justin-zhang.com/teaching/CS251

All of us after next week

PSO 14

K-D Trees, Point Trees

Announcements

1. Fill out the instructor feedback surveys (ty 40% of you)
2. Last review session on Friday (time TBD, location TBD (existence TBD))
3. No OH next week
    a. I will NOT be on duty

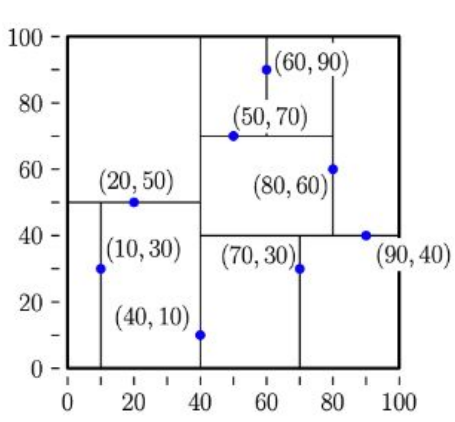
Announcements

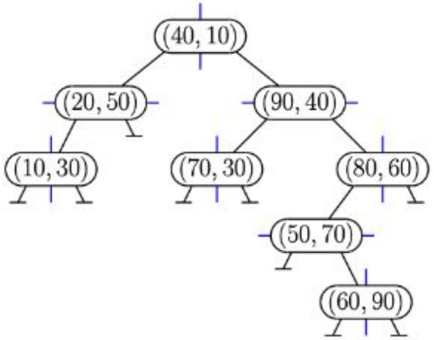
1. Fill out the instructor feedback surveys (ty 40% of you)
2. Last review session on Friday (time TBD, location TBD (existence TBD))
3. No OH next week
    a. I will NOT be on duty but..
    b. I *might* happen to be sitting around the commons from 12-2PM Sat,Sun,Mon,Tues
    c. I *might* be open to answering any questions if they *happen* to be asked
    d. I *might* be hungover

Question 1

(kd trees)

(1) Consider the kd-tree shown in the figure below. Assume a standard kd-tree where the cutting dimensions alternates between x and y with each level.





(1) Show the final tree after the operation **insert((70,50))**.

(2) Starting with the original tree, show the final tree after **delete((40,10))**.

(3) Starting with the original tree, show the final tree after **delete((80,60))**.

Question 2

Consider a **QuadTree** that indexes n points uniformly distributed in a square region $[0, 1] \times [0, 1]$. The QuadTree recursively subdivides each square into four equal quadrants until each quadrant contains at most one point.

Question:

(a) What is the expected **depth** $D(n)$ of the QuadTree?

(b) What is the total number of **leaf nodes** in the tree in terms of n?

(c) If we perform a **range query** for a square region of side length s, what is the expected number of leaf nodes that intersect this query region?

But first..

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa\cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

<u>Boyer-Moore</u>: Iteratively compare pattern P with target, going backward

T	a	**a**	a	a	a	a	a	a	a
P	b	**a**	a	a	a	a			

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa\cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	**a**	a	a	a	a	a	a	a	a
P	**b**	a	a	a	a	a			

T[0] does not equal P[0]! Next steps..

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa\cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	a	a	a
P	b	a	a	a	a	a			

T[0] does not equal P[0]! Next steps.. We mismatched on target a

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa \cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	a	a	a
P	b	a	a	a	a	a			

T[0] does not equal P[0]! Next steps.. We mismatched on target a

The last occurrence of pattern a

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa \cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	a	a	a
P	b	a	a	a	a	a			

Move P (to align target a with pattern a) OR (one after target mismatch)

Whichever moves P the *least* amount – in this ex. We move one after mismatch

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa \cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	a	a	a
P		b	a	a	a	a	a		

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa\cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

<u>Boyer-Moore</u>: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	**a**	a	a
P		b	a	a	a	a	**a**		

Fast forward..

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa \cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

Boyer-Moore: Iteratively compare pattern P with target, going backward

T	a	**a**	a	a	a	a	a	a	a
P		**b**	a	a	a	a	a		

Fast forward.. Same mismatch, jump 1

Question 3

(Backward pattern matching)

The Boyer-Moore algorithm is based upon backward pattern matching. Let us do a simple review via the following questions:

1. Run Boyer-Moore algorithm in the following worst-case scenario:

$$T := \underbrace{aaa \cdots a}_{9} \quad \text{and} \quad P := baaaaa.$$

<u>Boyer-Moore</u>: Iteratively compare pattern P with target, going backward

T	a	a	a	a	a	a	a	a	a
P			b	a	a	a	a	a	

Same thing will happen 1 more time

The example from last time

T	O	O	X	X	X	X	O	O	O
P	O	X	X	X	X	O	O	O	

The example from last time

Mismatch here

T	O	O	X	X	X	**X**	O	O	O
P	O	X	X	X	X	**O**	O	O	

The example from last time

We mismatched on target X
The last occurrence of pattern X

T	O	O	X	X	X	**X**	O	O	O
P	O	X	X	X	X	**O**	O	O	

Move P (to align target X with pattern X) OR (one after target mismatch)
Whichever moves P the *least* amount

The example from last time

We mismatched on target X
The last occurrence of pattern X

T	O	O	X	X	X	**X**	O	O	O
P		O	X	X	X	X	**O**	O	O

Move P (to align target X with pattern X) OR (one after target mismatch)
Whichever moves P the *least* amount

Last note: If there is no last occurrence of the target mismatch, default to one after mismatch

View this at your leisure – a longer example

T	a	b	c	a	b	x	a	x	x	x	x	x	b
P	x	x	b										
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P				x	x	b							
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P					x	x	b						
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P								x	x	b			
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P									x	x	b		
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P										x	x	b	
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P											x	x	b

The green is a comparison made

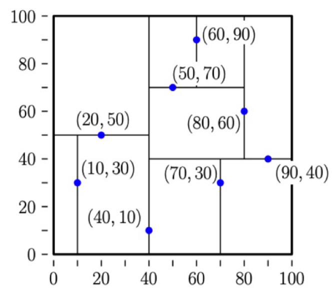
View this at your leisure – a longer example

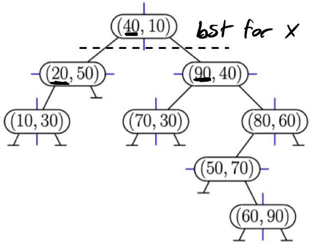
c not in the pattern
We move one after
(In this case, big jump)

T	a	b	c	a	b	x	a	x	x	x	x	x	b
P	x	x	b										
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P				x	x	b							
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P					x	x	b						
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P								x	x	b			
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P									x	x	b		
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P										x	x	b	
T	a	b	c	a	b	x	a	x	x	x	x	x	b
P											x	x	b

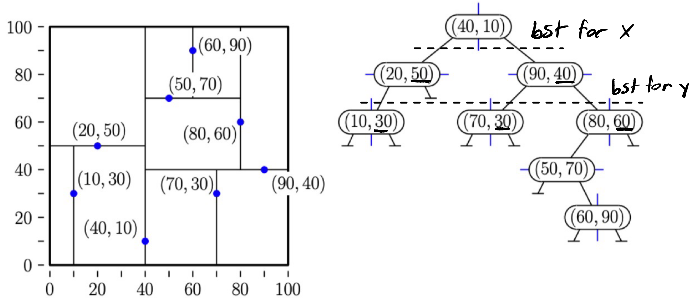
The green is a comparison made

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

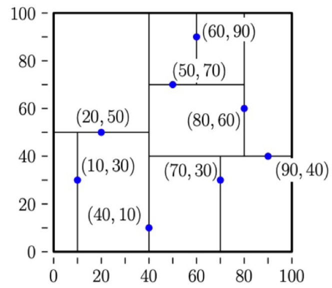


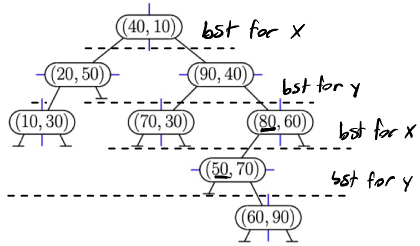


(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



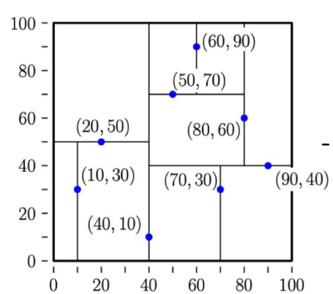
(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

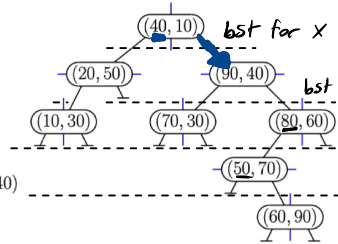




Now lets insert (70,50)

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.





bst for x

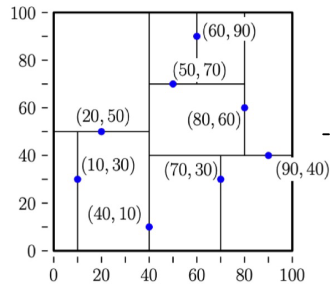
bst for y

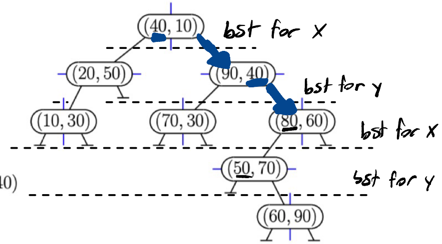
bst for x

bst for y

Now lets insert (70,50)

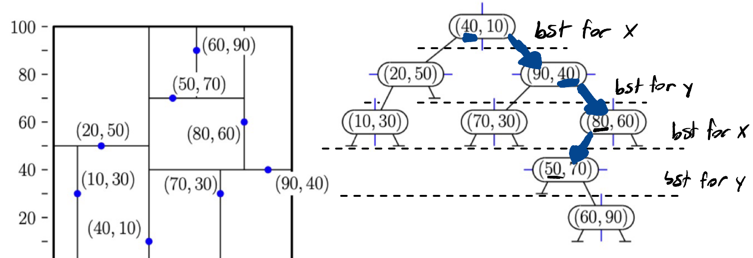
(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.





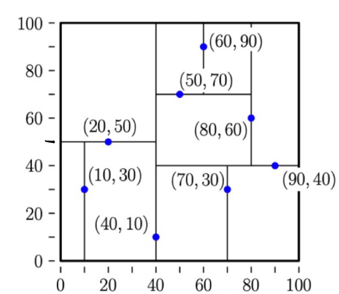
Now lets insert (70,50)

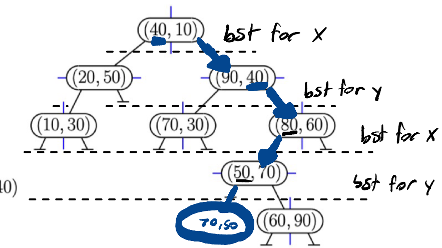
(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



Now lets insert (70,50)

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

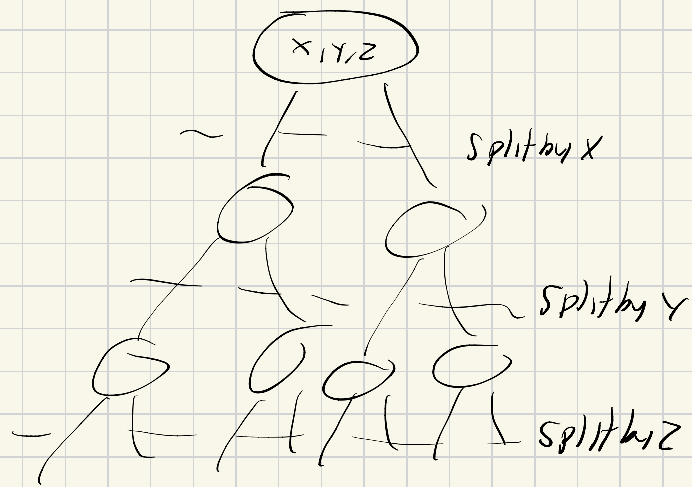




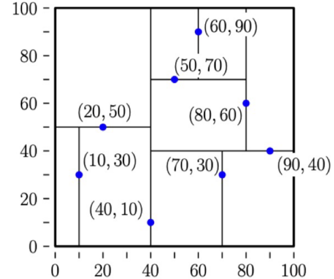
Now lets insert (70,50)

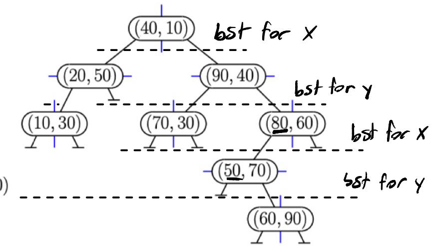
2d tree

3d tree



(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.





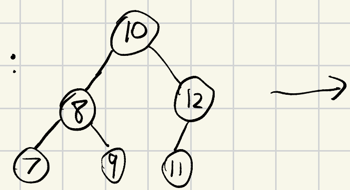
Now lets delete (40.10)

Recall deletion in normal BST

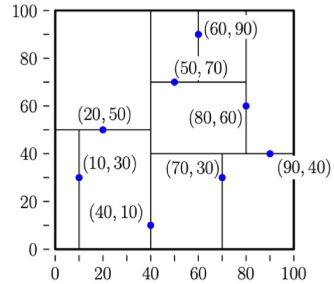
- Find Predecessor/Successor leaf to replace

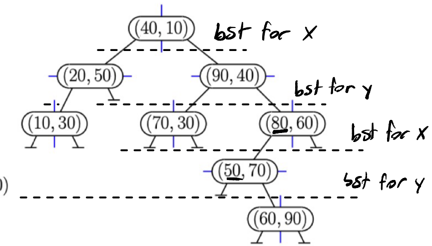
ex:

delete(10):



(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.





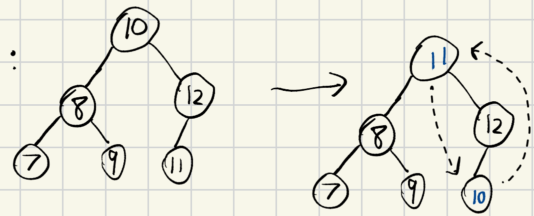
Now lets delete (40,10)

Recall deletion in normal BST

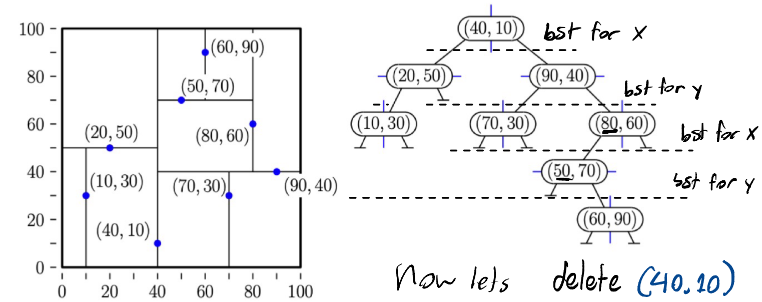
- Find Predecessor/Successor leaf to replace

ex:

delete(10):



(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



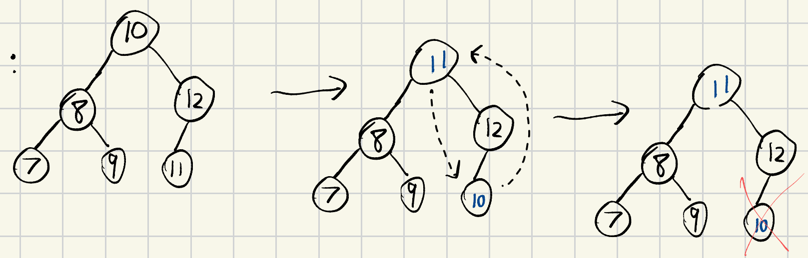
Now lets delete (40,10)

Recall deletion in normal BST

• Find Predecessor/Successor leaf to replace

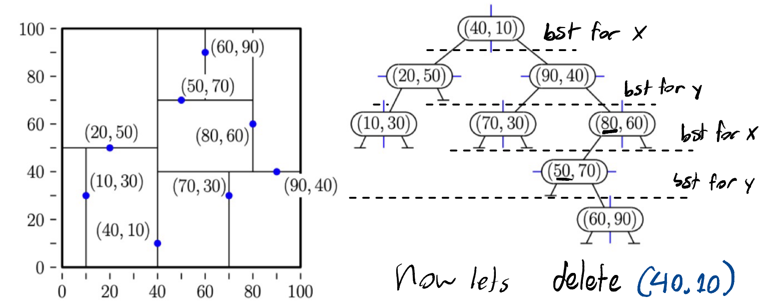
ex:

delete(10):



(If not at leaf continue recursively)

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



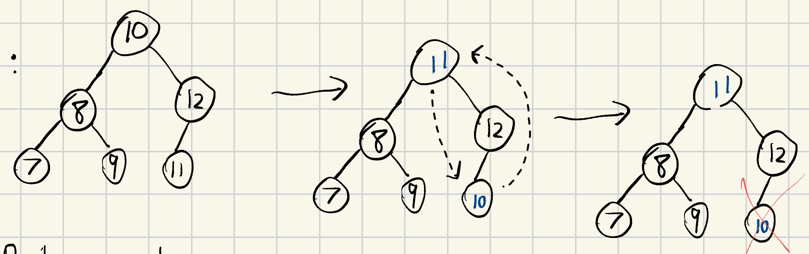
Now lets delete (40.10)

Recall deletion in normal BST

- Find predecessor/successor leaf to replace

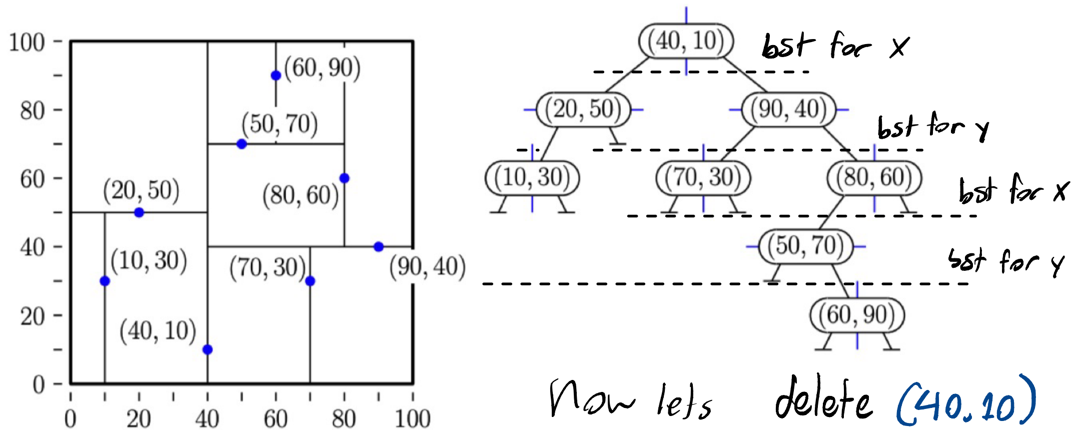
ex:

delete(10):



Deletion in KD-trees is the same.
except you choose successor/predecessor based on dimension

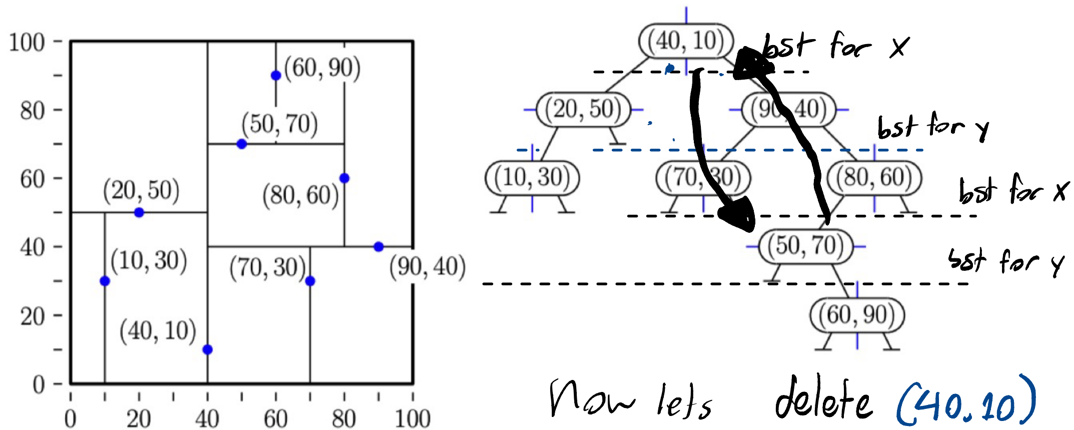
(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



Now lets delete (40.10)

1) Find successor for X=40

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

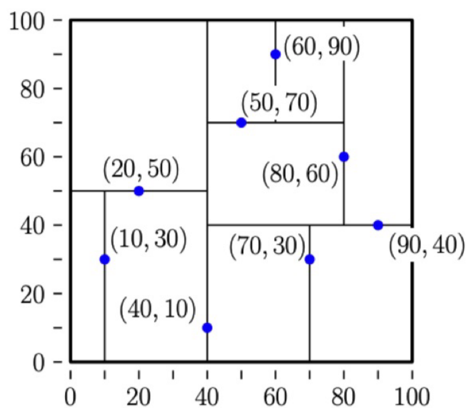


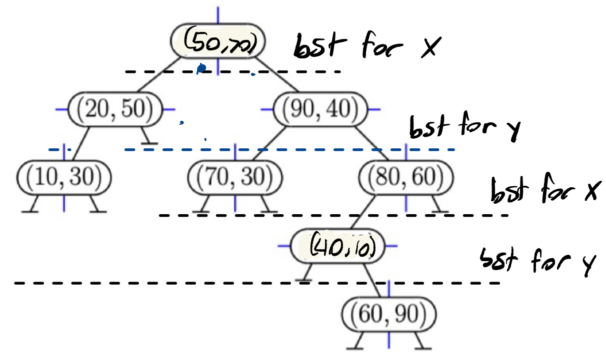
Now lets delete (40.10)

2) Swap with (40,10)

[Think about why this preserves k-d order]

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.



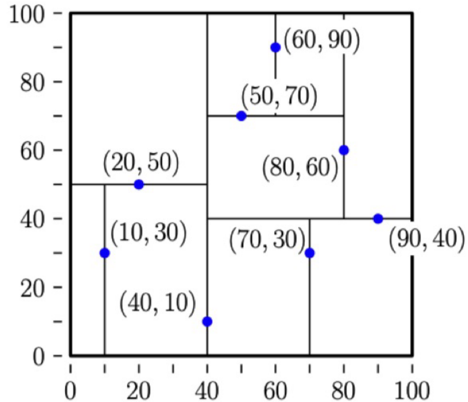


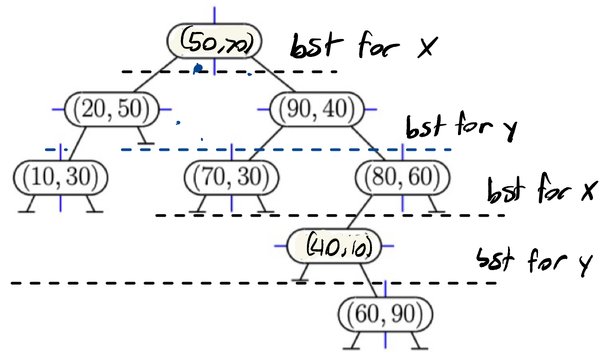
Now lets delete (40,10)

2) Swap with (40,10)

[Think about why this preserves k-d order]

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

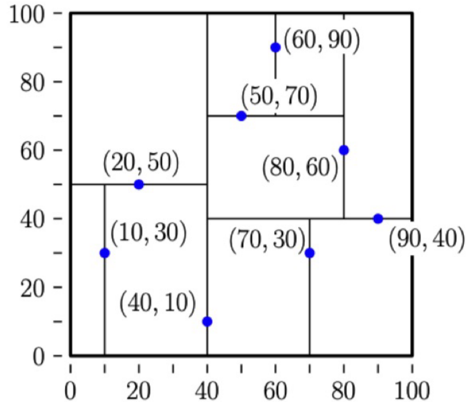


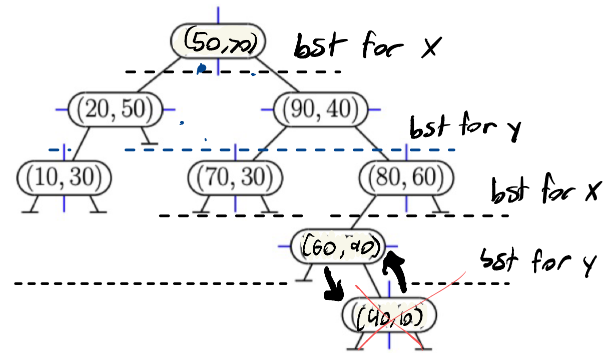


Now lets delete (40,10)

2.2) Not at leaf yet, find successor in y = 10 and swap.

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.

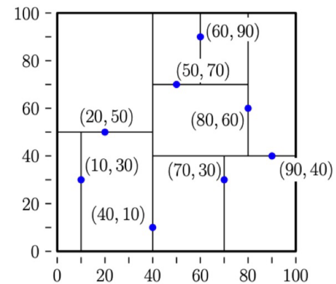


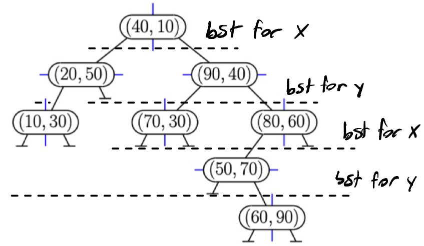


Now lets delete (40.10)

2.2) Not at leaf yet, find successor in $y = 10$ and swap.

(2) **(kd-trees)** Consider the kd-tree shown below. We assume it's a standard kd-tree where the cutting dimensions alternates between x and y with each level. Show the final tree after the operation **insert((70,50))**.





Now lets delete (80,60).

Left as Exercise

NOTE: If no successor, find a predecessor instead and vice/versa.

Question 2

Consider a **QuadTree** that indexes n points uniformly distributed in a square region $[0, 1] \times [0, 1]$. The QuadTree recursively subdivides each square into four equal quadrants until each quadrant contains at most one point.

Question:

(a) What is the expected **depth** $D(n)$ of the QuadTree?

(b) What is the total number of **leaf nodes** in the tree in terms of n?

(c) If we perform a **range query** for a square region of side length s, what is the expected number of leaf nodes that intersect this query region?

for now

I'm going to ignore this question and go over how we insert into a quad tree.

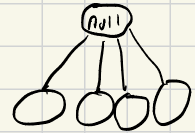
Question 1

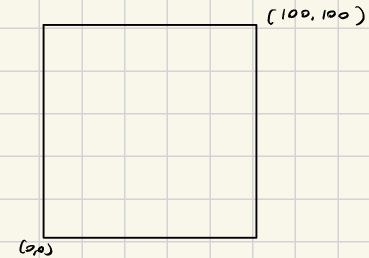
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

• Suppose root.pt = null

root.region = (0,0,100,100)

root.children = [ ]





1) Insert (35,40)

Question 1

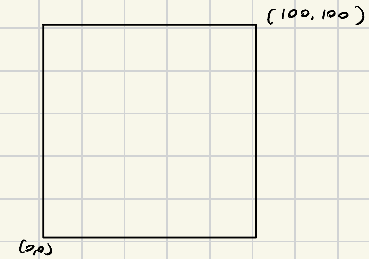
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

• Suppose root.pt = null

root.region = (0, 0, 100, 100)

root.children = [ ]

null



1) Insert (35,40)

```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

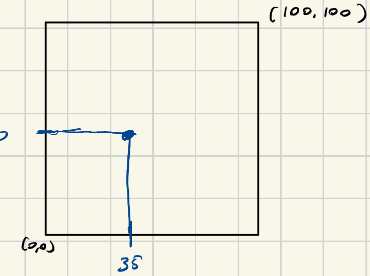
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

root.pt = (35,40)

root.region = (0,0,100,100)

root.children = [ ]



1) Insert (35,40)

```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

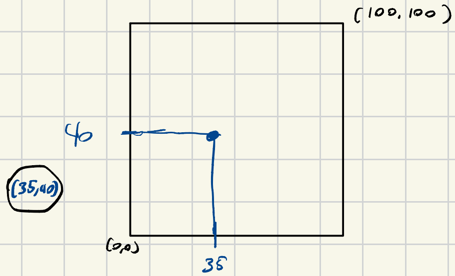
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)

root.pt = (35, 40)
root.region = [0, 0, 100, 100)
root.children = [ ]




```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

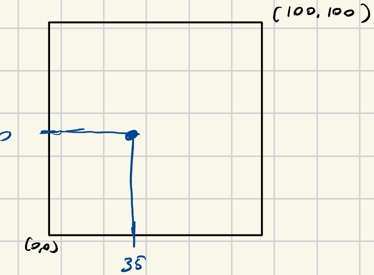
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)

root.pt = (35,40)
root.region = (0,0,100,100)
root.children = [ ]




```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

```
algorithm subdivide(root:node)
   xmin, ymin, xmax, ymax ← root.region
   xmid ← (xmin + xmax) / 2
   ymid ← (ymin + ymax) / 2
   root.children ← [
      Node(Region(xmin, ymid, xmid, ymax)), //NW
      Node(Region(xmid, ymid, xmax, ymax)), //NE
      Node(Region(xmin, ymin, xmid, ymid)), //SW
      Node(Region(xmid, ymin, xmax, ymid))] //SE
   if root.point is not Null then
      P ← root.point
      root.point ← Null
      for each quadrant in root.children do
         if insert(P, quadrant) then
            return
```

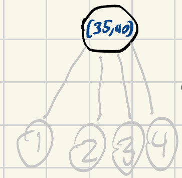
Question 1

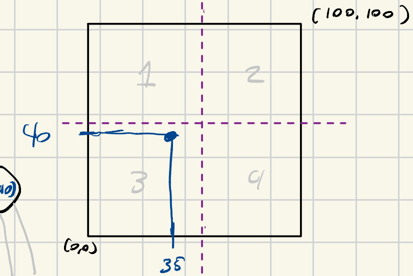
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)

root.pt = (35,40)
root.region = (0,0,100,100)
root.children = [ ]






```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

```
algorithm subdivide(root:node)
   xmin, ymin, xmax, ymax ← root.region
   xmid ← 50
   ymid ← 50
   root.children ← [
      Node(Region(xmin, ymid, xmid, ymax)), //NW
      Node(Region(xmid, ymid, xmax, ymax)), //NE
      Node(Region(xmin, ymin, xmid, ymid)), //SW
      Node(Region(xmid, ymin, xmax, ymid))] //SE
   if root.point is not Null then
      P ← root.point
      root.point ← Null
      for each quadrant in root.children do
         if insert(P, quadrant) then
            return
```

Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

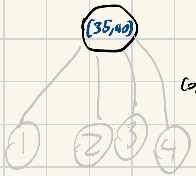
1) Insert (35,40)

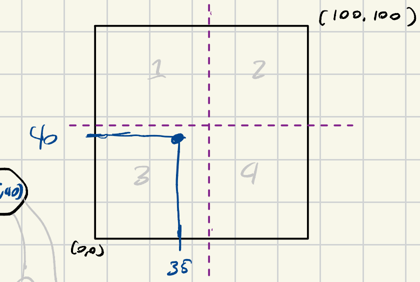
2) Insert (50,10)

root.pt = (35,40)

root.region = [0,0,100,100)

root.children = [ ]






```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

```
algorithm subdivide(root:node)
   xmin, ymin, xmax, ymax ← root.region
   xmid ← 50
   ymid ← 50
   root.children ← [
      Node(Region(xmin, ymid, xmid, ymax)), //NW
      Node(Region(xmid, ymid, xmax, ymax)), //NE
      Node(Region(xmin, ymin, xmid, ymid)), //SW
      Node(Region(xmid, ymin, xmax, ymid))] //SE
   if root.point is not Null then
      P ← root.point
      root.point ← Null
      for each quadrant in root.children do
         if insert(P, quadrant) then
            return
```

What does this intuitively do?

Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

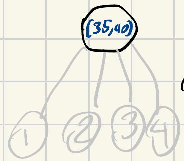
1) Insert (35,40)

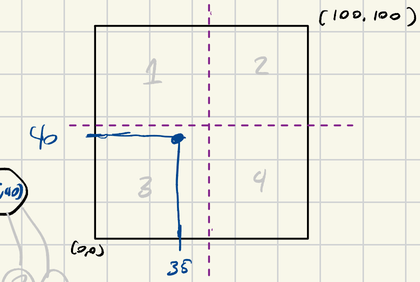
2) Insert (50,10)

root.pt = (35,40)

root.region = (0,0,100,100)

root.children = [ ]






```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

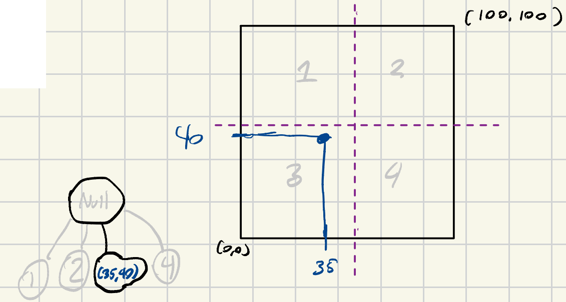
```
algorithm subdivide(root:node)
   xmin, ymin, xmax, ymax ← root.region
   xmid ← 50
   ymid ← 50
   root.children ← [
      Node(Region(xmin, ymid, xmid, ymax)), //NW
      Node(Region(xmid, ymid, xmax, ymax)), //NE
      Node(Region(xmin, ymin, xmid, ymid)), //SW
      Node(Region(xmid, ymin, xmax, ymid))] //SE
   if root.point is not Null then
      insert root into the quadrant it belongs
      (and set original root pt to null)
```

Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)




```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

```
algorithm subdivide(root:node)
   xmin, ymin, xmax, ymax ← root.region
   xmid ← 50
   ymid ← 50
   root.children ← [
      Node(Region(xmin, ymid, xmid, ymax)), //NW
      Node(Region(xmid, ymid, xmax, ymax)), //NE
      Node(Region(xmin, ymin, xmid, ymid)), //SW
      Node(Region(xmid, ymin, xmax, ymid))] //SE
   if root.point is not Null then
      insert root into the quadrant it belongs
```

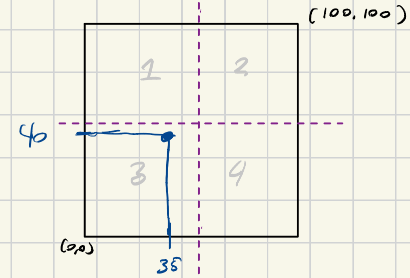
- we do this on the second insert!
- Still need to insert (50,10)

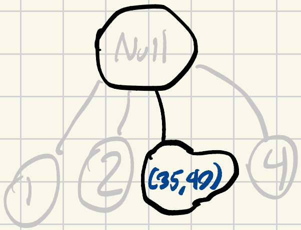
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)

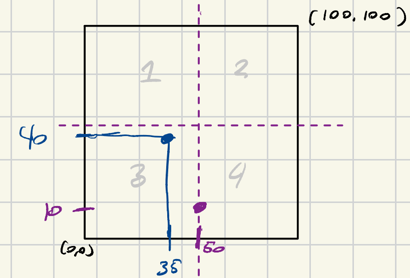





```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

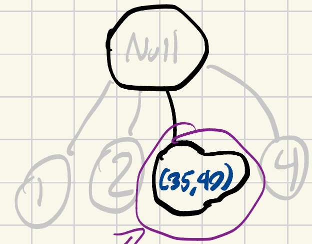
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).



1) Insert (35,40)

2) Insert (50,10)

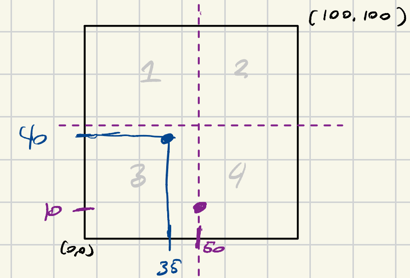



```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

- insert into quadrant 3

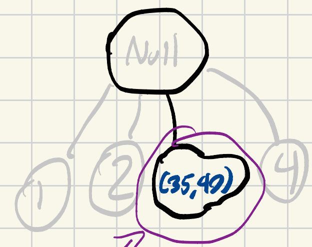
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).



1) Insert (35,40)

2) Insert (50,10)

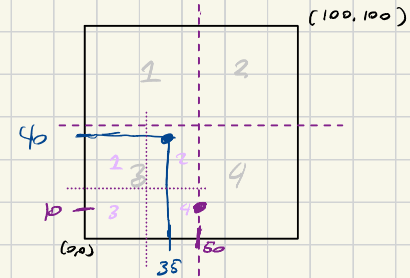



```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

- insert into quadrant 3
- this node has no children, need to further subdivide, then move into new subquadrant

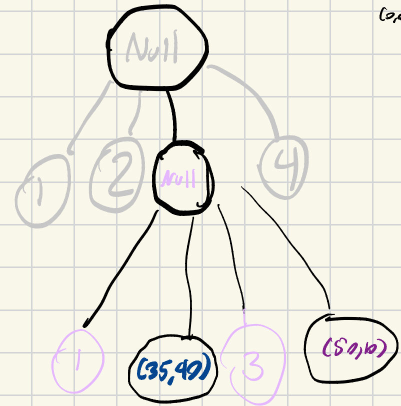
Question 1

(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).



1) Insert (35,40)

2) Insert (50,10)




```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```

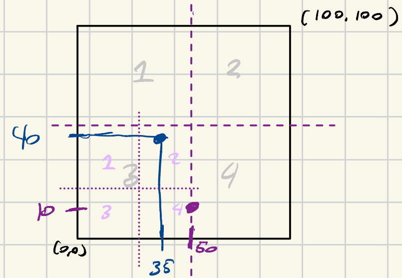
- insert into quadrant 3
- this node has no children, need to further subdivide and move into new subquadrant

Question 1

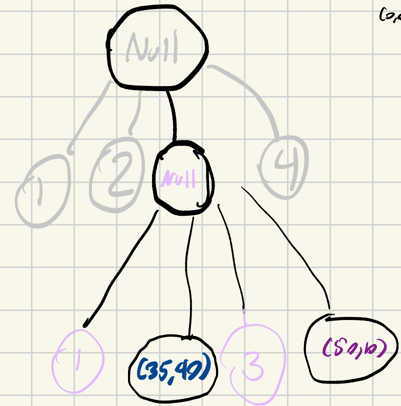
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

1) Insert (35,40)

2) Insert (50,10)




```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```



Take away

Insert always inserts at a leaf.

Allows for log n height

Question 1

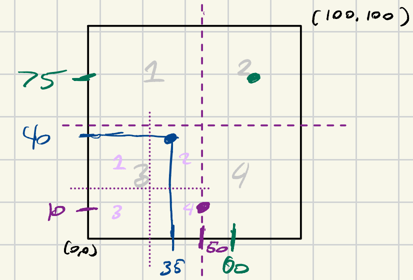
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

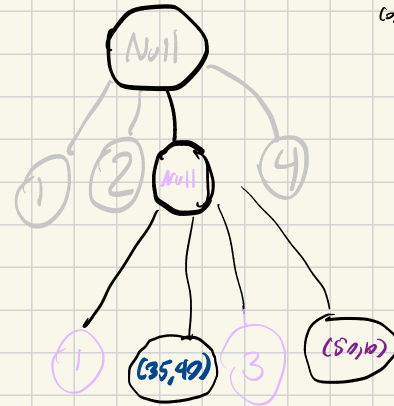
1) Insert (35,40)

2) Insert (50,10)

3) Insert (60,75)

```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```





Question 1

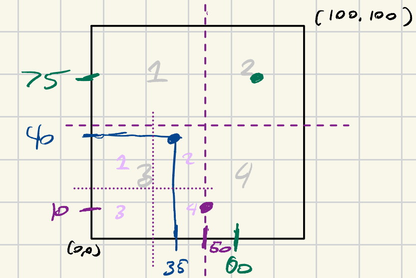
(1) **(Quadtrees)** Insert the following points (in order) into an empty Point-region Quadtree: (35,40), (50,10), (60,75), (80,65), (85,15), (5,45), (25,35), (90,5).

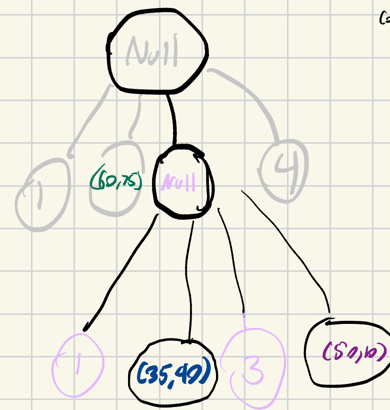
1) Insert (35,40)

2) Insert (50,10)

3) Insert (60,75)

```
algorithm insert(P:point, root:node) → bool
   if not inboundary(root.region, P) then
      return false
   end if
   if root.point is Null and |root.children| = 0 then
      root.point ← P
      return true
   end if
   if |root.children| = 0 then
      subdivide(root)
   end if
   for each quadrant in root.children do
      if insert(P, quadrant) then
         return true
      end if
   end for
end algorithm
```





Question 2

Consider a **QuadTree** that indexes n points uniformly distributed in a square region $[0,1] \times [0,1]$. The QuadTree recursively subdivides each square into four equal quadrants until each quadrant contains at most one point.

Question:

(a) What is the expected **depth** $D(n)$ of the QuadTree?

(b) What is the total number of **leaf nodes** in the tree in terms of n?

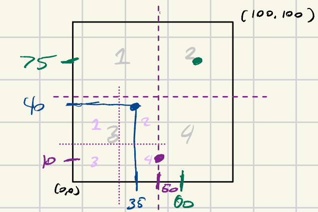
(c) If we perform a **range query** for a square region of side length s, what is the expected number of leaf nodes that intersect this query region?

What is the takeaway of the example for this question?

a) When we evenly subdivide into 4 regions

NOT Guaranteed!!

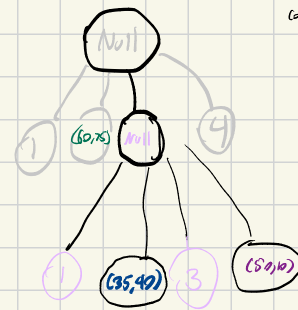
$$4^d \approx n$$



b. # non-null leaves $= n$

leaves $\approx n$

c. geometry ahh question.



Poll

What do YOU want to see at the last practice session?

- Asymptotic analysis
- Graph problems

Do you want us to keep doing Mult. Choice or do more Free response-y questions?