KD-Trees

CMSC 420

Goodbye `Comparable`s!

- Every data structure that we have dealt with so far has operated on elements with a 1-1 mapping towards $\mathbb{N}$, the set of naturals.
- Multi-dimensional points don't have that ordering.
- Norms (lengths) are not good for ordering since there exist **infinitely many points with the same norm!**
- Examples: circles in 2D, spheres in 3D, hyperspheres,…

The ‘K’ in KD-Tree

- KD-Trees were invented by Dr. Jon Bentley
- The phrase “KD- Trees” is kind of a misnomer ☹
  - K is really a strictly positive integer, with K = 1 being a classic BST with all of its good and bad characteristics.
  - But the term “KD-Tree” prevails instead of 2D – Tree, 3D – Tree, etc.
- As K grows larger, some operations become more expensive.

Speaking of operations…

- The classic key-value store operations are **still there**
  - Insert, delete, search, …
- But with spatial data structures, we **have more things that we can do**!
  - Nearest Neighbor Queries and m - Nearest Neighbor queries: Which points are our closest neighbors in the database given a distance metric?
    - Euclidean
    - Manhattan
    - Hamming
    - …
  - Range Queries: is a point within a given hypersphere?
  - Ray shooting (does a line segment that originates from a given point in space pass some other point)

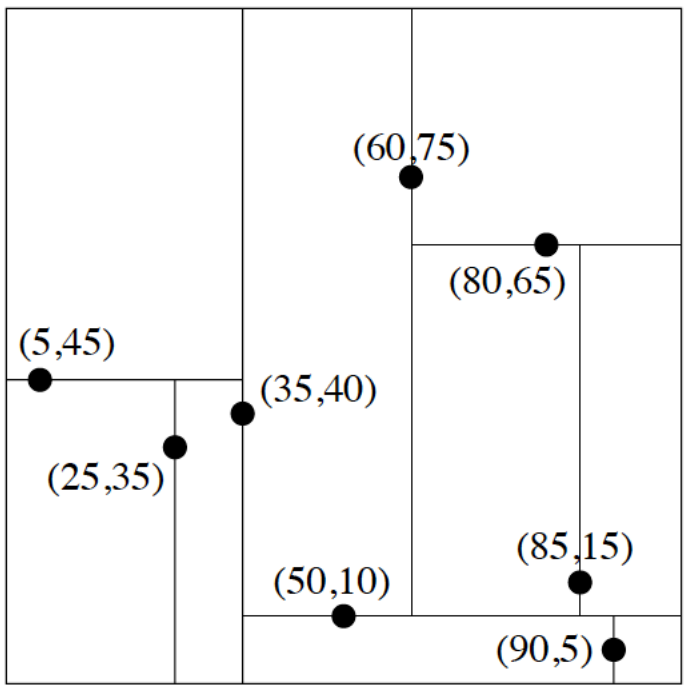
KD-Trees: intuition

- No matter what K is, the KD-Tree will **always** look like a binary tree.
  - That is, a tree with fanout **exactly** two.
- Levels of the tree will be associated with a different dimension!
  - Root level with x coordinate.
  - Children of root with y coordinate.
  - Grandchildren of root with z coordinate
  - ….
- Levels “wrap around” dimensions: after K levels, we fall back to x, then to y and so on.

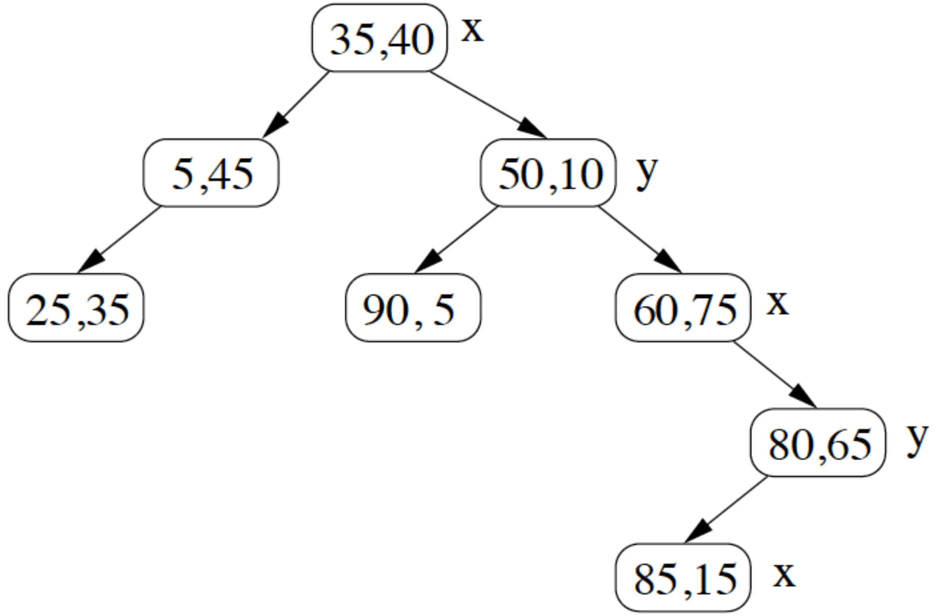
KD-Tree example

(For readability, all slide examples will assume k = 2)

2D space



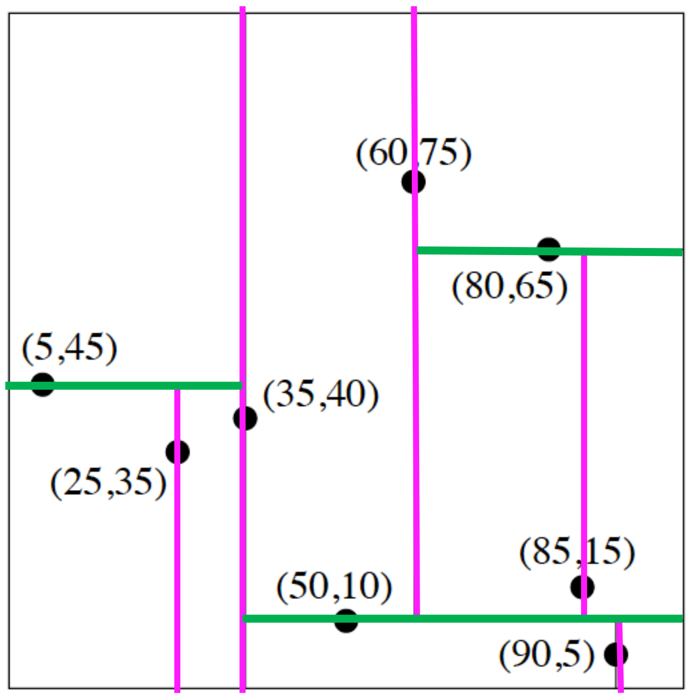
Corresponding Tree



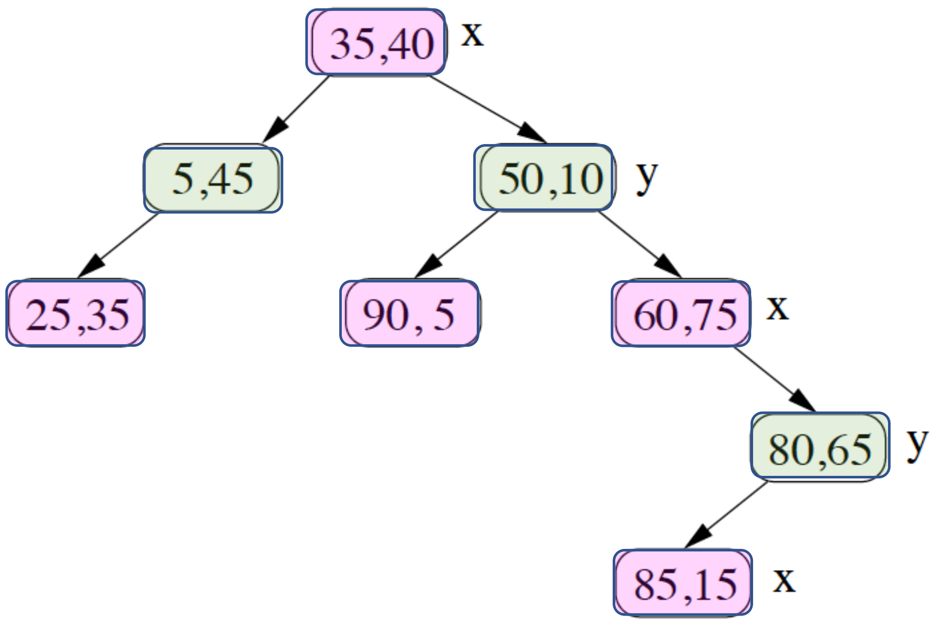
KD-Tree example

(For readability, all slide examples will assume k = 2)

2D space



Corresponding Tree



Insertion

- When we insert, we have to be careful:
  a) To *alternate our dimensions!*
  b) To obey the BST property; points whose current dimension value is bigger than or equal to the visited node's point's current dimension value should be inserted into the right subtree, and vice versa.

Insertion Examples

2D space

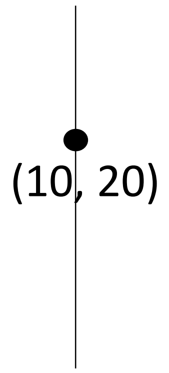
Corresponding Tree

`root`

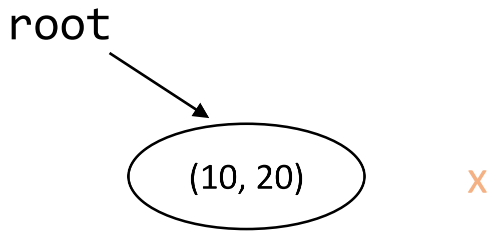
`null`

Insertion Examples

2D space

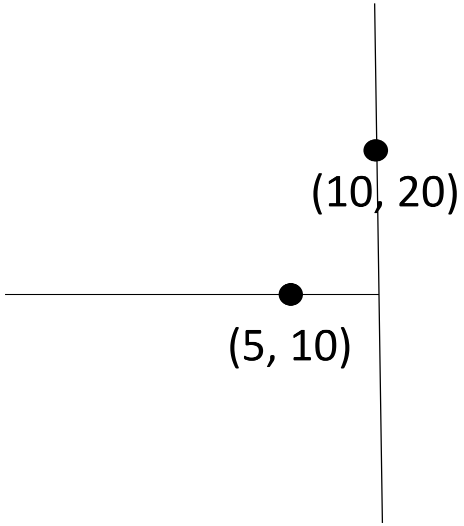


Corresponding Tree

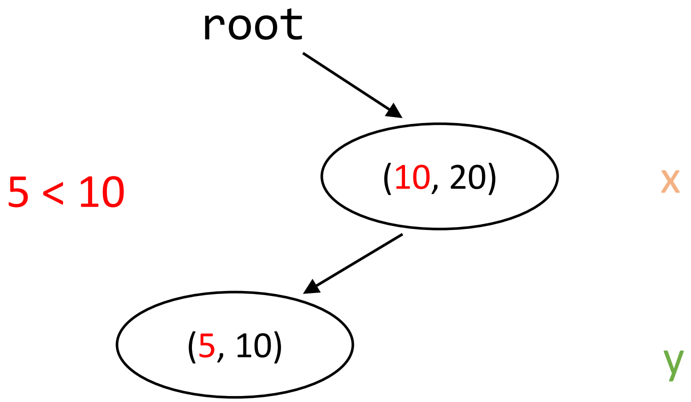


Insertion Examples

2D space

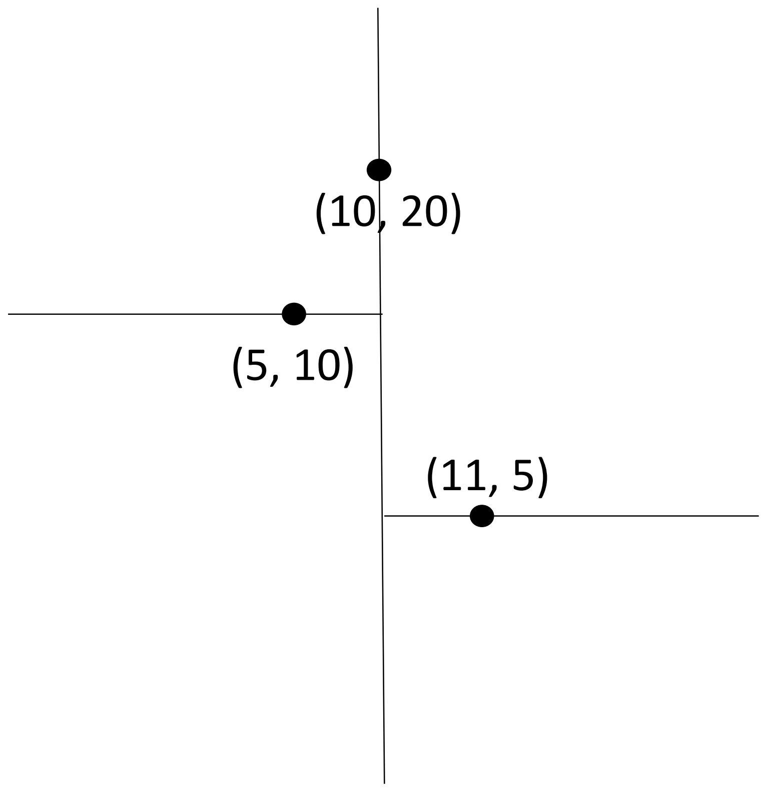


Corresponding Tree

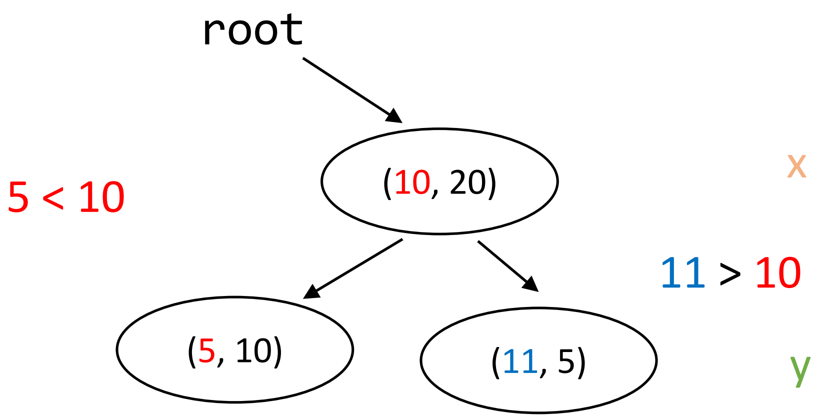


Insertion Examples

2D space

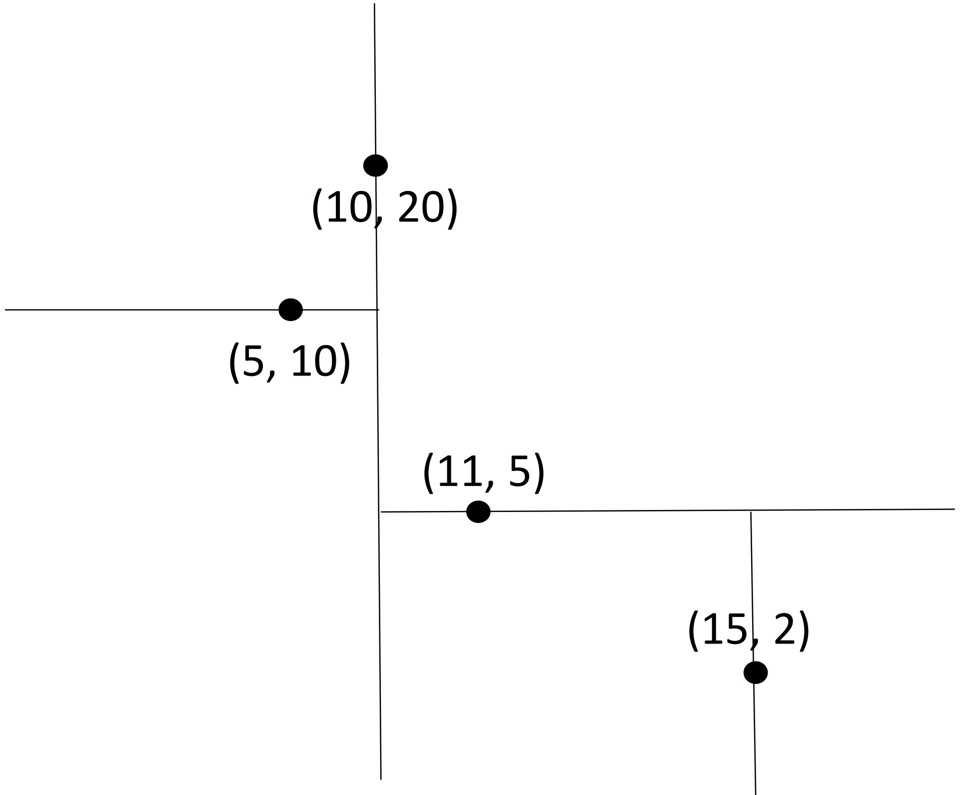


Corresponding Tree

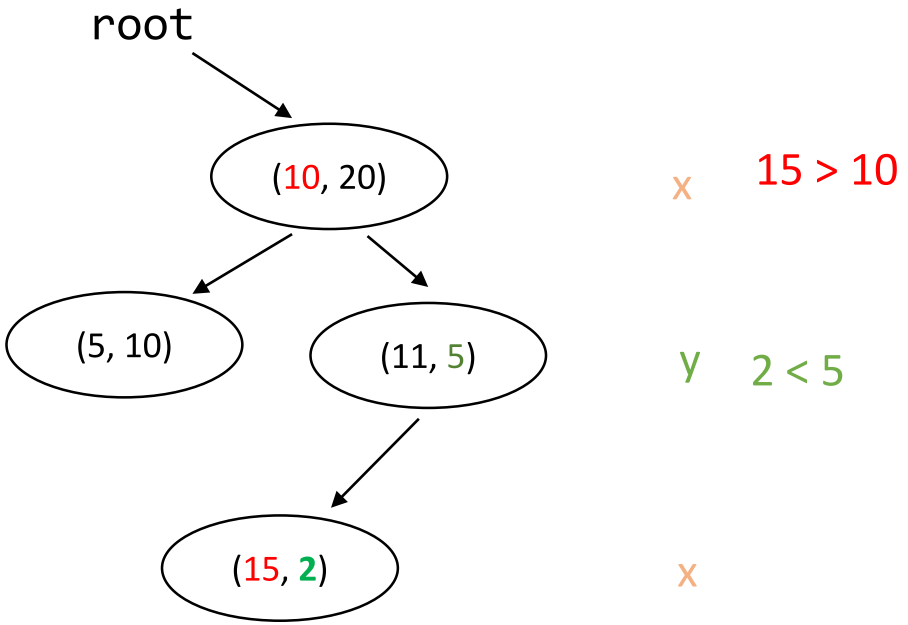


Insertion Examples

2D space

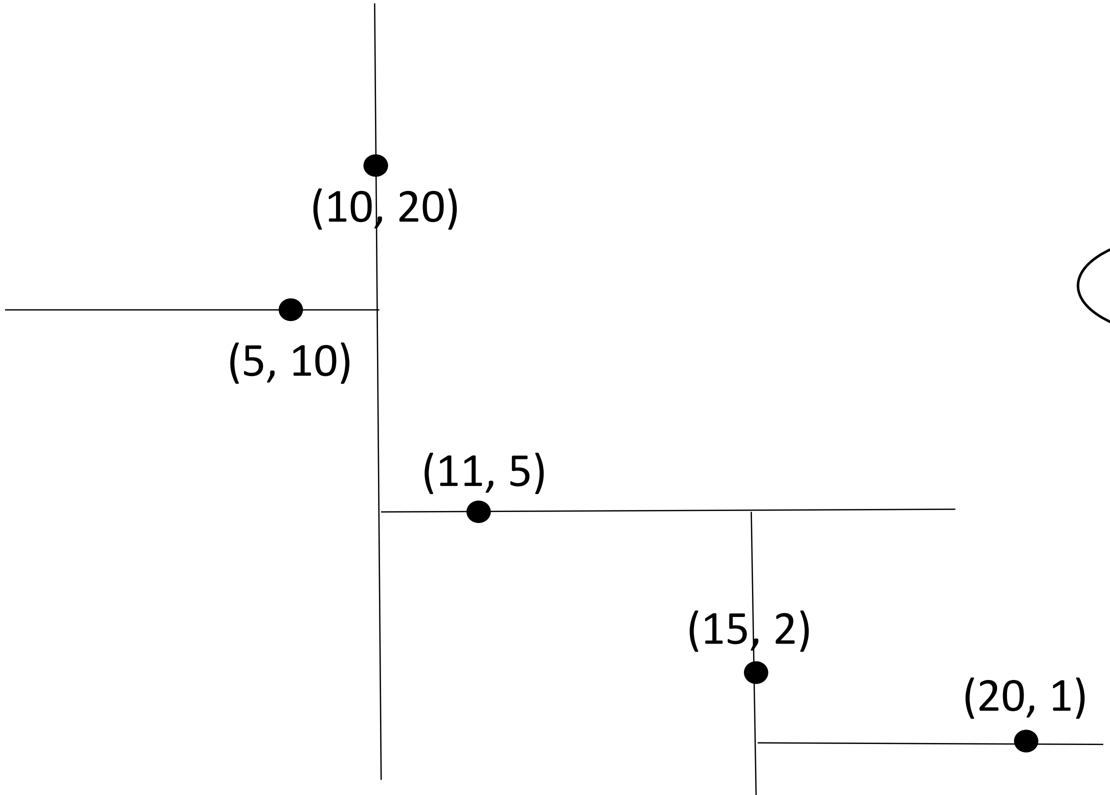


Corresponding Tree

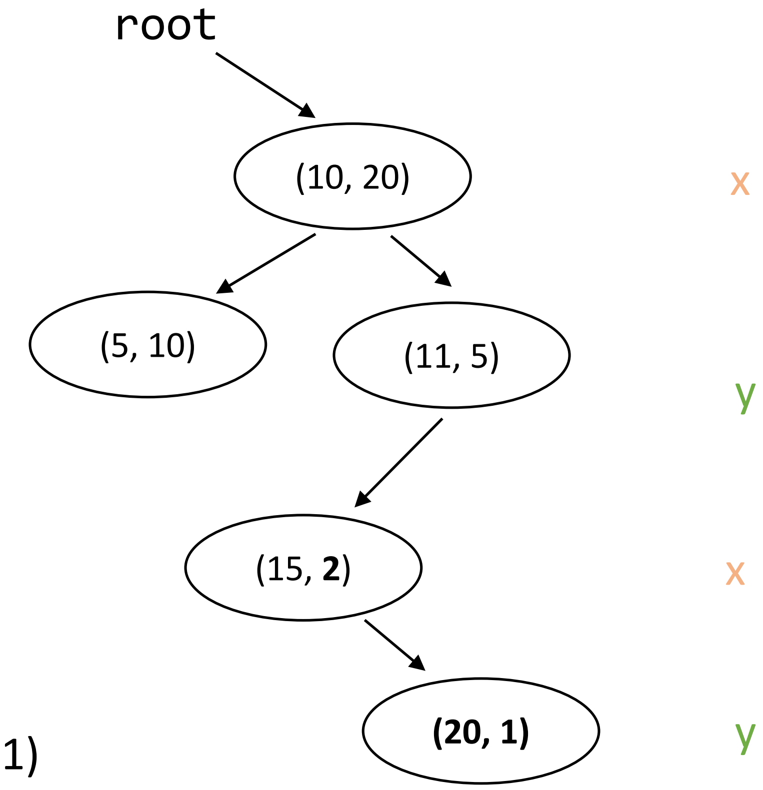


Insertion Examples

2D space

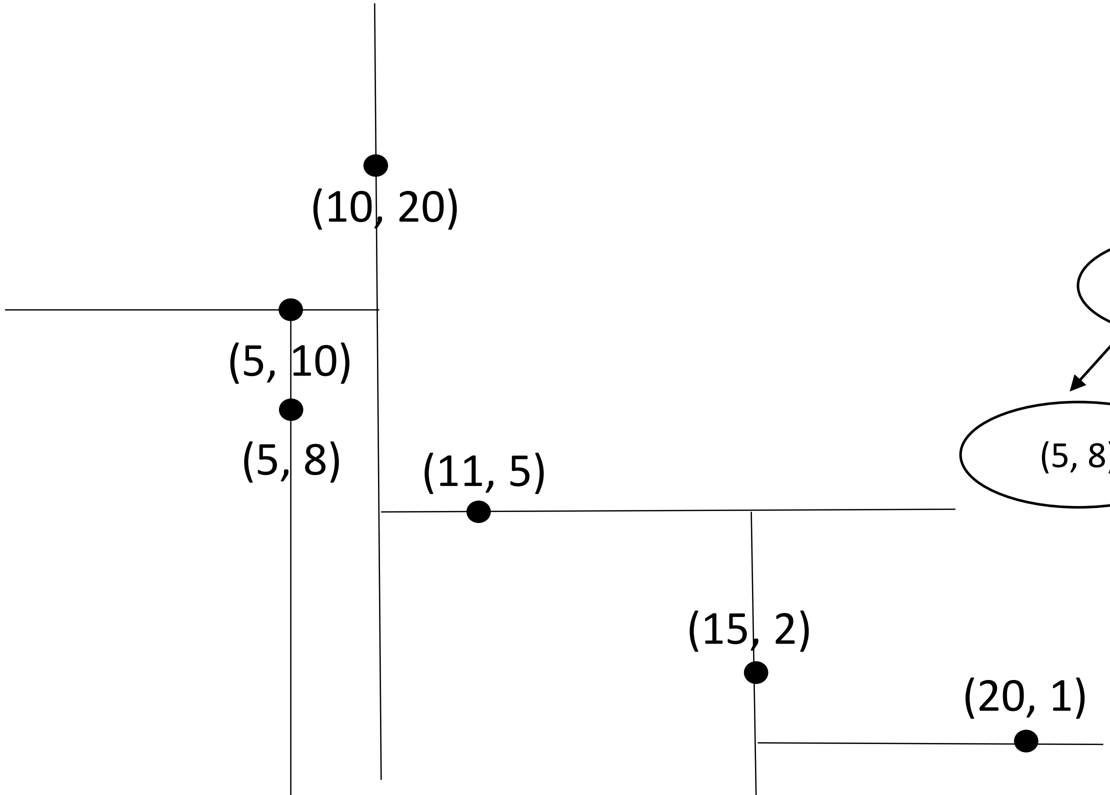


Corresponding Tree

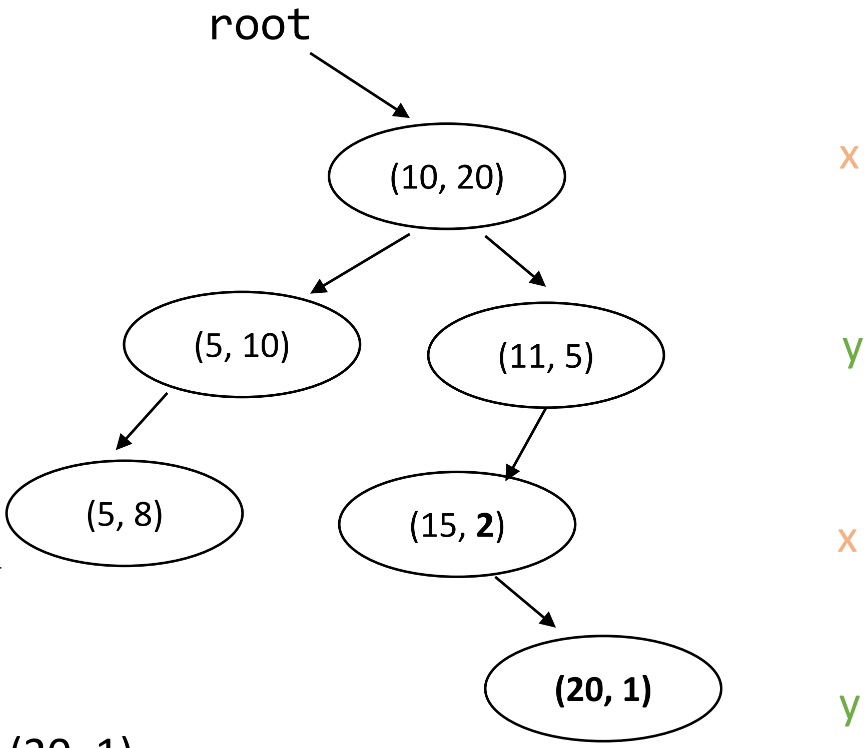


Insertion Examples

2D space



Corresponding Tree



Deletion

- Remember the BST cases:
  1. Left and right child null? Return null (leaf node that gets erased)
  2. Left child non-null and right child null? Replace node with left subtree.
  3. Right child non-null? Exchange node's key with that of the inorder successor node, and recursively delete that key from your right subtree.

Deletion

- Remember the BST cases:
  1. Left and right child null? Return null (leaf node that gets erased)
  2. Left child non-null and right child null? Replace node with left subtree.
  3. Right child non-null? Exchange node's key with that of the inorder successor node, and recursively delete that key from your right subtree.
- **This won't fly in KD-Trees,** for two reasons:
  - In case 2, replacing the node with the left subtree changes the semantics of **every one of the left subtree's nodes' dimension splitting!**
  - In case 3, the notion of an "inorder successor" is now hazy at best (remember, we've moved away from **`Comparable`**s!)

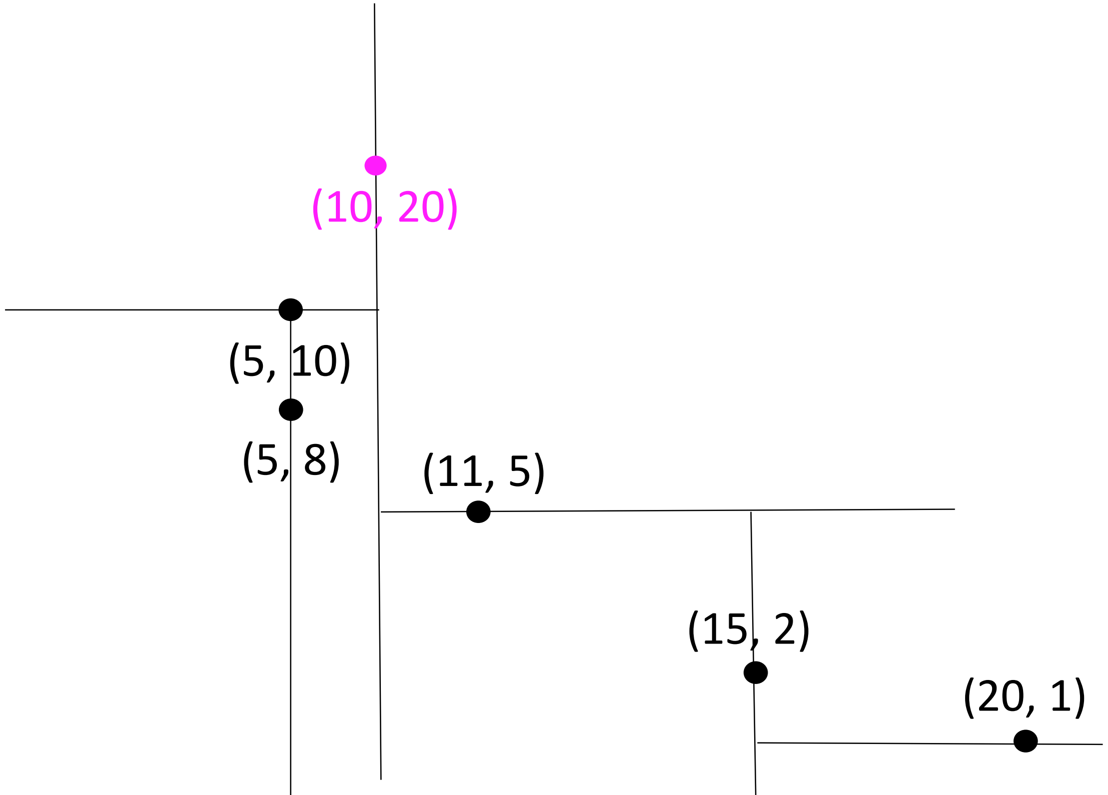
Deletion

- Remember the BST cases:
  1. Left and right child null? Return null (node gets erased)
  2. Left child non-null and right child null? Replace node with left subtree.
  3. Right child non-null? Exchange node's key with that of the inorder successor node, and recursively delete that key from your right subtree.
- **This won't fly in KD-Trees,** for two reasons:
  - In case 2, replacing the node with the left subtree changes the semantics of **every one of the left subtree's nodes' dimension splitting!**
  - In case 3, the notion of an "inorder successor" is now hazy at best (remember, we've moved away from **`Comparable`**s!)
- So what can we do?

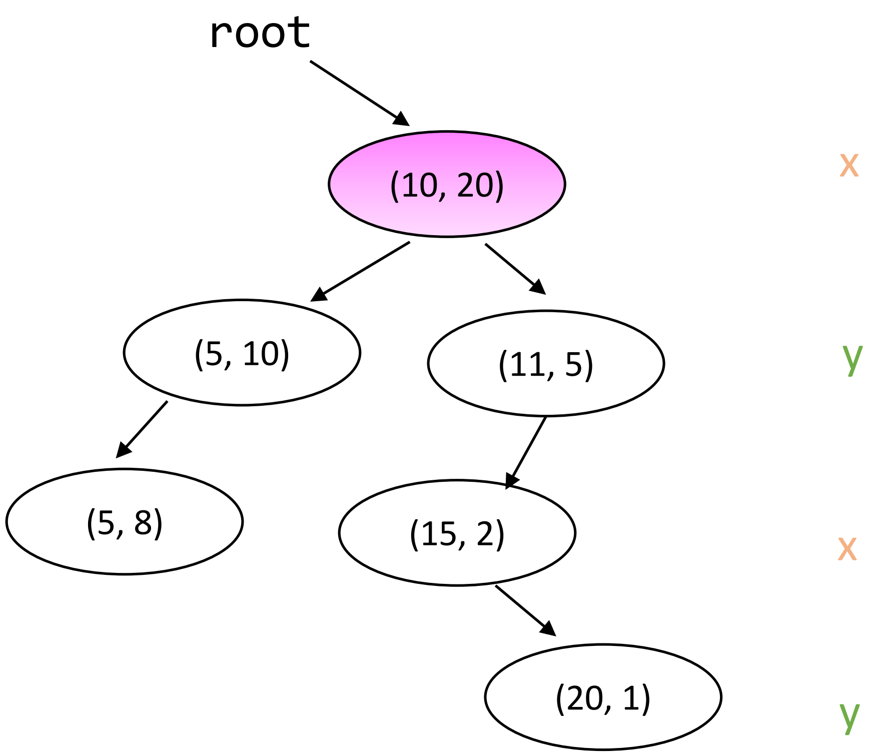
Deletion

- Suppose that we want to delete (10, 20)

2D space



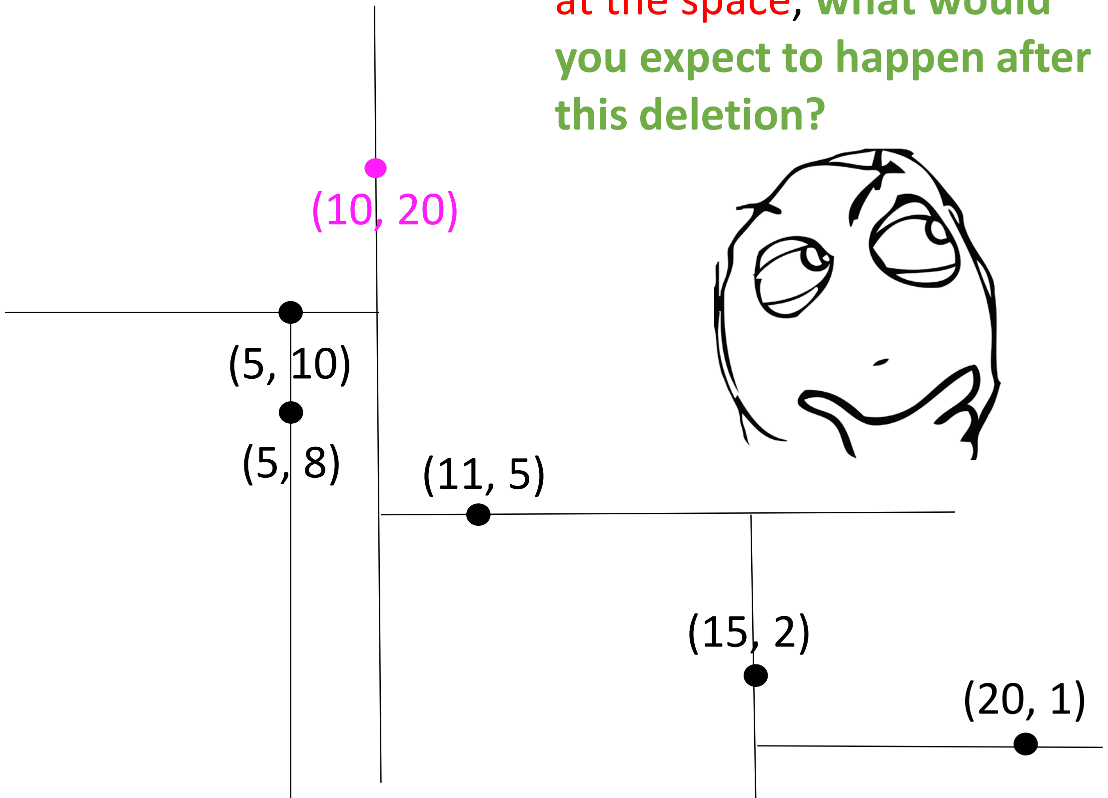
Corresponding KD-Tree



Deletion

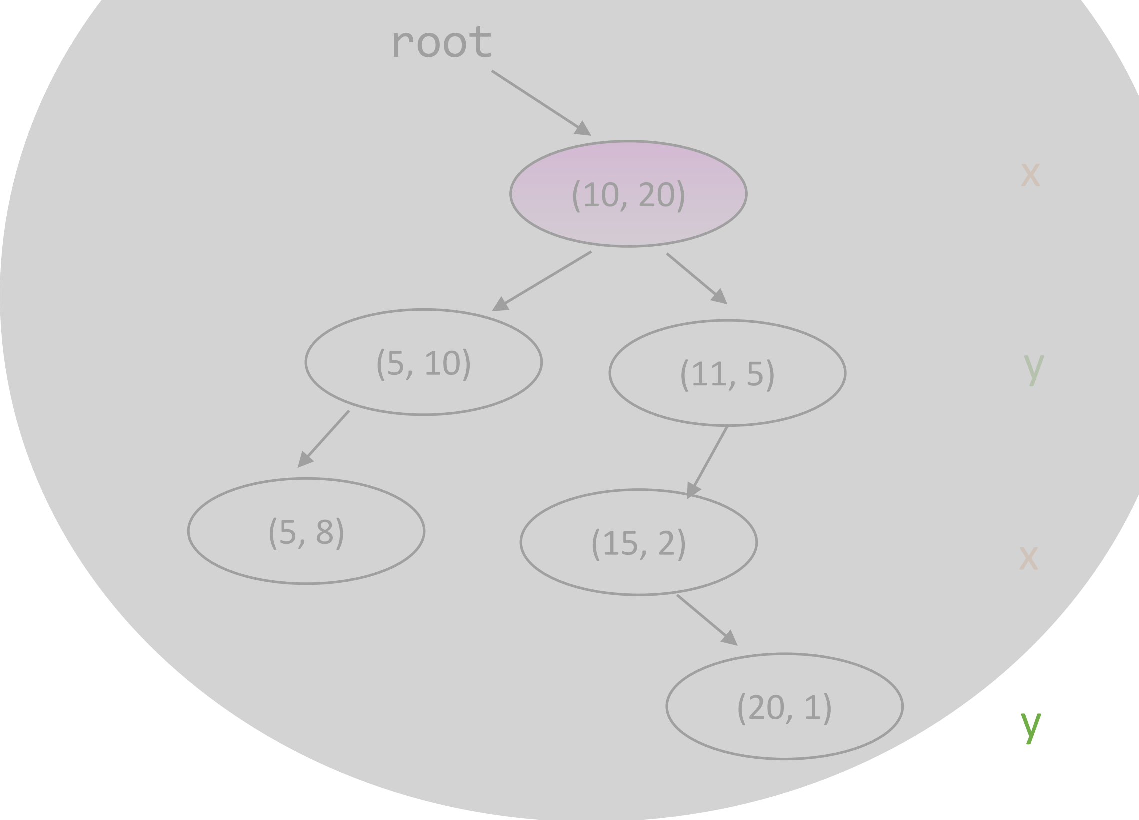
- Suppose that we want to delete (10, 20)

2D space



If we momentarily forget about the tree and only look at the space, **what would you expect to happen after this deletion?**

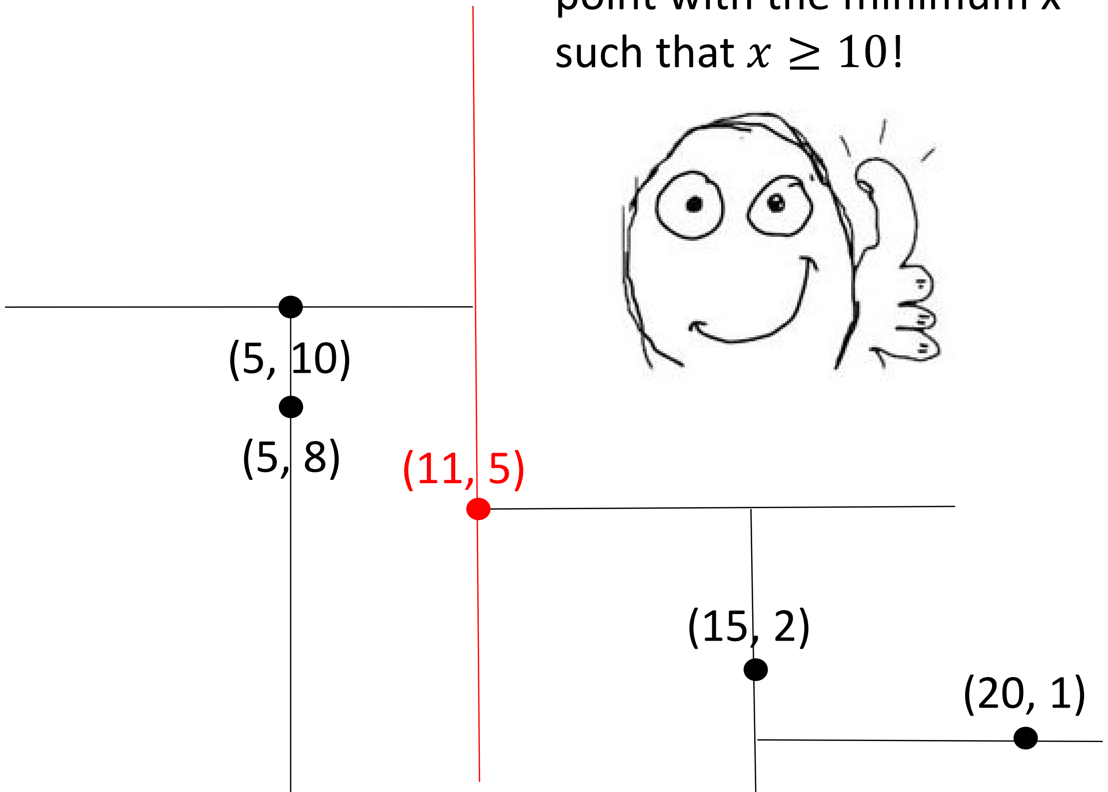
Corresponding KD-Tree



Deletion

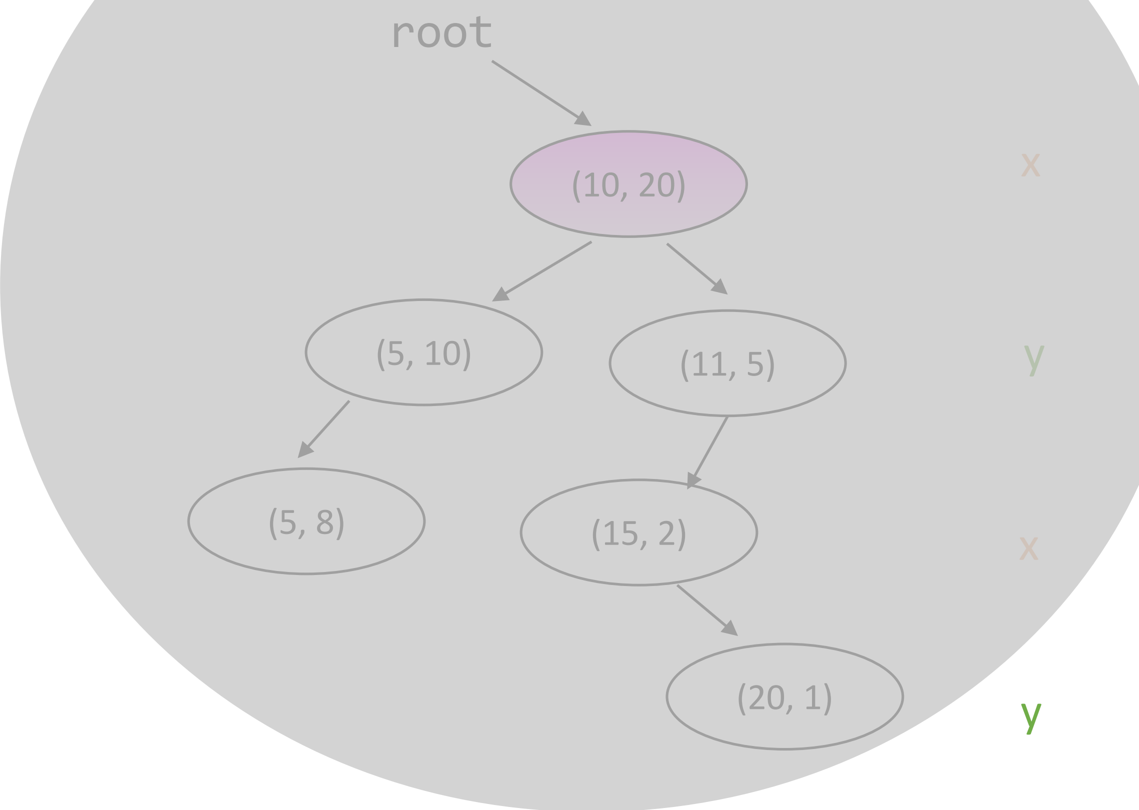
- Suppose that we want to delete (10, 20)

2D space



We would have to move the red vertical line towards the point with the minimum x such that $x \geq 10$!

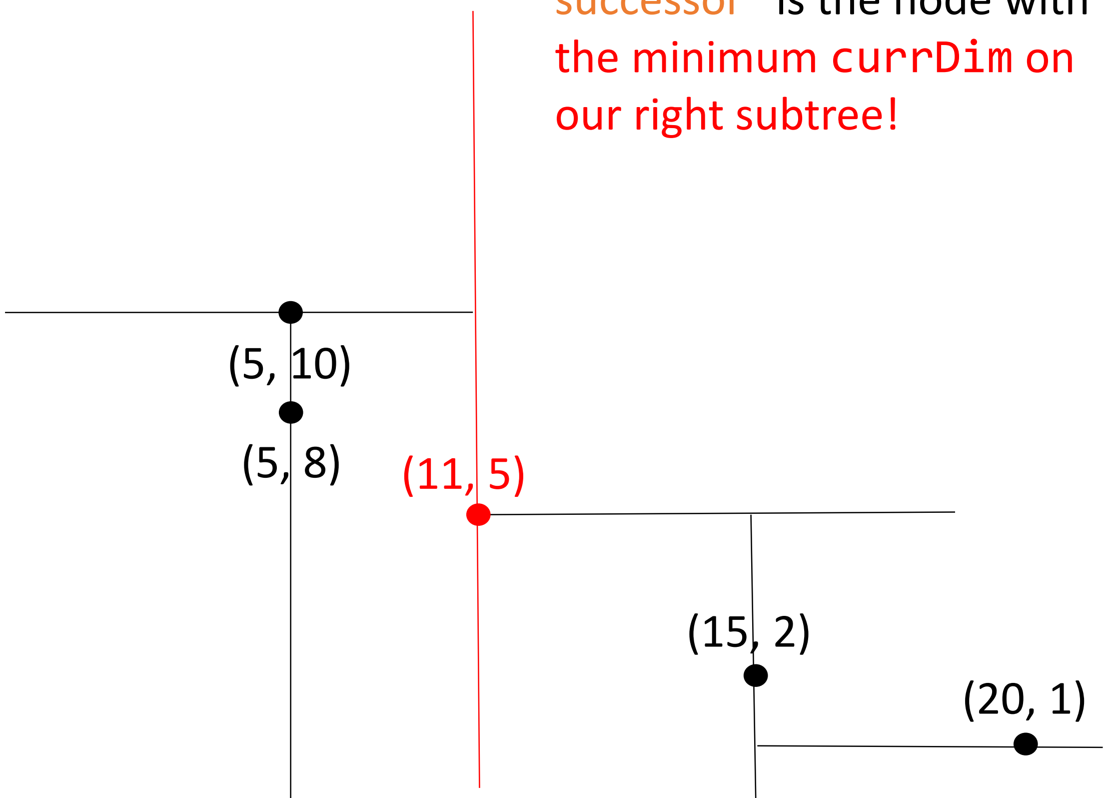
Corresponding KD-Tree



Deletion

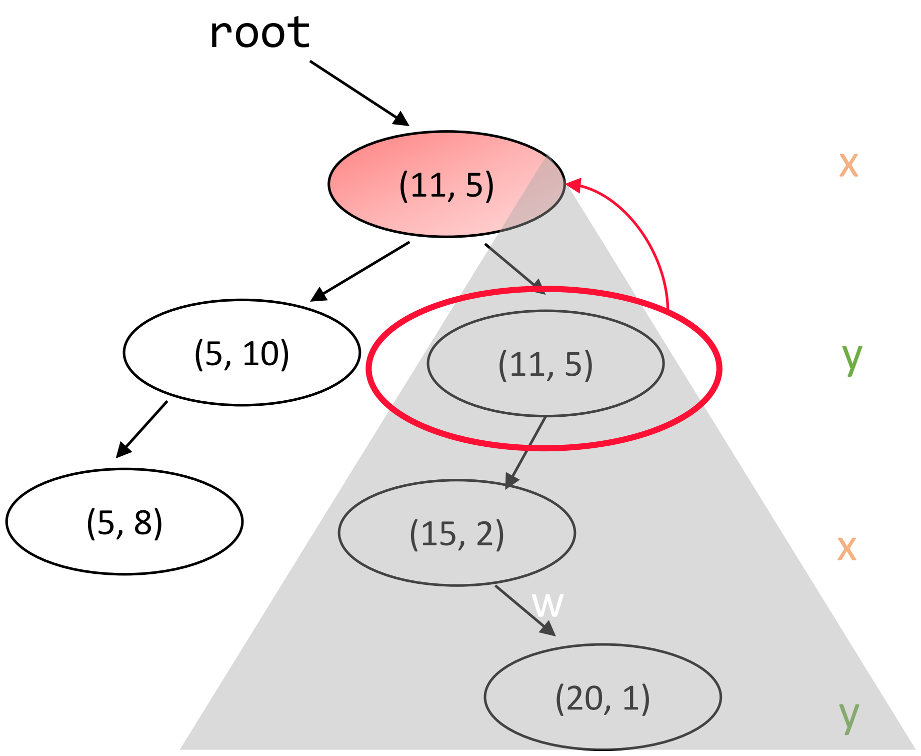
- Suppose that we want to delete (10, 20)

2D space



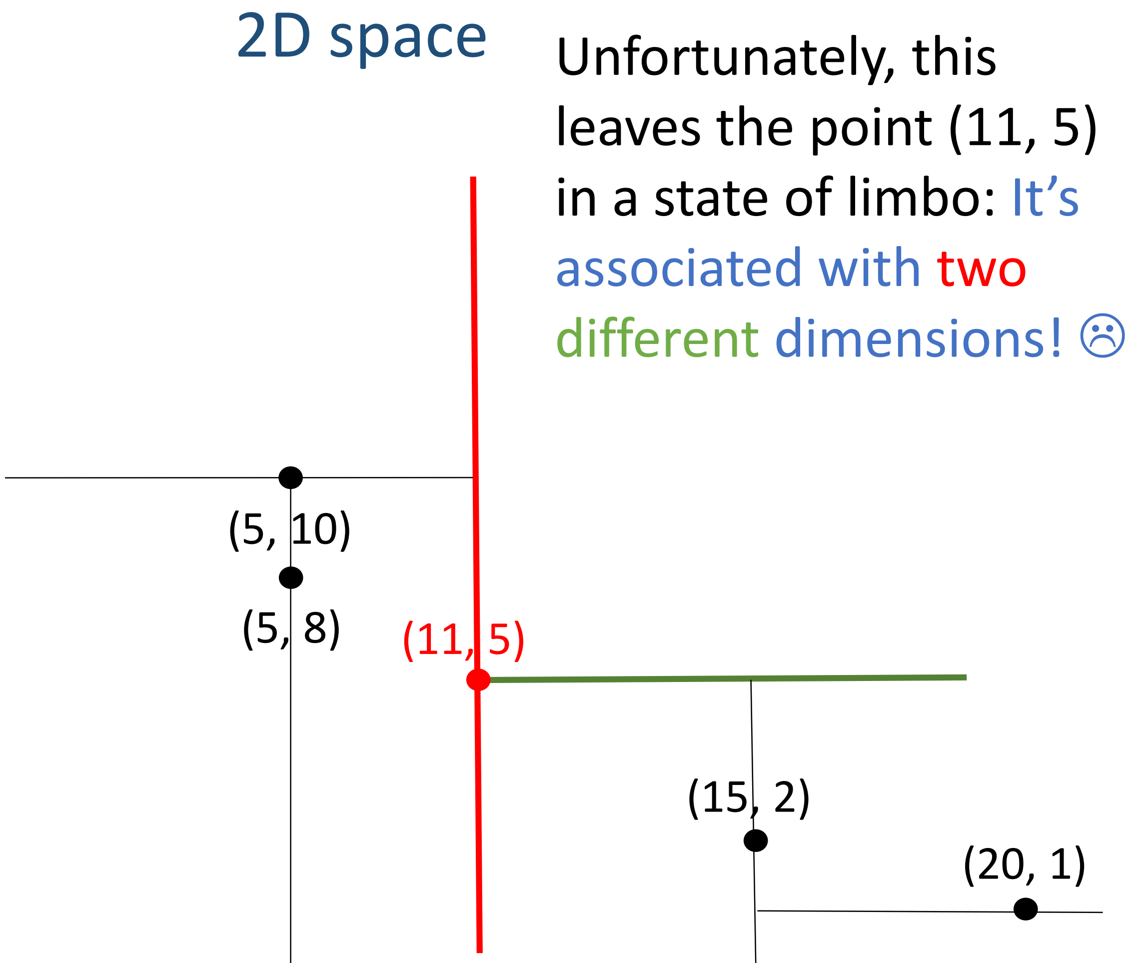
In terms of our tree, this means that the "inorder successor" is the node with the minimum `currDim` on our right subtree!

Corresponding KD-Tree

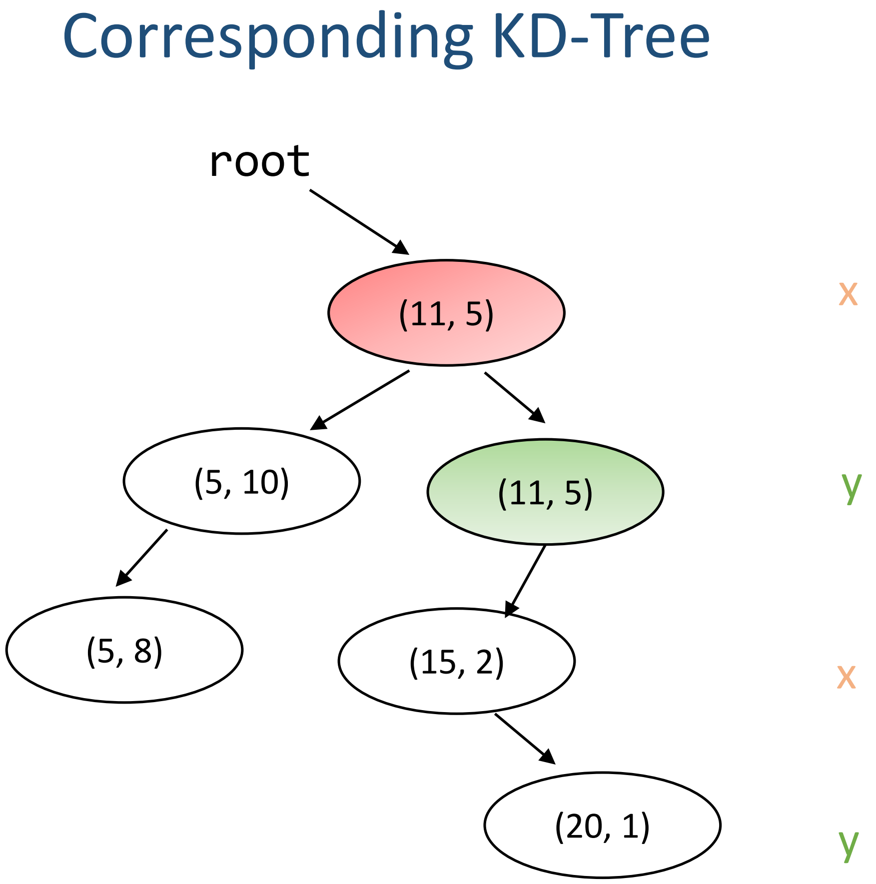


Deletion

- Suppose that we want to delete (10, 20)

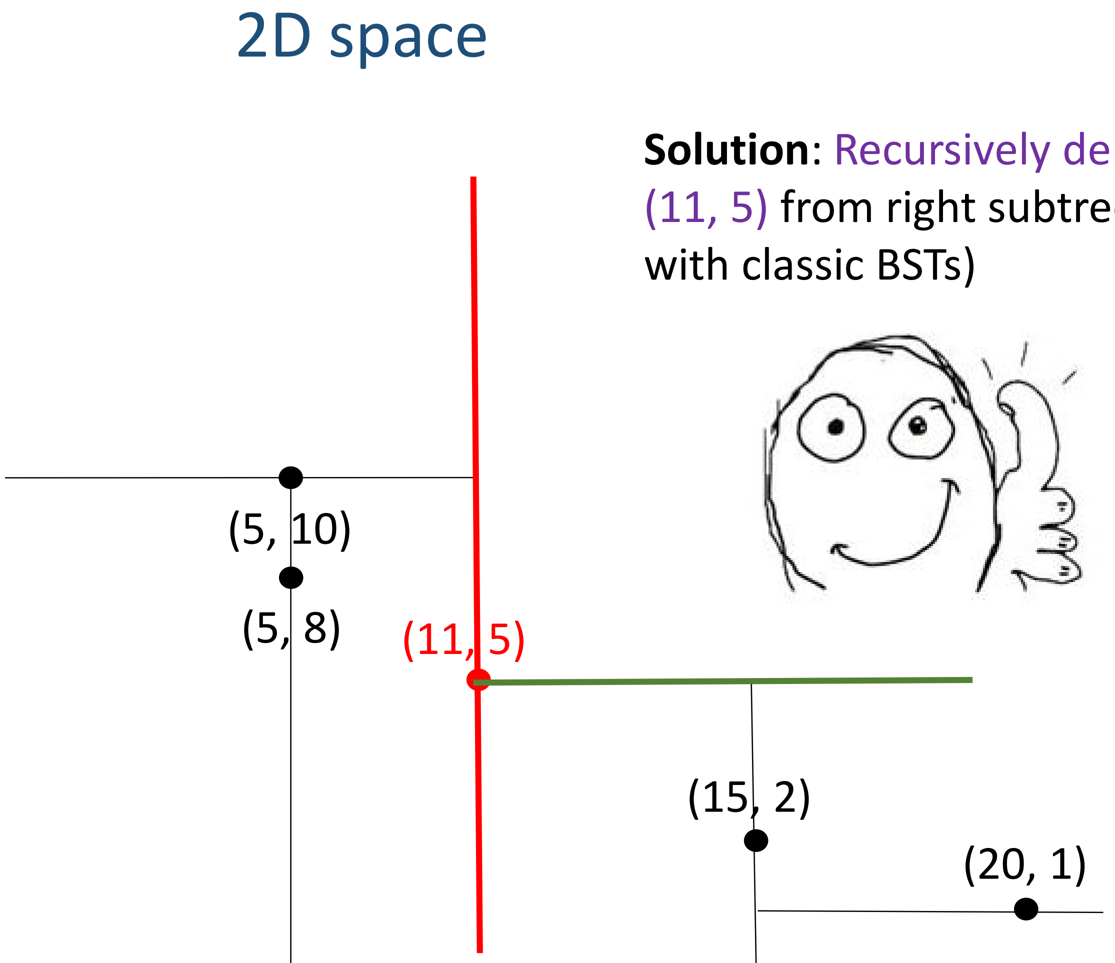


Unfortunately, this leaves the point (11, 5) in a state of limbo: It's associated with two different dimensions! ☹

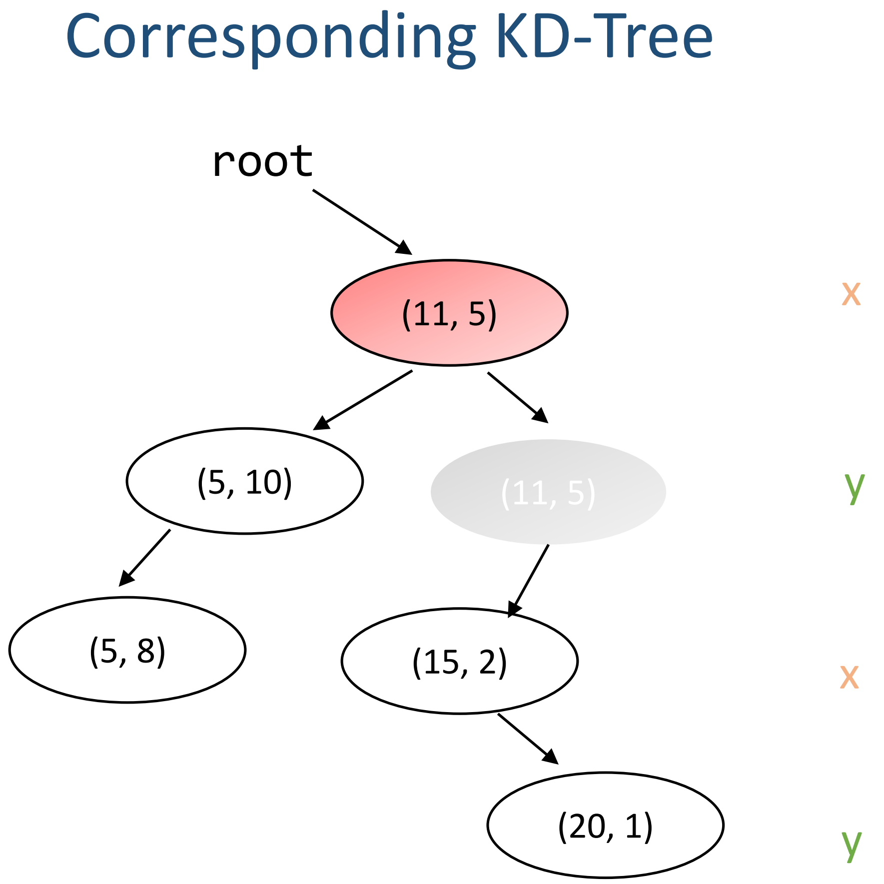


Deletion

- Suppose that we want to delete (10, 20)



Solution: Recursively delete (11, 5) from right subtree (as with classic BSTs)

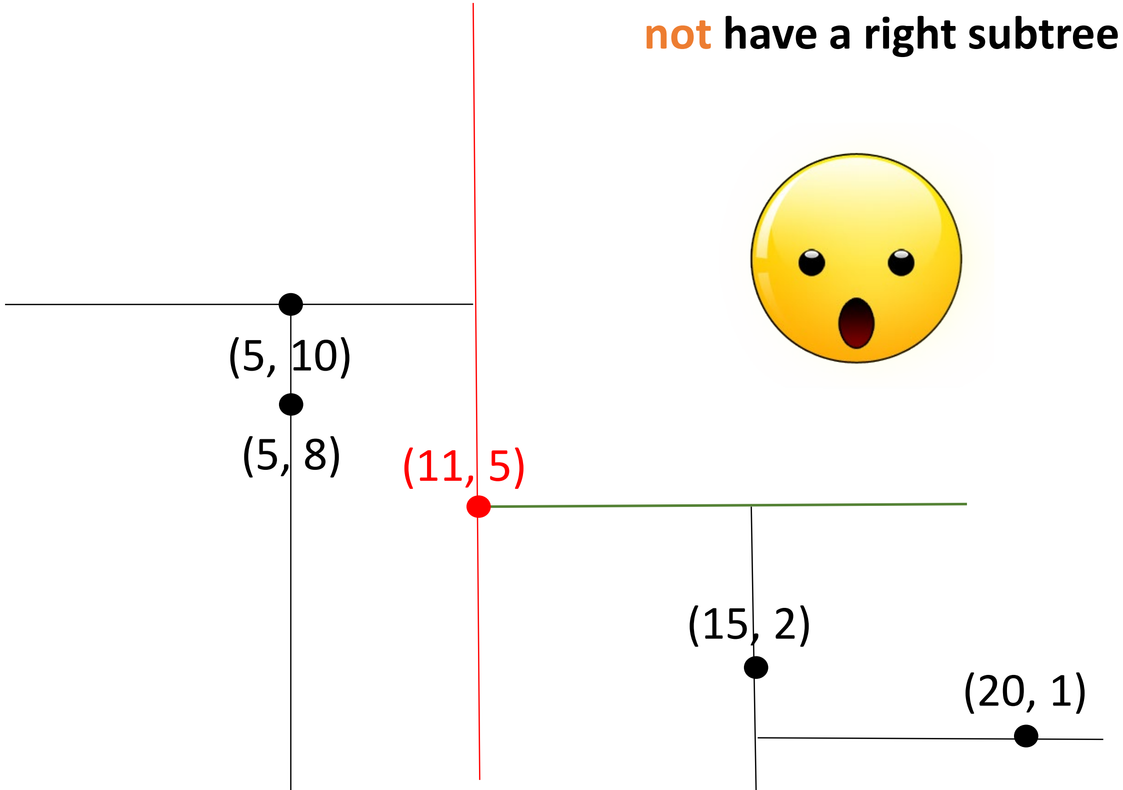


Deletion

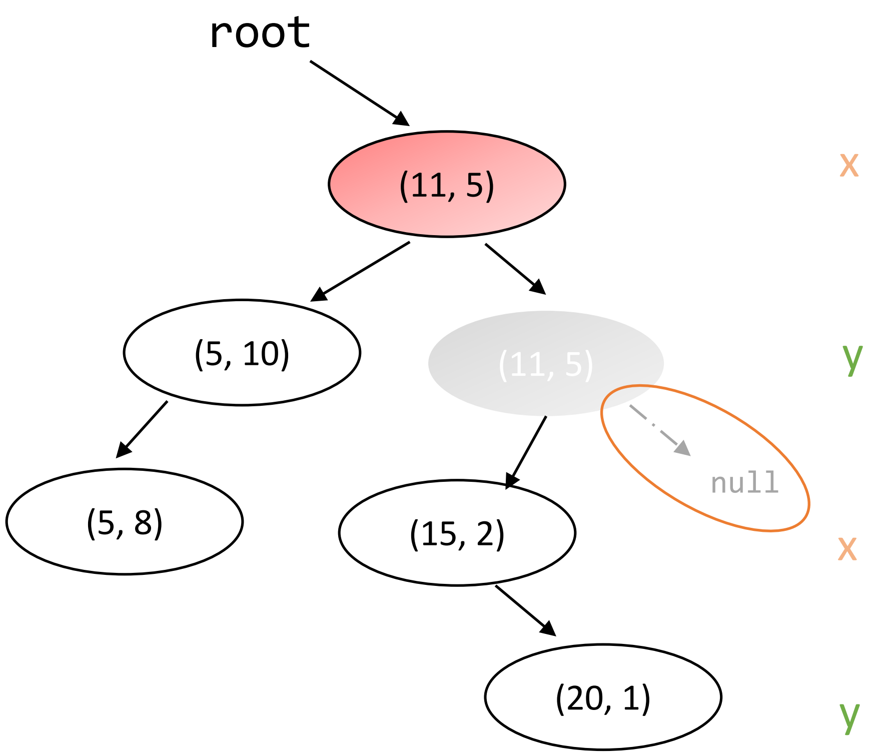
- Suppose that we want to delete (10, 20)

2D space

BUT HOLD ON! (11, 5) does not have a right subtree!



Corresponding KD-Tree

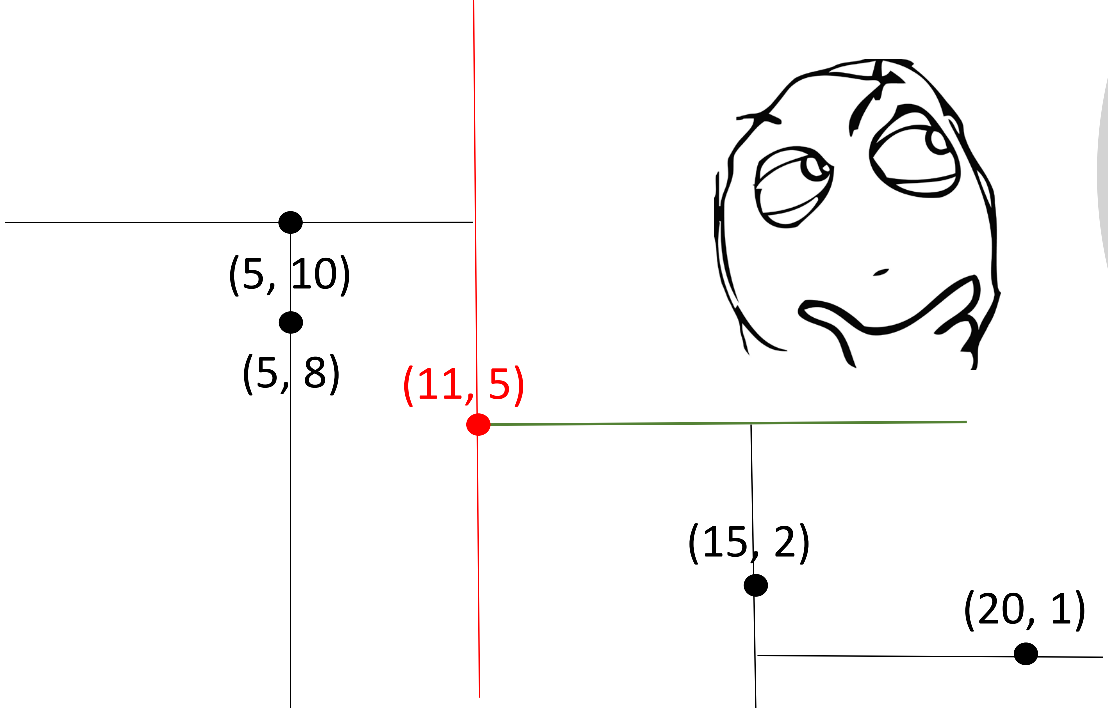


Deletion

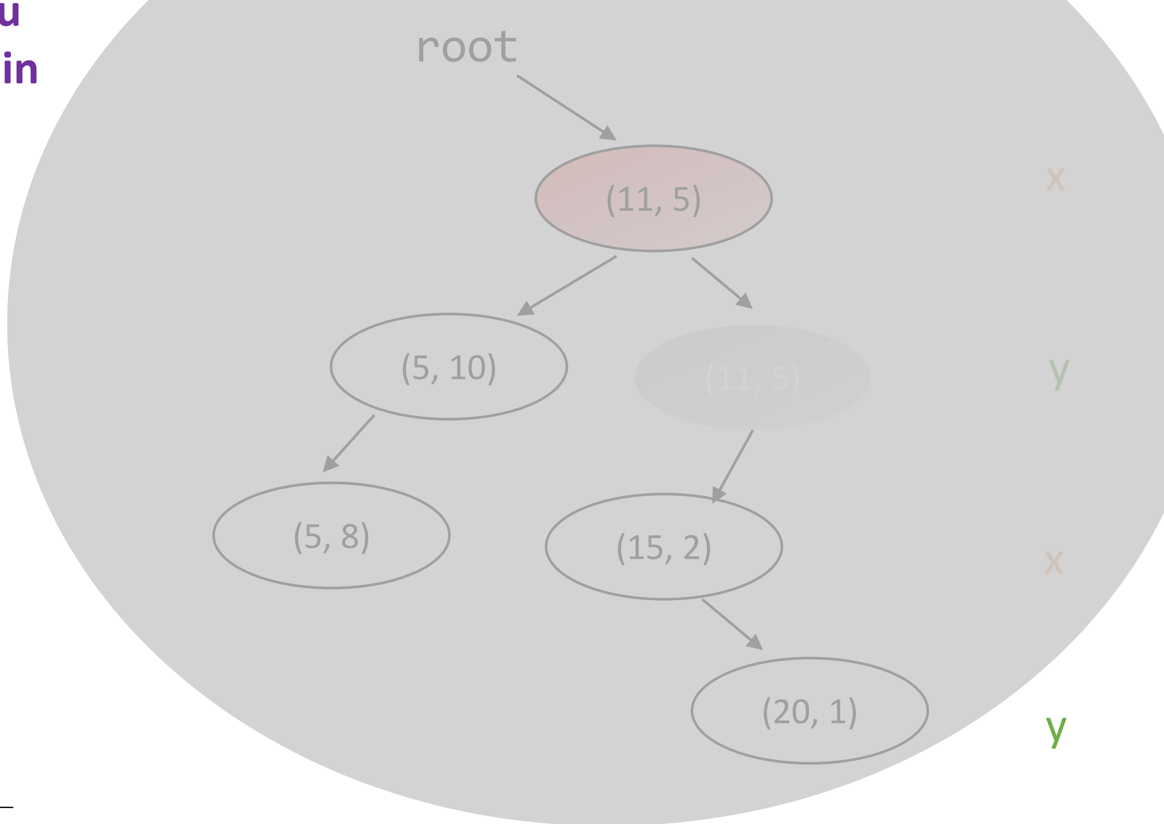
- Suppose that we want to delete (10, 20)

2D space

Same deal: how would you expect this to be rectified in our 2D space?



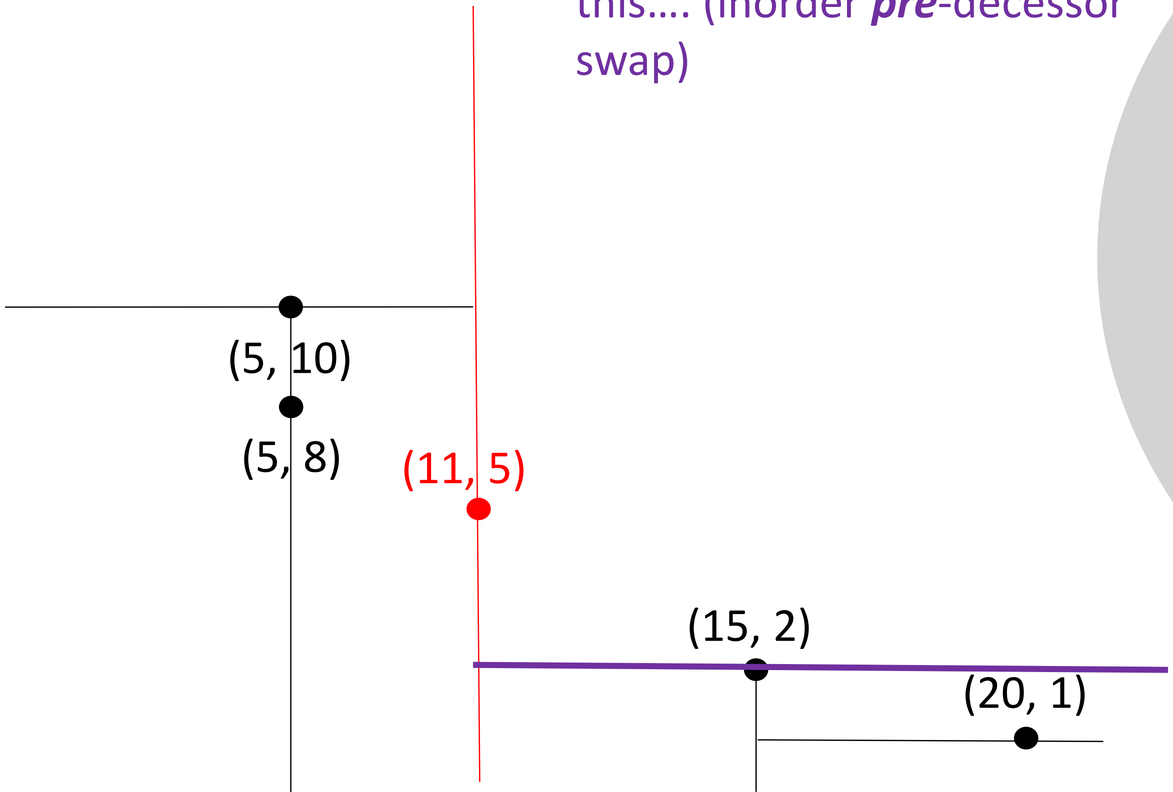
Corresponding KD-Tree

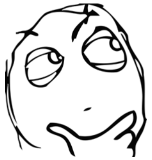


Deletion

- Suppose that we want to delete (10, 20)

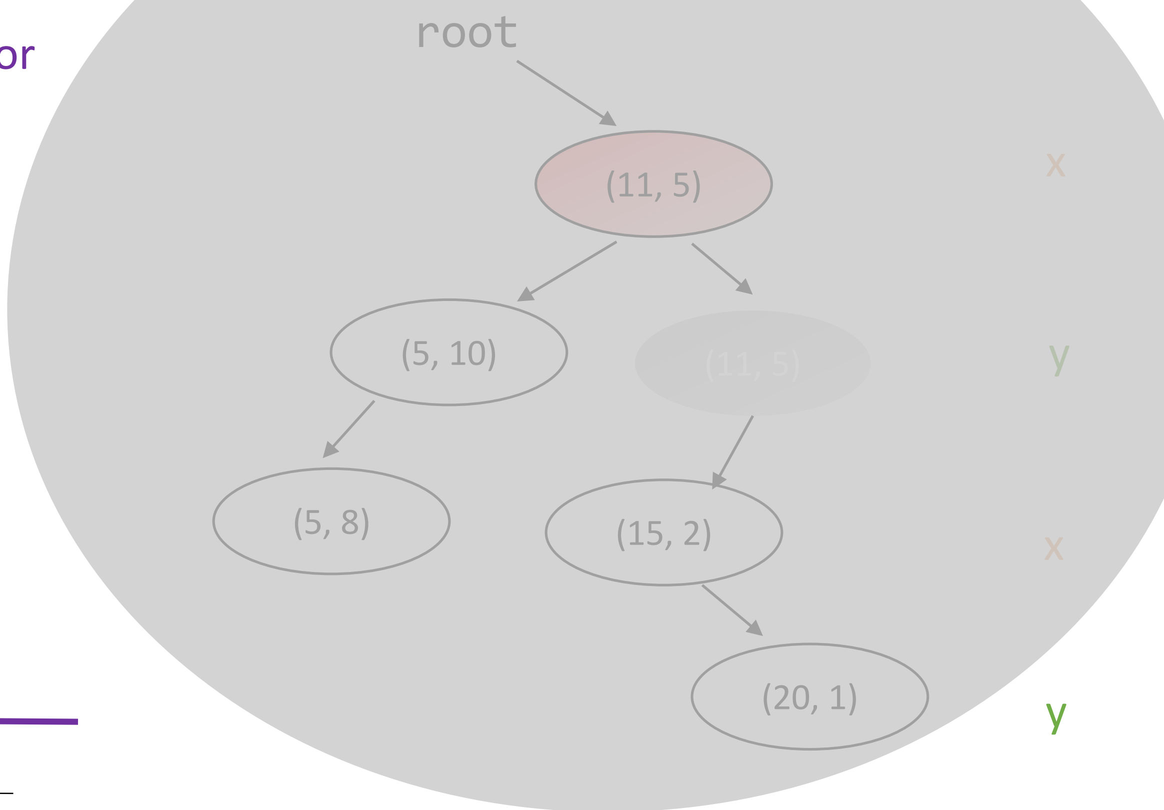
2D space





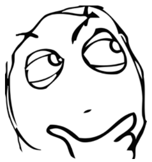
I *could* think about doing this…. (inorder ***pre***-decessor swap)

Corresponding KD-Tree



Deletion

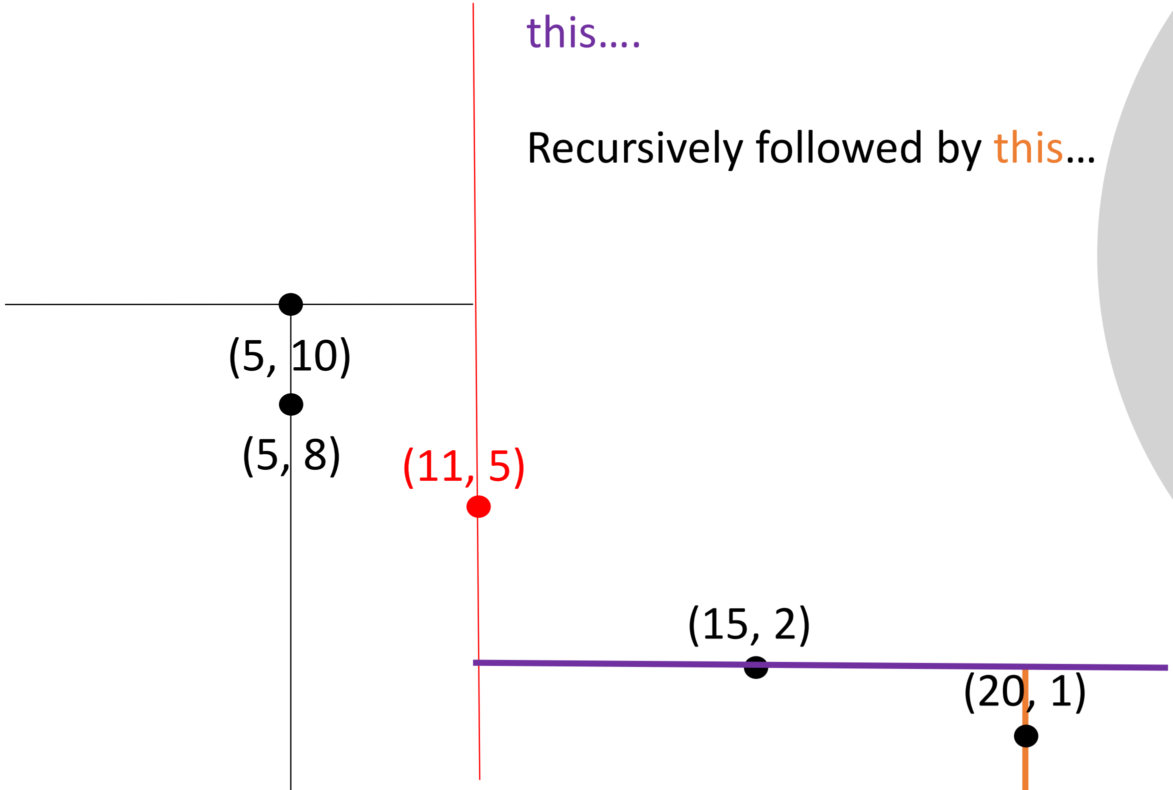
- Suppose that we want to delete (10, 20)

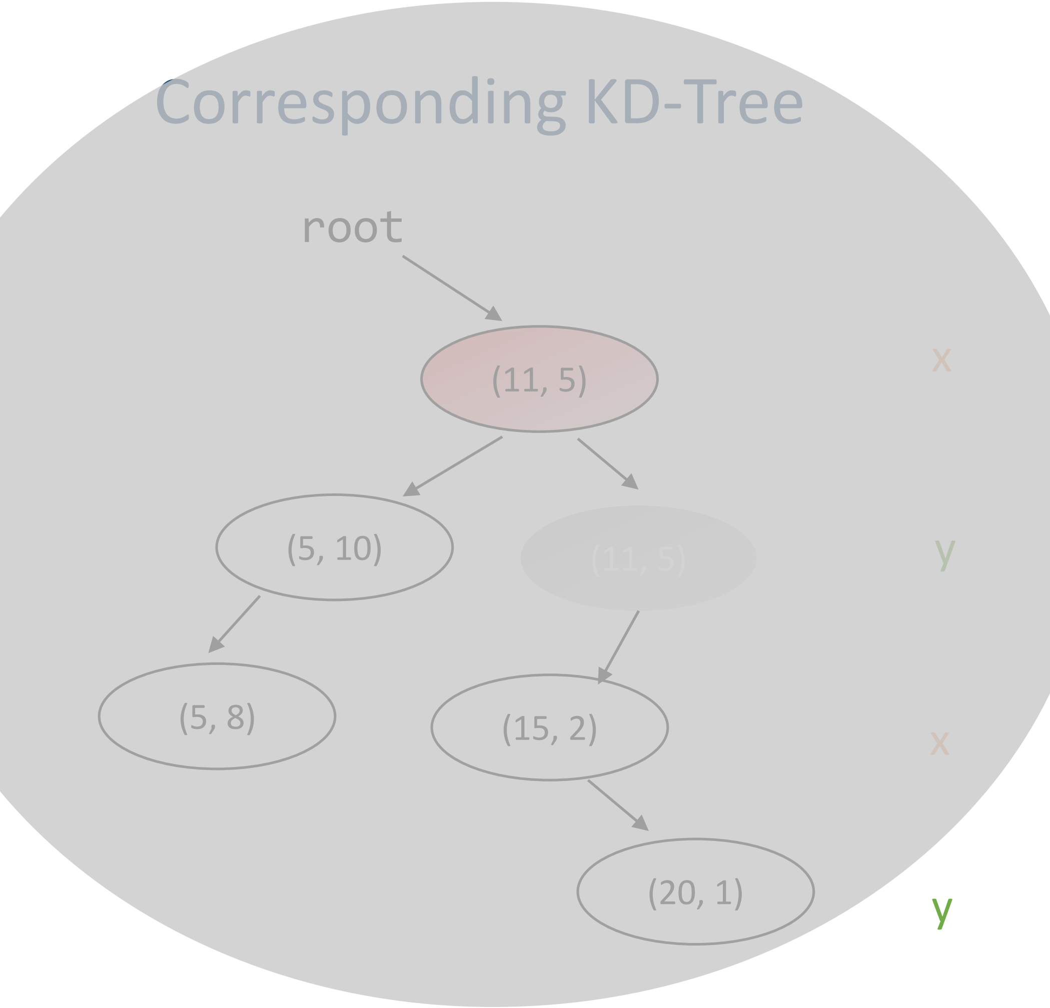


2D space

I *could* think about doing this….

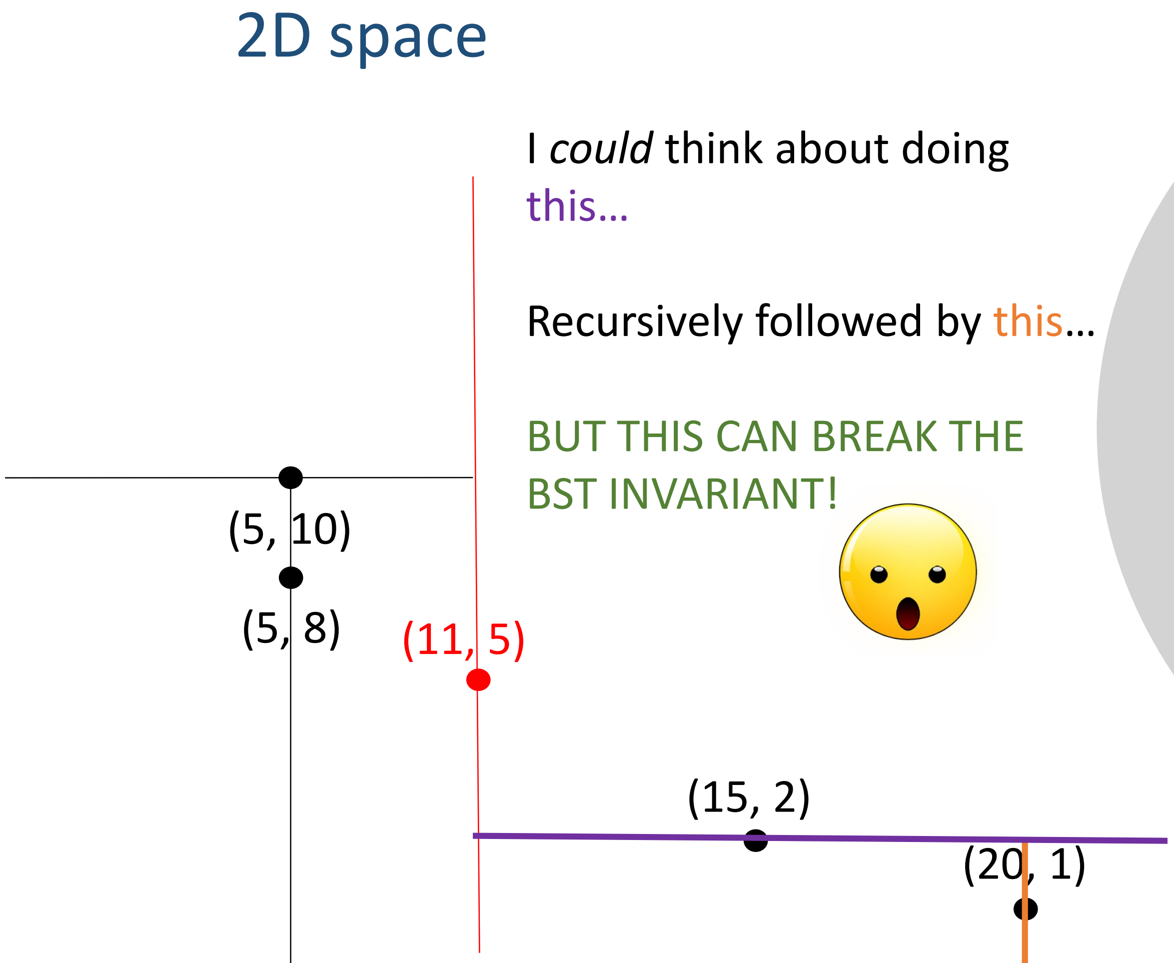
Recursively followed by this…

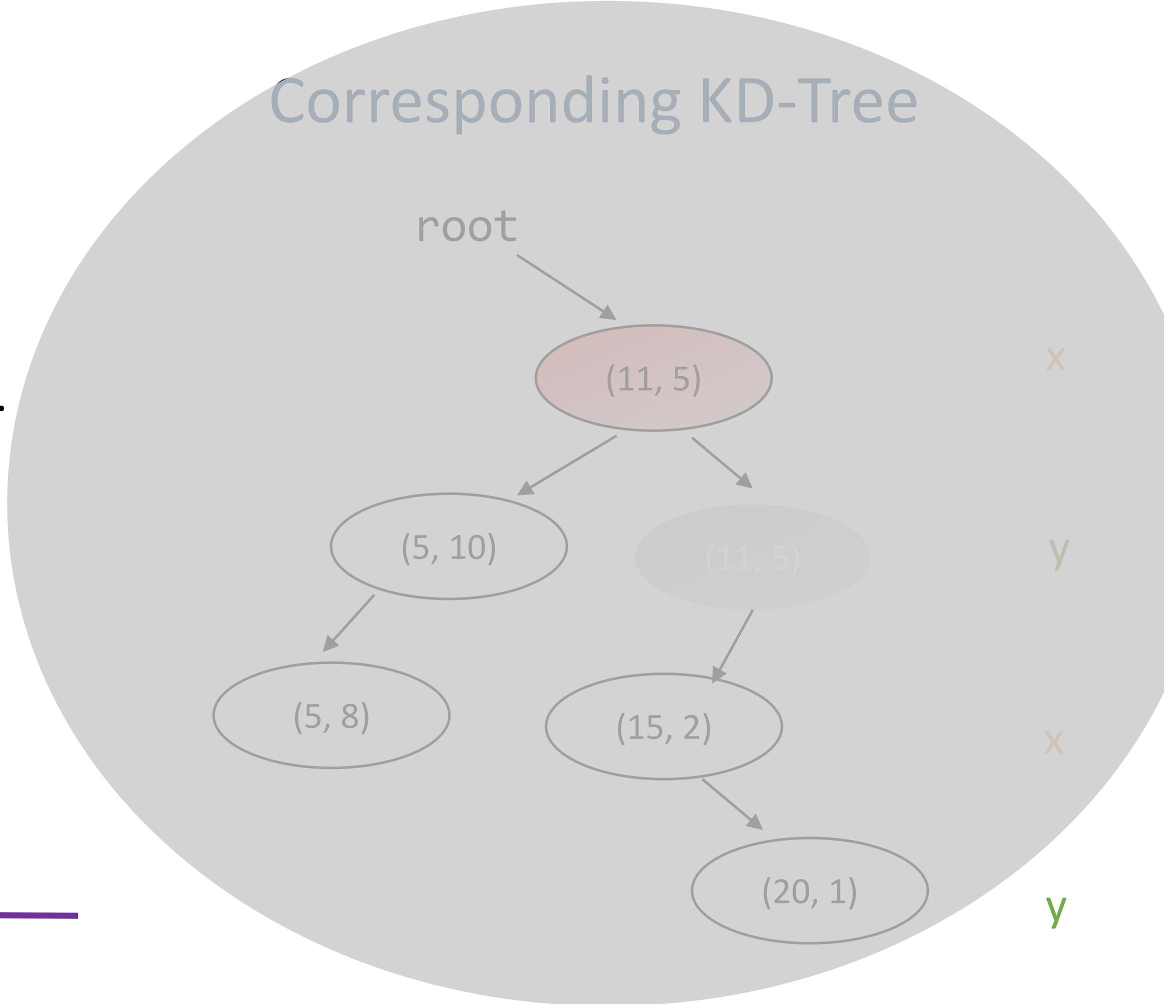




Deletion

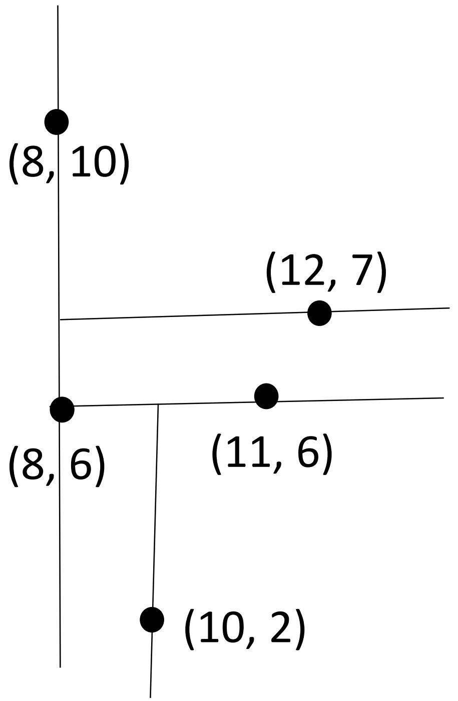
- Suppose that we want to delete (10, 20)

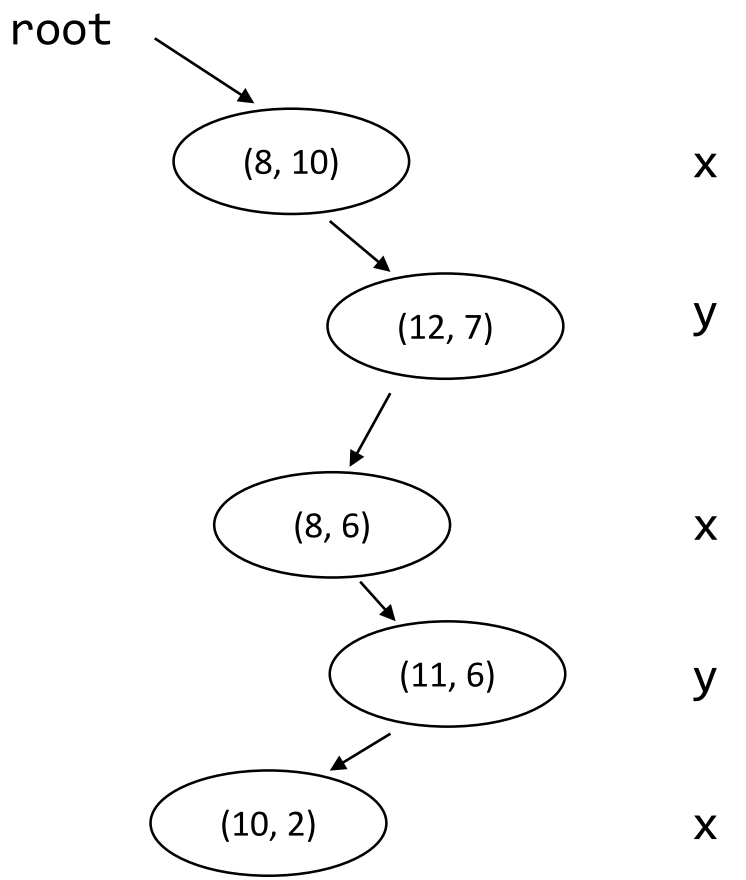




Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:

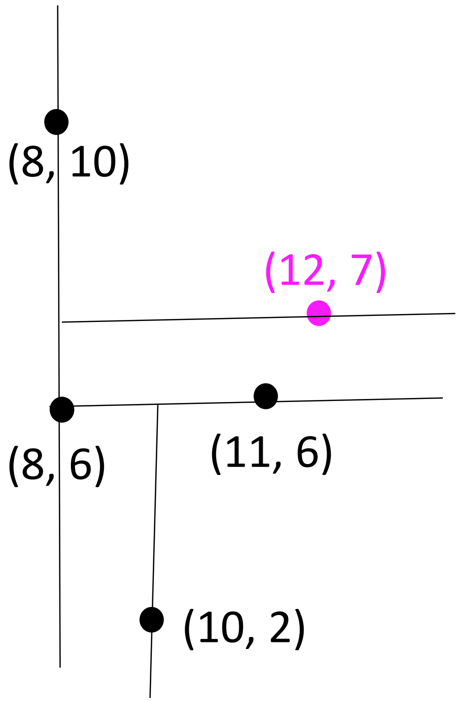


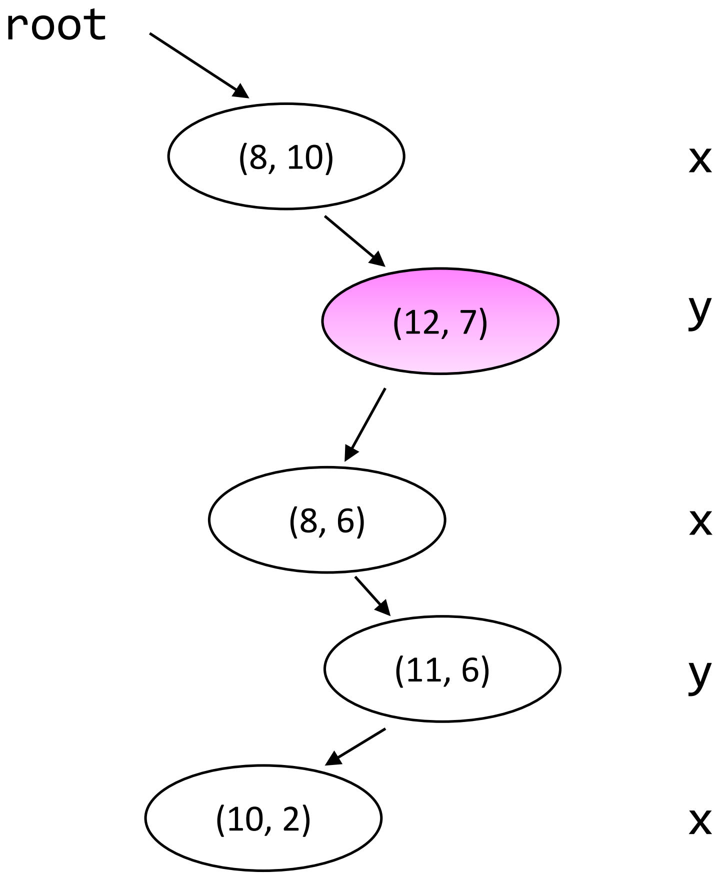


Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:

Task: Delete (12, 7)

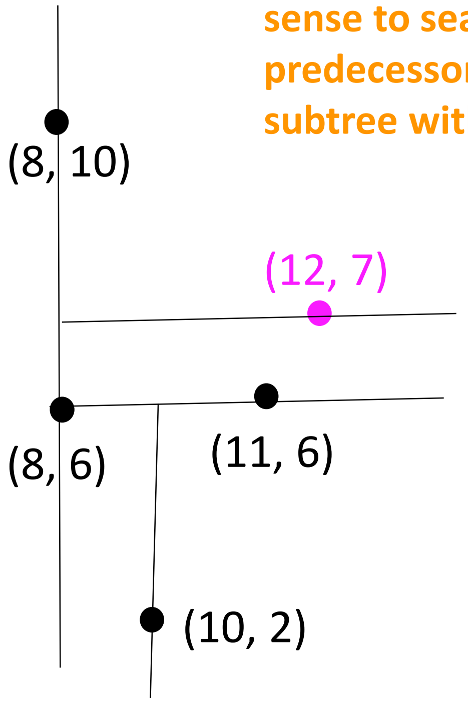


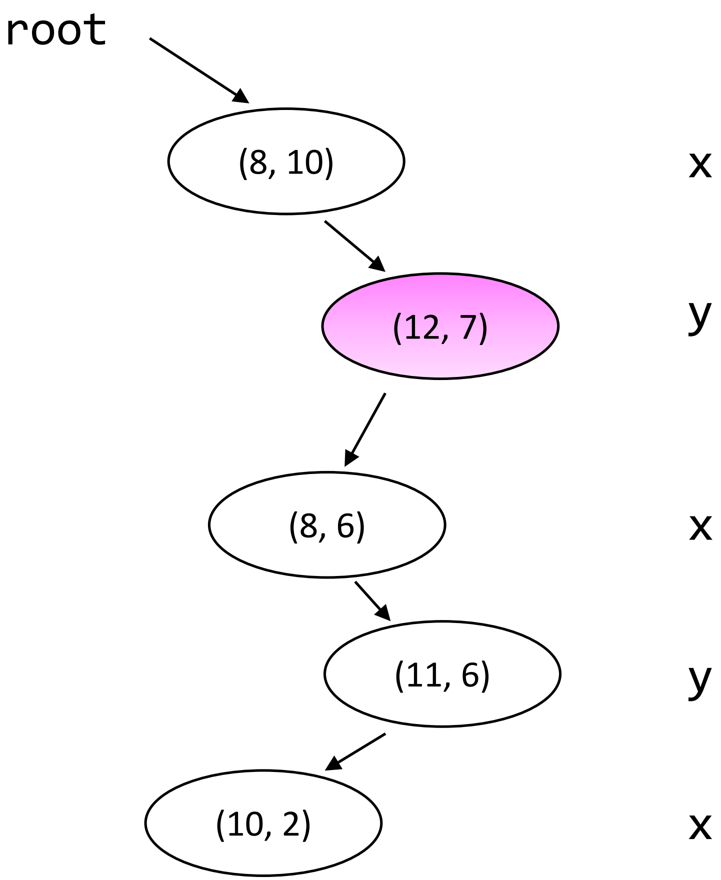


Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:

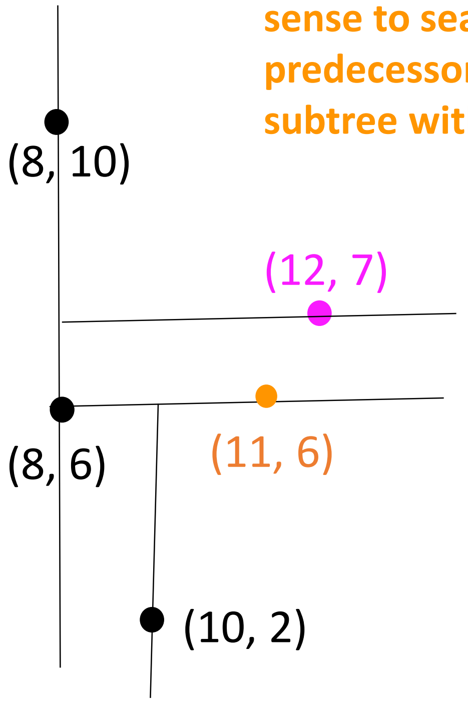
Since right subtree is null and we can't find an "inorder successor", **maybe it makes sense to search for the "inorder predecessor" (node in the left subtree with maximal y value)?**



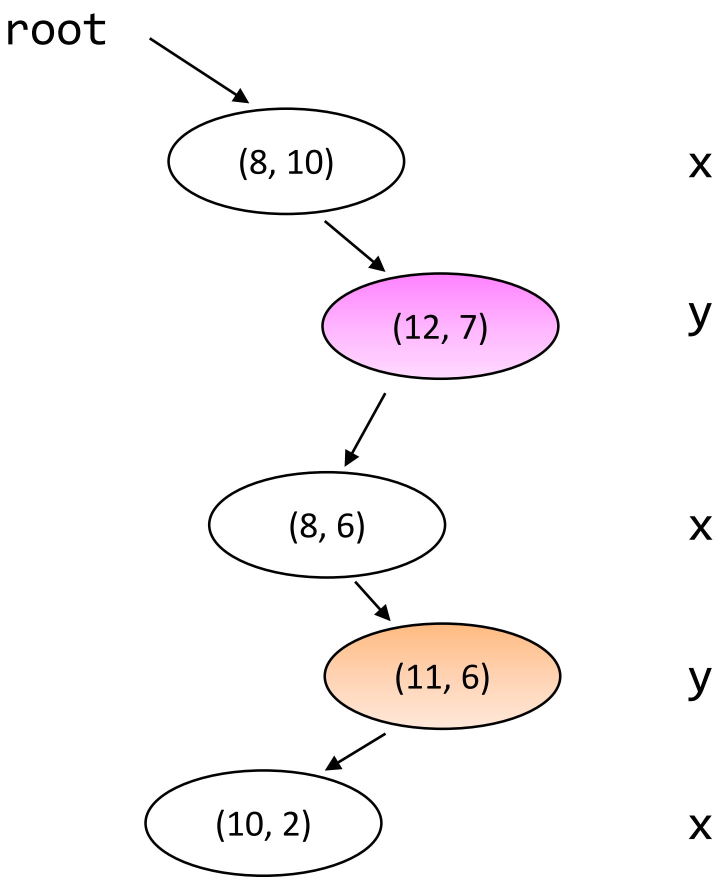


Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



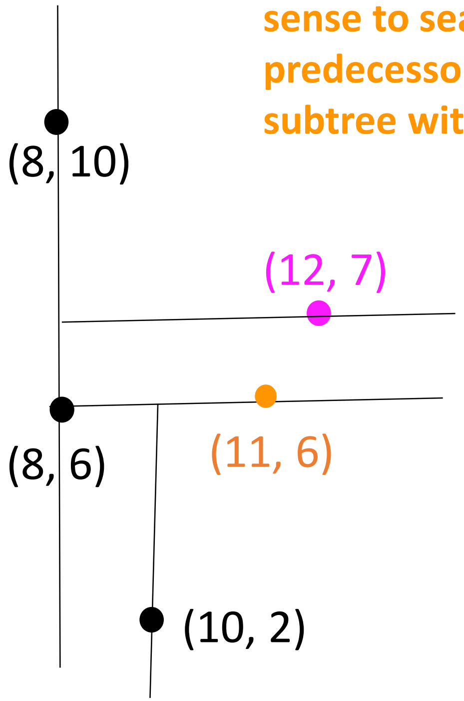
Since right subtree is null and we can't find an "inorder successor", **maybe it makes sense to search for the "inorder predecessor" (node in the left subtree with maximal y value)?**

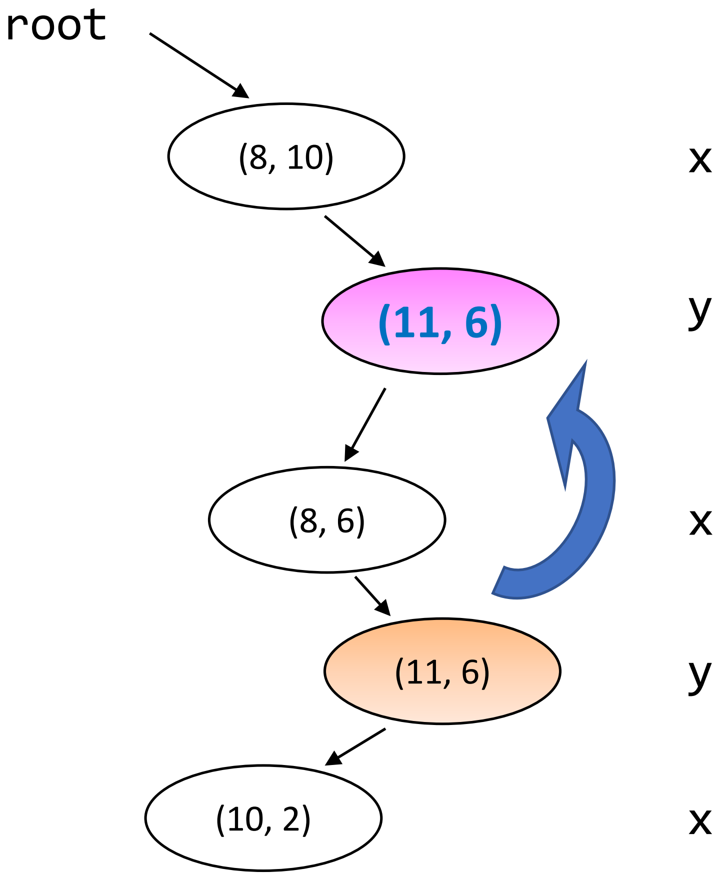


Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:

Since right subtree is null and we can't find an "inorder successor", **maybe it makes sense to search for the "inorder predecessor" (node in the left subtree with maximal y value)?**

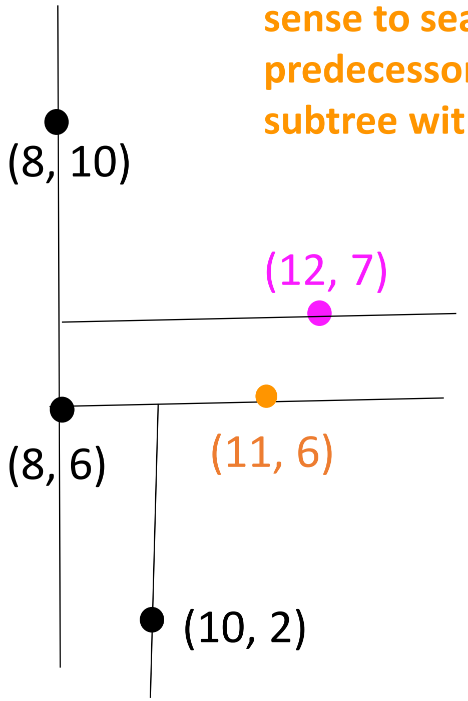


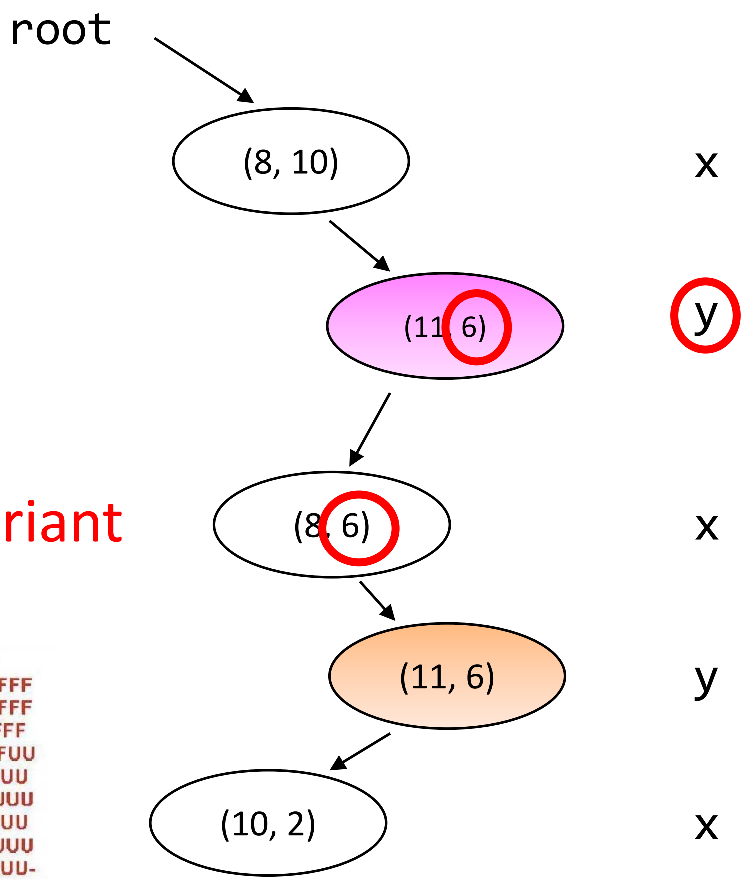


Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:

Since right subtree is null and we can't find an "inorder successor", **maybe it makes sense to search for the "inorder predecessor" (node in the left subtree with maximal y value)?**

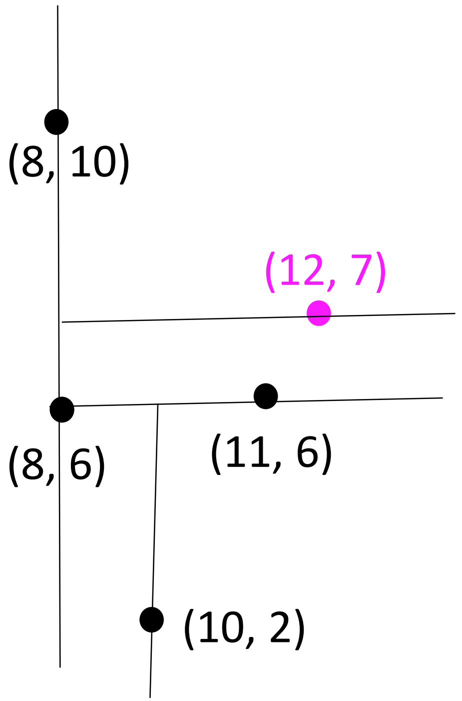




BST invariant broken!

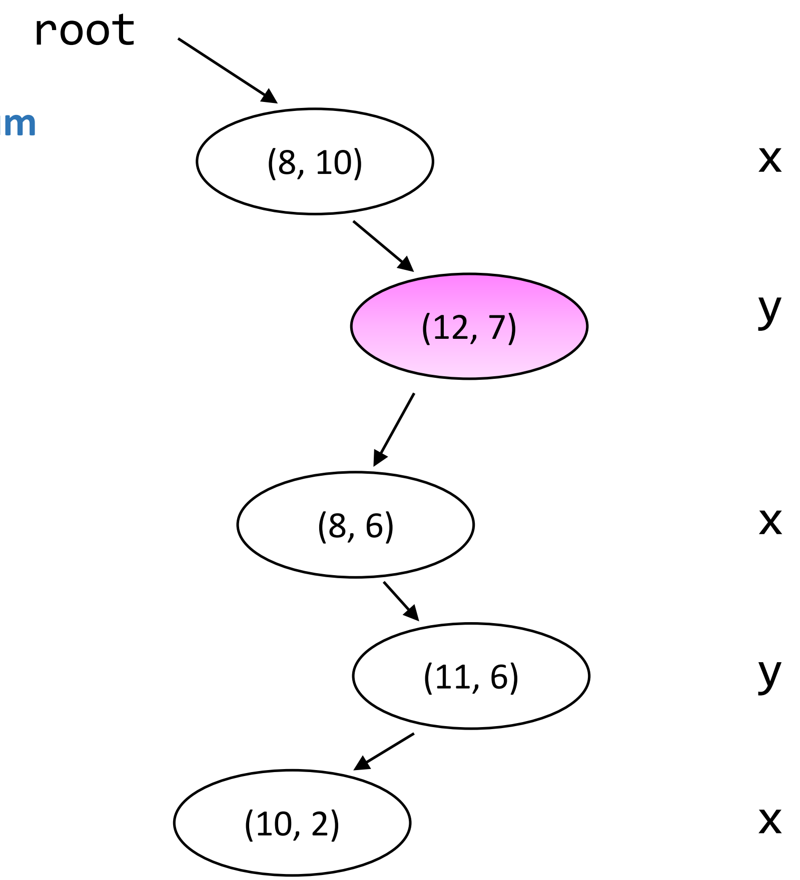
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



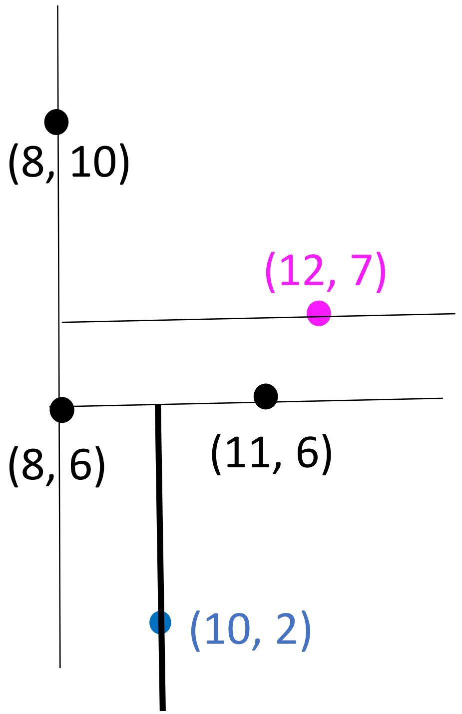
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**



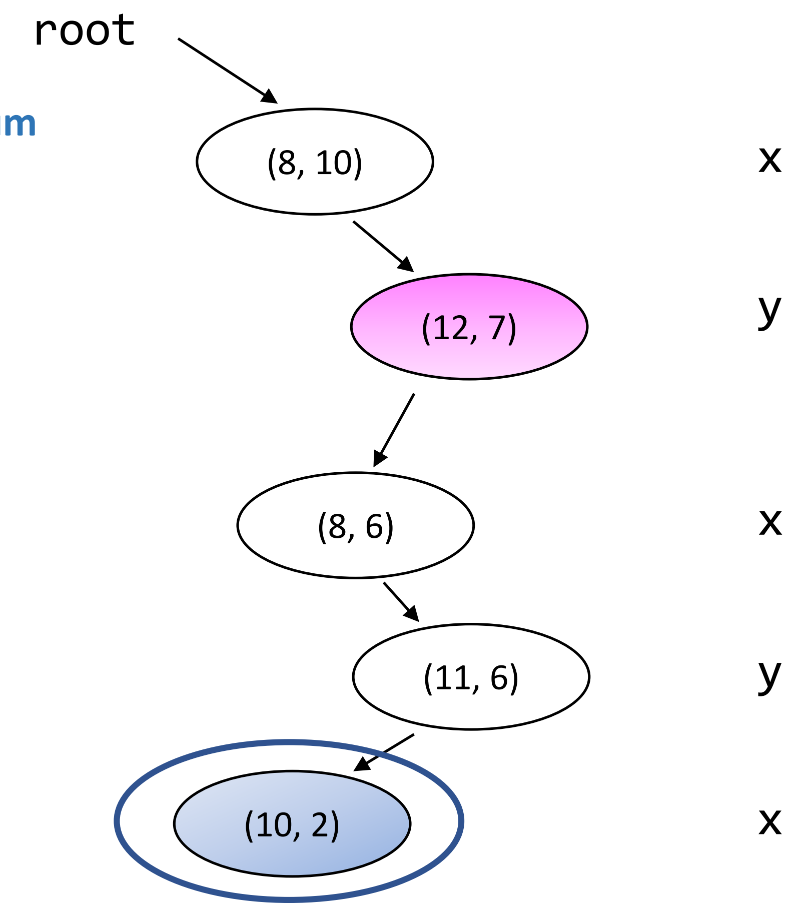
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



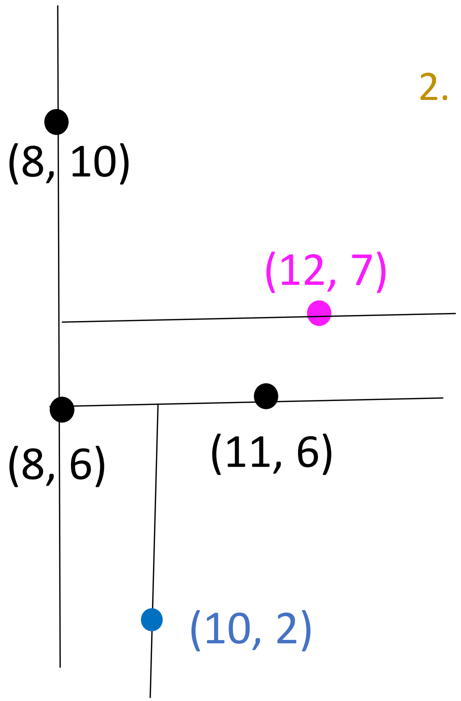
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**



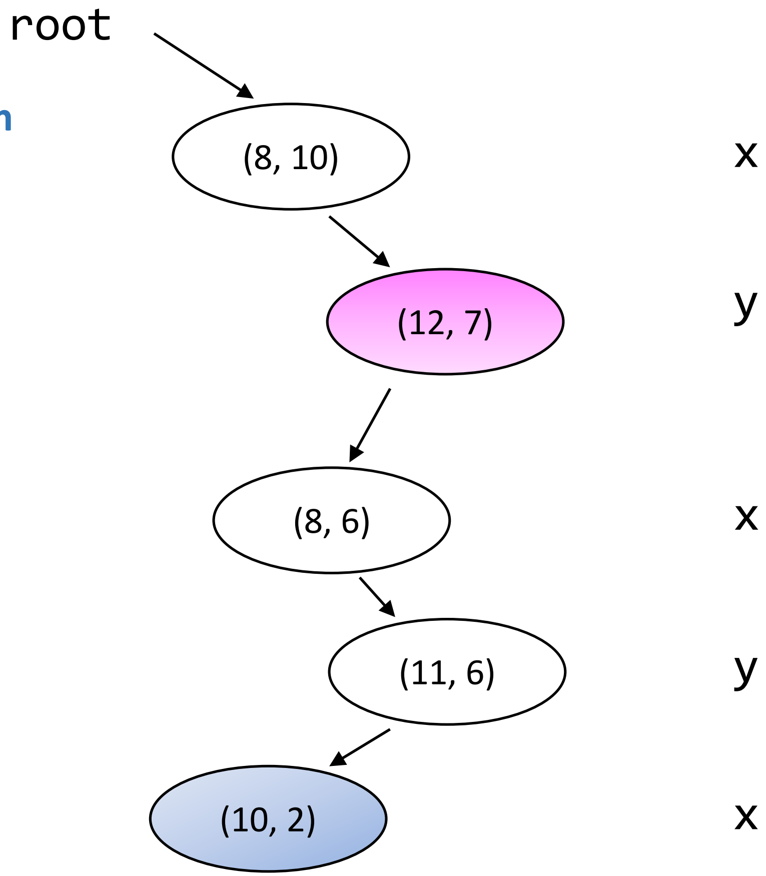
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



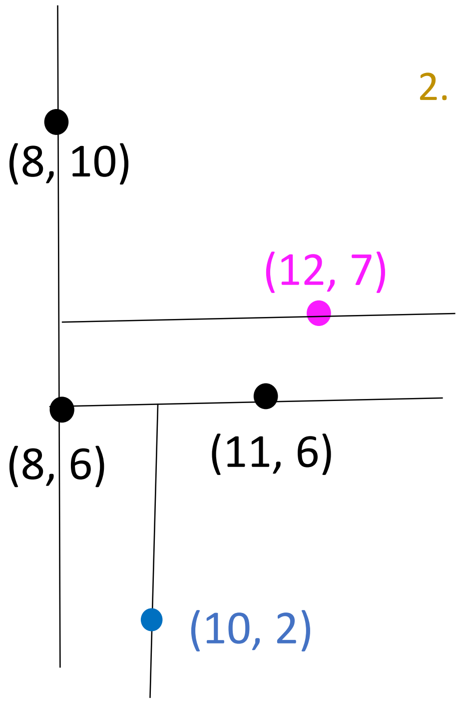
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.



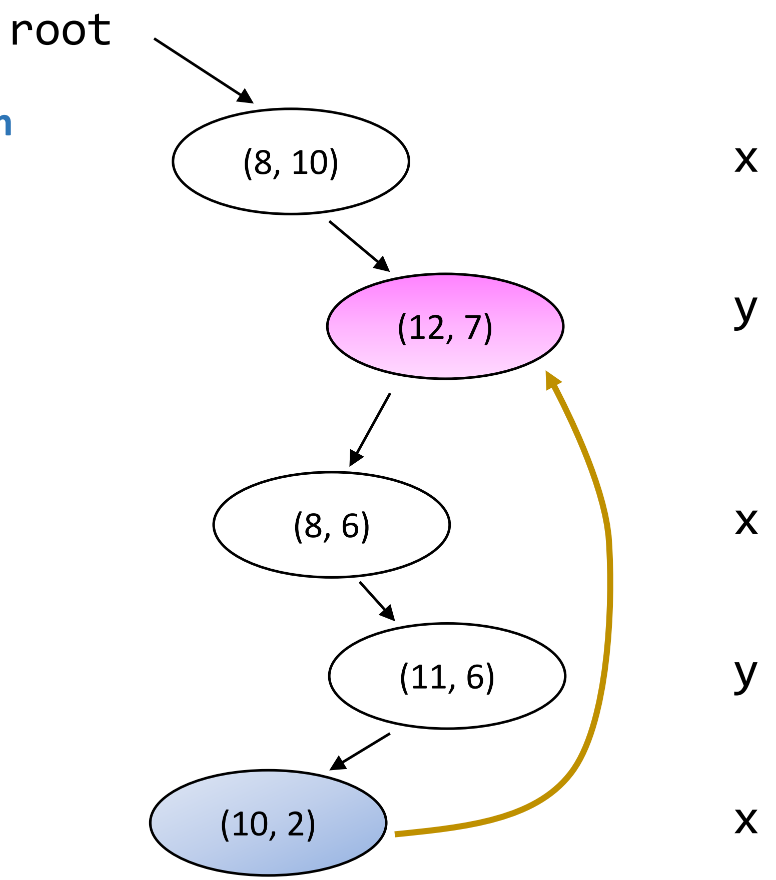
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



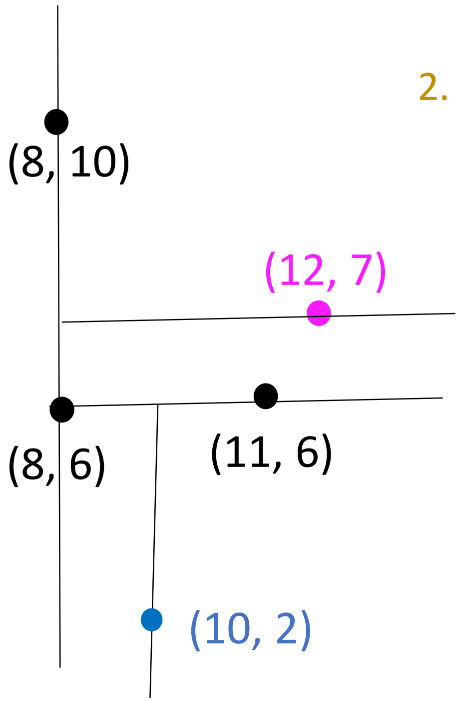
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.



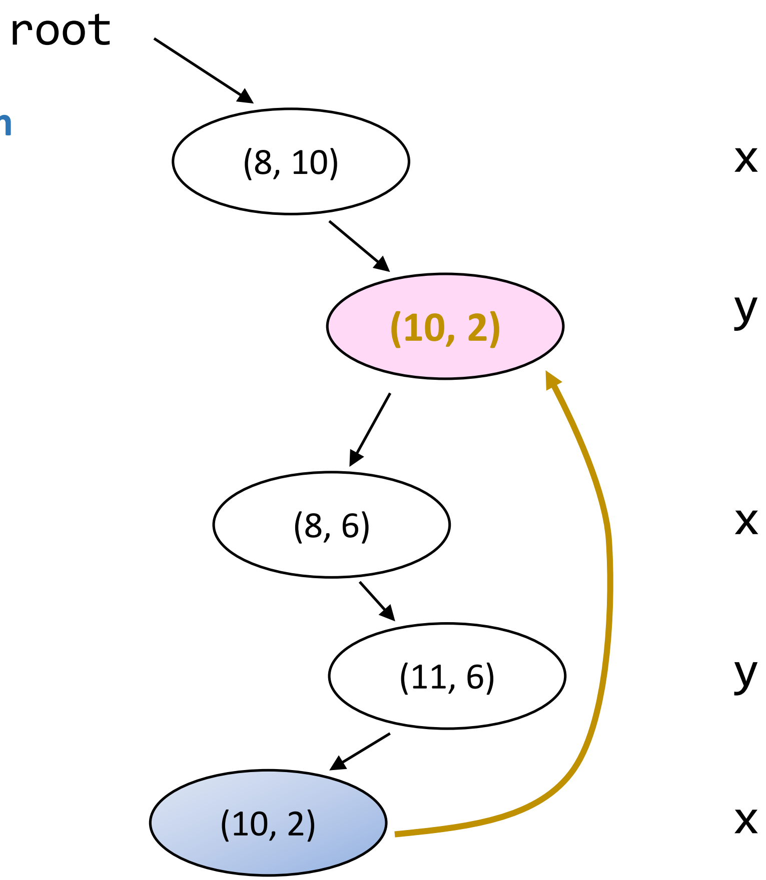
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



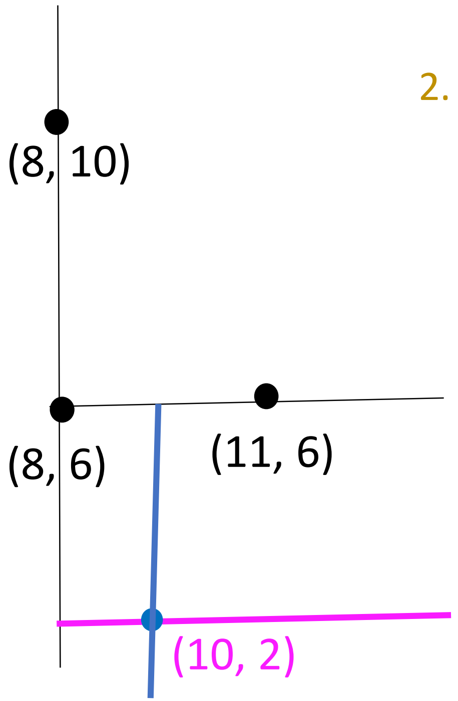
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.



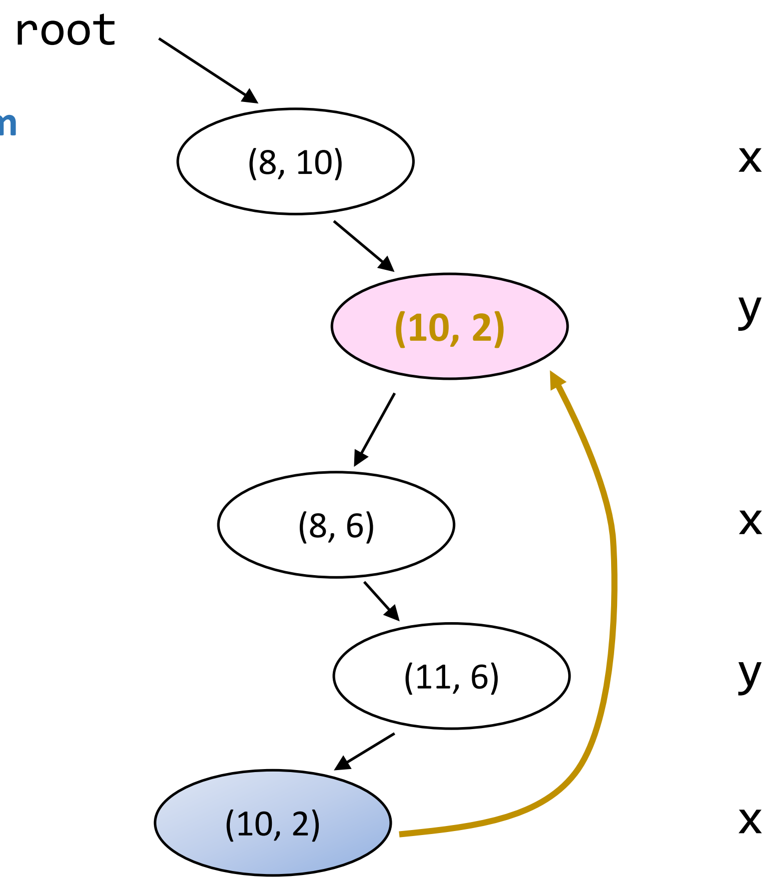
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



Solution:

1. Find the point with the **minimum** **current dimension value** from the **left subtree.**
2. Copy that point to current node.

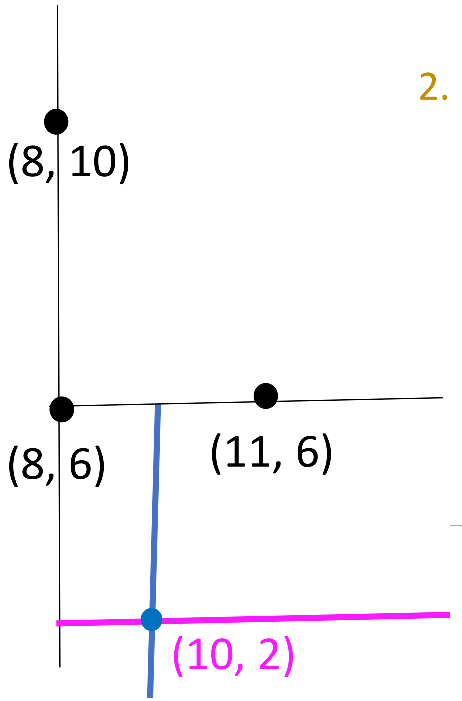


Breaking the invariant

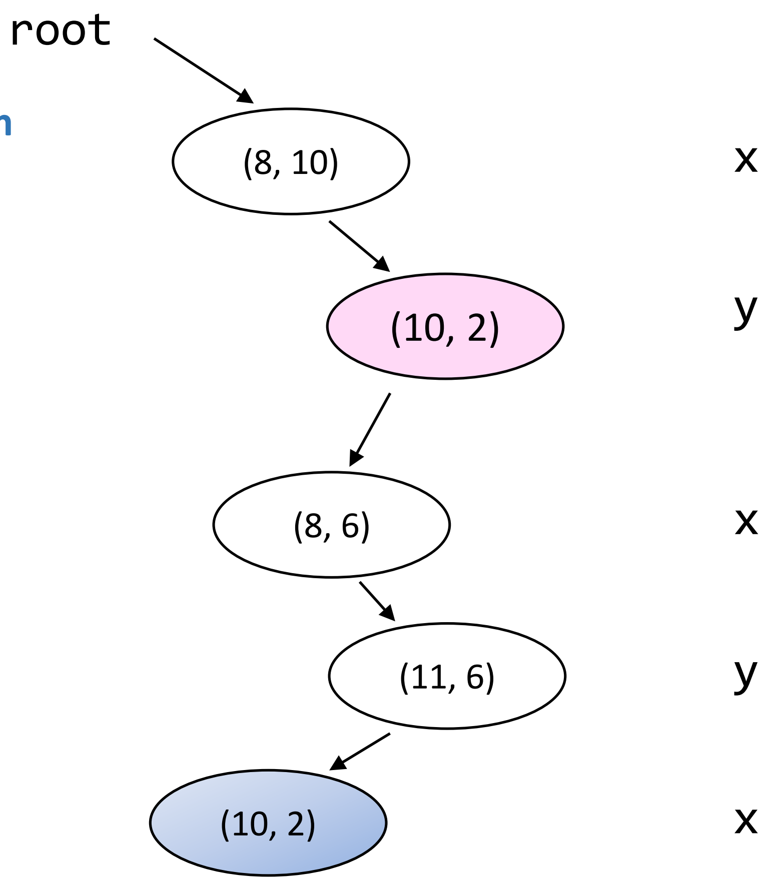
- Consider the spatial decomposition and KD-Tree that follow:

Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.

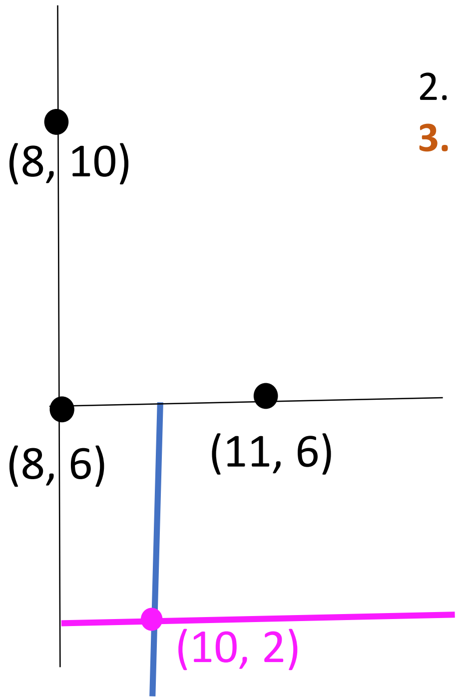


This "dual identity" of (10, 2) can't last long...



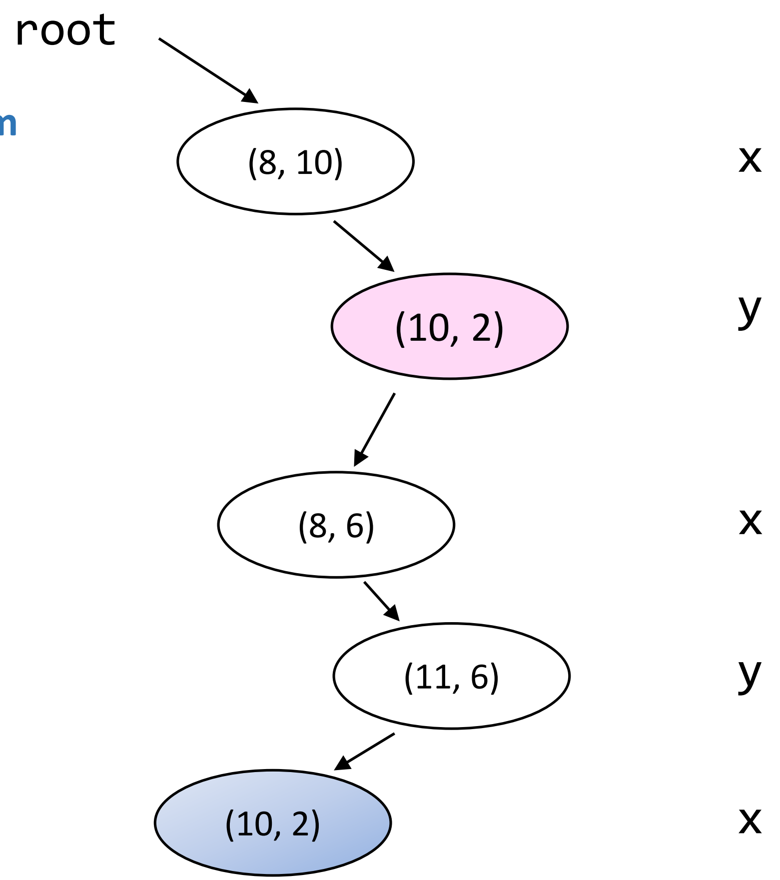
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



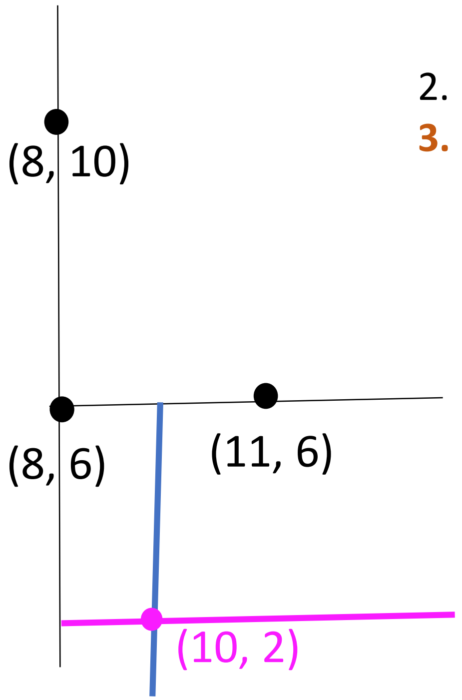
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.
3. **Make left subtree the right subtree!** (left is now `null`)



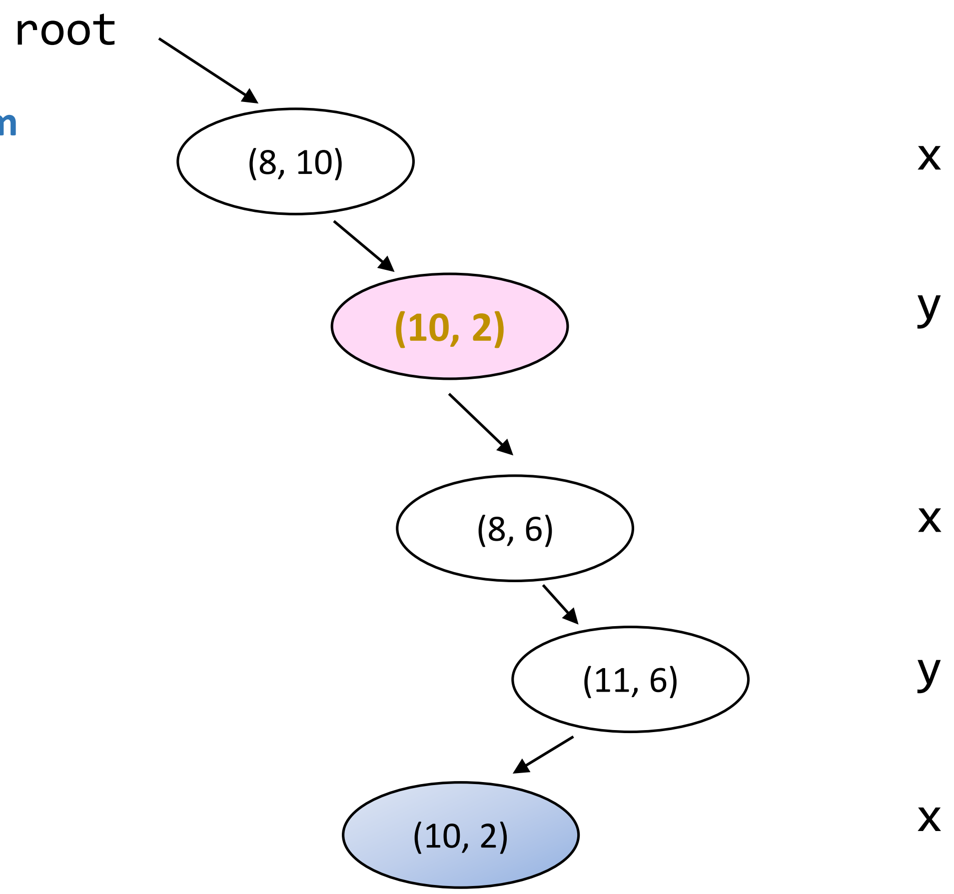
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



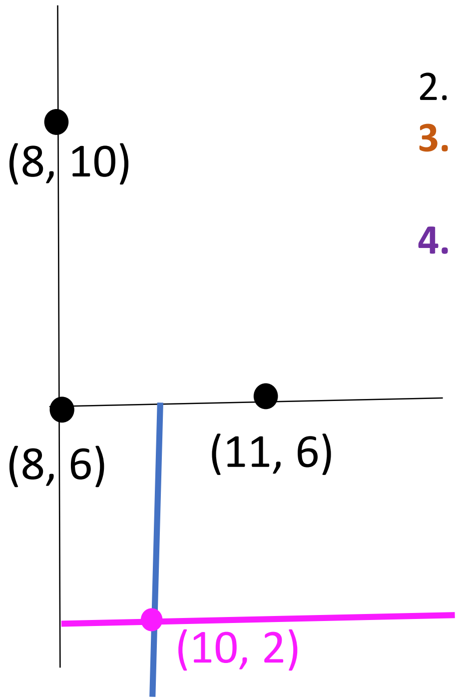
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.
3. **Make left subtree the right subtree!** (left is now `null`)



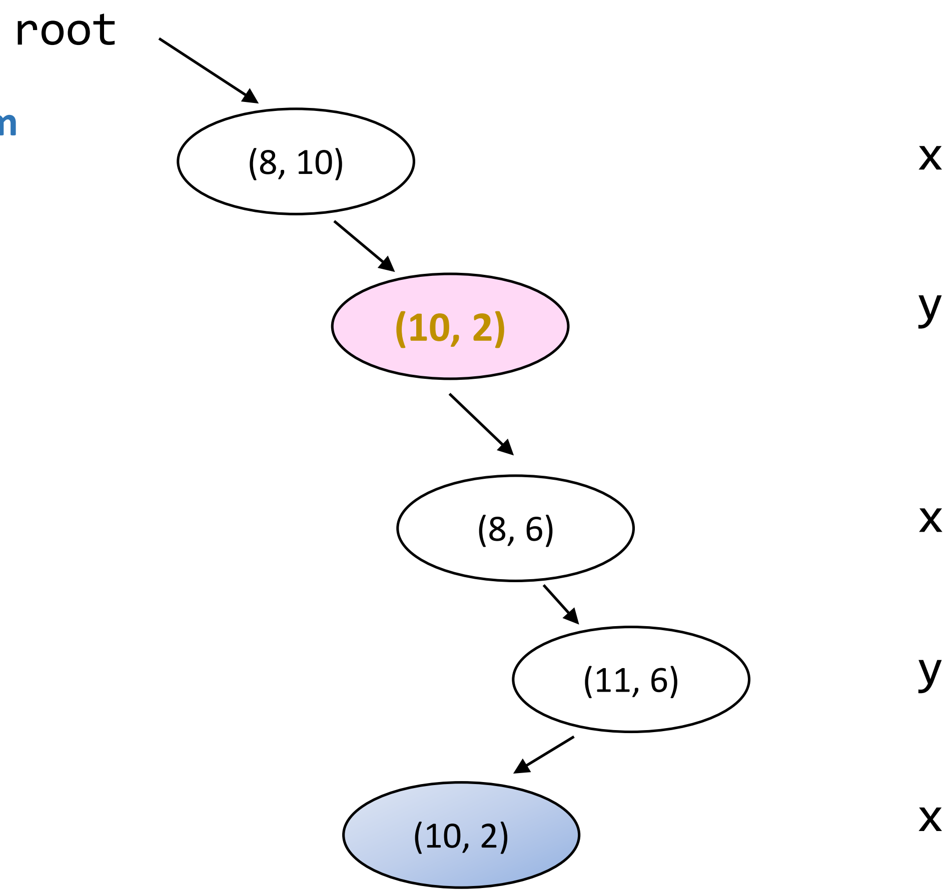
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



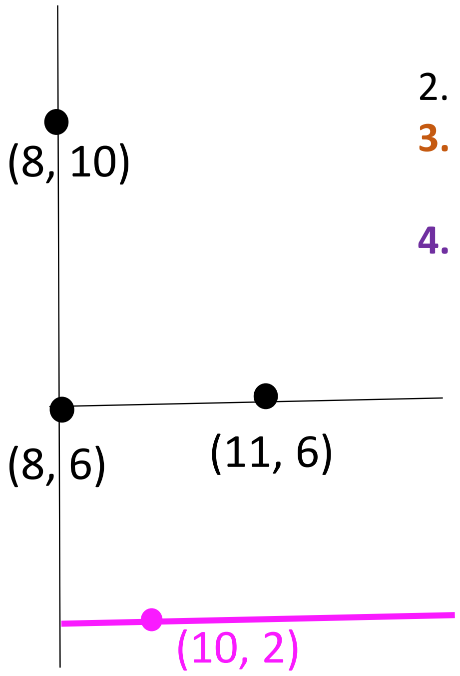
Solution:

1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.
3. **Make left subtree the right subtree!** (left is now `null`)
4. **Recursively delete** the node whose key you copied



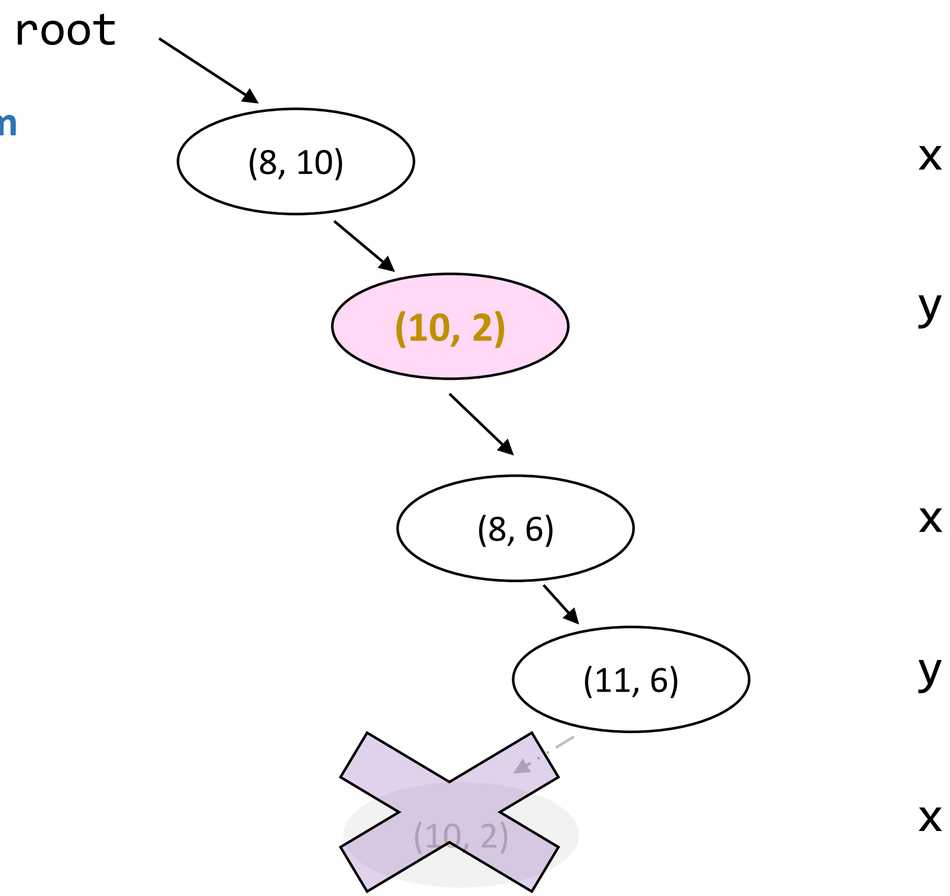
Breaking the invariant

- Consider the spatial decomposition and KD-Tree that follow:



Solution:

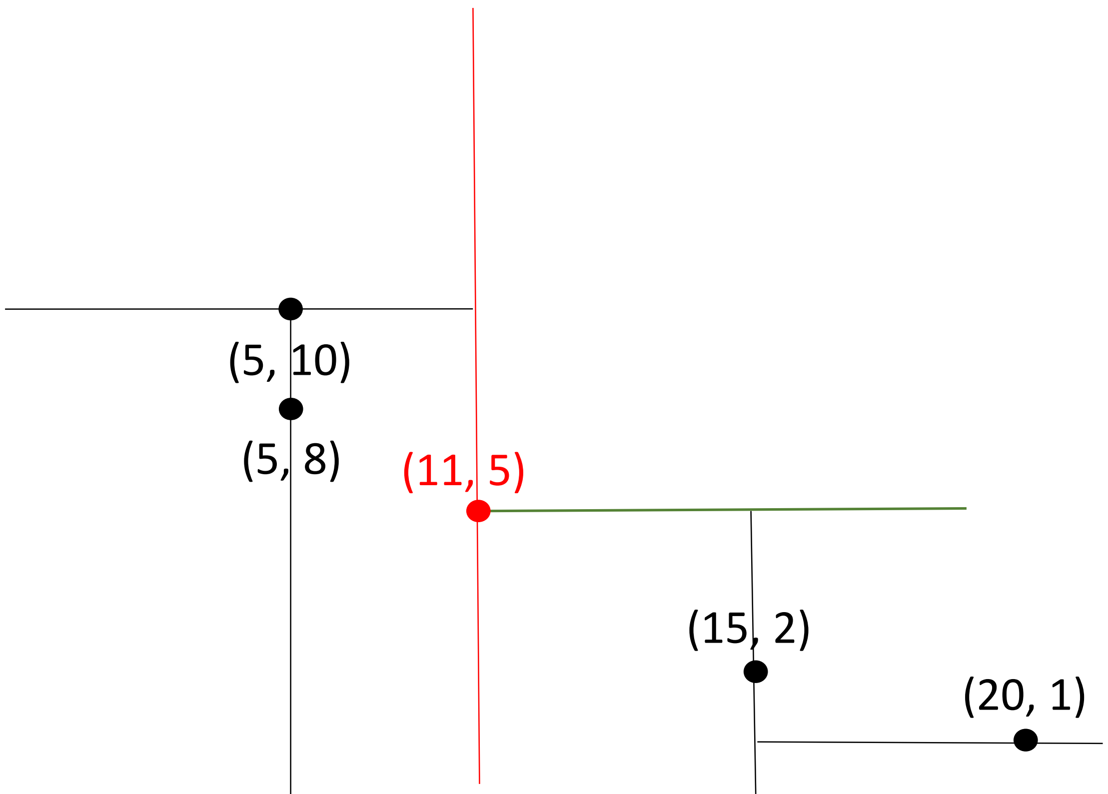
1. Find the point with the **minimum current dimension value** from the **left subtree.**
2. Copy that point to current node.
3. **Make left subtree the right subtree!** (left is now `null`)
4. **Recursively delete** the node whose key you copied



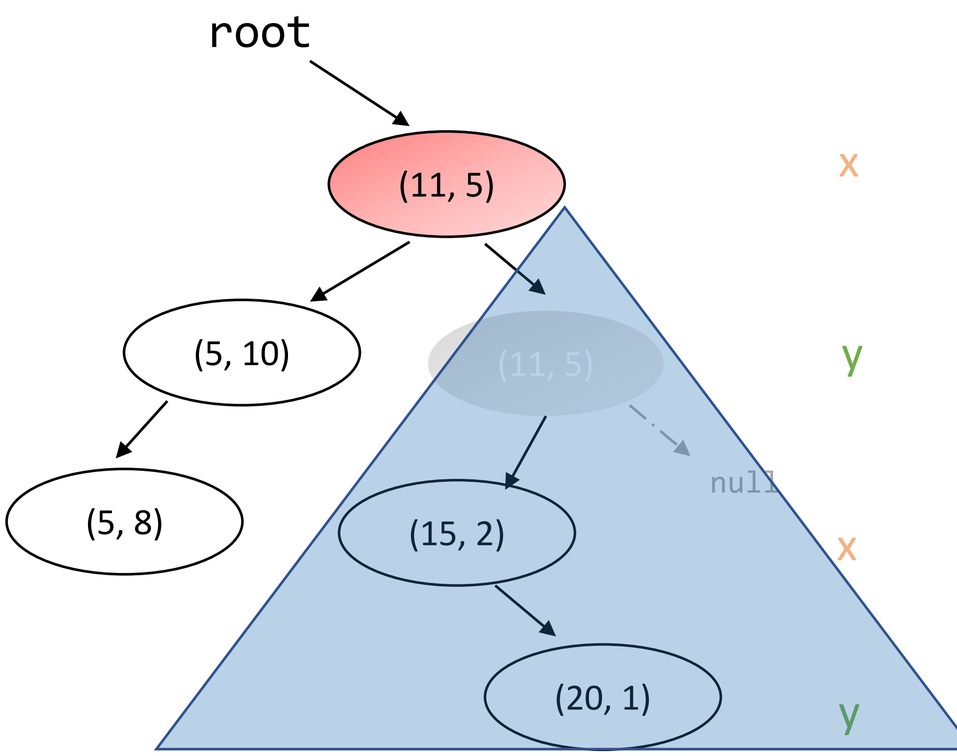
Deletion

- Reminder: we are faced with deleting (11, 5) from the root's right subtree

2D space



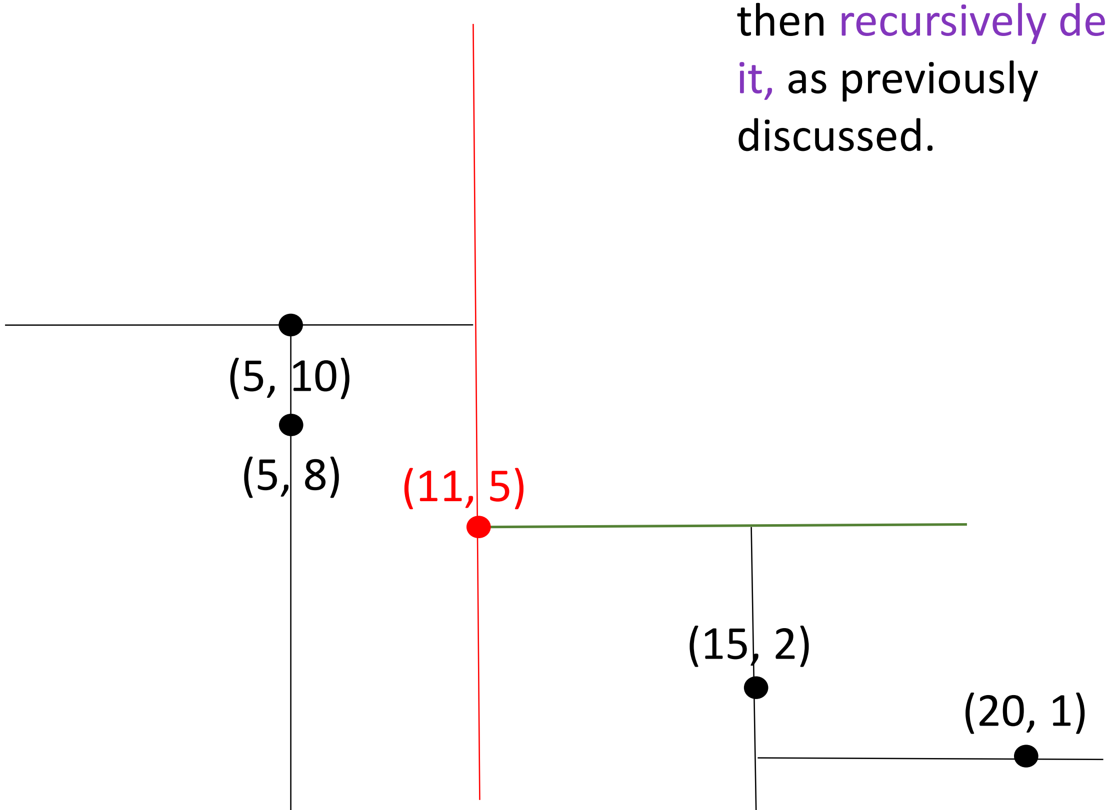
Corresponding KD-Tree



Deletion

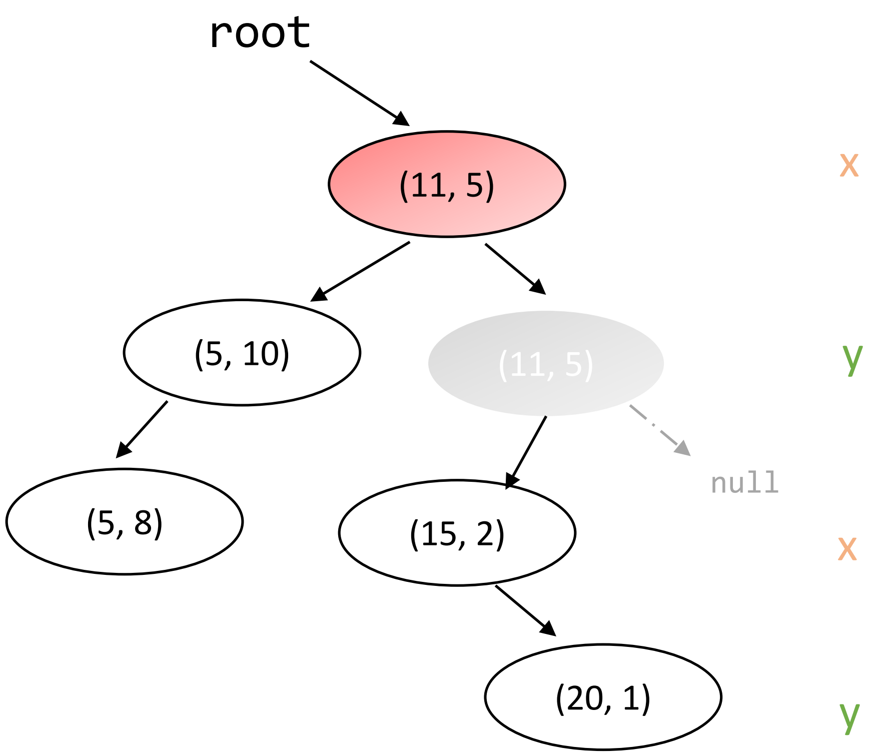
- Reminder: we are faced with deleting (11, 5) from the root's right subtree

2D space



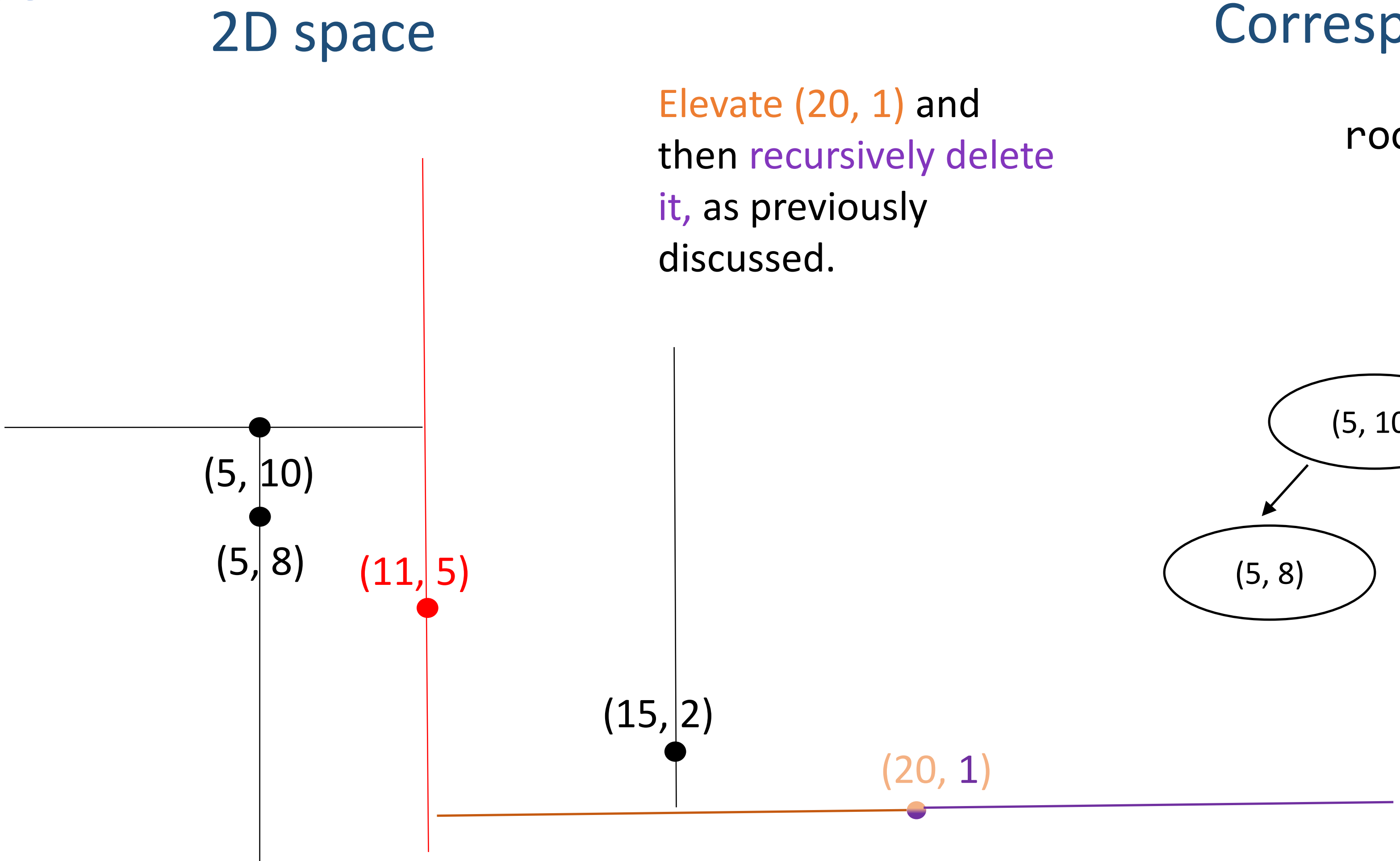
Elevate (20, 1) and then recursively delete it, as previously discussed.

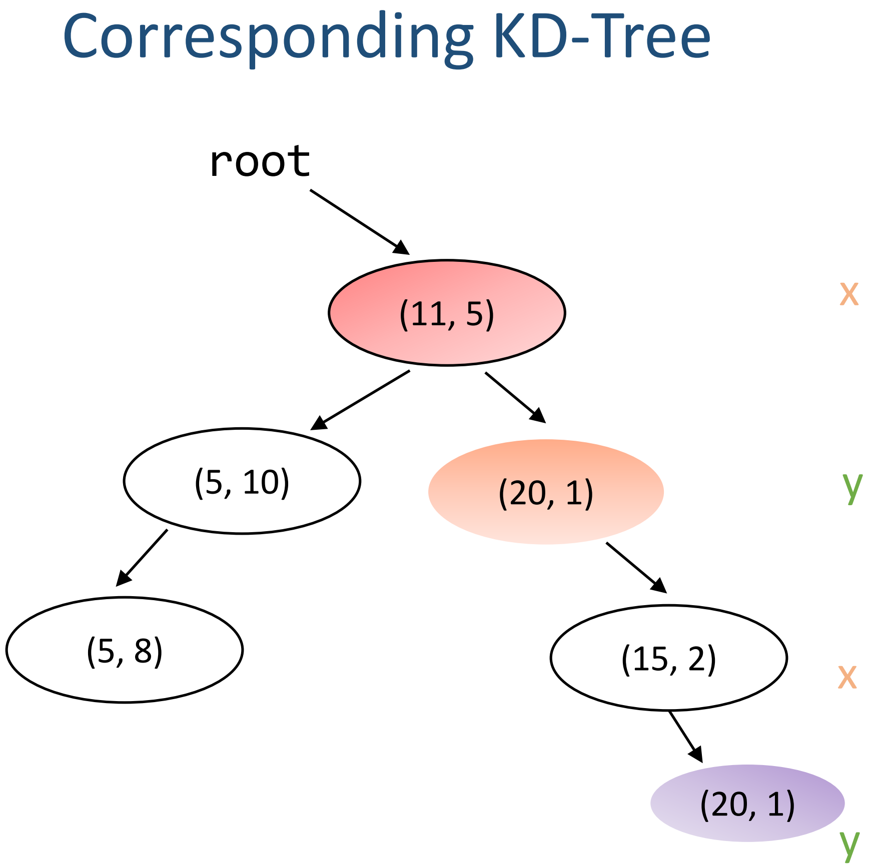
Corresponding KD-Tree



Deletion

- Reminder: we are faced with deleting (11, 5) from the root's right subtree



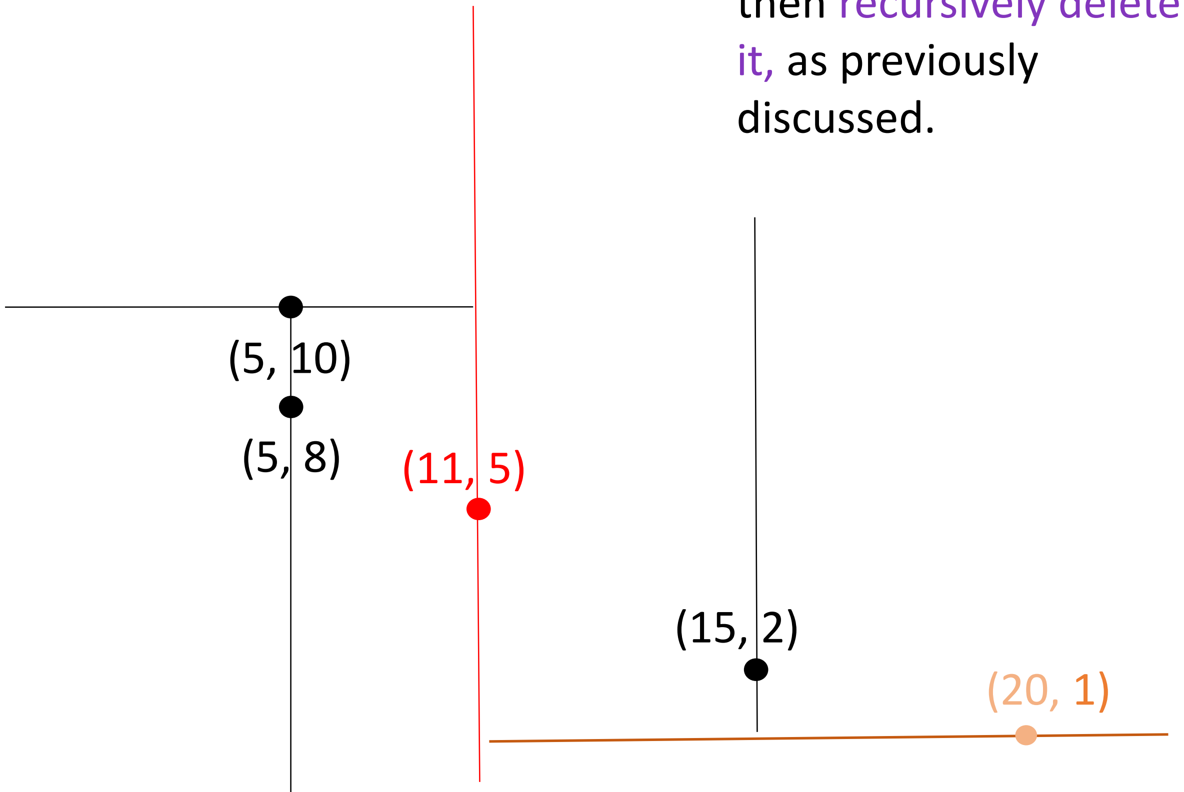


Deletion

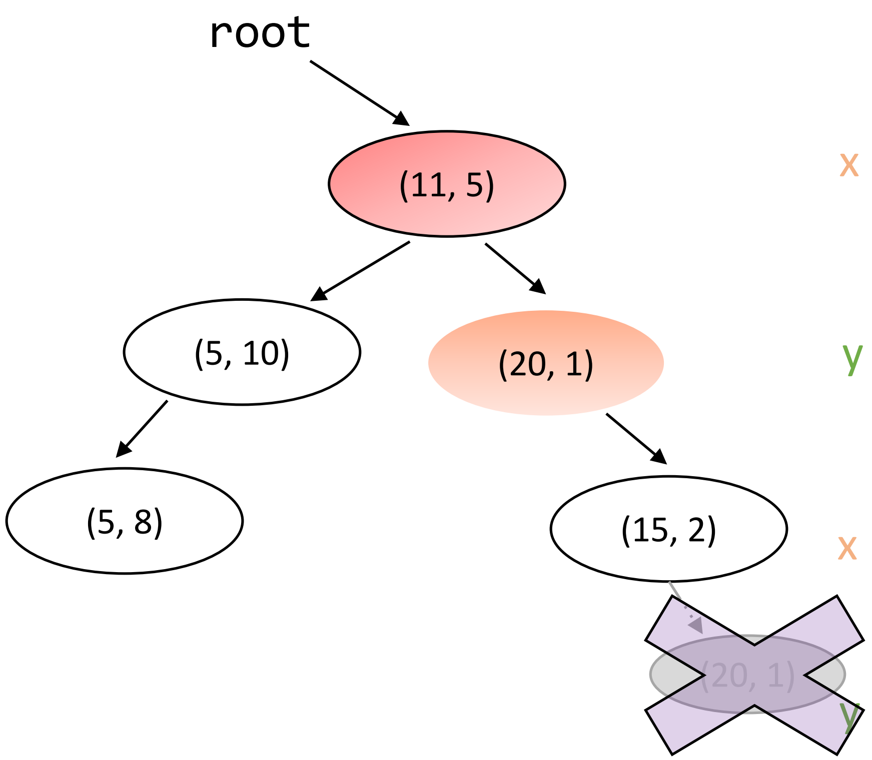
- Reminder: we are faced with deleting (11, 5) from the root's right subtree

2D space

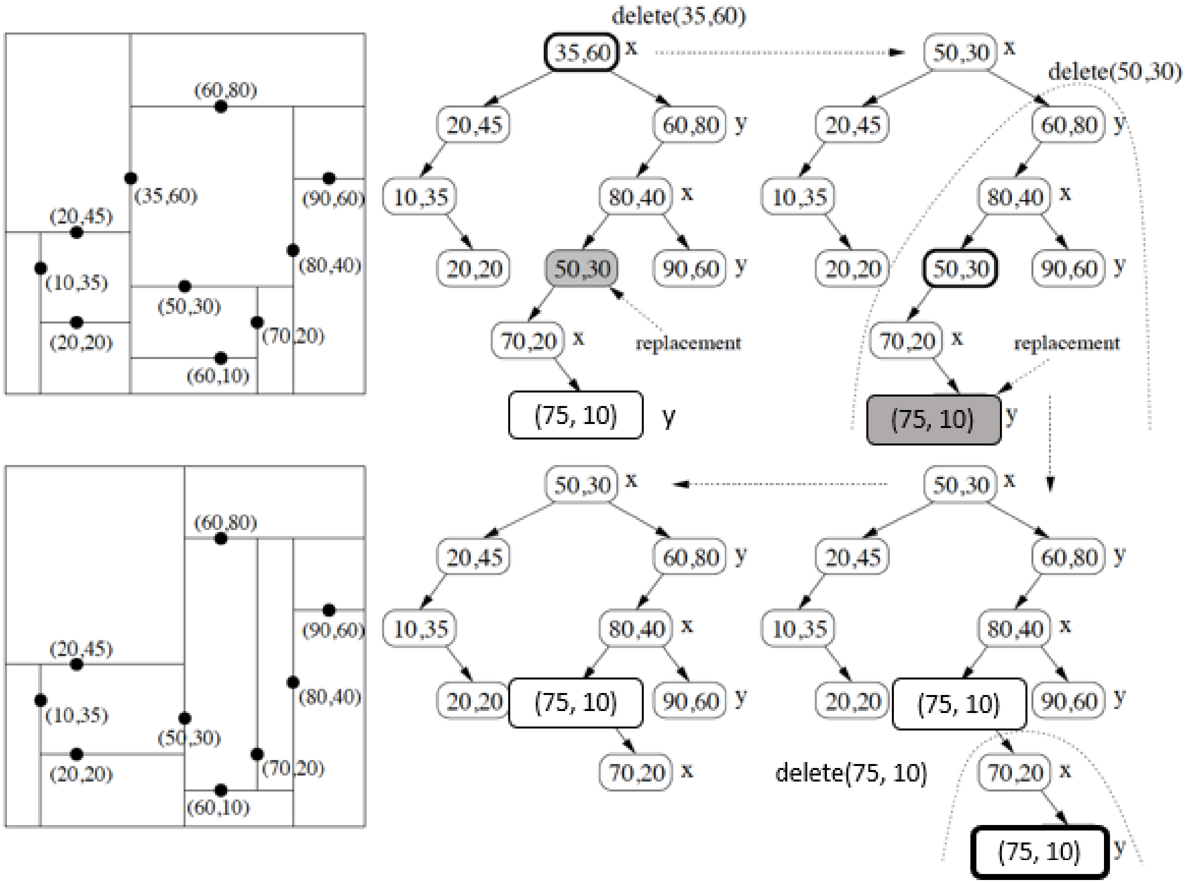
Elevate (20, 1) and then recursively delete it, as previously discussed.



Corresponding KD-Tree



A more complex deletion



Search

- Search works in the exact same way as insertion.
- Since it’s not interesting in terms of code, let’s see how ***efficient*** we expect it to be…

Analyzing KDTree efficiency

- *On average,* what will the height of a KD-Tree with n nodes be?

$\log_2 n$

$(\log_2 n)^2$

$\log_2 n < h < (\log_2 n)^2$

Something Else

Analyzing KDTree efficiency

- *On average,* what will the height of a KD-Tree with n nodes be?

Uniform distribution of keys implied!

$\log_2 n$	$(\log_2 n)^2$	$\log_2 n < h < (\log_2 n)^2$	Something Else

- The average – case analysis is exactly the same as that of a classic binary tree!
- So, an adversary can still make a KD-Tree pretty unbalanced ☹

Range

- KD-Trees (and other spatial data structures) allow us to perform *range* queries.
- **Intuition:** Create a k-dimensional hypersphere around a given point (the "anchor" point) and report all the points in that hypersphere (perhaps in sorted order) except that given point.
- **Formalization:** Let $\vec{p}$ be a k-dimensional vector, $r \in \mathbb{R}^{>0}$ and $d(\cdot,\cdot)$ be some distance metric. Then, a range query $Q(\vec{p}, r)$ on our database $D \subseteq \mathbb{R}^k$ is defined as the set

$$\{\vec{x} \in D \mid 0 < d(x, \vec{p}) \leq r\}$$

Range

- KD-Trees (and other spatial data structures) allow us to perform *range* queries.
- **Intuition:** Create a k-dimensional hypersphere around a given point (the "anchor" point) and report all the points in that hypersphere (perhaps in sorted order) except that given point.
- **Formalization:** Let $\vec{p}$ be a k-dimensional vector, $r \in \mathbb{R}^{>0}$ and $d(\cdot,\cdot)$ be some distance metric. Then, a range query $Q(\vec{p}, r)$ on our database $D \subseteq \mathbb{R}^k$ is defined as the set

$$\{\vec{x} \in D \mid 0 < d(x, \vec{p}) \leq r\}$$

Convention #1: Our ranges will be ***closed*** (in the project too!)

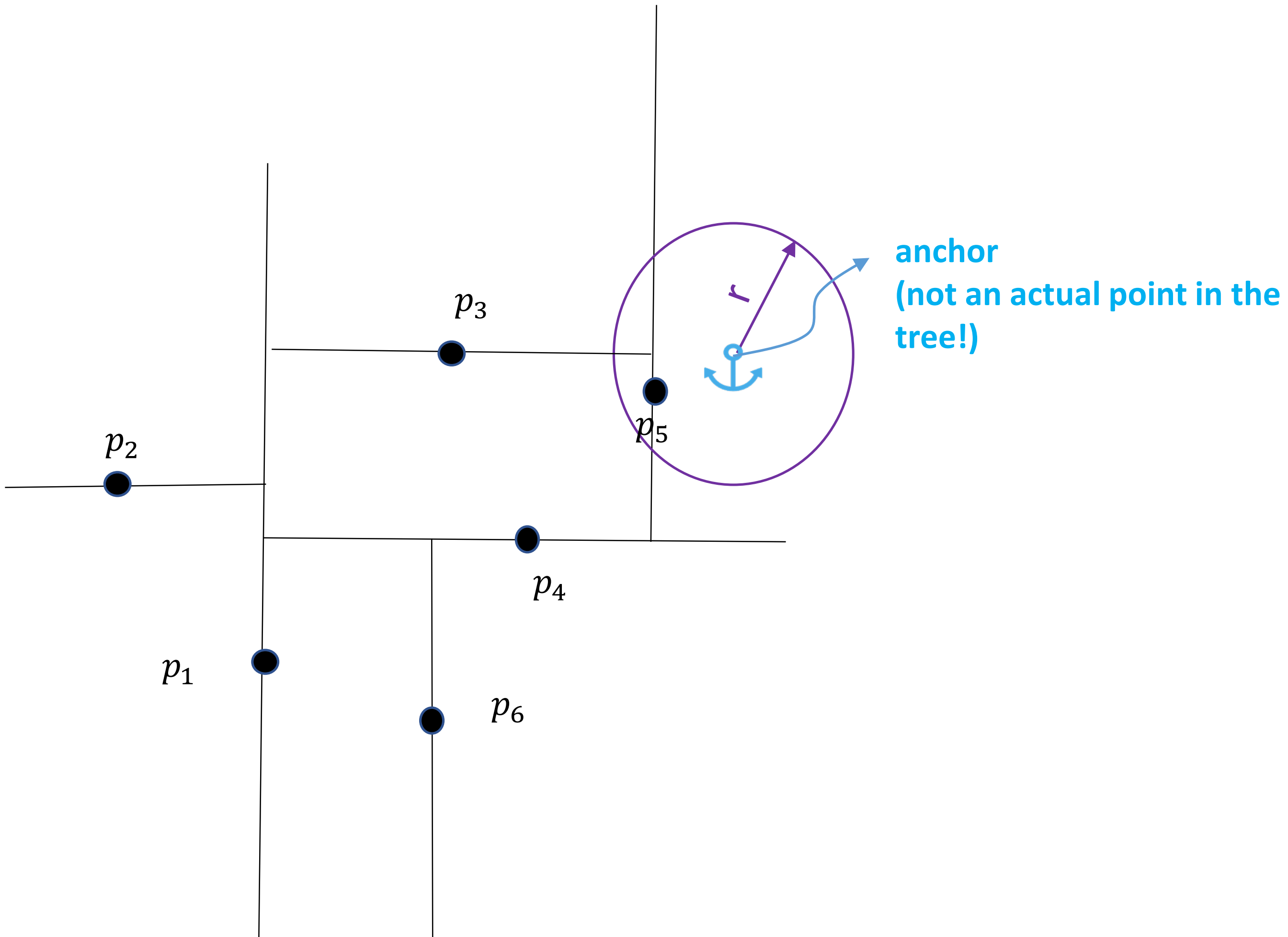
Convention #2: We do not report the "anchor" point itself (also in the project).

Range Query examples

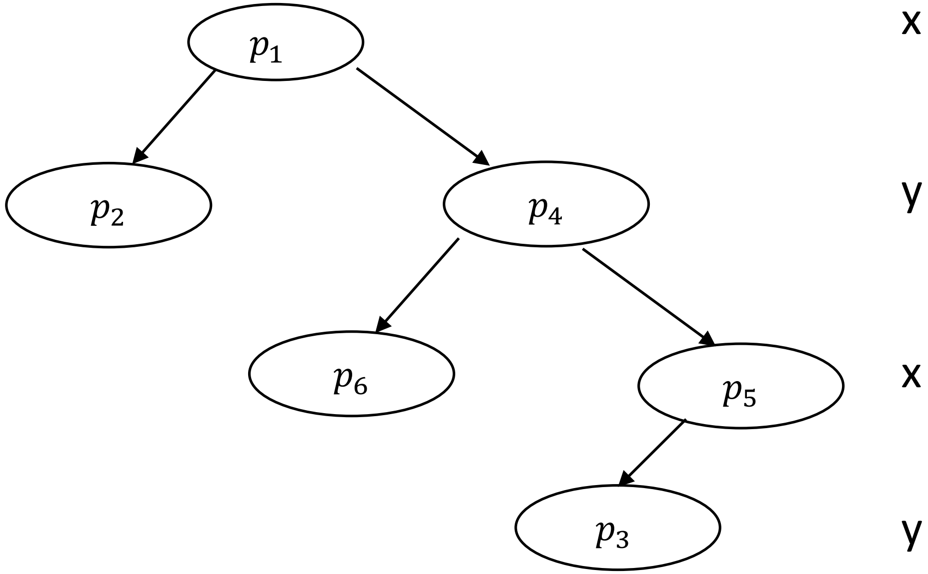
- Let's consider some range queries.
- These are things to remember as we go:
  1. Ranges are inclusive.
  2. The "anchor" point *(center of range)* is NOT REQUIRED to be in the KD-Tree proper!
     - That is, it's not required to be a $\vec{x} \in D$!
  3. The anchor point should not be reported *(so if it is actually part of the tree and we visit it as we descend…)*

Range Query examples

2D space

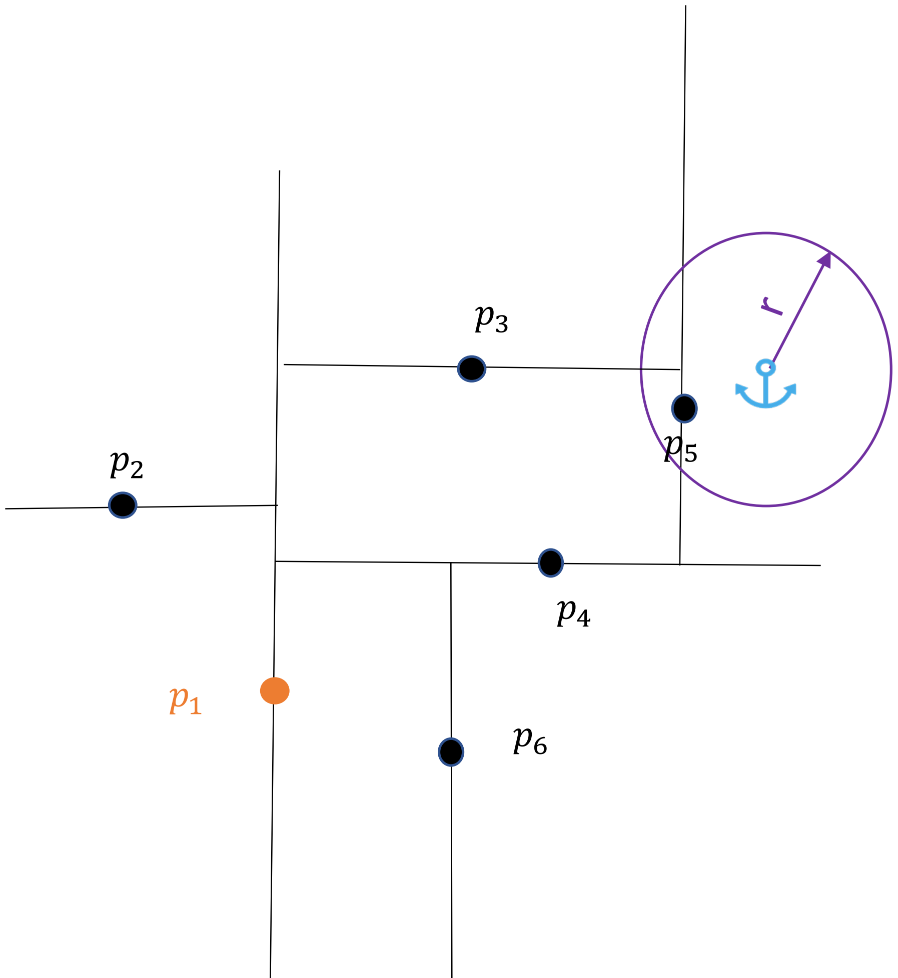


Corresponding KD-tree

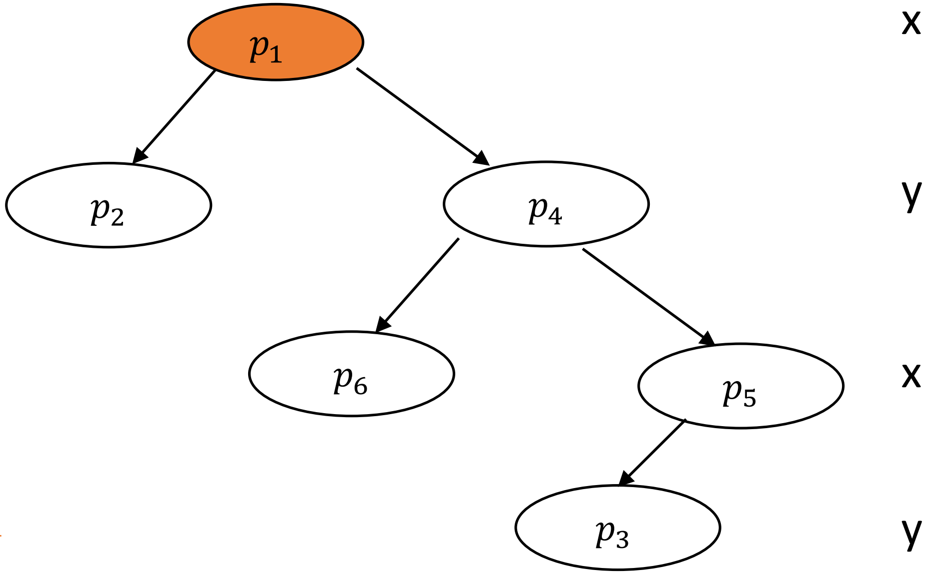


Range Query examples

2D space



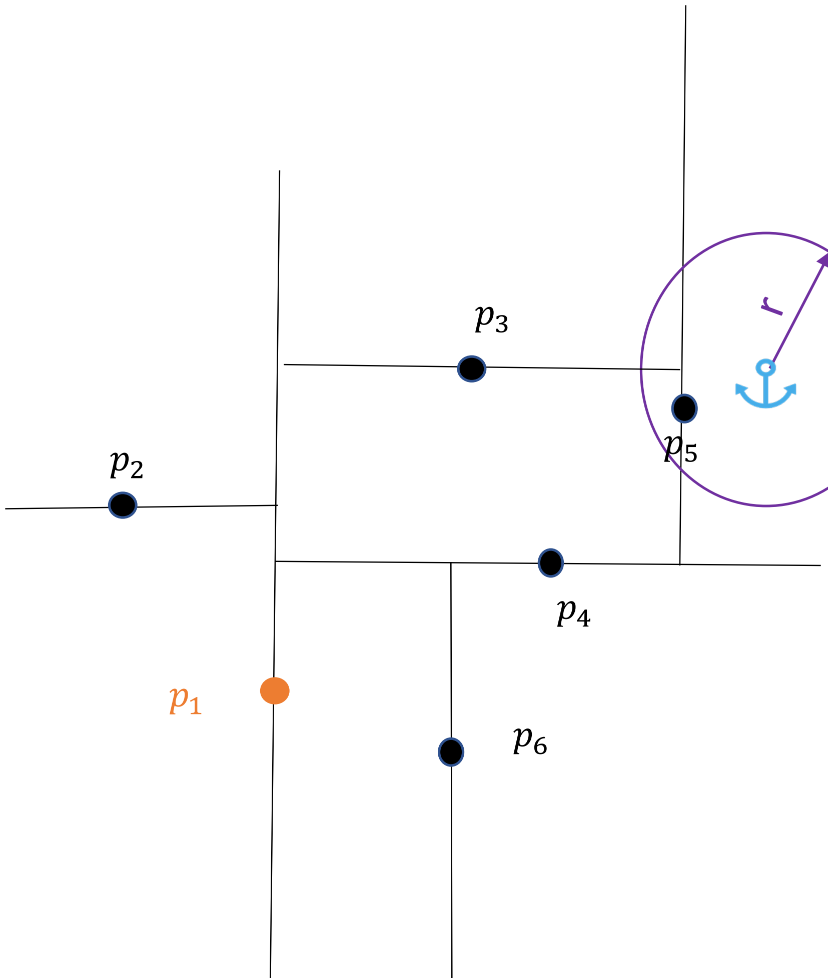
Corresponding KD-tree



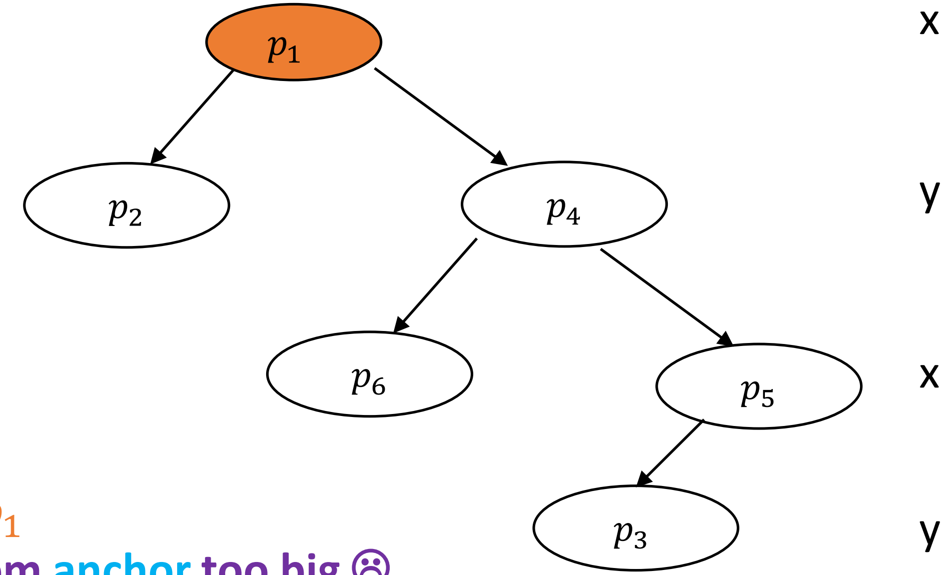
1. **Visit:** The root, p_1

Range Query examples

2D space



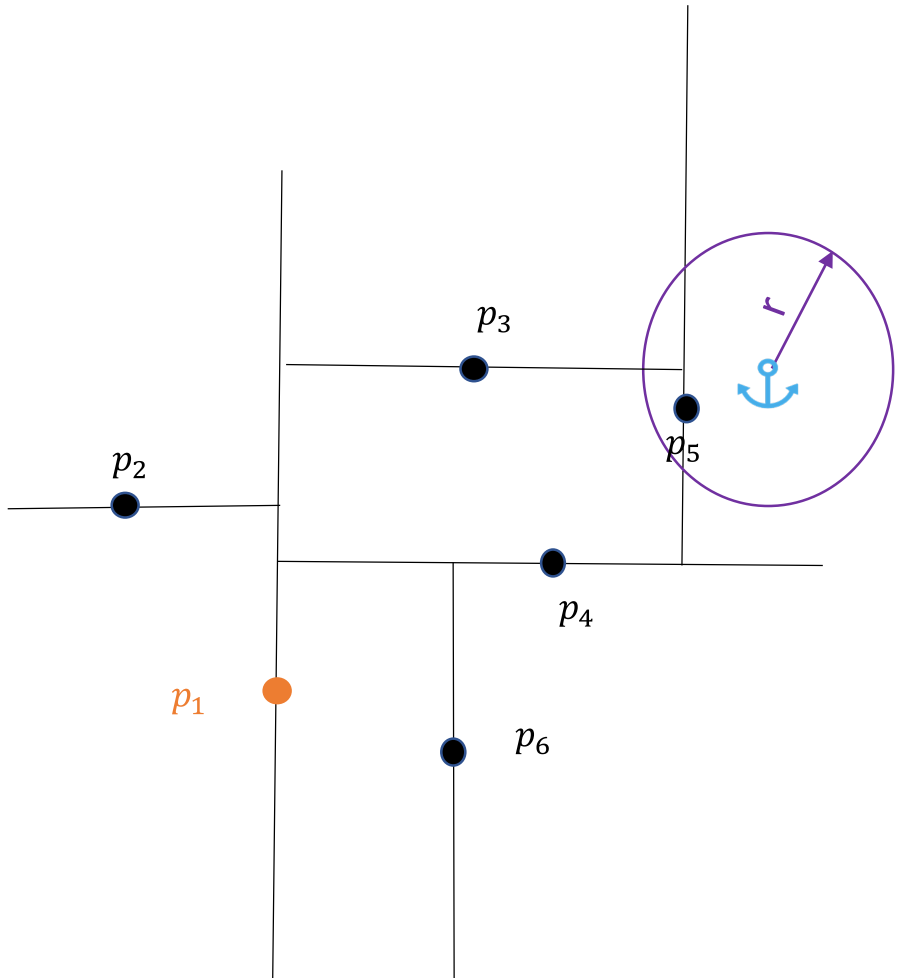
Corresponding KD-tree



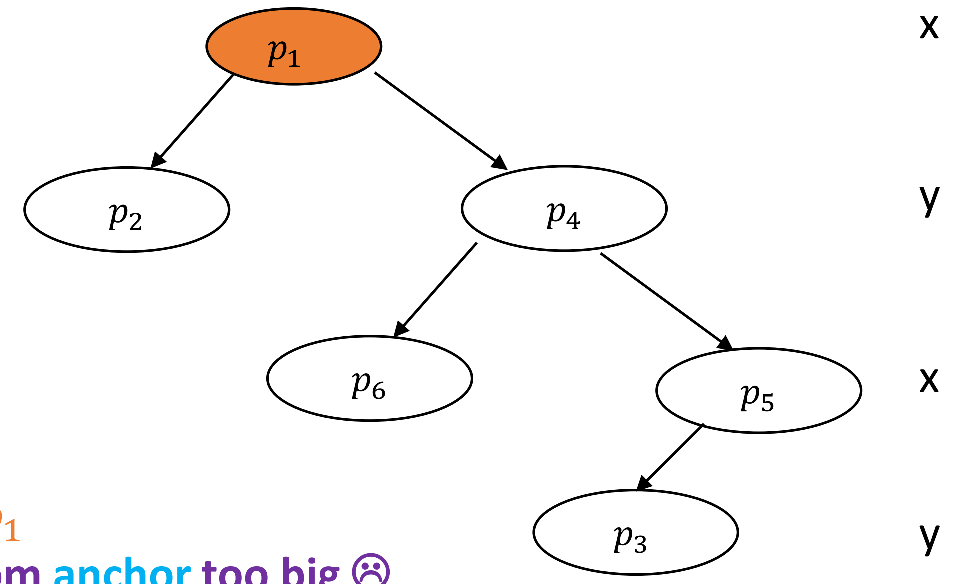
1. **Visit:** The root, p_1
2. **Test:** **distance from anchor too big** ☹

Range Query examples

2D space

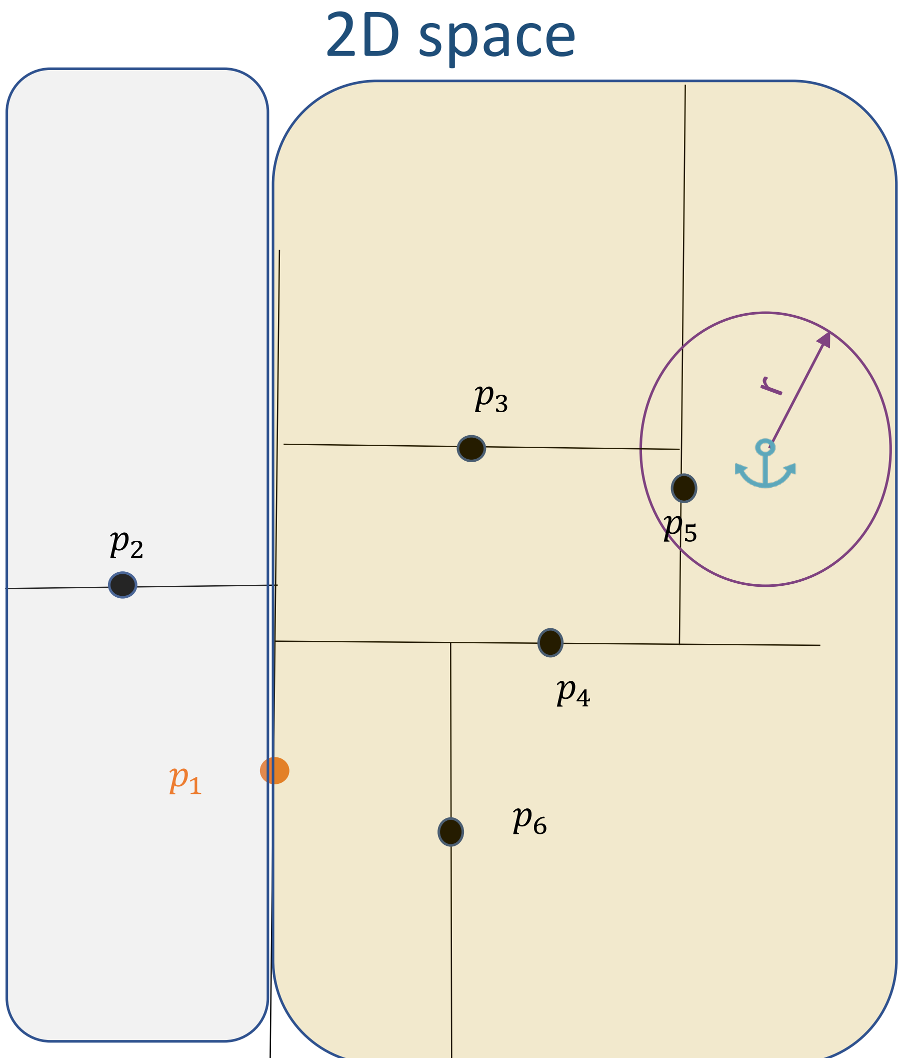


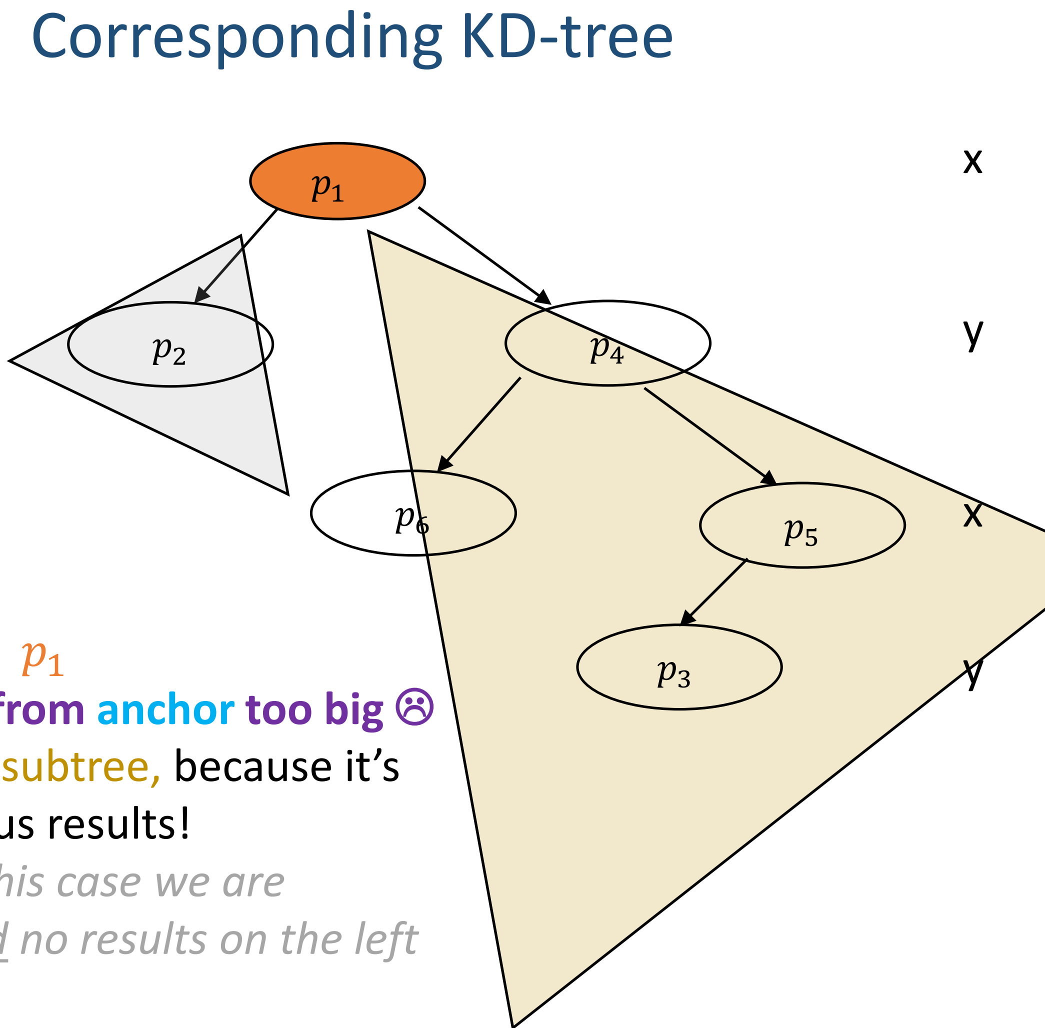
Corresponding KD-tree



1. **Visit:** The root, p_1
2. **Test:** **distance from anchor too big** ☹
3. **Recurse:** *where, and why?*

Range Query examples

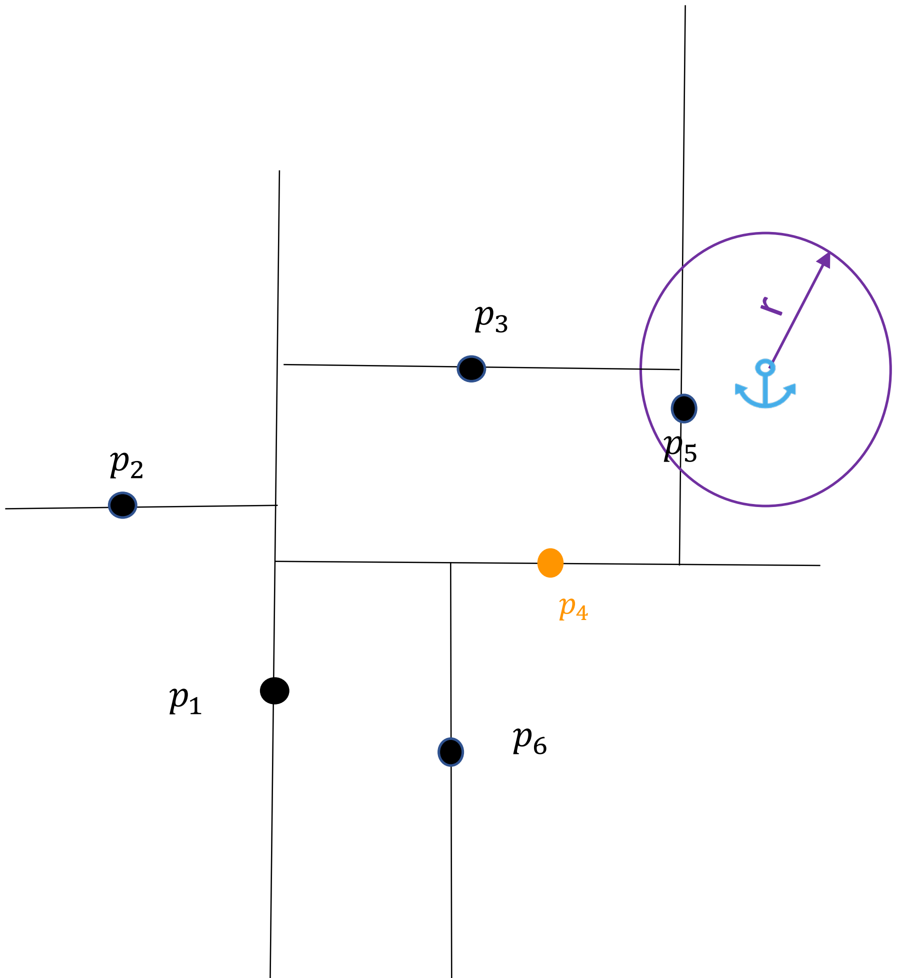




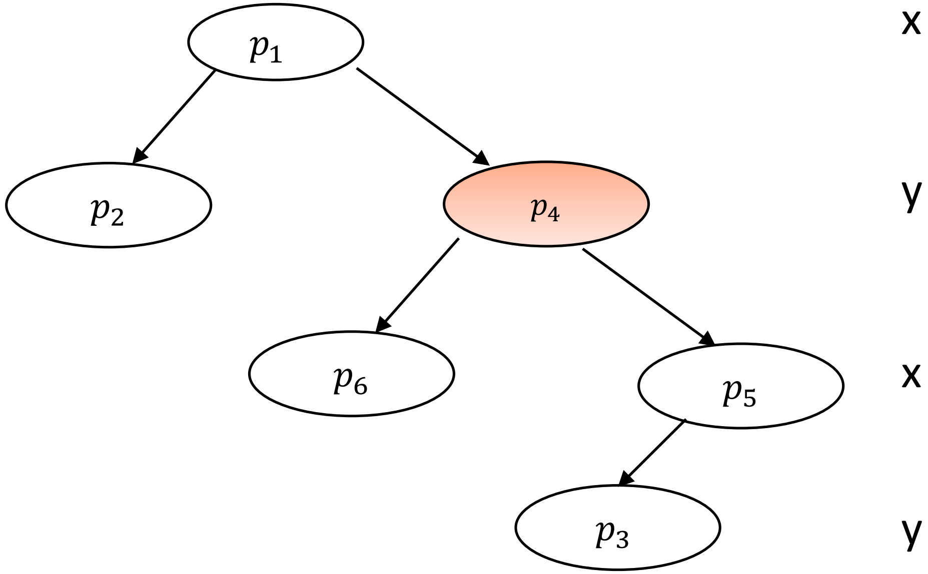
1. **Visit:** The root, p_1
2. **Test: distance from anchor too big** ☹
3. **Recurse:** Right subtree, because it's likelier to give us results!
   - *In fact, in this case we are* *guaranteed no results on the left subtree*

Range Query examples

2D space



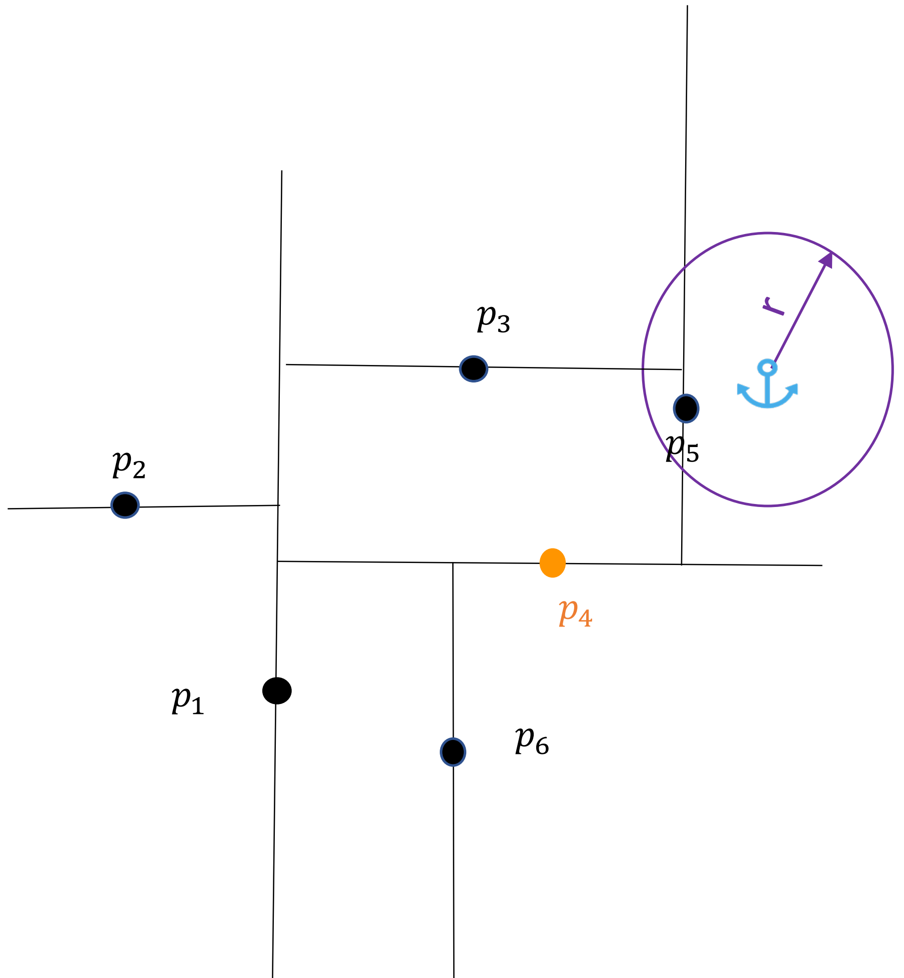
Corresponding KD-tree



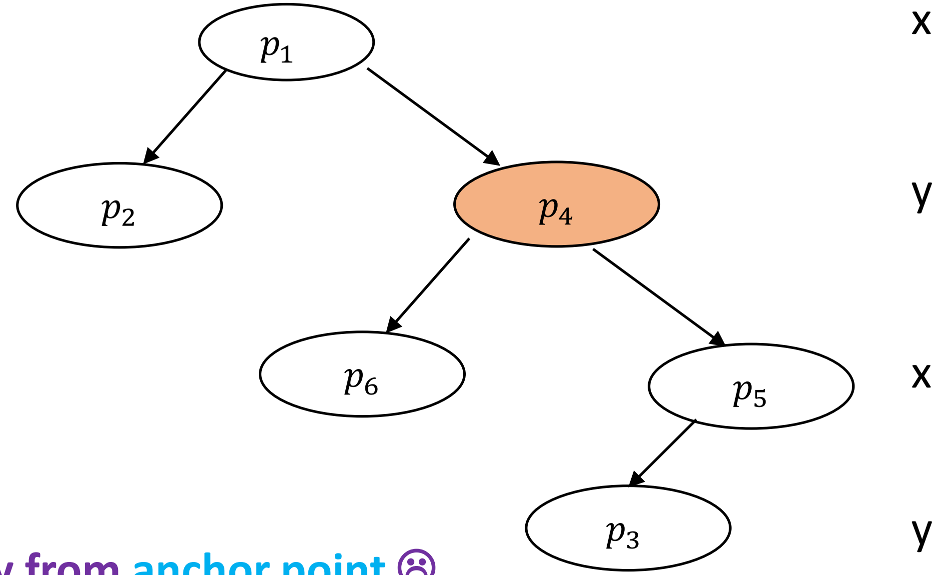
1. **Visit:** p_4

Range Query examples

2D space



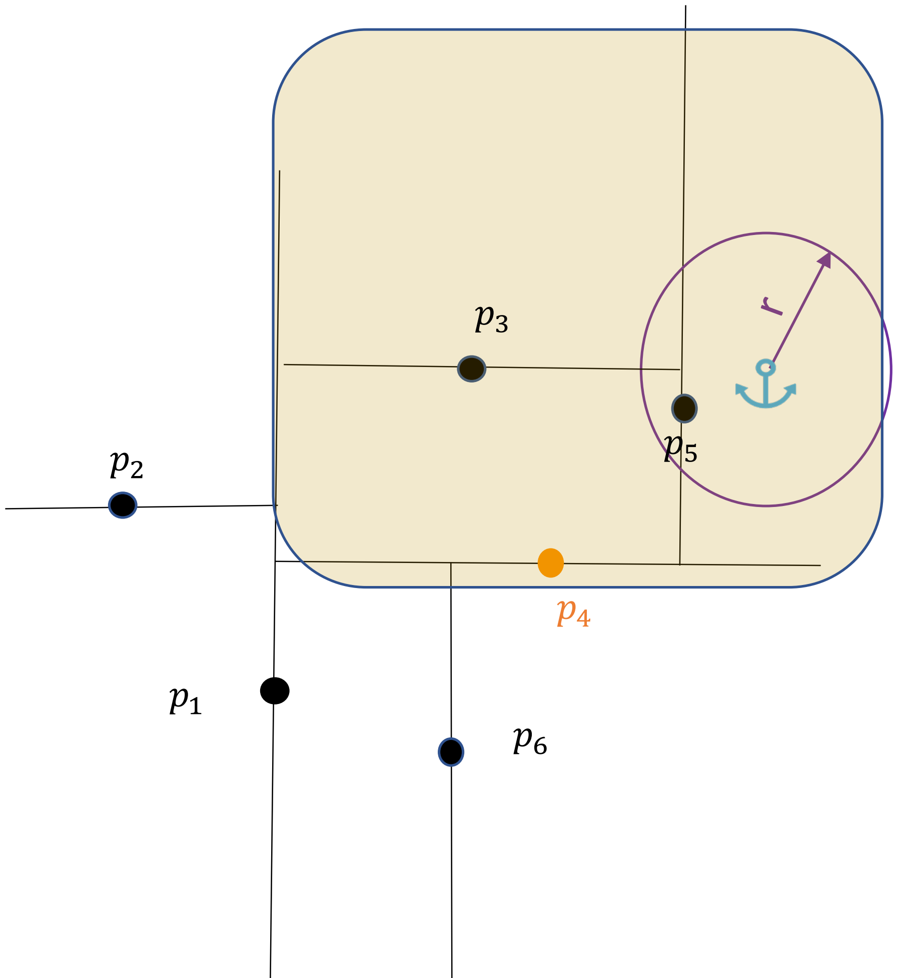
Corresponding KD-tree



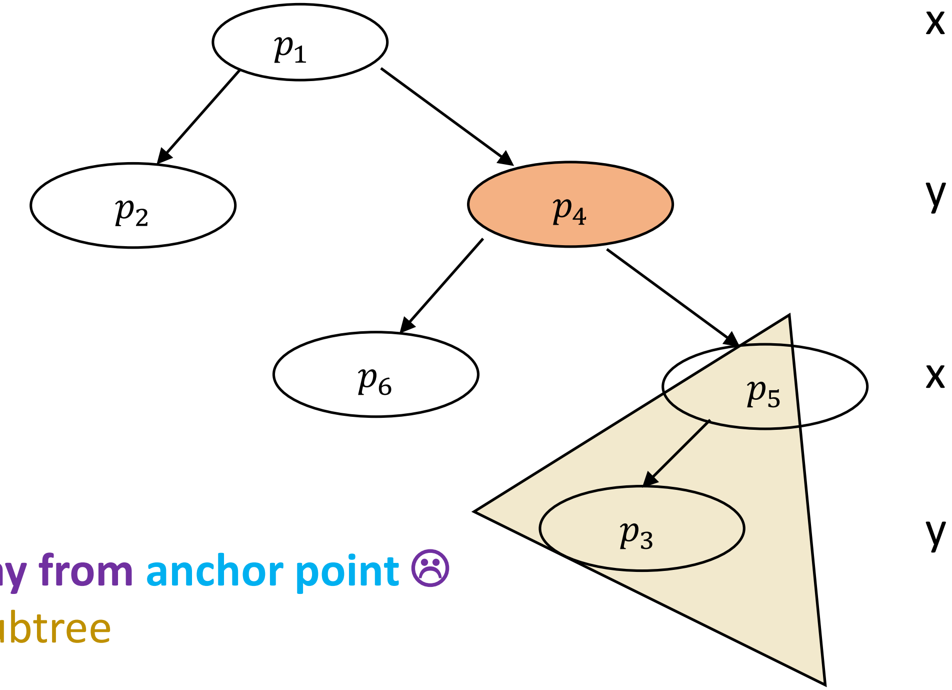
1. **Visit:** p_4
2. **Test: Too far away from anchor point ☹**

Range Query examples

2D space



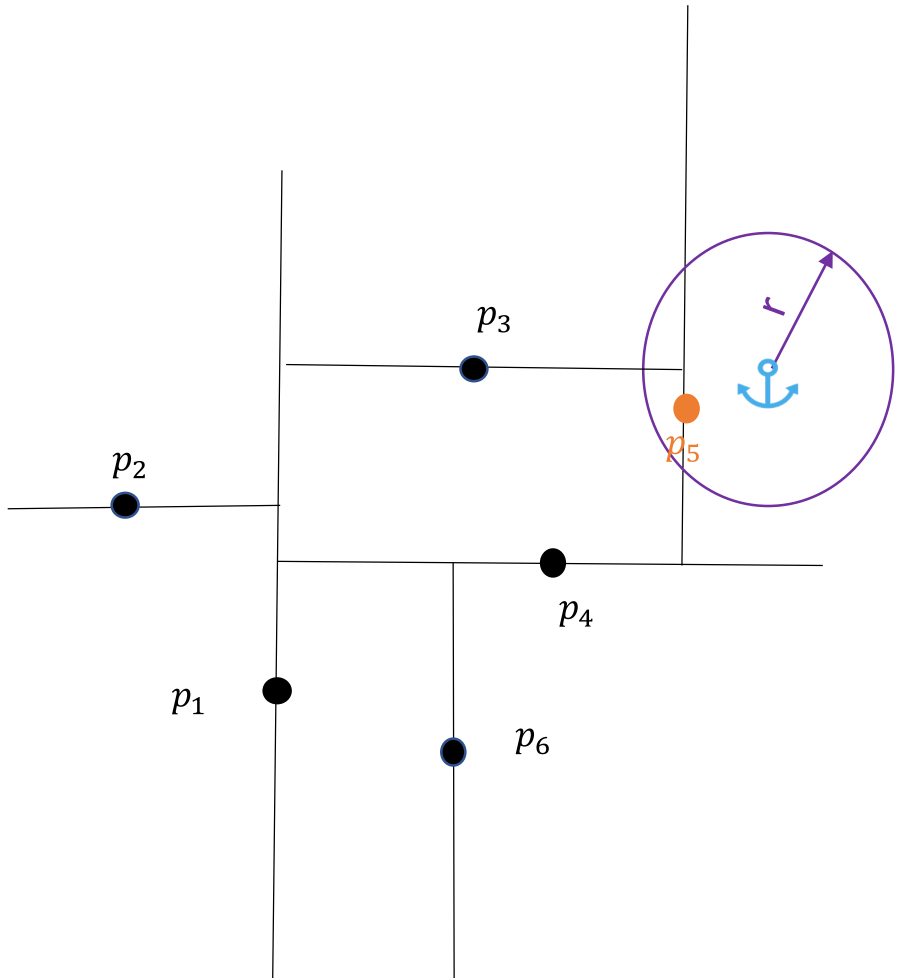
Corresponding KD-tree



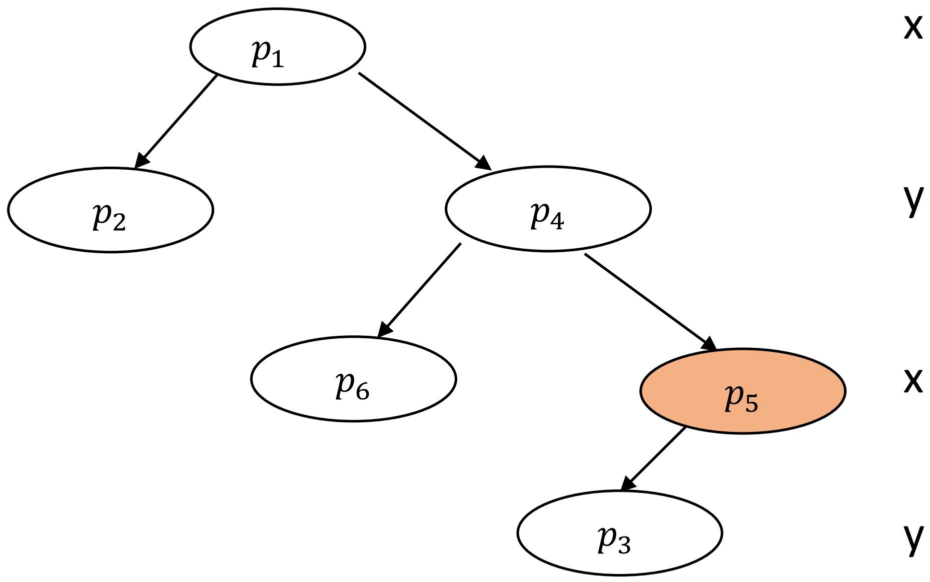
1. **Visit:** p_4
2. **Test:** **Too far away from anchor point ☹**
3. **Recurse:** Right subtree

Range Query examples

2D space



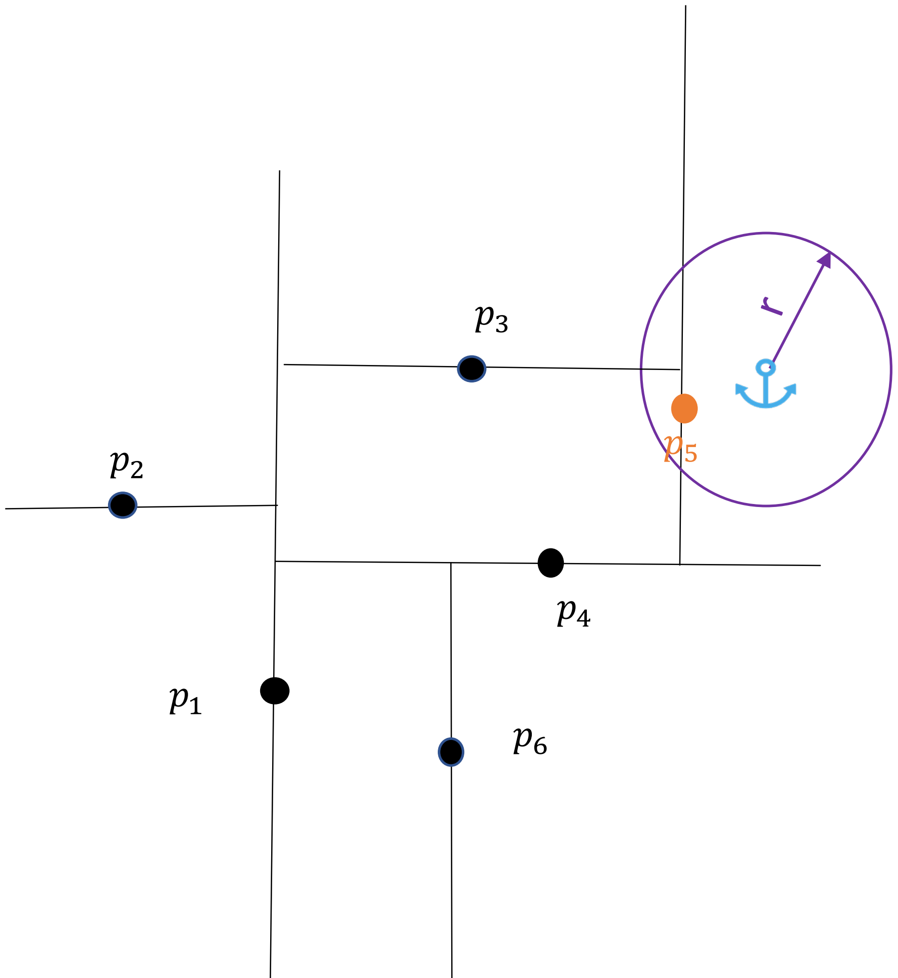
Corresponding KD-tree



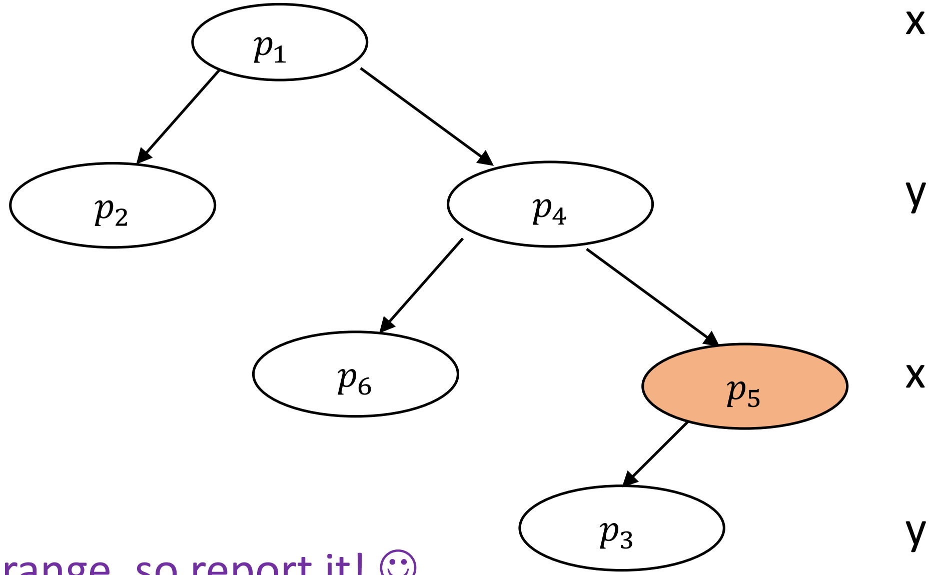
1. **Visit:** p_5

Range Query examples

2D space

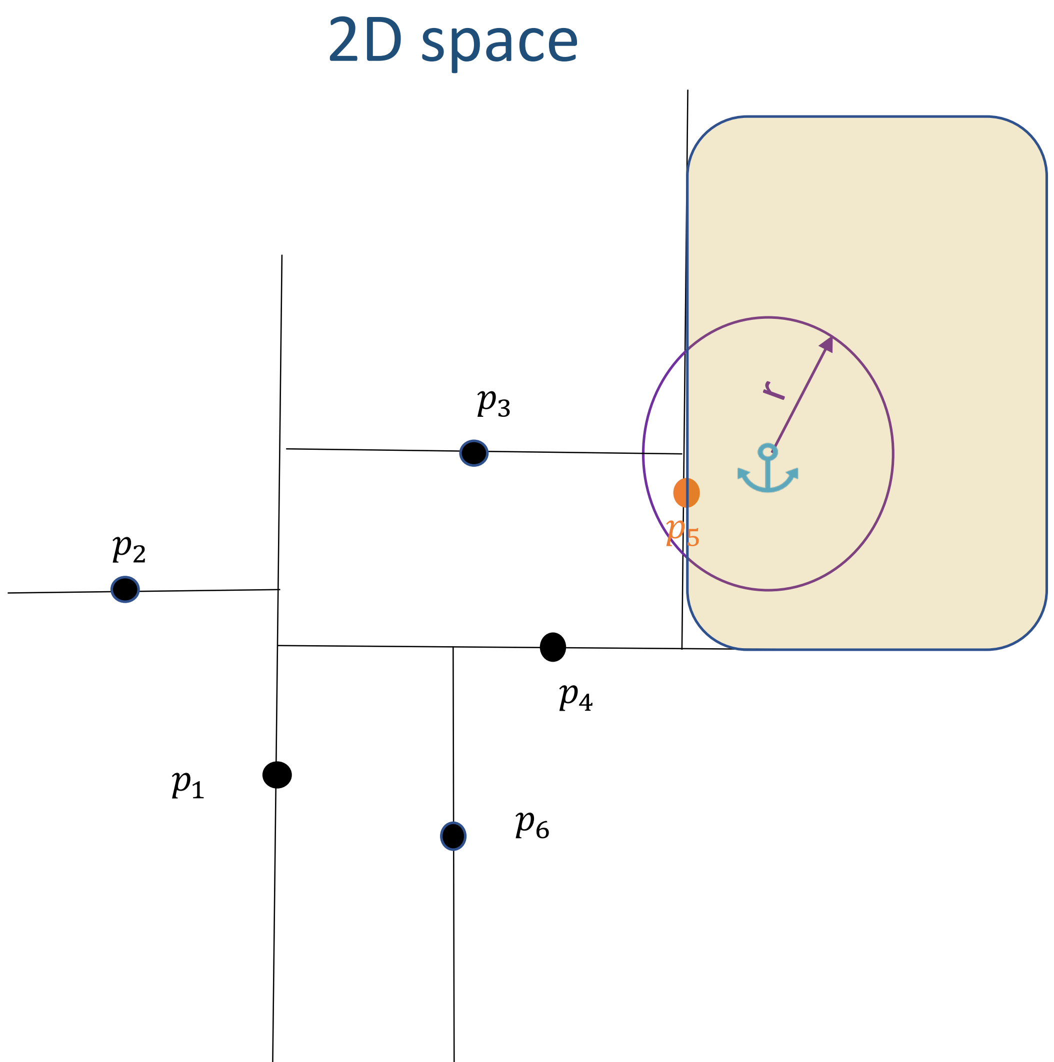


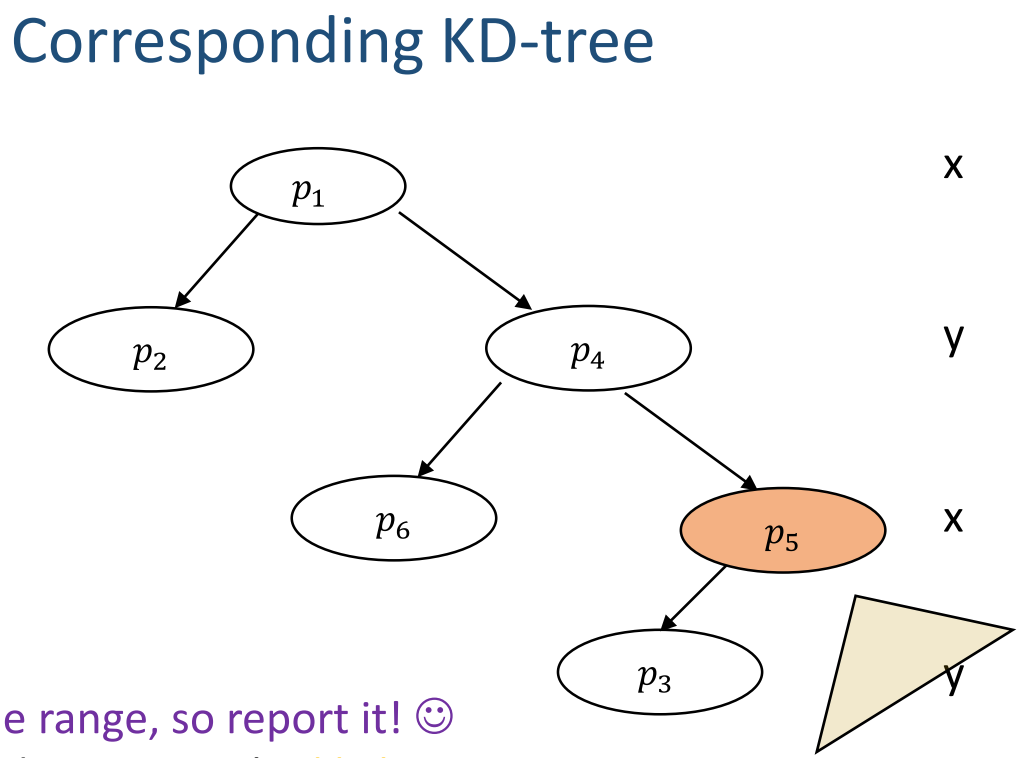
Corresponding KD-tree



1. **Visit:** p_5
2. **Test:** It's within the range, so report it! ☺

Range Query examples

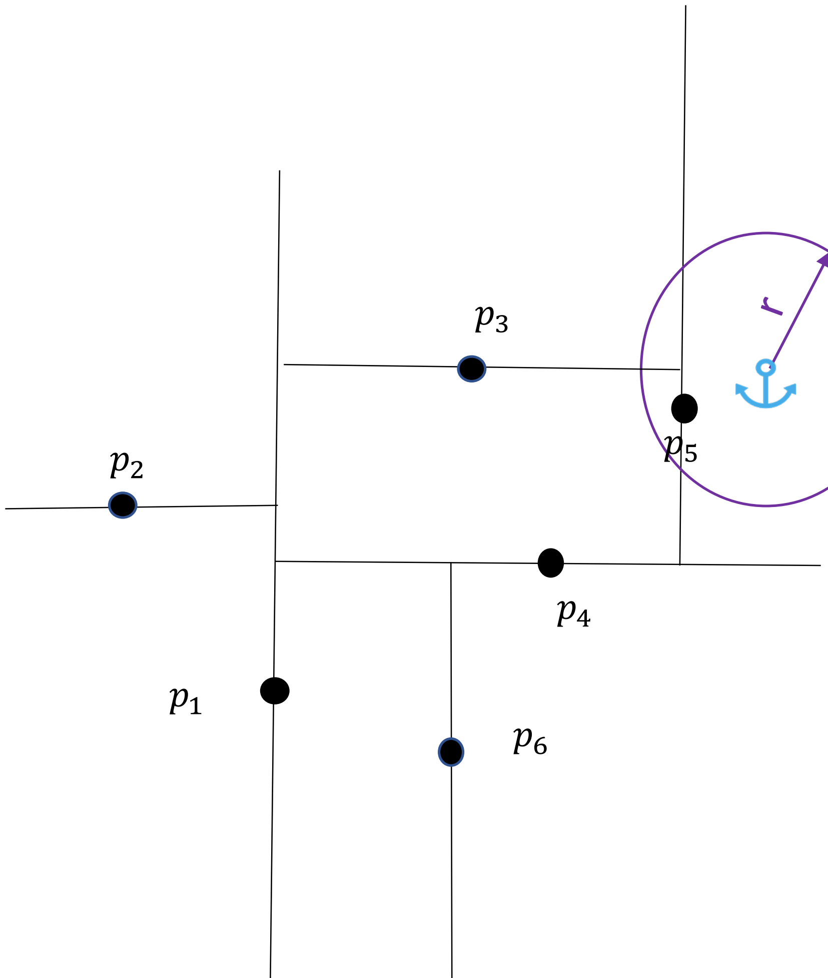




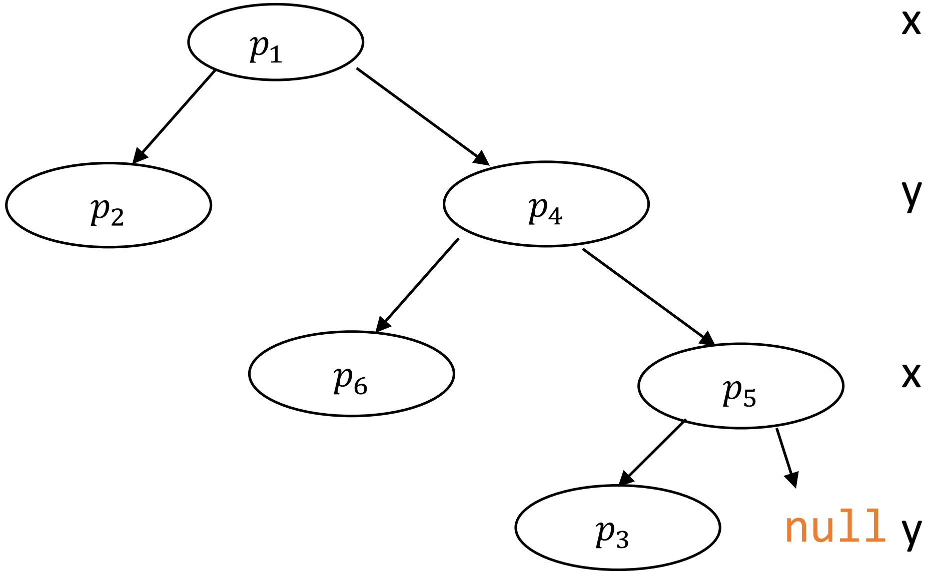
1. **Visit:** p_5
2. **Test:** It's within the range, so report it! ☺
3. **Recurse:** To the right, since we're likelier to find results that way!

Range Query examples

2D space



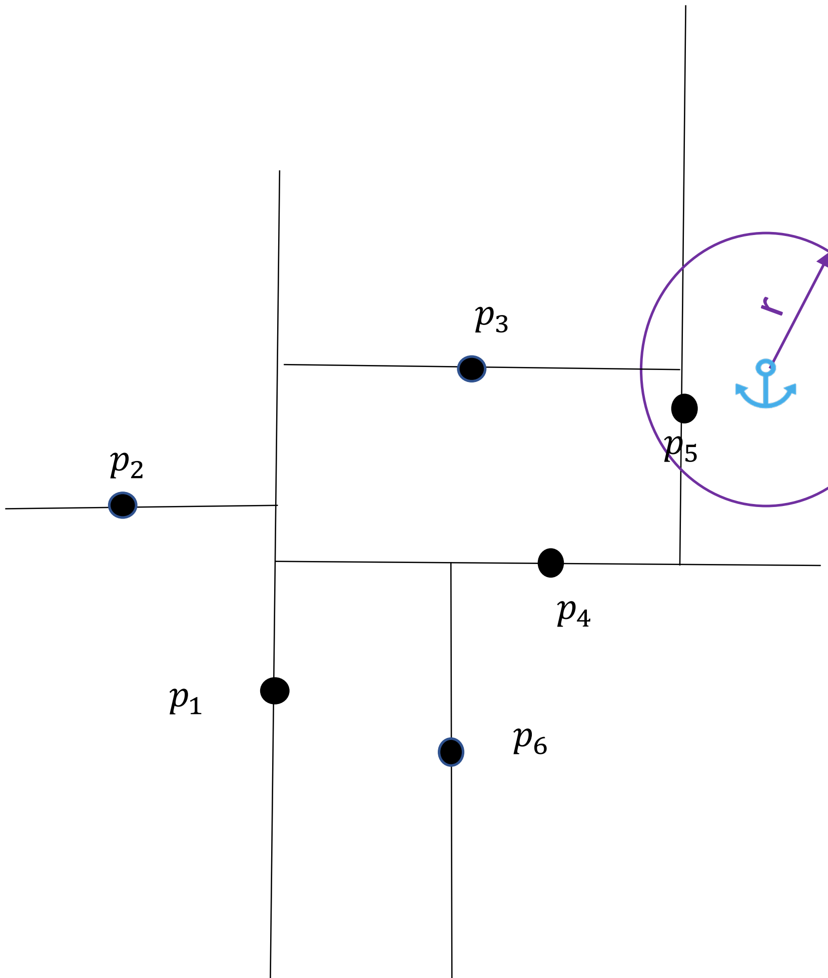
Corresponding KD-tree



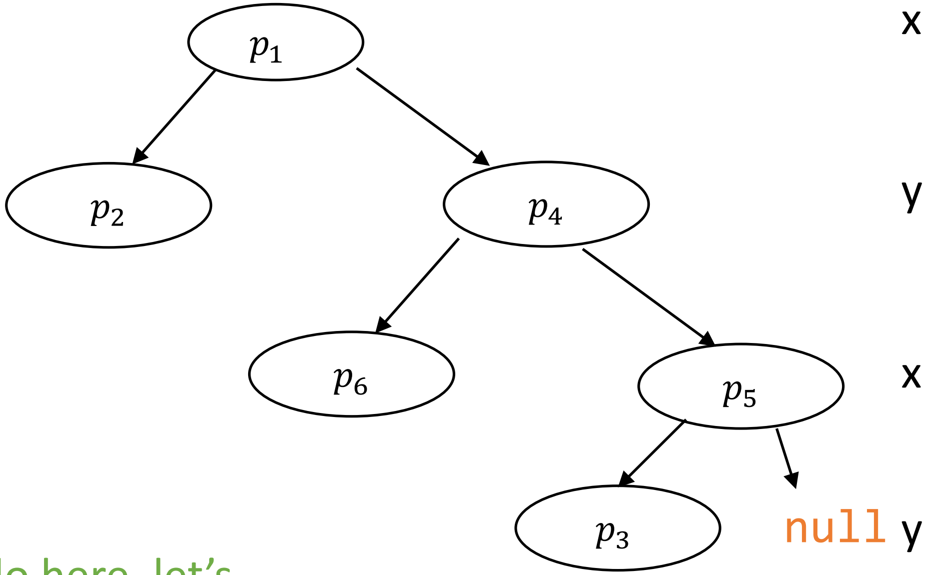
1. **Visit:** `null`

Range Query examples

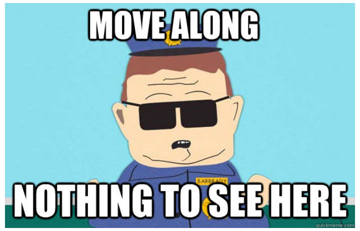
2D space



Corresponding KD-tree

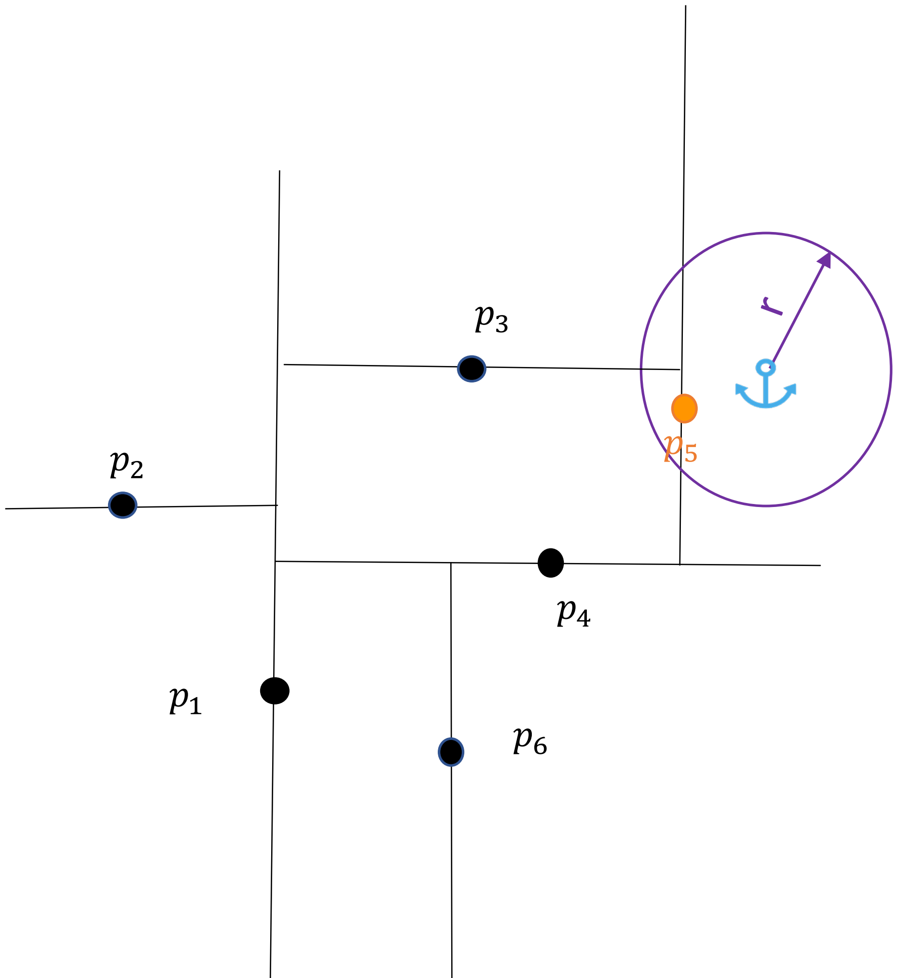


1. **Visit:** `null`
2. There's nothing to do here, let's backtrack!

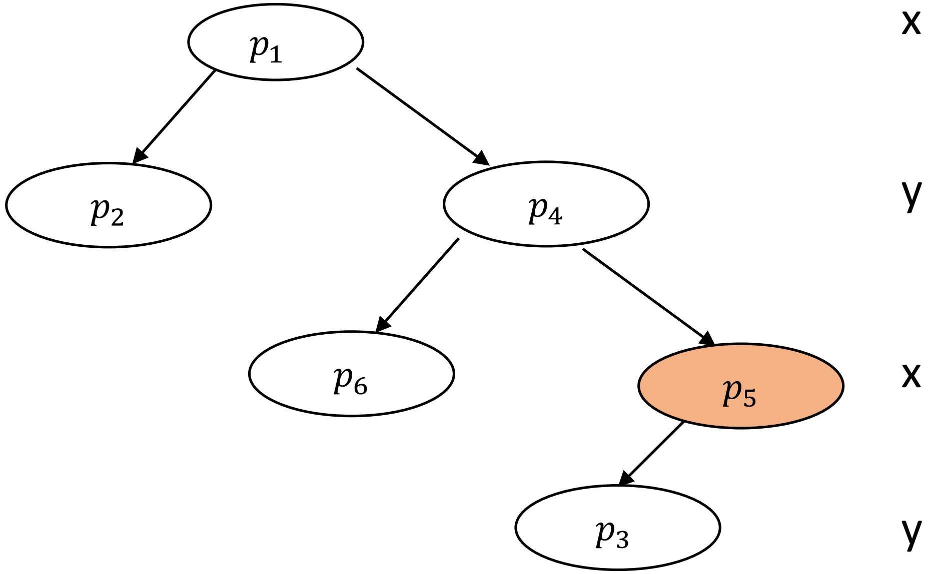


Range Query examples

2D space



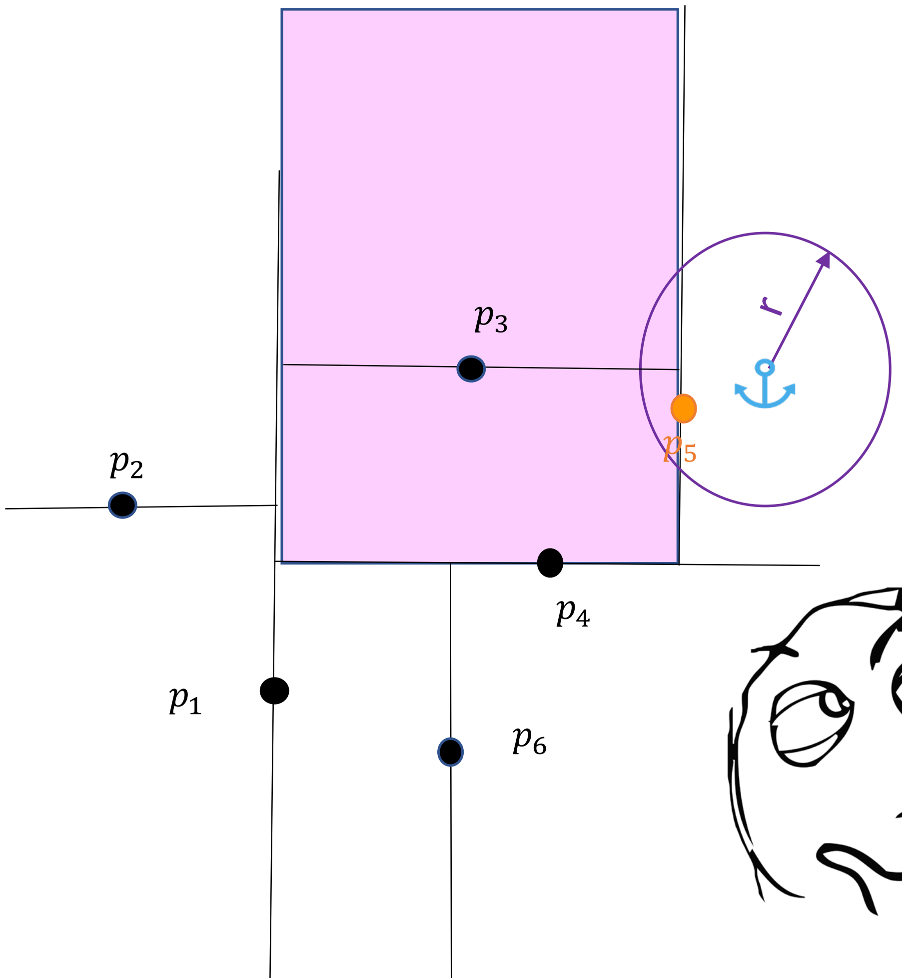
Corresponding KD-tree



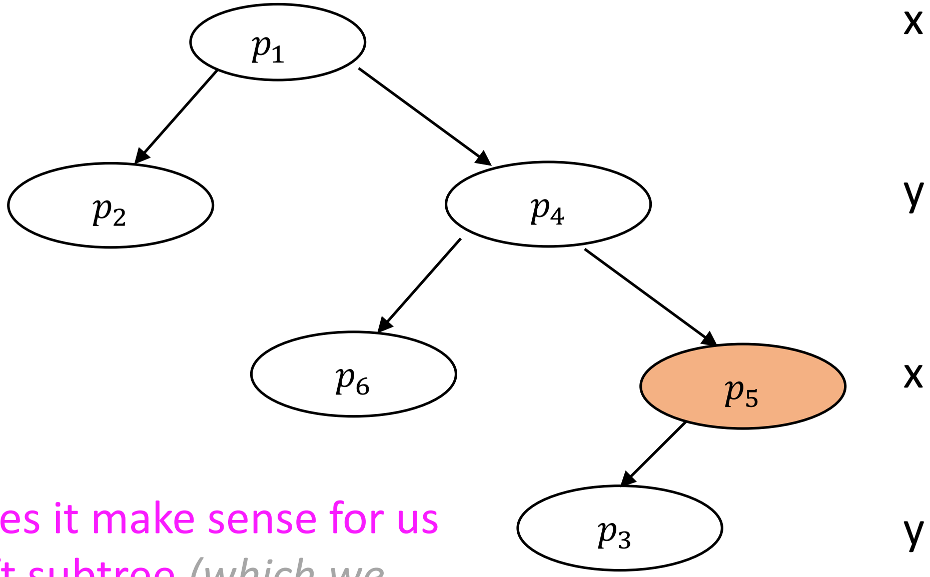
1. **Backtrack to: p_5**

Range Query examples

2D space



Corresponding KD-tree



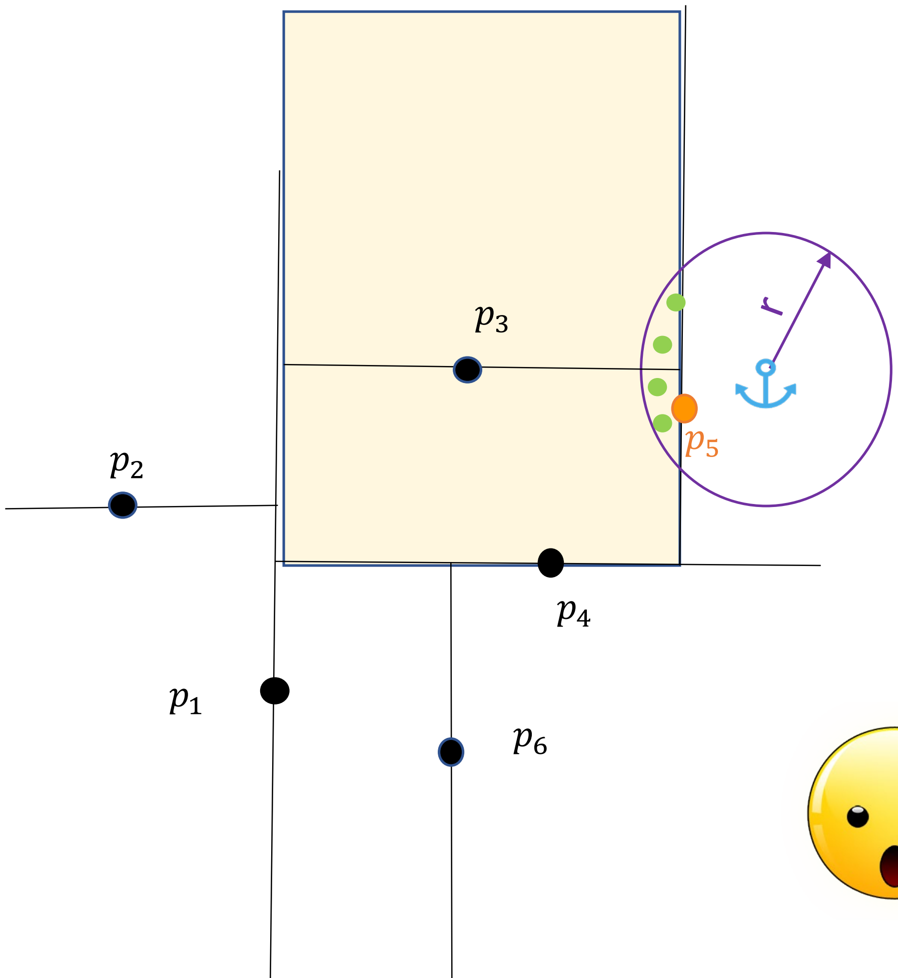
1. **Backtrack to:** p_5 Does it make sense for us to recurse to the left subtree *(which we disregarded earlier)* ?

Yes

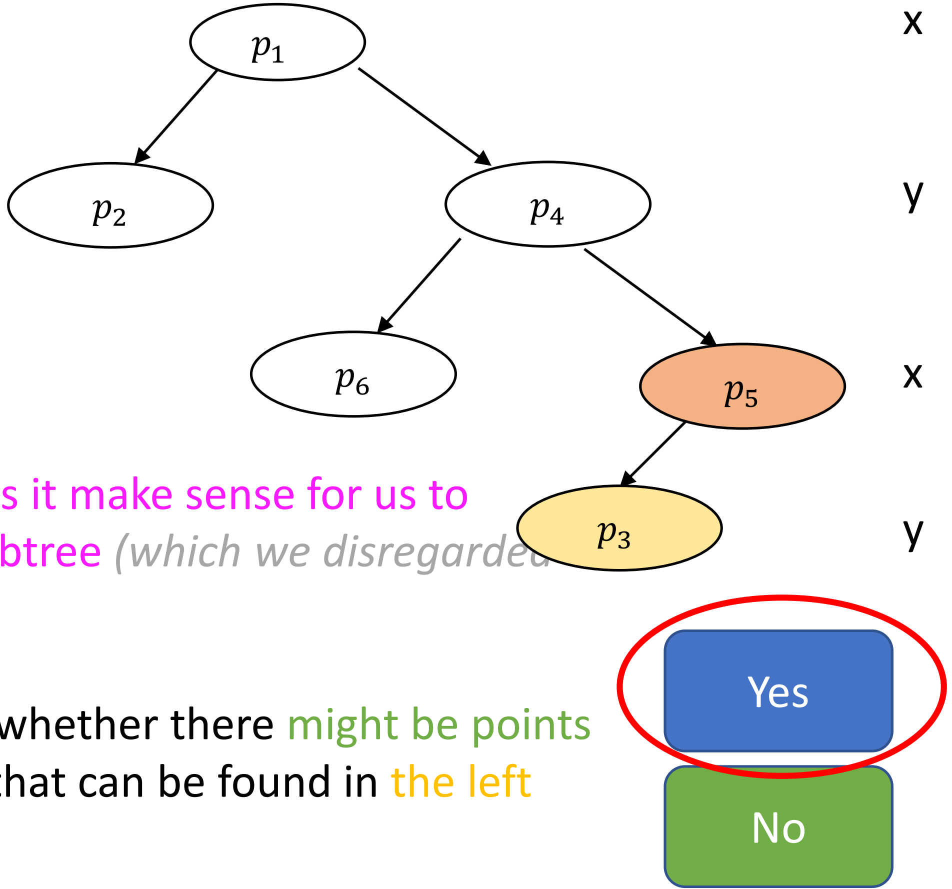
No

Range Query examples

2D space



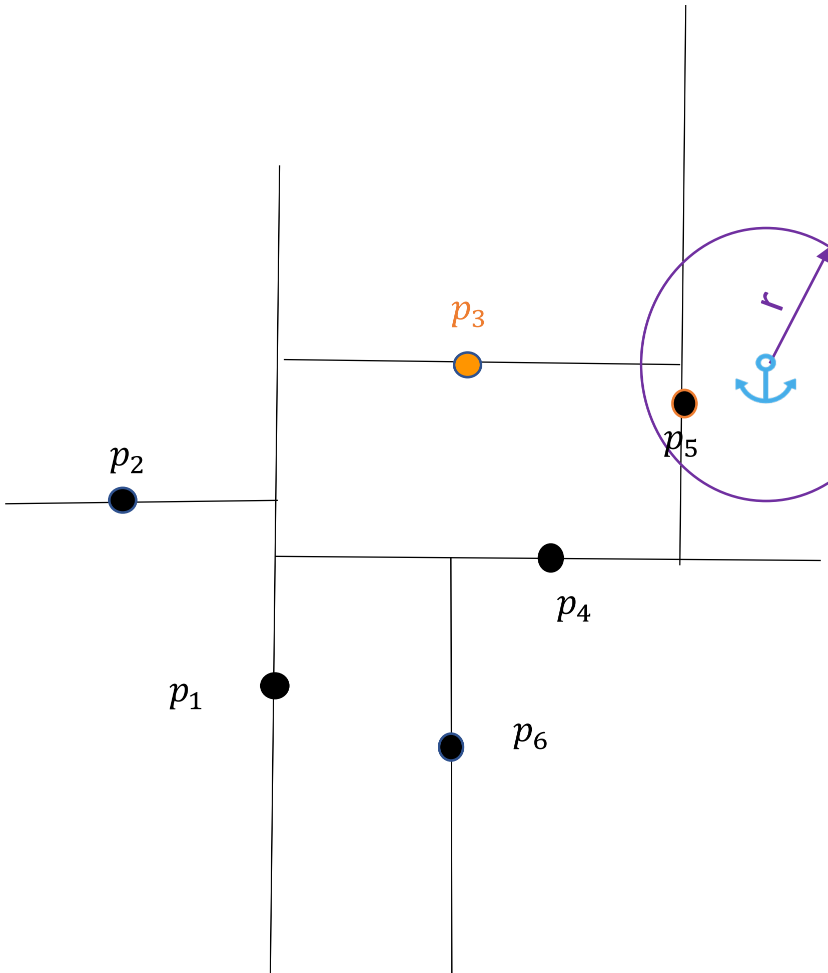
Corresponding KD-tree



1. **Backtrack to:** p_5 Does it make sense for us to recurse to the left subtree *(which we disregarded earlier)* ?
   - **Yes, of course!**
   - We can't be sure whether there might be points within the range that can be found in the left subtree!

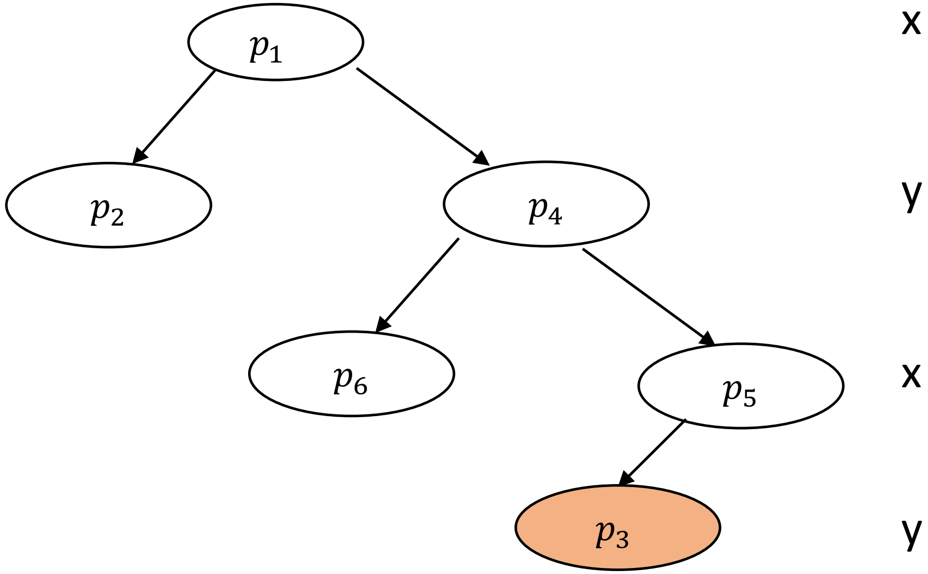
Range Query examples

2D space



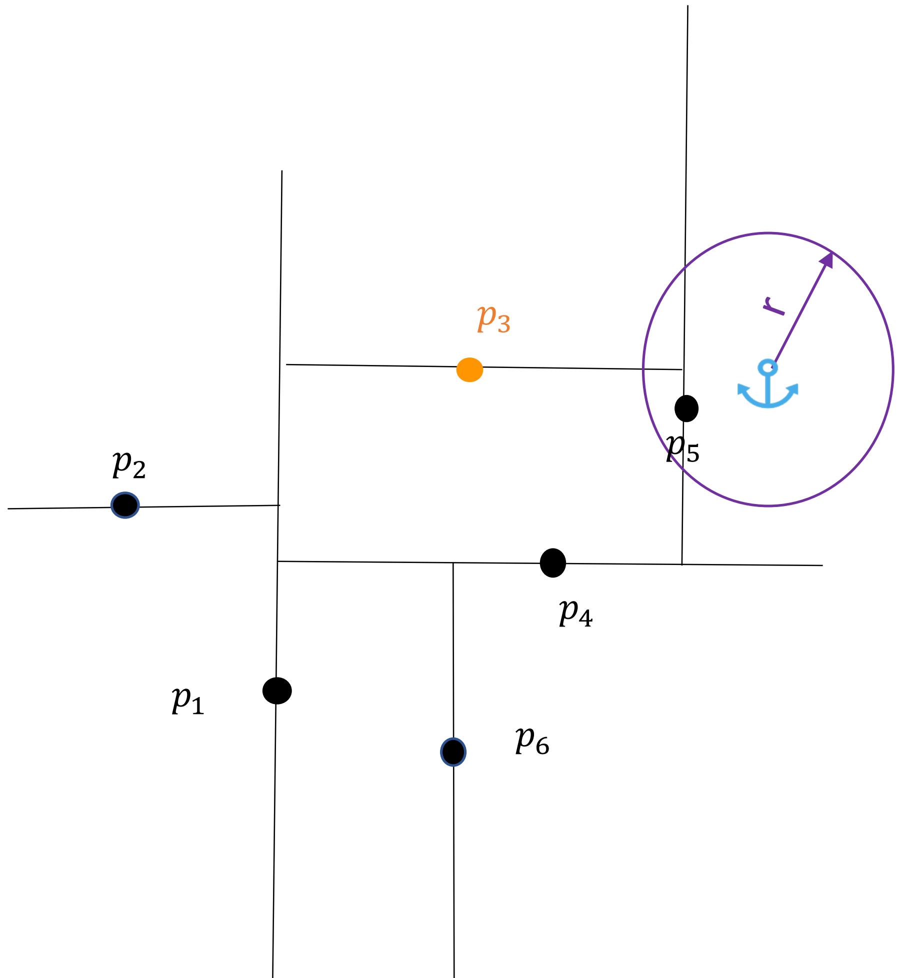
1. **Visit:** p_3

Corresponding KD-tree

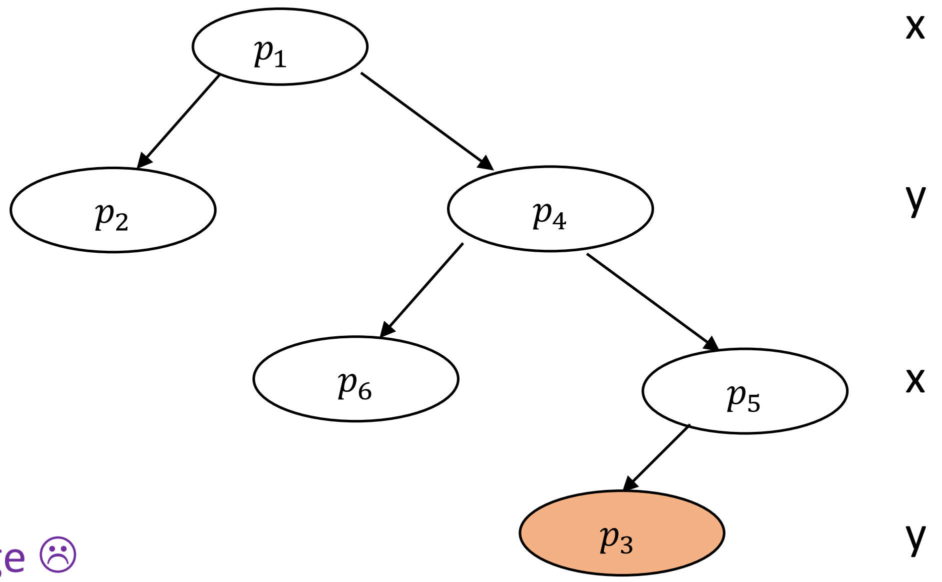


Range Query examples

2D space



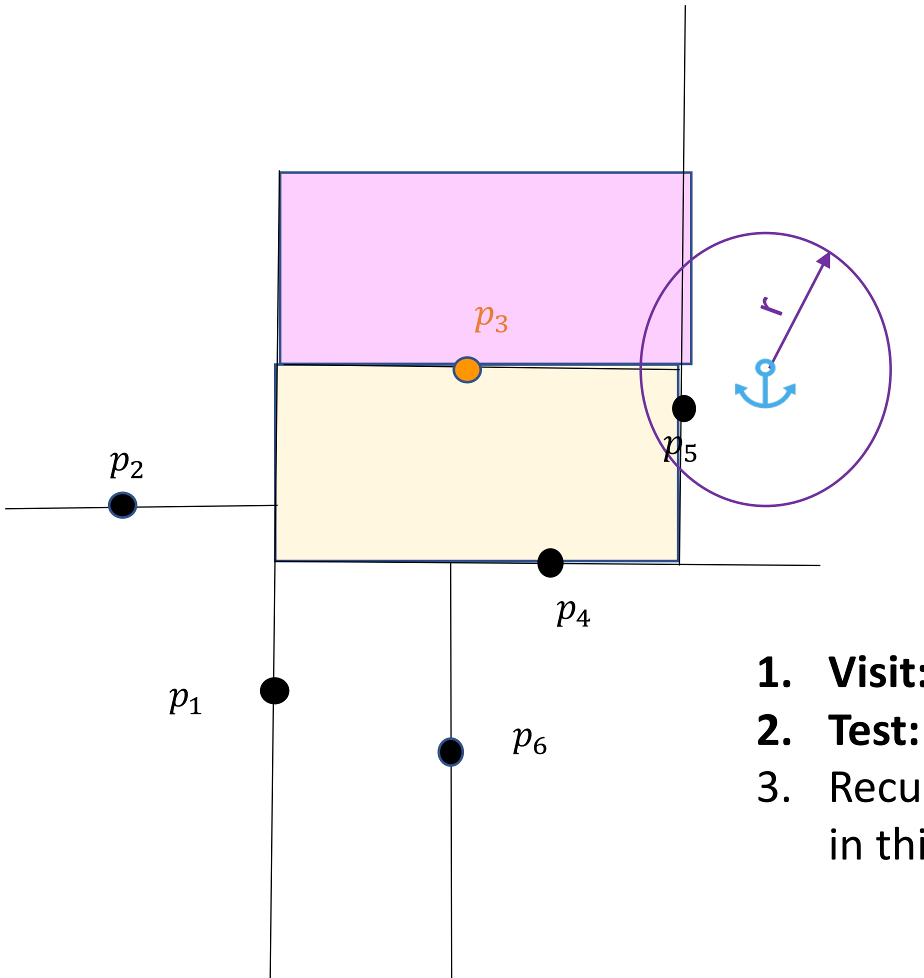
Corresponding KD-tree



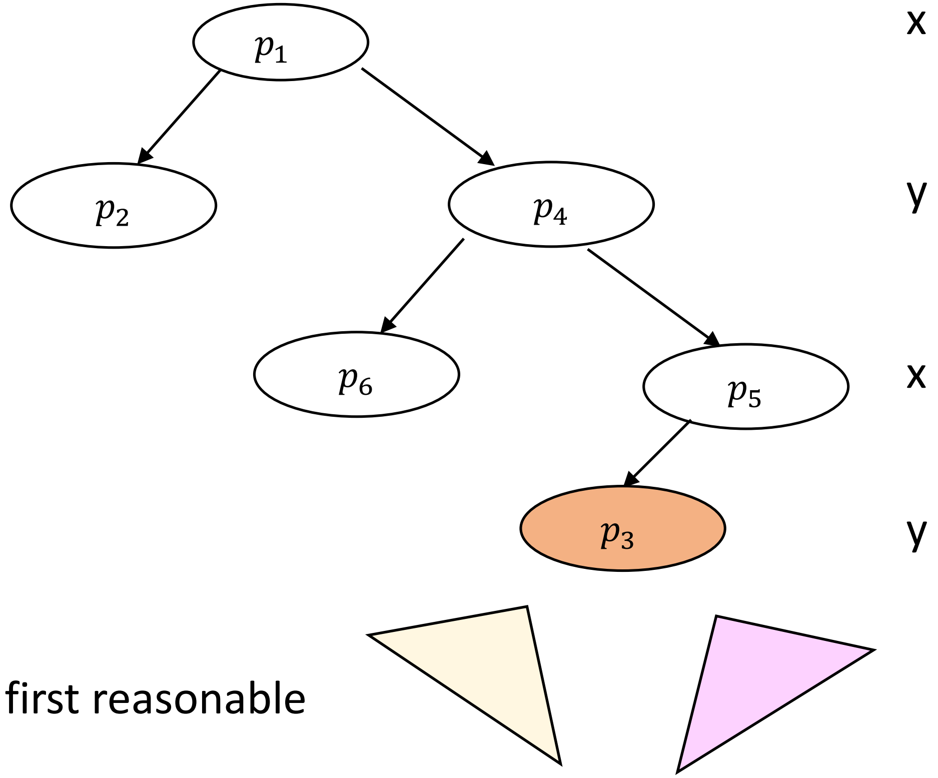
1. **Visit:** p_3
2. **Test:** Not within range ☹

Range Query examples

2D space



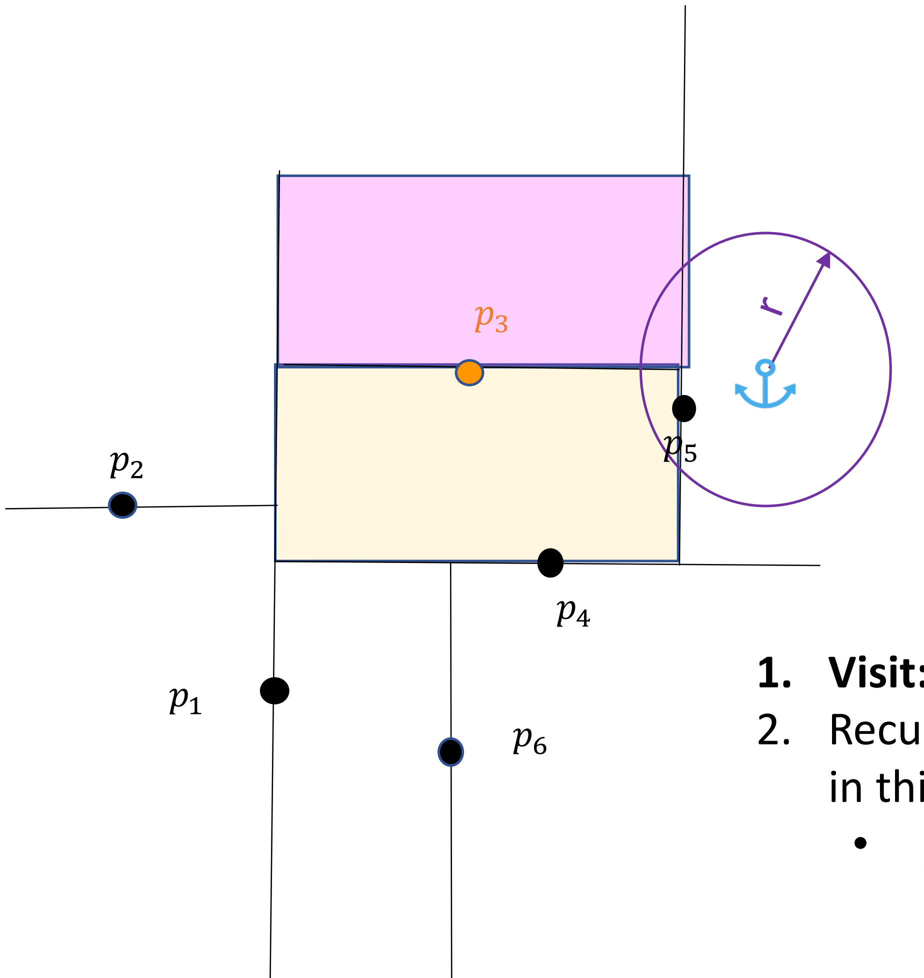
Corresponding KD-tree



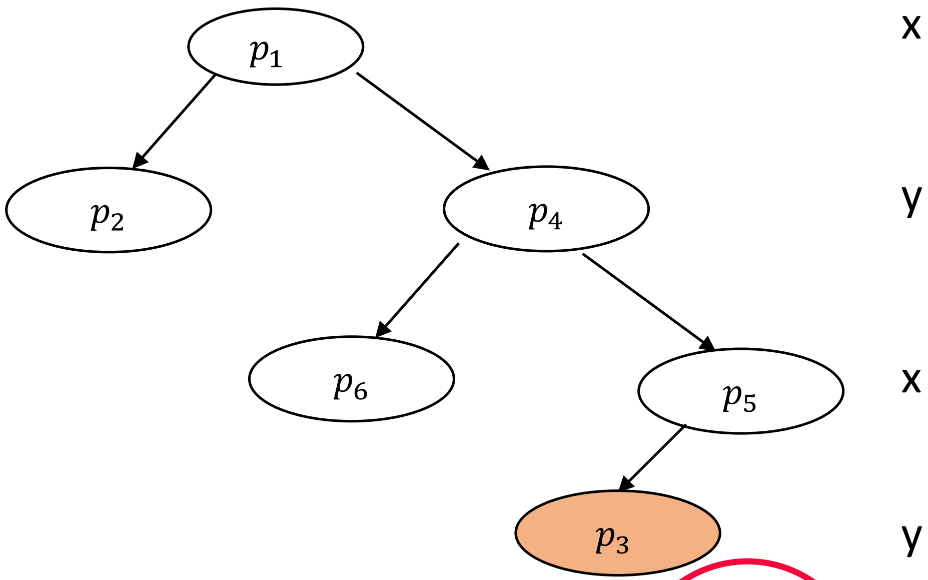
1. **Visit:** p_3
2. **Test:** Not within range ☹
3. Recursing on **either** left or right child first reasonable in this special case! :O

Range Query examples

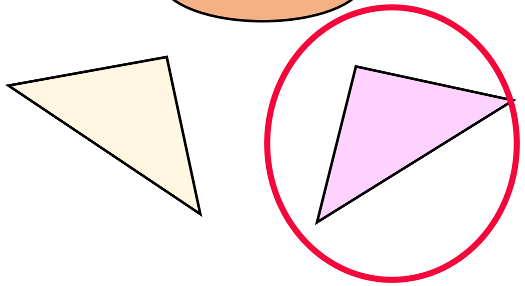
2D space



Corresponding KD-tree

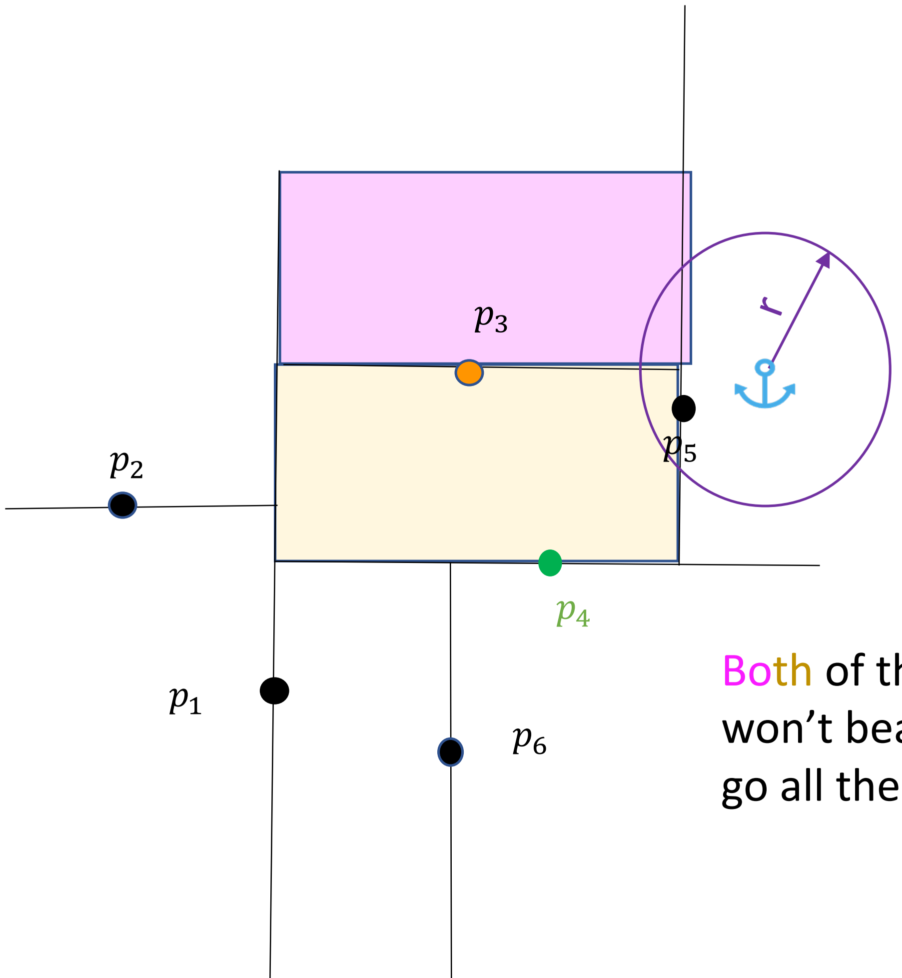


1. **Visit:** p_3 **Test:** Not within range ☹
2. Recursing on **either** left or right child first reasonable in this special case! :O
   - **By convention**, let's pick right (≥`currDim`) first

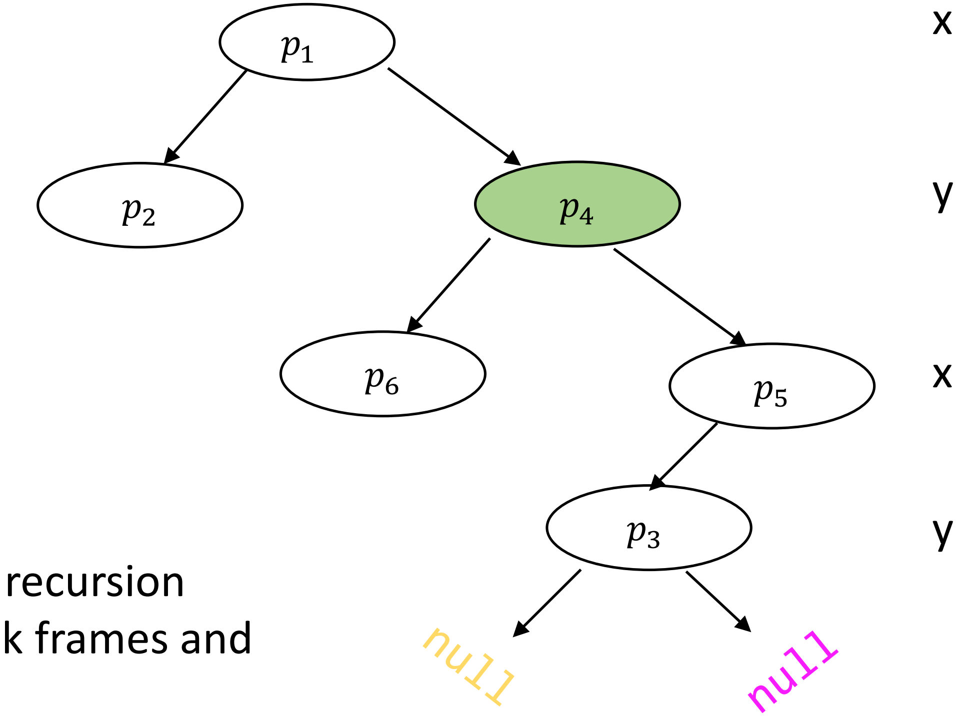


Range Query examples

2D space



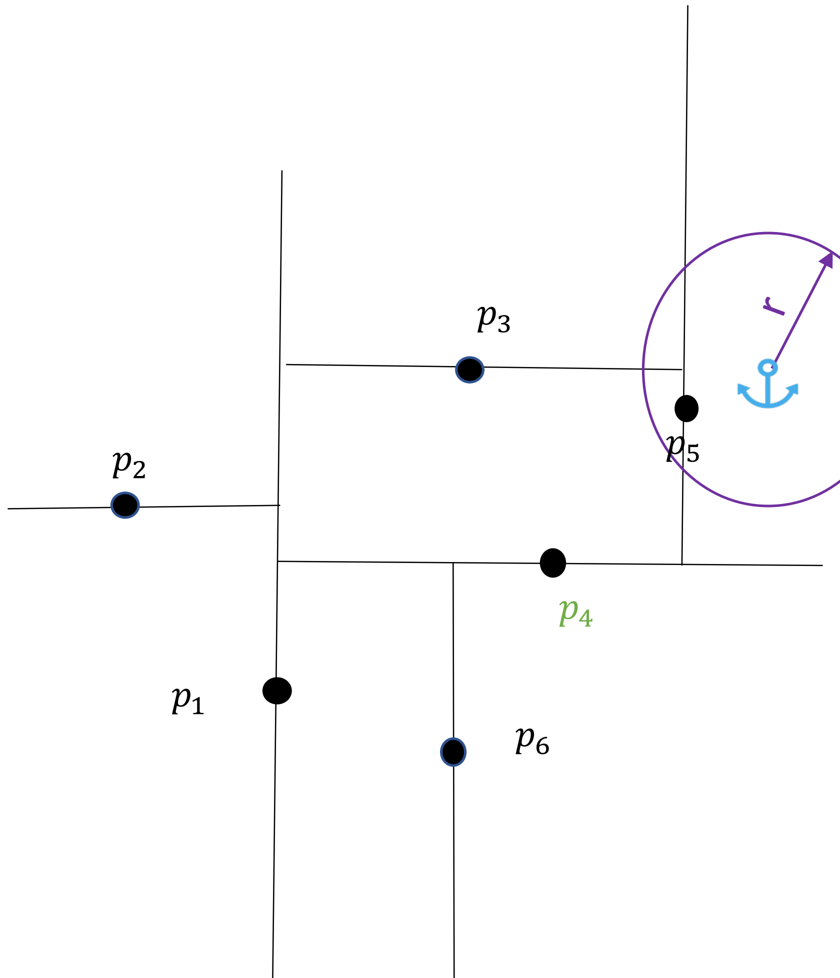
Corresponding KD-tree



Both of these children are `null`, so the recursion won't bear any fruit… let's pop some stack frames and go all the way back to p_4

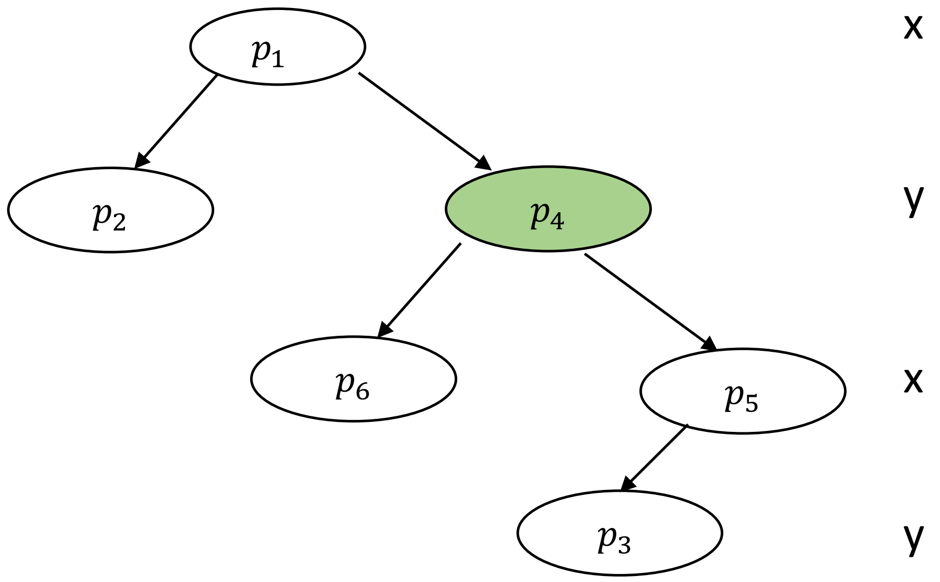
Range Query examples

2D space



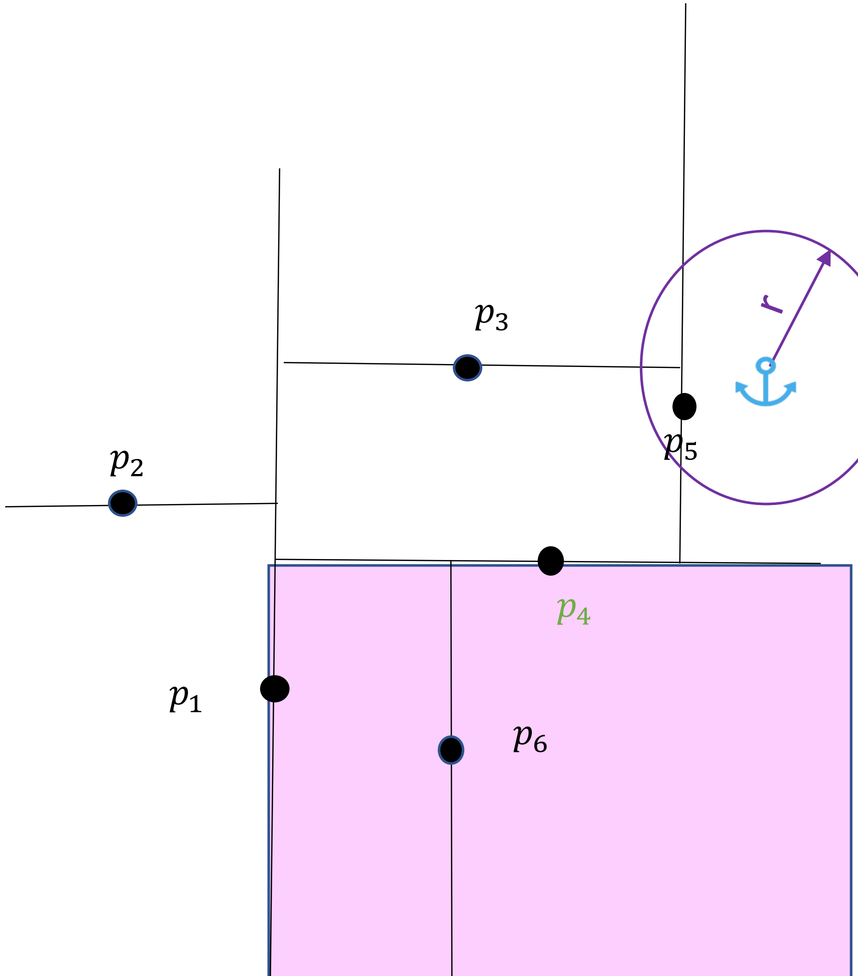
1. **Backtrack to:** p_4

Corresponding KD-tree

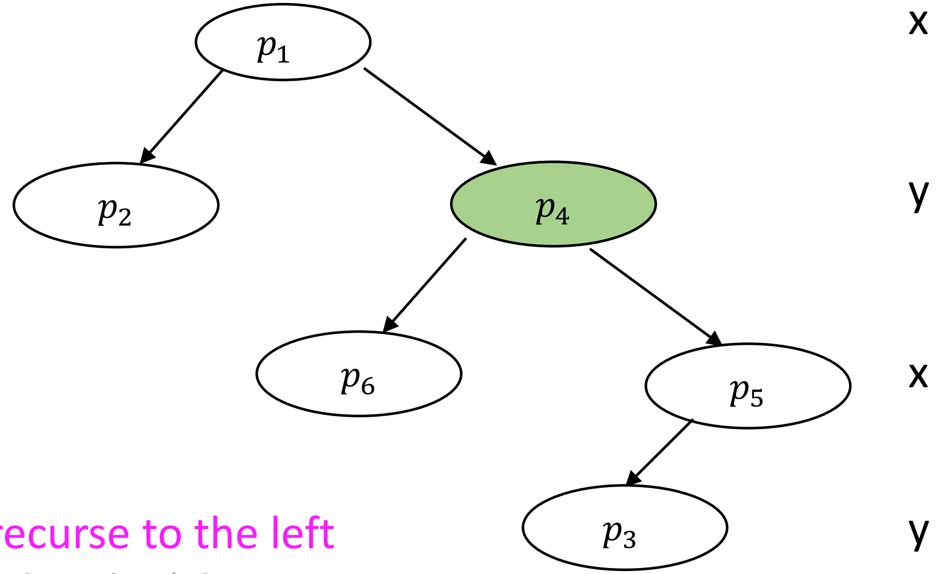


Range Query examples

2D space



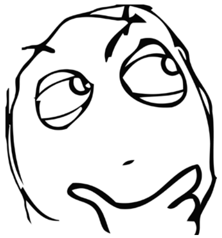
Corresponding KD-tree



1. **Backtrack to:** p_4
2. Does it make sense for us to recurse to the left subtree *(which we disregarded earlier)* ?

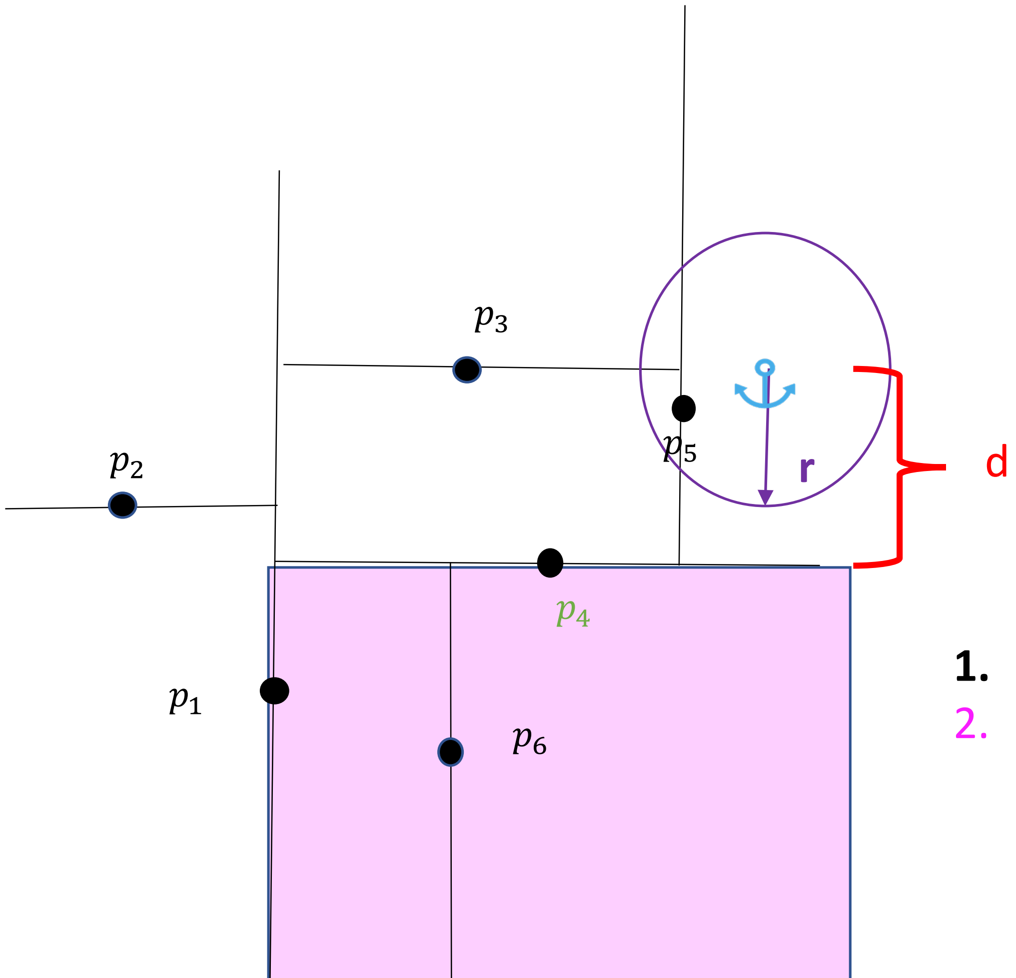
Yes

No

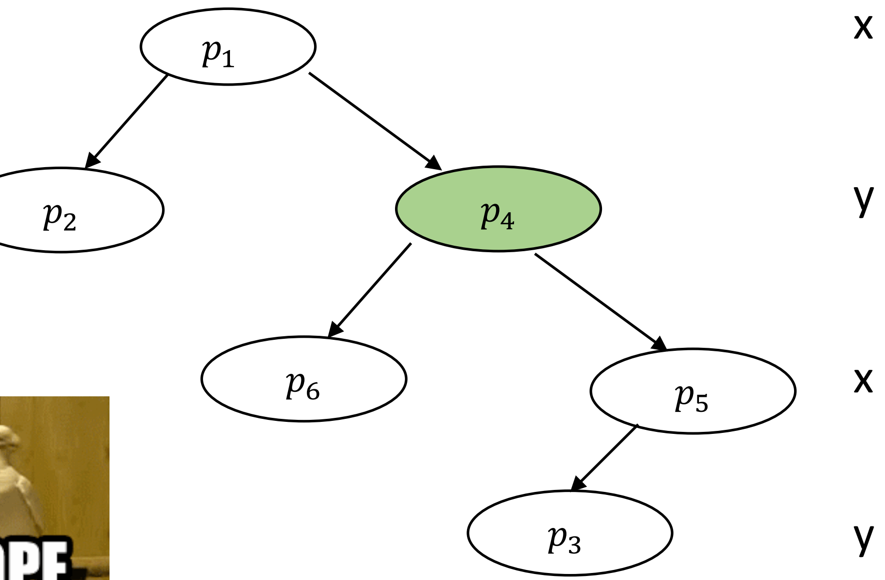


Range Query examples

2D space



Corresponding KD-tree



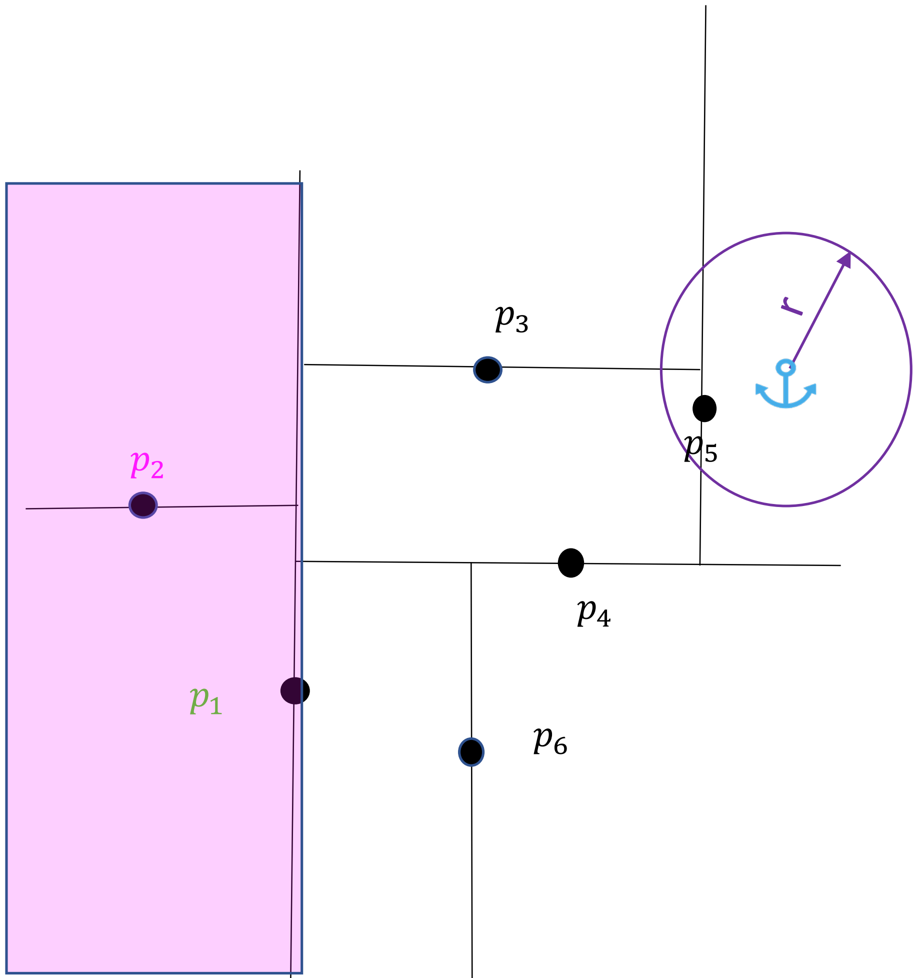
1. **Backtrack to:** p_4
2. Does it make sense for us to recurse to the left subtree *(which we disregarded earlier)* ?
   - The y-distance d between p_4 and the anchor point is greater than r!

Yes

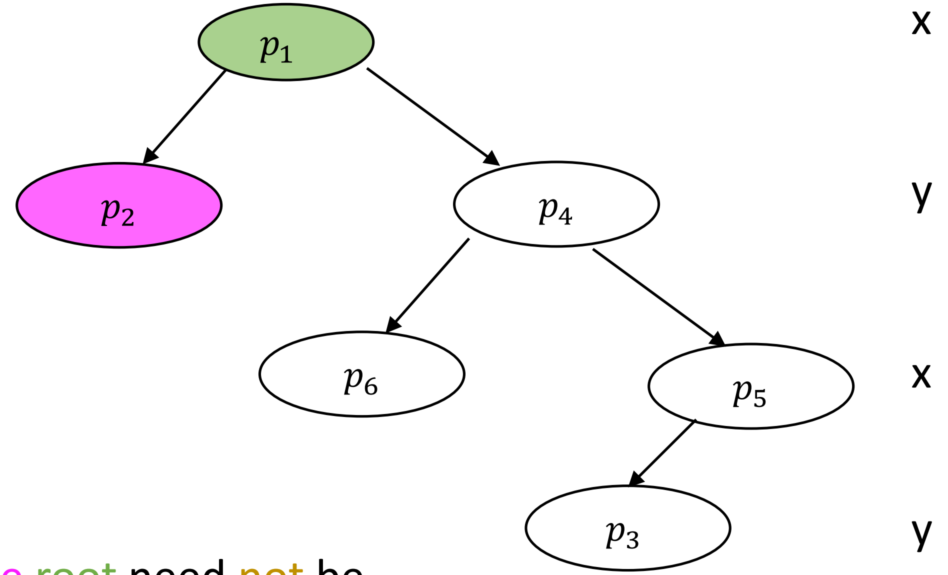
No

Range Query examples

2D space



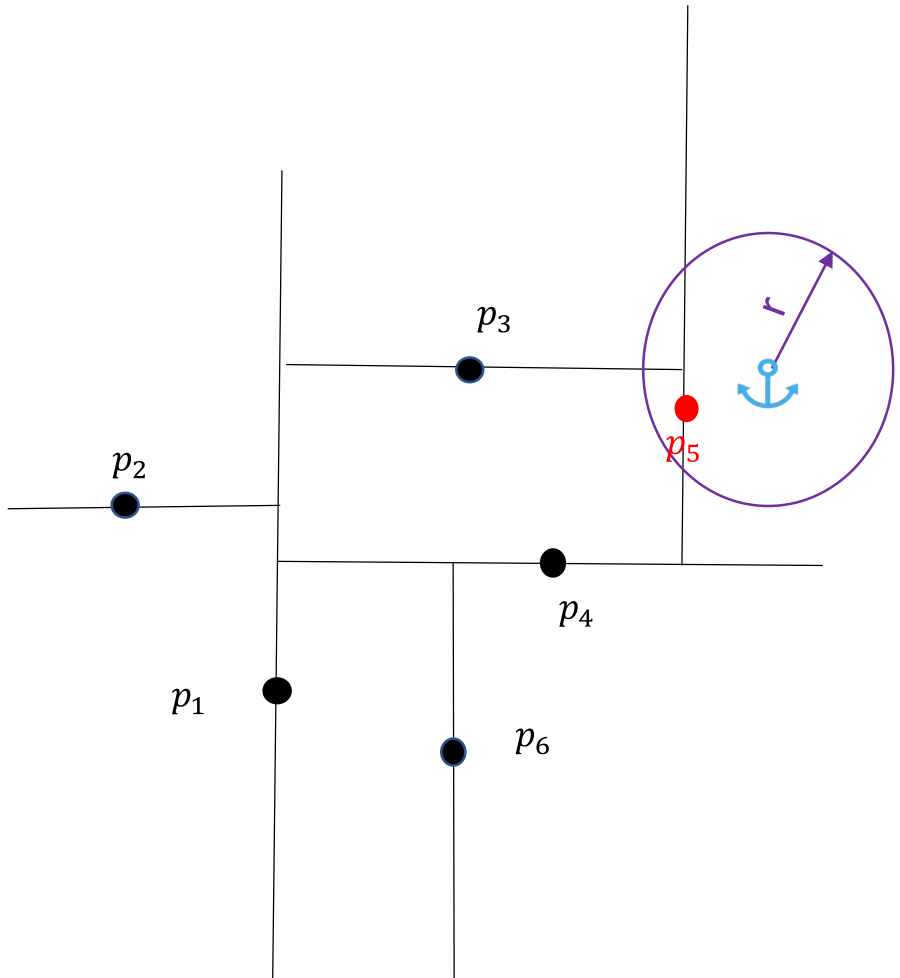
Corresponding KD-tree



Similarly, the left subtree of the root need not be examined, since the x-distance between p_2 and the anchor is greater than r!

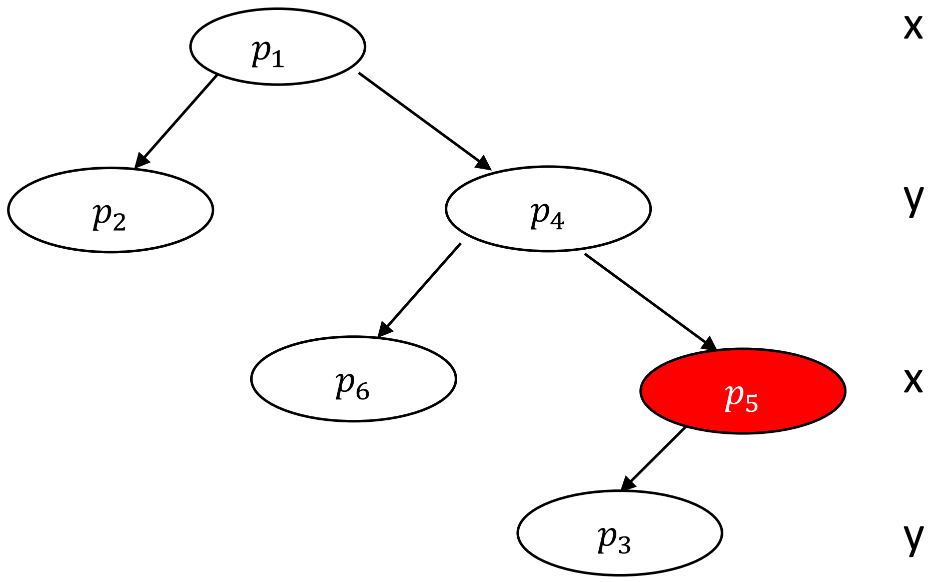
Range Query examples

2D space



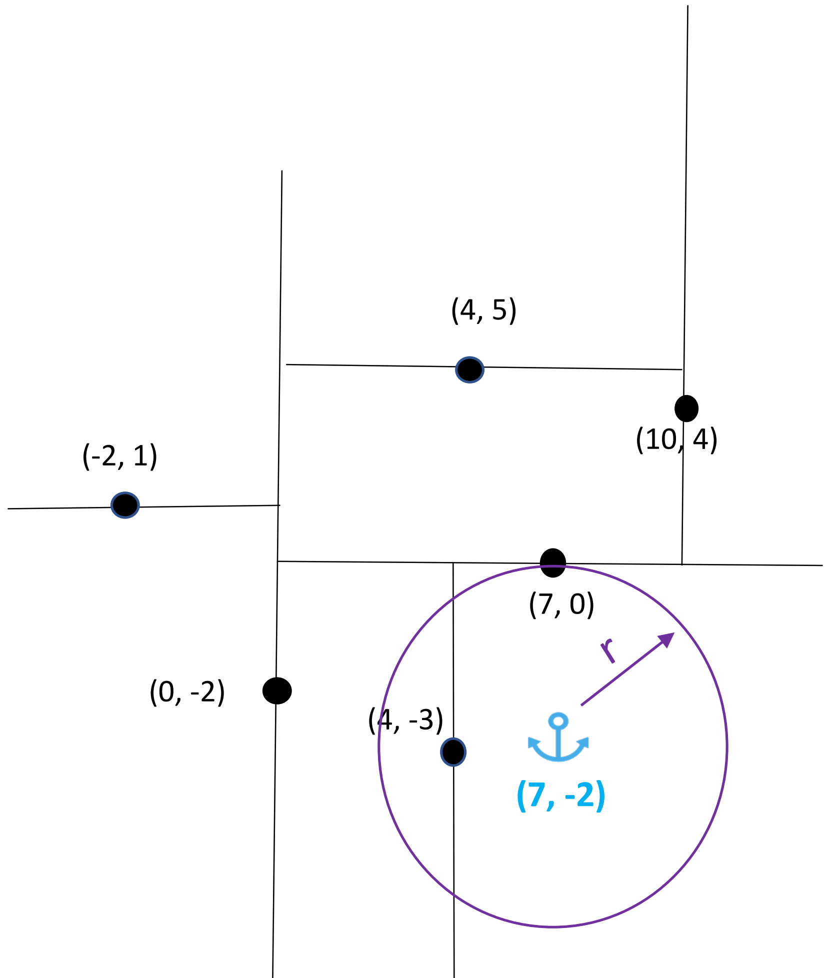
Final result: $\{p_5\}$

Corresponding KD-tree

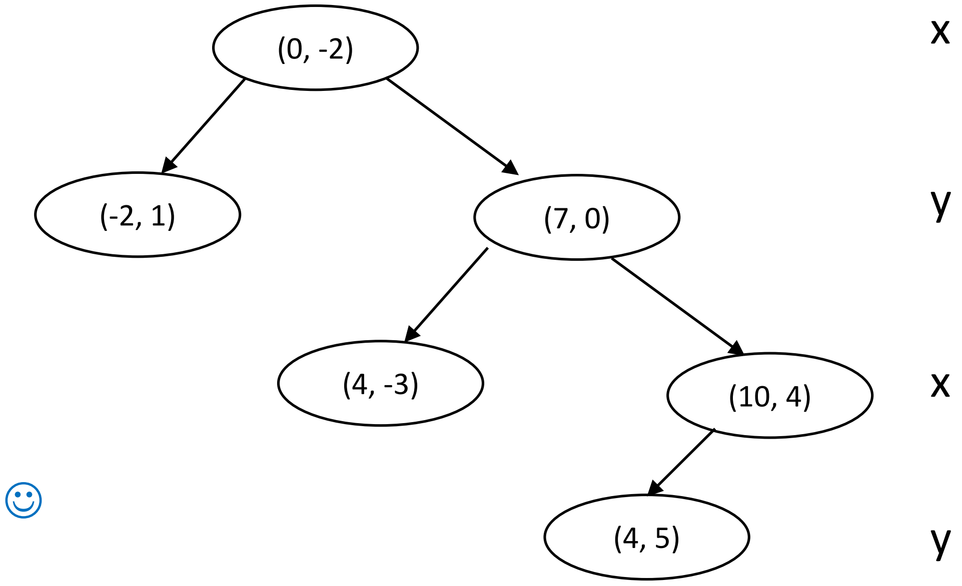


Range Query examples

2D space



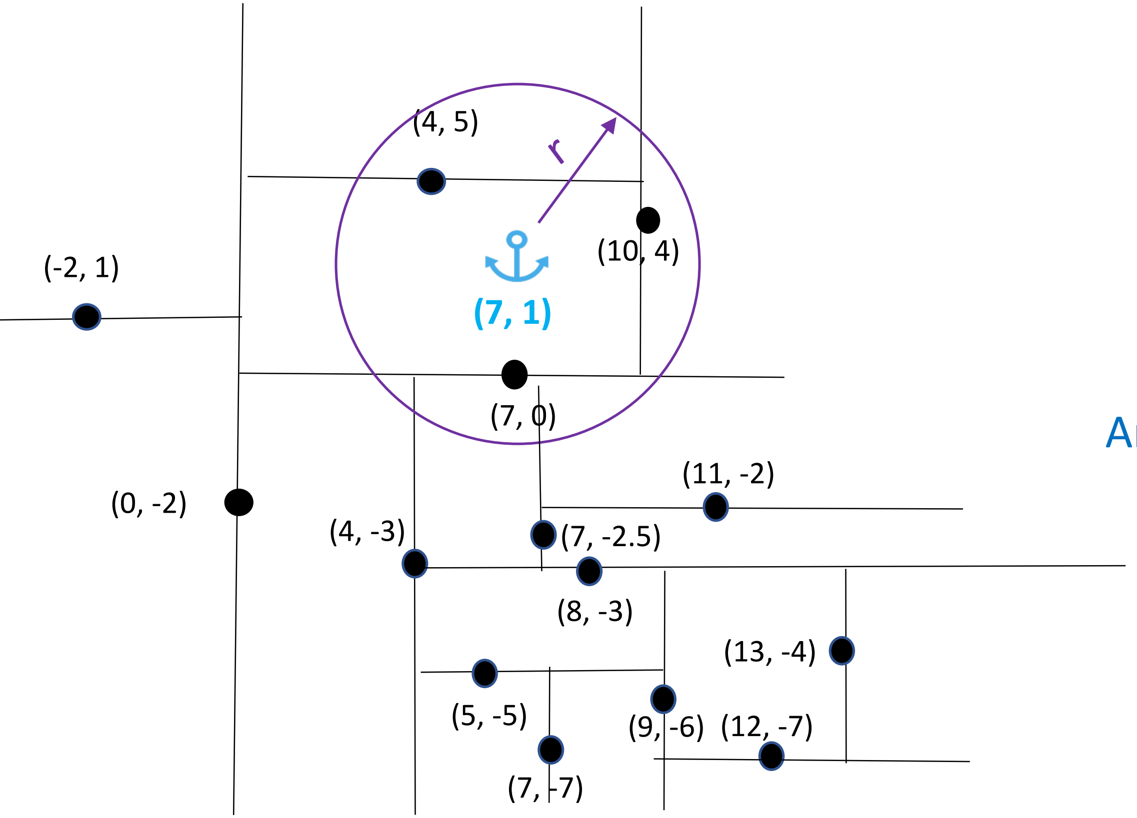
Corresponding KD-tree



Let's try to trace this one ☺

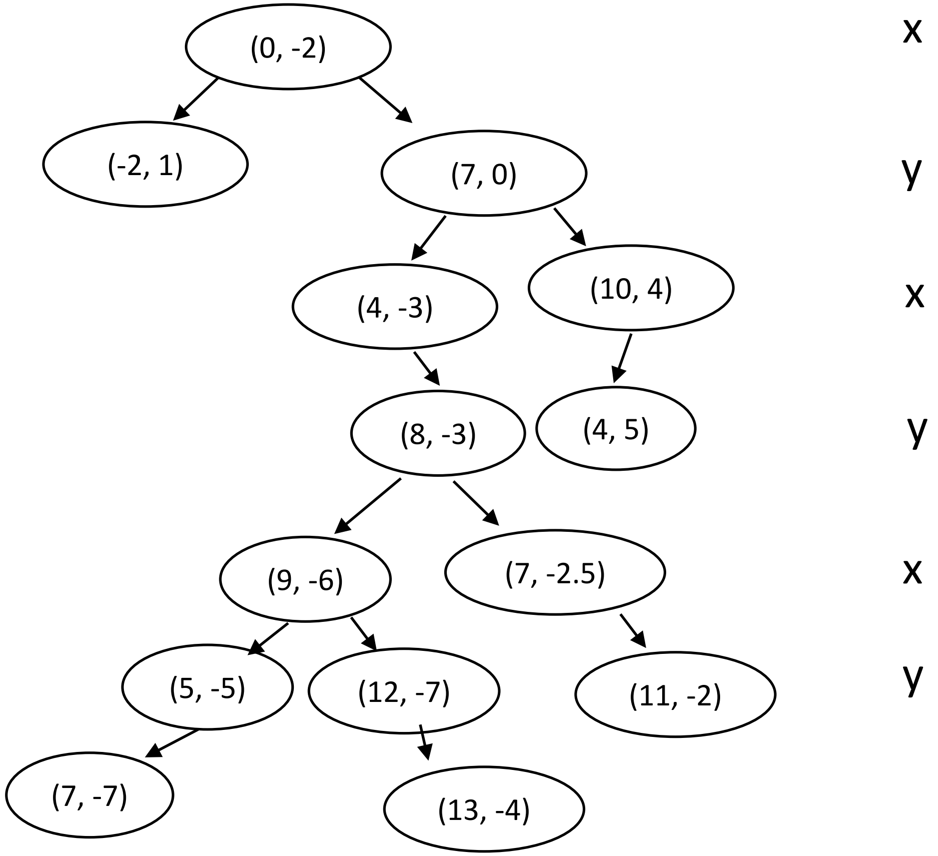
Range Query examples

2D space



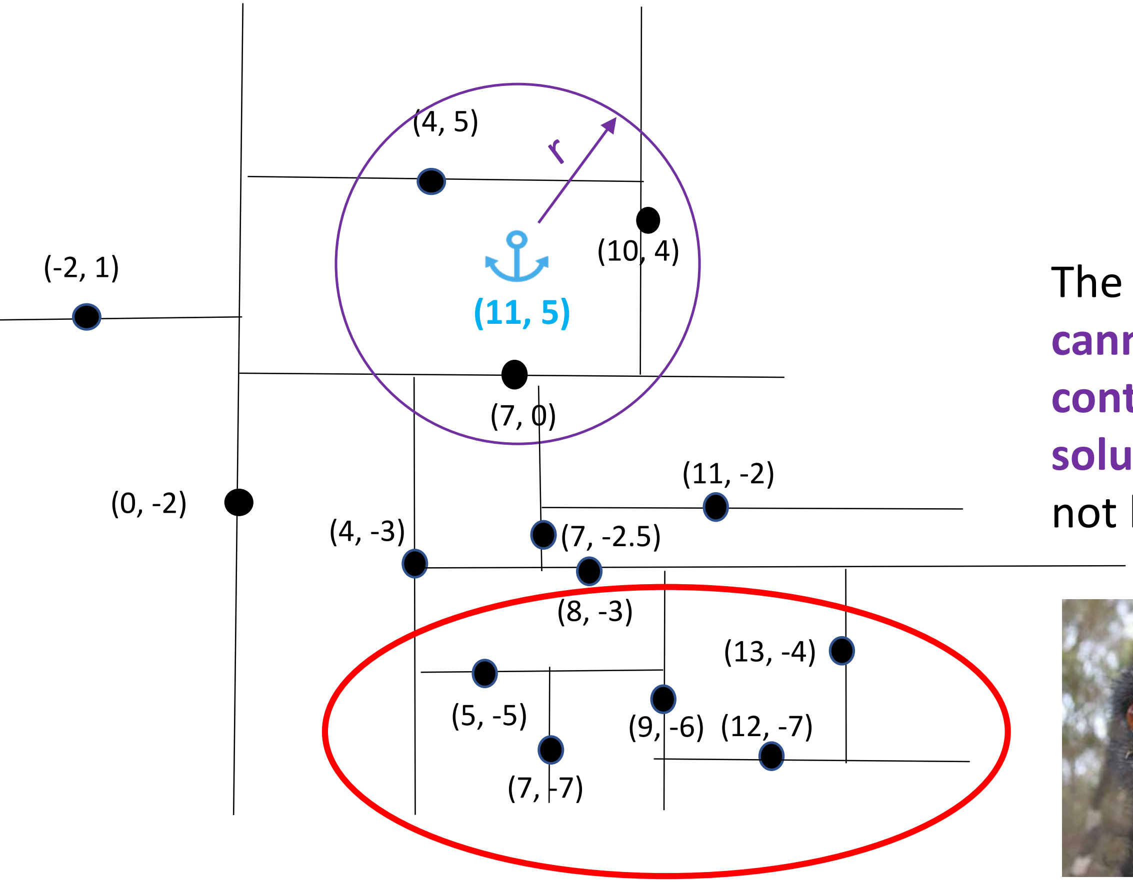
And this one! ☺

Corresponding KD-tree



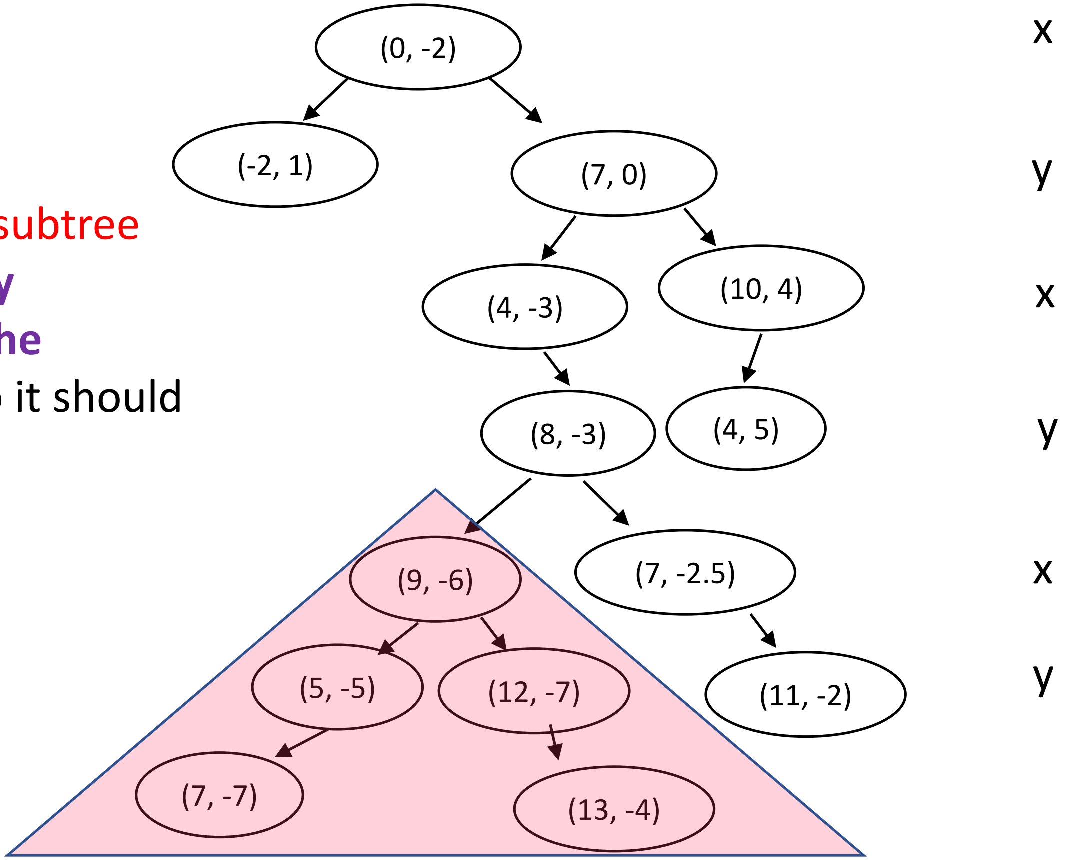
Range Query examples

2D space



Corresponding KD-tree

The entire red subtree **cannot possibly contribute to the solution set**, so it should not be visited!



35.7% of the tree won't be visited!

Take-home messages

1. As we go down the tree, we behave **greedily,** by traversing the subtree likeliest to give us answers.
   - This is important in an application that mutates a global collection of the answers but whose tree-traversing thread can die for whatever reason!
2. When we backtrack up the tree, we potentially prune away large portions of the dataset since we are *guaranteed* to not be able to improve upon our search!
   - A tree-like structure like a KD-Tree helps **a ton** with this!
   - For dense datasets, this slows down as we approach the point, and speeds up as we get away from it!

Nearest neighbor: idea

- Maintain a current "best guess" for the closest neighbor and update it as you go down the tree
- Initially this will be a tuple `(null,+∞)`
- Once we visit the root, we will update it to `(root,`distance(query, root)`)`

Nearest neighbor: idea

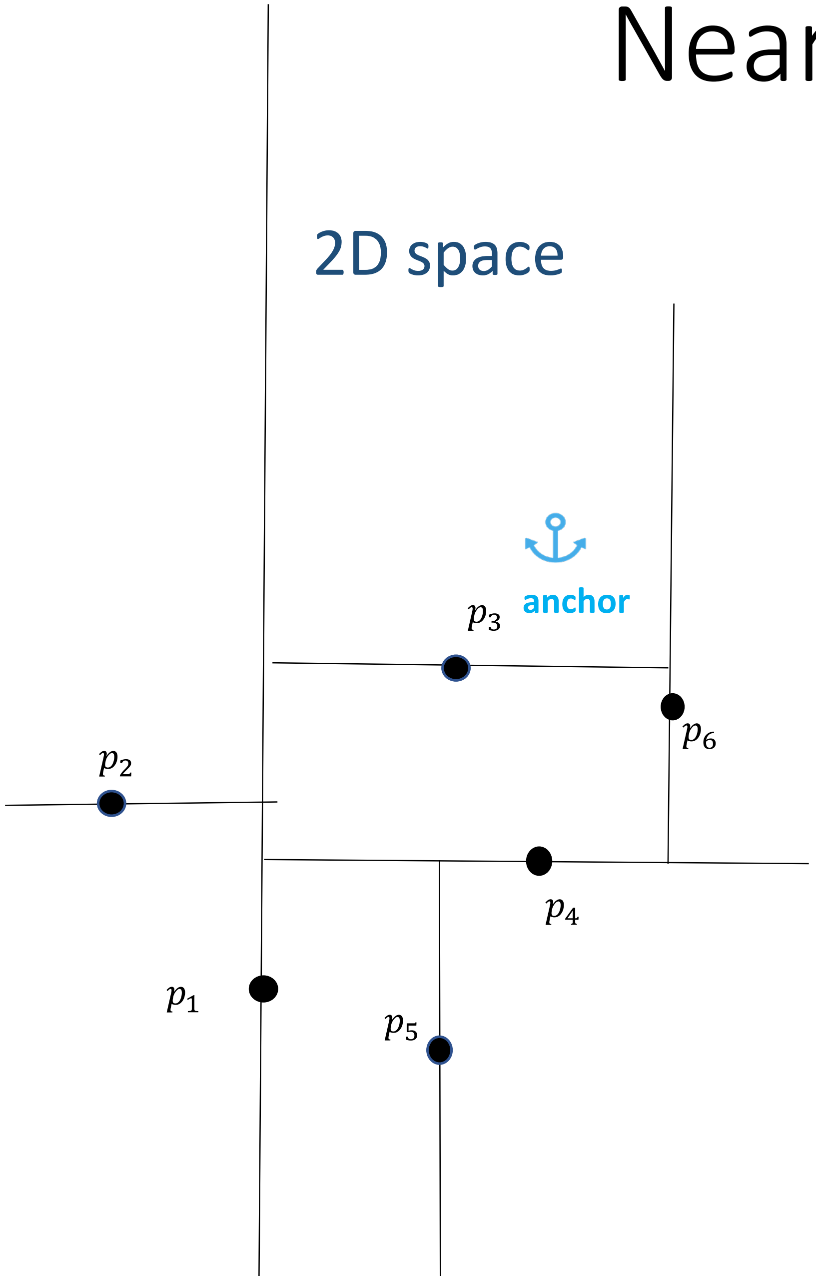
- Maintain a current "best guess" for the closest neighbor and update it as you go down the tree
- Initially this will be a tuple `(null,+∞)`
- Once we visit the root, we will update it to `(root,`distance(query, root)`)`
- Then, we have to decide the order of visiting subtrees.

Nearest neighbor: idea

- Maintain a current "best guess" for the closest neighbor and update it as you go down the tree
- Initially this will be a tuple `(null,+∞)`
- Once we visit the root, we will update it to `(root,`distance(query, root)`)`
- Then, we have to decide the order of visiting subtrees.
  - **Similar approach to range queries**: visit the subtree where you're likelier to improve ***first***!

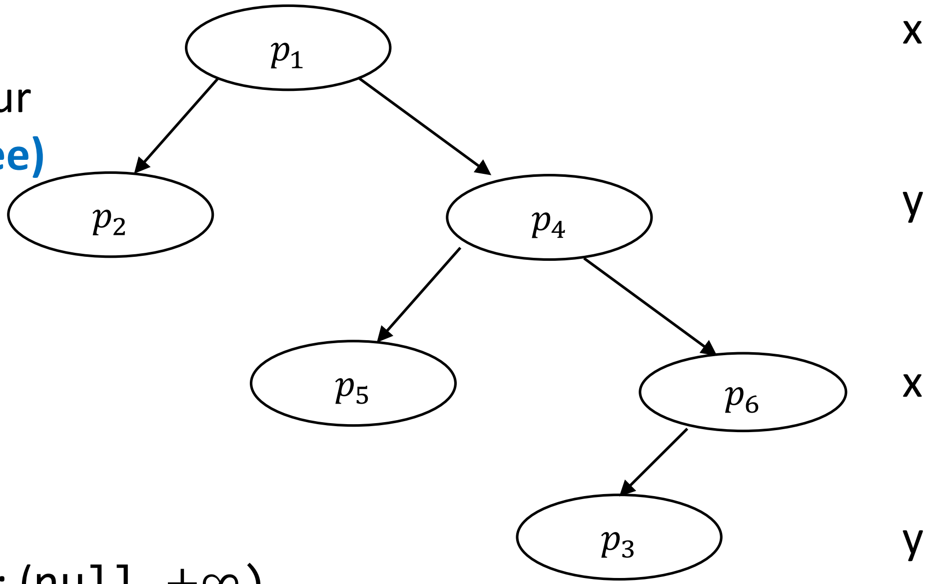
Nearest neighbor example

2D space



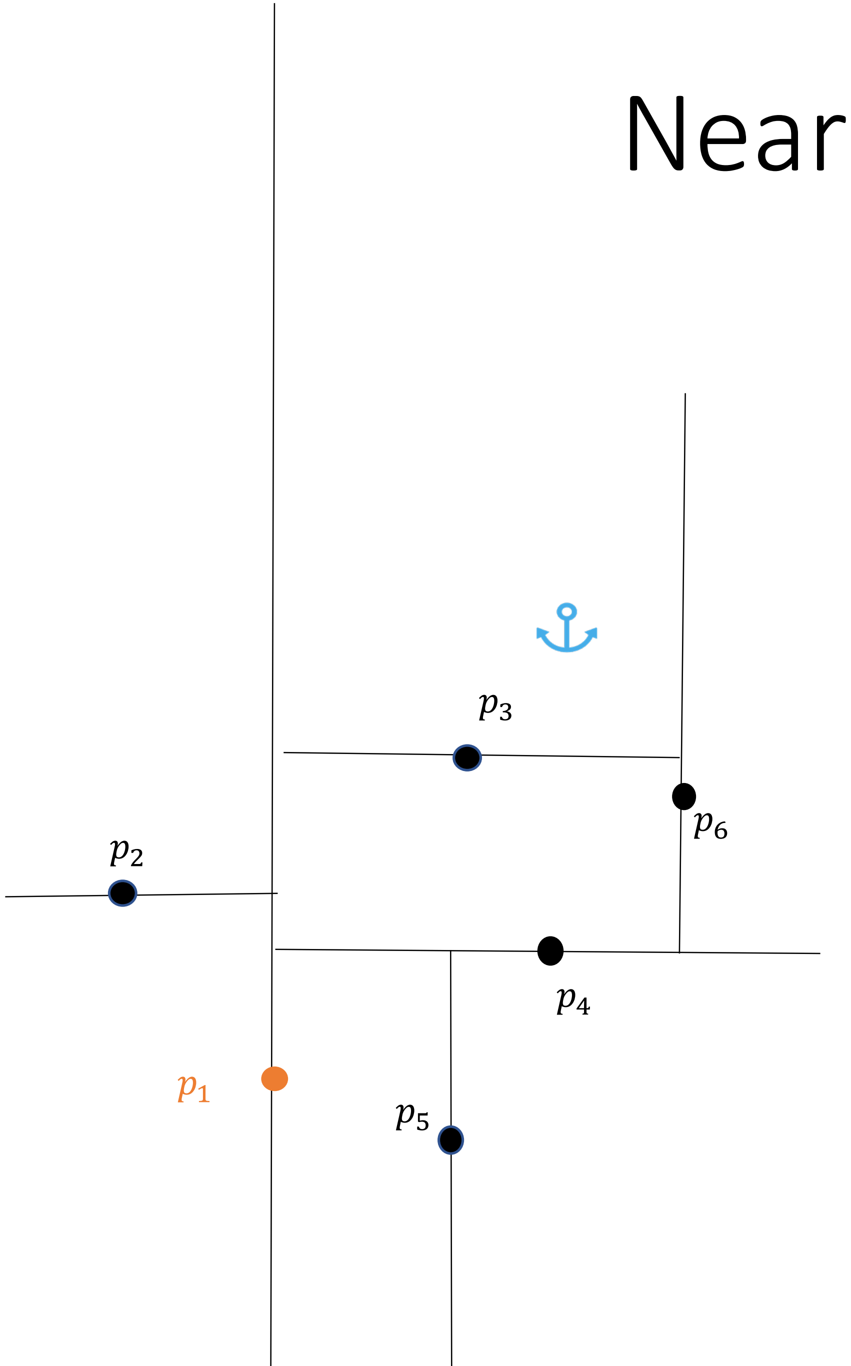
Corresponding KD-tree

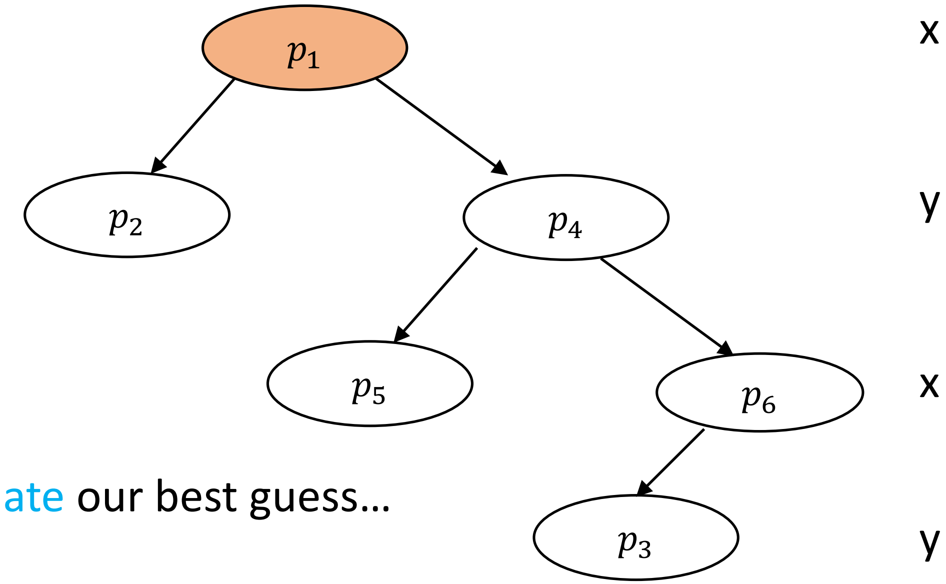
Task: Find the **nearest neighbor** of our **anchor point! (also a point in the tree)**



Current best guess: $(\texttt{null}, +\infty)$

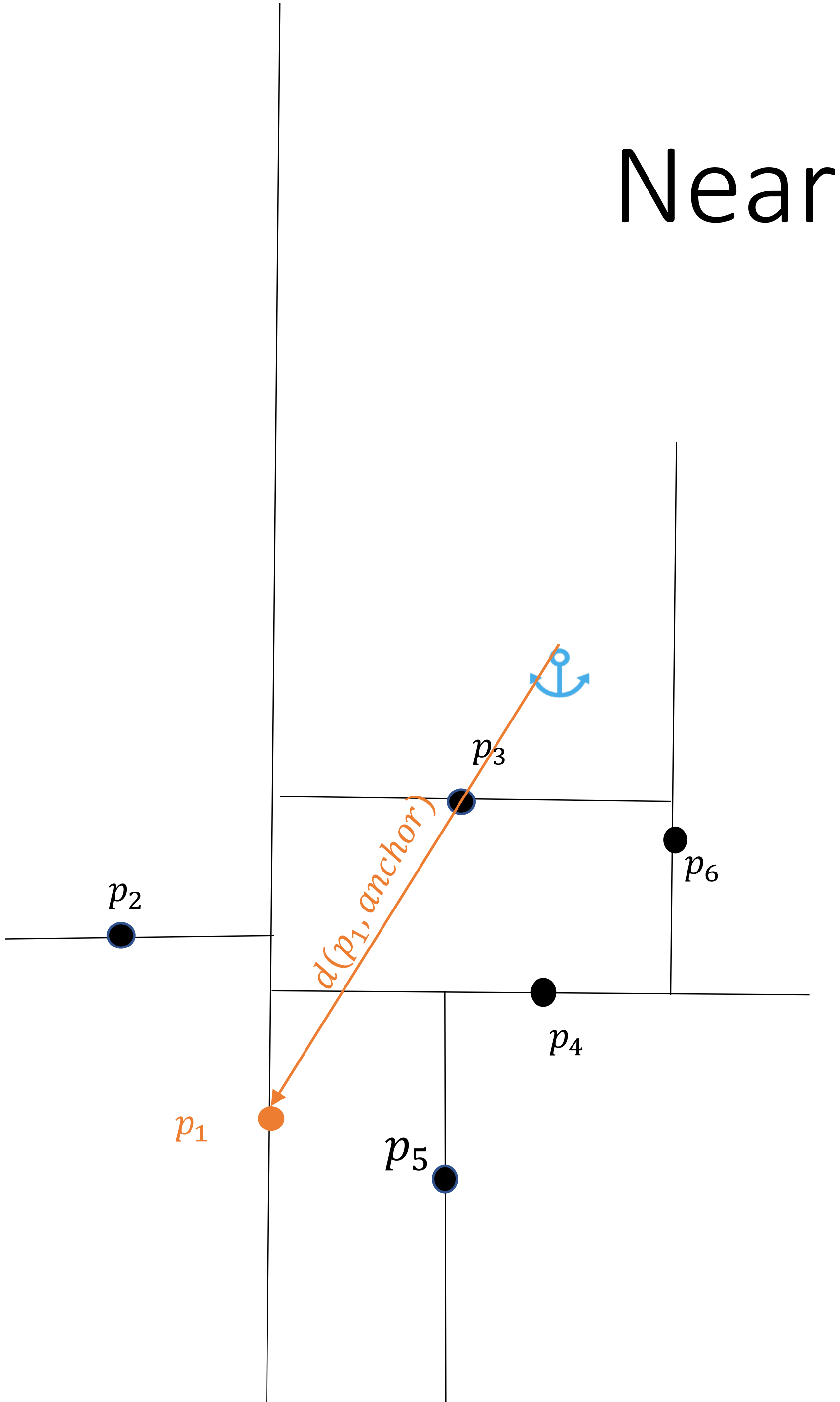
Nearest neighbor example

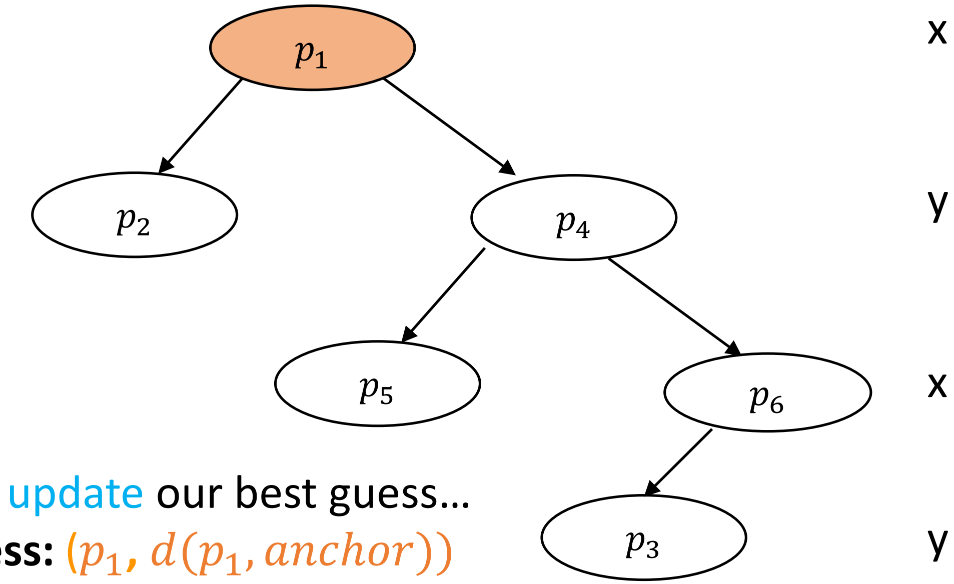




1. Visiting p_1 has us update our best guess…

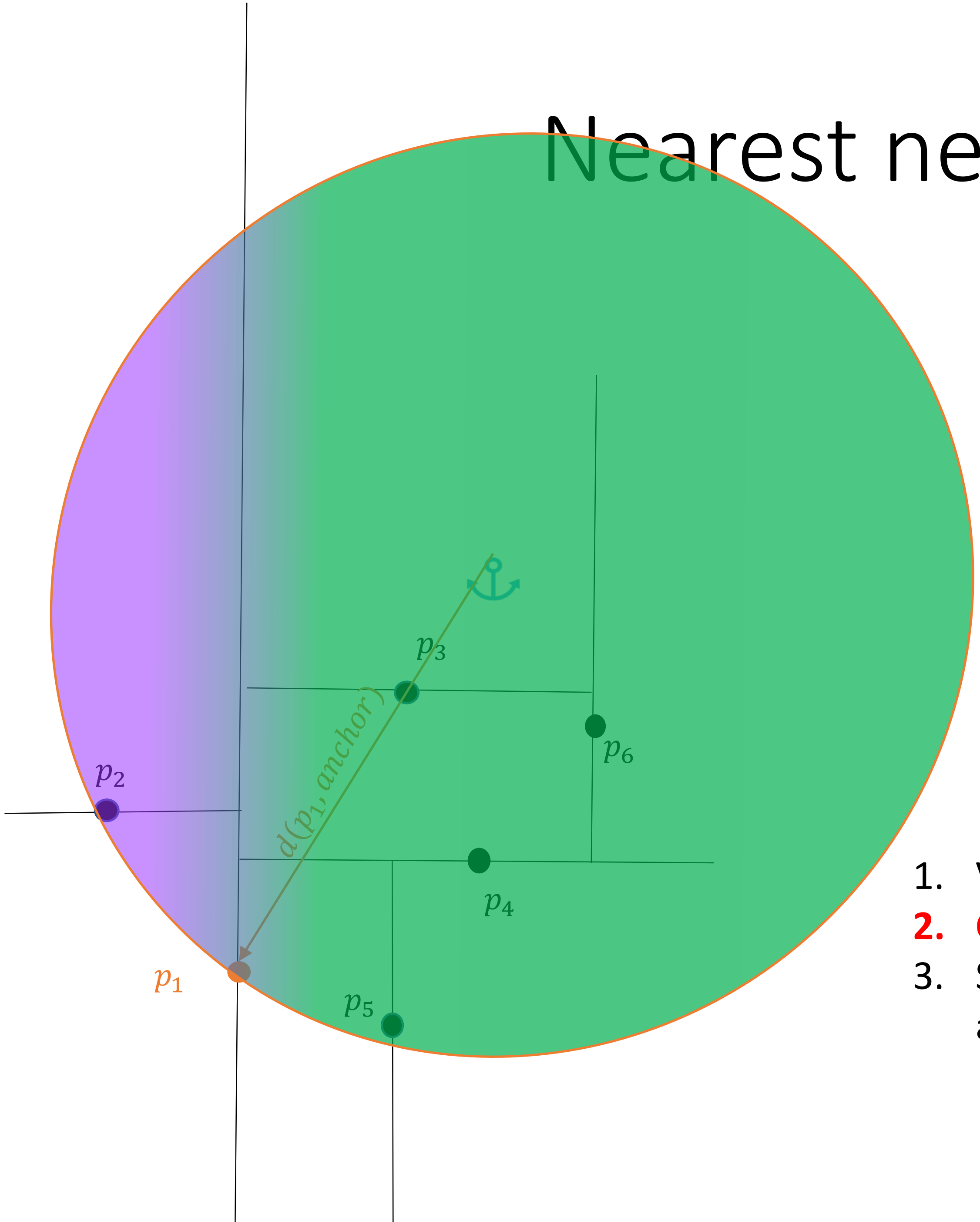
Nearest neighbor example

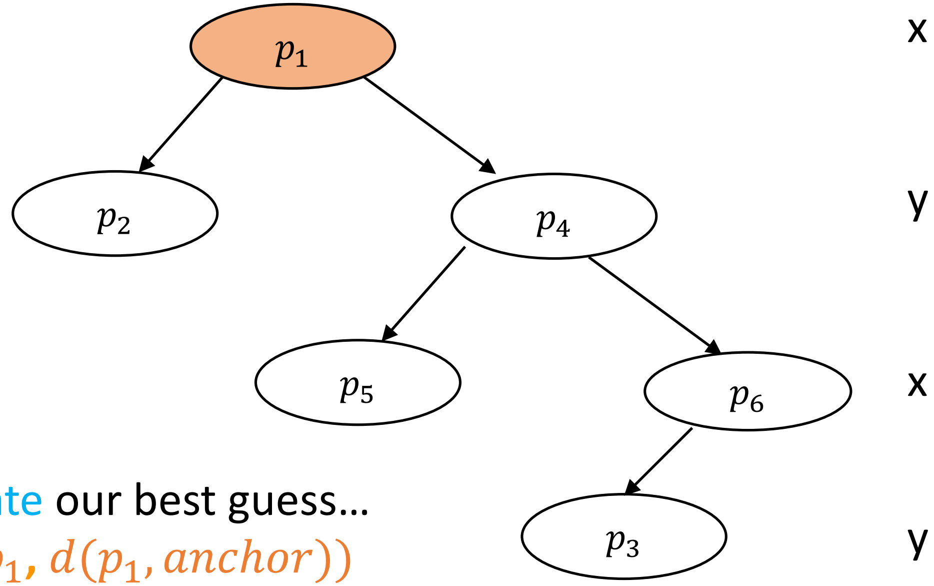




1. Visiting p_1 has us update our best guess…
2. **Current best guess:** $(p_1, d(p_1, anchor))$

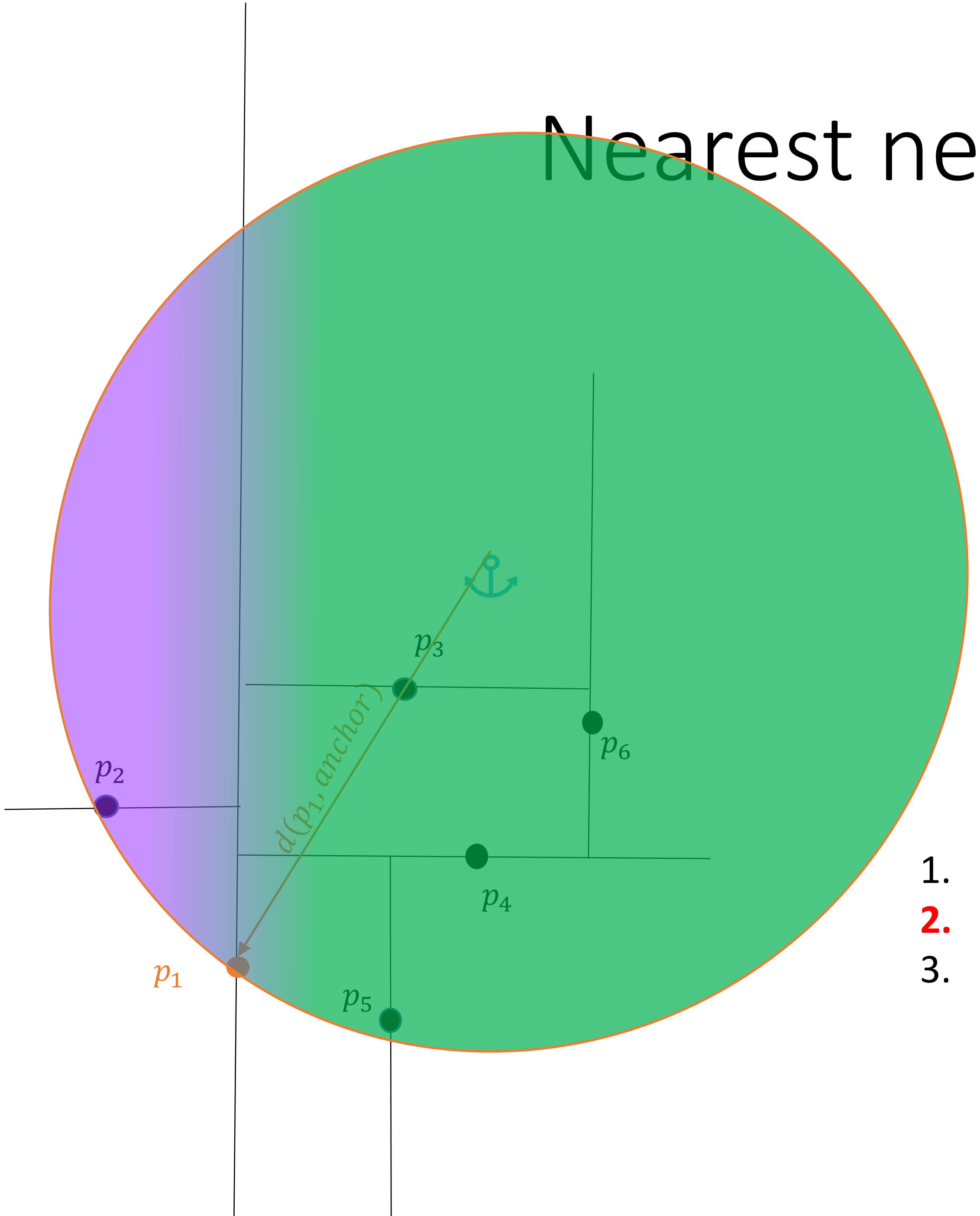
Nearest neighbor example

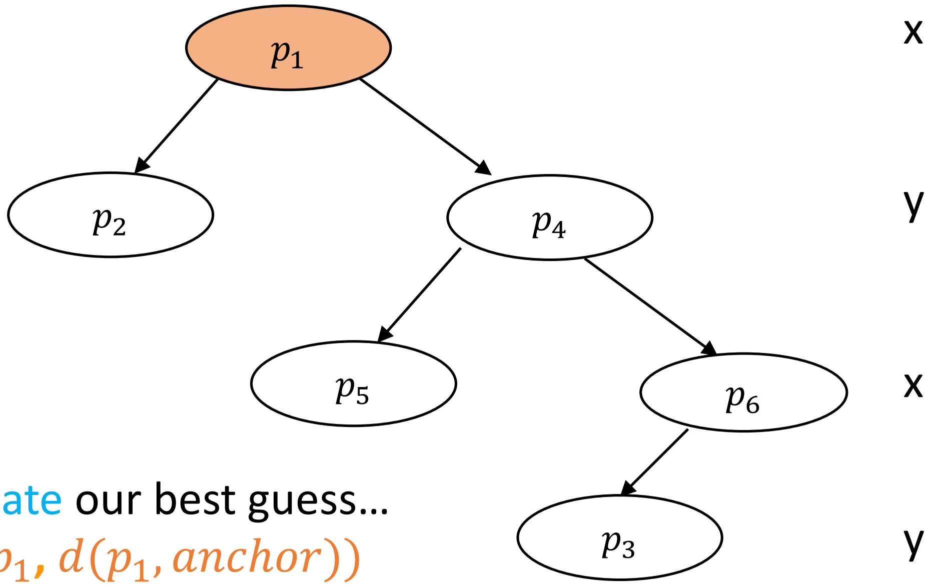




1. Visiting p_1 has us update our best guess…
2. **Current best guess:** $(p_1, d(p_1, anchor))$
3. Since the "tightest" circle found so far intersects both **the left** and the **right** half-planes, we could go ***either left or right!***

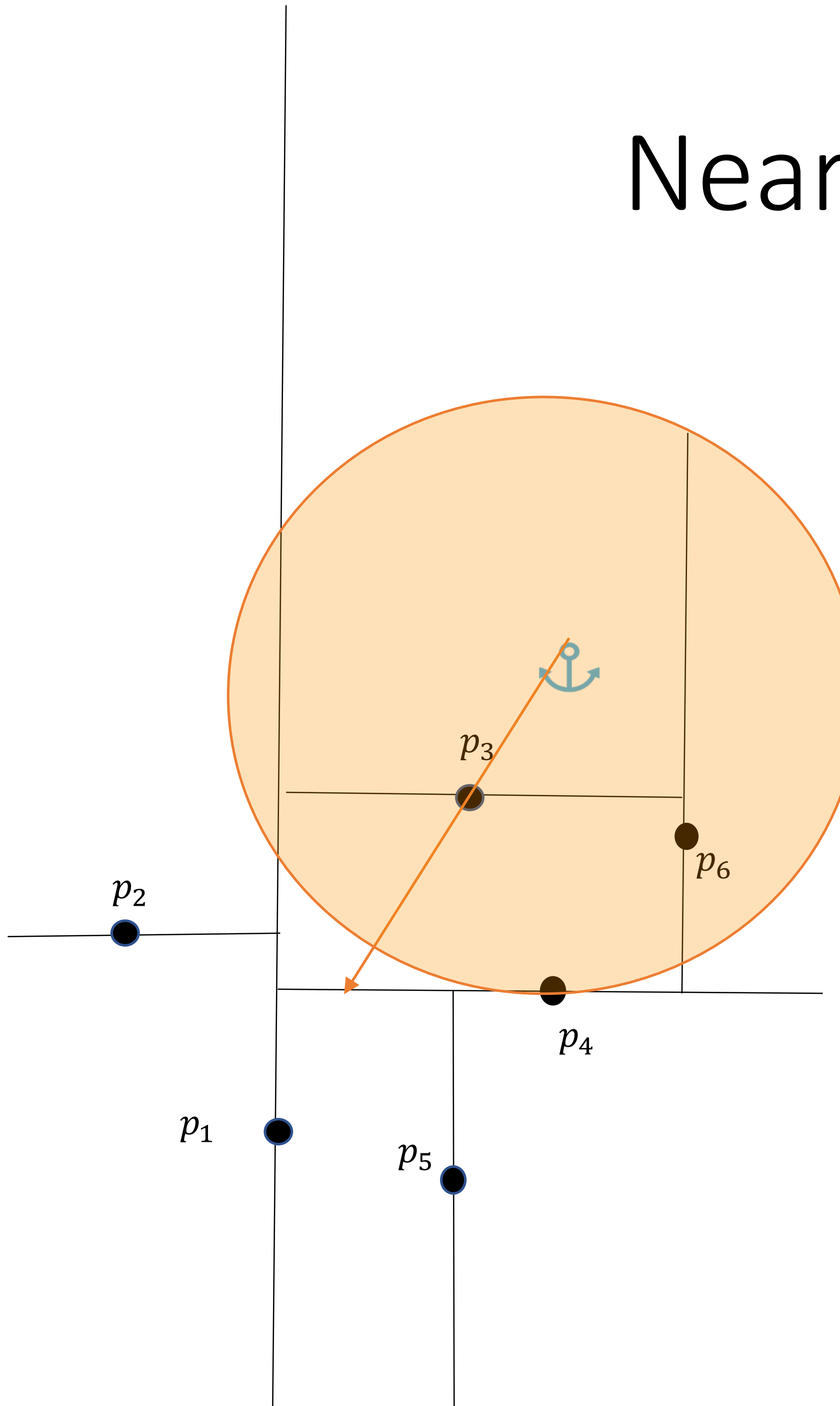
Nearest neighbor example

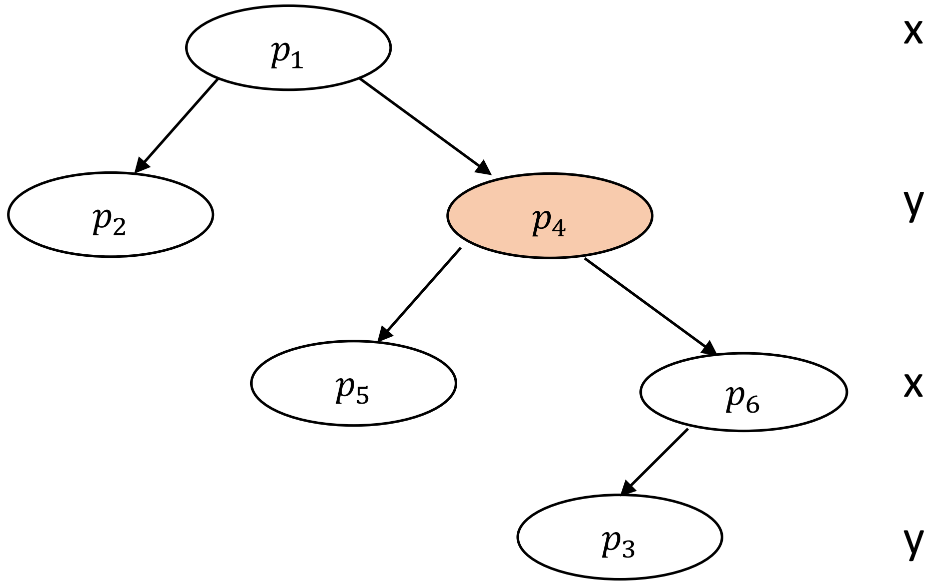




1. Visiting p_1 has us update our best guess…
2. **Current best guess:** $(p_1, d(p_1, anchor))$
3. Since the "tightest" circle found so far intersects both **the left** and the **right** half-planes, we could go ***either left or right!***
   - We choose to go right ***first*** **because the x-coordinate of the anchor is closer to our right!**

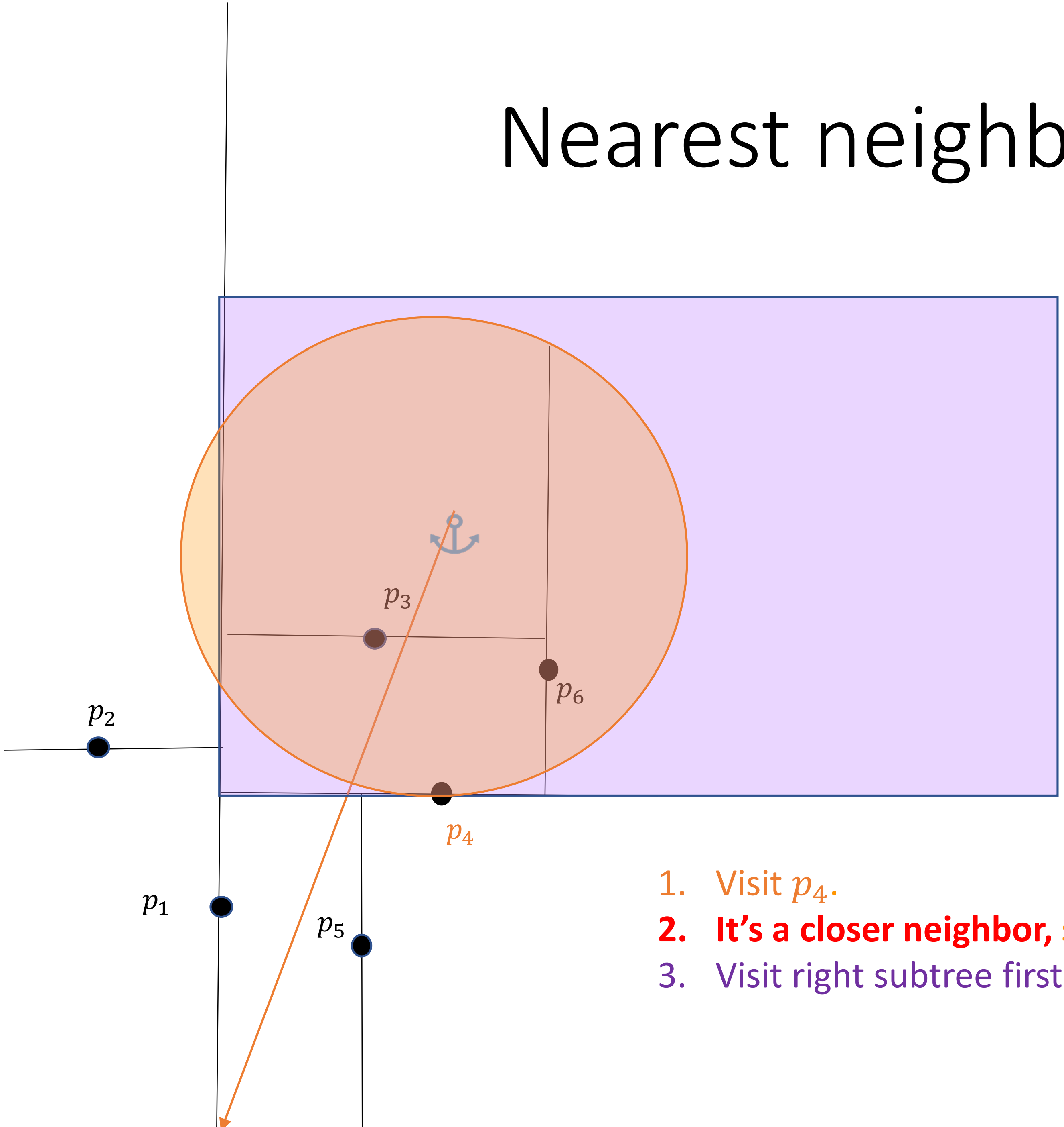
Nearest neighbor example

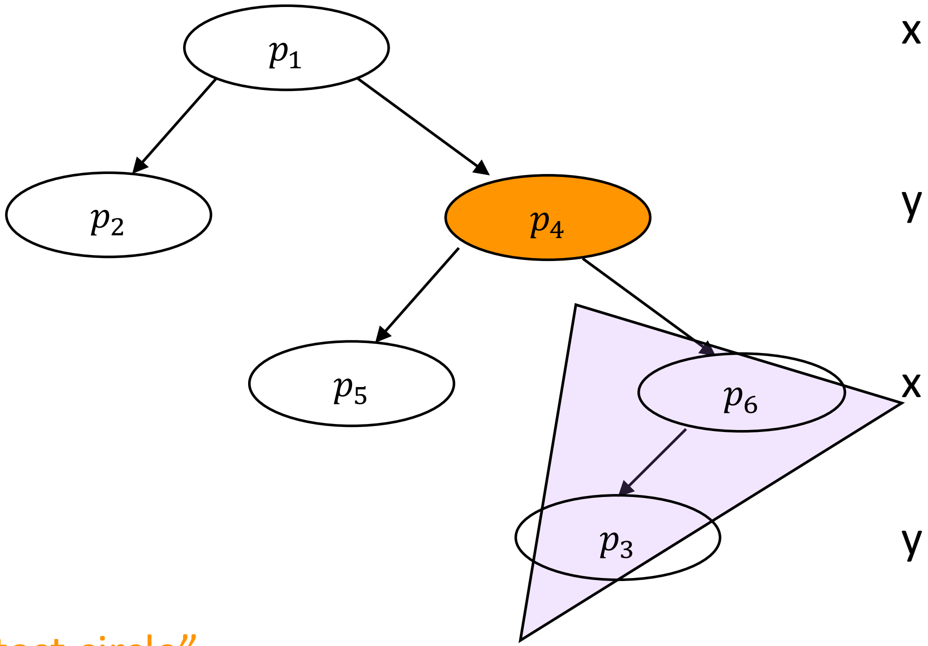




1. Visit p_4.
2. **It's a closer neighbor,** shrink "tightest circle".

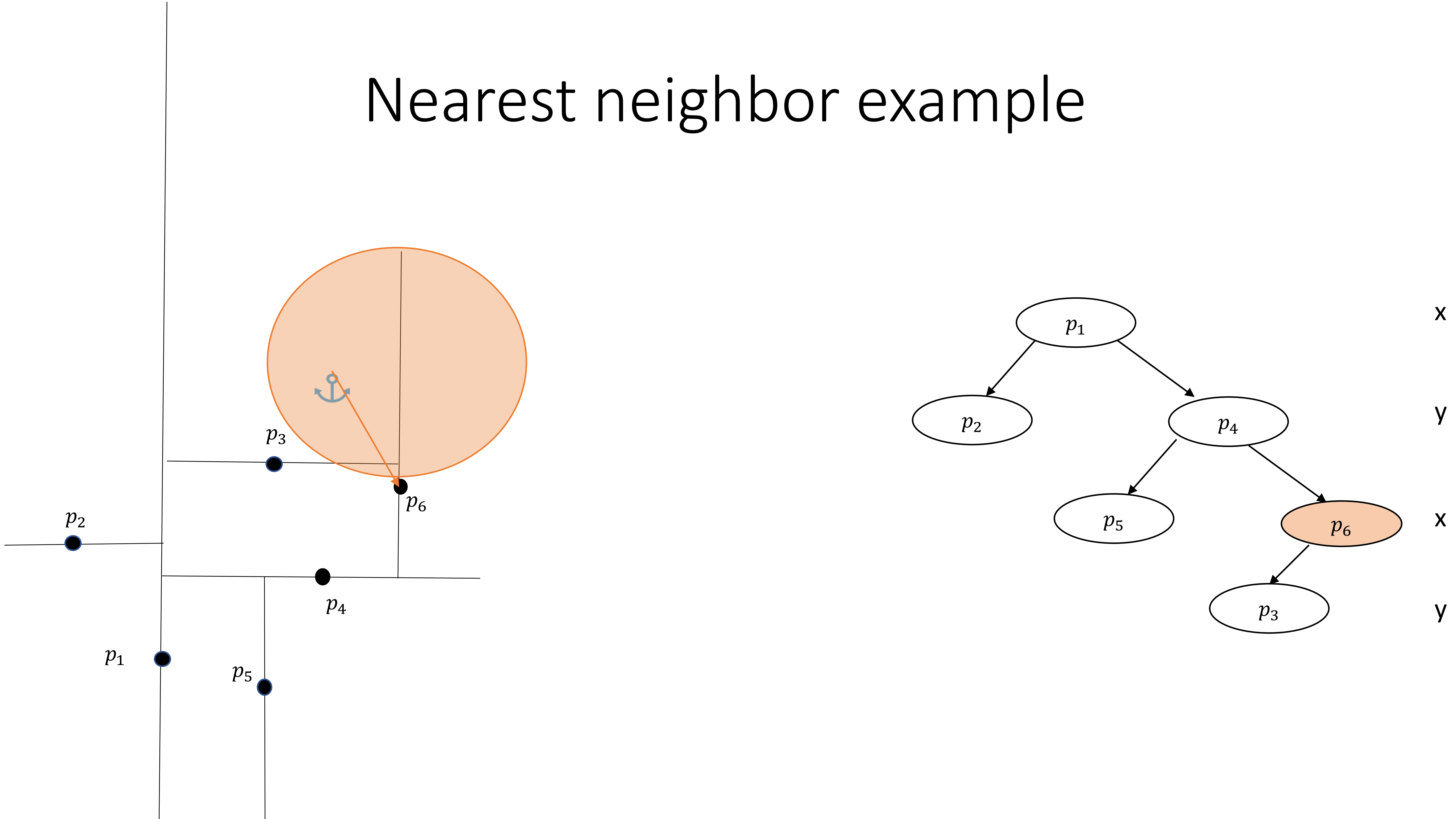
Nearest neighbor example



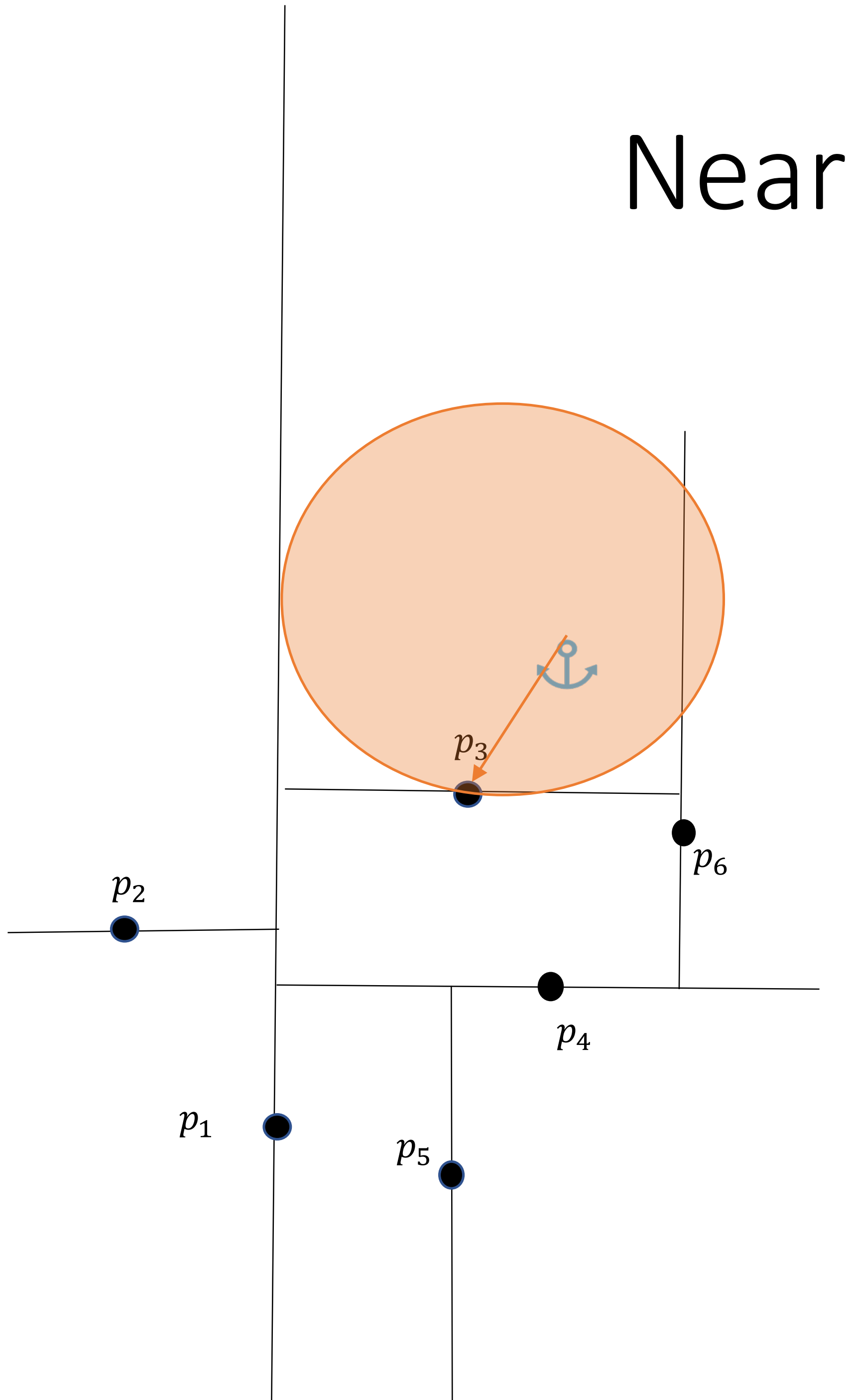


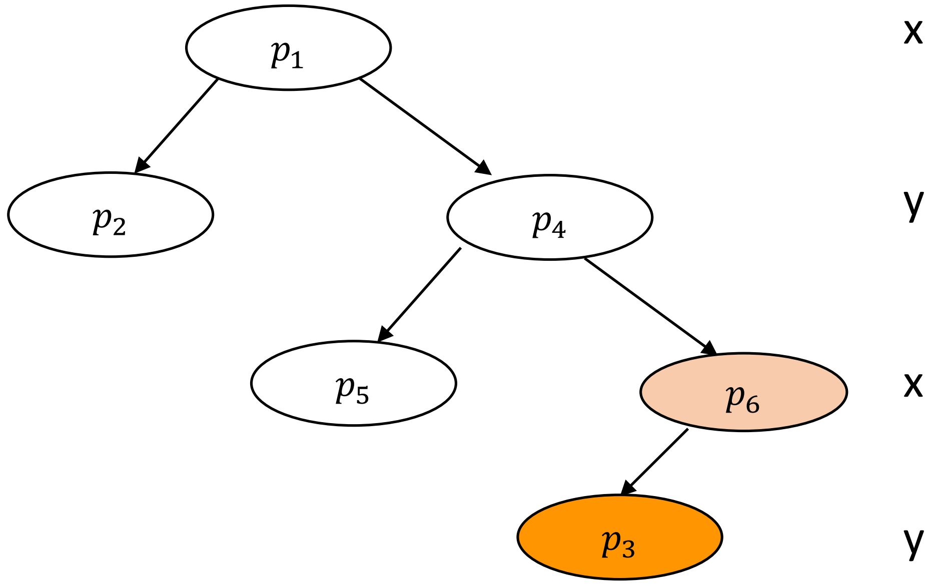
1. Visit p_4.
2. **It's a closer neighbor,** shrink "tightest circle".
3. Visit right subtree first since it's likelier to give us a better guess!

Nearest neighbor example

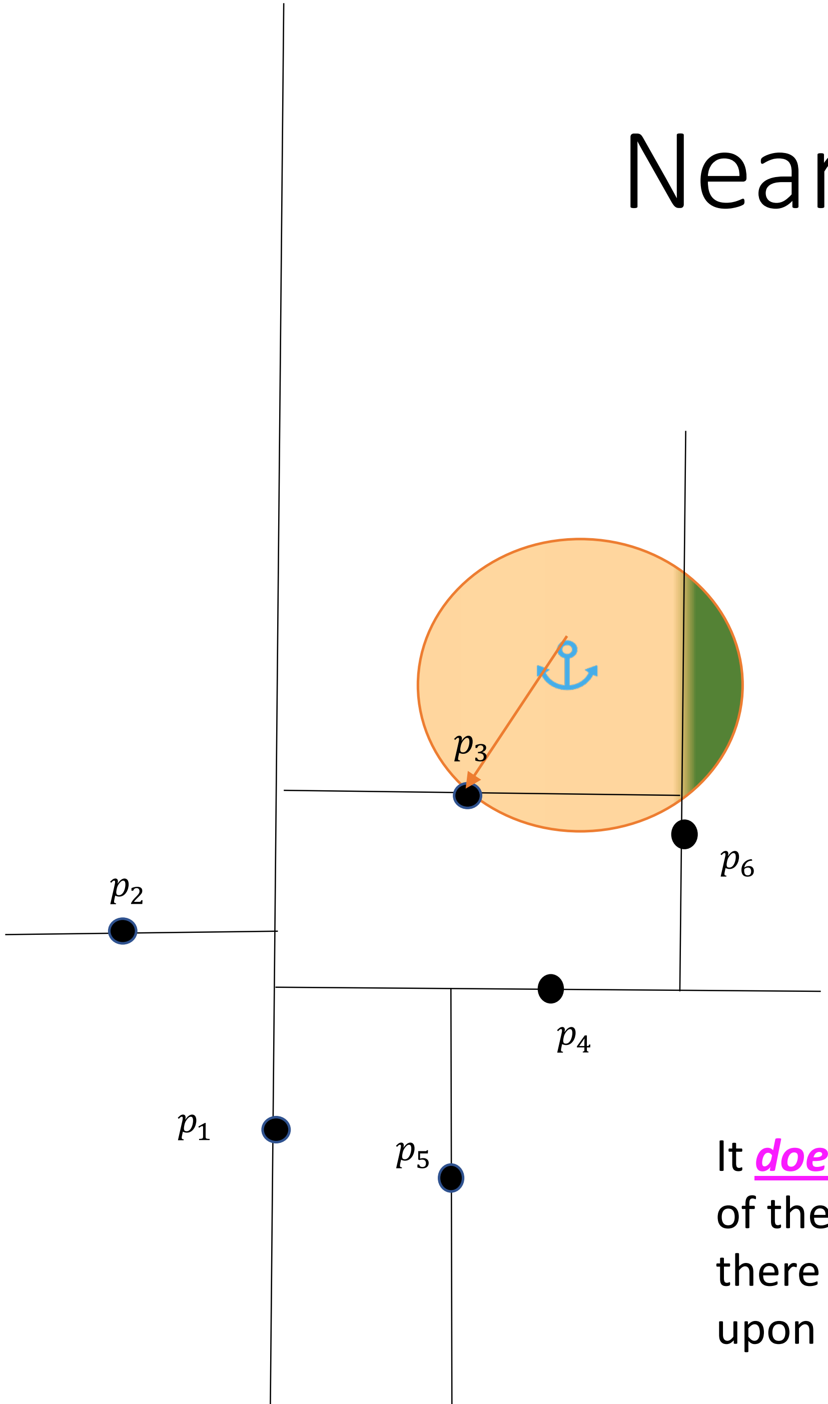


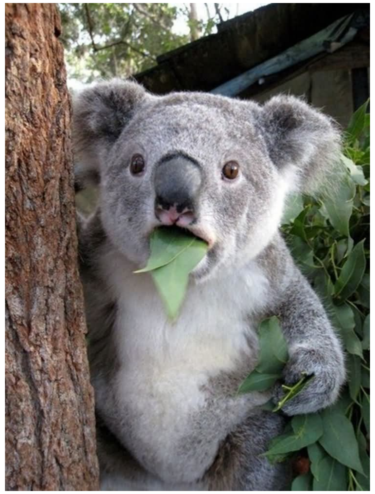
Nearest neighbor example

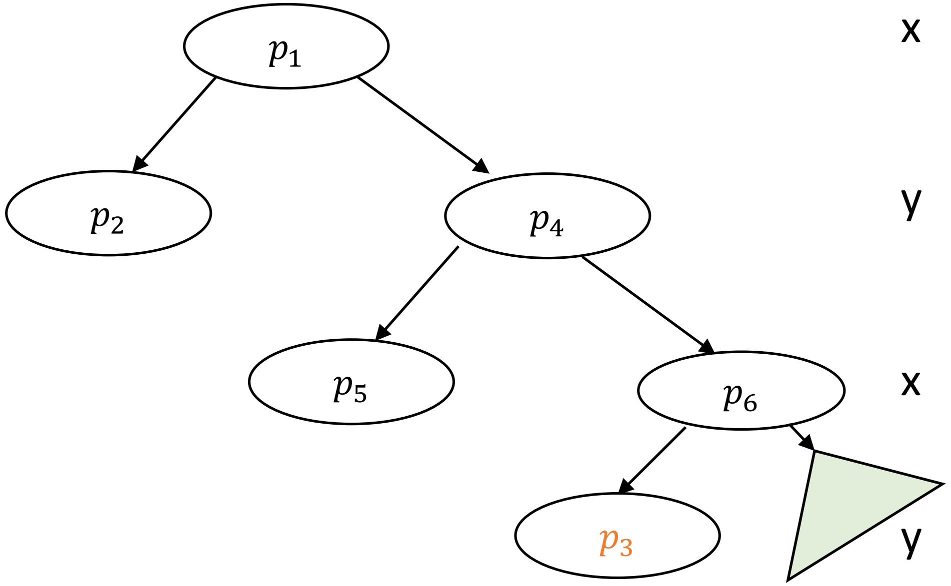




Nearest neighbor example

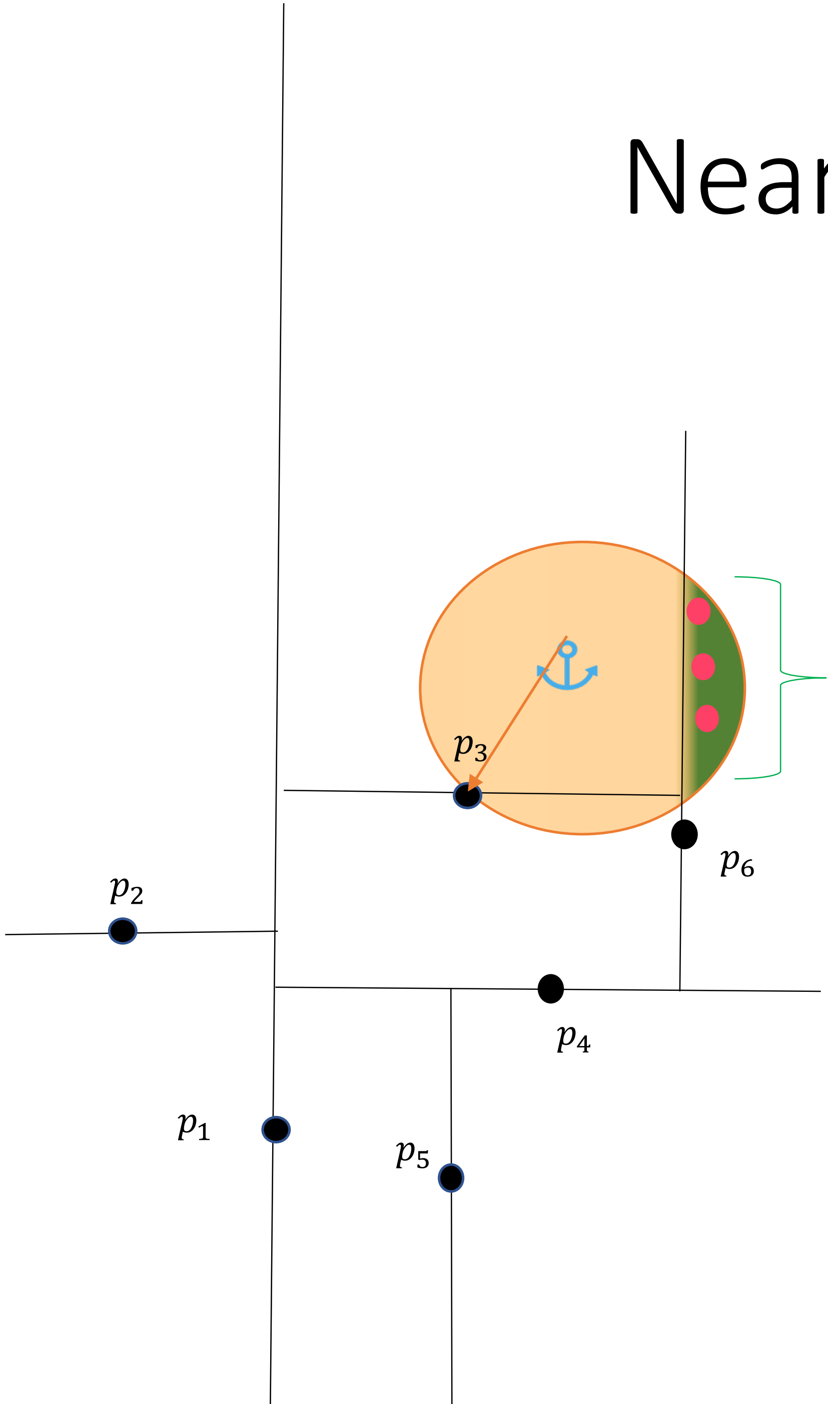




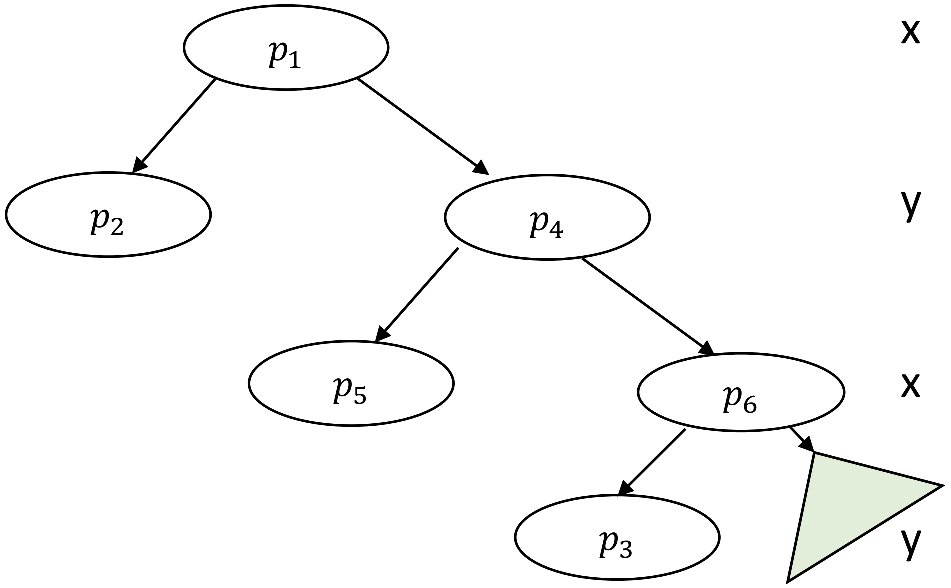


It ***does*** make sense for us to look on the right subtree of p_6, because of the green intersection above! We currently **can't be certain** that there **aren't** any nodes in the green intersection that don't improve upon p_3 as our choice of nearest neighbor!

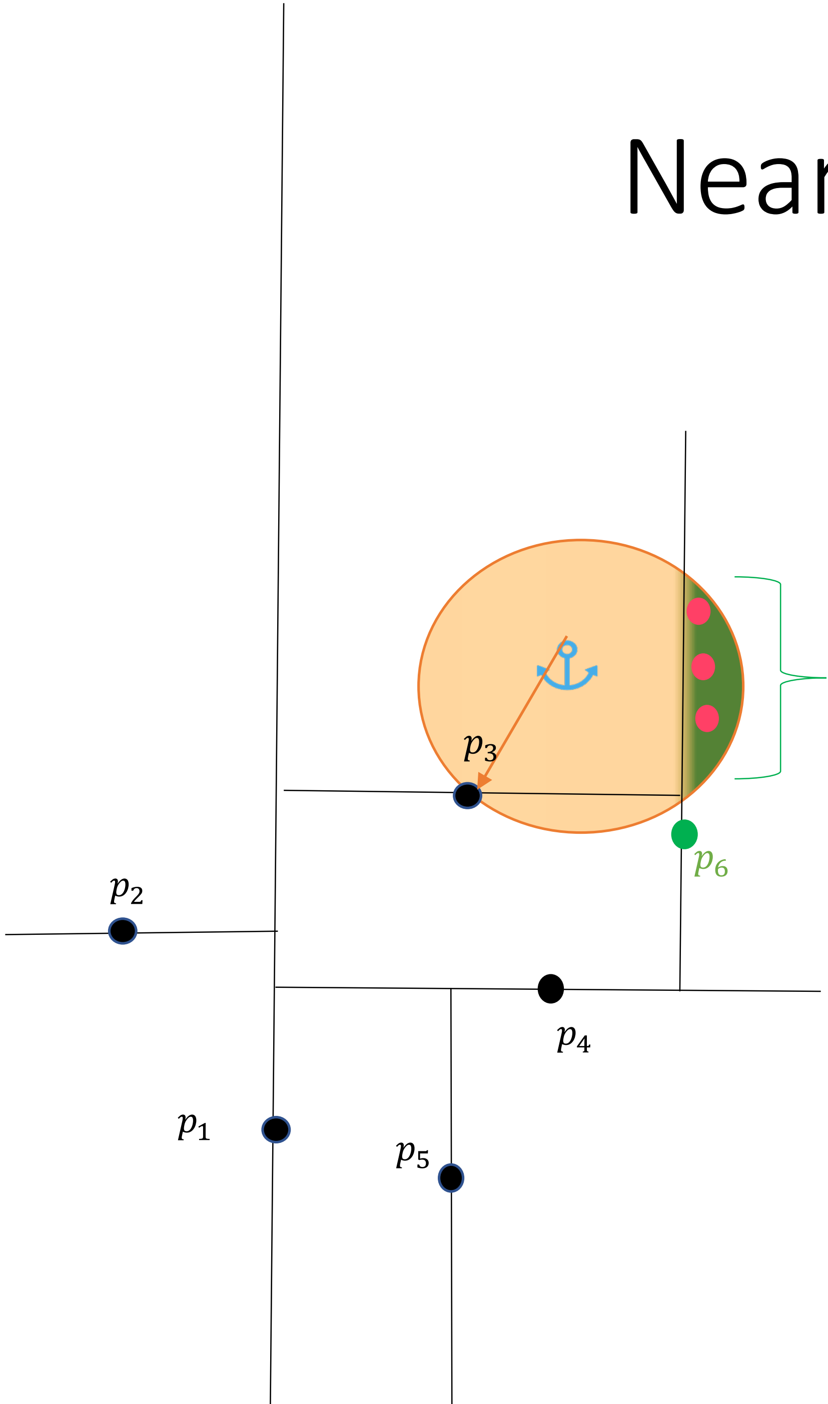
Nearest neighbor example



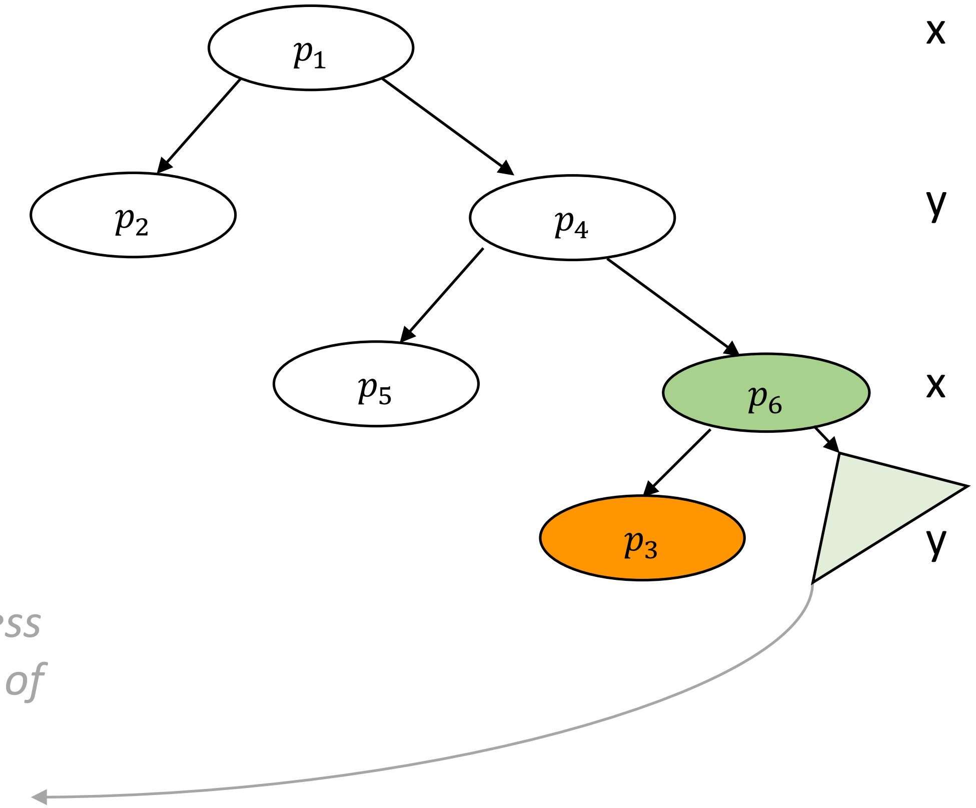
Non-empty intersection *might* contain **many neighbors closer to anchor than** p_3!



Nearest neighbor example

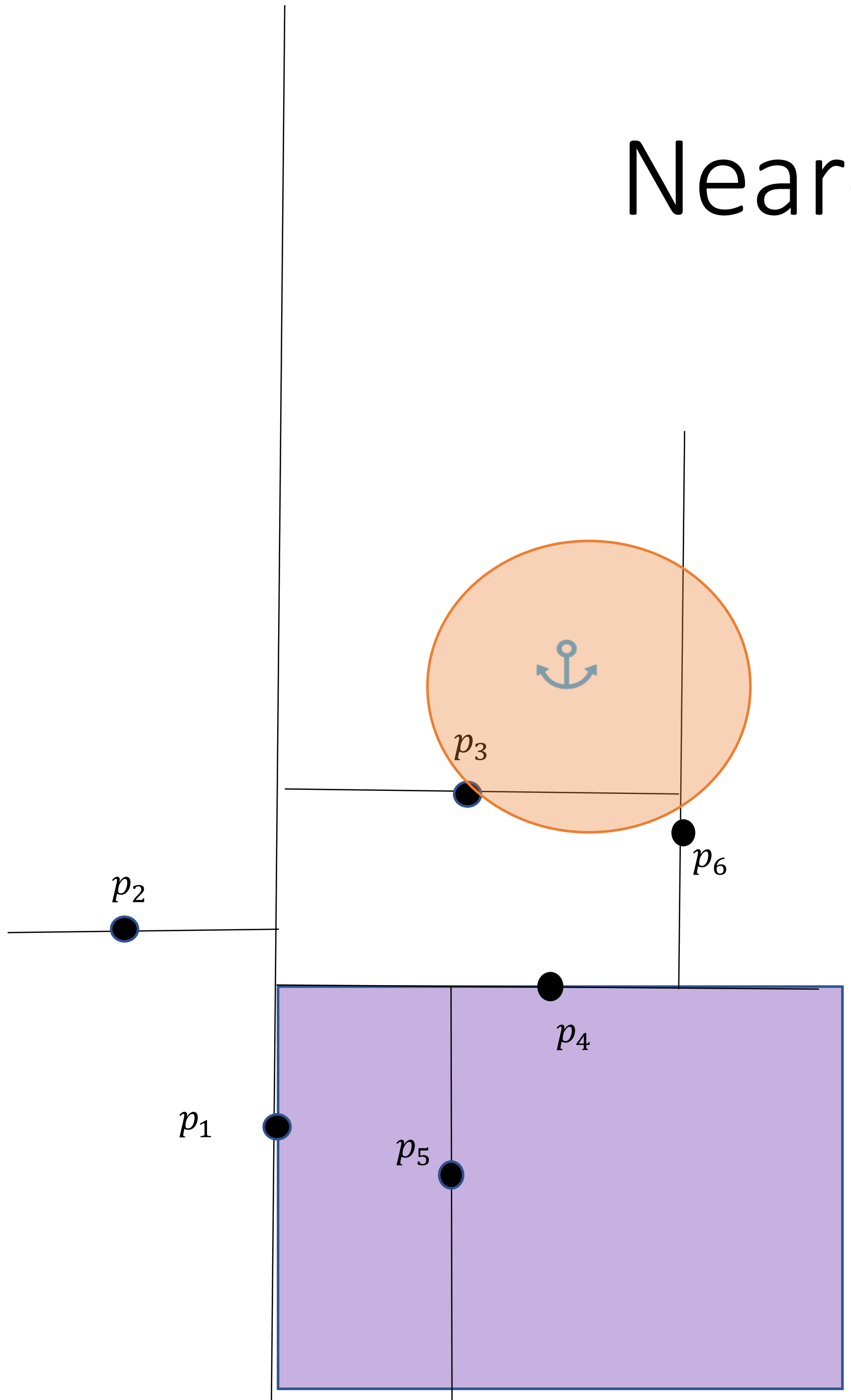


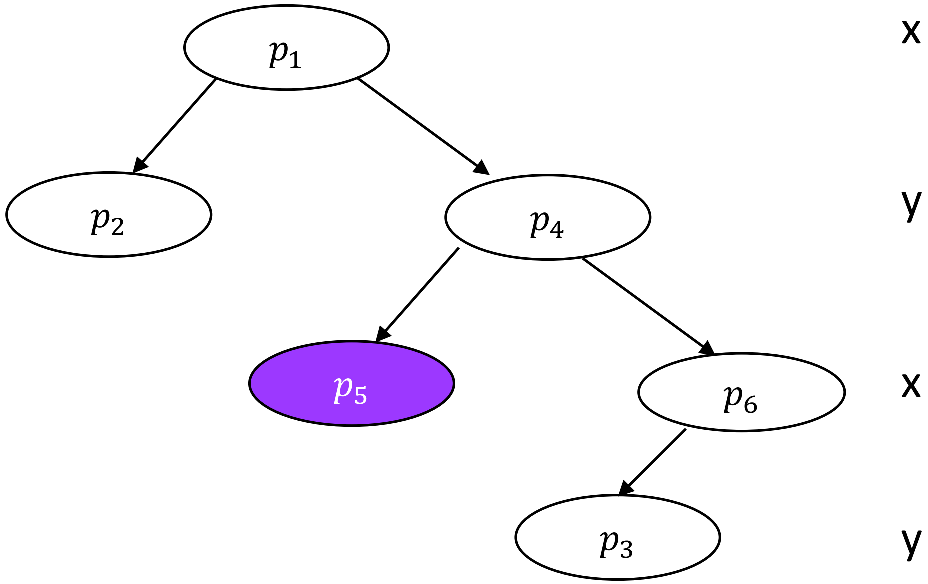
Non-empty intersection *might* contain **many neighbors closer to anchor than** p_3**!**



(Of course, in this case, no progress can be made since the right child of p_6 *is* **`null`***...)*

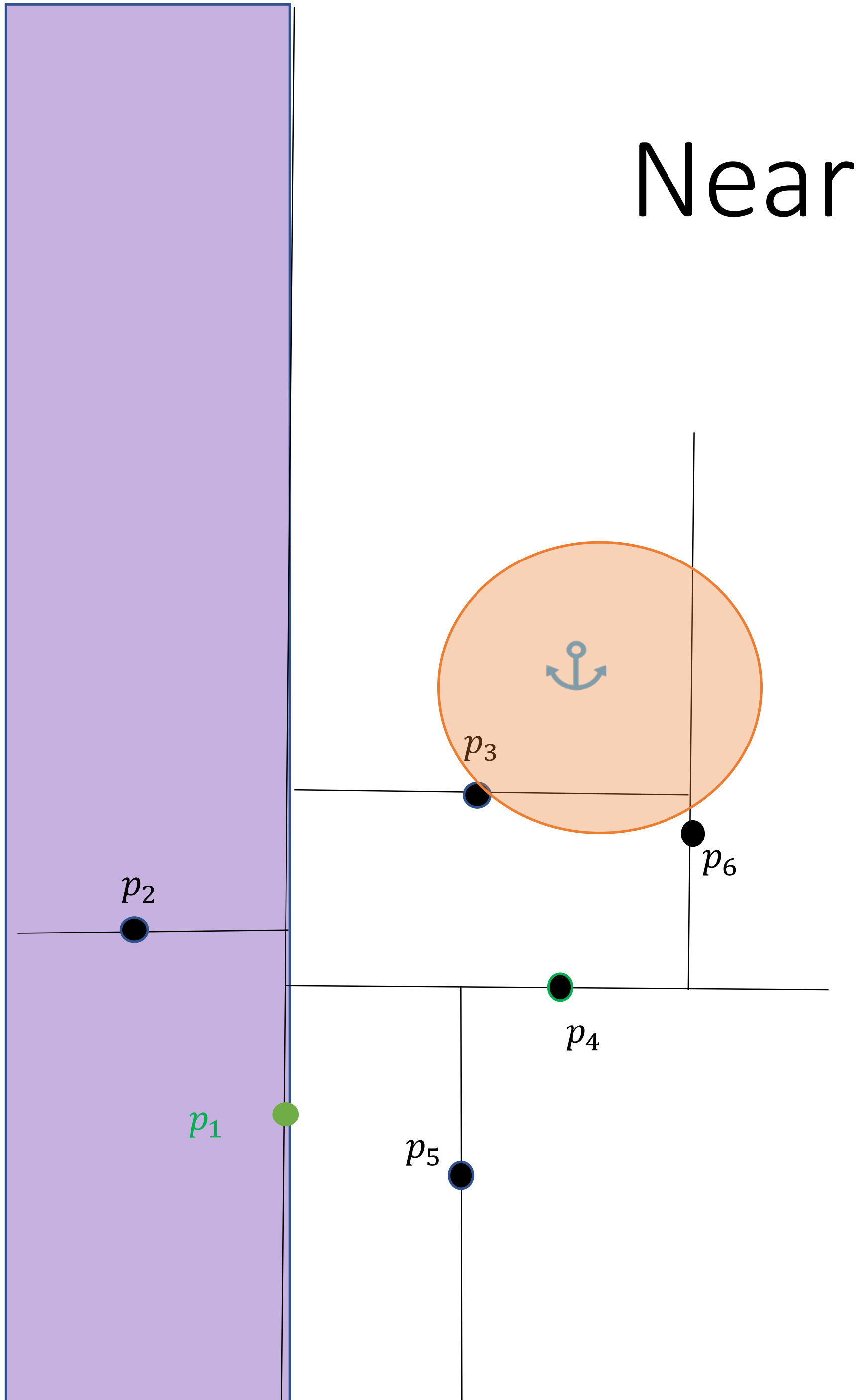
Nearest neighbor example

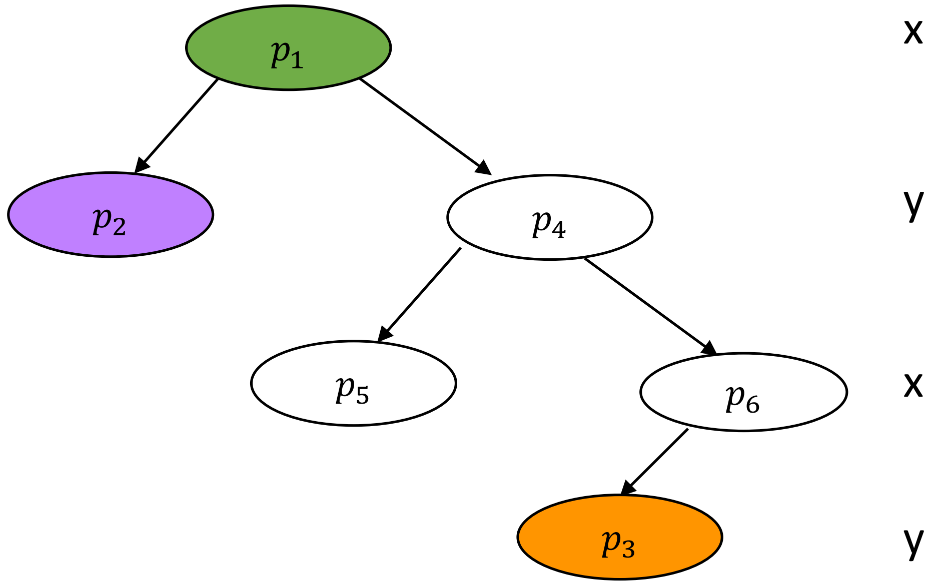




When moving back up to p_4, however, **it does not make any sense to recurse** to p_4's left subtree, since the **candidate circle does not intersect that half-plane…**

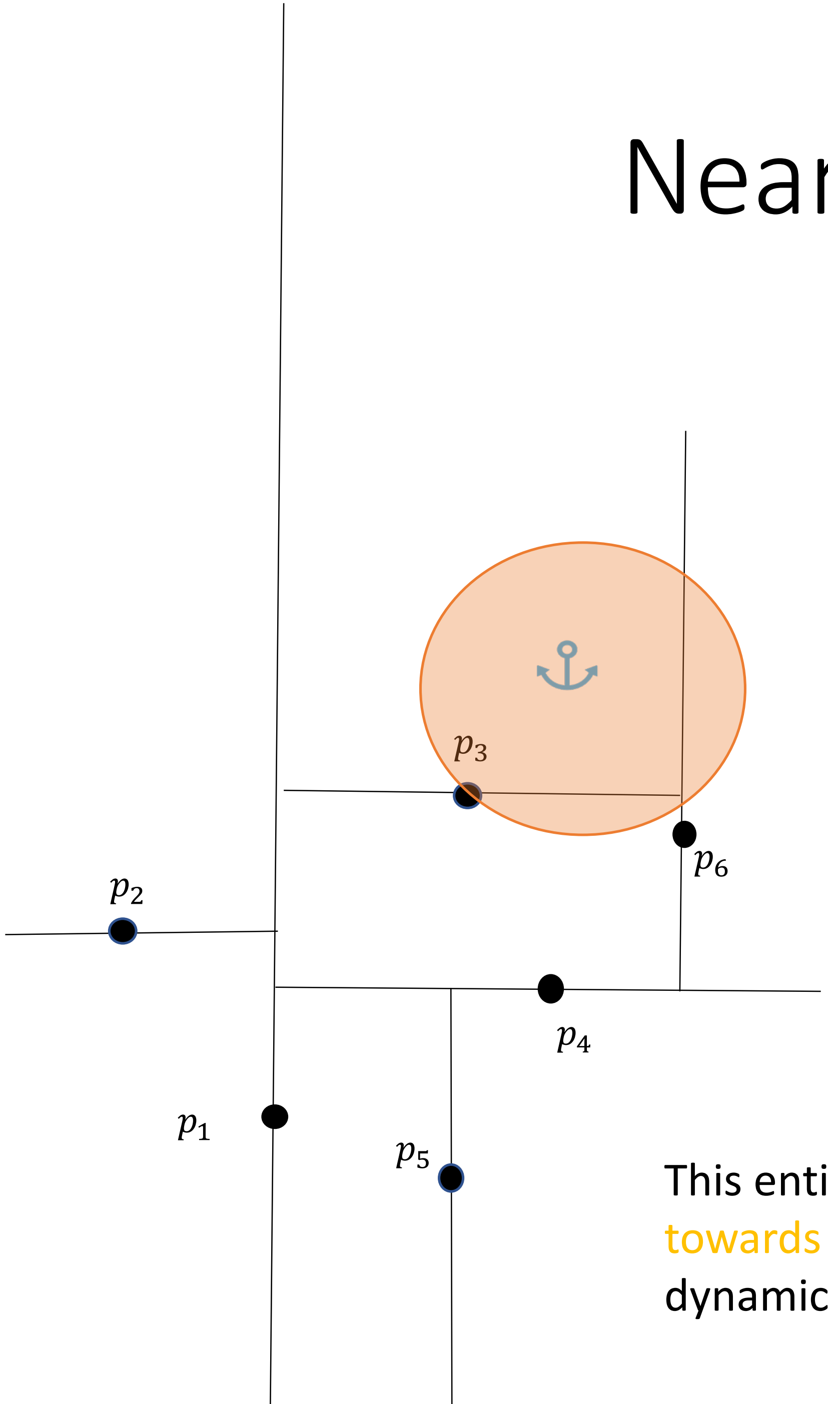
Nearest neighbor example

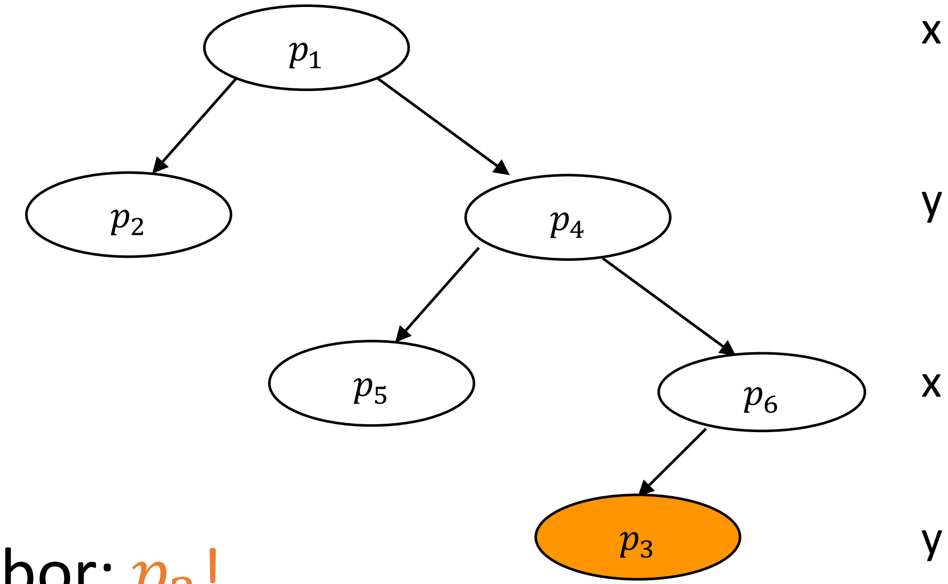




Similarly, it would be **useless** to reach into the left subtree of p_1**…**

Nearest neighbor example





Nearest neighbor: p_3!

This entire process is an example of a ***branch-and-bound*** **technique**: We only branch towards solutions that are bounded above by the currently best-cost solution, dynamically improving the bound.

m-nearest neighbors

- **Important note:** This is the only time in your lives where you will see m being used to describe a "$m$any"-nearest neighbors query.
- EVERYBODY ELSE IN THE WORLD uses k. Everybody.
  - We choose m instead of k to not confuse you with the dimensionality of the KD-Tree.

m-nearest neighbors

- **Important note:** This is the only time in your lives where you will see m being used to describe a "$m$any"-nearest neighbors query.
- EVERYBODY ELSE IN THE WORLD uses k. Everybody.
  - We choose m instead of k to not confuse you with the dimensionality of the KD-Tree.
- To be able to answer this kind of query correctly, we need to be able to maintain some kind of sorted collection of **exactly** m points.

m-nearest neighbors

- **<u>Important note:</u>** This is the only time in your lives where you will see m being used to describe a "$m$any"-nearest neighbors query.
- EVERYBODY ELSE IN THE WORLD uses k. Everybody.
  - We choose m instead of k to not confuse you with the dimensionality of the KD-Tree.
- To be able to answer this kind of query correctly, we need to be able to maintain some kind of sorted collection of **exactly** m points.
- What would you use?

Linked List

A balanced binary tree

A stack

Something else (what?)

m-nearest neighbors

- **Important note:** This is the only time in your lives where you will see m being used to describe a "$m$any"-nearest neighbors query.
- EVERYBODY ELSE IN THE WORLD uses k. Everybody.
  - We choose m instead of k to not confuse you with the dimensionality of the KD-Tree.
- To be able to answer this kind of query correctly, we need to be able to maintain some kind of sorted collection of **exactly** m points.
- What would you use?

A priority queue!

Linked List

A balanced binary tree

A stack

Something else (what?)

m-nearest neighbors

- **Important note:** This is the only time in your lives where you will see m being used to describe a "$m$any"-nearest neighbors query.
- EVERYBODY ELSE IN THE WORLD uses k. Everybody.
    - We choose m instead of k to not confuse you with the dimensionality of the KD-Tree.
- To be able to answer this kind of query correctly, we need to be able to maintain some kind of sorted collection of **exactly** m points.
- What would you use?

And not just **any** priority queue....

Linked List

A balanced binary tree

A stack

Something else (what?)

Bounded priority queues

- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:

Bounded priority queues

- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.

Bounded priority queues

- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.
  2. If the BPQ contains **less than** m elements and a new one arrives, we insert it in its appropriate position.

Bounded priority queues

- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.
  2. If the BPQ contains **less than** m elements and a new one arrives, we insert it in its appropriate position.
  3. If a new element to be inserted encounters a BPQ **with** m **elements** in it…

Bounded priority queues

- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.
  2. If the BPQ contains **less than** m elements and a new one arrives, we insert it in its appropriate position.
  3. If a new element to be inserted encounters a BPQ **with** m **elements** in it…
     a) If the element's priority would place it **after** the m^{th} element (so, beyond the queue's end), we will not insert it.

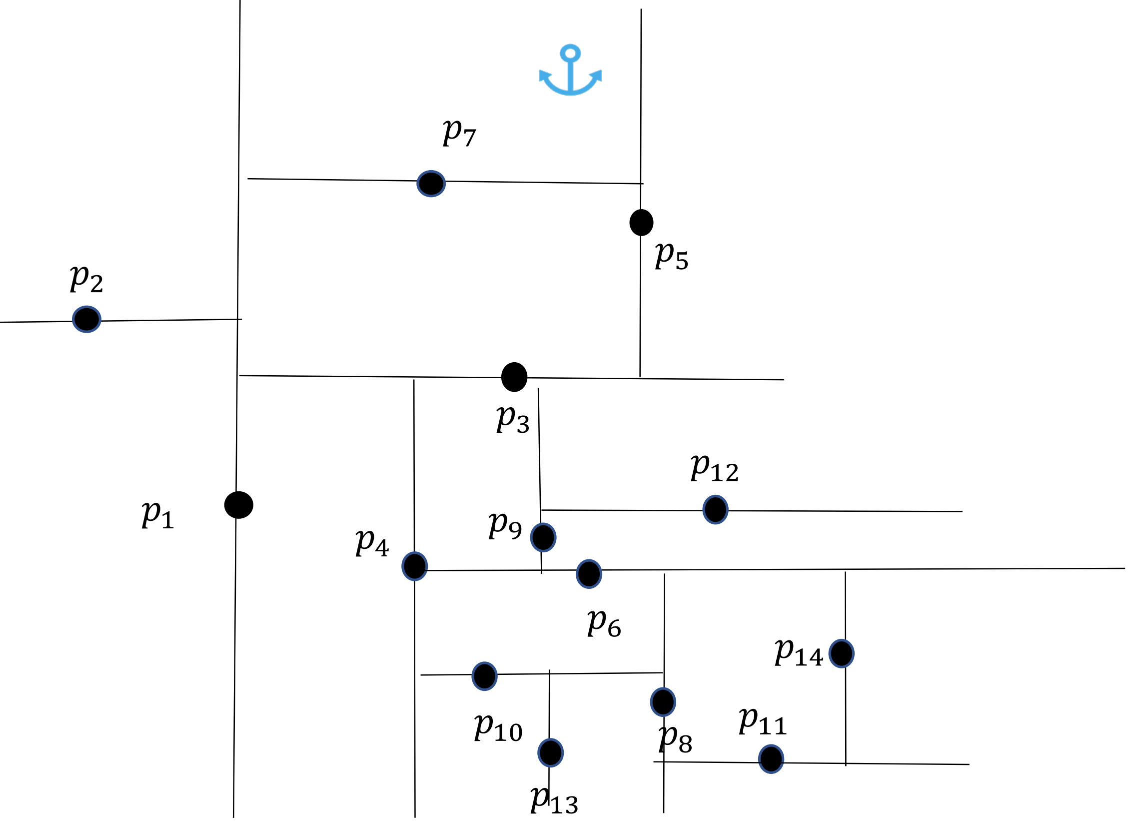
Bounded priority queues

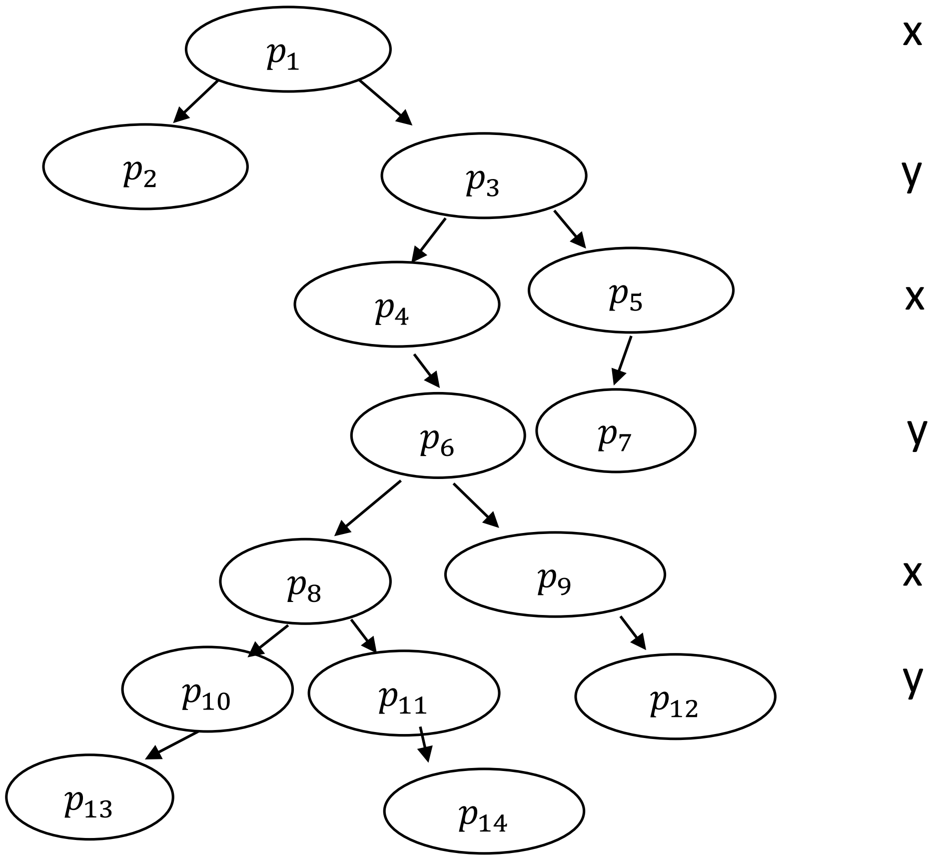
- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.
  2. If the BPQ contains **less than** m elements and a new one arrives, we insert it in its appropriate position.
  3. If a new element to be inserted encounters a BPQ **with** m **elements** in it…
     a) If the element's priority would place it **after** the m^{th} element (so, beyond the queue's end), we will not insert it.
     b) If the element's priority **is exactly the same** as the the m^{th} elements as that of the last element's, then **we also do not insert it** (remember: Priority Queues break ties with **FIFO order**)

Bounded priority queues

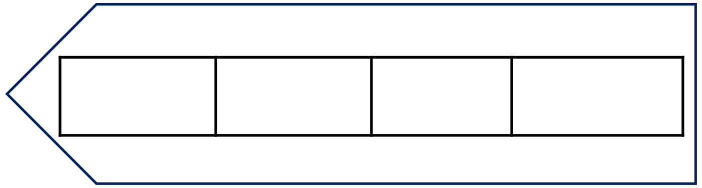
- We assume **any** implementation of a Priority Queue.
  - *(But really, you should probably use binary heaps for these kinds of problems).*
- A **B**ounded **P**riority **Q**ueue (hereafter **BPQ**) behaves like any PQ, except for the following details:
  1. At any point in time, it cannot hold more than m elements.
  2. If the BPQ contains **less than** m elements and a new one arrives, we insert it in its appropriate position.
  3. If a new element to be inserted encounters a BPQ **with** m **elements** in it…
     a) If the element's priority would place it **after** the m^{th} element (so, beyond the queue's end), we will not insert it.
     b) If the element's priority **is exactly the same** as the the m^{th} elements as that of the last element's, then **we also do not insert it** (remember: Priority Queues break ties with **FIFO order**)
     c) If the element's priority would place it **before** the m^{th} element, we insert it at the appropriate position and throw away the last element.

4-NN example

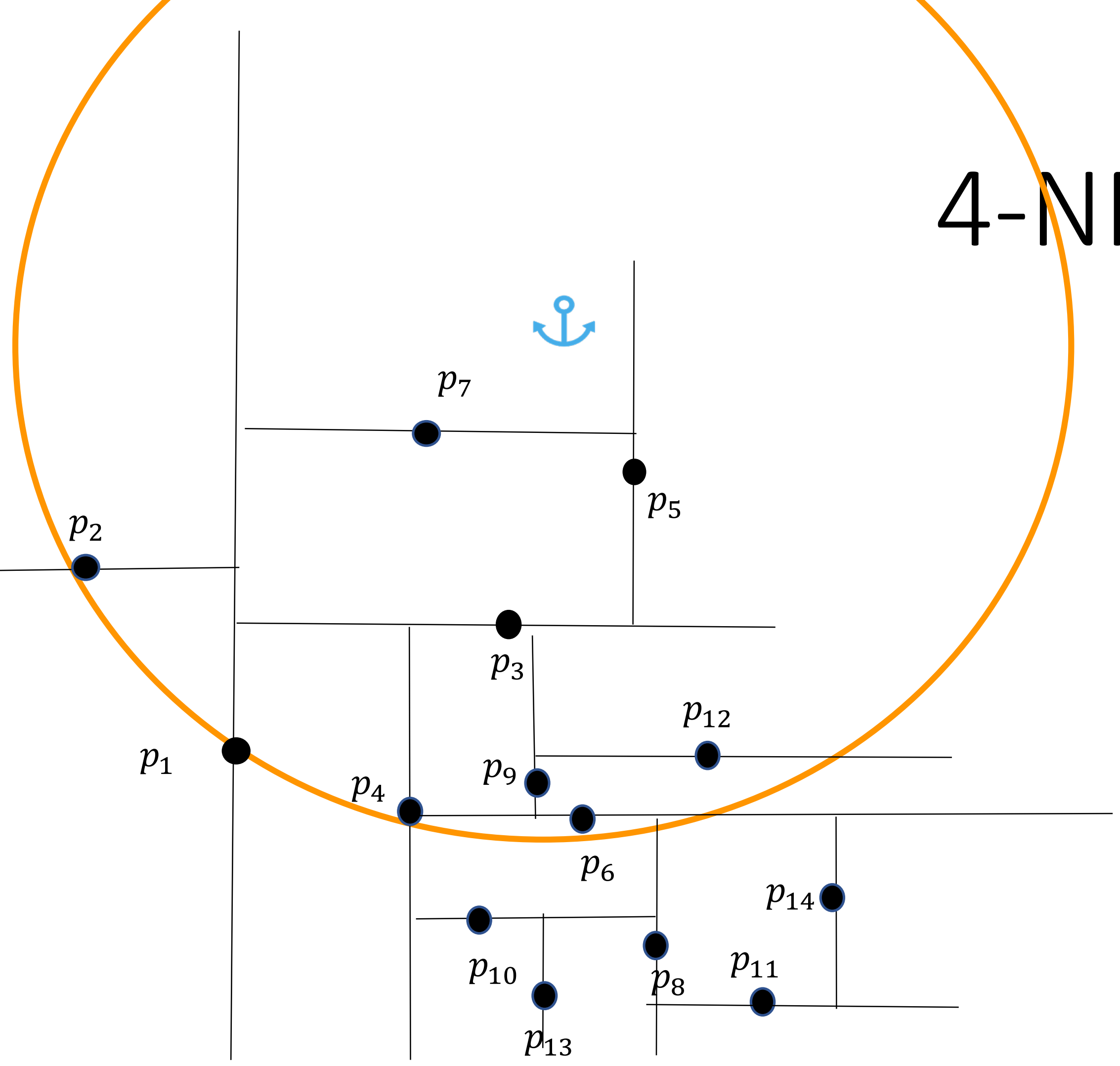


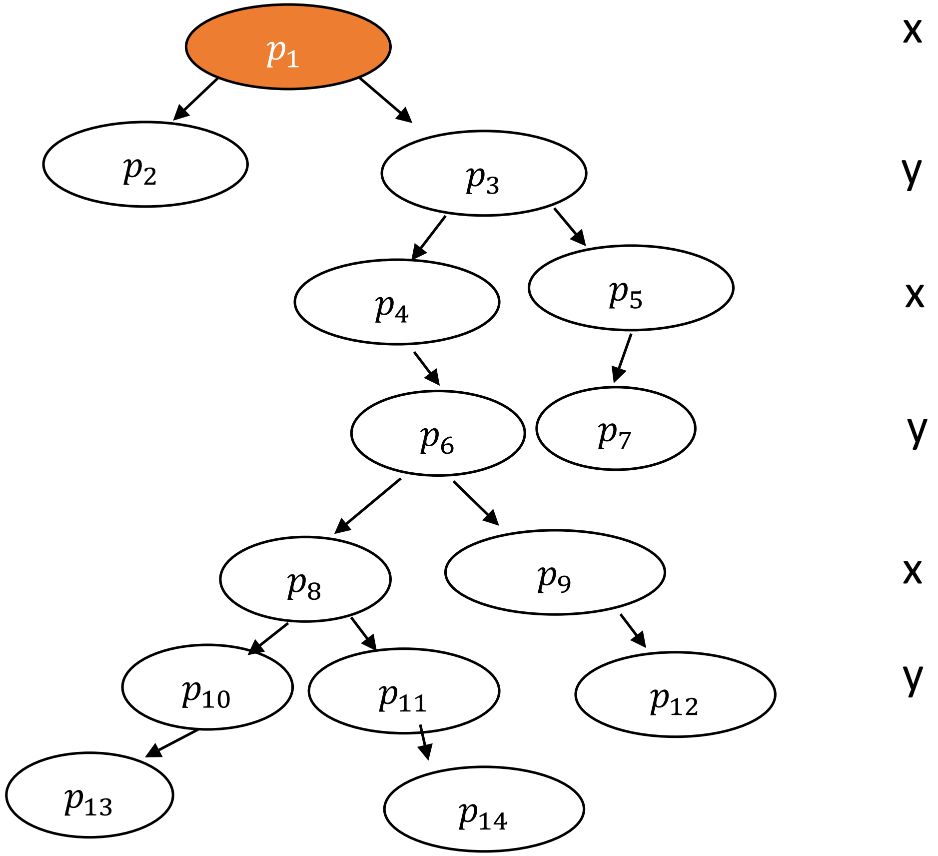


BPQ

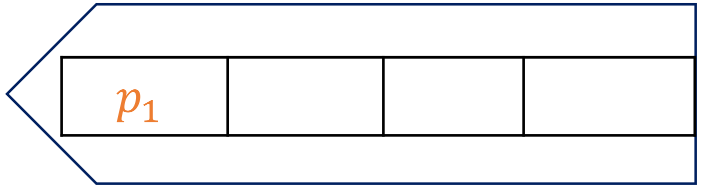


4-NN example

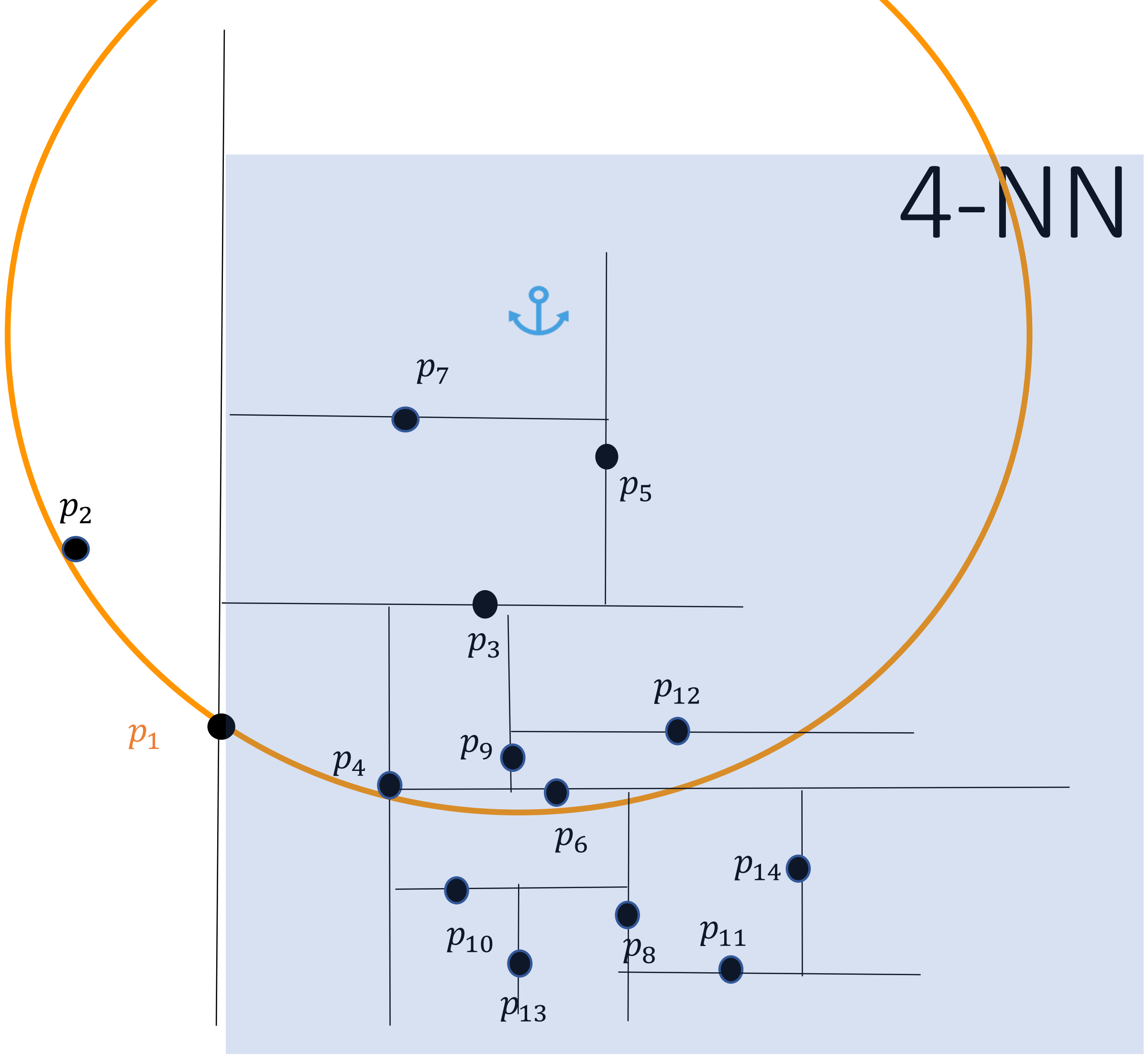


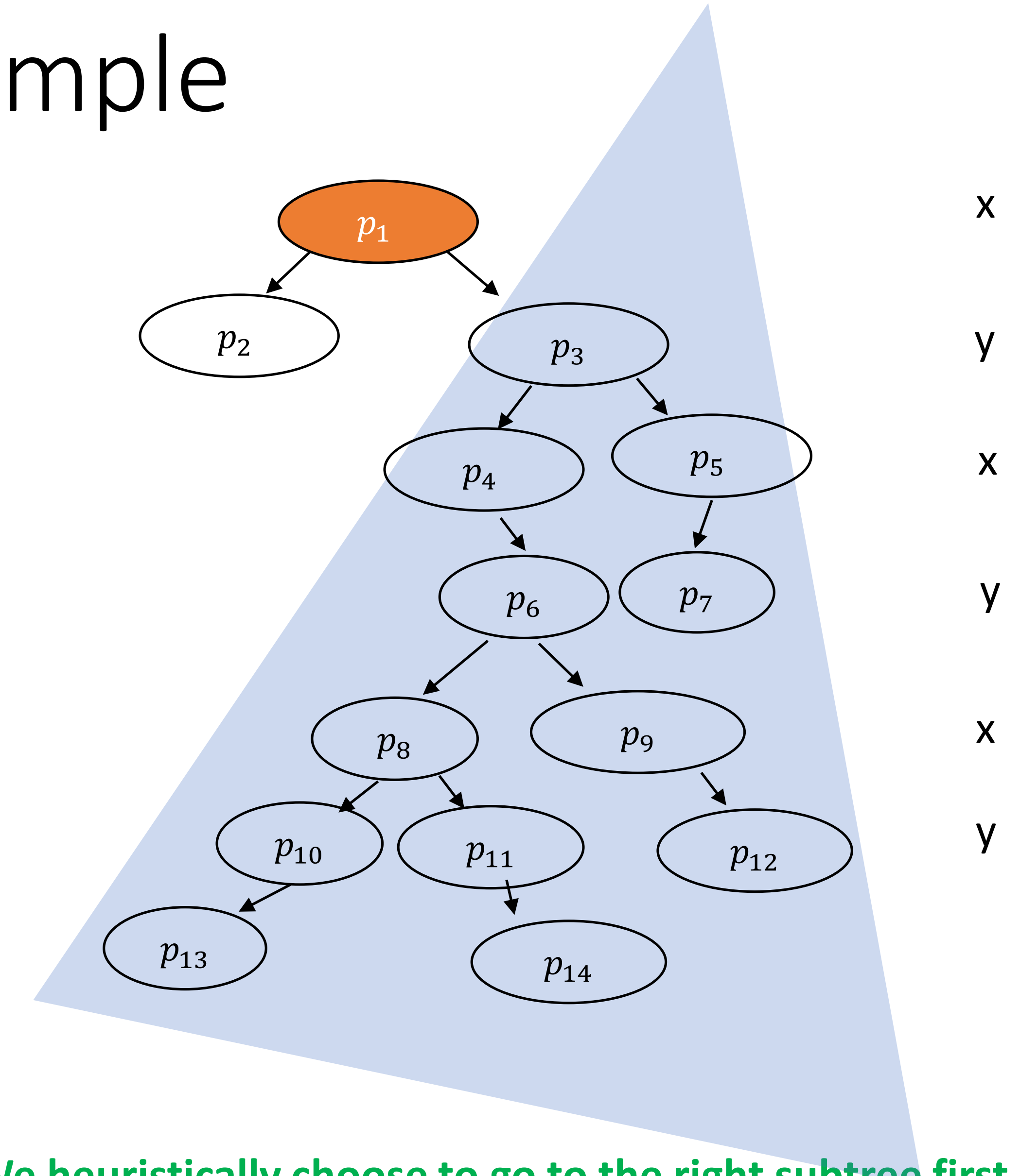


BPQ

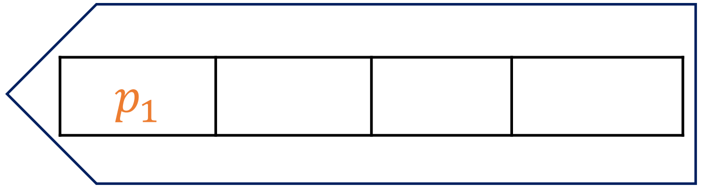


4-NN example



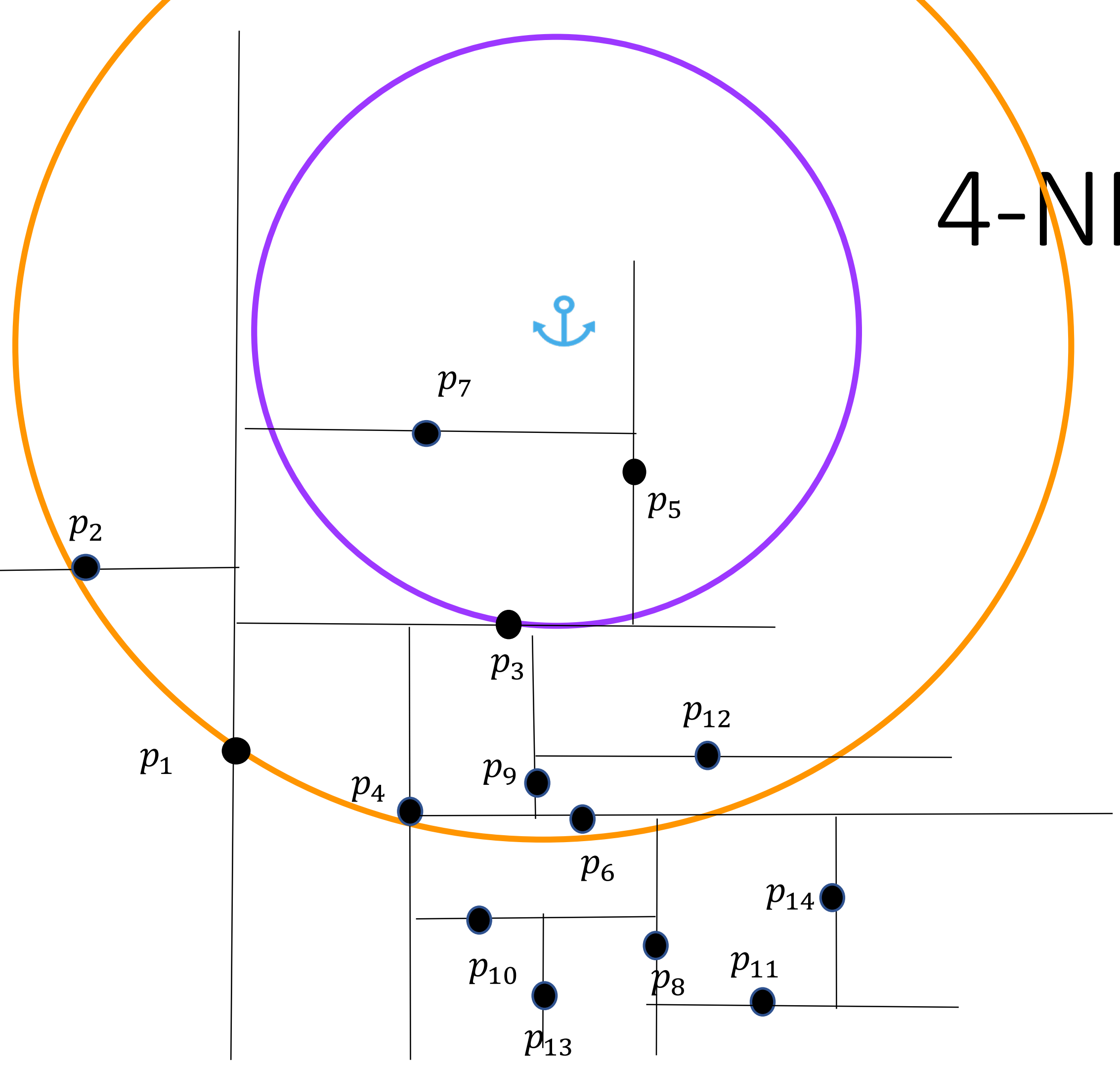


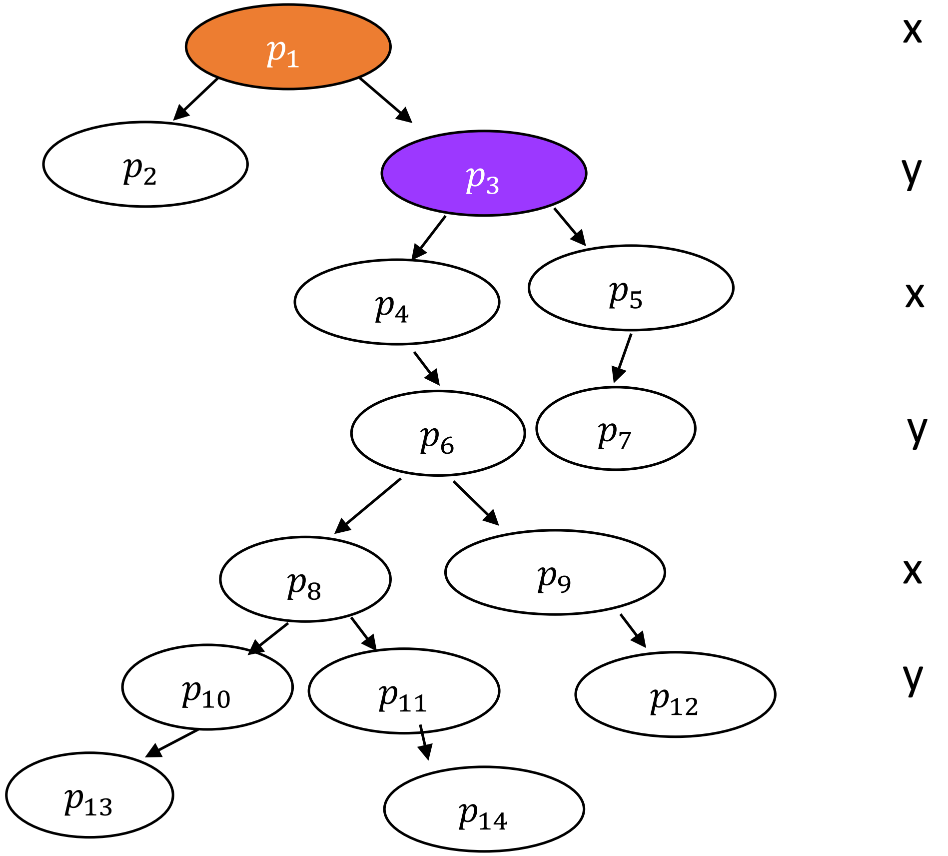
BPQ



We heuristically choose to go to the right subtree first since the anchor's x is on the right of our own!

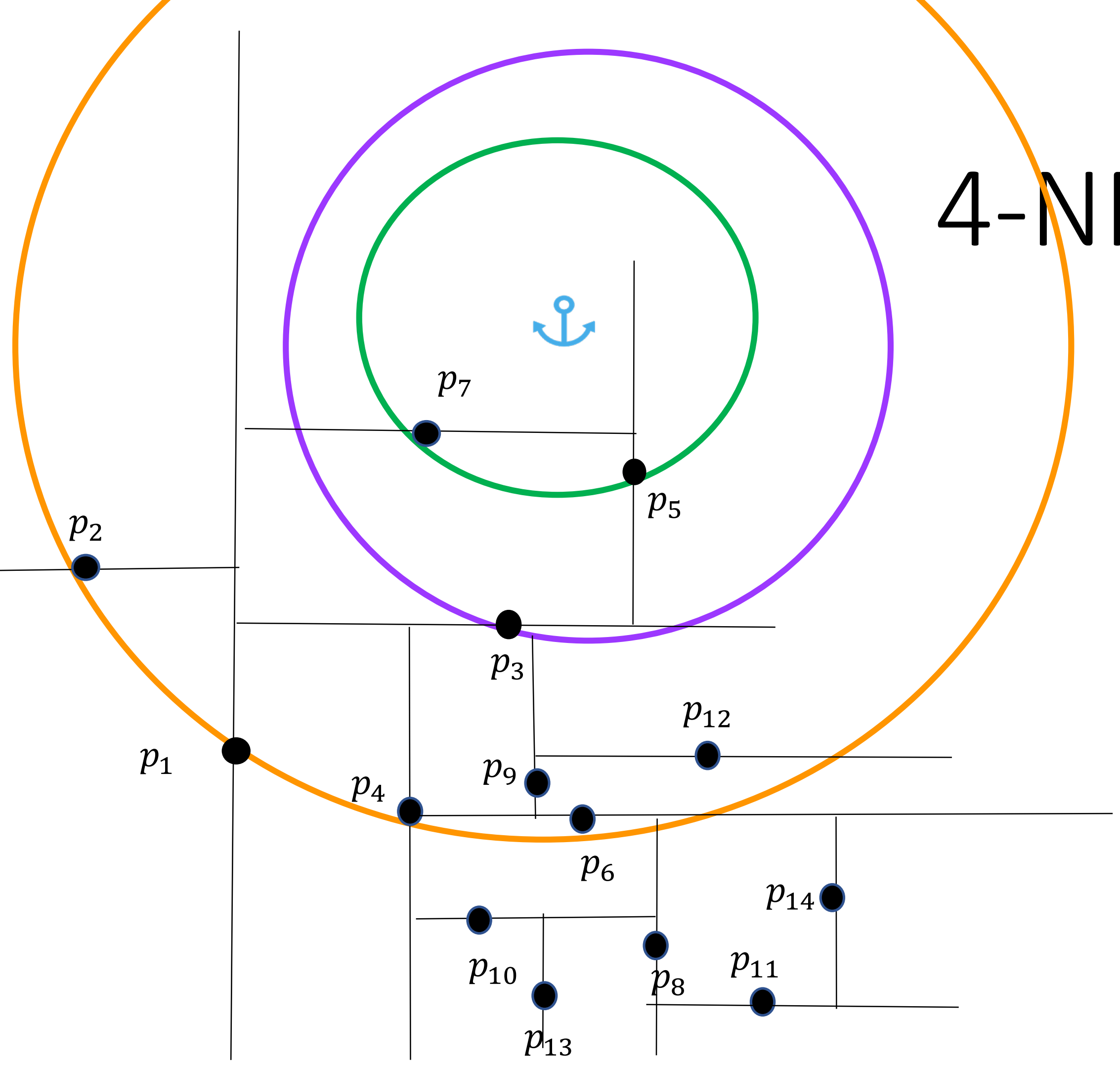
4-NN example

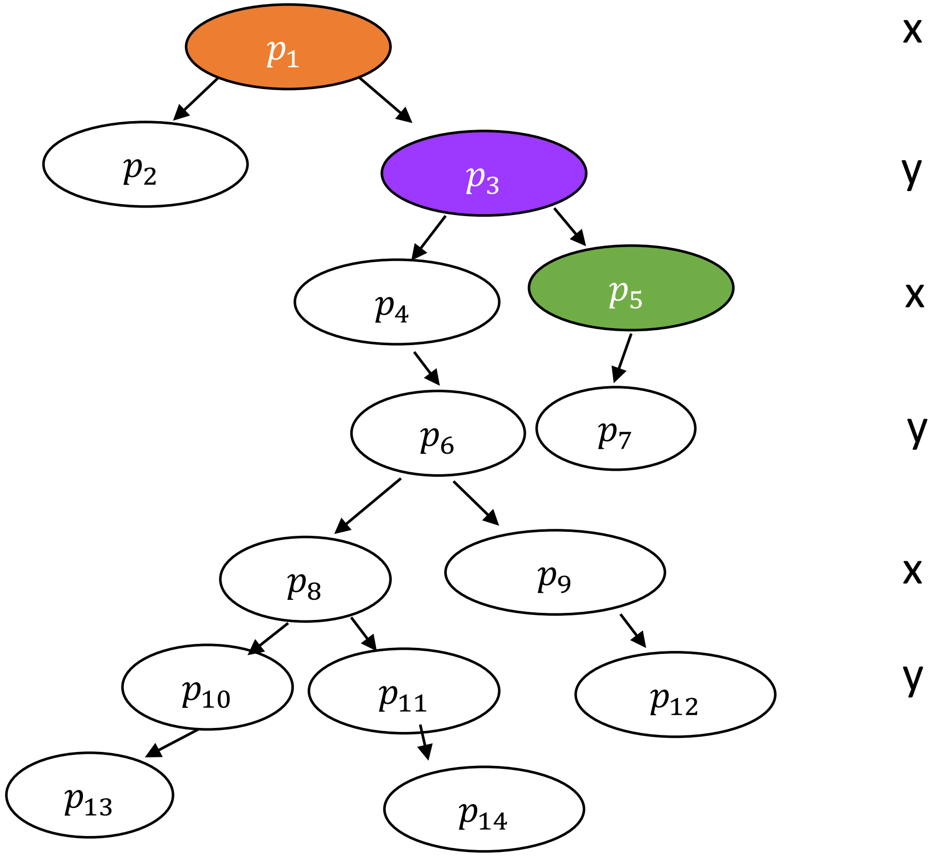




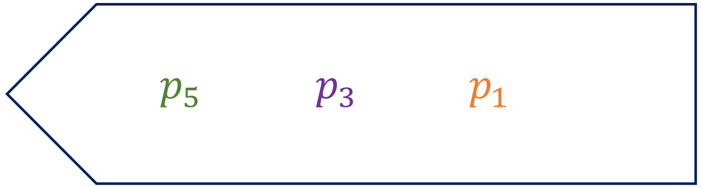
BPQ

4-NN example

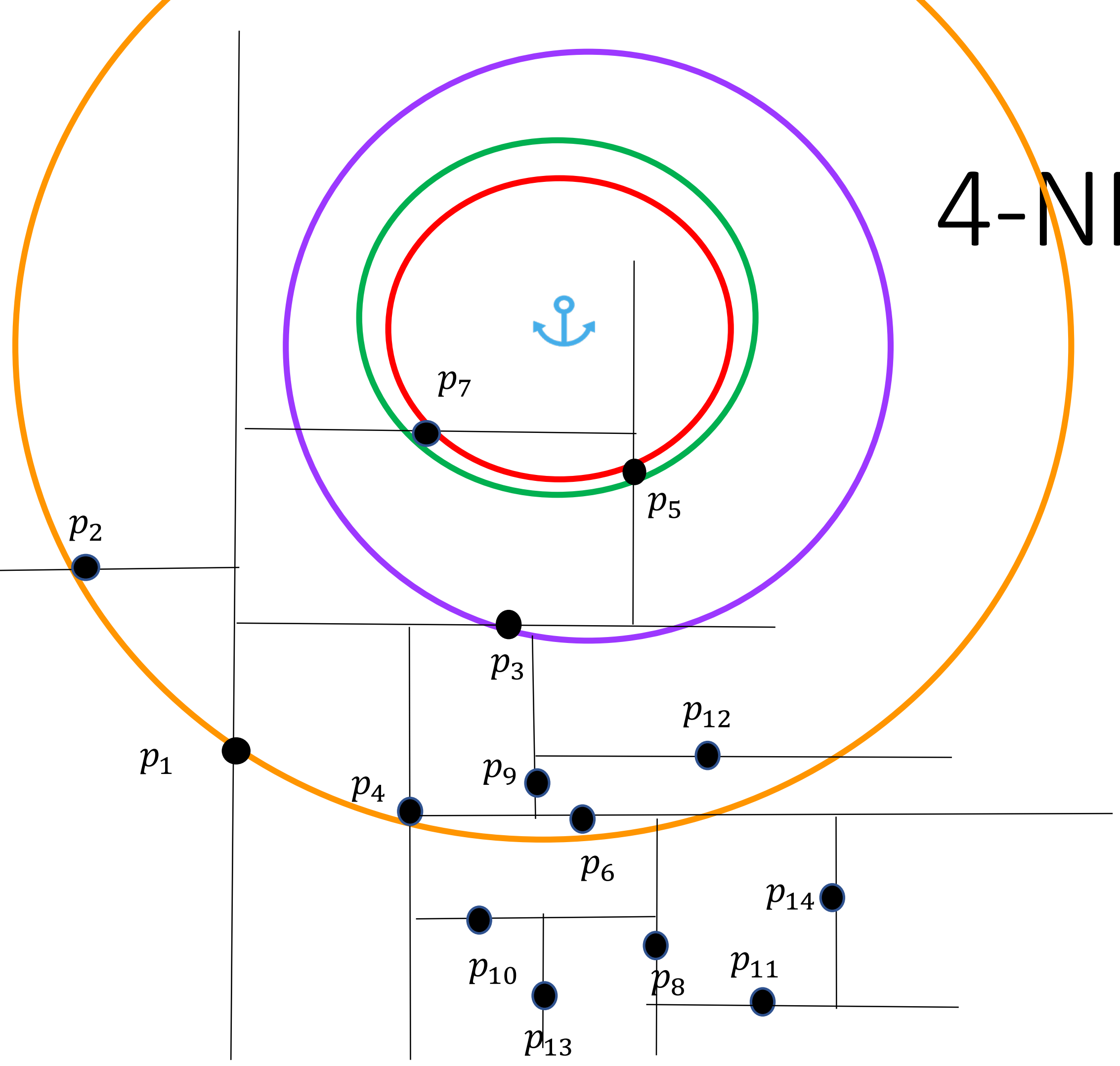


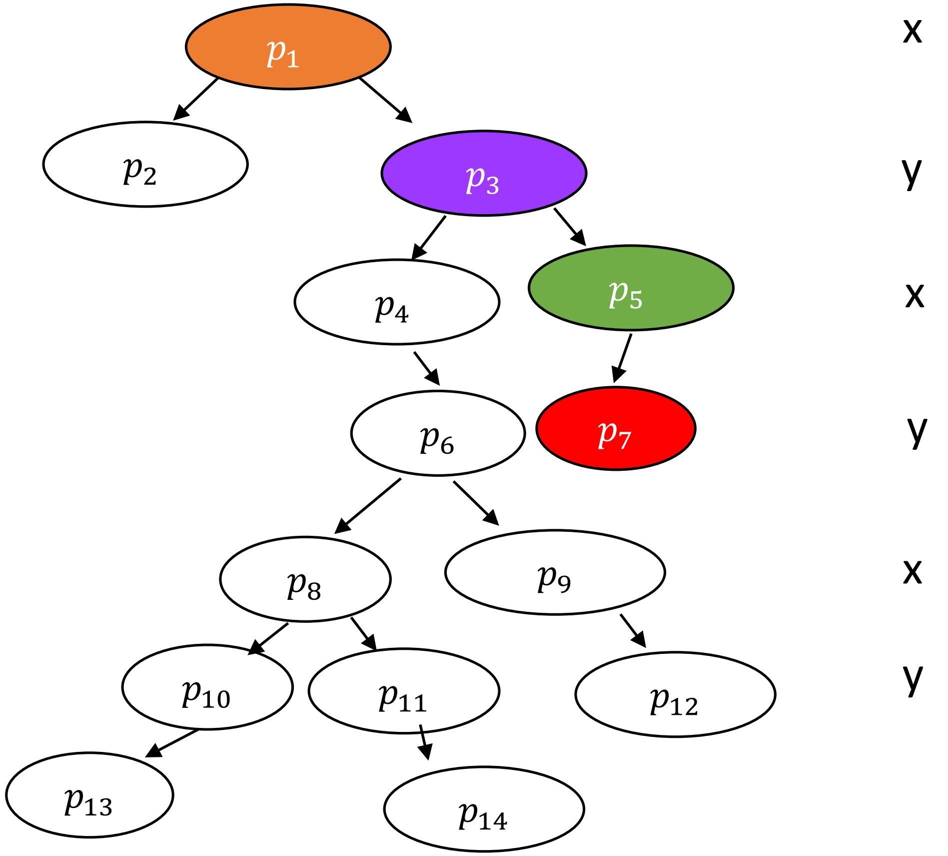


BPQ

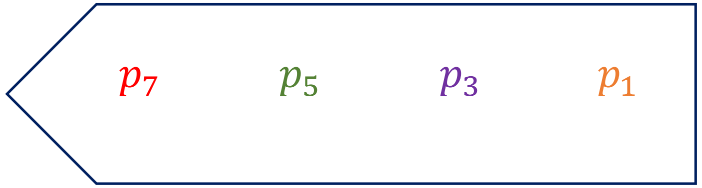


4-NN example

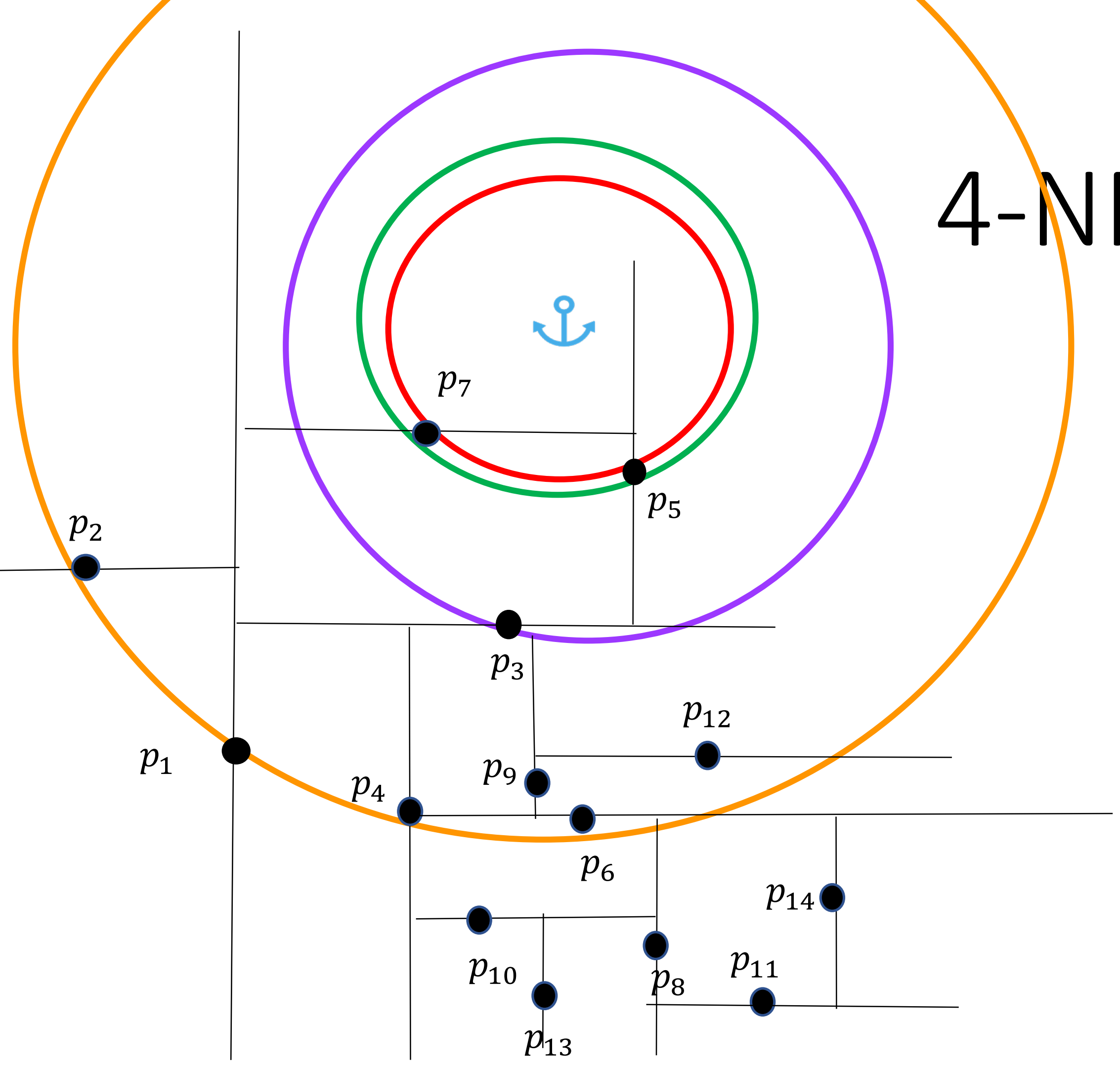


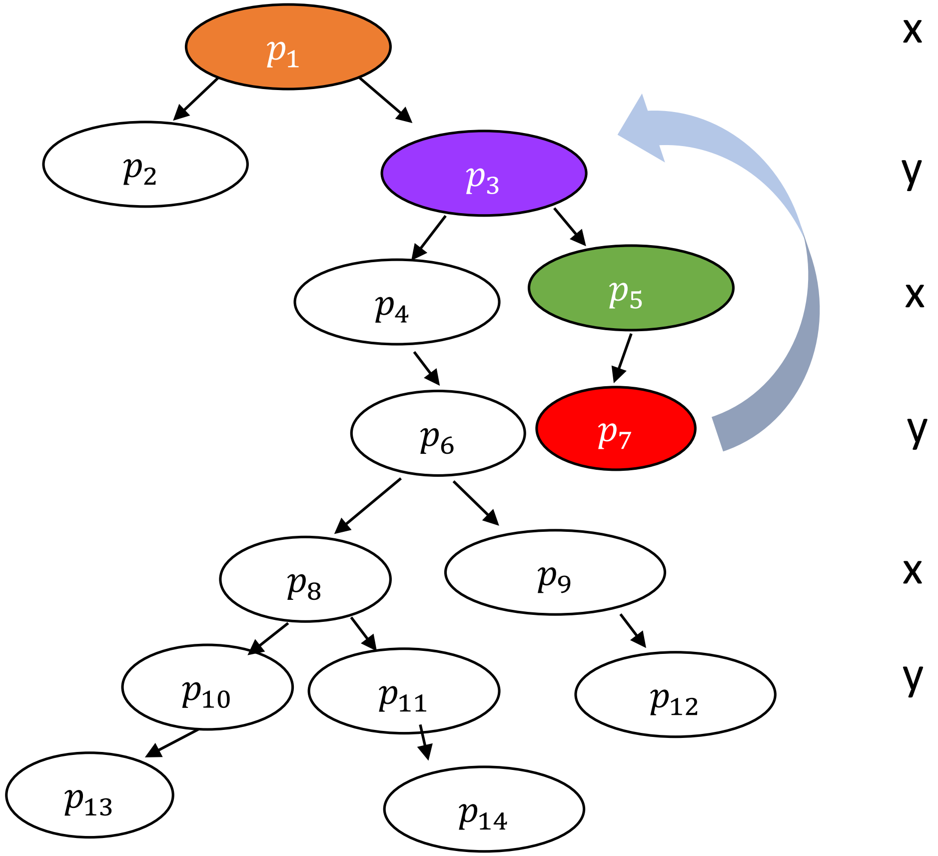


BPQ

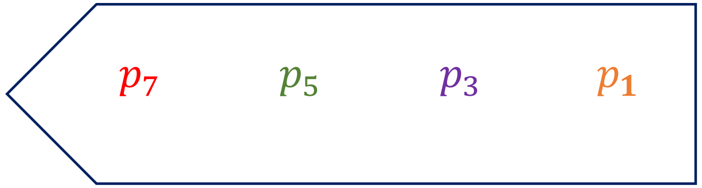


4-NN example

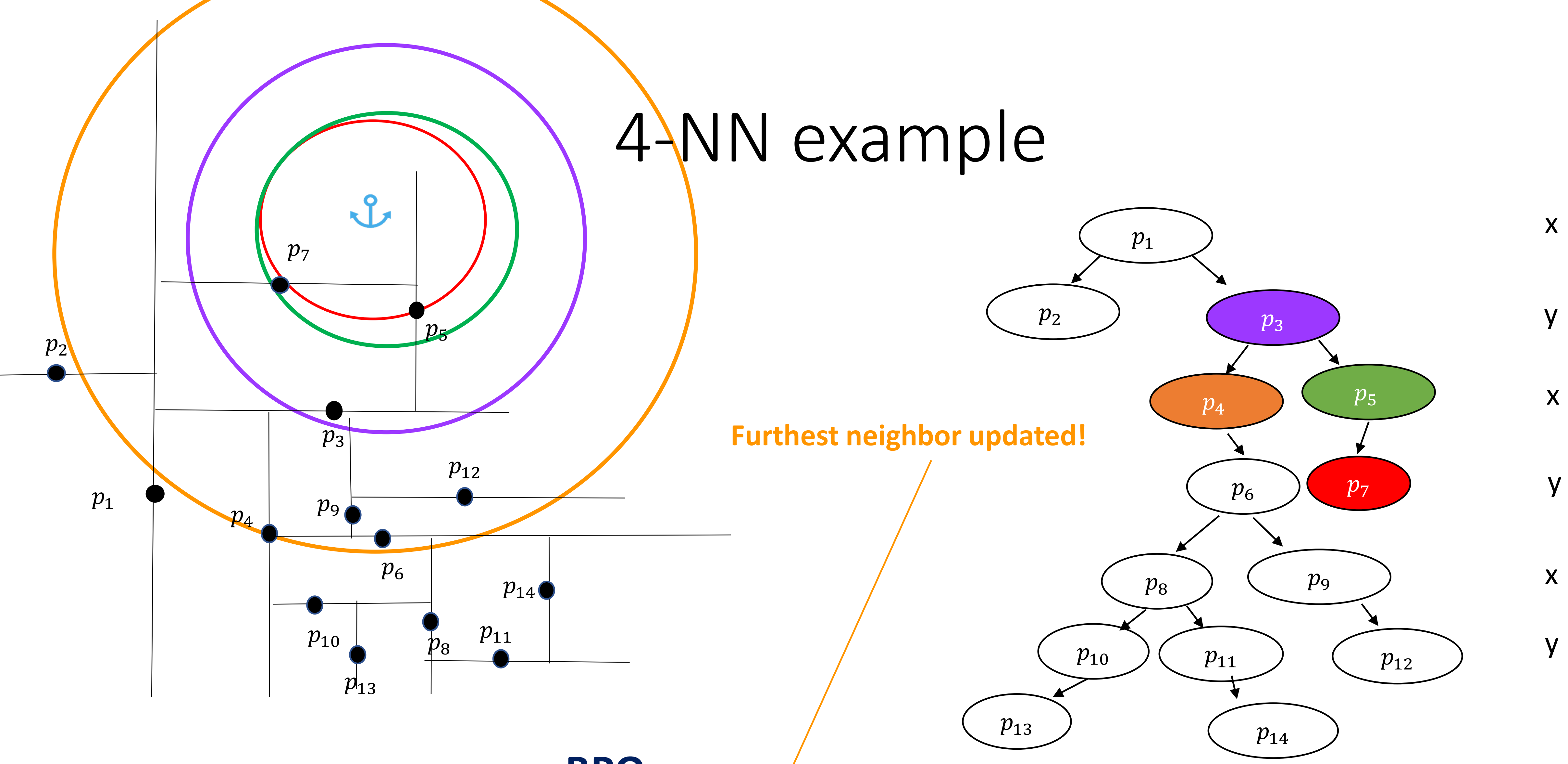




BPQ



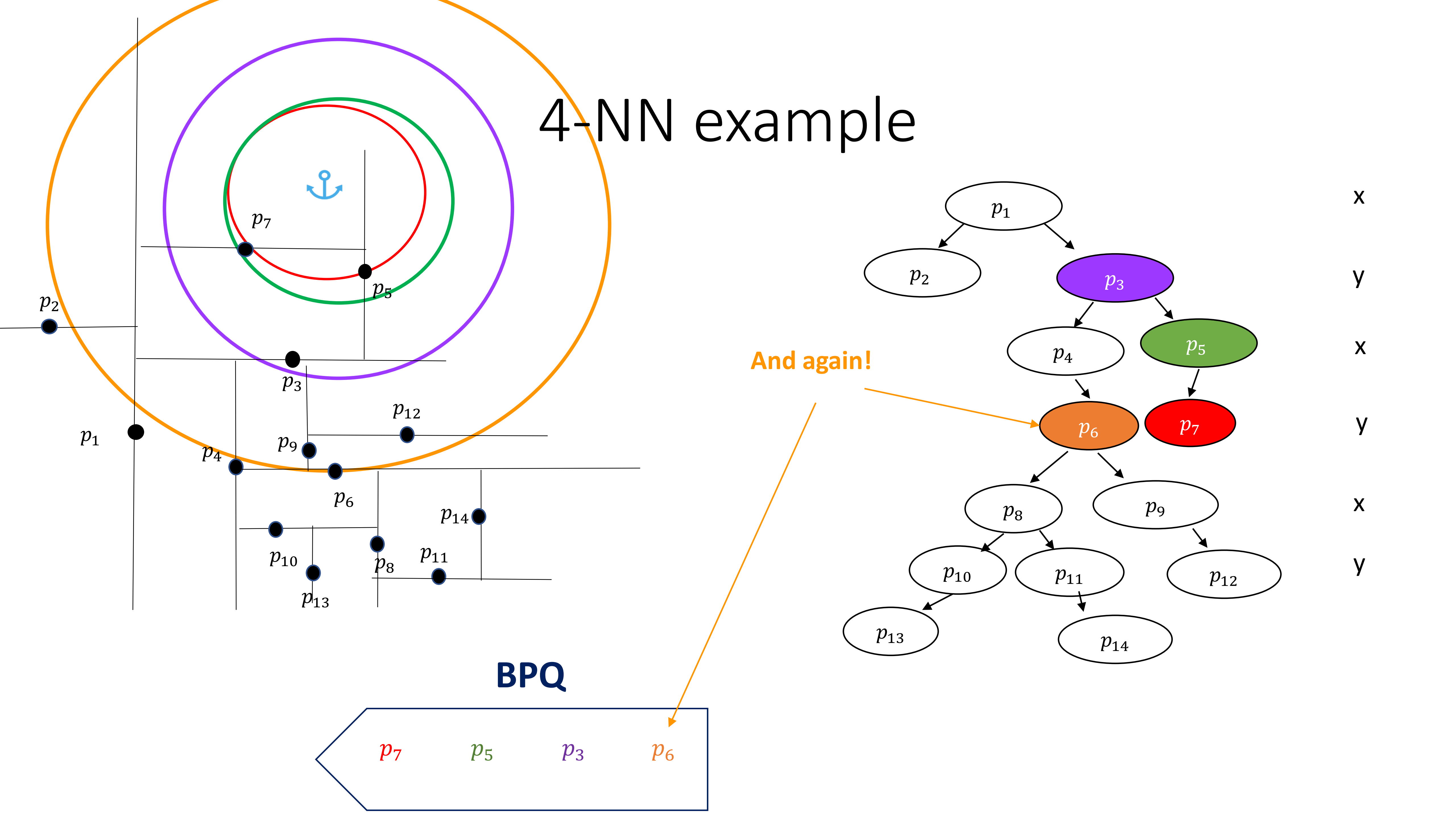
4-NN example



BPQ

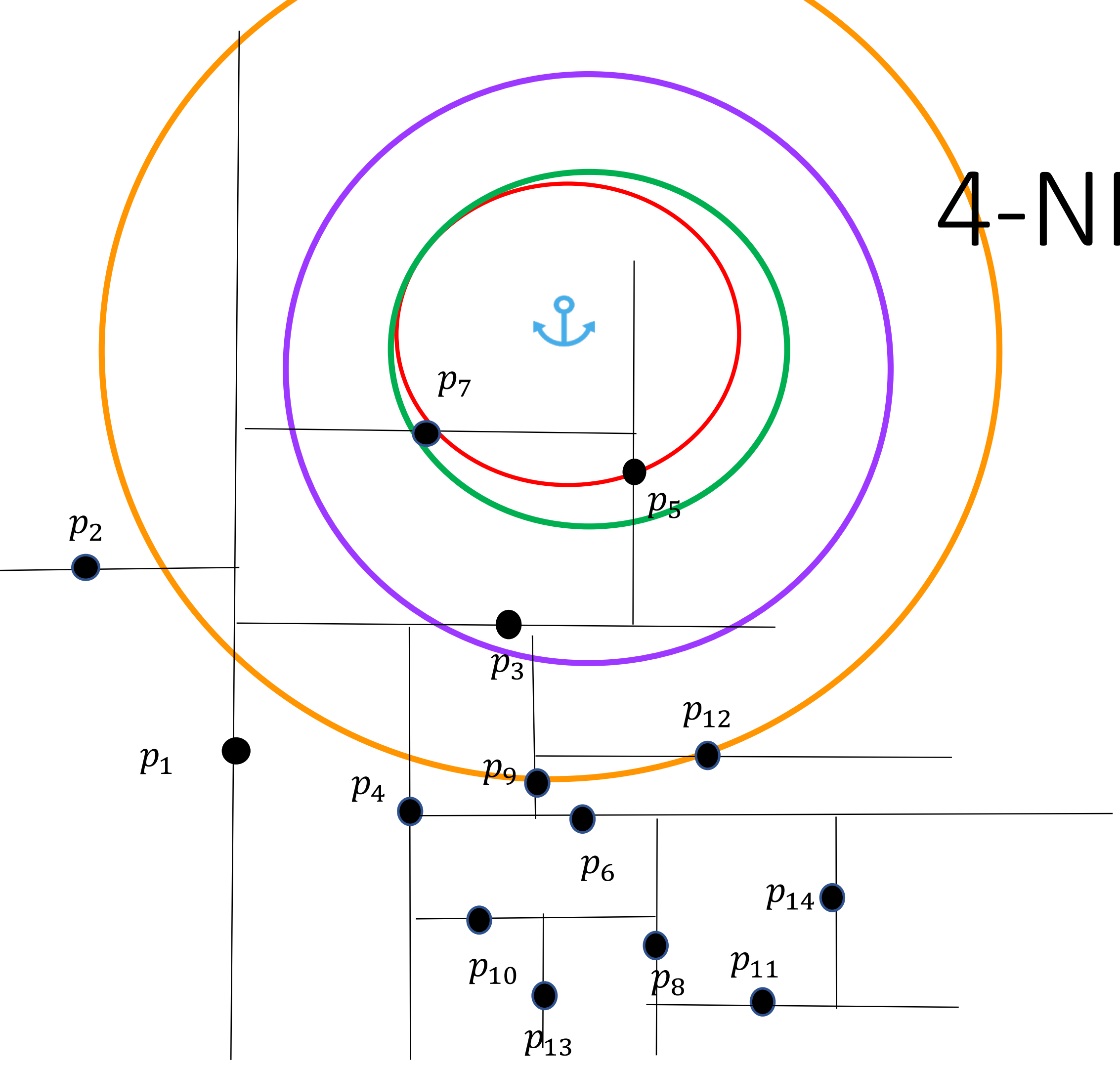
p_7 p_5 p_3 p_4

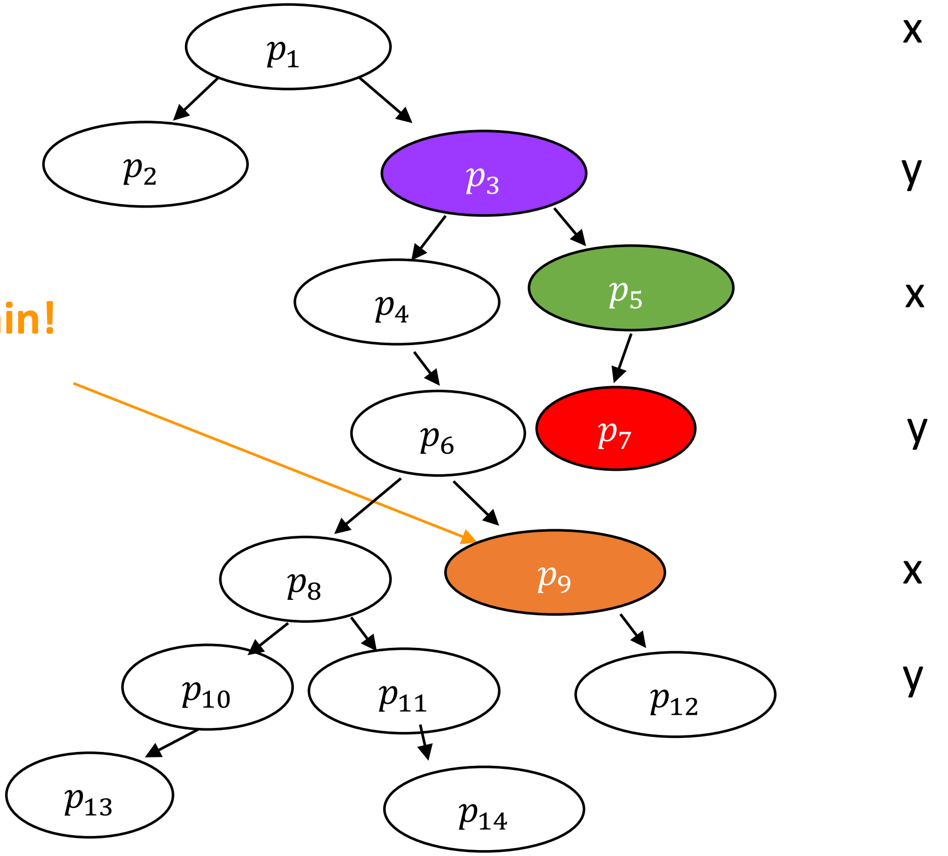
4-NN example



p_1
p_2
p_3
p_4
p_5
p_6
p_7
p_8
p_9
p_{10}
p_{11}
p_{12}
p_{13}
p_{14}
x
y
x
y
x
y
And again!
BPQ
p_7 p_5 p_3 p_6

4-NN example

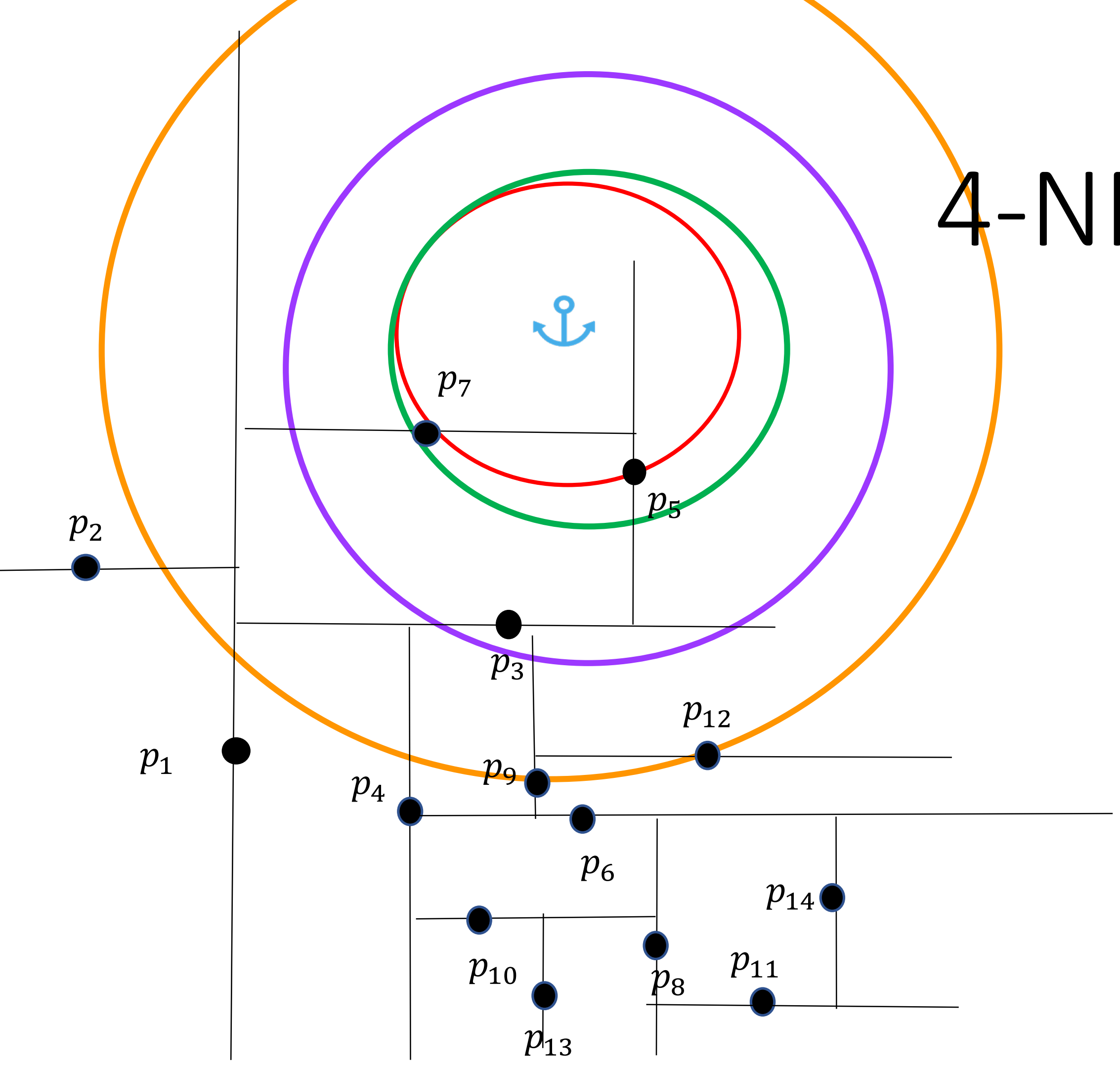


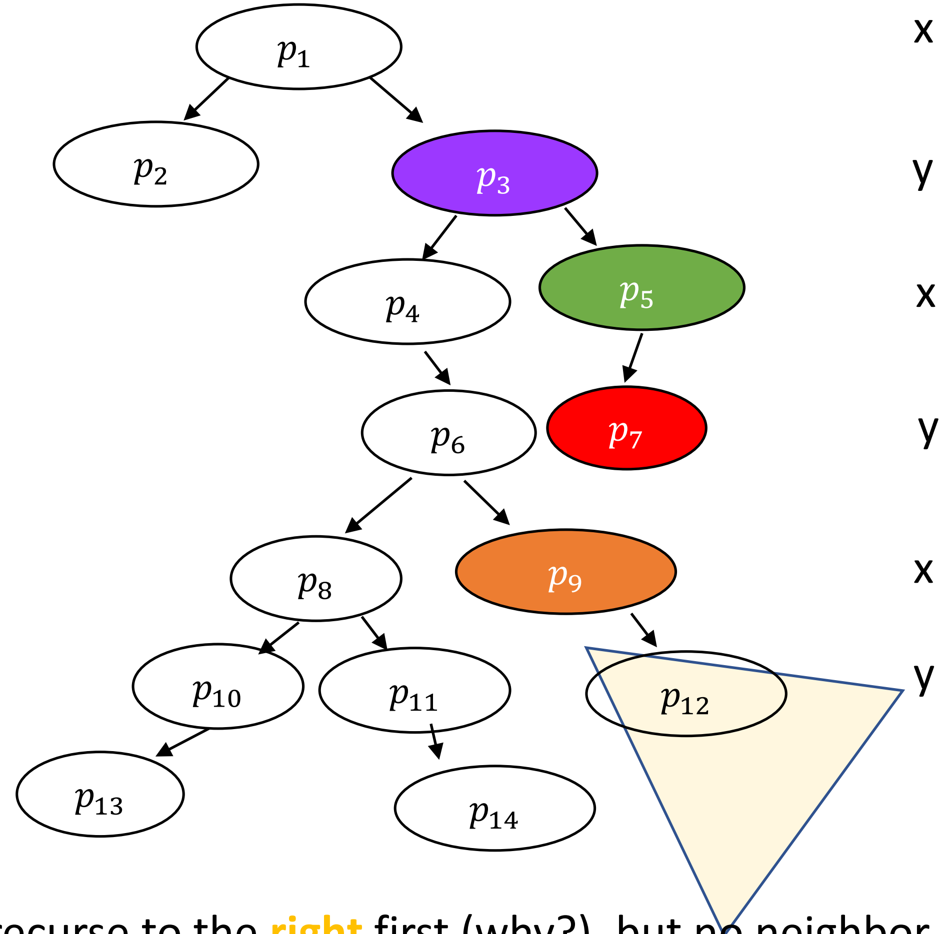


And again!

BPQ

4-NN example



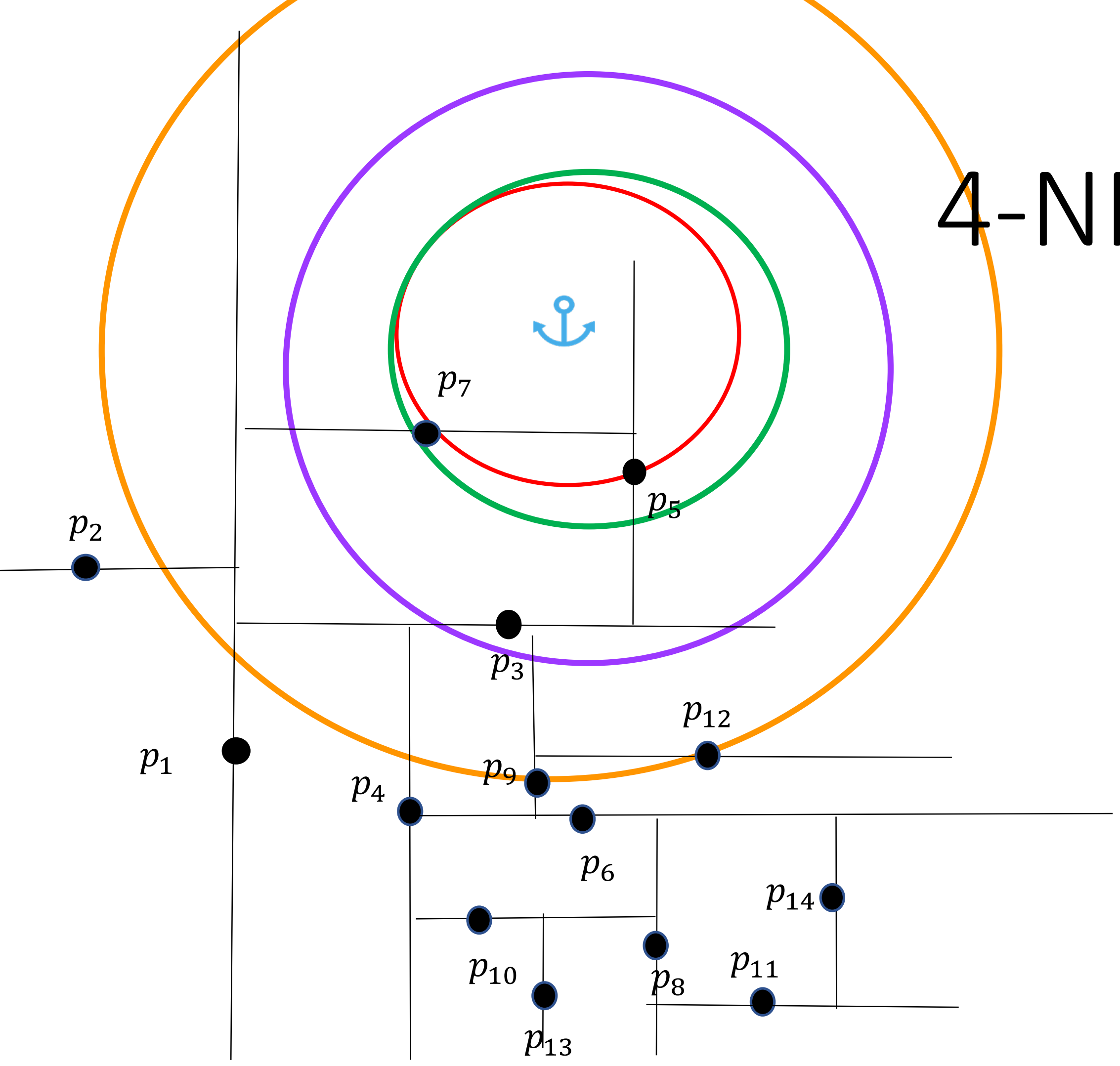


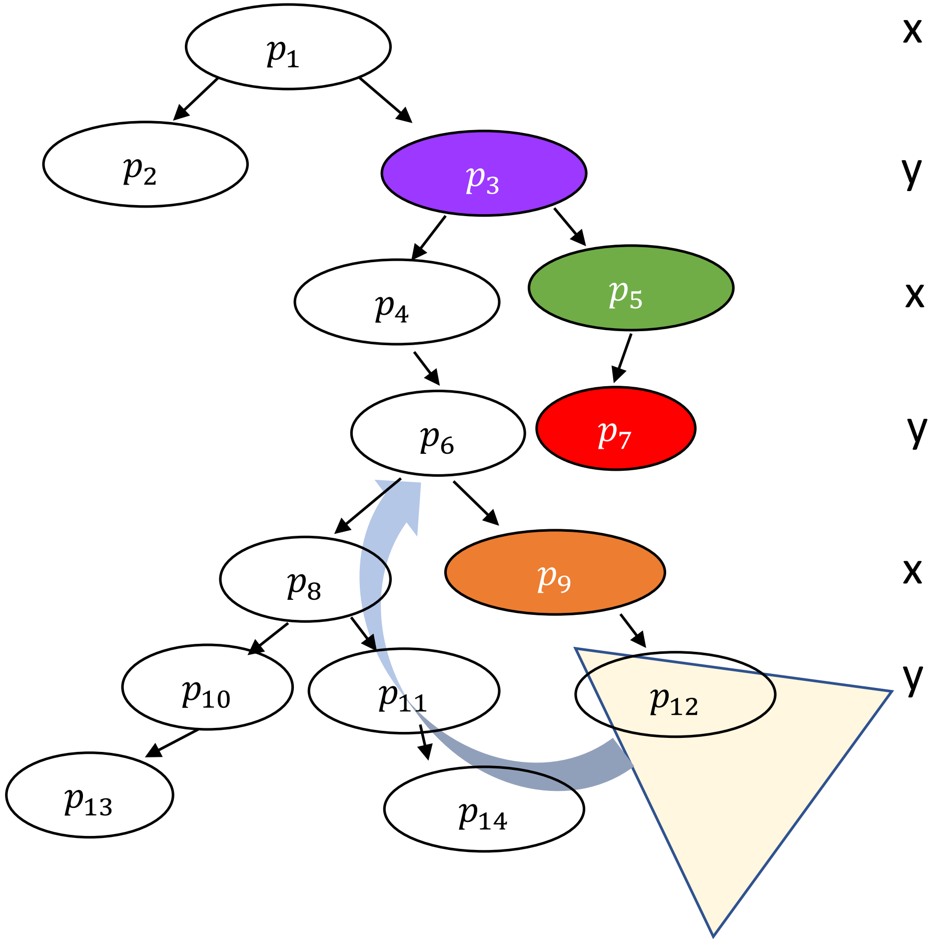
BPQ

p_7	p_5	p_3	p_9

We recurse to the right first (why?), but no neighbor update is made since p_{12} is **exactly** as far away as the **worst** neighbor found so far (p_9)!

4-NN example



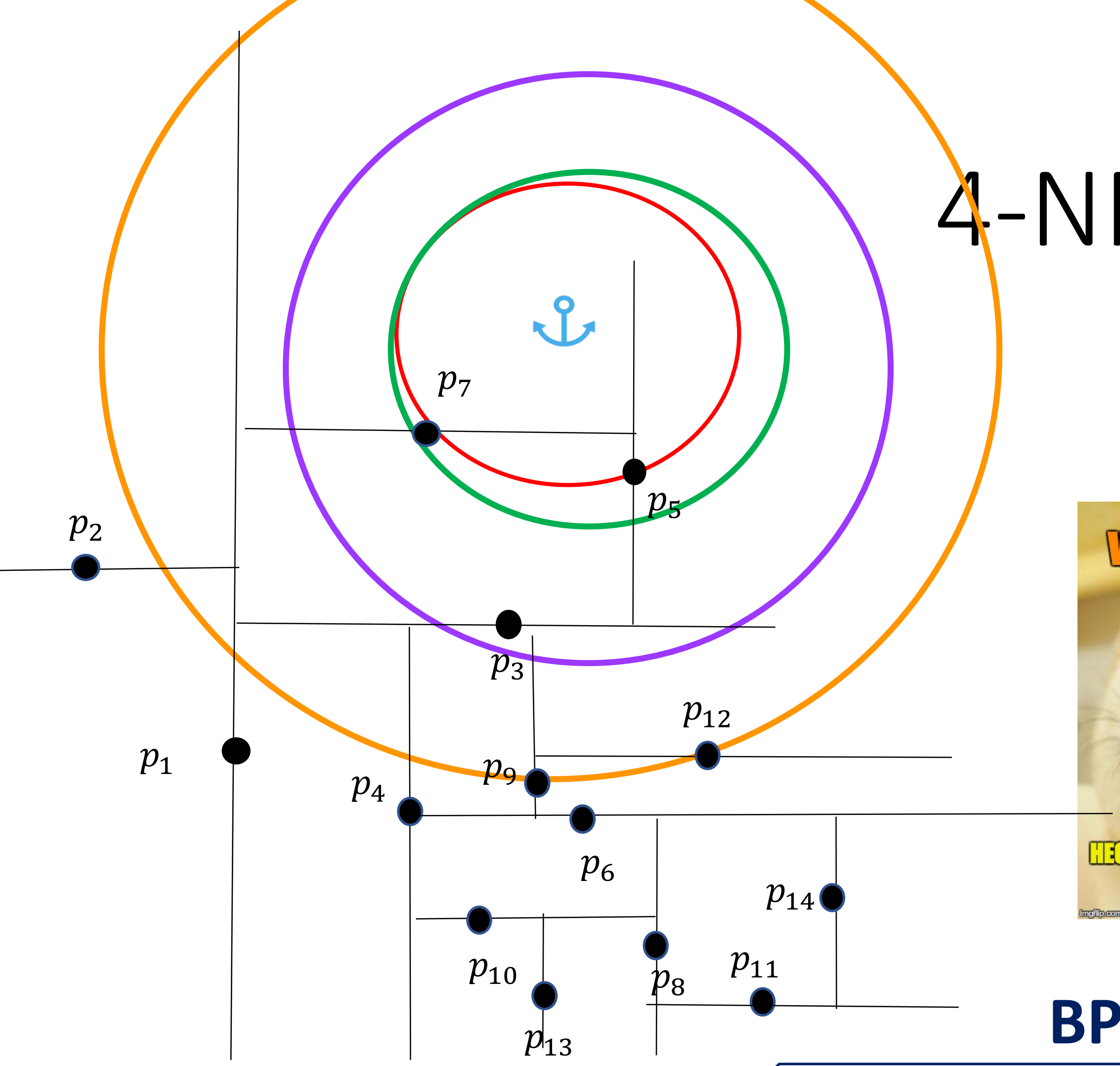


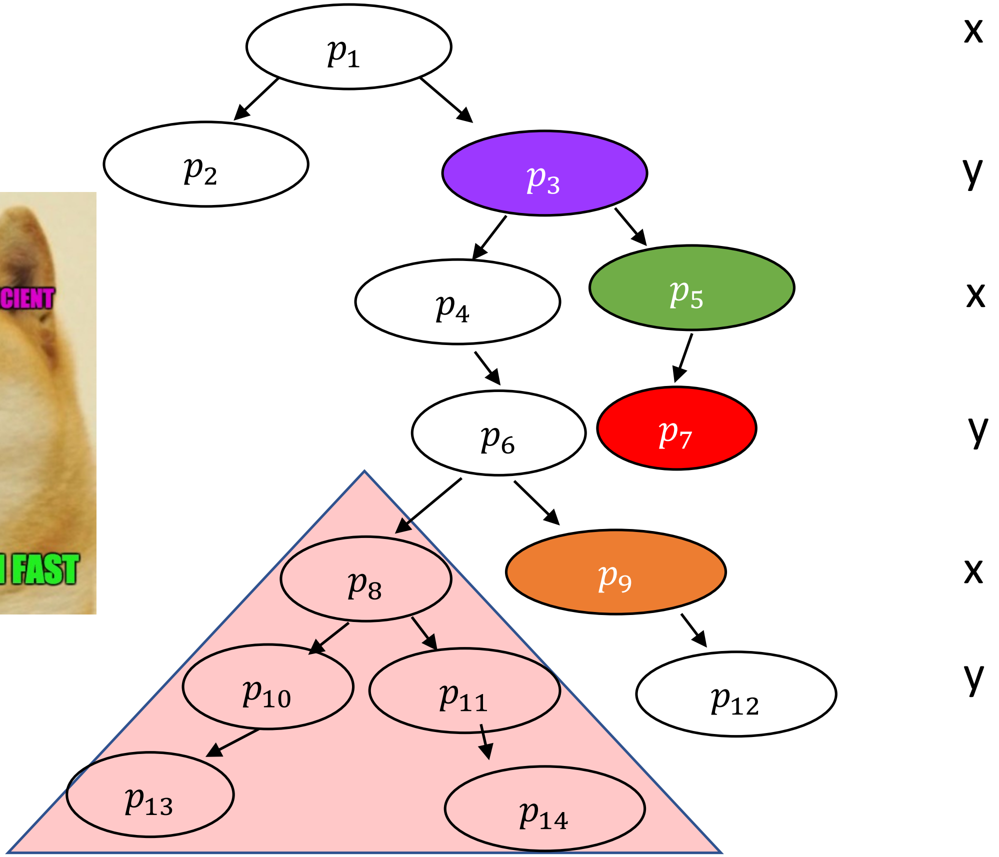
BPQ

p_7	p_5	p_3	p_9

4-NN example

p_7 p_5 p_3 p_9



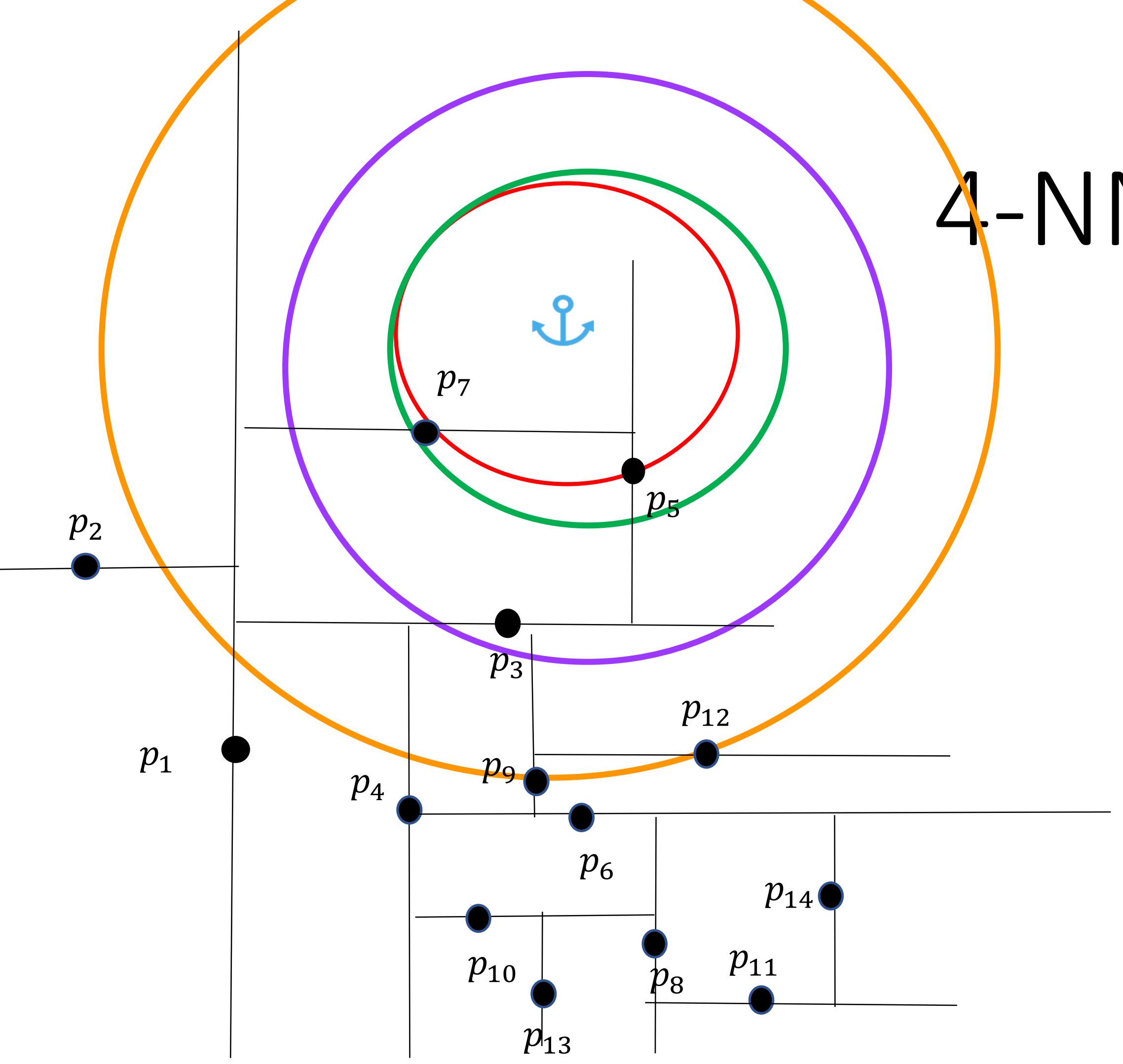


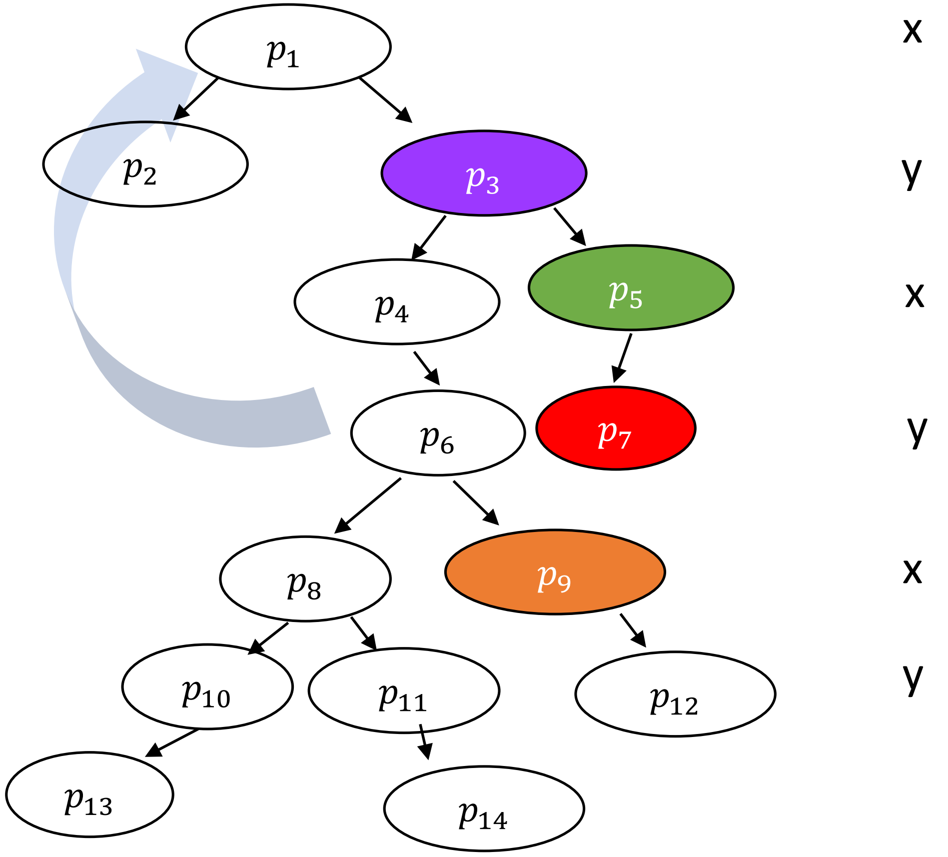
BPQ

p_7 p_5 p_3 p_9

This entire subtree will not be visited, because the worst candidate circle does not intersect the relevant half-plane! ☺

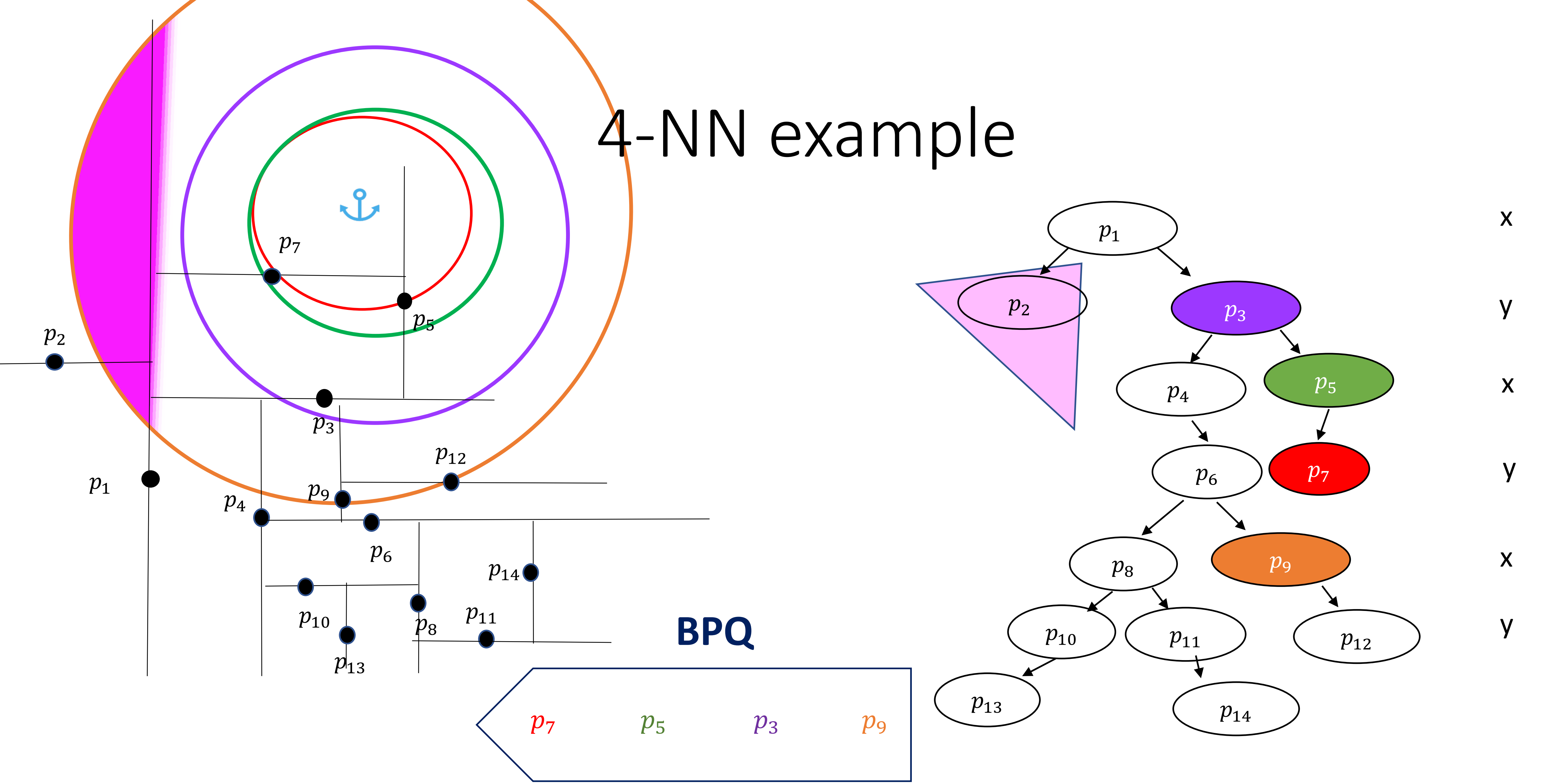
4-NN example





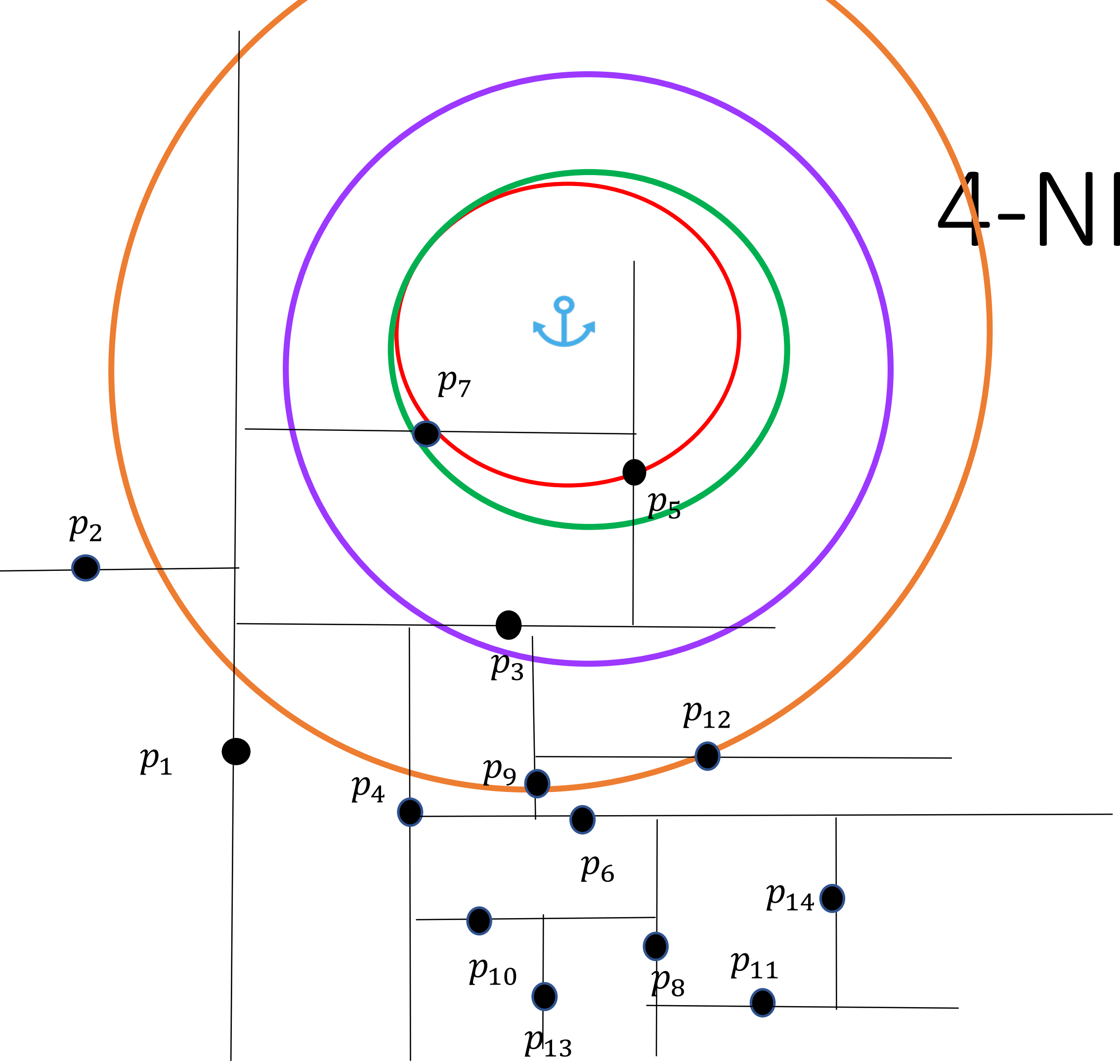
BPQ

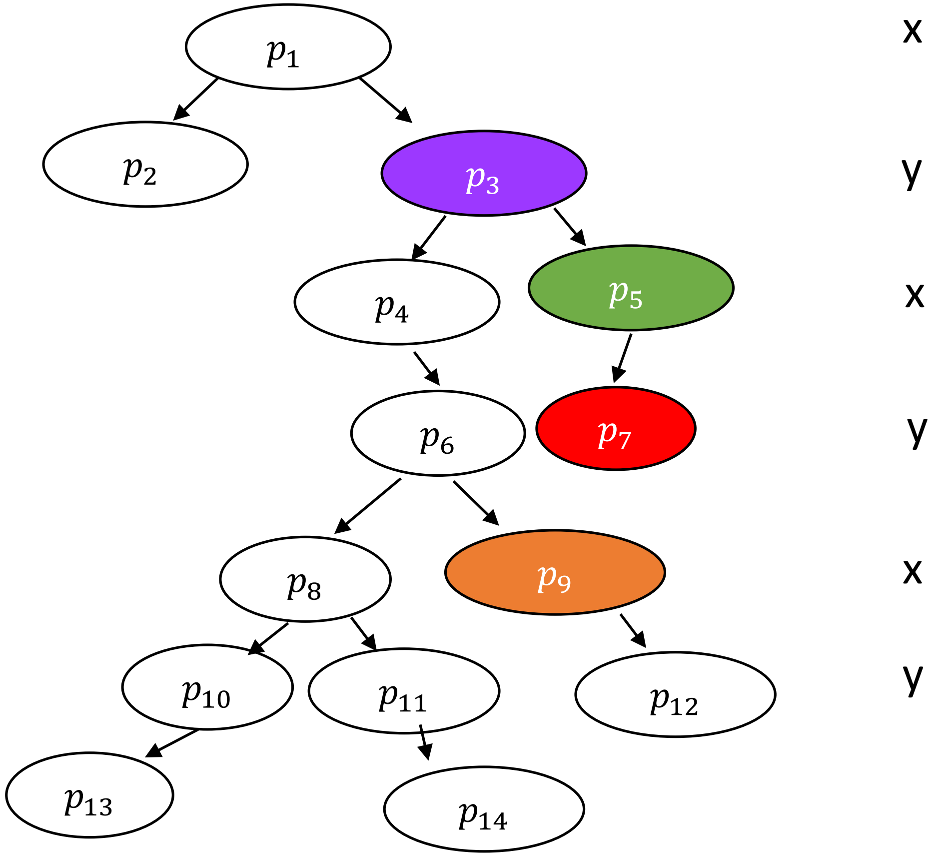
4-NN example



We ***must*** visit the root's left subtree because of possible improvement over p_9, despite the fact that no such improvement is made here!

4-NN example





BPQ

p_7 p_5 p_3 p_9

DONE!

Complexity of nearest neighbor

- Complexity of nearest neighbor in a KD-Tree is unfortunately exponential on k ☹ (the dimension of the tree)
- In fact, on average (balanced KD-Tree) it is:

$$\mathcal{O}(2^k + \log_2 n)$$

Complexity of nearest neighbor

- Complexity of nearest neighbor in a KD-Tree is unfortunately exponential on k ☹ (the dimension of the tree)
- In fact, on average (balanced KD-Tree) it is:

$$\mathcal{O}(2^k + \log_2 n)$$

- Despite this, people *have* used KD-Trees for low-dimensional m-nearest neighbor queries in Machine Learning.

Complexity of nearest neighbor

- Complexity of nearest neighbor in a KD-Tree is unfortunately exponential on k ☹ (the dimension of the tree)
- In fact, on average (balanced KD-Tree) it is:

$$\mathcal{O}(2^k + \log_2 n)$$

- Despite this, people *have* used KD-Trees for low-dimensional m-nearest neighbor queries in Machine Learning.
- State-of-the-art approaches for solving m-nearest neighbors are ***multi-dimensional hashing –based algorithms.***
  - Jason will post resources and can answer questions after lecture / in office hours.