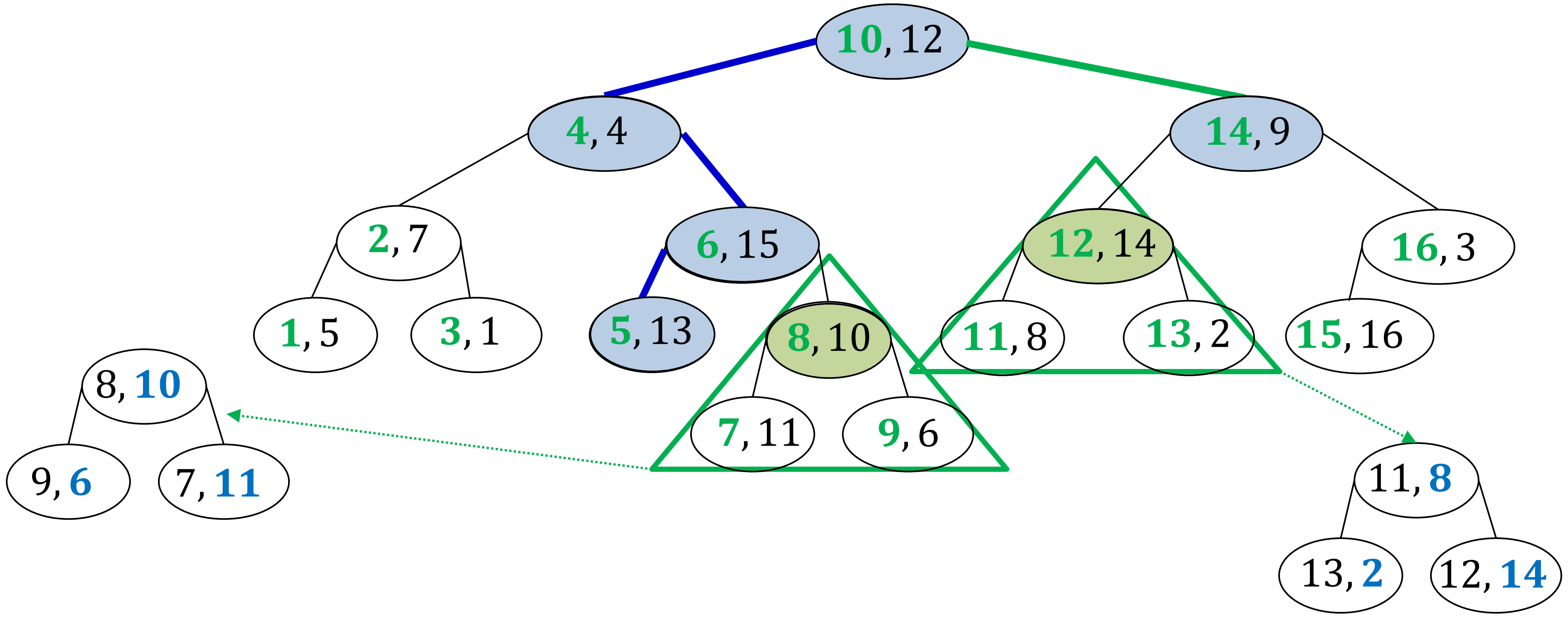
- Consider $2D$ range search *BST-RangeSearch*$(T, 5, 14, 5, 9)$
- First perform *BST-RangeSearch*$(T, 5, 14)$
  - let A be the set of nodes *BST-RangeSearch*$(T, 5, 14)$ returns
    - $A = \{(10{,}12), (6{,}15), (5{,}13), (14{,}9), (8{,}10), (7{,}11), (9{,}6), (12{,}14), (11{,}8), (13{,}2)\}$
  - let 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



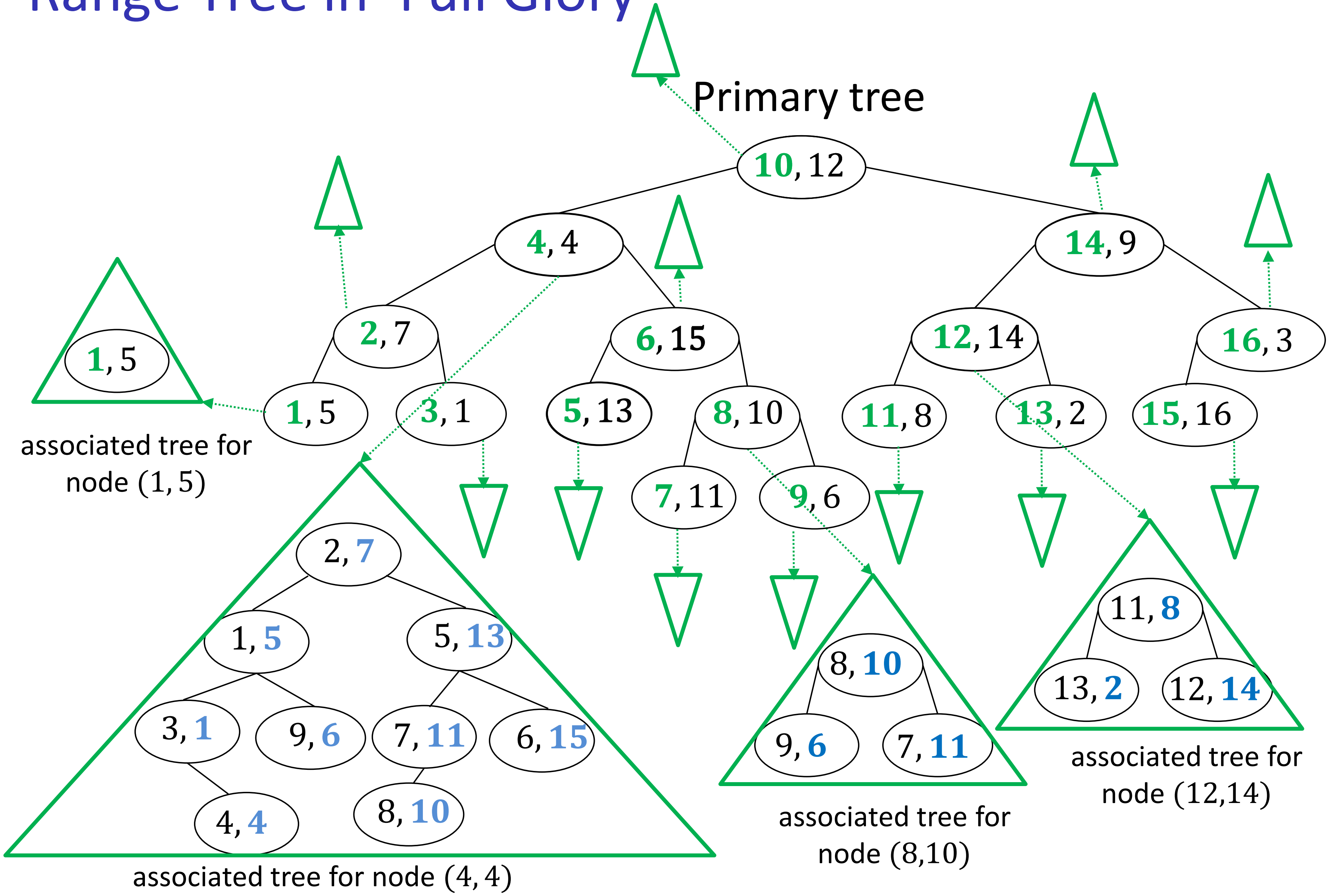
- Consider $2D$ range search *BST-RangeSearch*$(T, 5, 14, 5, 9)$
- First perform only **partial** *BST-RangeSearch*$(T, 5, 14)$
  - find boundary and topmost inside nodes, takes $O(\log n)$ time
- Next
  - for boundary nodes, check if **both** x and y coordinates are in the range, takes $O(\log n)$ time 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

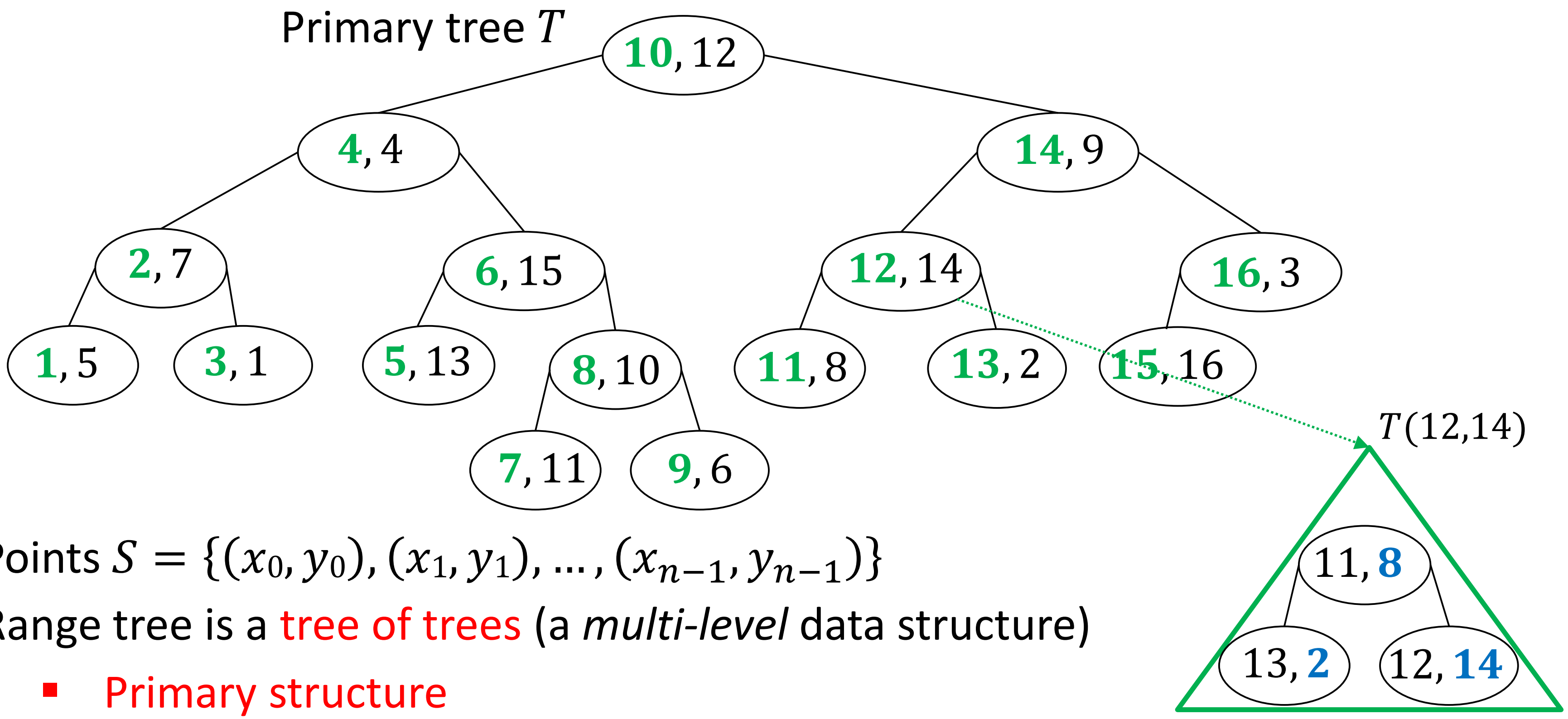


- Need to search subtrees by y-coordinate, but they are x-coordinate based
- Brute-force solution
  - create an associate balanced BST tree for each node v
  - stores the same items as the main (primary) subtree rooted at node v
  - but key is y-coordinate

Range Tree in 'Full Glory'

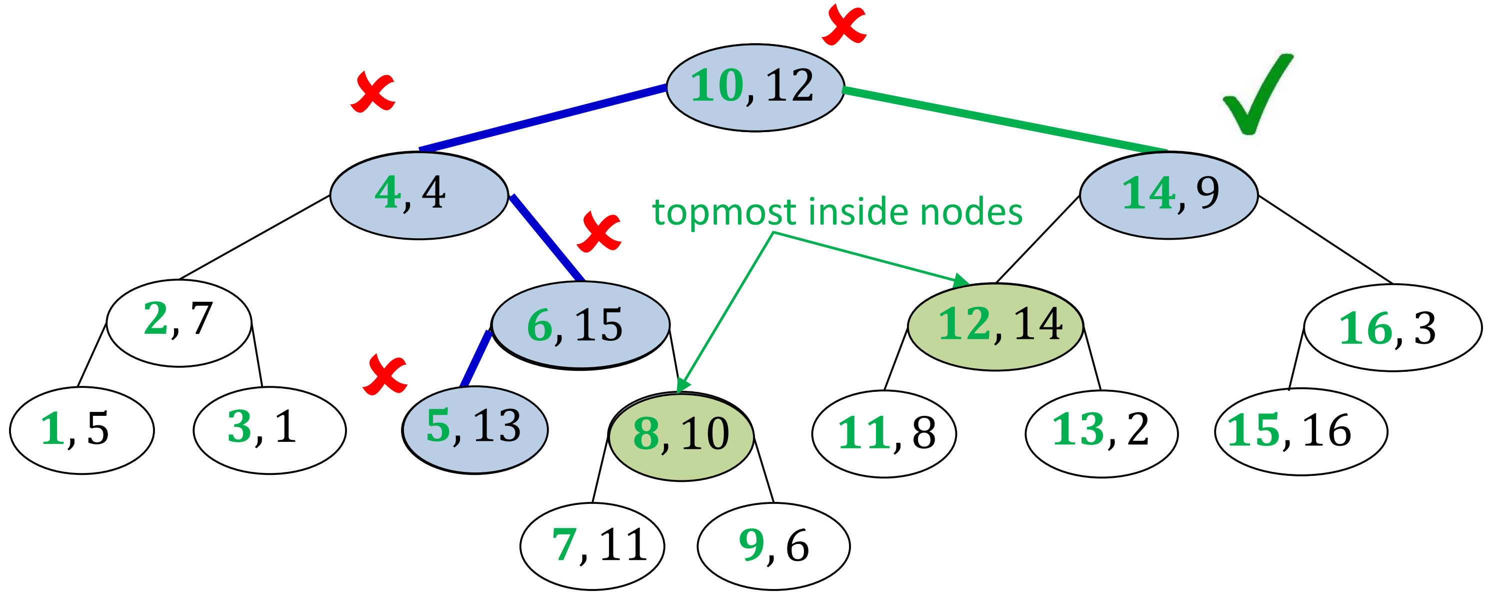


2-dimensional Range Trees Full Definition



- 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 $T(v)$

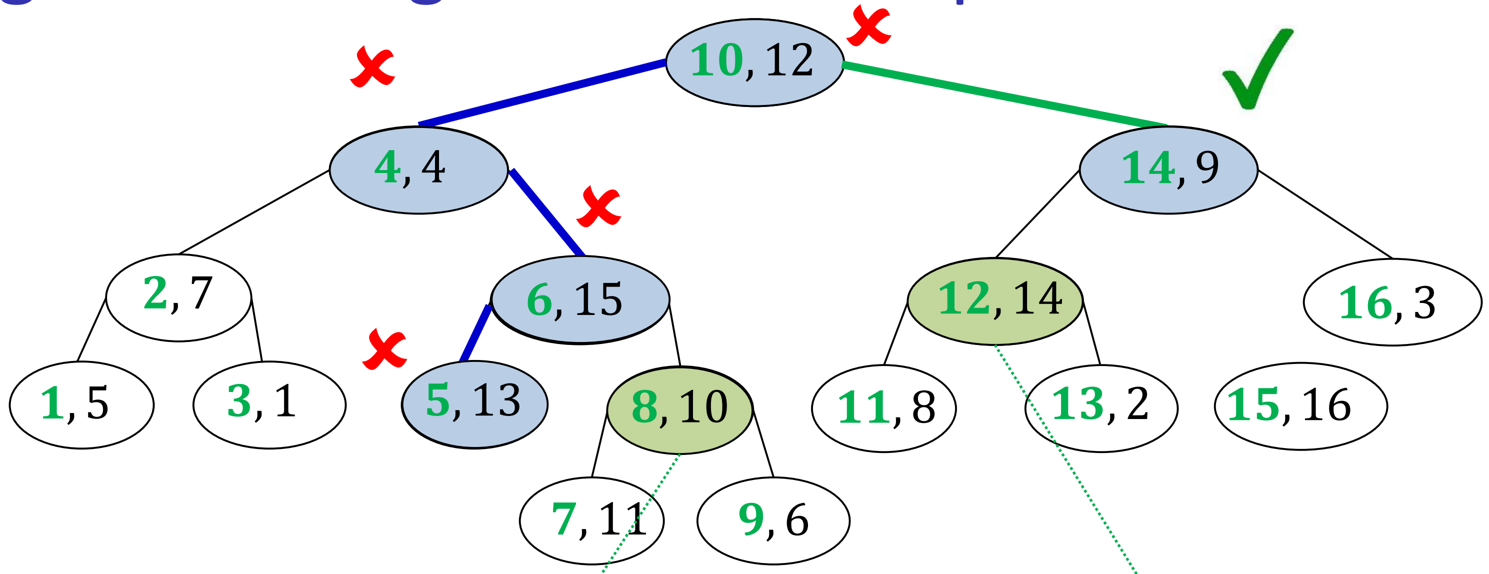
Range search in 2D Range Tree Overview



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

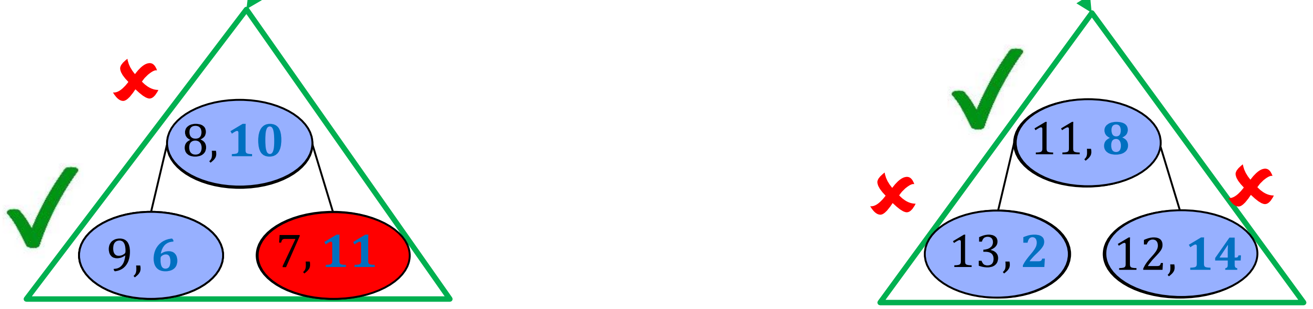
1. Perform *modified BST-RangeSearch*$(T, 5, 14)$
   - find boundary and topmost inside nodes, but do not go through the inside subtrees
   - modified version takes $O(\log n)$ time
     - does not visit all the nodes in valid range for *BST-RangeSearch*$(T, 5, 14)$
2. Check if boundary nodes have valid 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

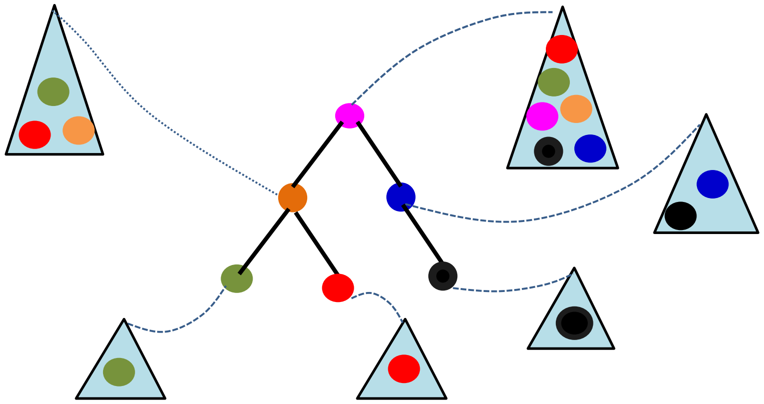


- $RangeTree::RangeSearch(T, 5, 14, 5, 9)$
- For every topmost inside node v, search in associated tree $BST\text{-}RangeSearch(T(v), 5, 9)$

$BST\text{-}rangeSearch(T(8{,}10), 5{,}9)$ $BST\text{-}RangeSearch(T(12{,}14), 5{,}9)$



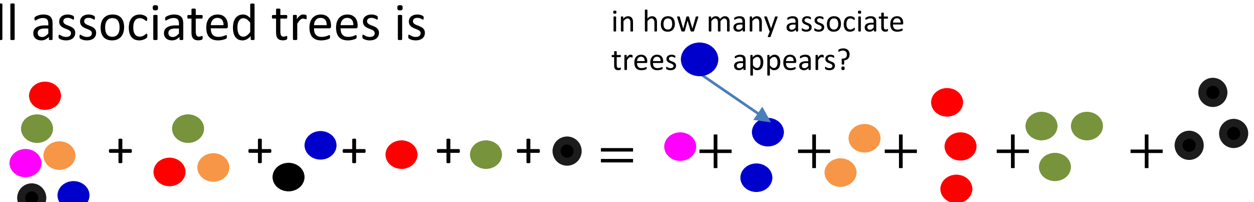
Range Tree Space Analysis



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

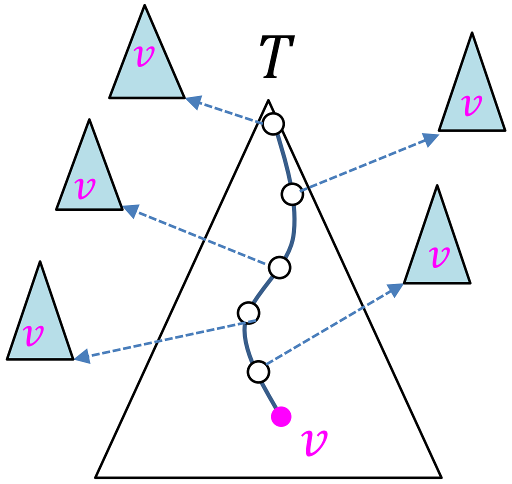
- Space for all associated trees is

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



$$= \sum_{v\in T} \underbrace{\text{\#of ancestors of } v}_{\le c\log n}$$

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



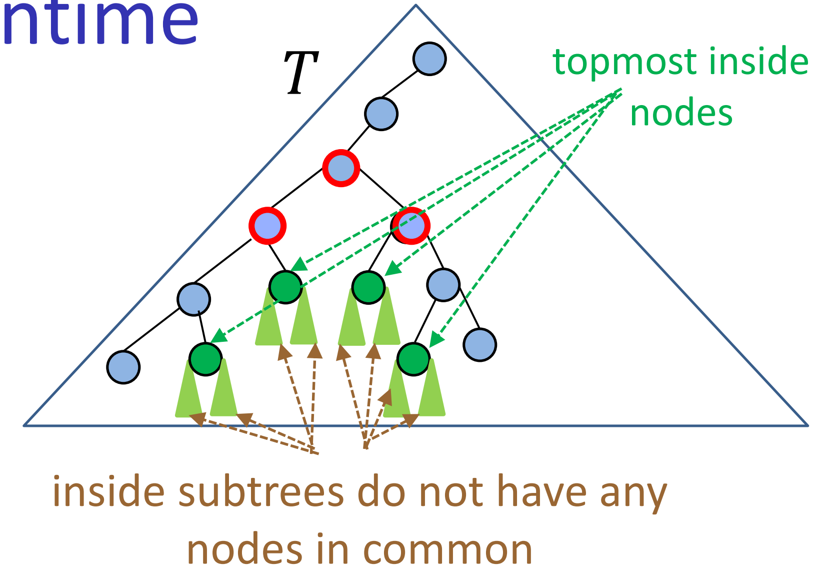
\#of ancestors of v

- 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
    - rotation at v changes $S(v)$ and hence requires re-build of $T(v)$
  - instead of rotations, can allow certain imbalance, rebuild entire subtree if violated
    - no details

Range Trees: Range Search Runtime



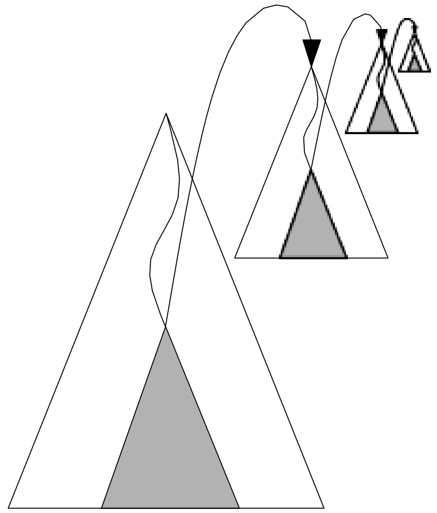
- Find boundary nodes in the primary tree and check if keys are in the range
    - $O(\log n)$
- Find topmost inside nodes in primary tree
    - $O(\log n)$
- For each topmost inside node v, perform range search for y-range in associate tree
    - $O(\log n)$ topmost inside nodes
    - let s_v be #items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

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

- 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 for higher dimensions
  - convention: points on split lines belong to the right/top side
- kd-trees
  - linear space
  - range search is $O(s + \sqrt{n})$
  - inserts/deletes destroy balance and range search time
    - fix with occasional rebuilt
  - convention: points on split lines belong to the right/top side
- 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