Advanced Algorithms – COMS31900

Orthogonal Range Searching

Raphaël Clifford

Slides by Benjamin Sach

Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.

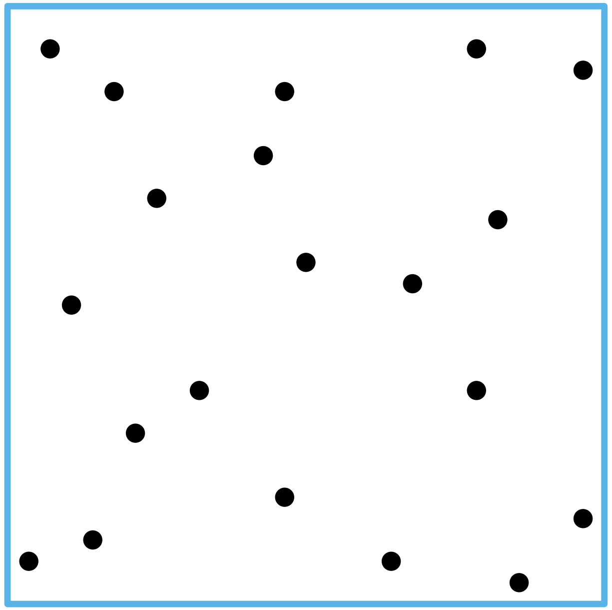
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



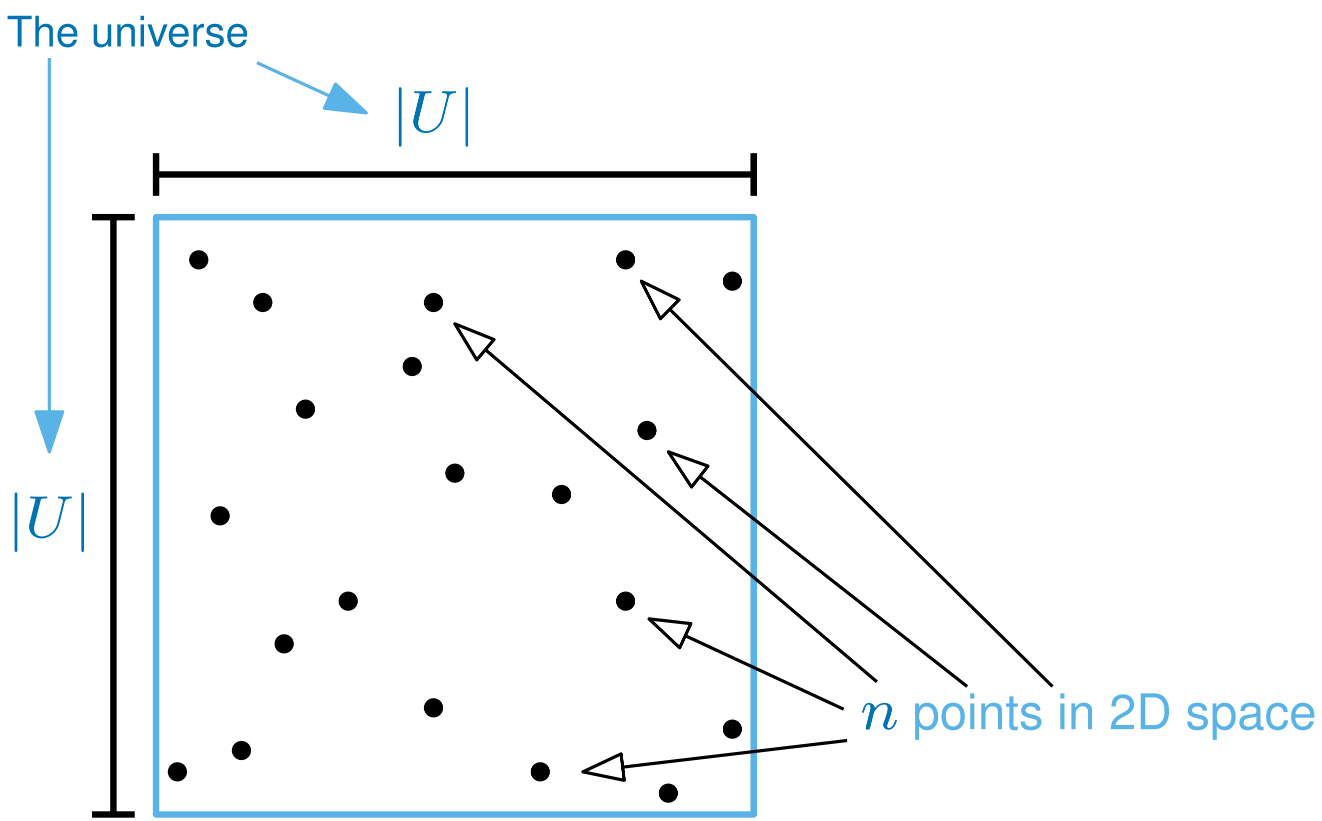
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



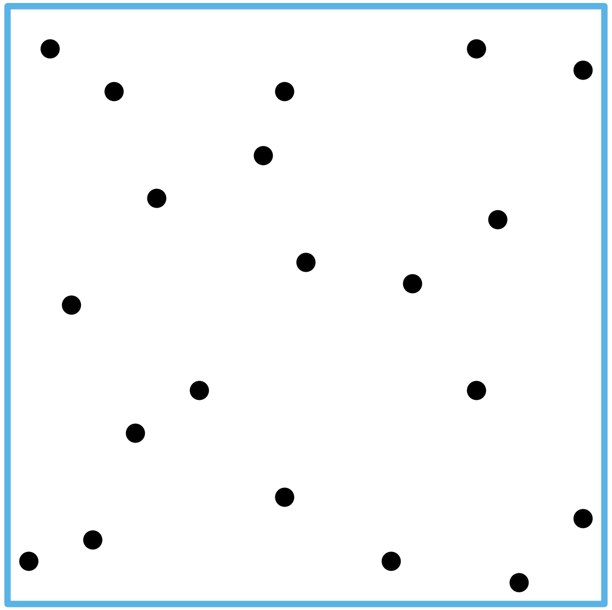
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



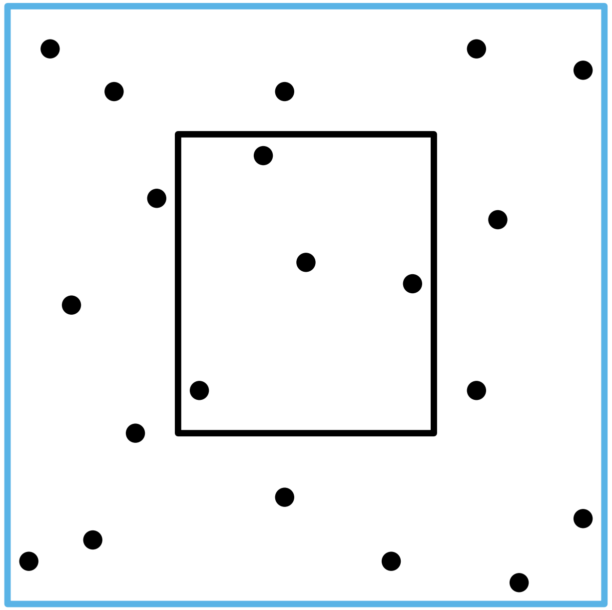
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



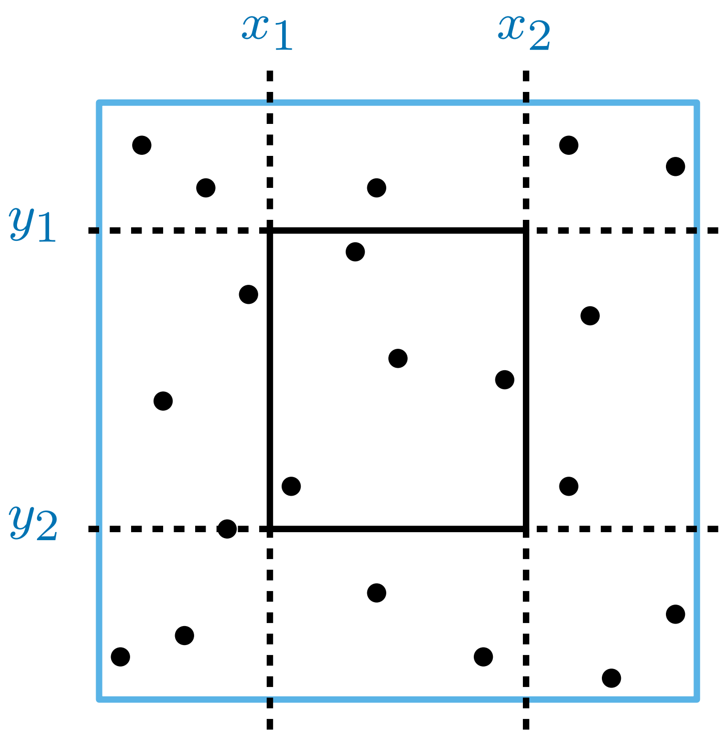
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



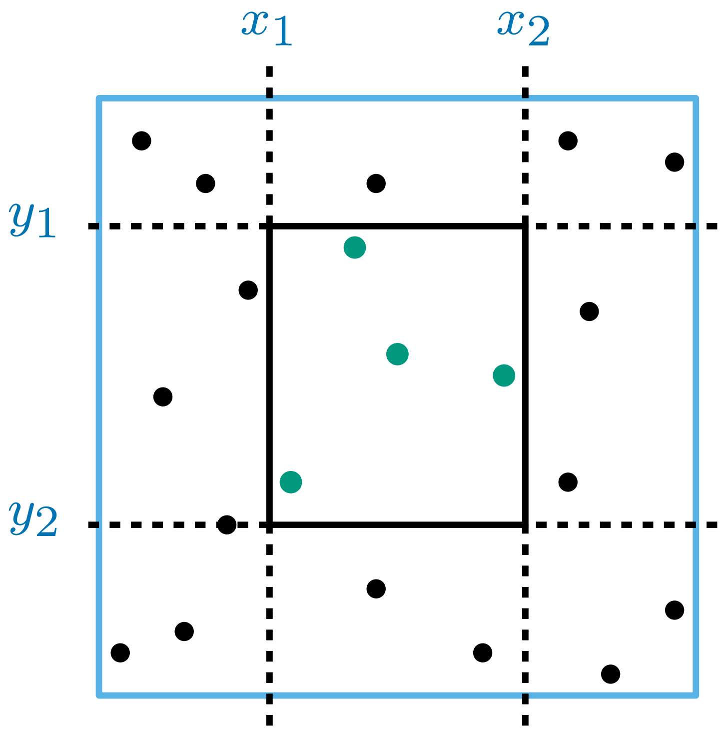
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



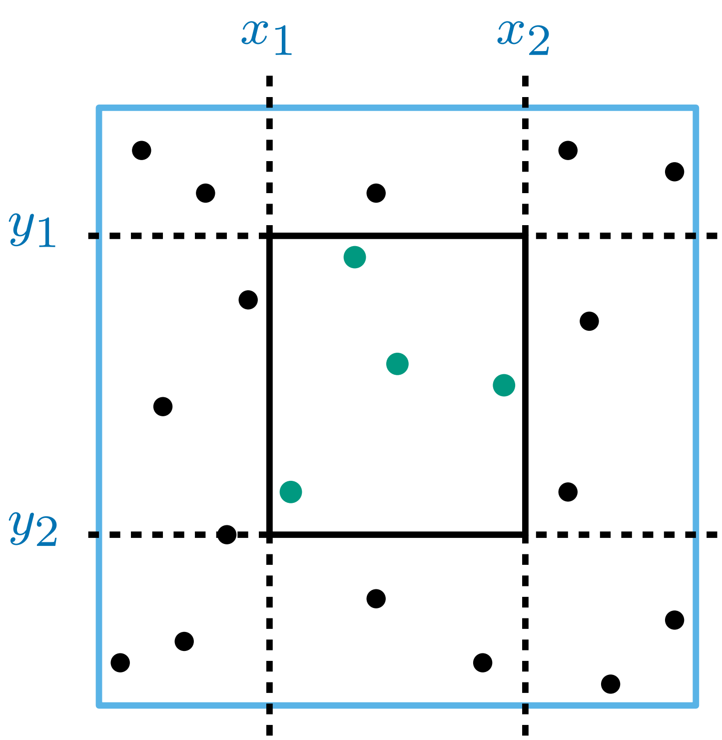
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



A classic database query

"find all employees aged between 21 and 48 with salaries between £23k and £36k"

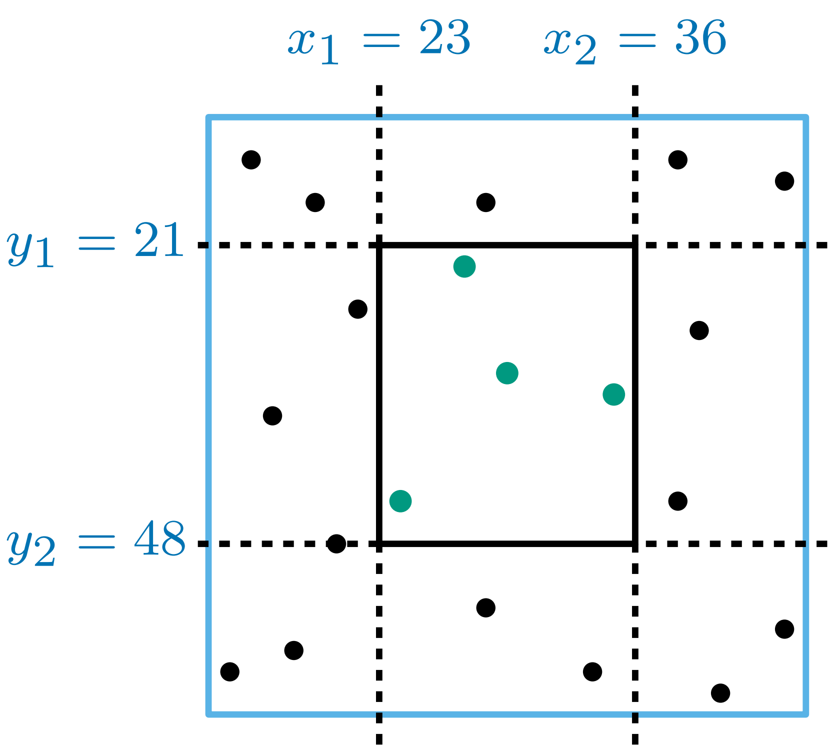
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



A classic database query

"find all employees aged between 21 and 48 with salaries between £23k and £36k"

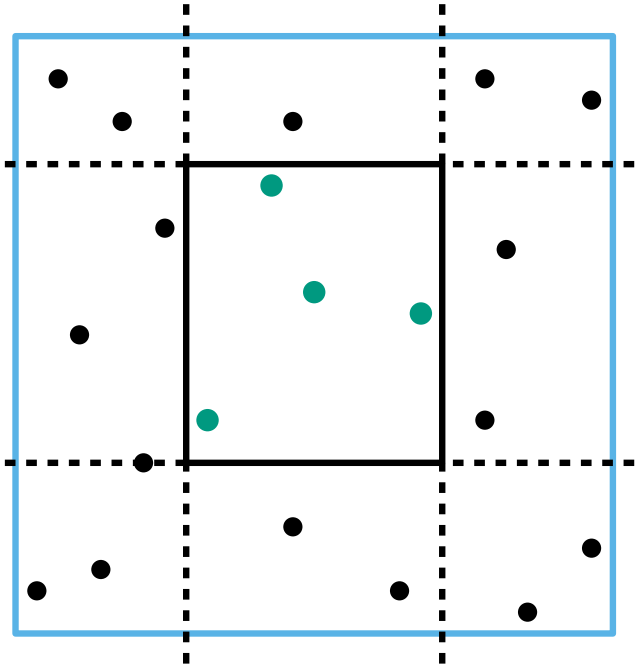
Orthogonal range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Orthogonal range searching

- A **d-dimensional** **range searching data structure** stores n distinct points

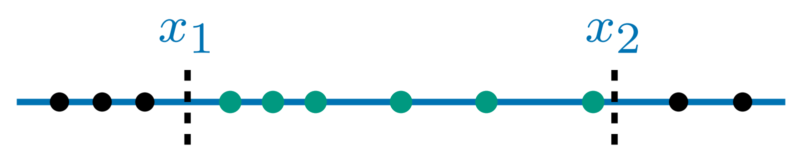
  each point has d coordinates

  (we assume d is a constant)

---

for $d = 1$, the lookup(x_1, x_2) operation

returns every point with $x_1 \leqslant x \leqslant x_2$.

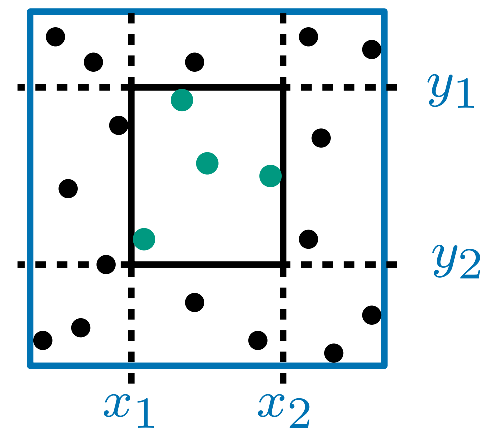


---

for $d = 2$, the lookup(x_1, x_2, y_1, y_2) operation

returns every point with

$$x_1 \leqslant x \leqslant x_2 \text{ and } y_1 \leqslant y \leqslant y_2.$$

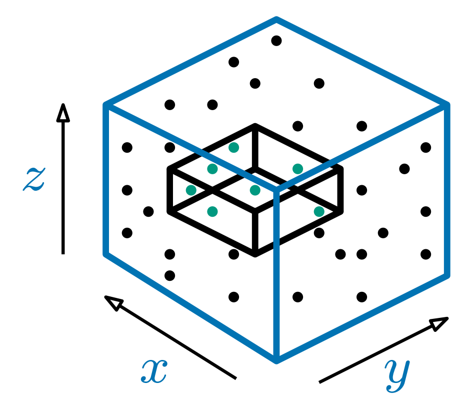


---

for $d = 3$, the lookup$(x_1, x_2, y_1, y_2, z_1, z_2)$ operation

returns every point with

$$x_1 \leqslant x \leqslant x_2,$$
$$y_1 \leqslant y \leqslant y_2 \text{ and}$$
$$z_1 \leqslant z \leqslant z_2.$$



Orthogonal range searching

- A **d-dimensional range searching data structure** stores n distinct points

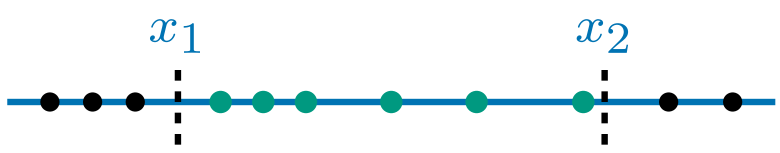
  each point has d coordinates

  (we assume d is a constant)

---

for $d = 1$, the lookup(x_1, x_2) operation

returns every point with $x_1 \leqslant x \leqslant x_2$.

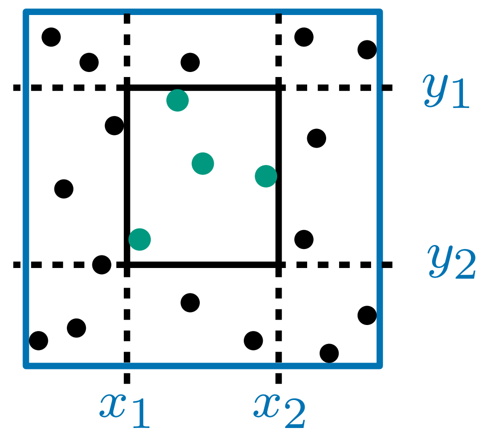


---

for $d = 2$, the lookup(x_1, x_2, y_1, y_2) operation

returns every point with

$x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



---

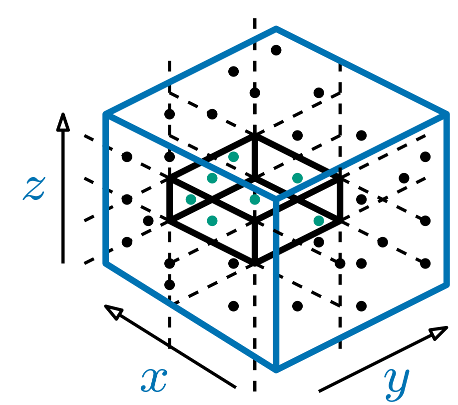
for $d = 3$, the lookup$(x_1, x_2, y_1, y_2, z_1, z_2)$ operation

returns every point with

$x_1 \leqslant x \leqslant x_2$,

$y_1 \leqslant y \leqslant y_2$ and

$z_1 \leqslant z \leqslant z_2$.



Orthogonal range searching

- A **d-dimensional range searching data structure** stores n distinct points

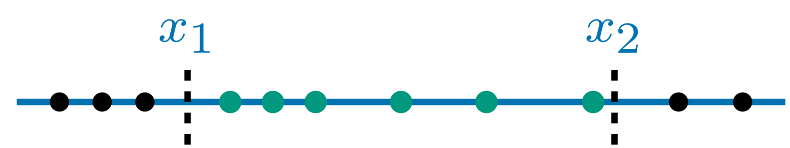
  each point has d coordinates

  (we assume d is a constant)

---

for $d = 1$, the lookup(x_1, x_2) operation

returns every point with $x_1 \leqslant x \leqslant x_2$.

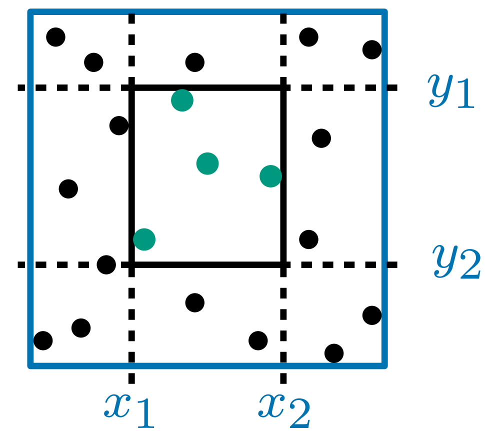


---

for $d = 2$, the lookup(x_1, x_2, y_1, y_2) operation

returns every point with

$x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



---

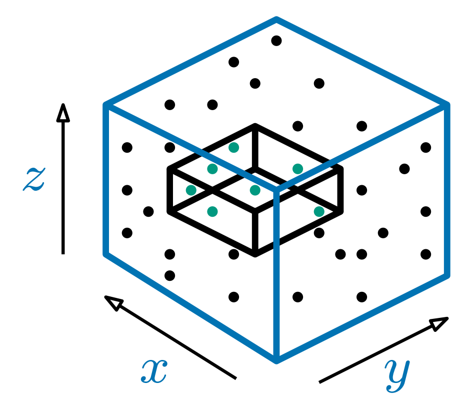
for $d = 3$, the lookup$(x_1, x_2, y_1, y_2, z_1, z_2)$ operation

returns every point with

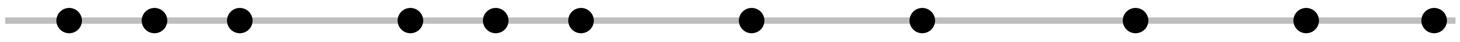
$x_1 \leqslant x \leqslant x_2$,

$y_1 \leqslant y \leqslant y_2$ and

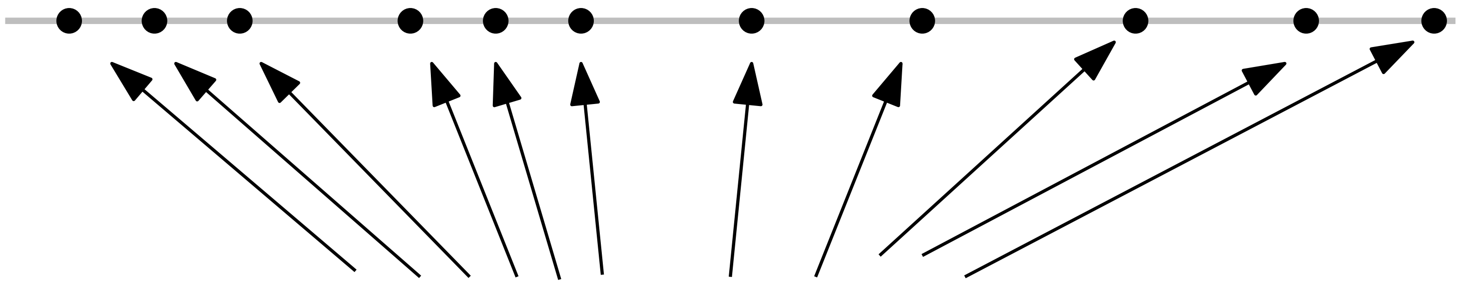
$z_1 \leqslant z \leqslant z_2$.



Starting simple. . . 1D range searching

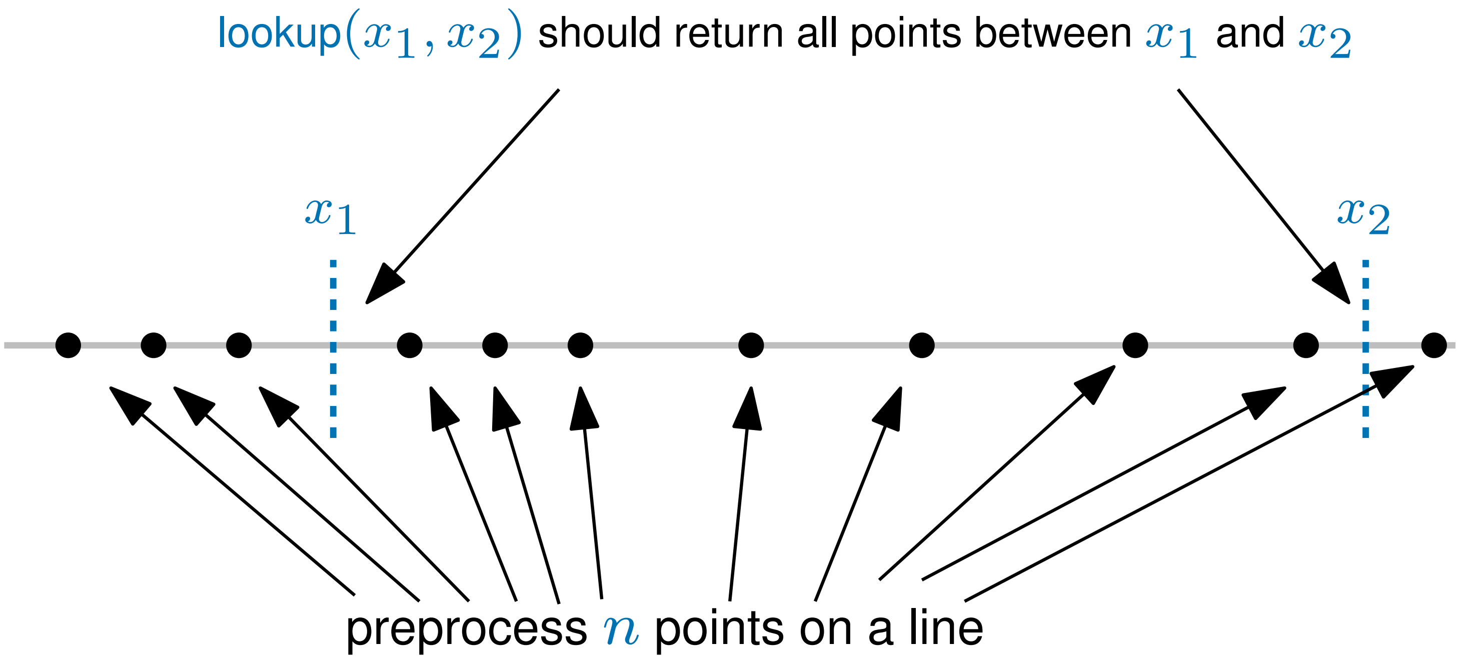


Starting simple. . . 1D range searching

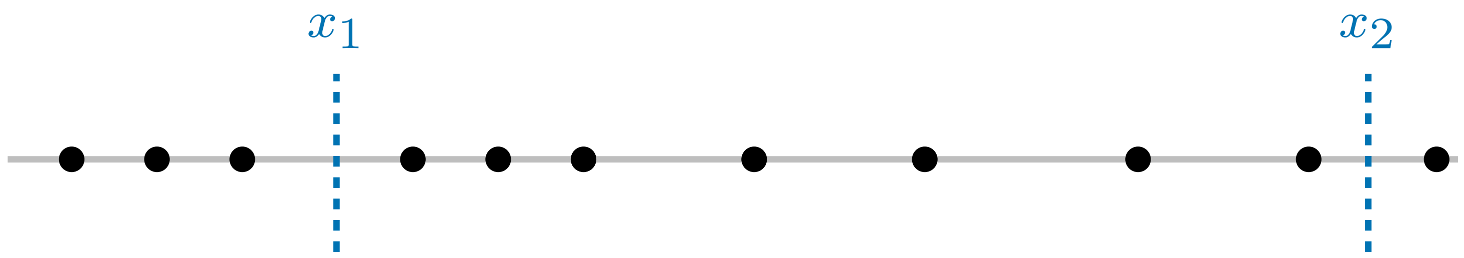


preprocess n points on a line

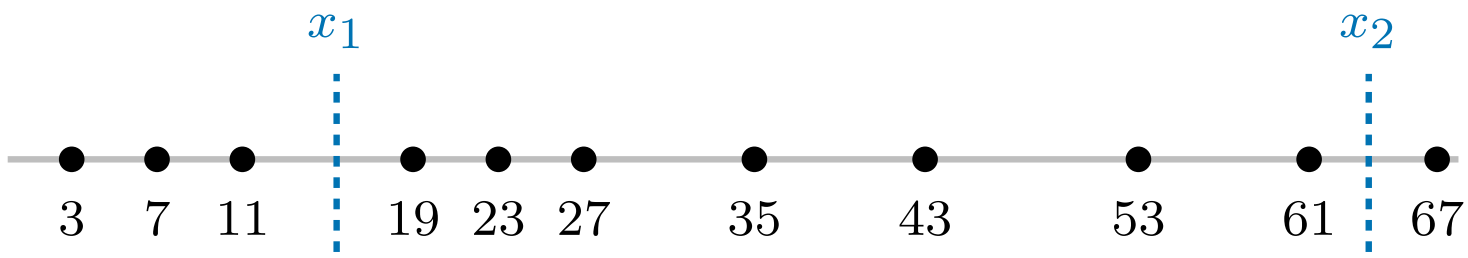
Starting simple. . . 1D range searching



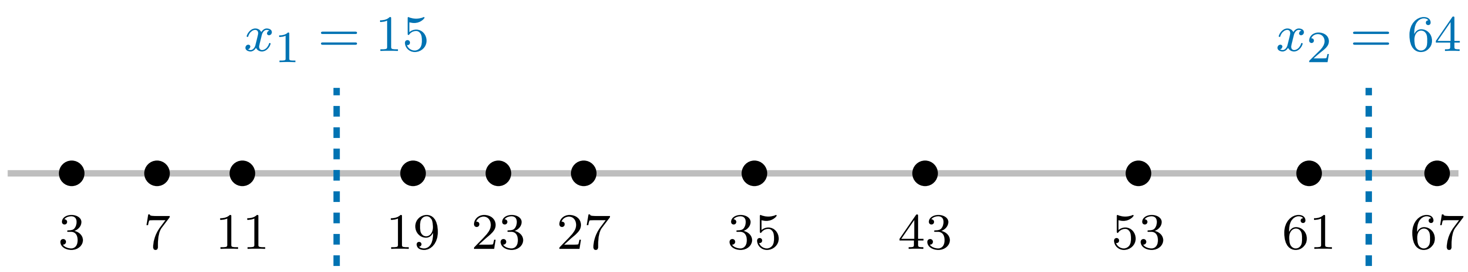
Starting simple. . . 1D range searching



Starting simple. . . 1D range searching



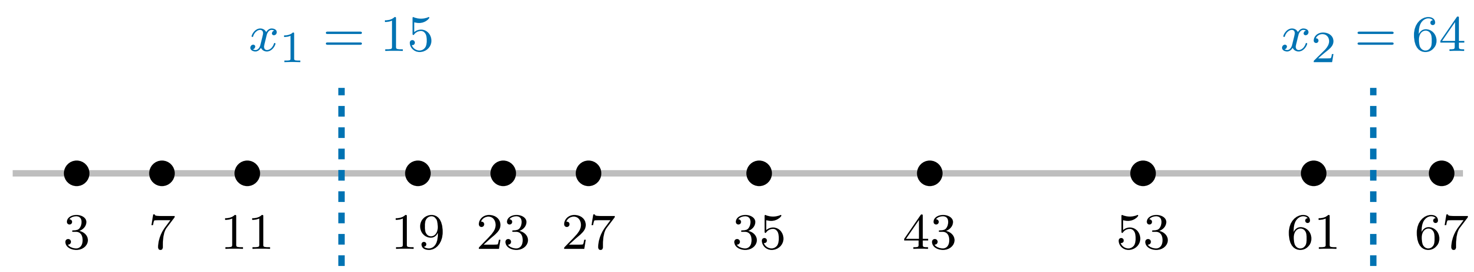
Starting simple. . . 1D range searching



Starting simple. . . 1D range searching

build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time



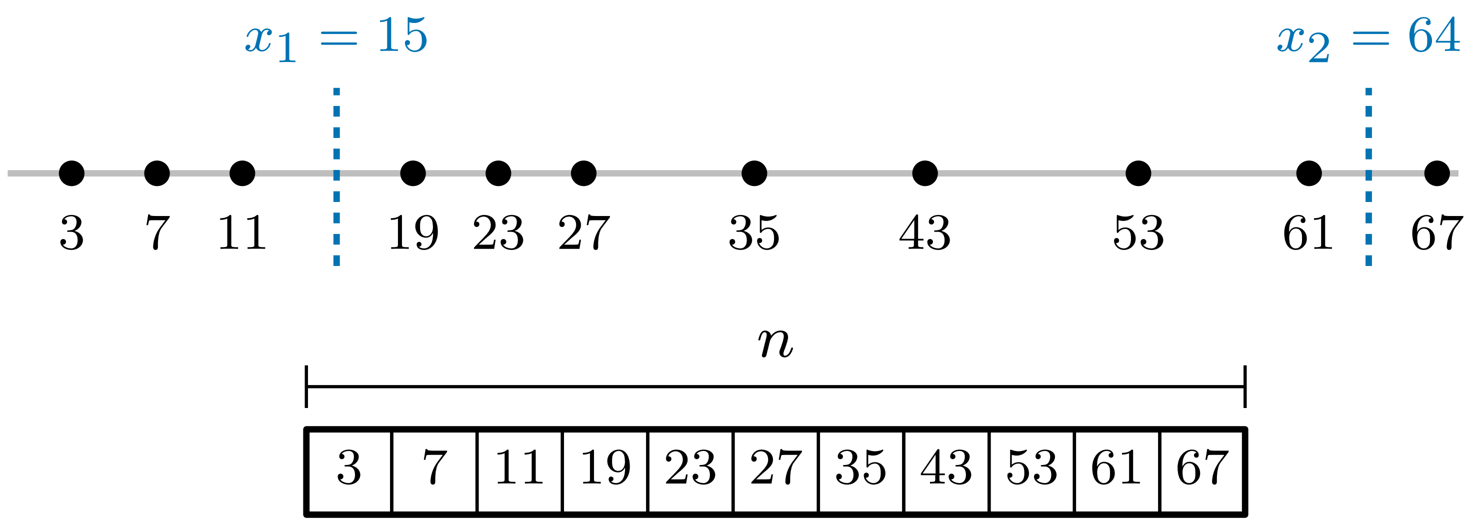
3	7	11	19	23	27	35	43	53	61	67

Starting simple. . . 1D range searching

build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space



Starting simple. . . 1D range searching

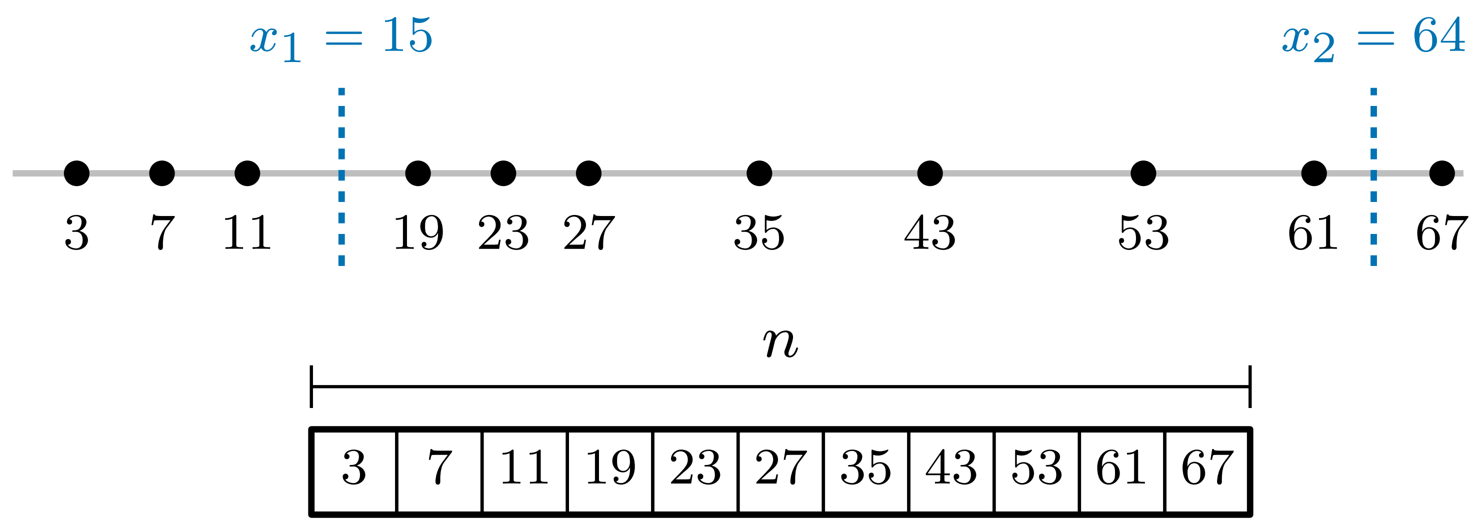
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple... 1D range searching

build a sorted array containing the x-coordinates

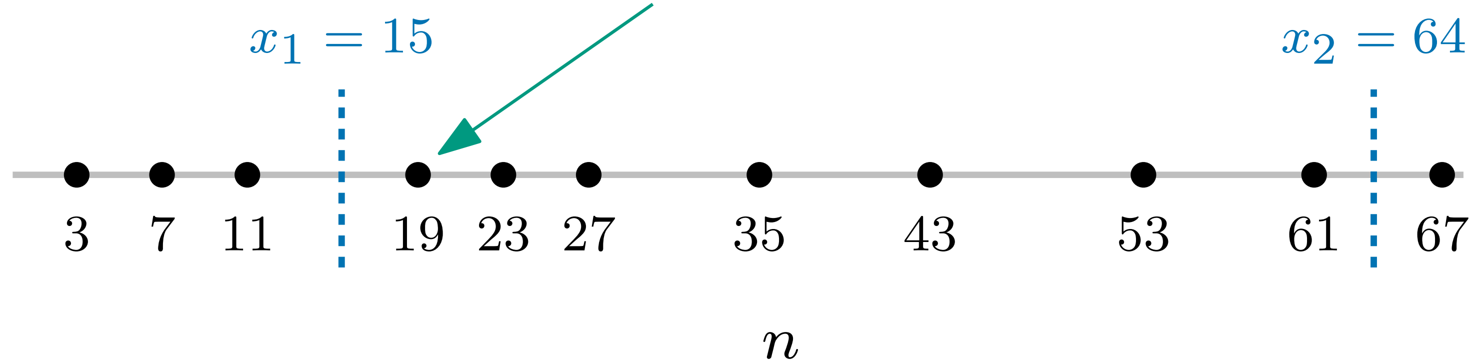
in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right

(i.e. the closest point to the right)



3	7	11	19	23	27	35	43	53	61	67

Starting simple. . . 1D range searching

build a sorted array containing the x-coordinates

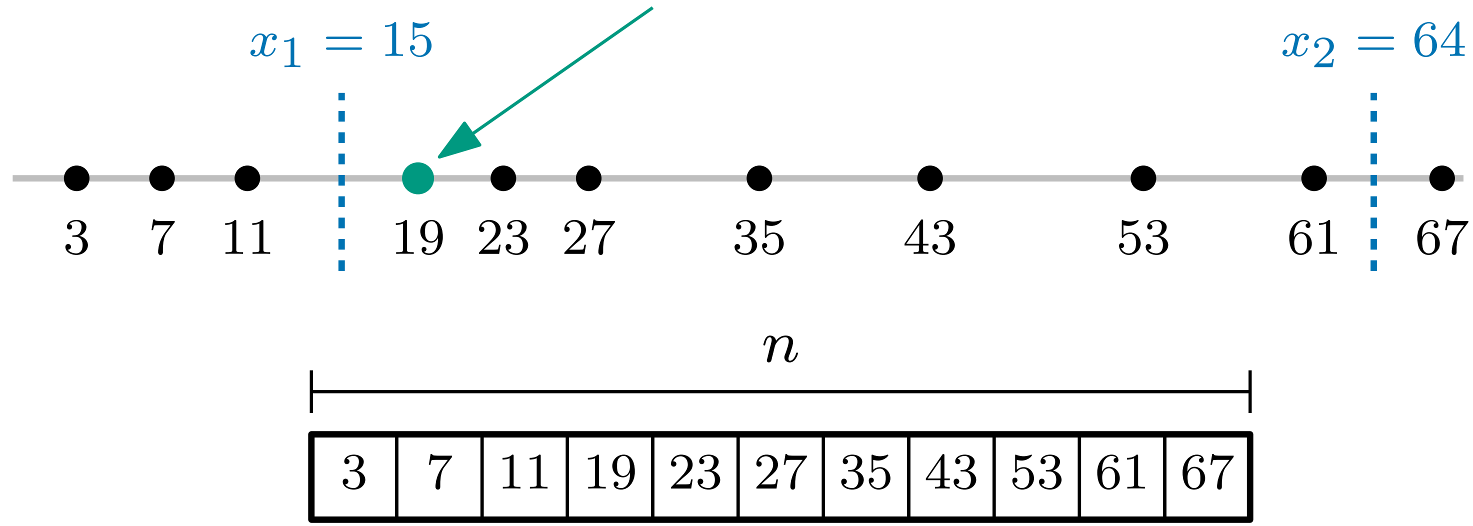
in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right

(i.e. the closest point to the right)



Starting simple. . . 1D range searching

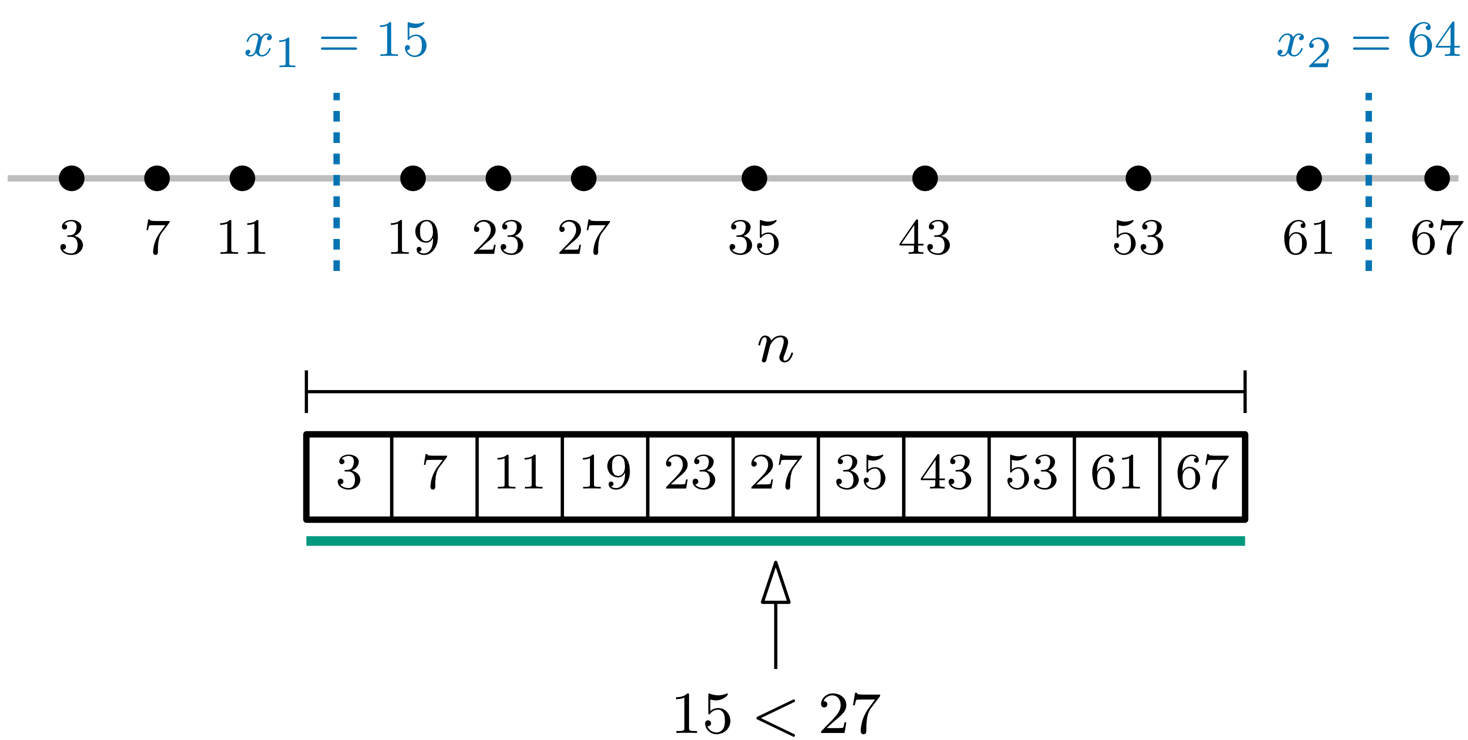
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple... 1D range searching

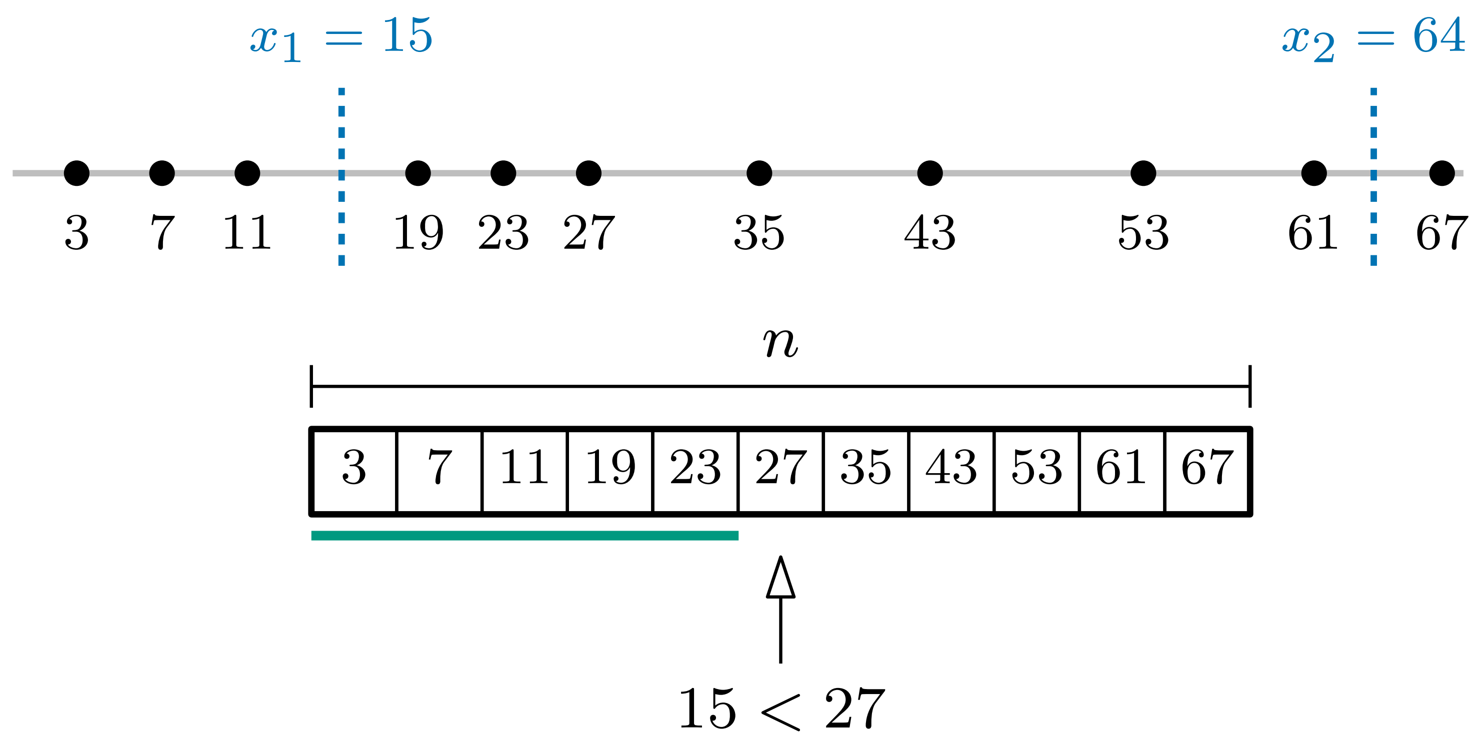
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

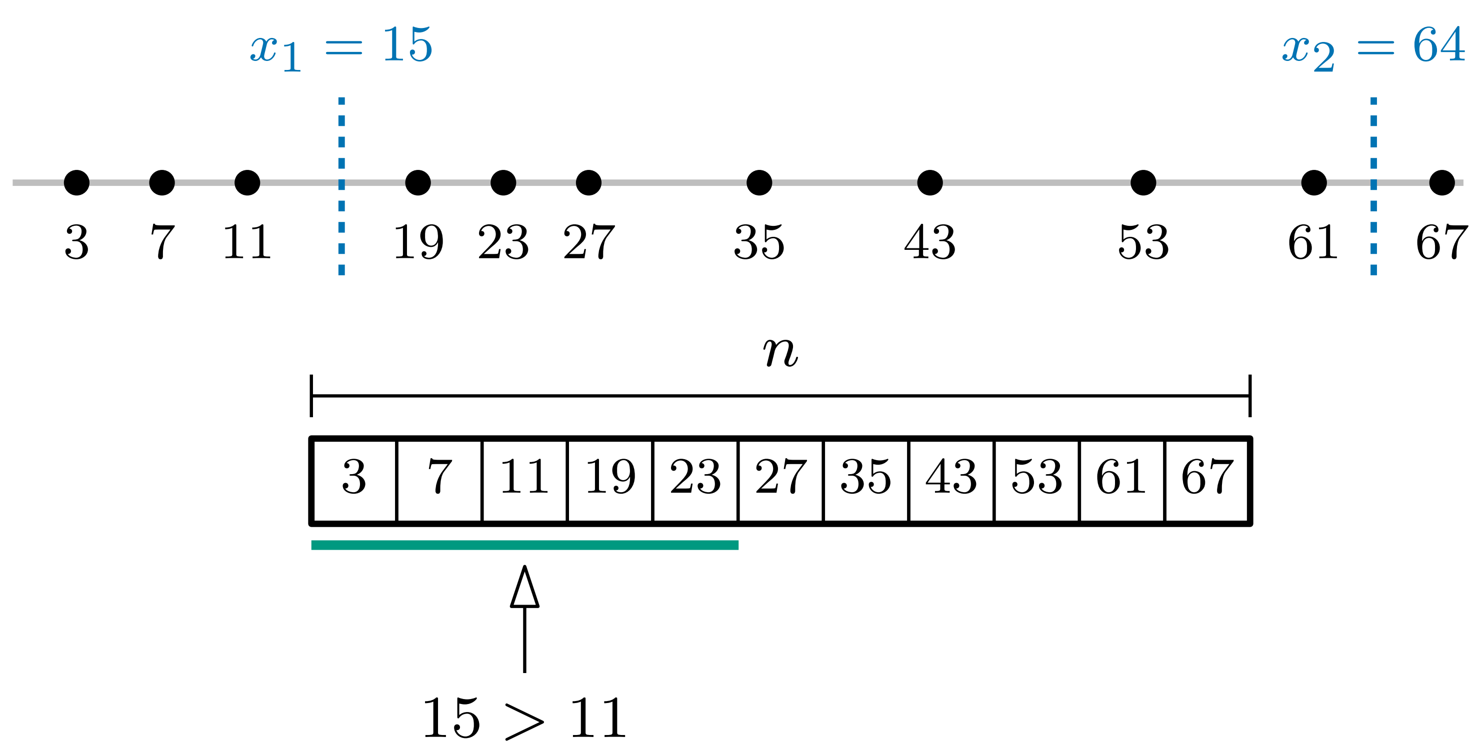
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right



Starting simple… 1D range searching

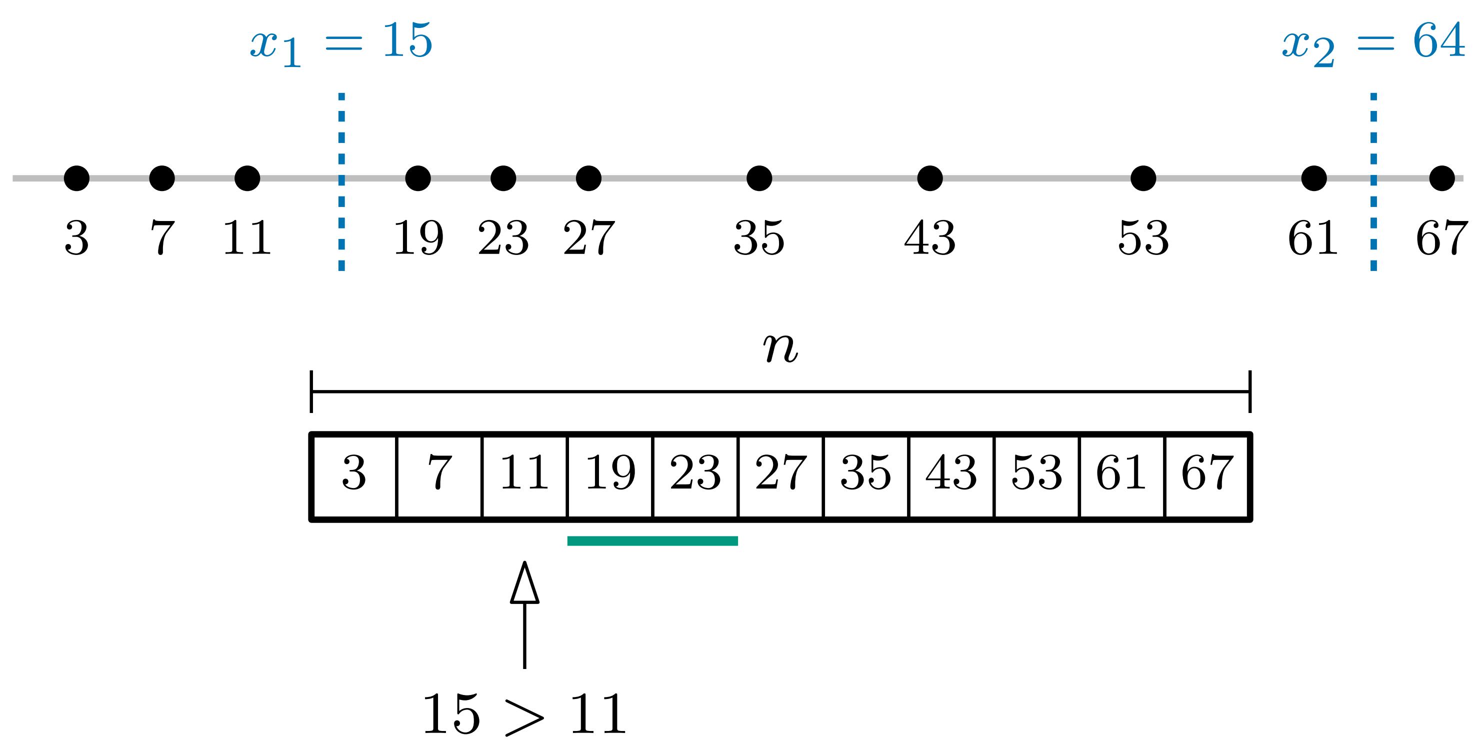
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)…

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

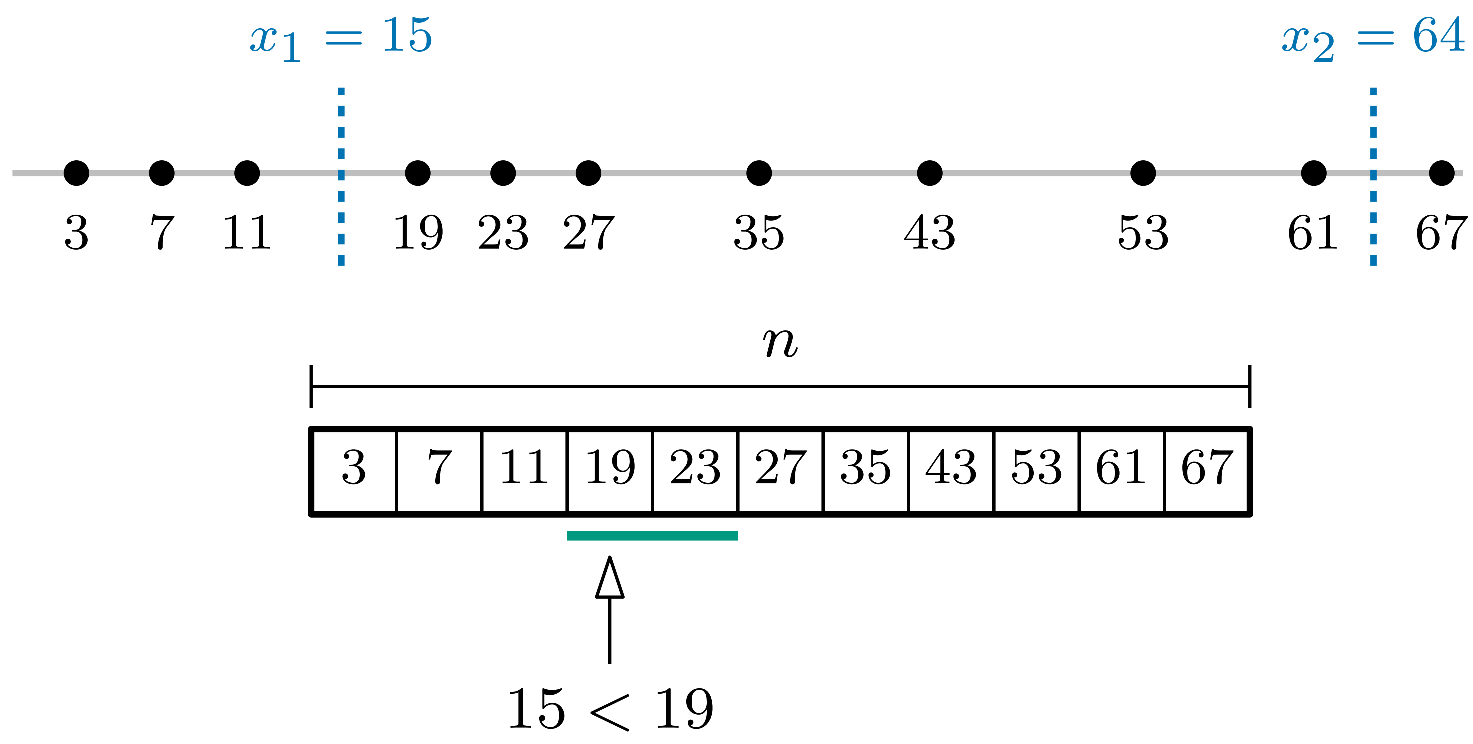
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple... 1D range searching

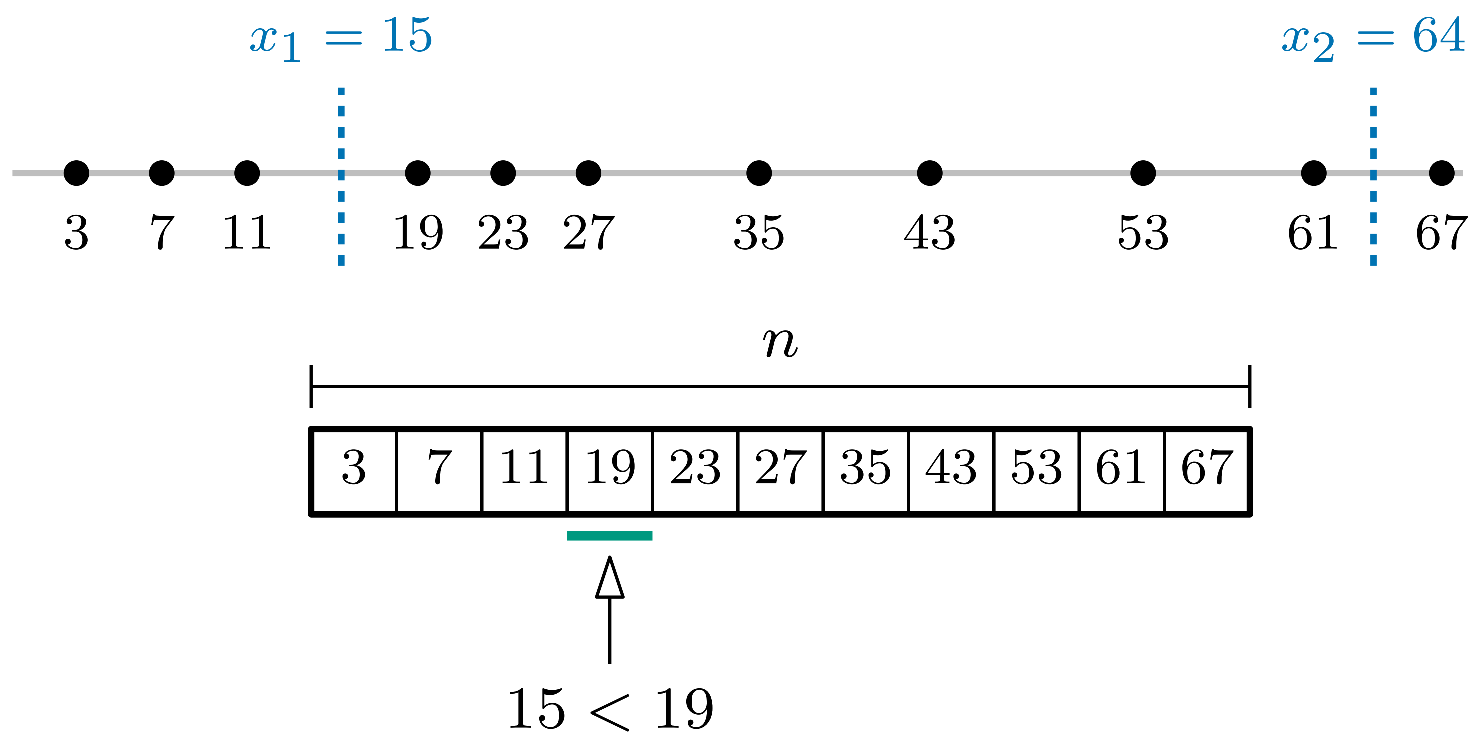
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

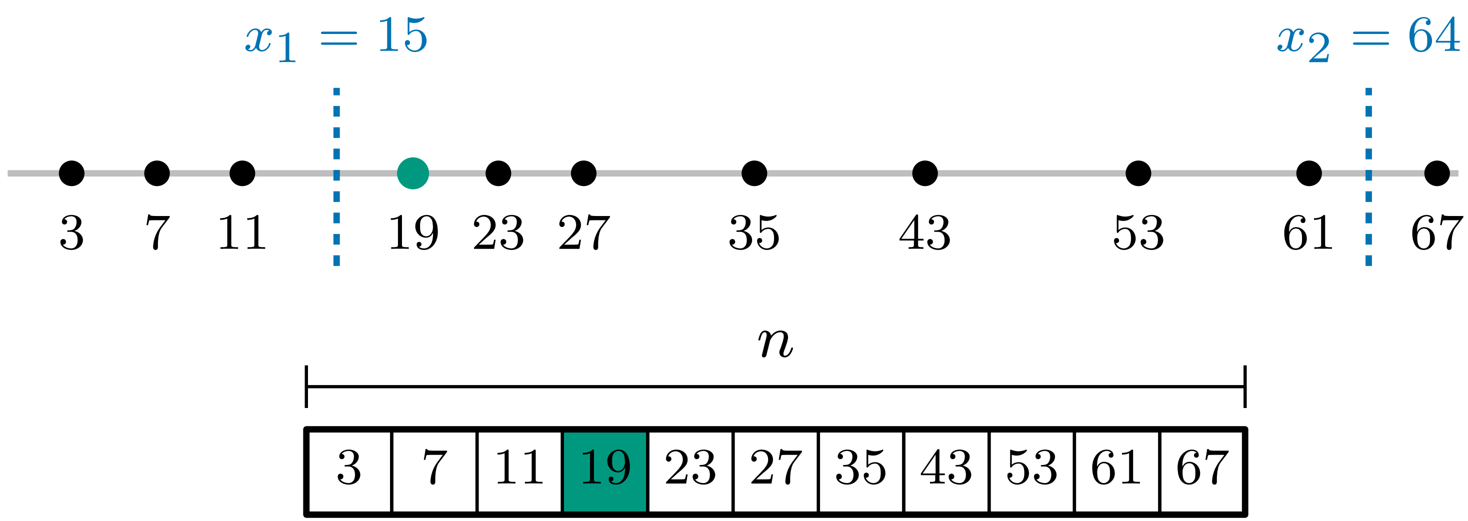
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

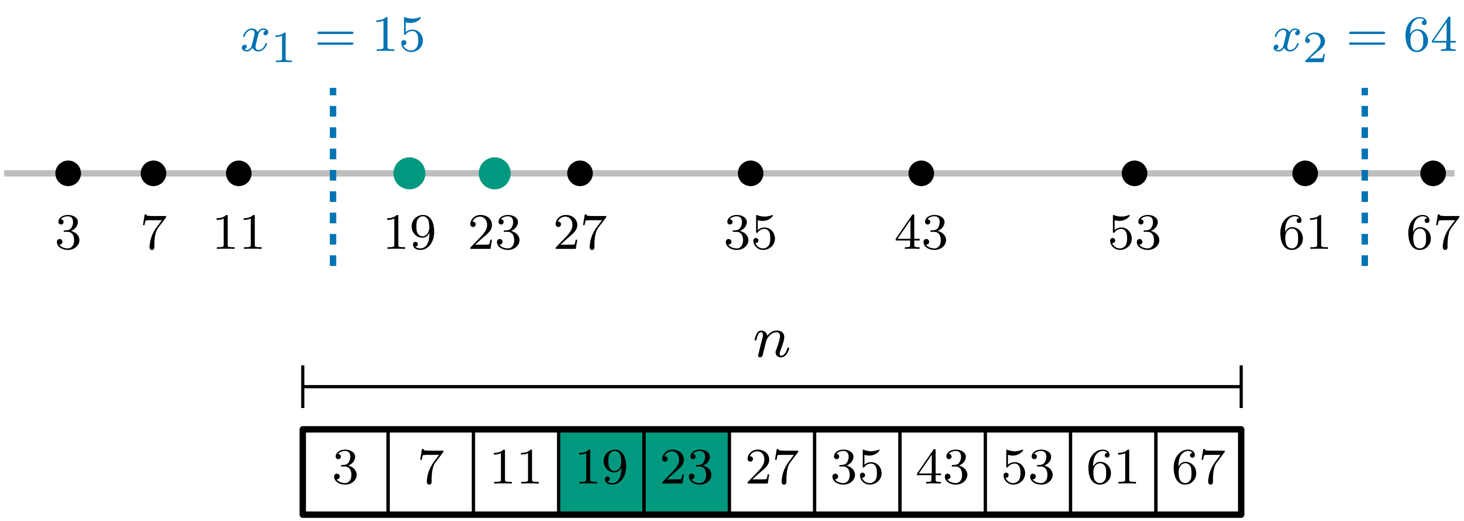
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

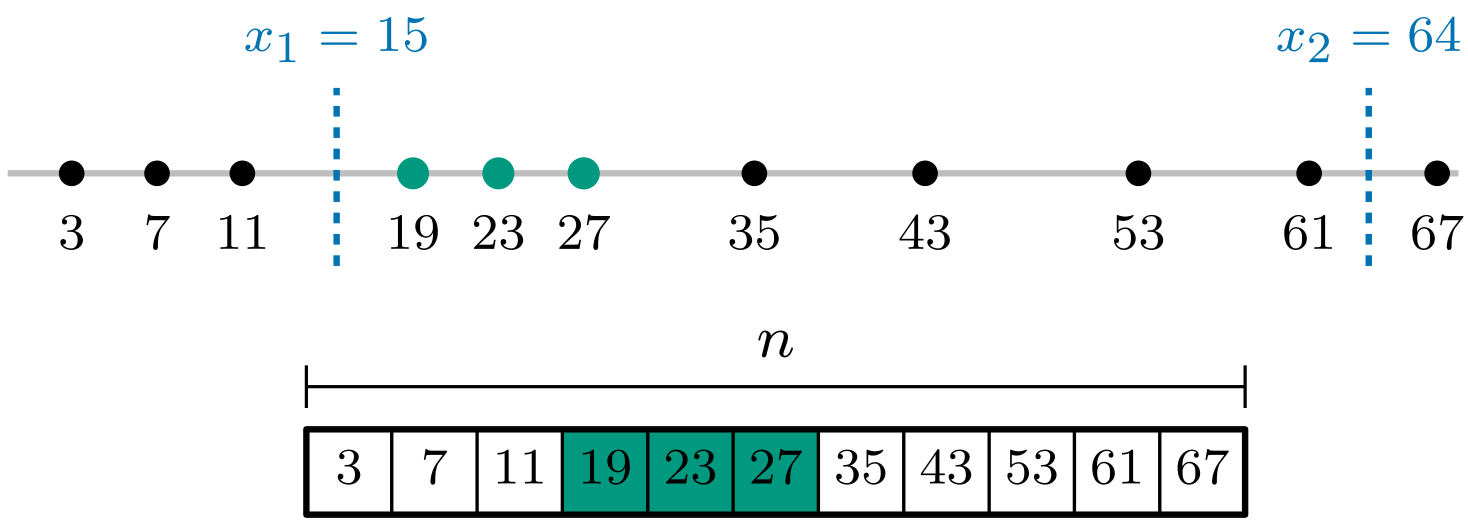
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then ‘walk’ right



Starting simple... 1D range searching

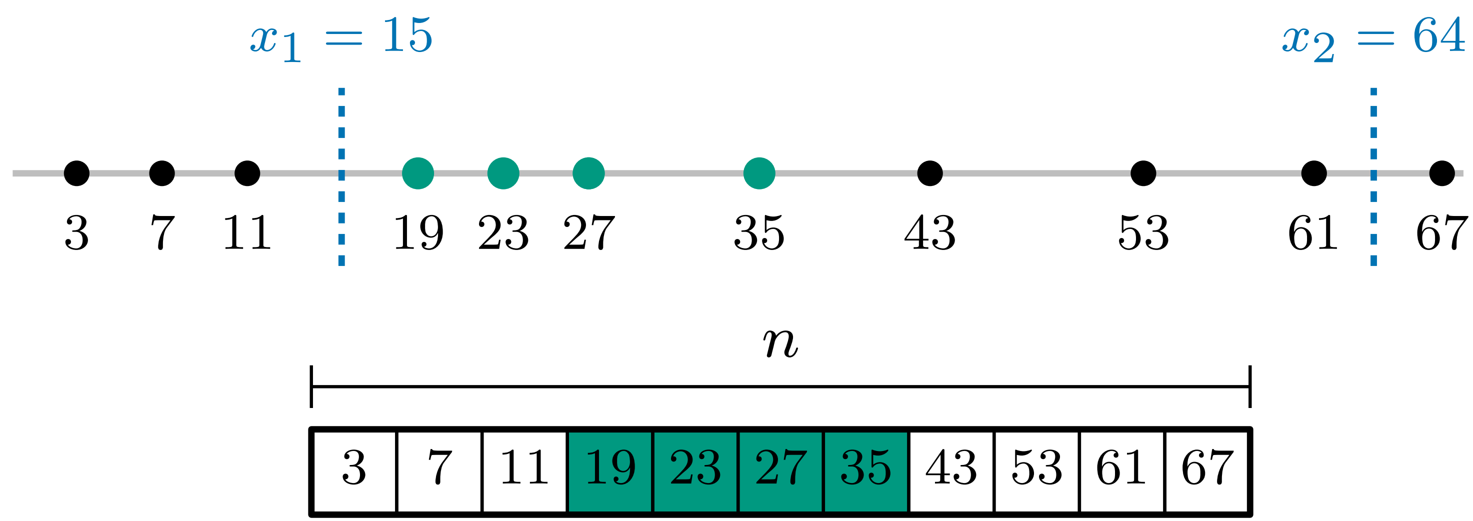
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

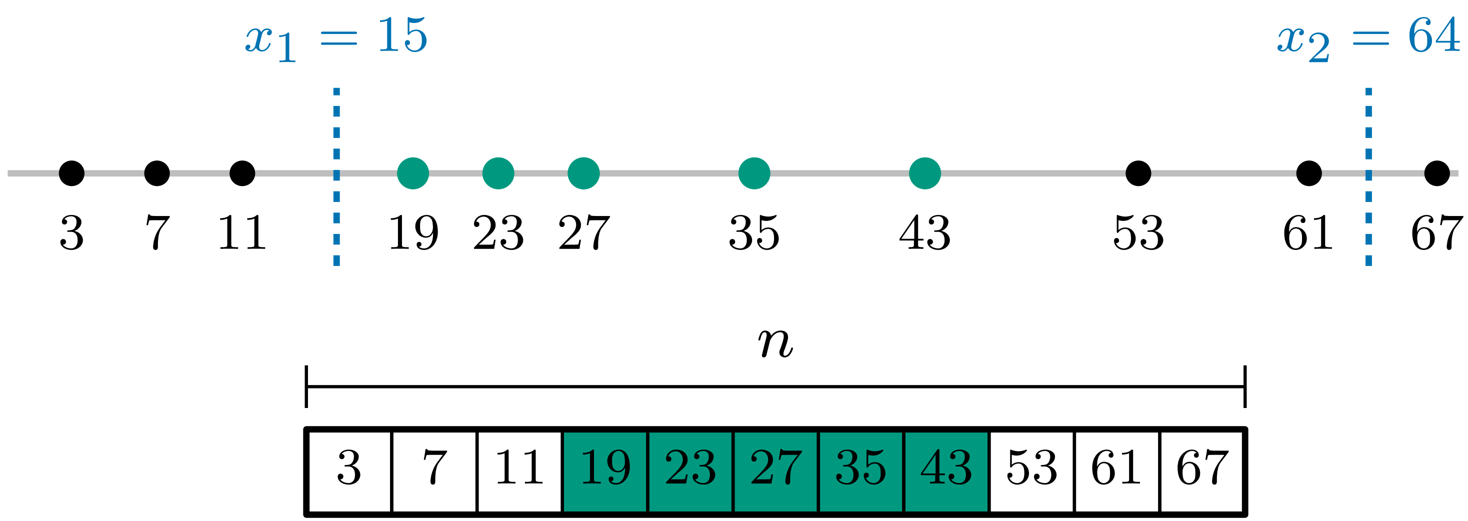
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

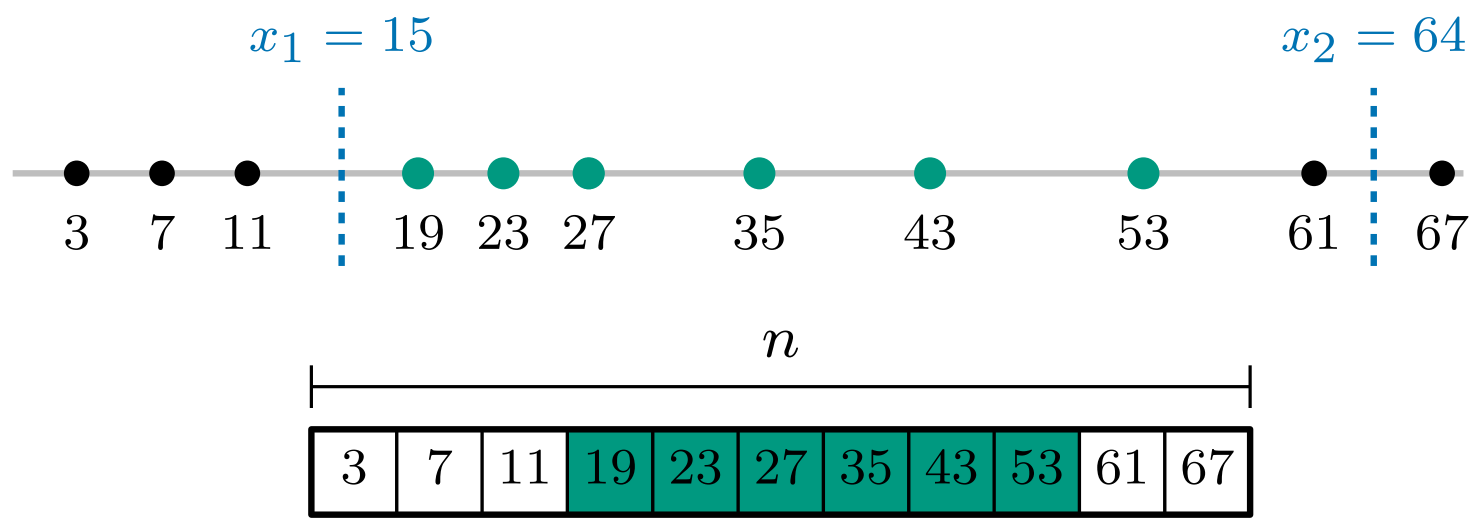
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right



Starting simple... 1D range searching

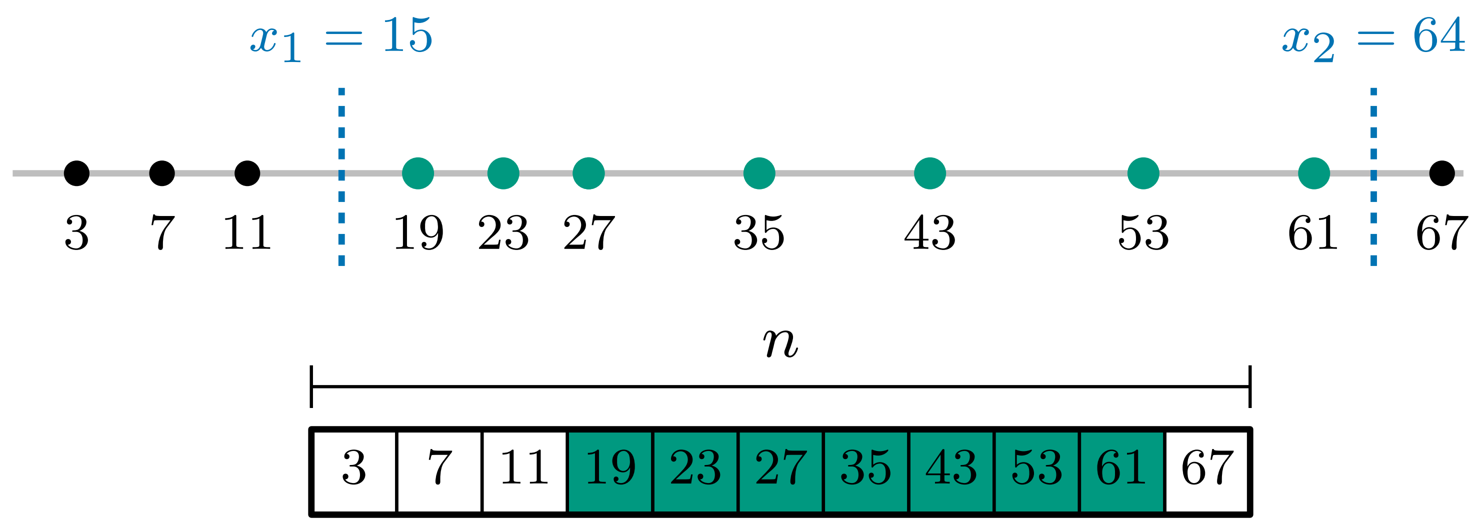
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

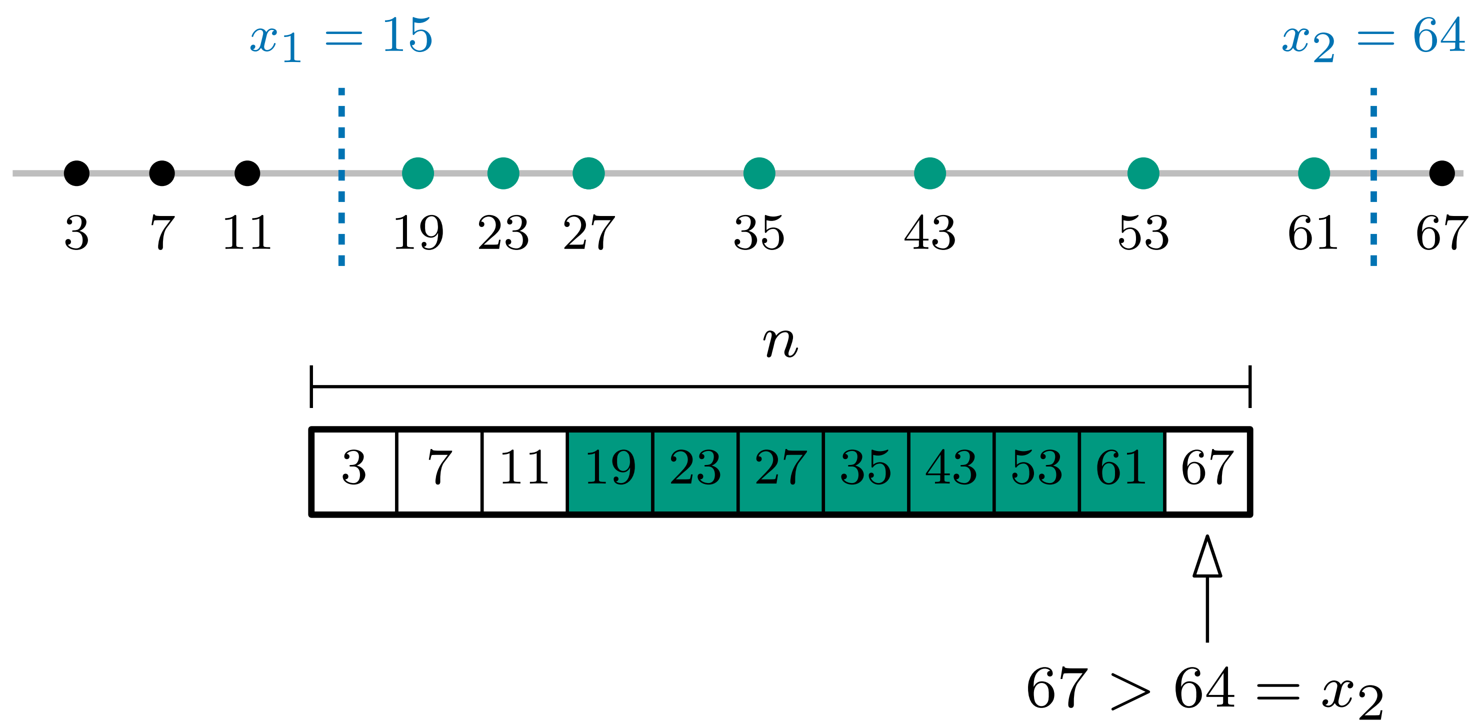
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

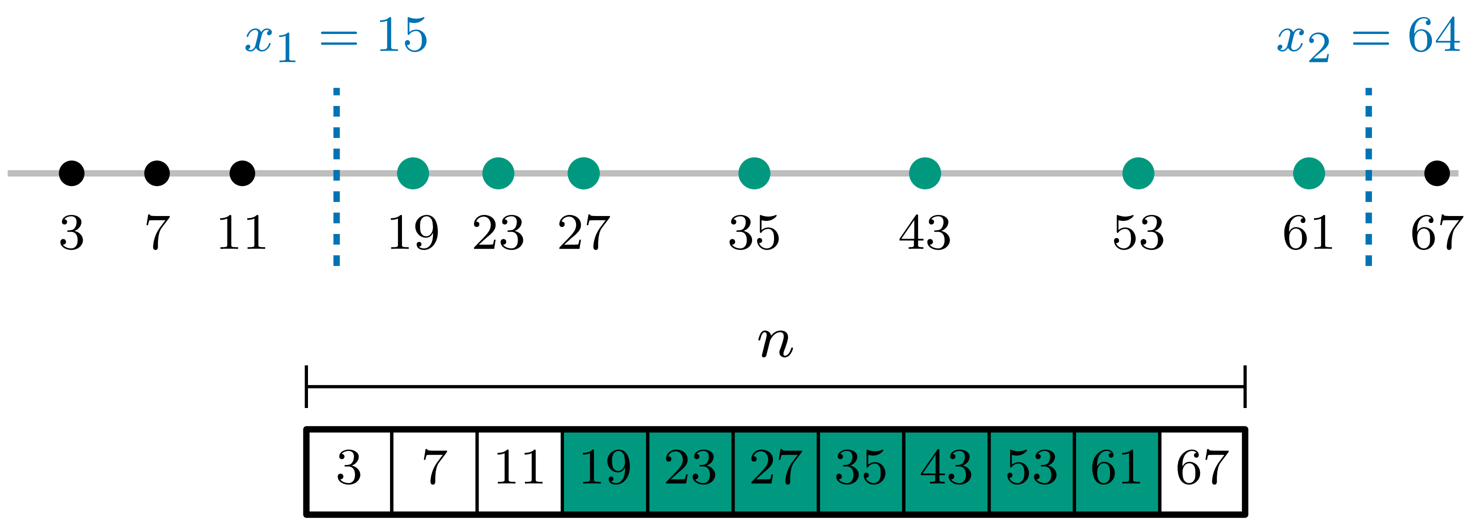
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2). . .

find the successor of x_1 by binary search and then 'walk' right



Starting simple. . . 1D range searching

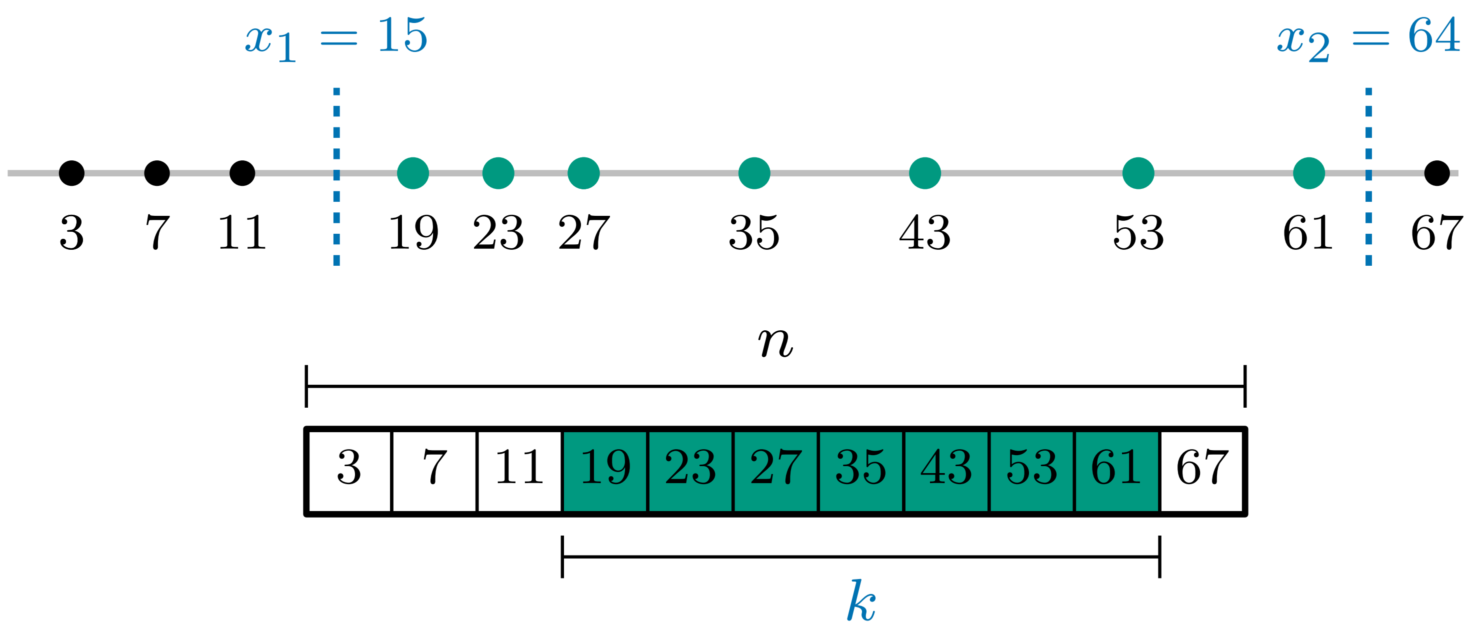
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup(x_1, x_2)...

find the successor of x_1 by binary search and then 'walk' right



lookups take $O(\log n + k)$ time (k is the number of points reported)

Starting simple. . . 1D range searching

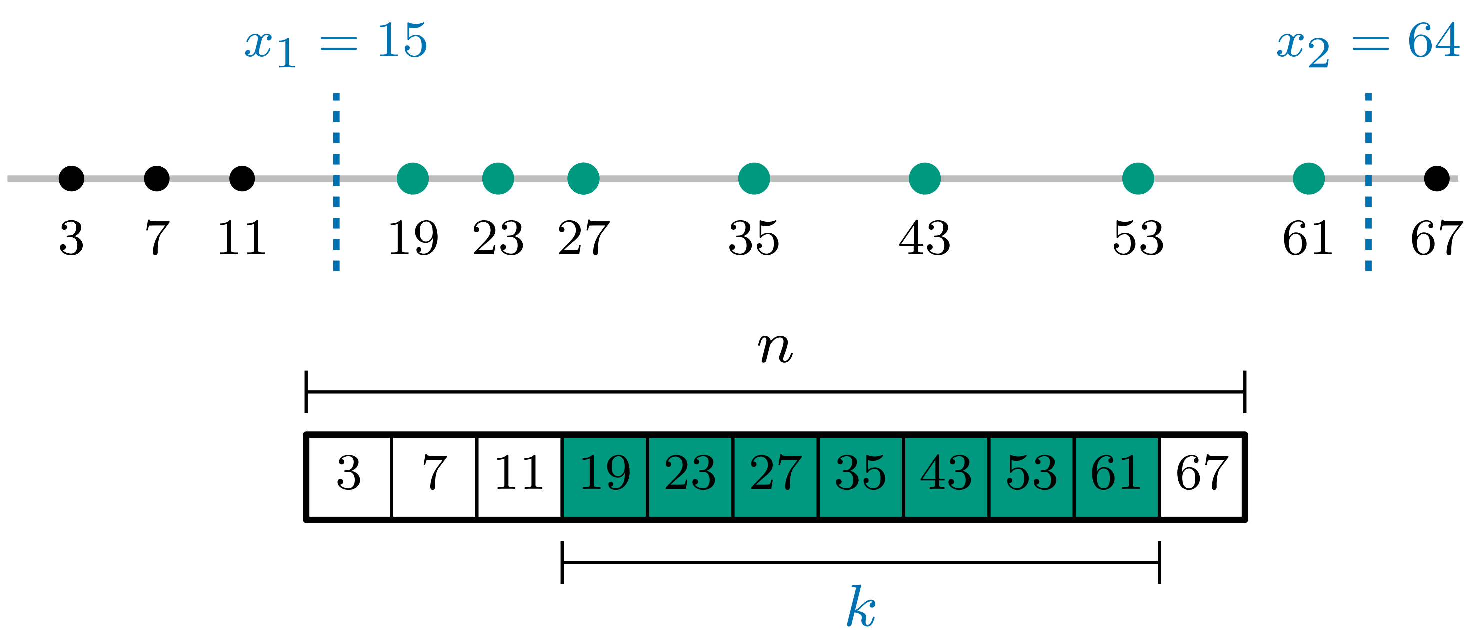
build a sorted array containing the x-coordinates

in $O(n \log n)$ preprocessing (prep.) time

and $O(n)$ space

to perform lookup$(x_1, x_2)\ldots$

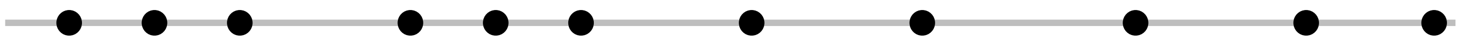
find the successor of x_1 by binary search and then 'walk' right



lookups take $O(\log n + k)$ time (k is the number of points reported)

this is called being 'output sensitive'

Starting simple. . . 1D range searching

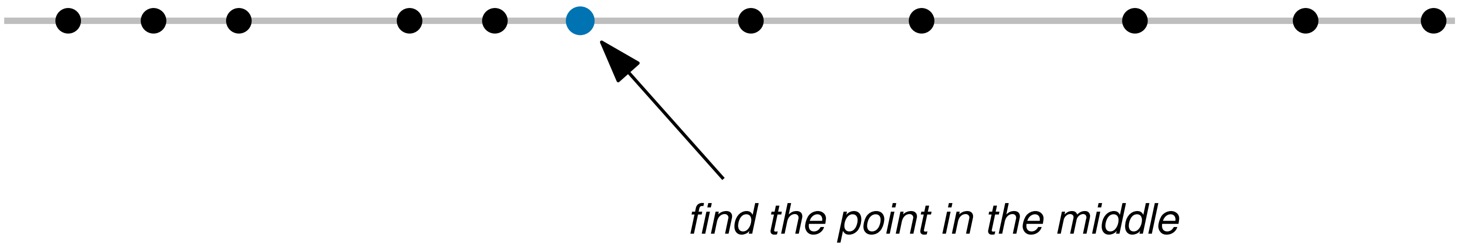


Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .

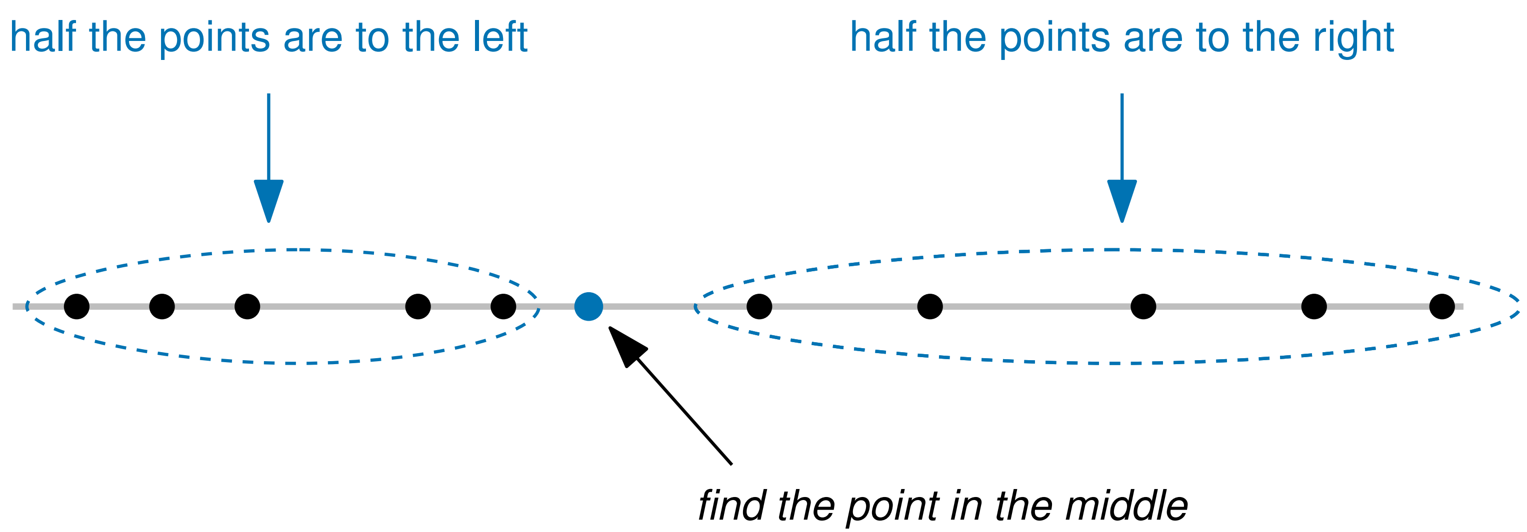
Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



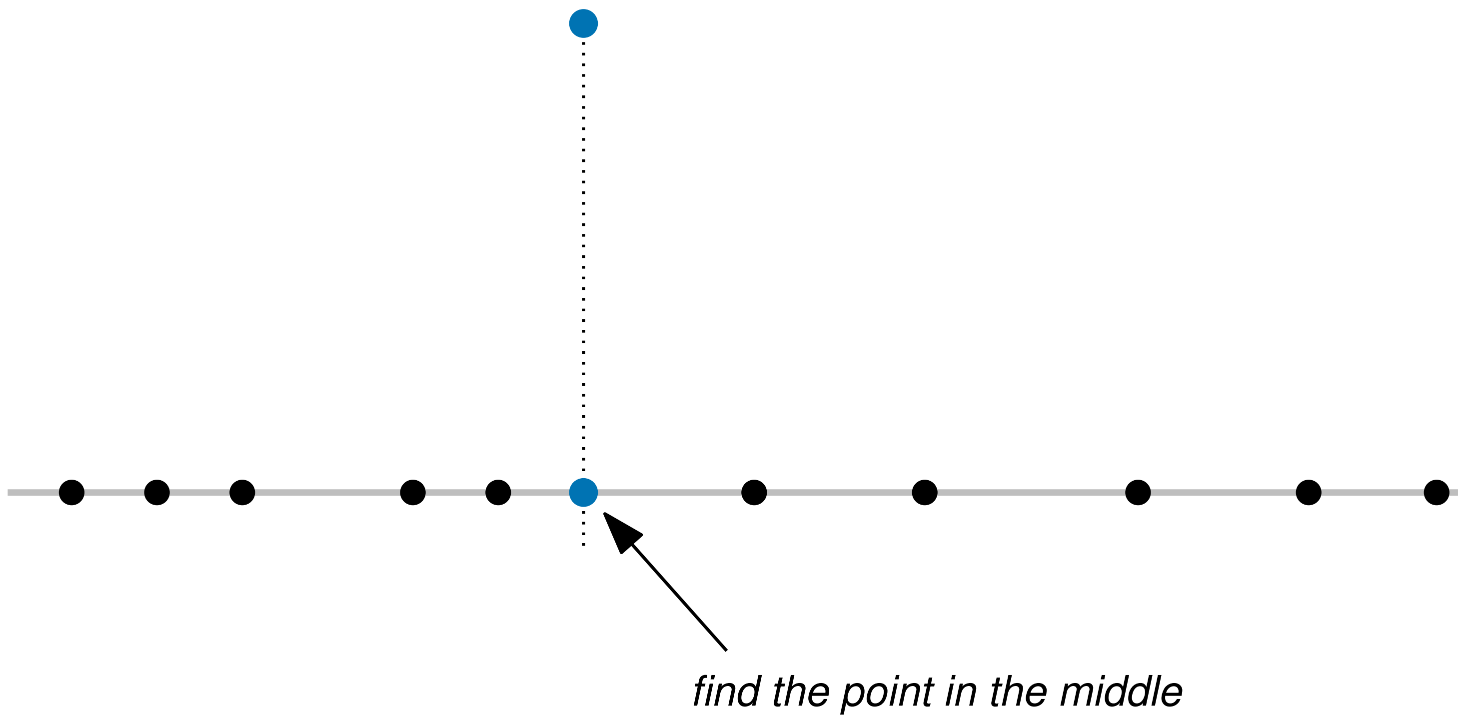
Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



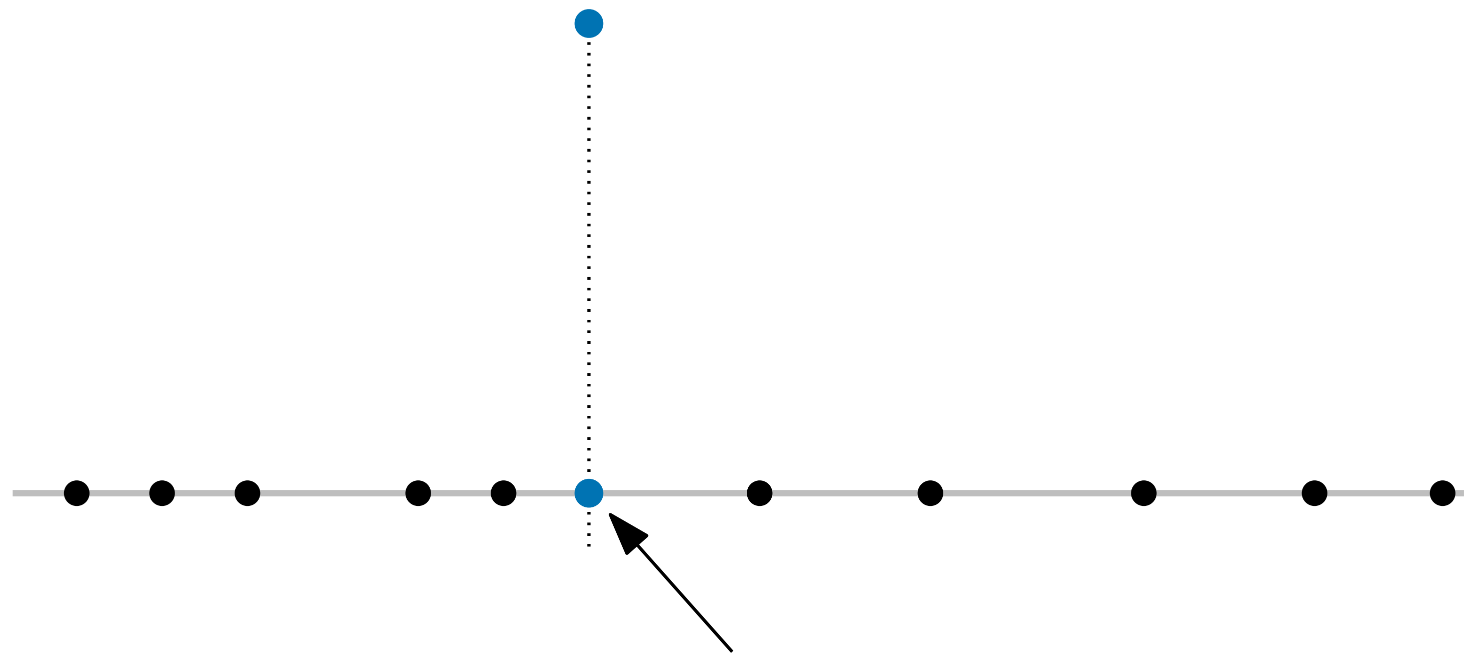
Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .

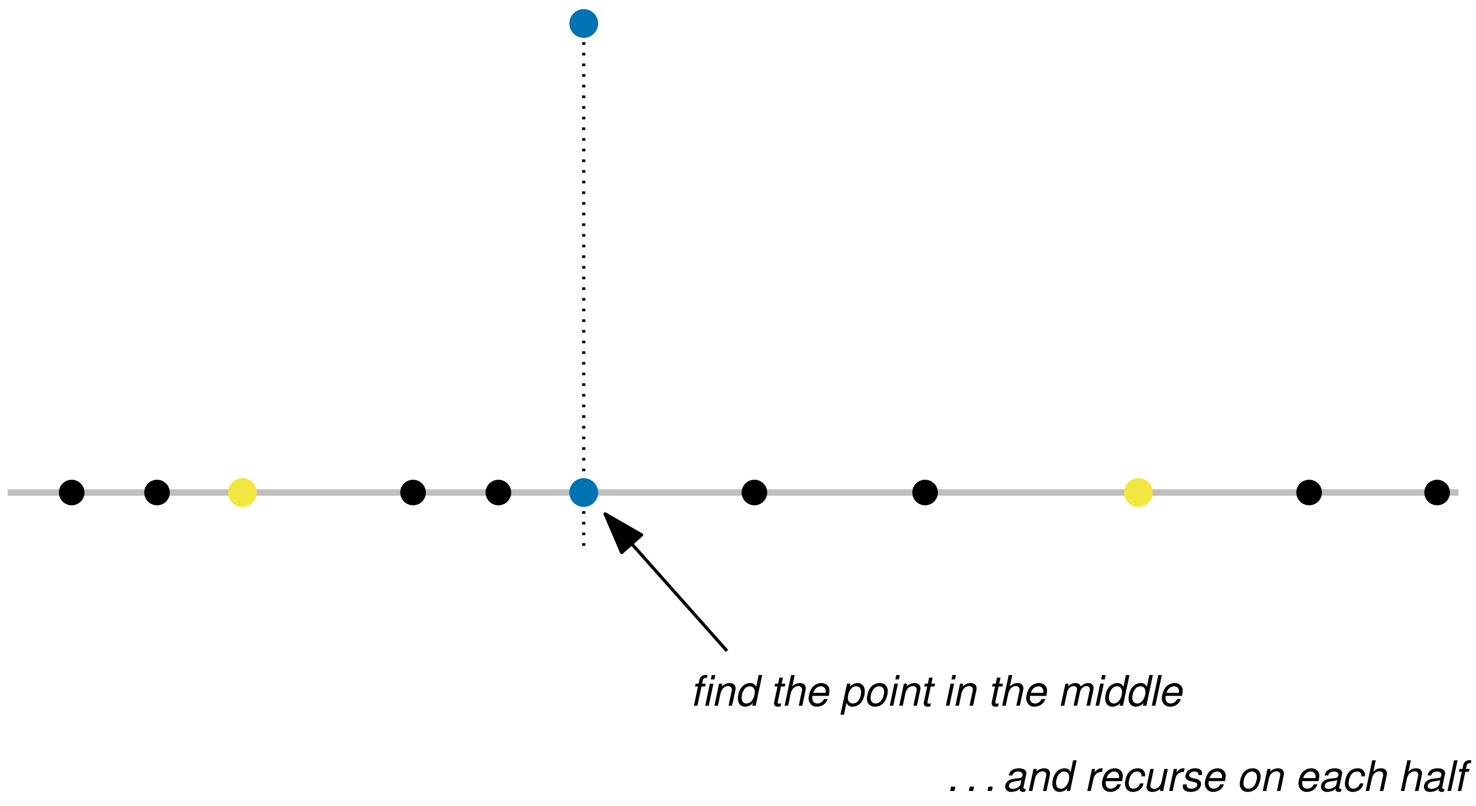


find the point in the middle

. . . and recurse on each half

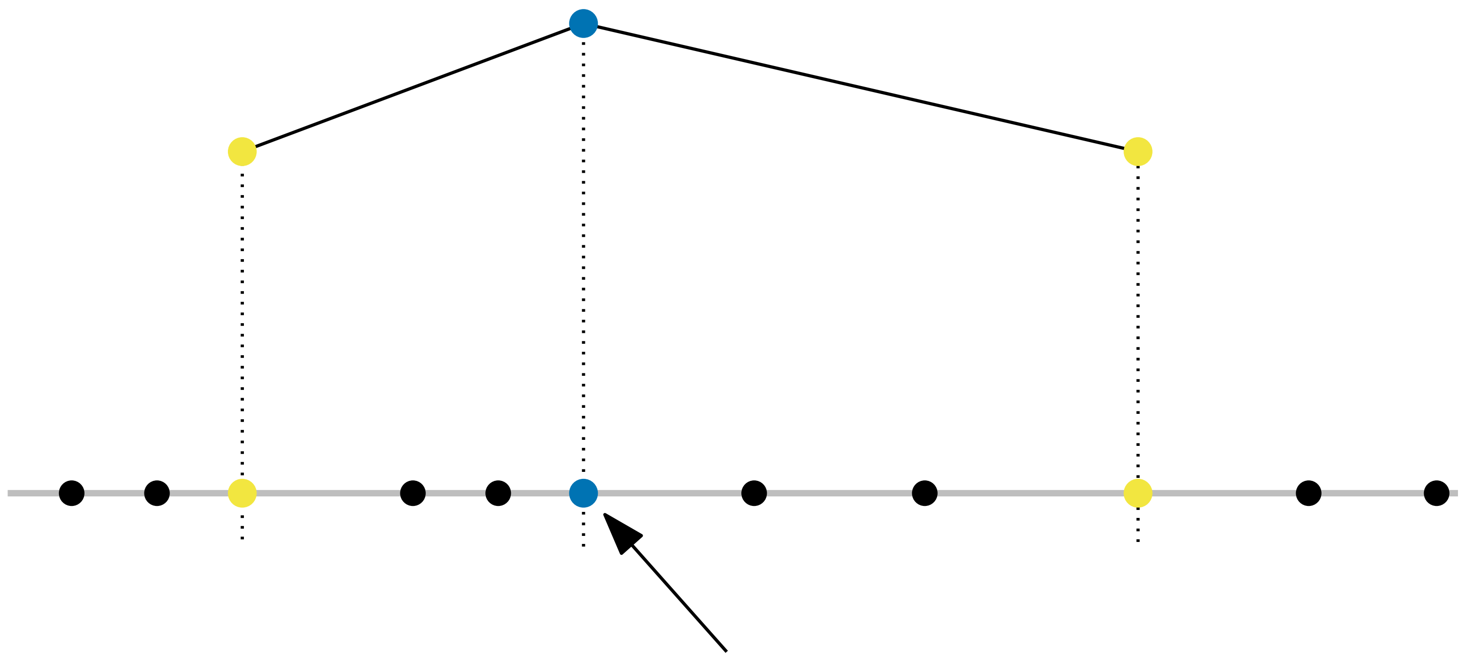
Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .

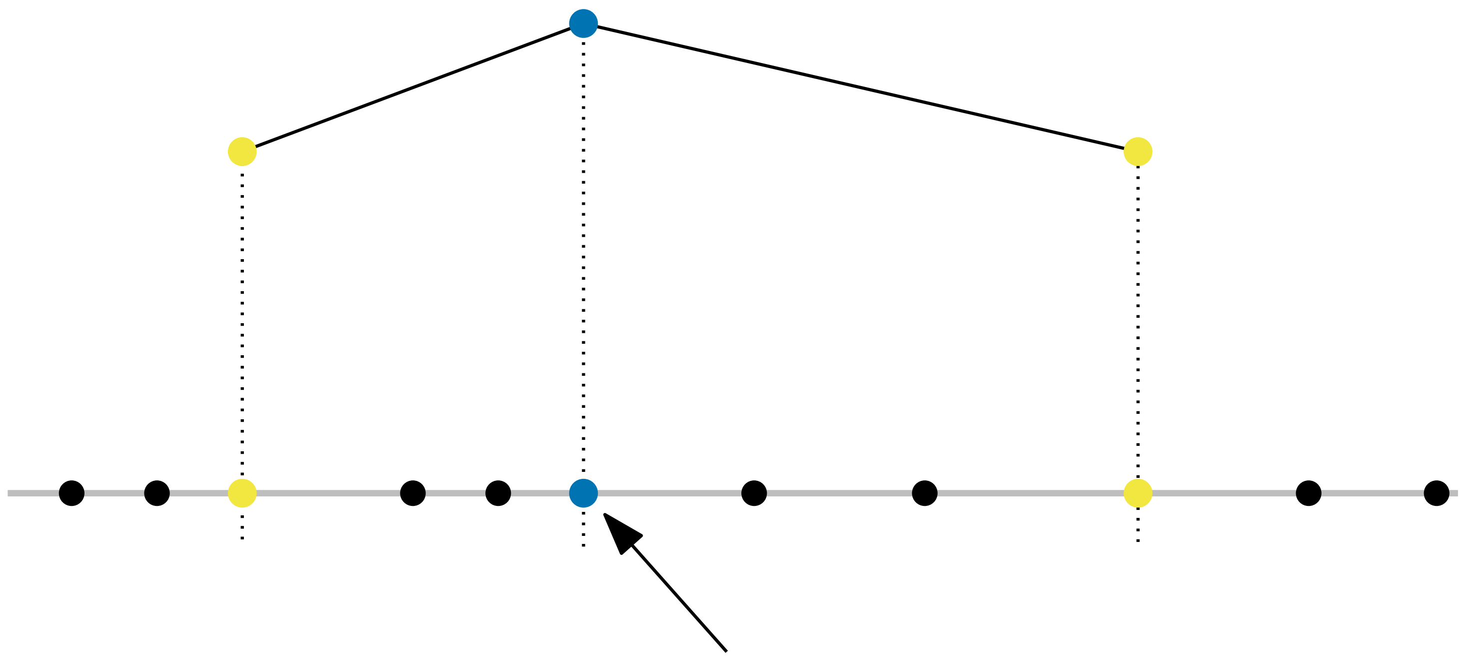


find the point in the middle

. . . and recurse on each half

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



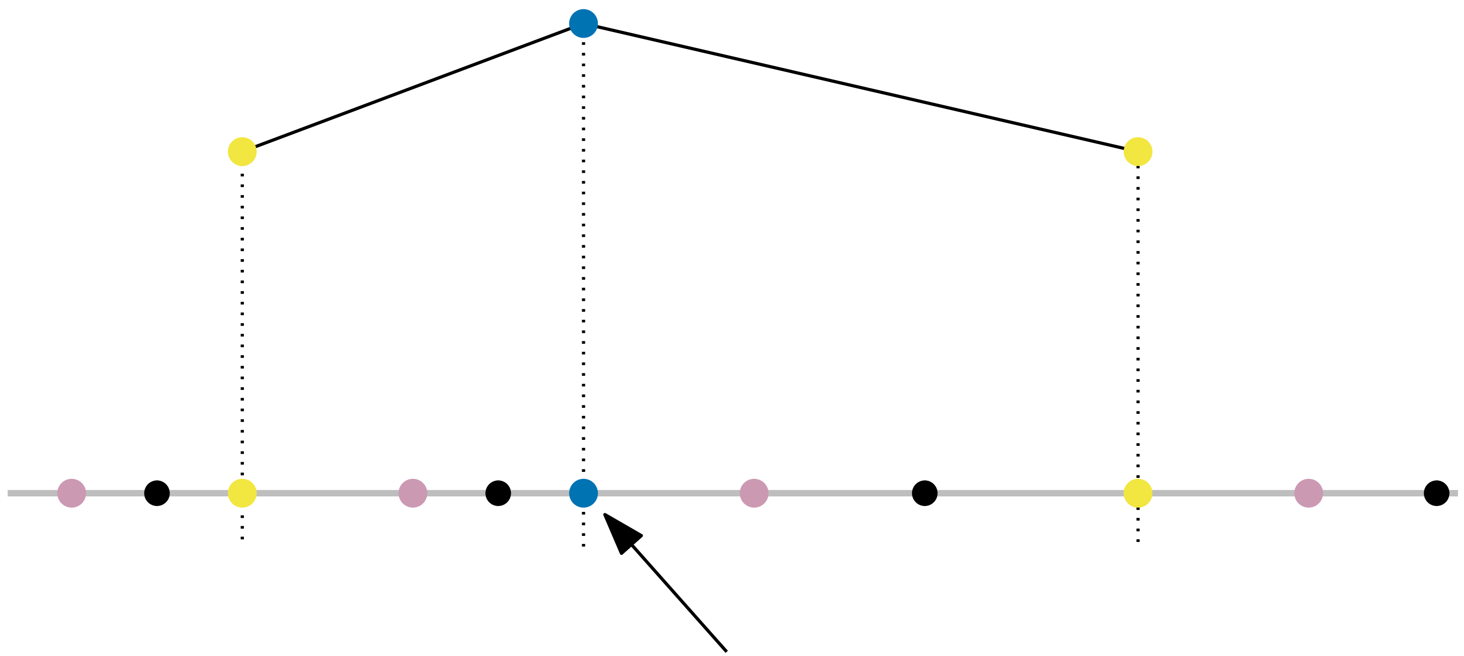
find the point in the middle

. . . and recurse on each half

(in a tie, pick the left)

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



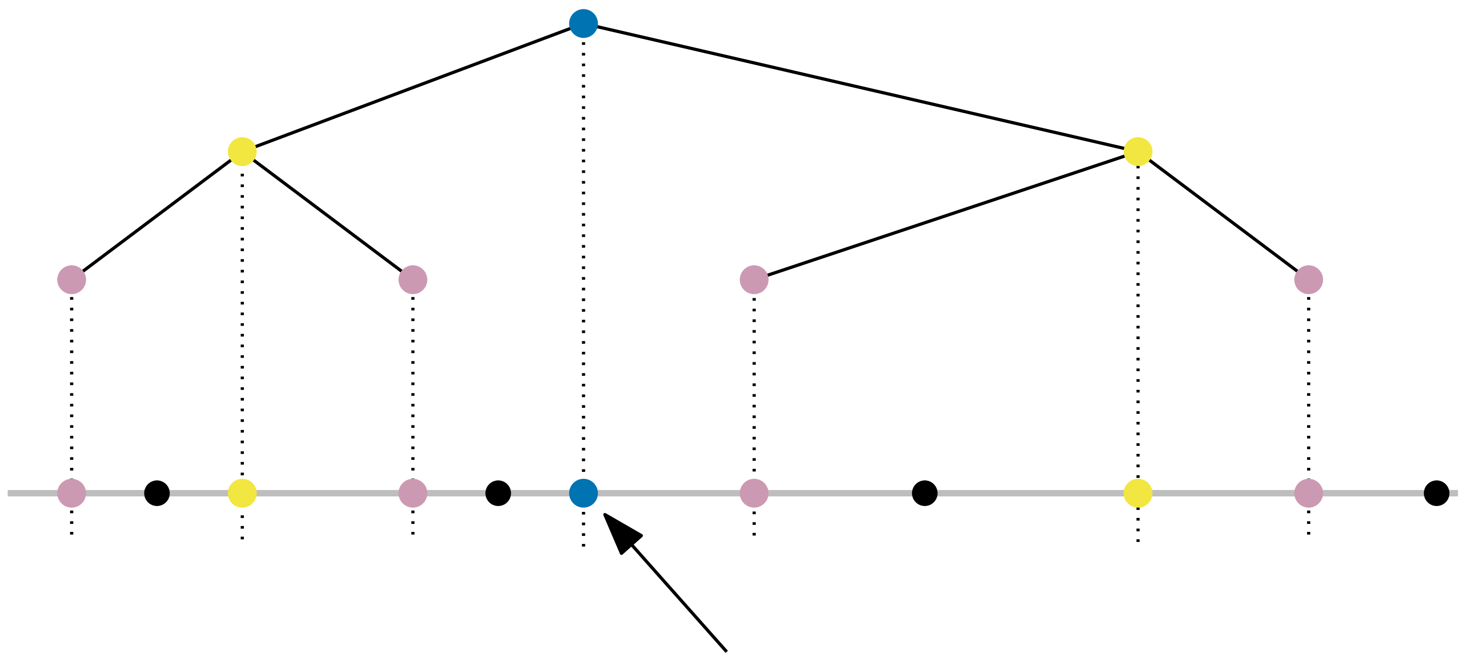
find the point in the middle

. . . and recurse on each half

(in a tie, pick the left)

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



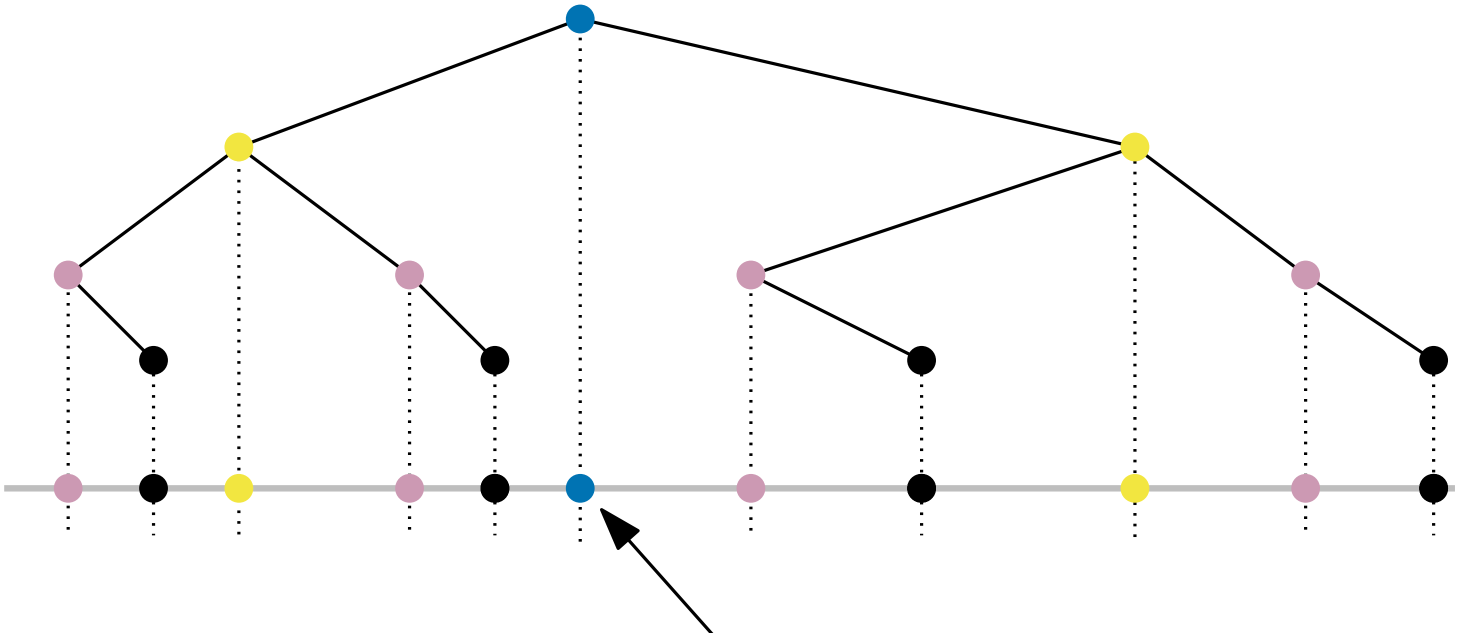
find the point in the middle

. . . and recurse on each half

(in a tie, pick the left)

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



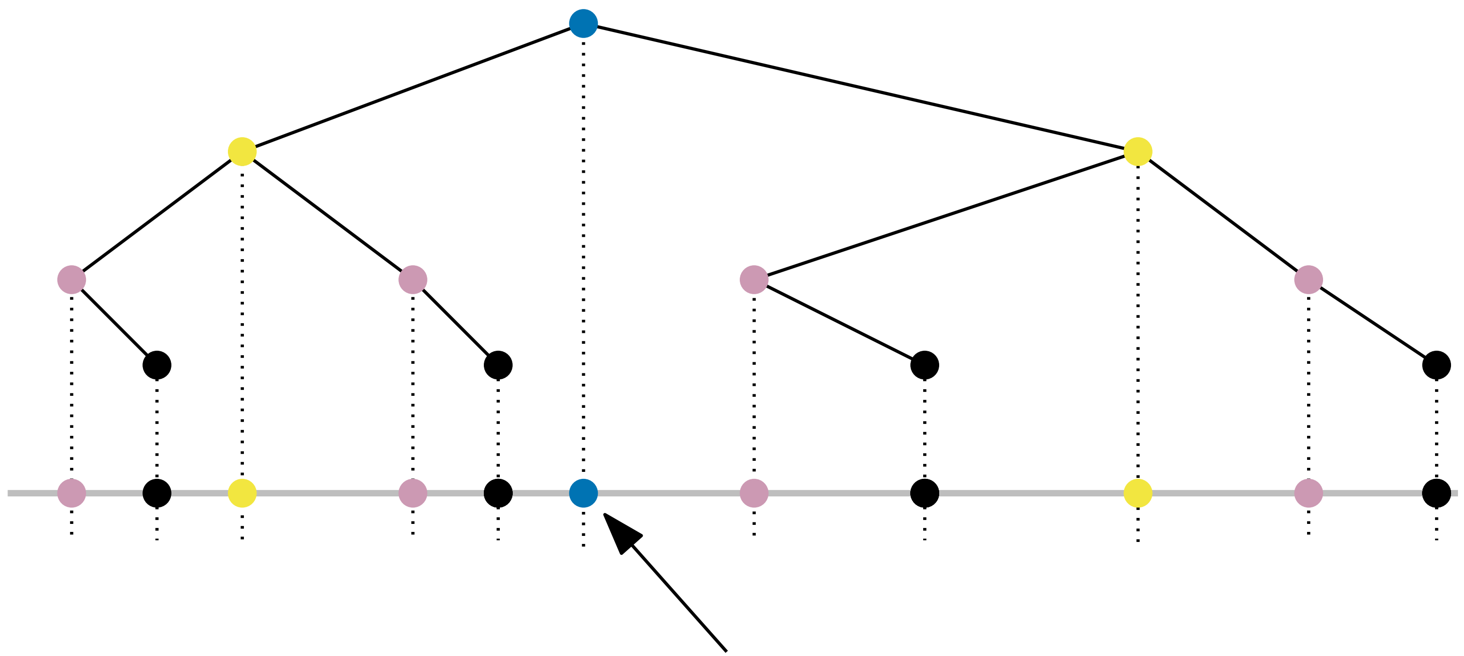
find the point in the middle

. . . and recurse on each half

(in a tie, pick the left)

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



find the point in the middle

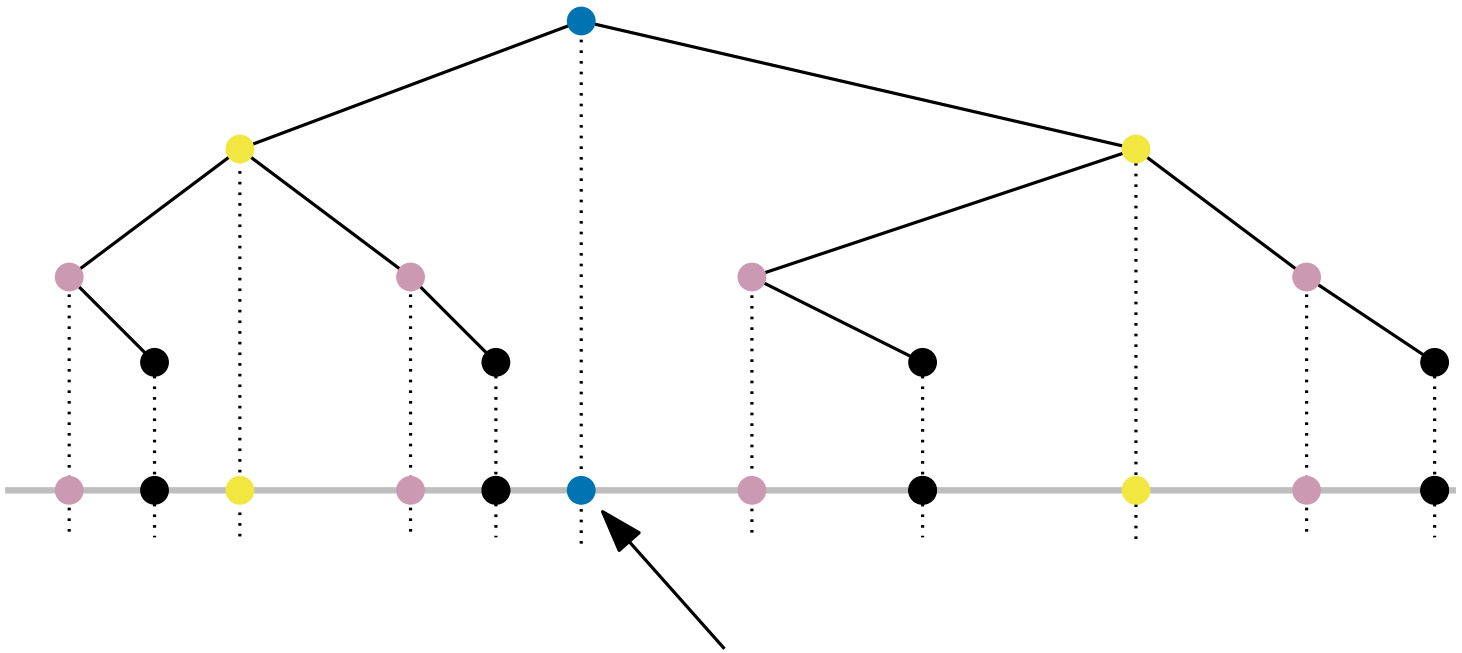
. . . and recurse on each half

(in a tie, pick the left)

We can store the tree in $O(n)$ space *(it has one node per point)*

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



find the point in the middle

. . . and recurse on each half

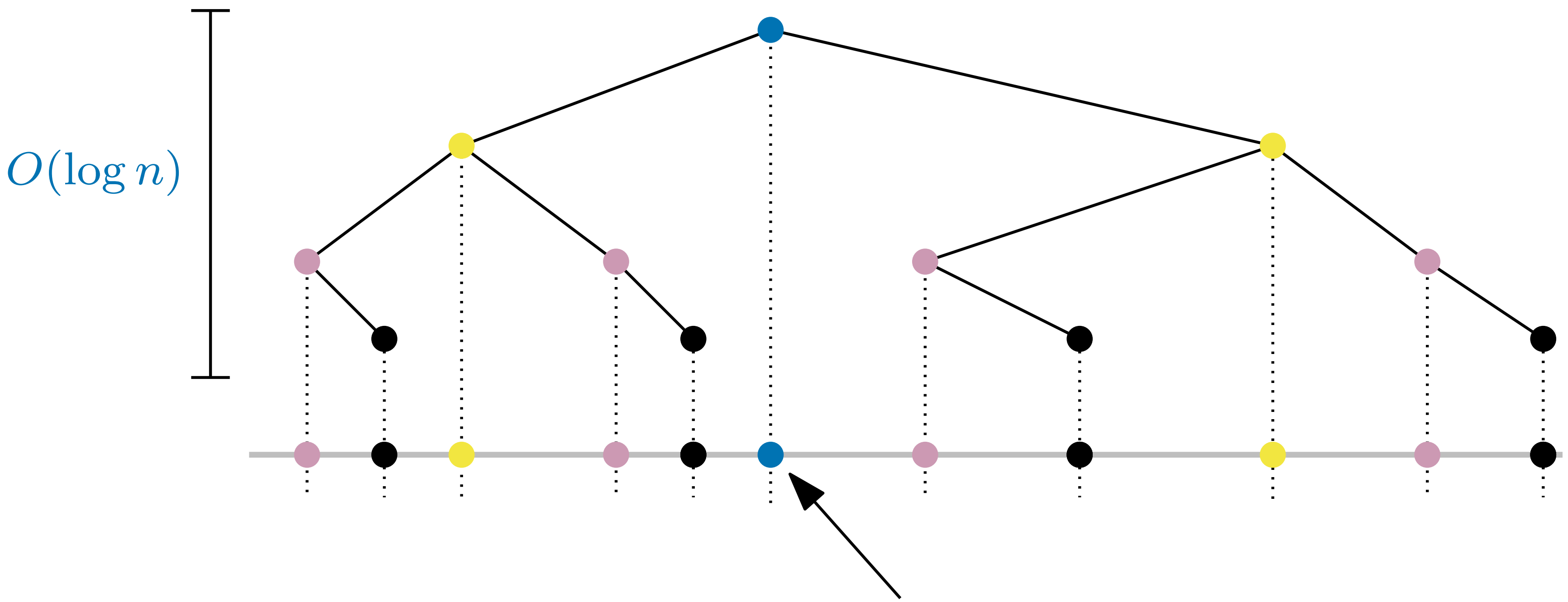
(in a tie, pick the left)

We can store the tree in $O(n)$ space *(it has one node per point)*

It has $O(\log n)$ depth

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



. . . and recurse on each half

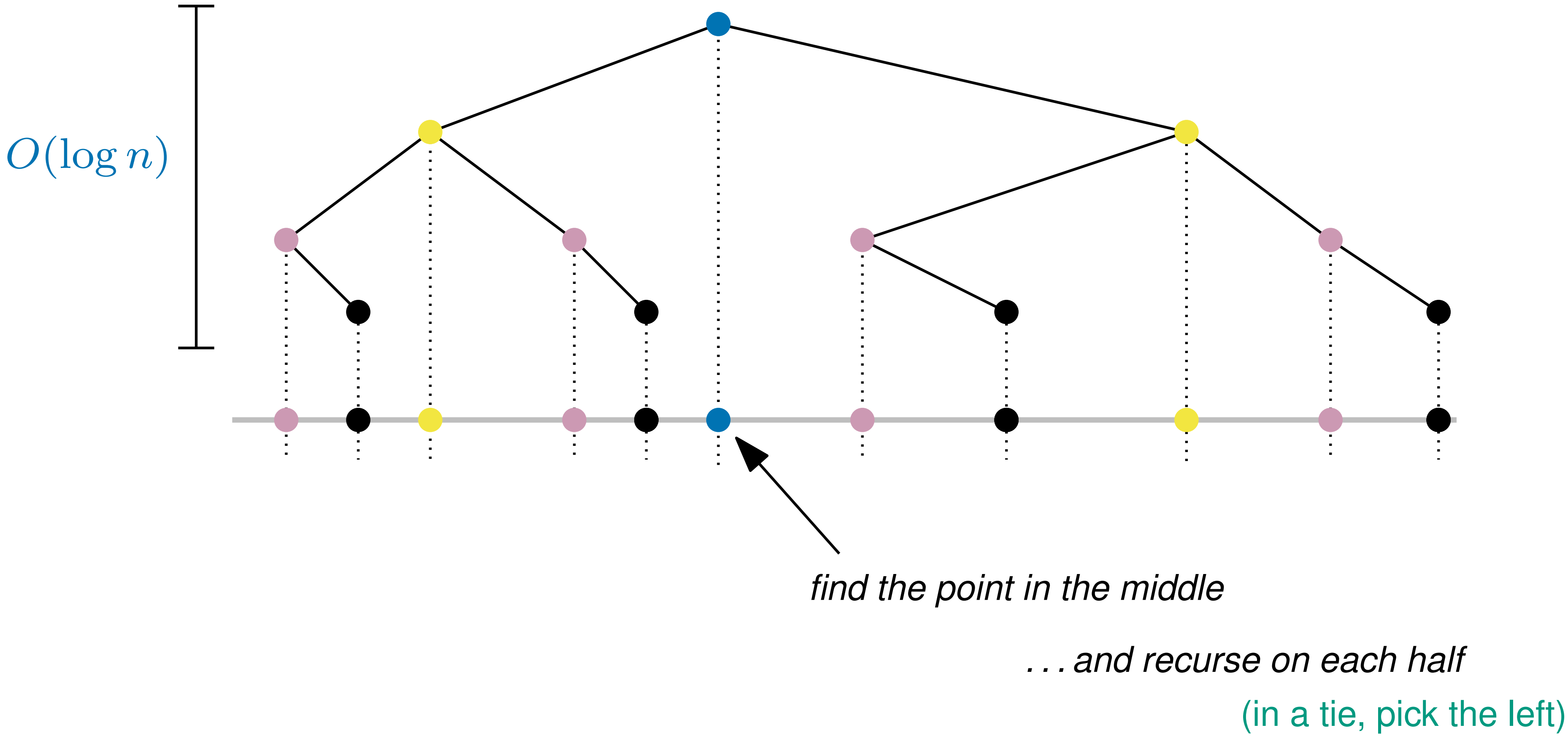
(in a tie, pick the left)

We can store the tree in $O(n)$ space *(it has one node per point)*

It has $O(\log n)$ depth

Starting simple... 1D range searching

alternatively we could build a balanced tree...

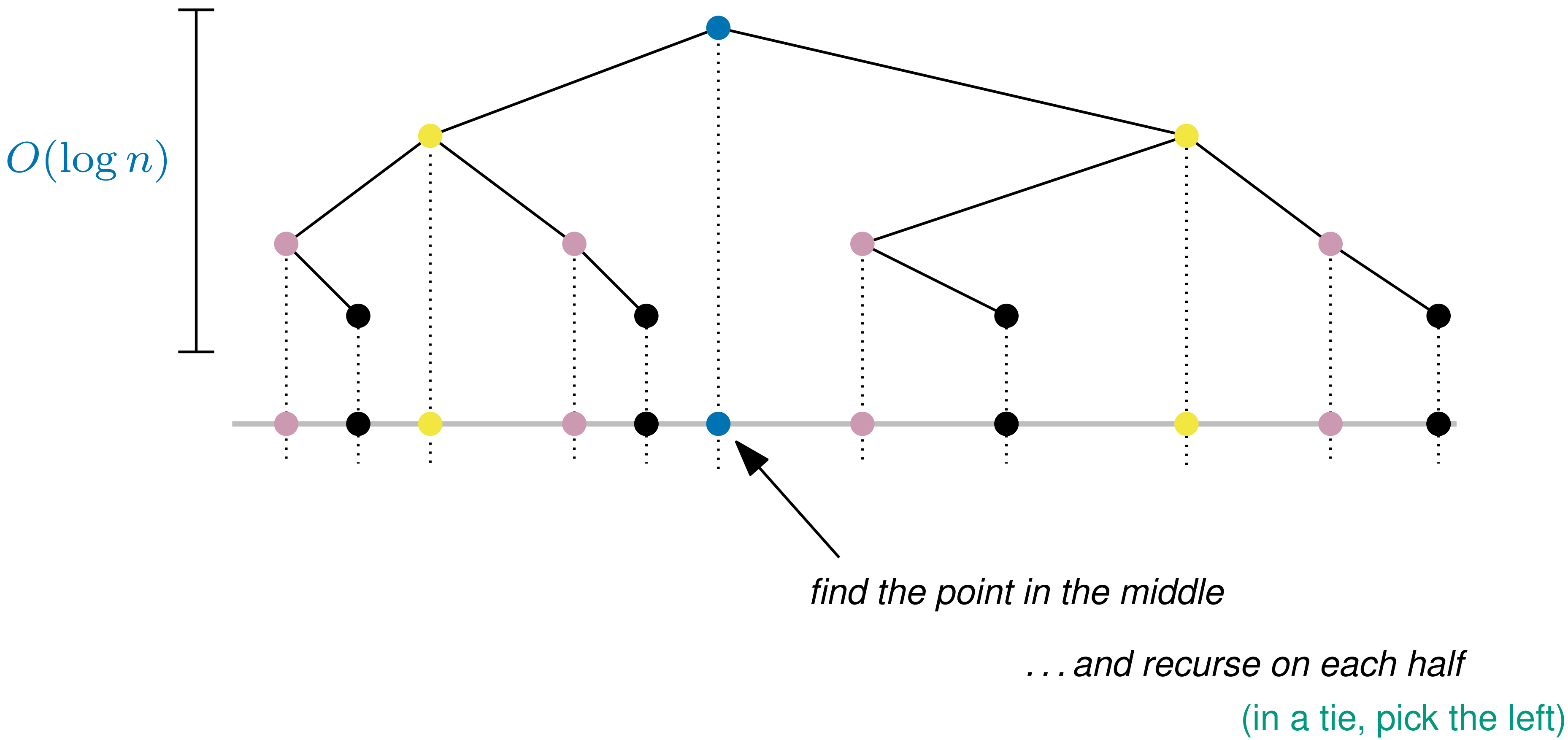


We can store the tree in $O(n)$ space *(it has one node per point)*

It has $O(\log n)$ depth and can be built in $O(n \log n)$ time

Starting simple... 1D range searching

alternatively we could build a balanced tree...

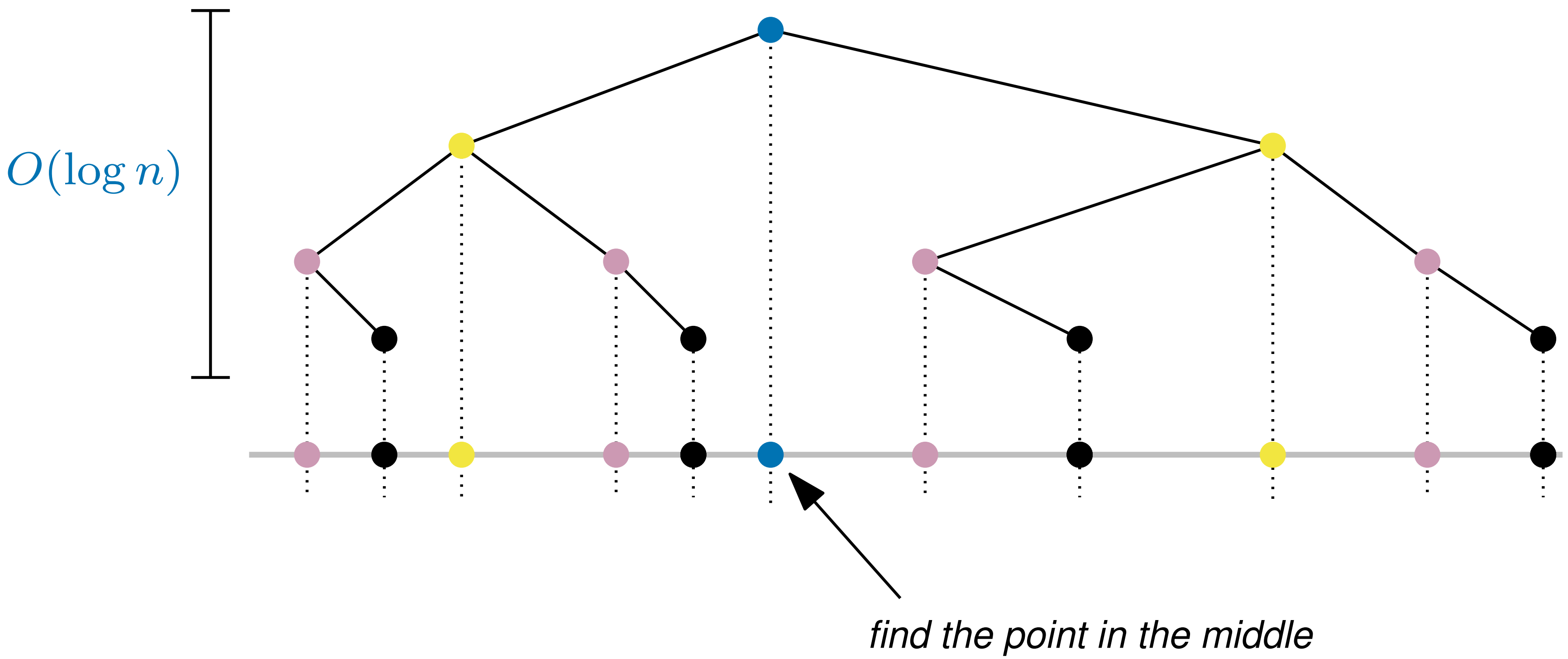


We can store the tree in $O(n)$ space *(it has one node per point)*

It has $O(\log n)$ depth and can be built in $O(n \log n)$ time

Starting simple. . . 1D range searching

alternatively we could build a balanced tree. . .



. . . and recurse on each half

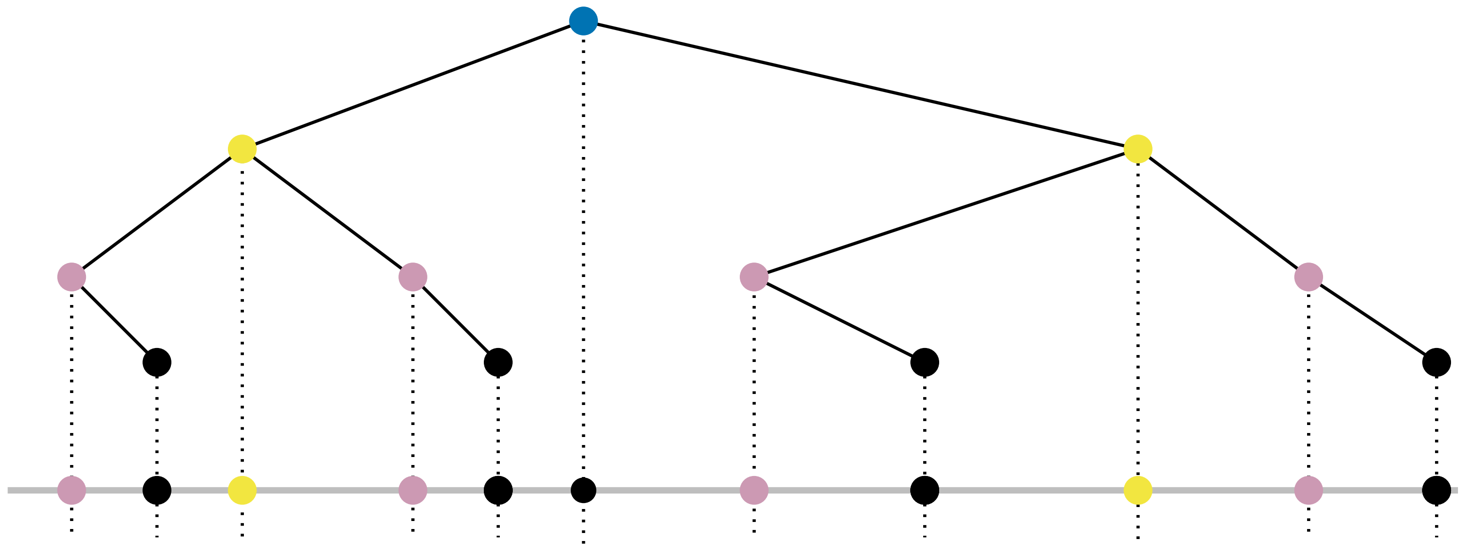
(in a tie, pick the left)

We can store the tree in $O(n)$ space *(it has one node per point)*

It has $O(\log n)$ depth and can be built in $O(n \log n)$ time ($O(n)$ time if the points are sorted)

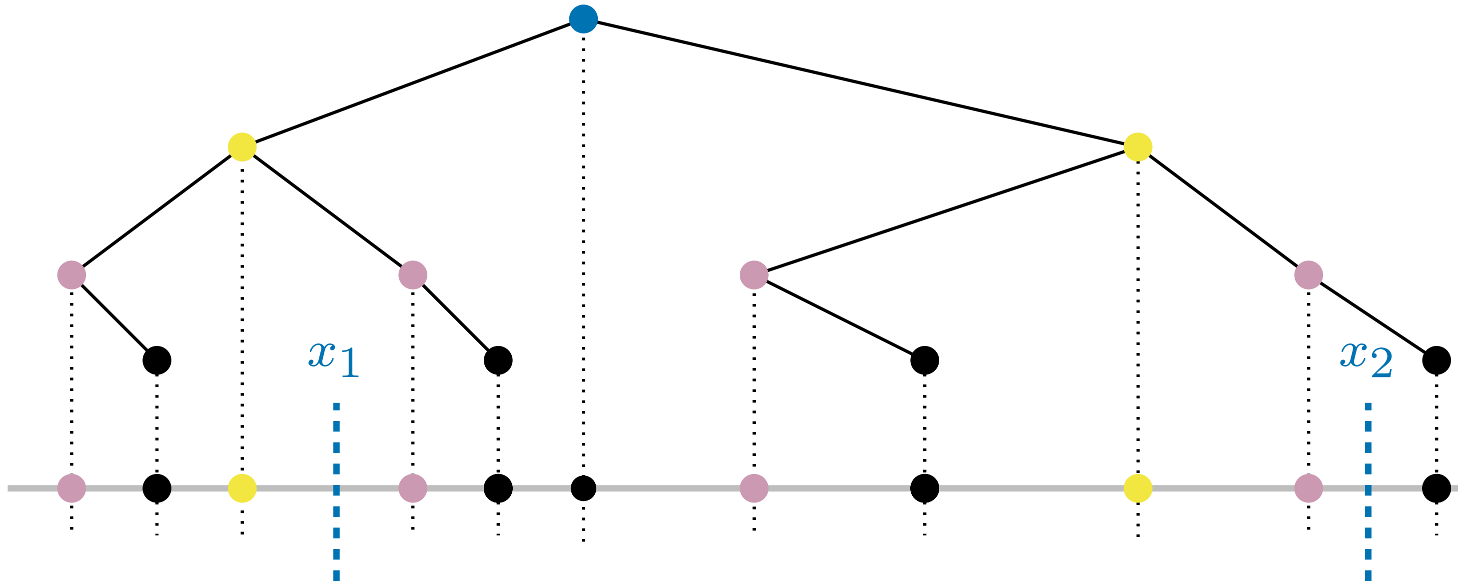
Starting simple. . . 1D range searching

how do we do a lookup?



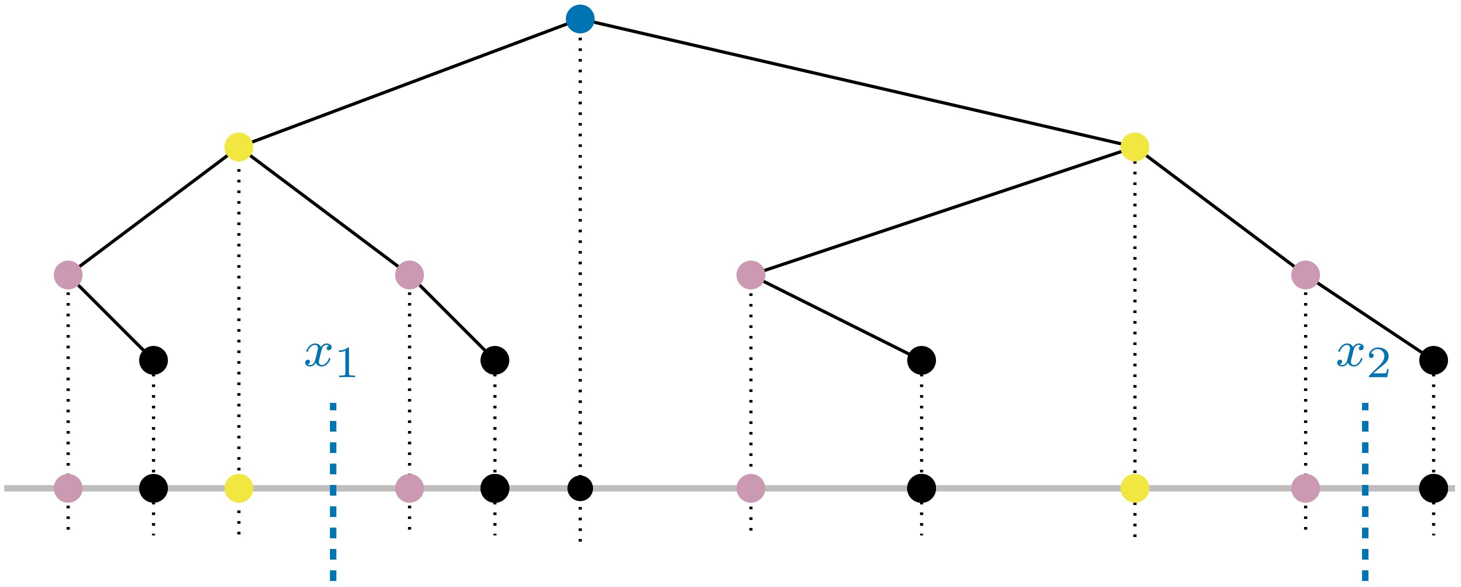
Starting simple. . . 1D range searching

how do we do a lookup?



Starting simple. . . 1D range searching

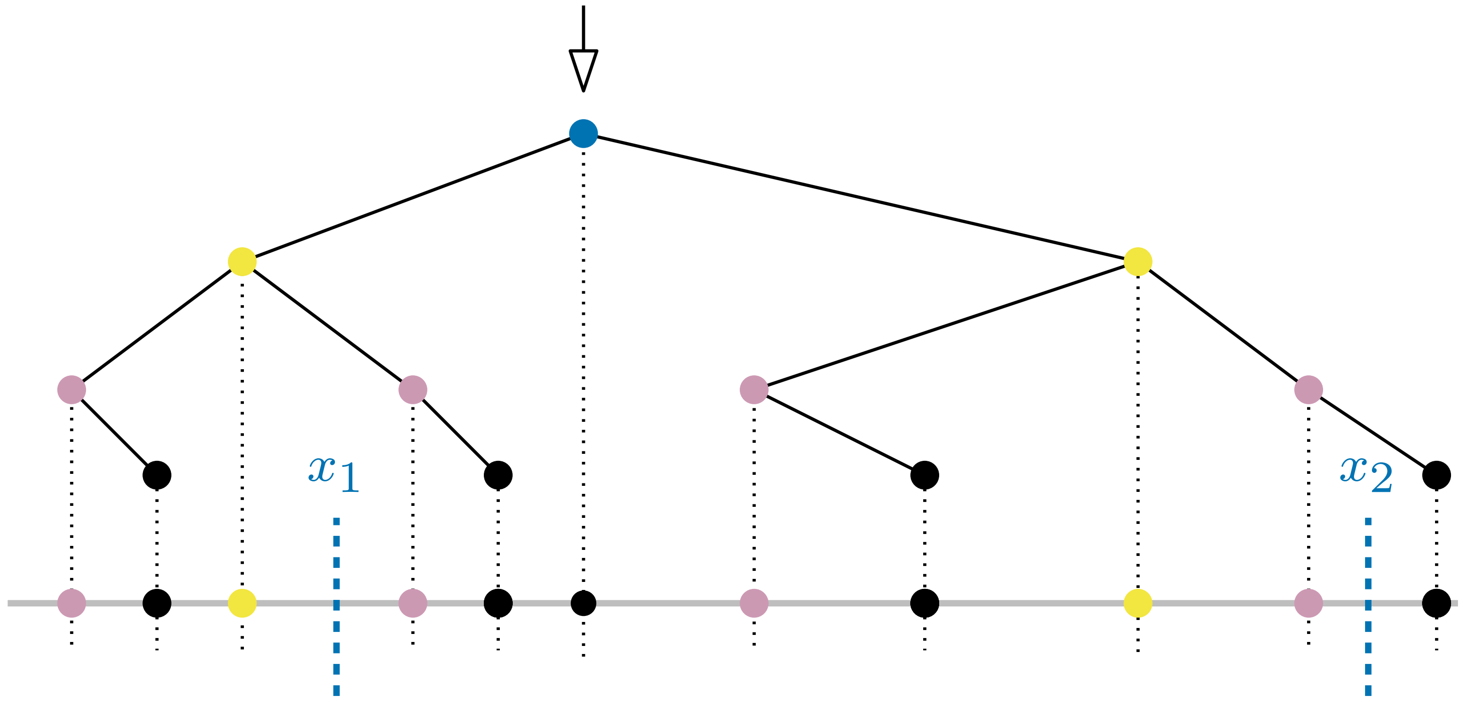
how do we do a lookup?



Step 1: find the successor of x_1

Starting simple... 1D range searching

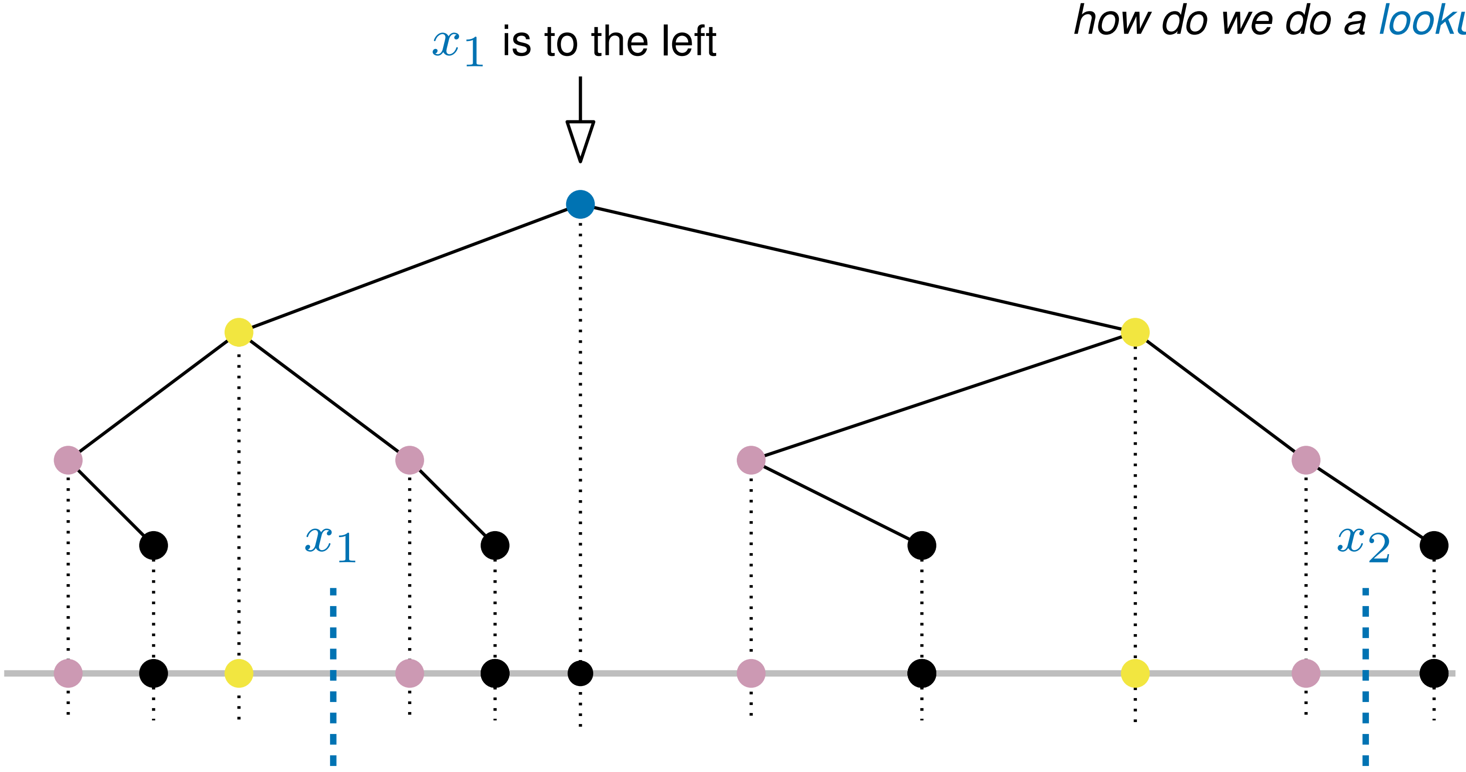
how do we do a lookup?



Step 1: find the successor of x_1

Starting simple... 1D range searching

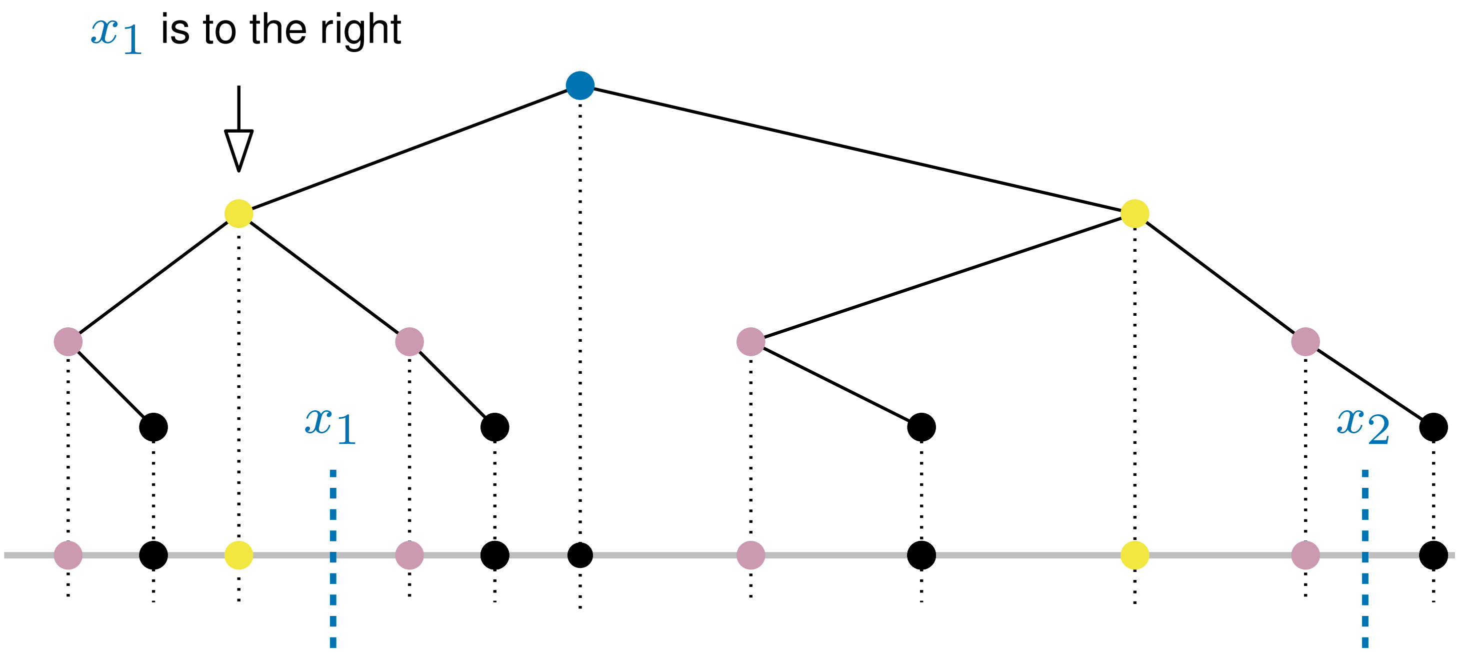
how do we do a lookup?



Step 1: find the successor of x_1

Starting simple... 1D range searching

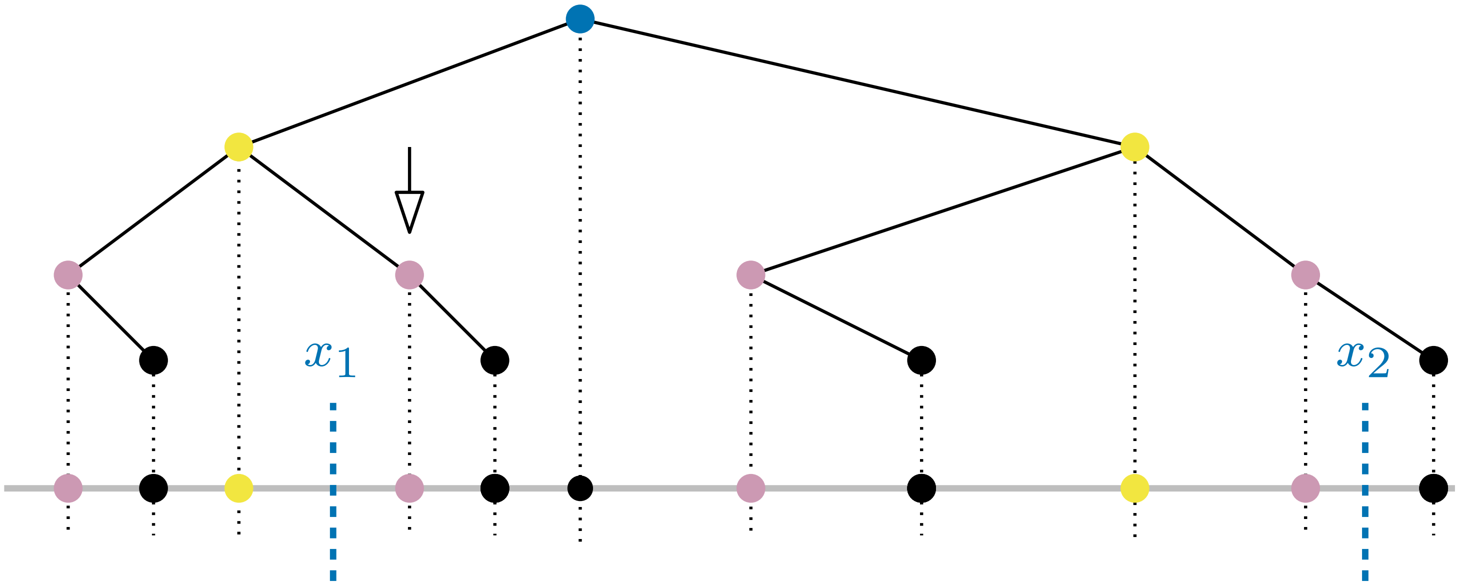
how do we do a lookup?



Step 1: find the successor of x_1

Starting simple. . . 1D range searching

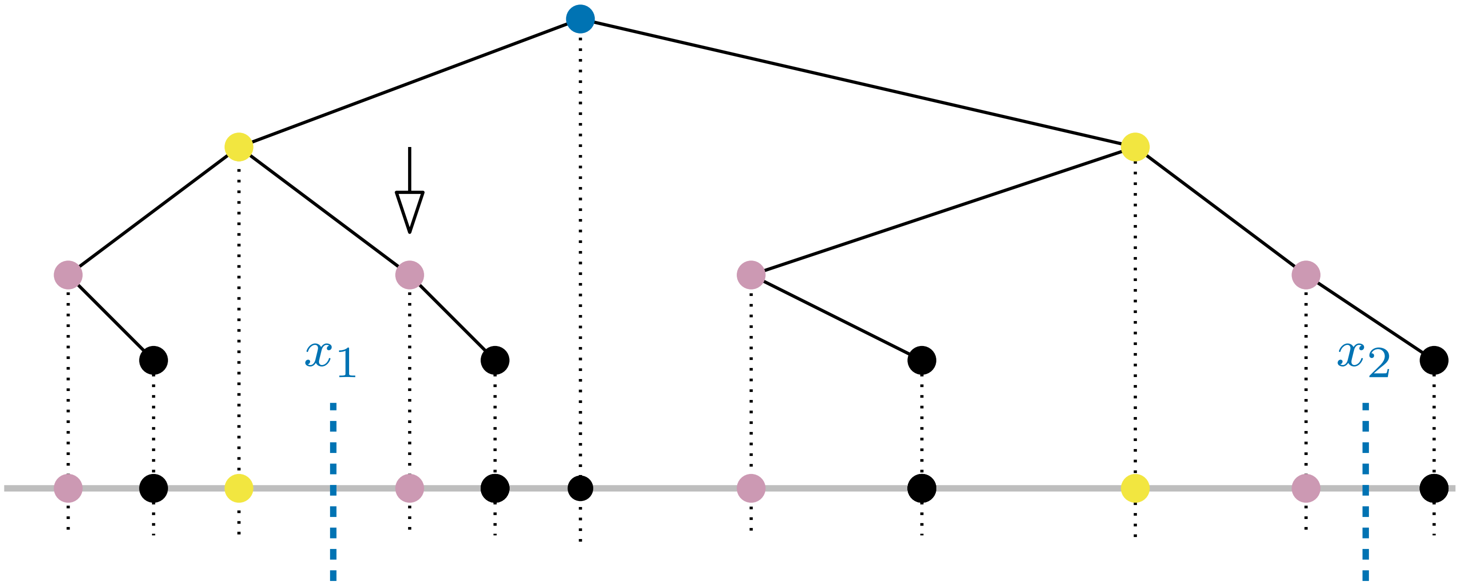
how do we do a lookup?



Step 1: find the successor of x_1

Starting simple. . . 1D range searching

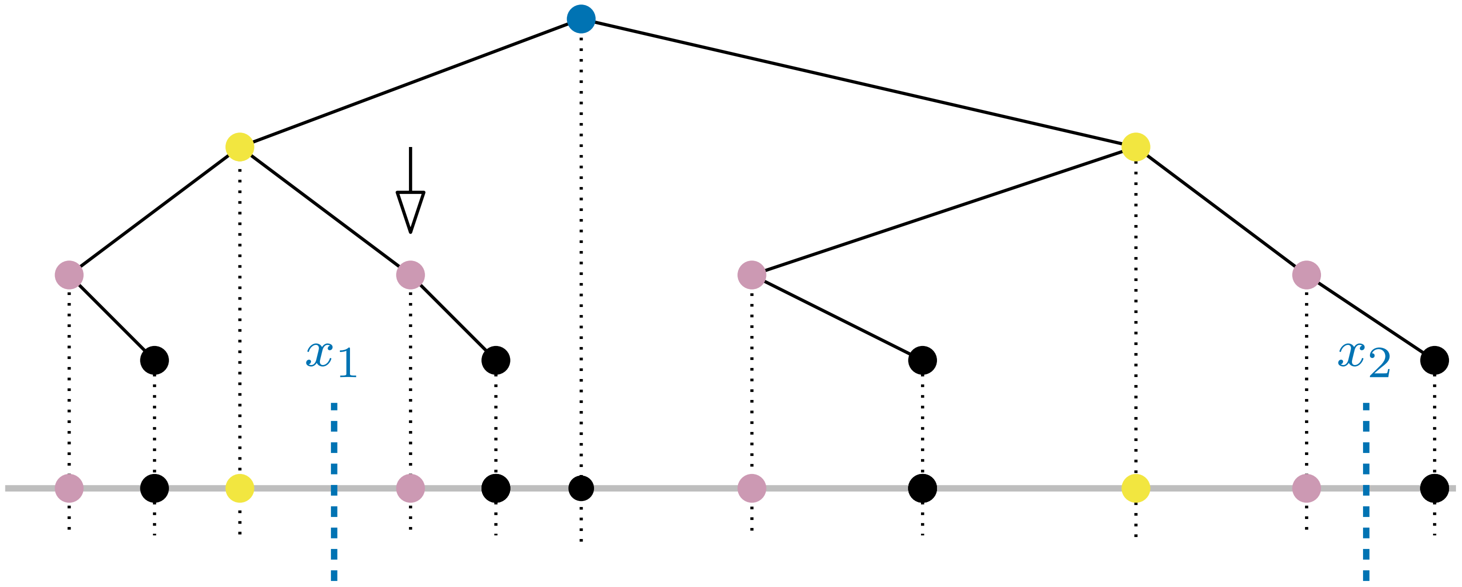
how do we do a lookup?



Step 1: find the successor of x_1 in $O(\log n)$ time

Starting simple... 1D range searching

how do we do a lookup?

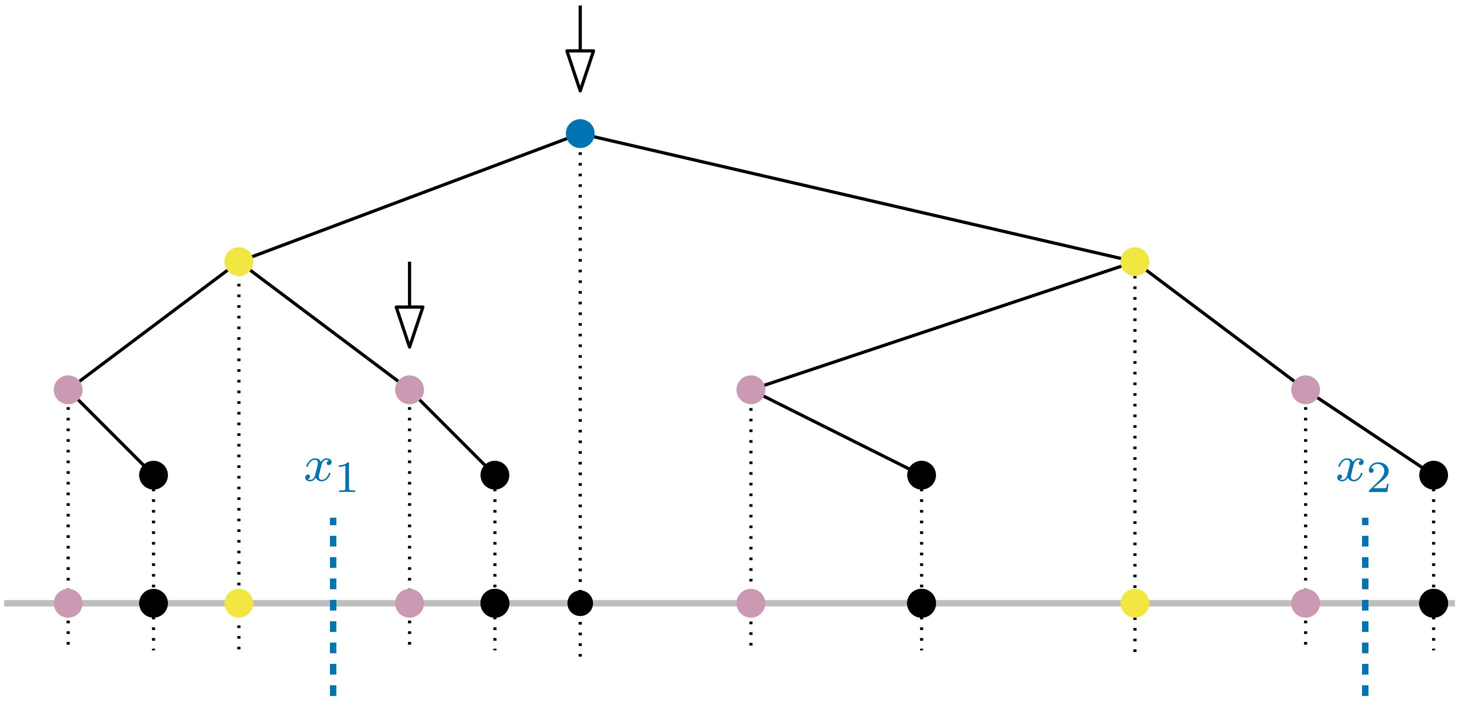


Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2

Starting simple. . . 1D range searching

how do we do a lookup?

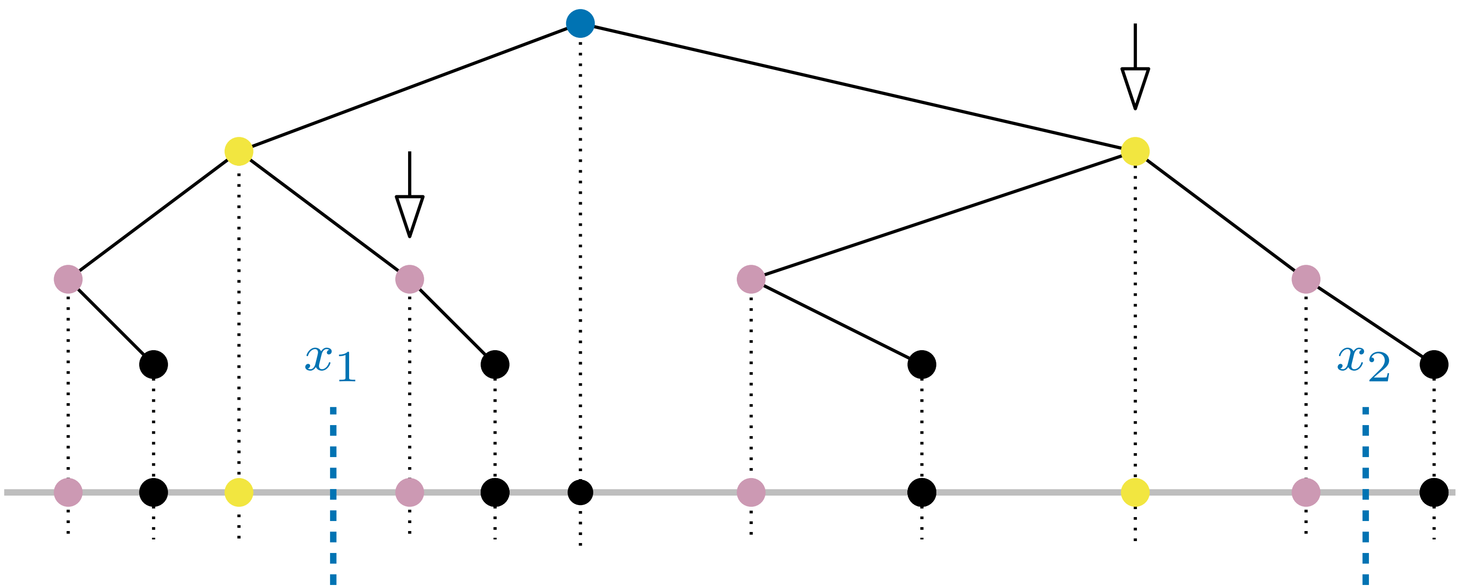


Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2

Starting simple. . . 1D range searching

how do we do a lookup?

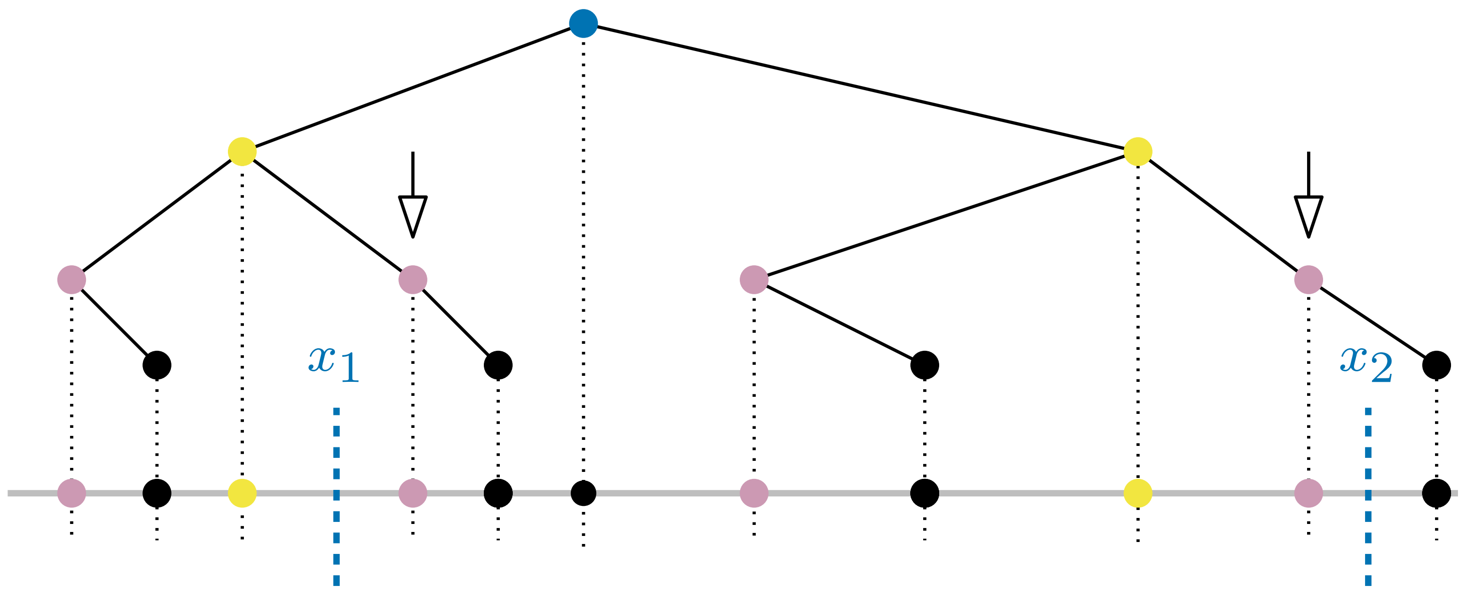


Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2

Starting simple. . . 1D range searching

how do we do a lookup?

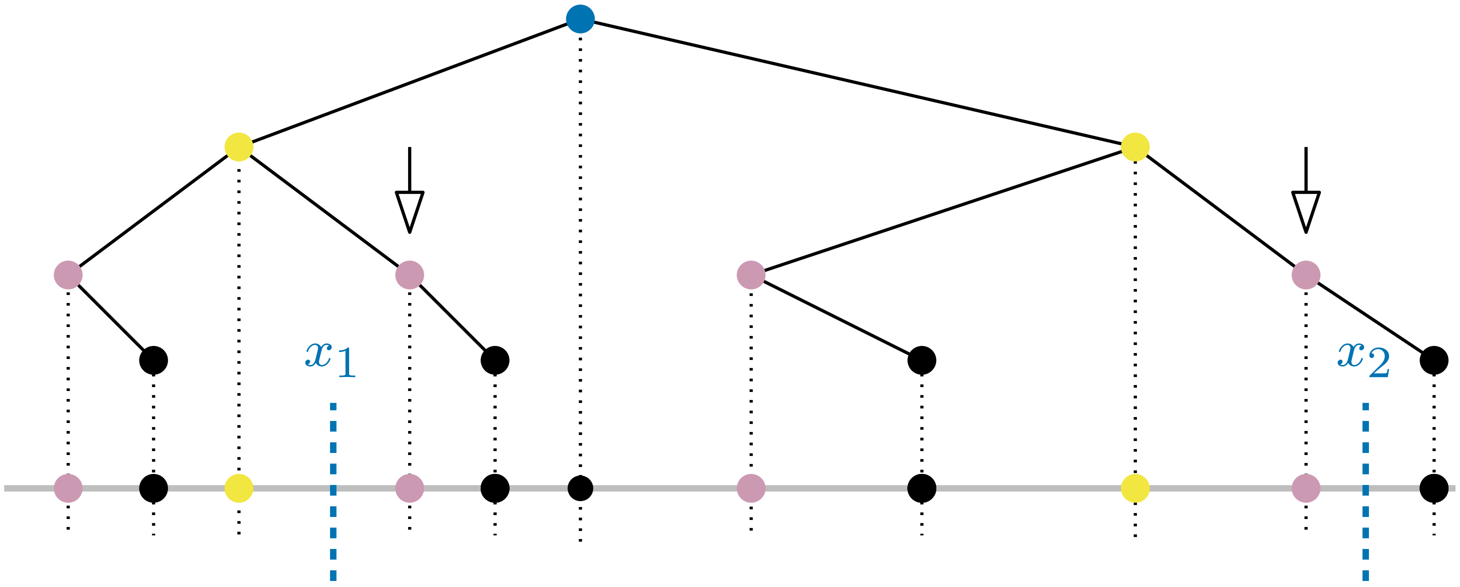


Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2

Starting simple. . . 1D range searching

how do we do a lookup?

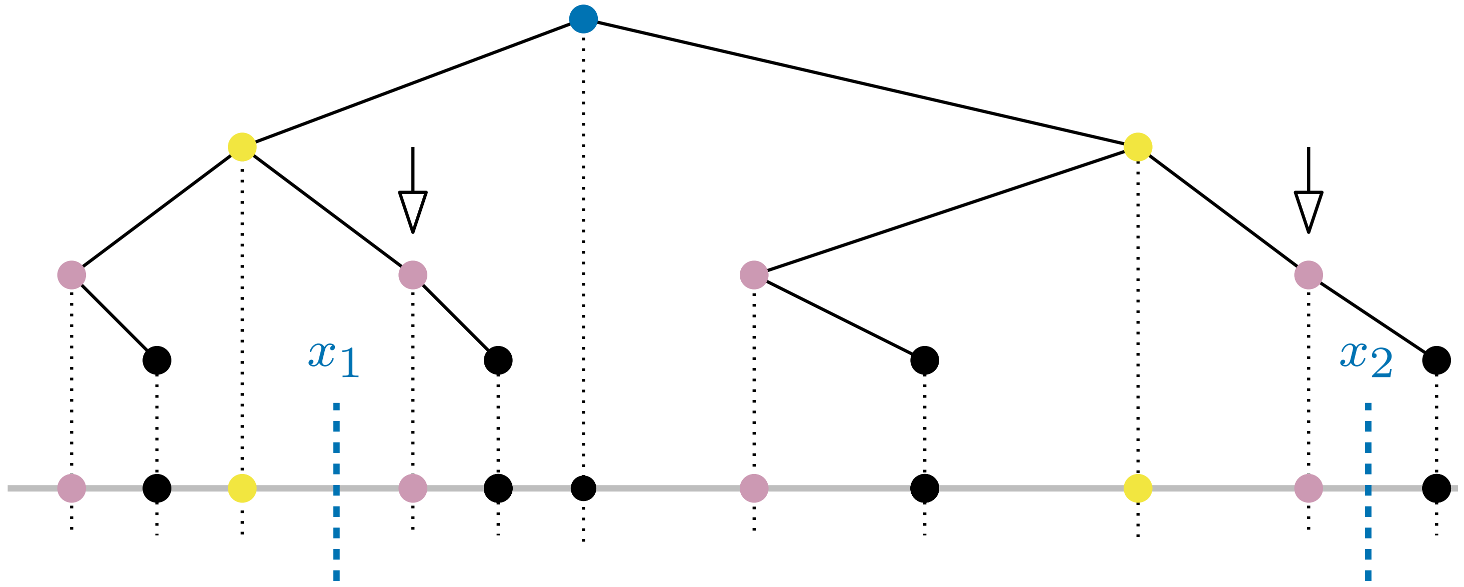


Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

Starting simple... 1D range searching

how do we do a lookup?



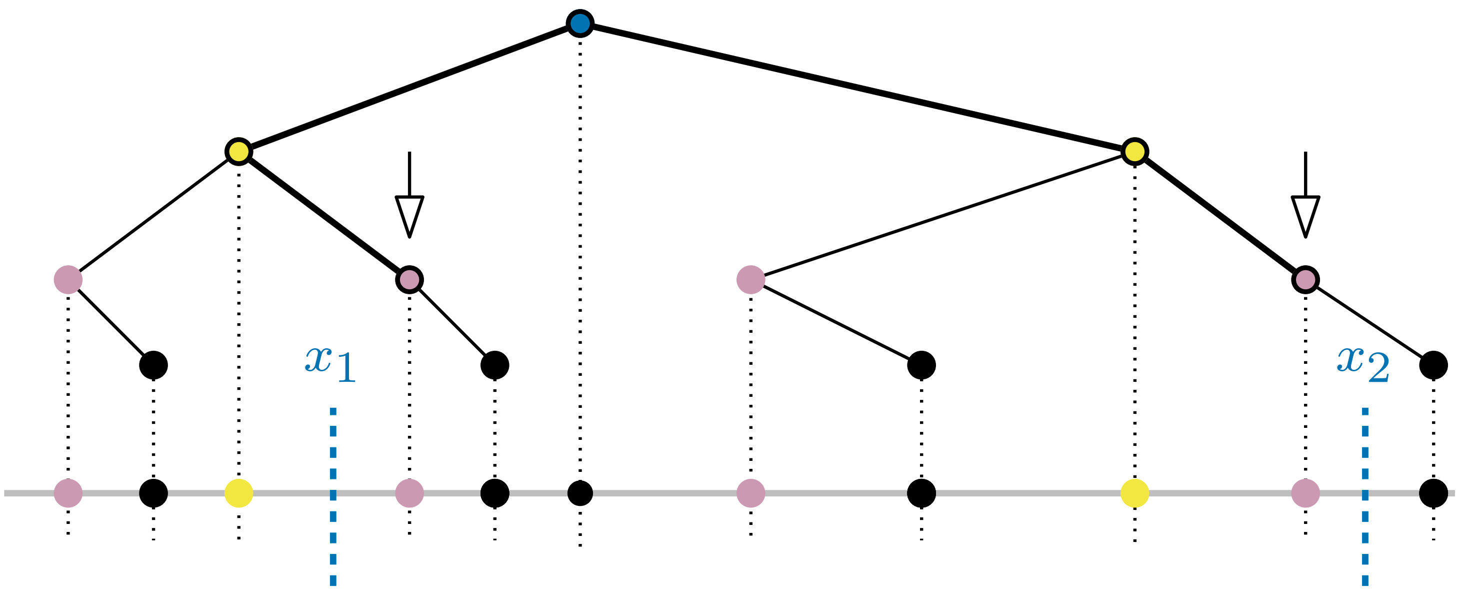
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



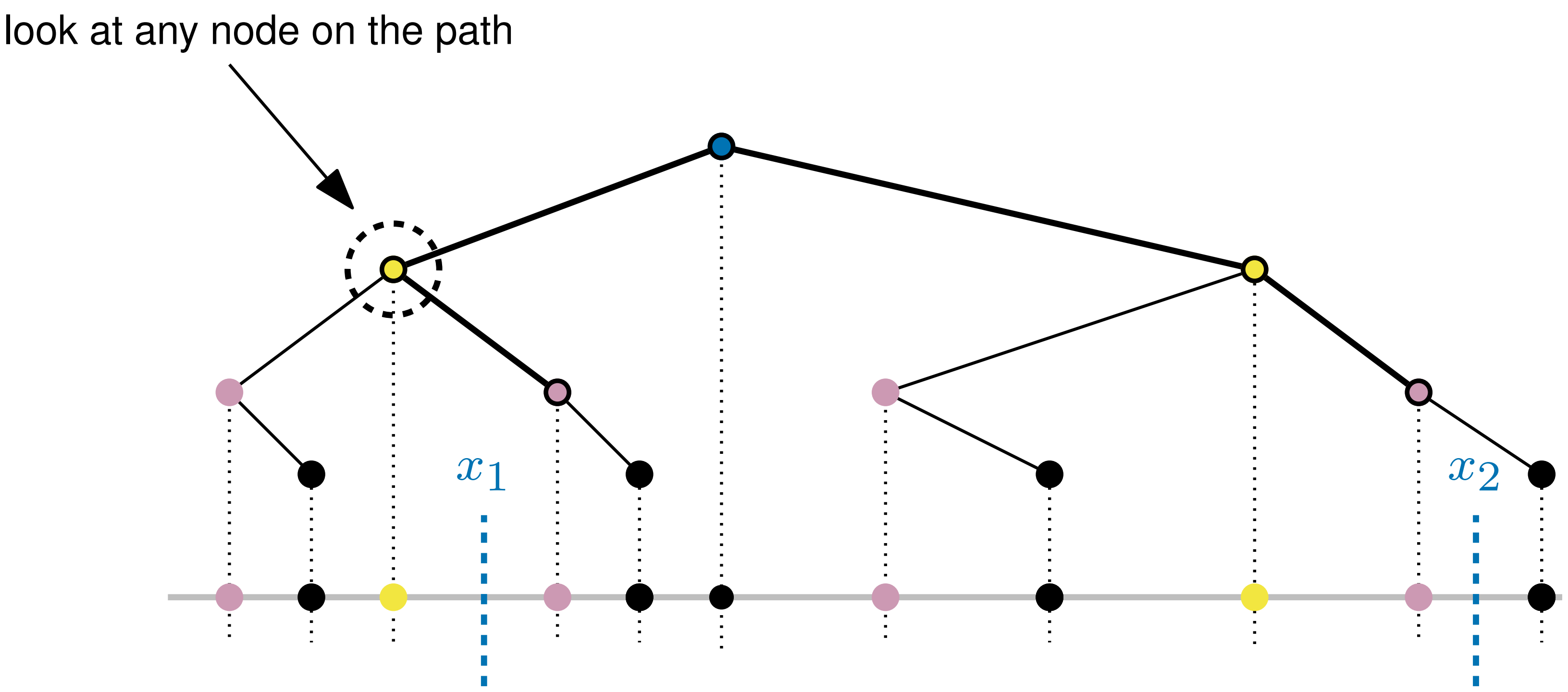
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



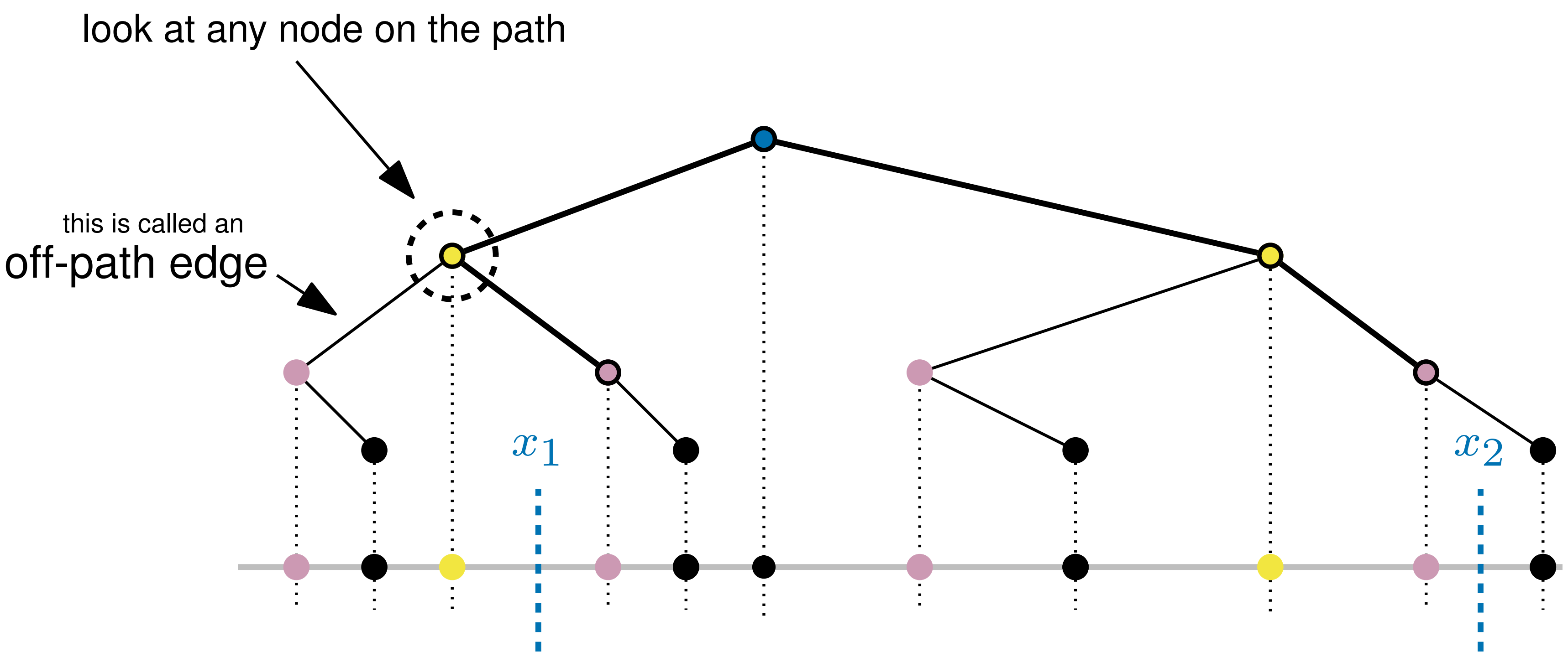
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



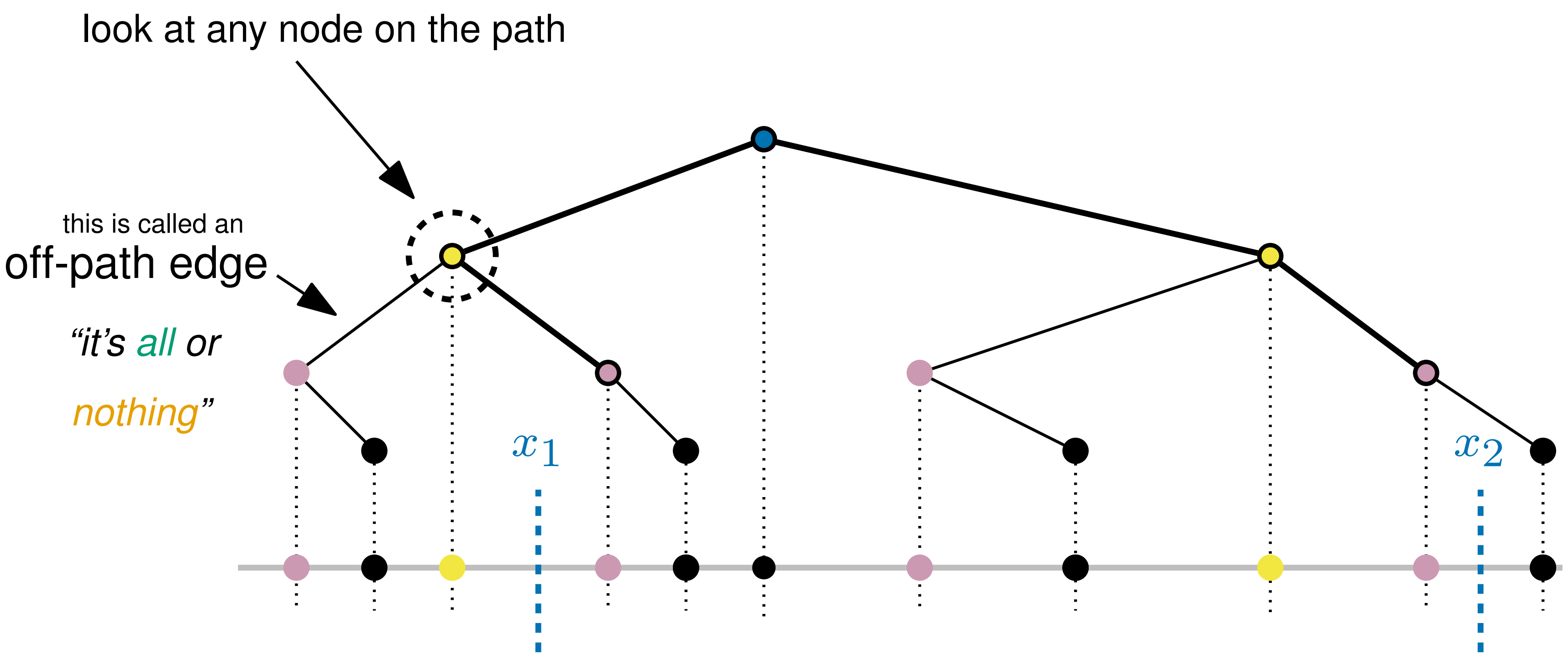
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



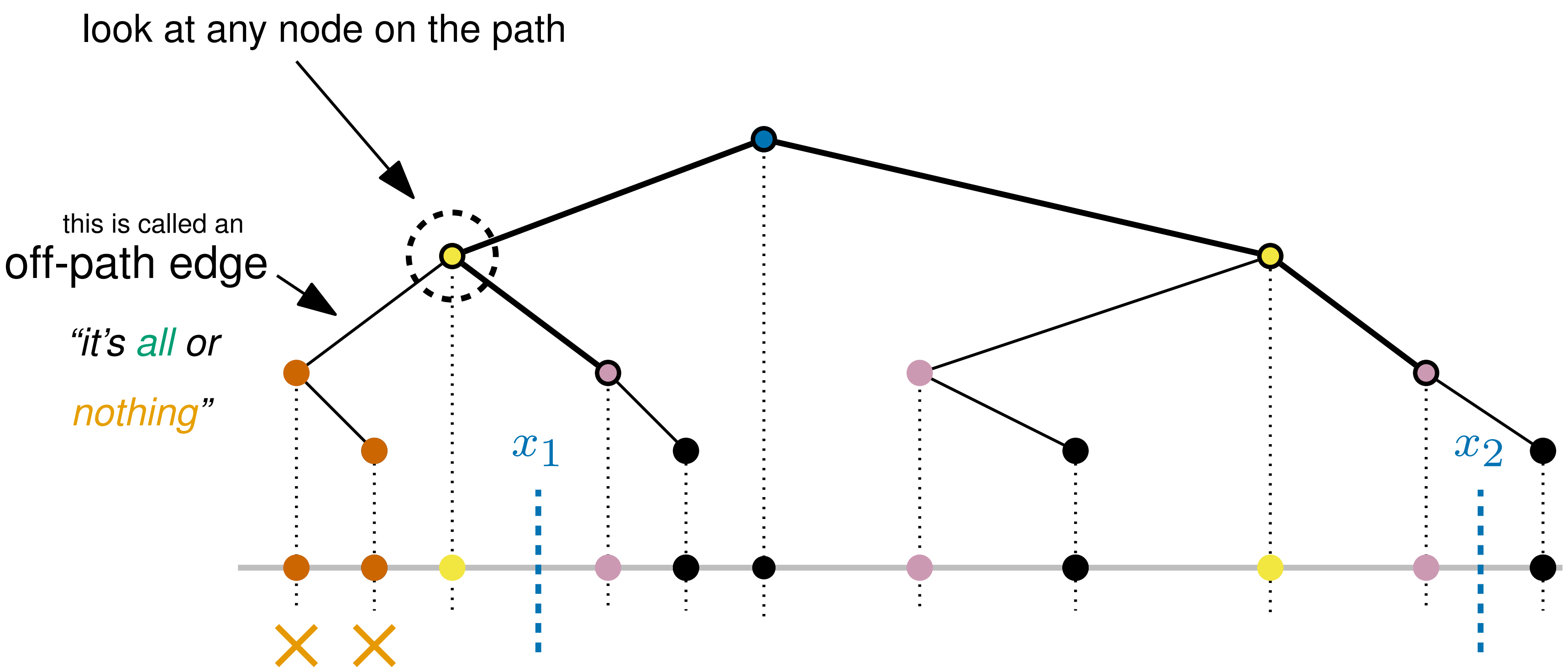
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



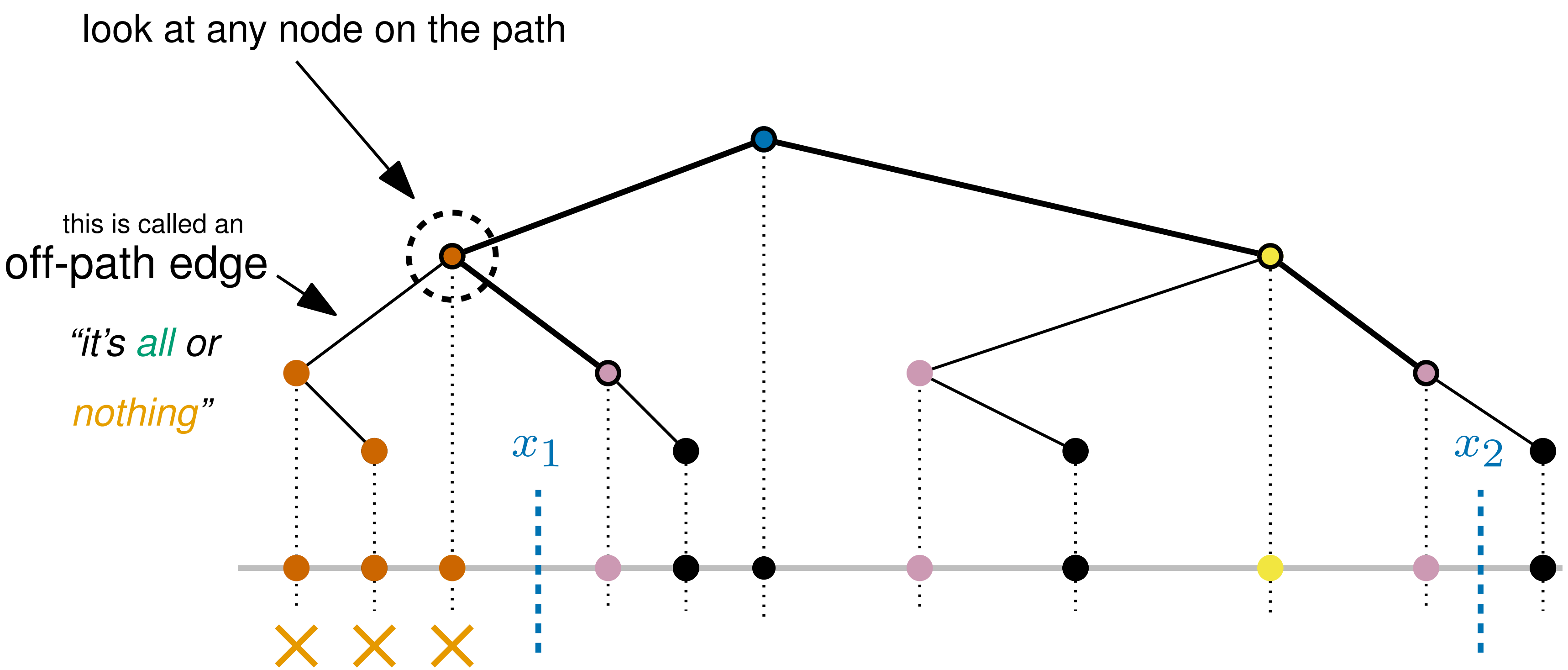
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



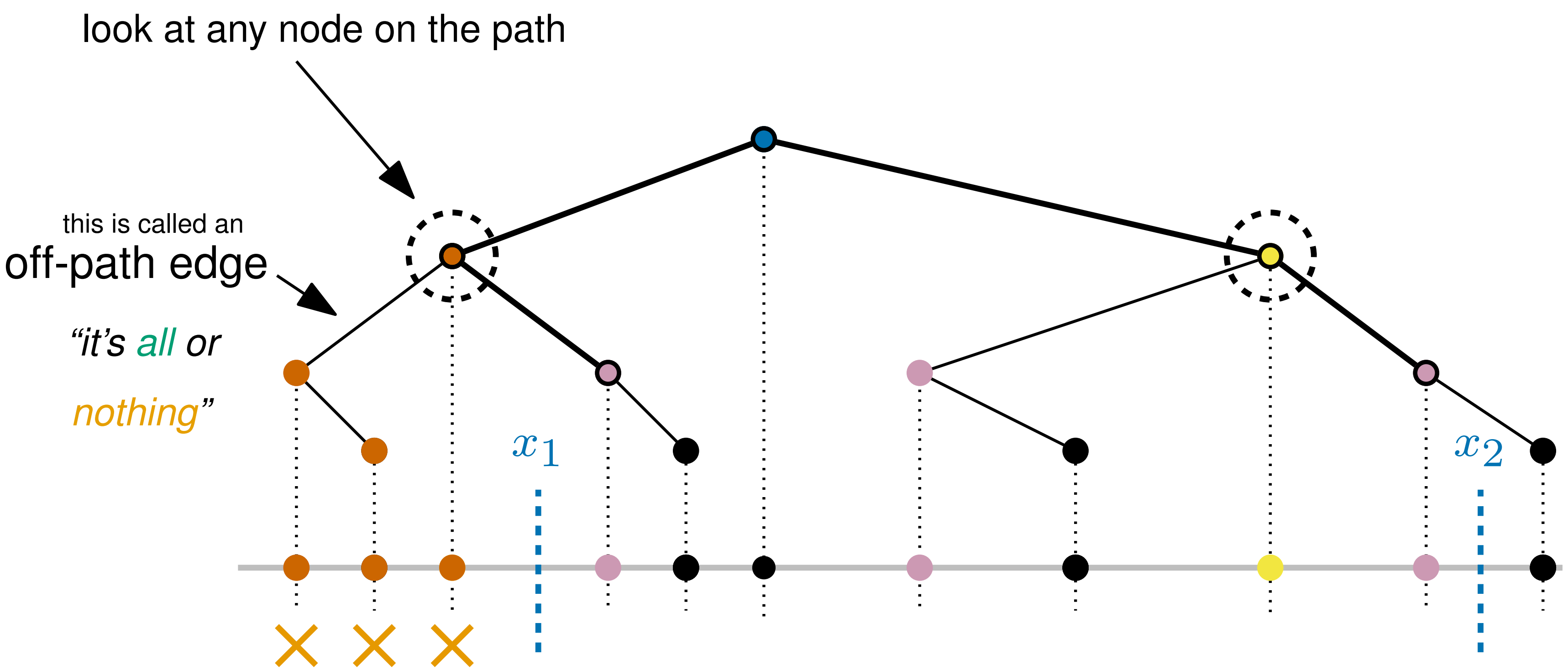
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



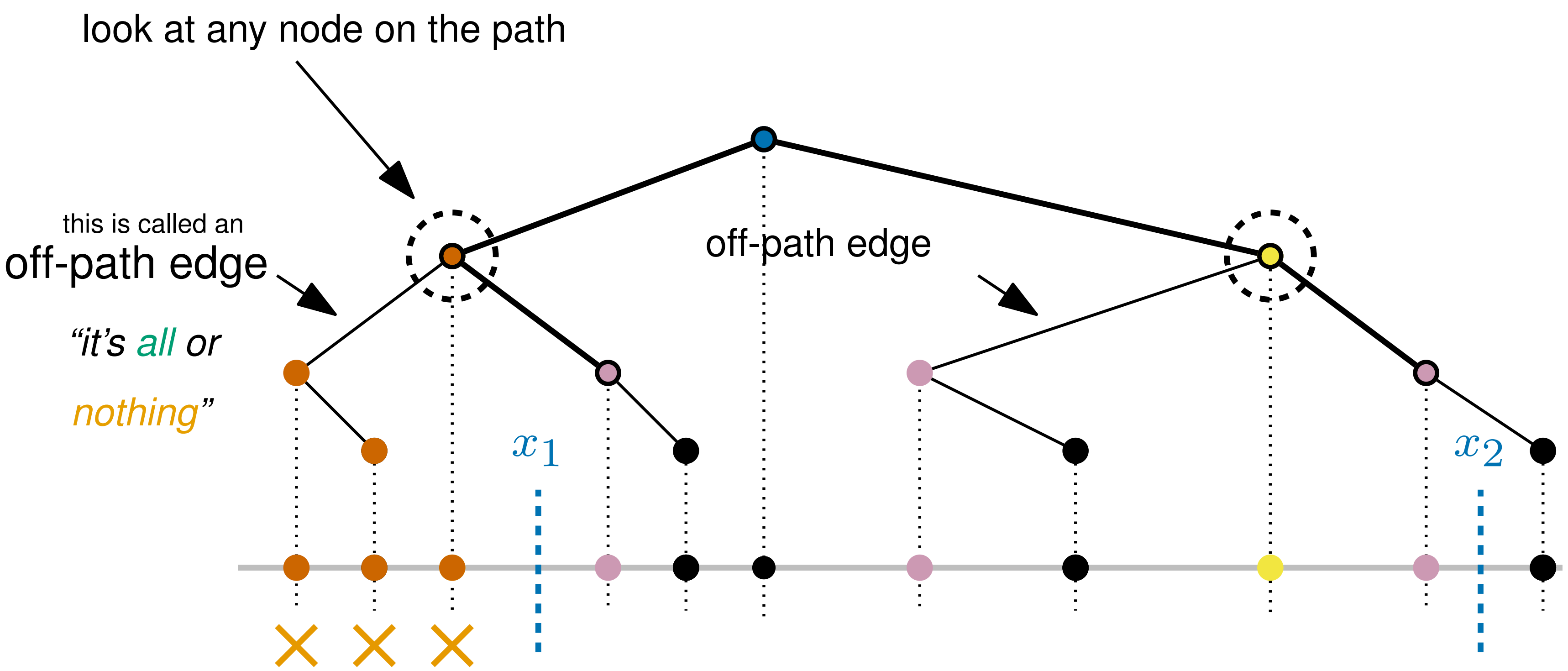
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



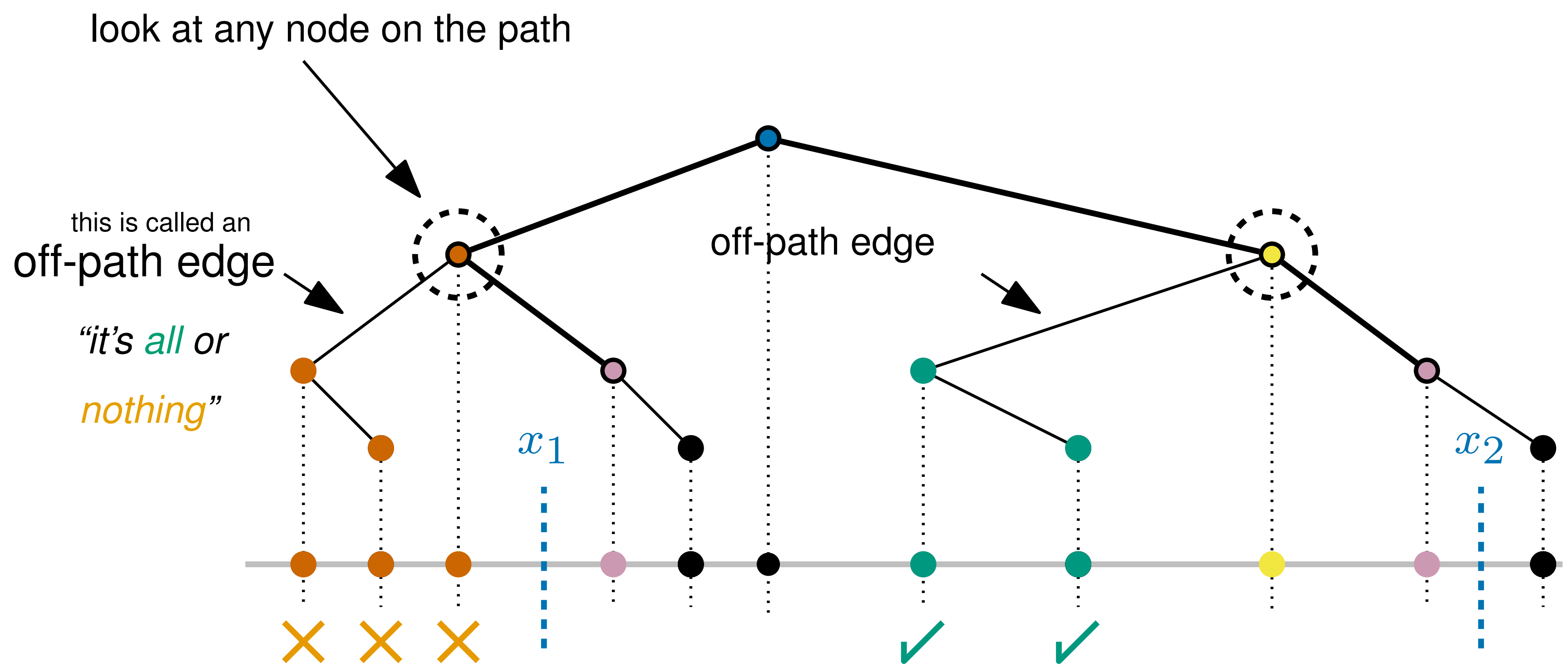
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



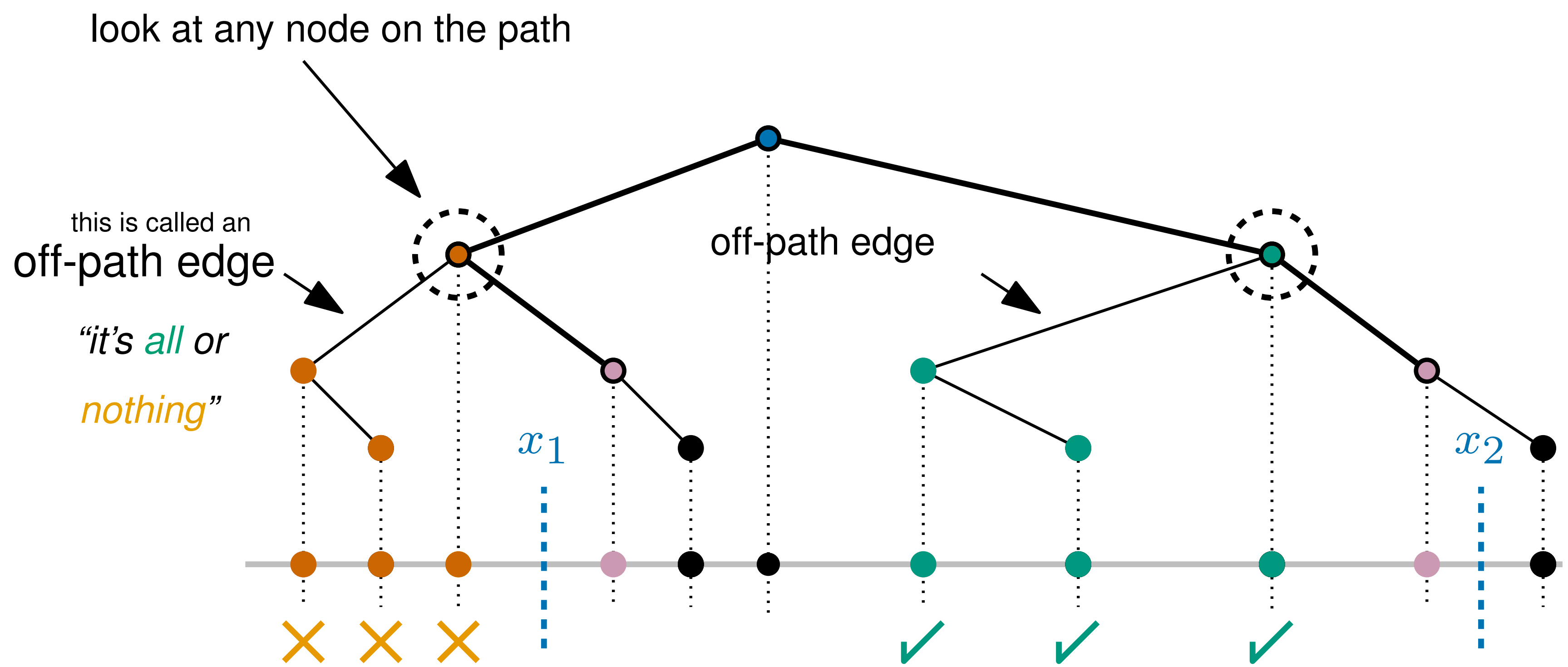
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



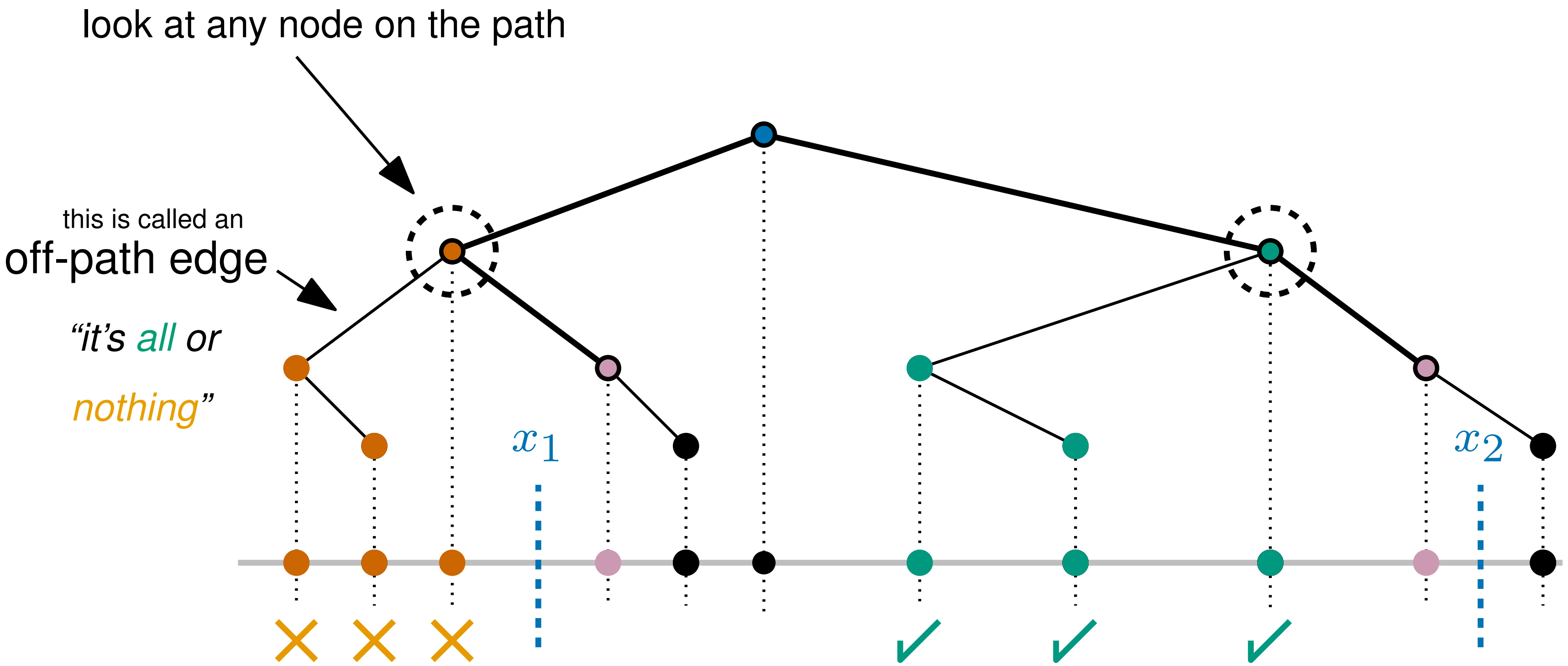
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



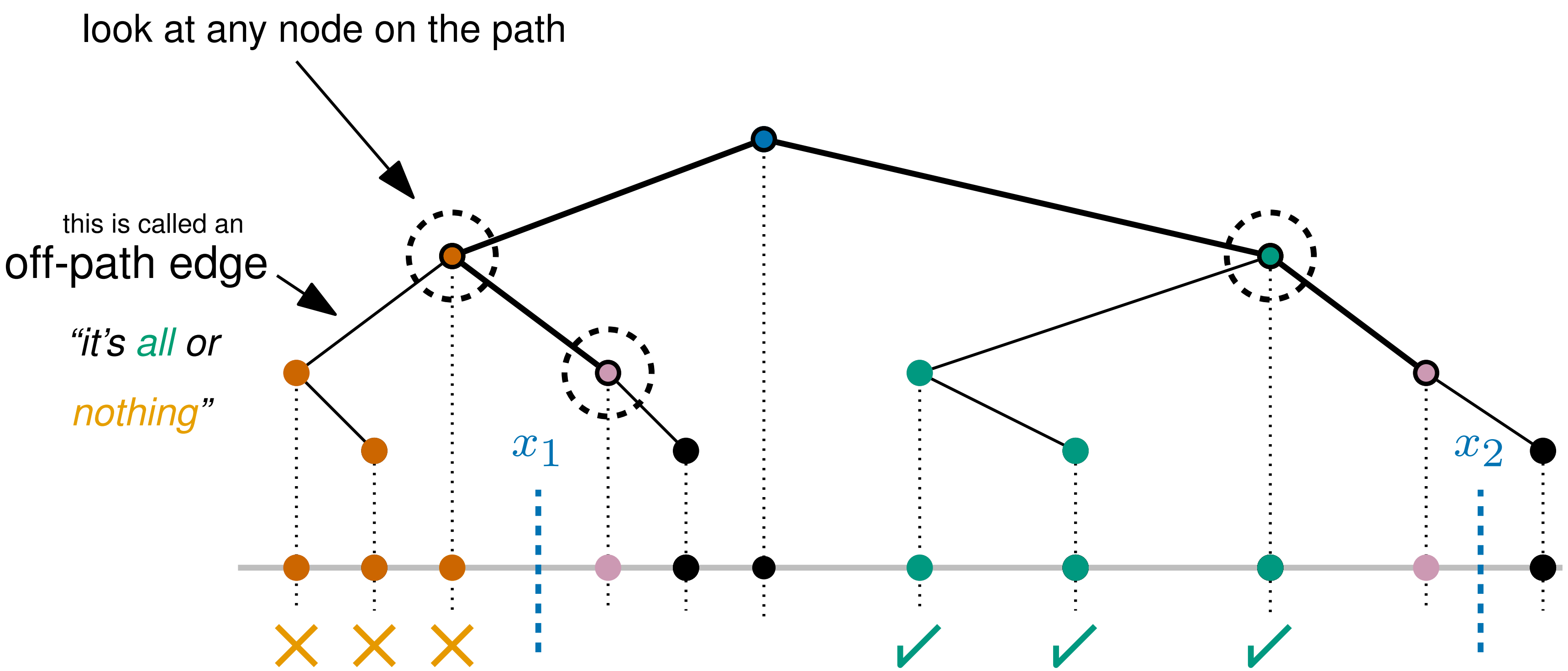
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



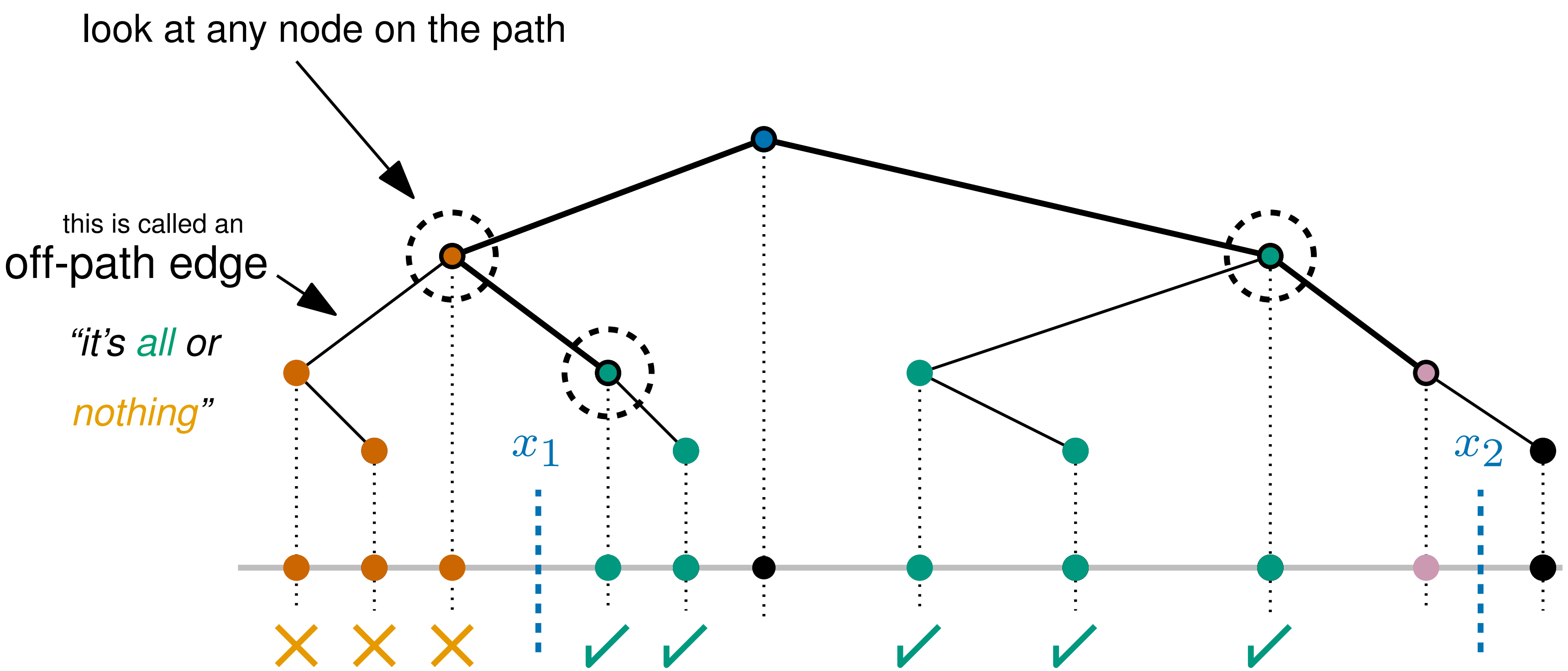
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



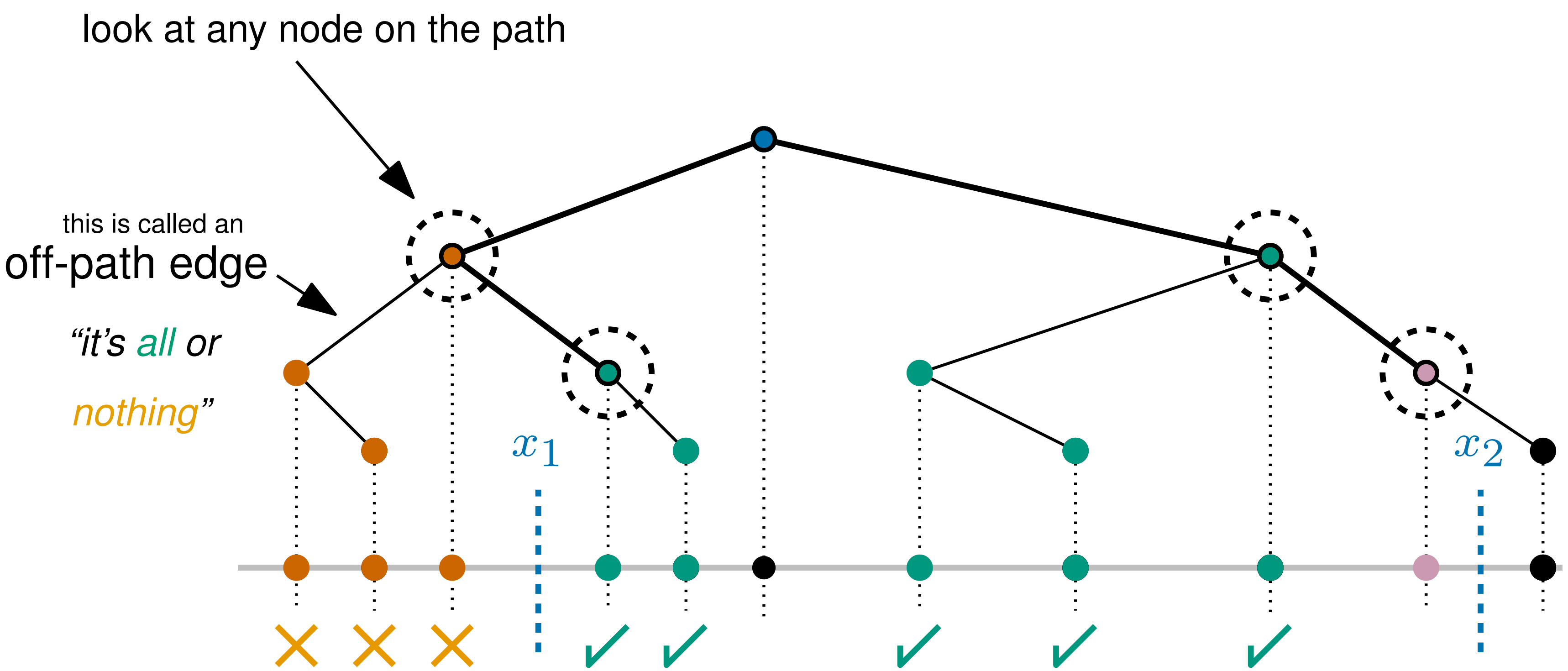
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple. . . 1D range searching

how do we do a lookup?



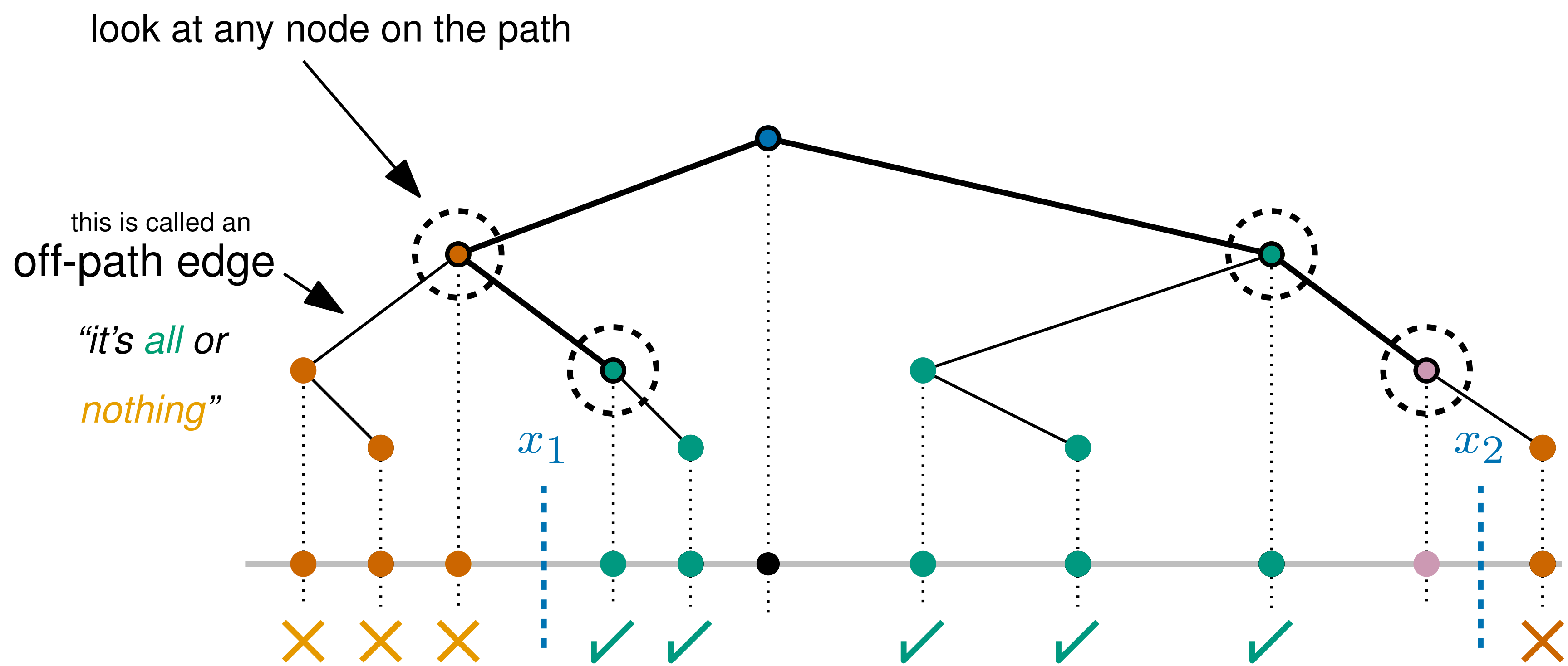
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

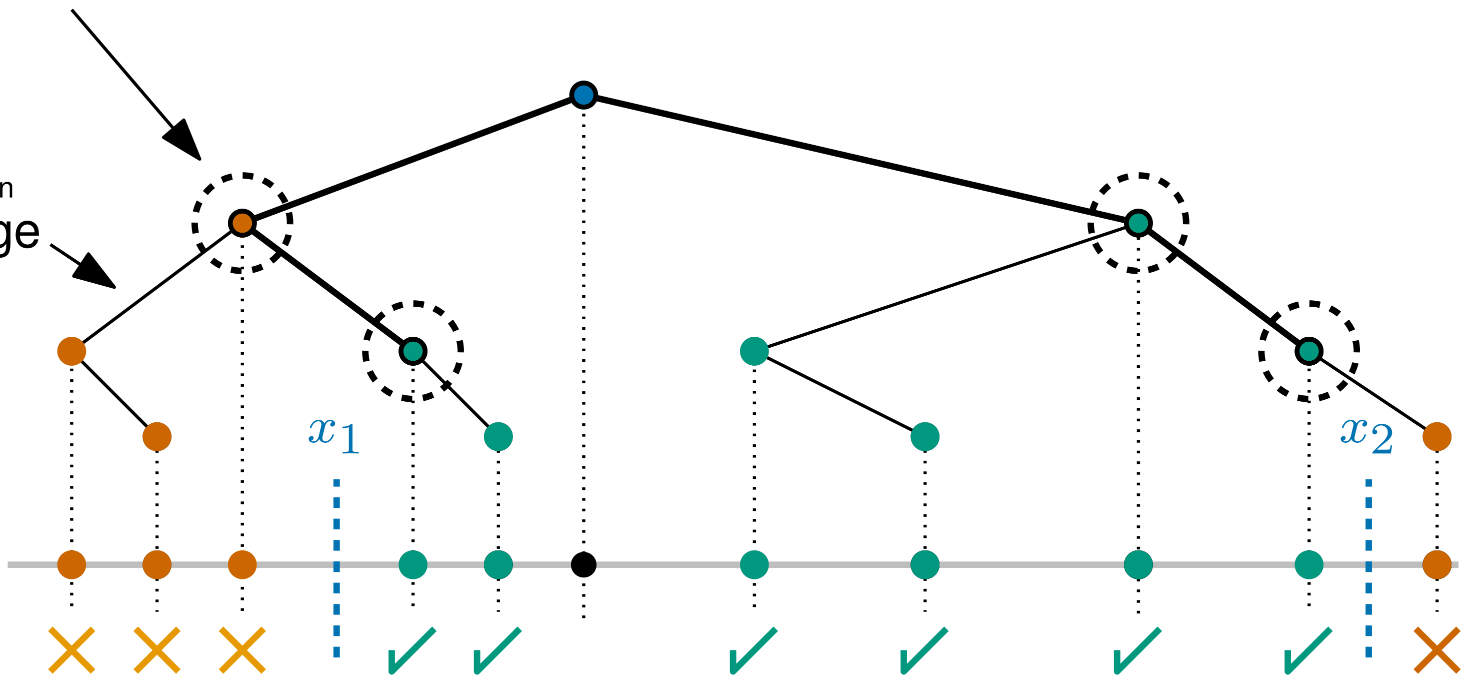
Starting simple. . . 1D range searching

how do we do a lookup?

look at any node on the path

this is called an
off-path edge

"it's all or nothing"



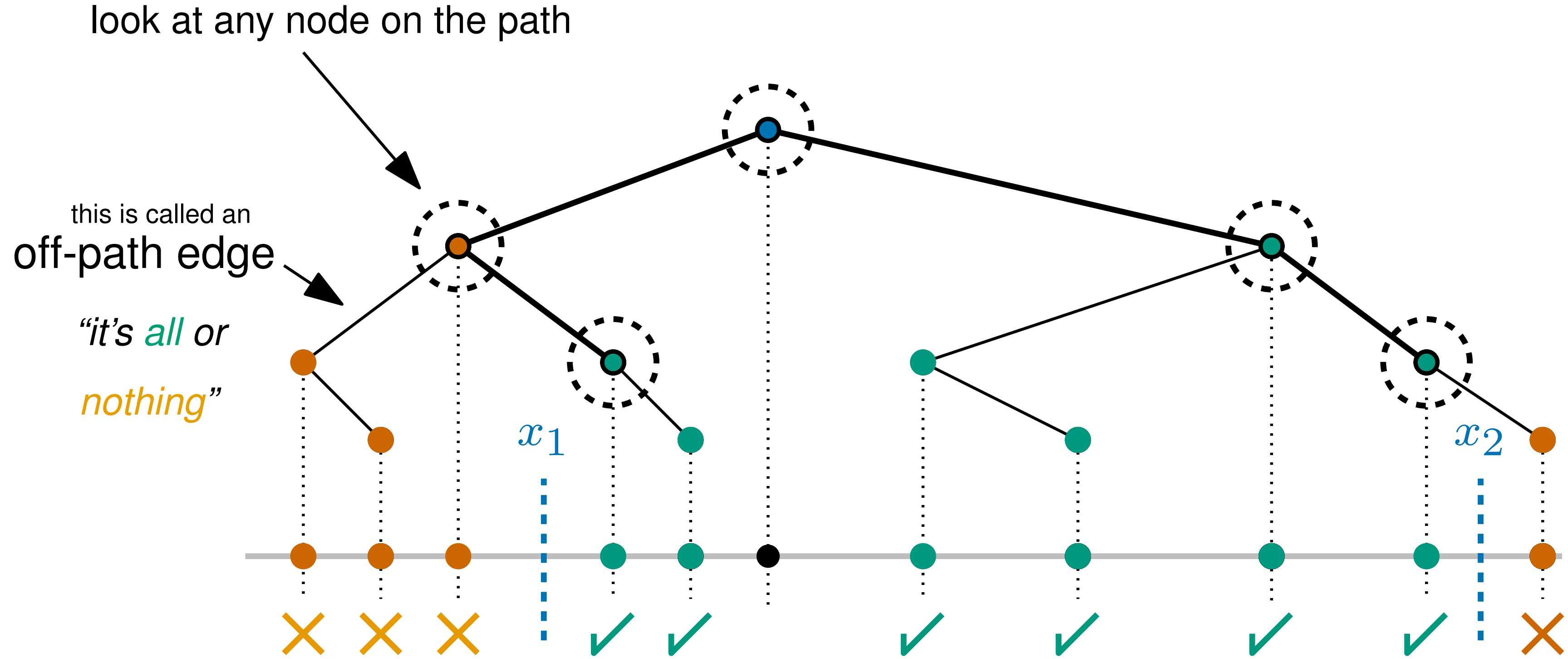
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

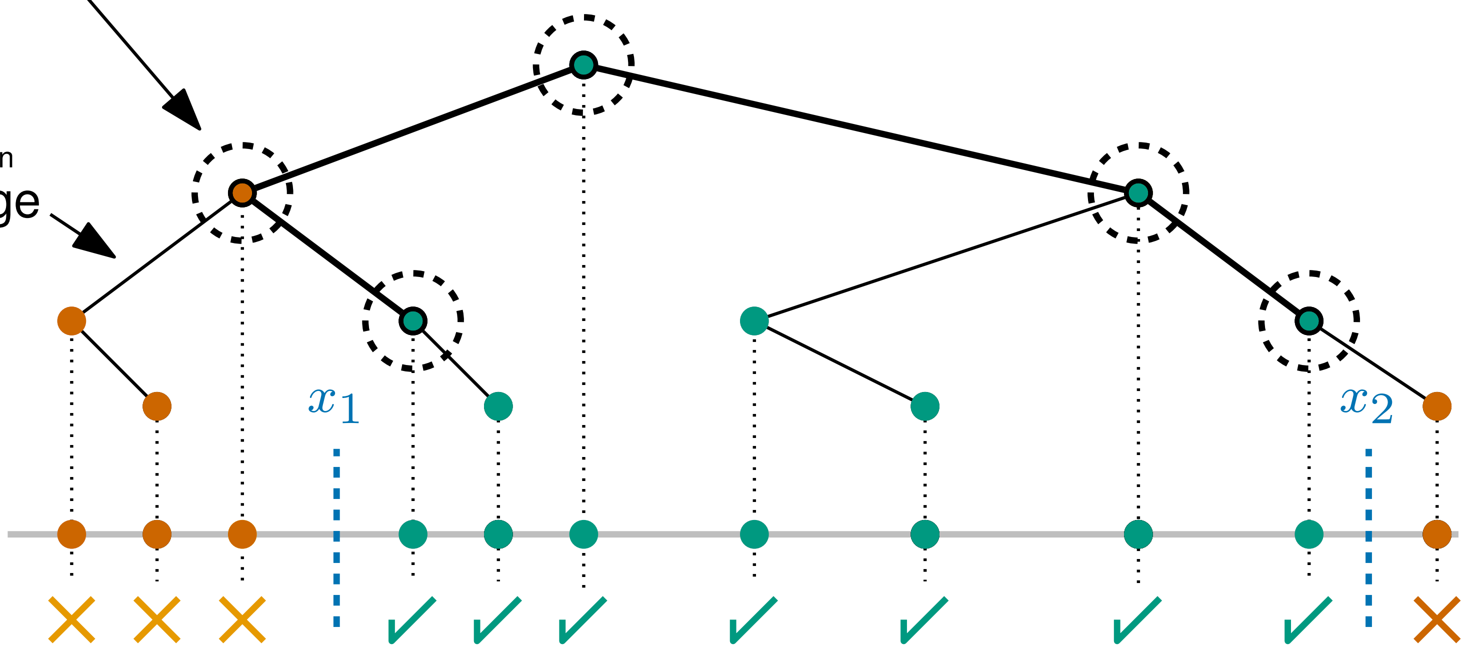
Starting simple. . . 1D range searching

how do we do a lookup?

look at any node on the path

this is called an off-path edge

"it's all or nothing"



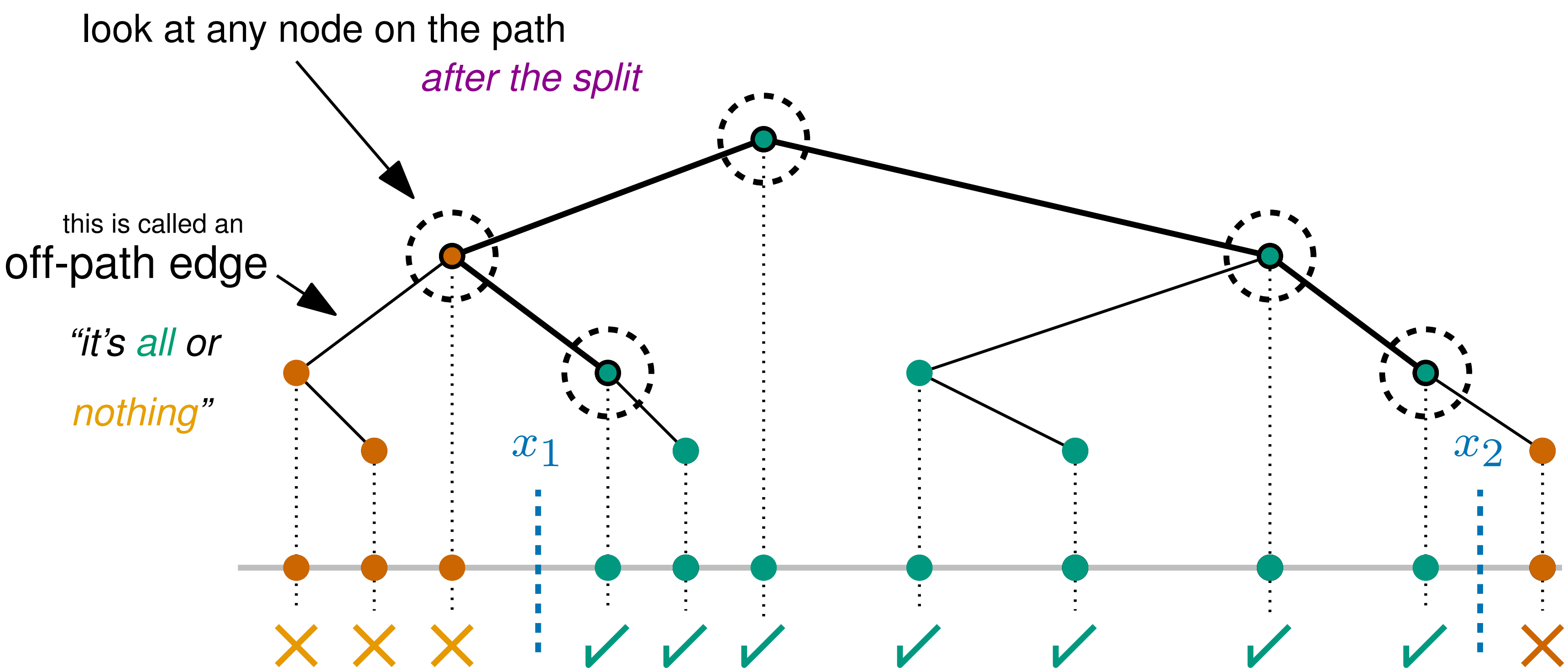
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



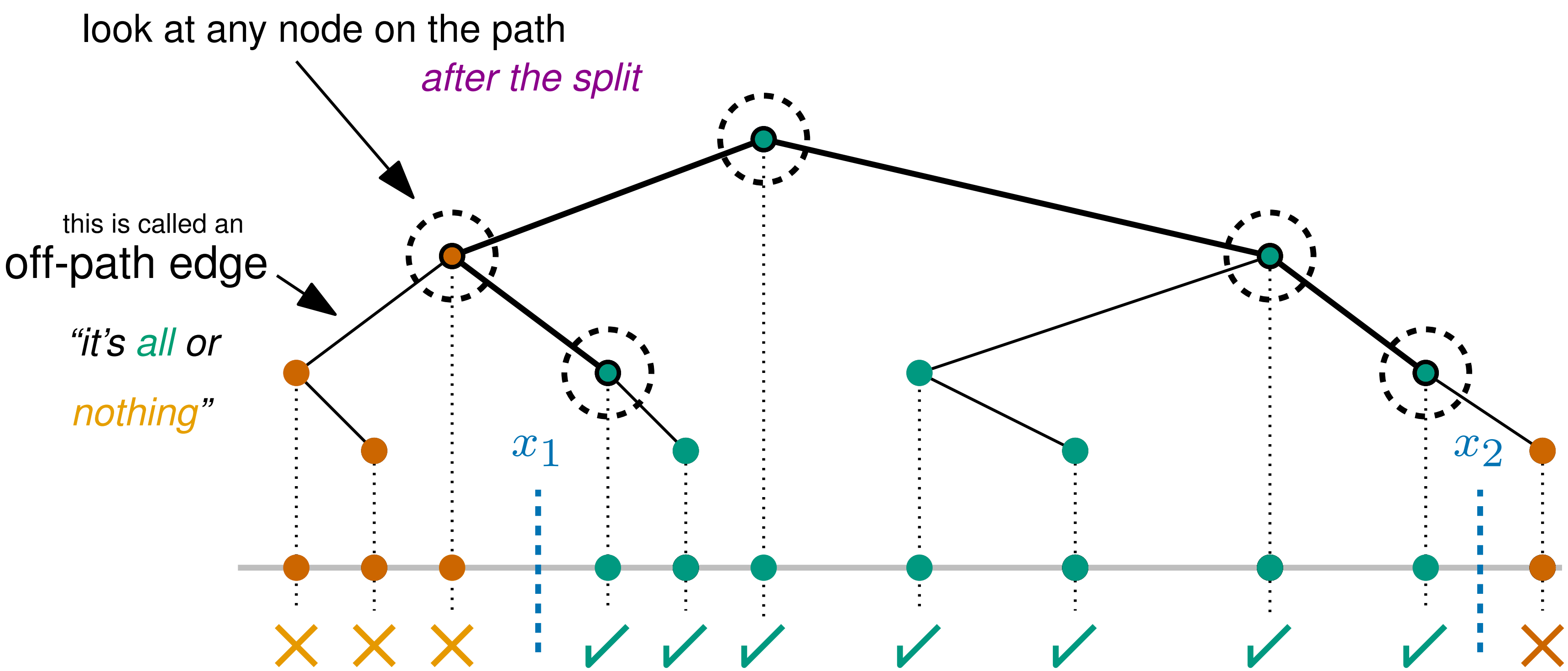
Step 1: find the successor of x_1 in $O(\log n)$ time

Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

Starting simple... 1D range searching

how do we do a lookup?



Step 1: find the successor of x_1 in $O(\log n)$ time

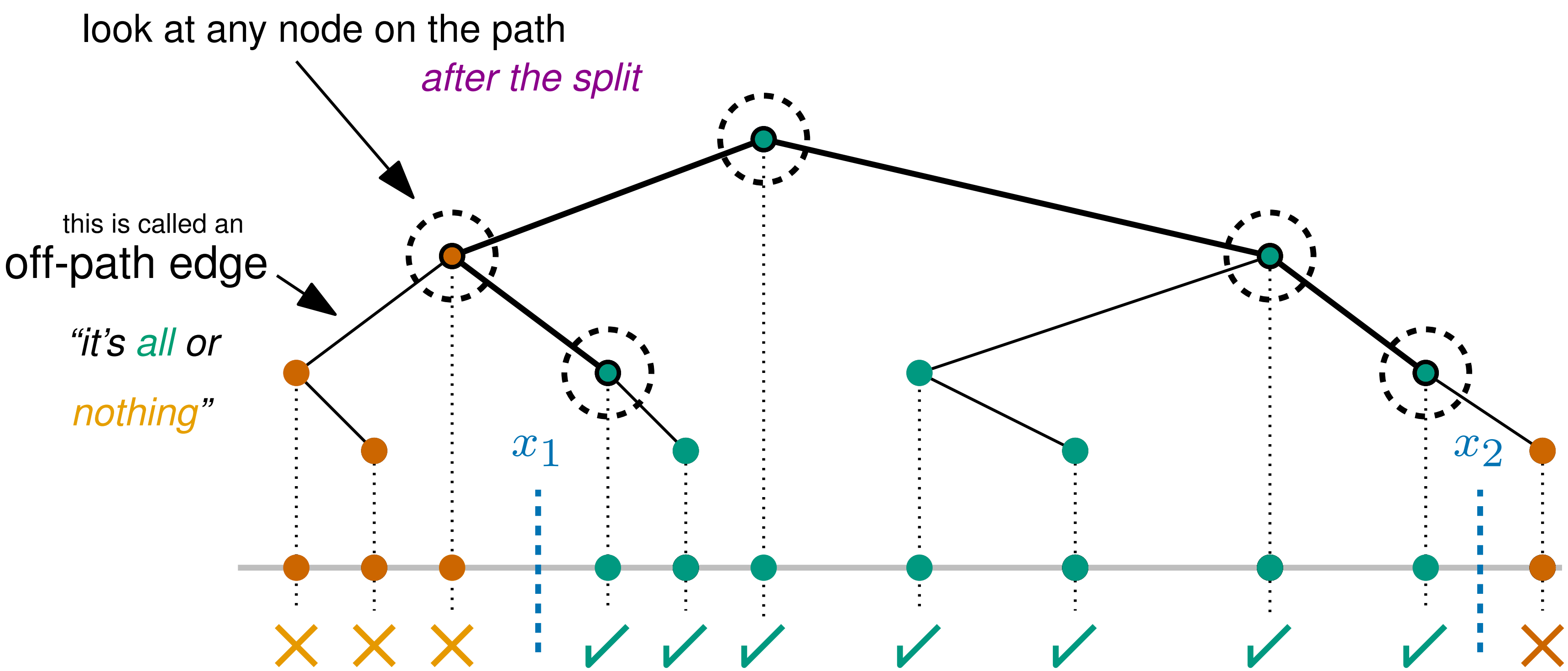
Step 2: find the predecessor of x_2 in $O(\log n)$ time

which points in the tree should we output?

those in the $O(\log n)$ selected subtrees on the path

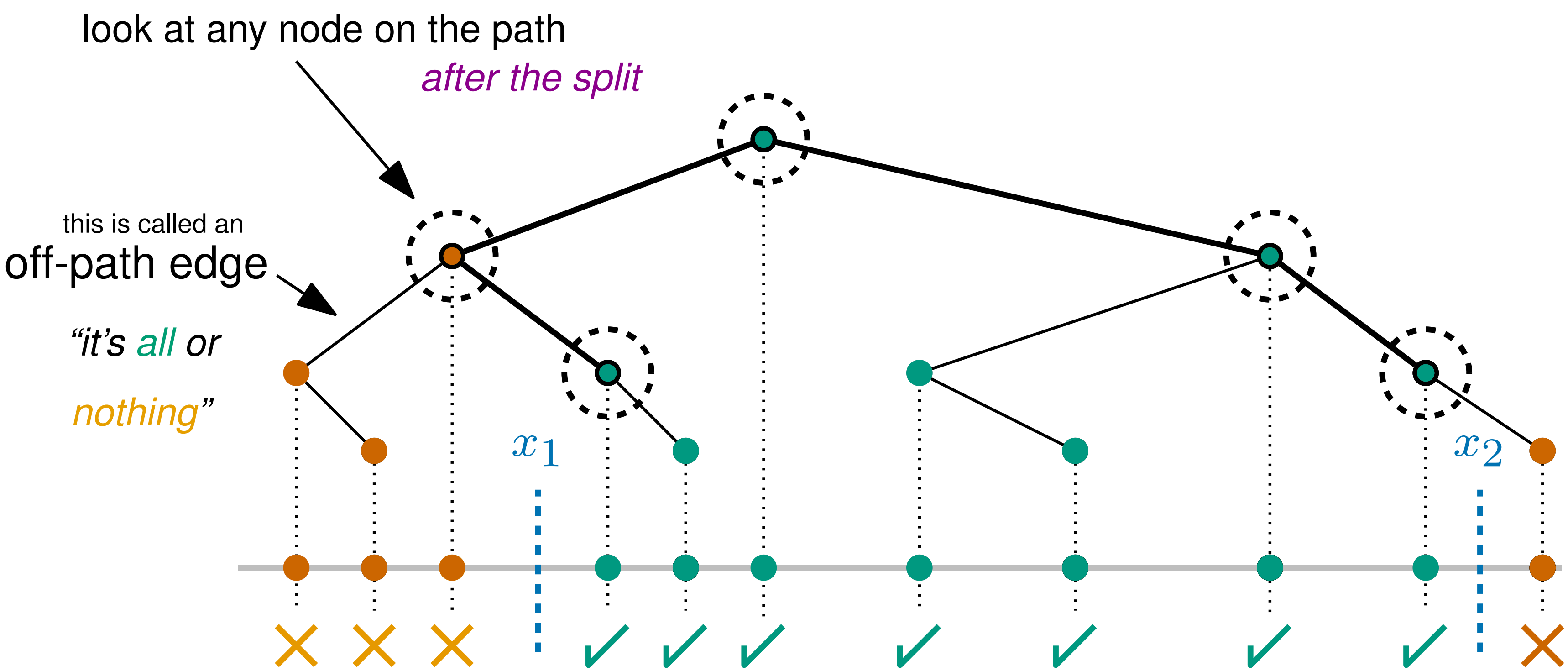
Starting simple. . . 1D range searching

how do we do a lookup?



Starting simple. . . 1D range searching

how do we do a lookup?



as before

lookups take $O(\log n + k)$ time (k is the number of points reported)

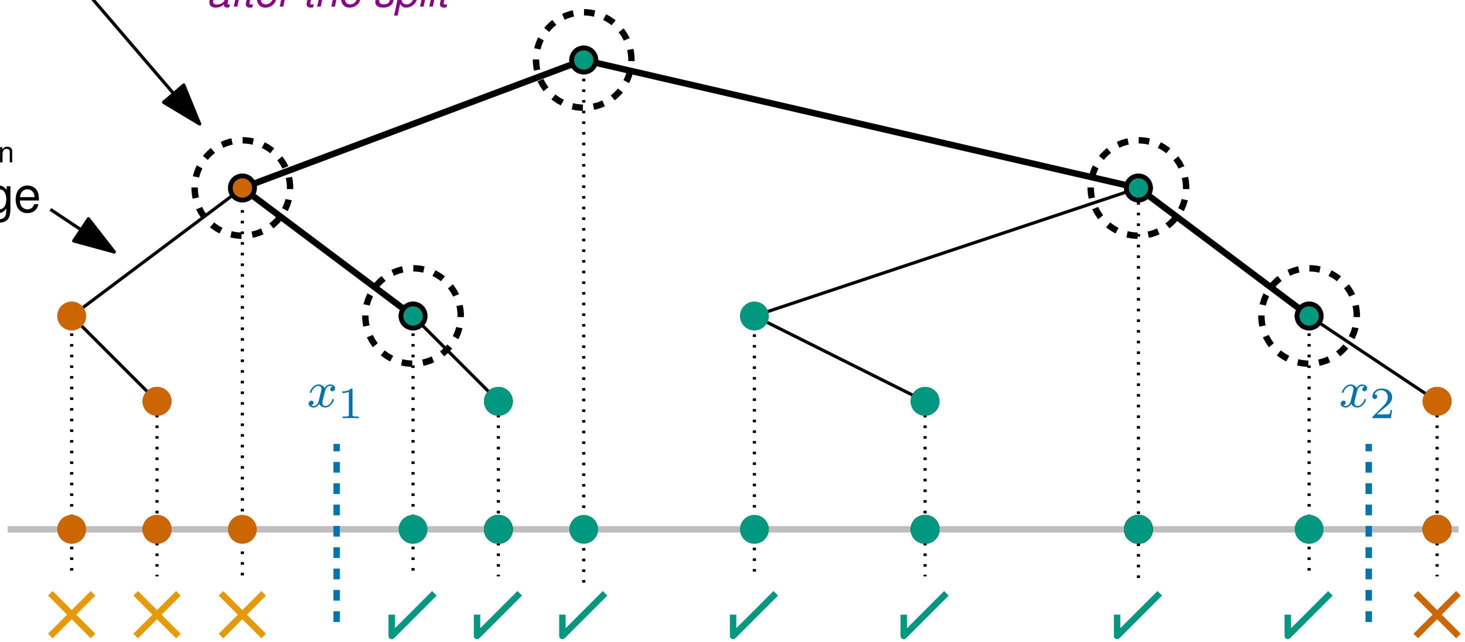
Starting simple. . . 1D range searching

how do we do a lookup?

look at any node on the path *after the split*

this is called an off-path edge

“it’s all or nothing”



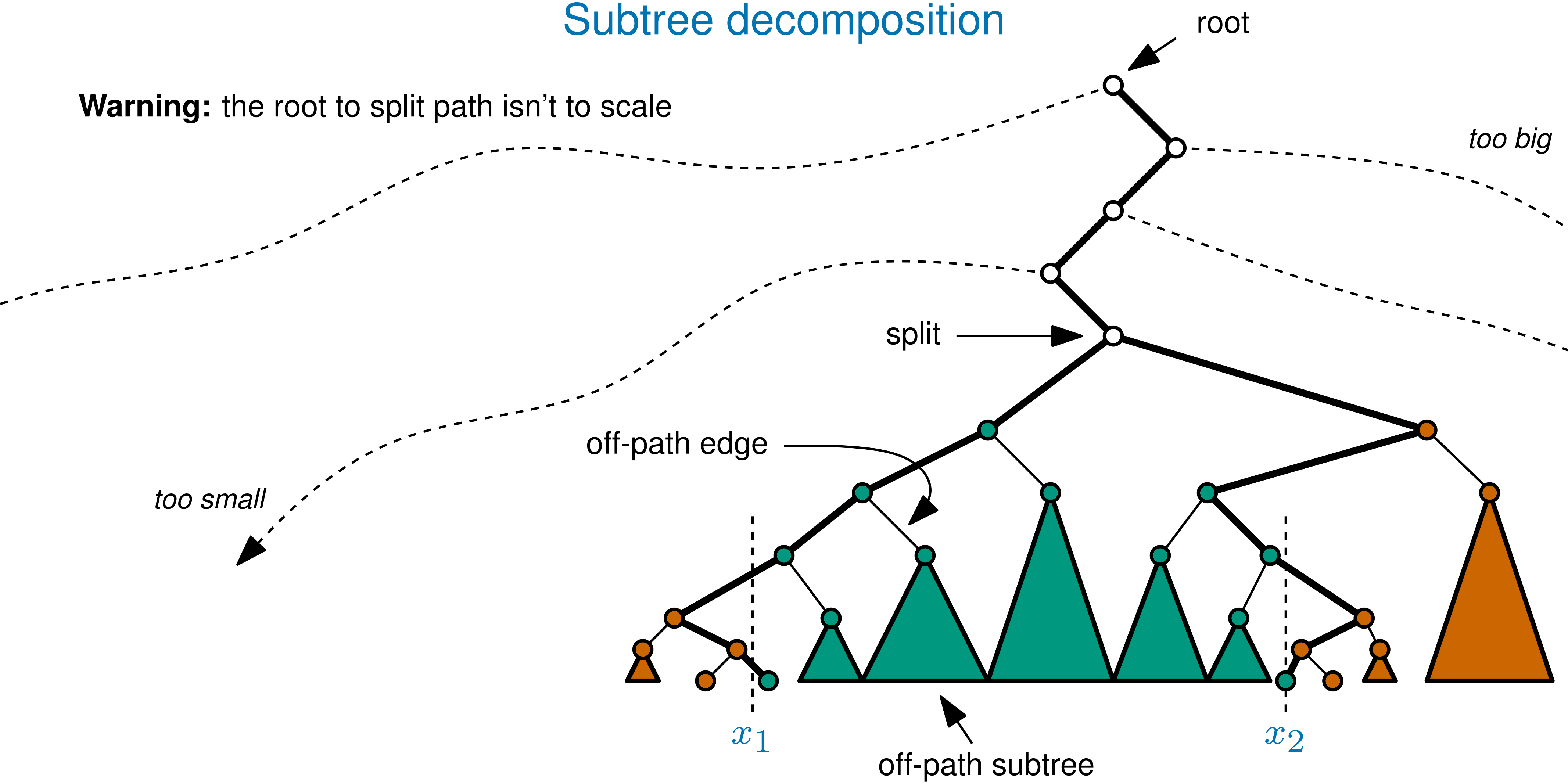
as before

lookups take $O(\log n + k)$ time (k is the number of points reported)

so what have we gained?

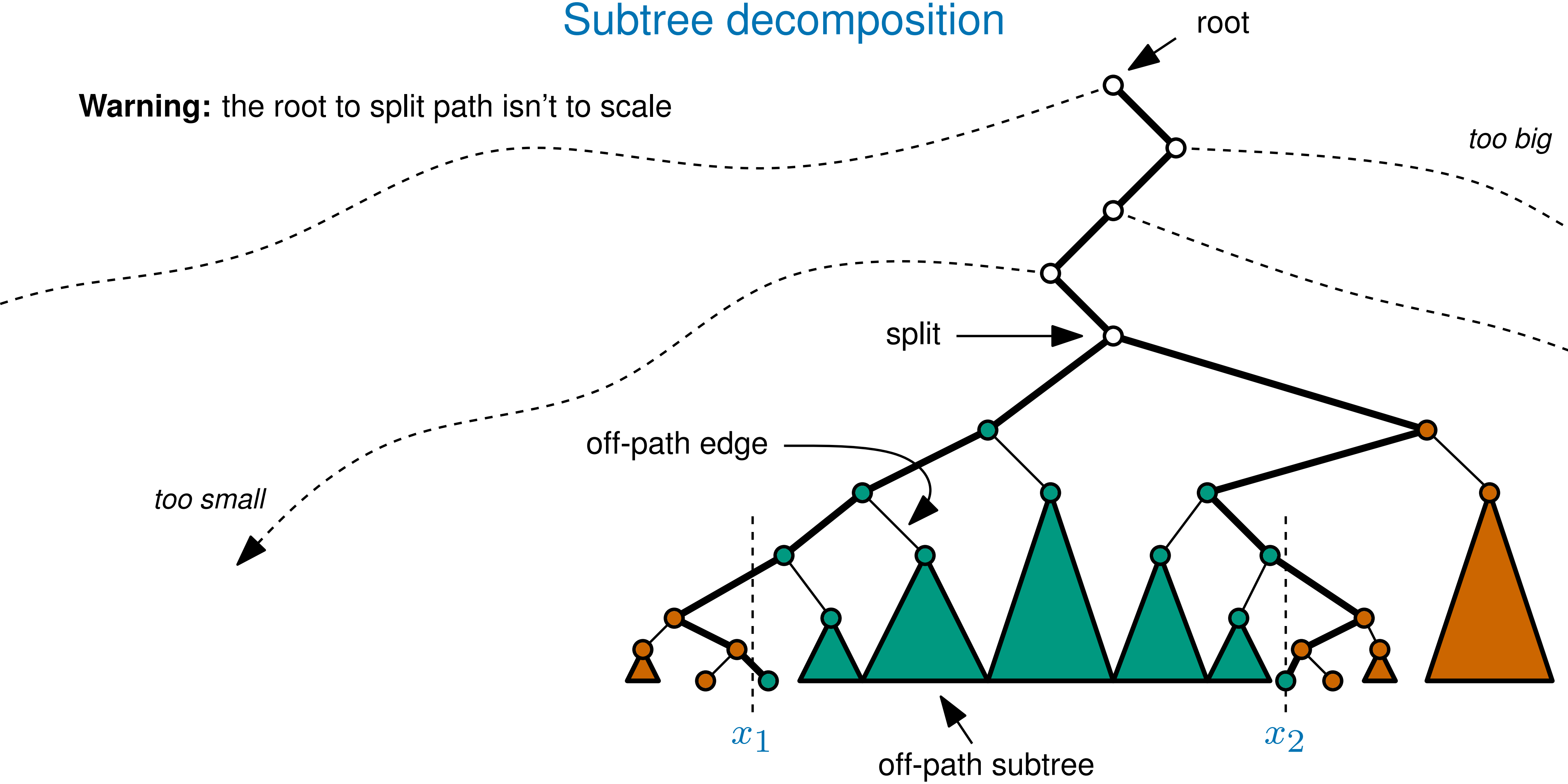
Subtree decomposition

Warning: the root to split path isn't to scale



Subtree decomposition

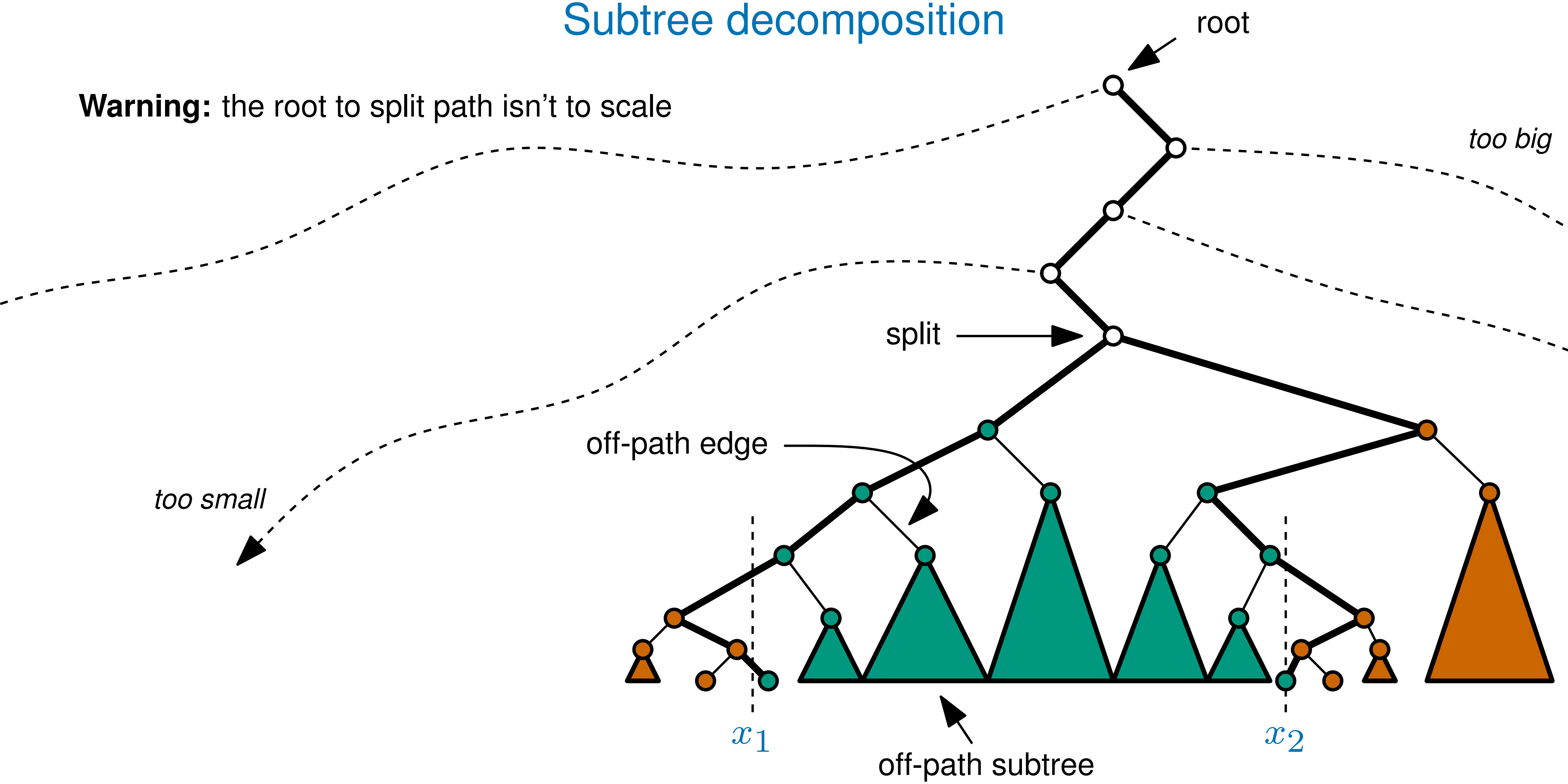
Warning: the root to split path isn't to scale



after the paths to x_1 and x_2 split. . .

Subtree decomposition

Warning: the root to split path isn't to scale

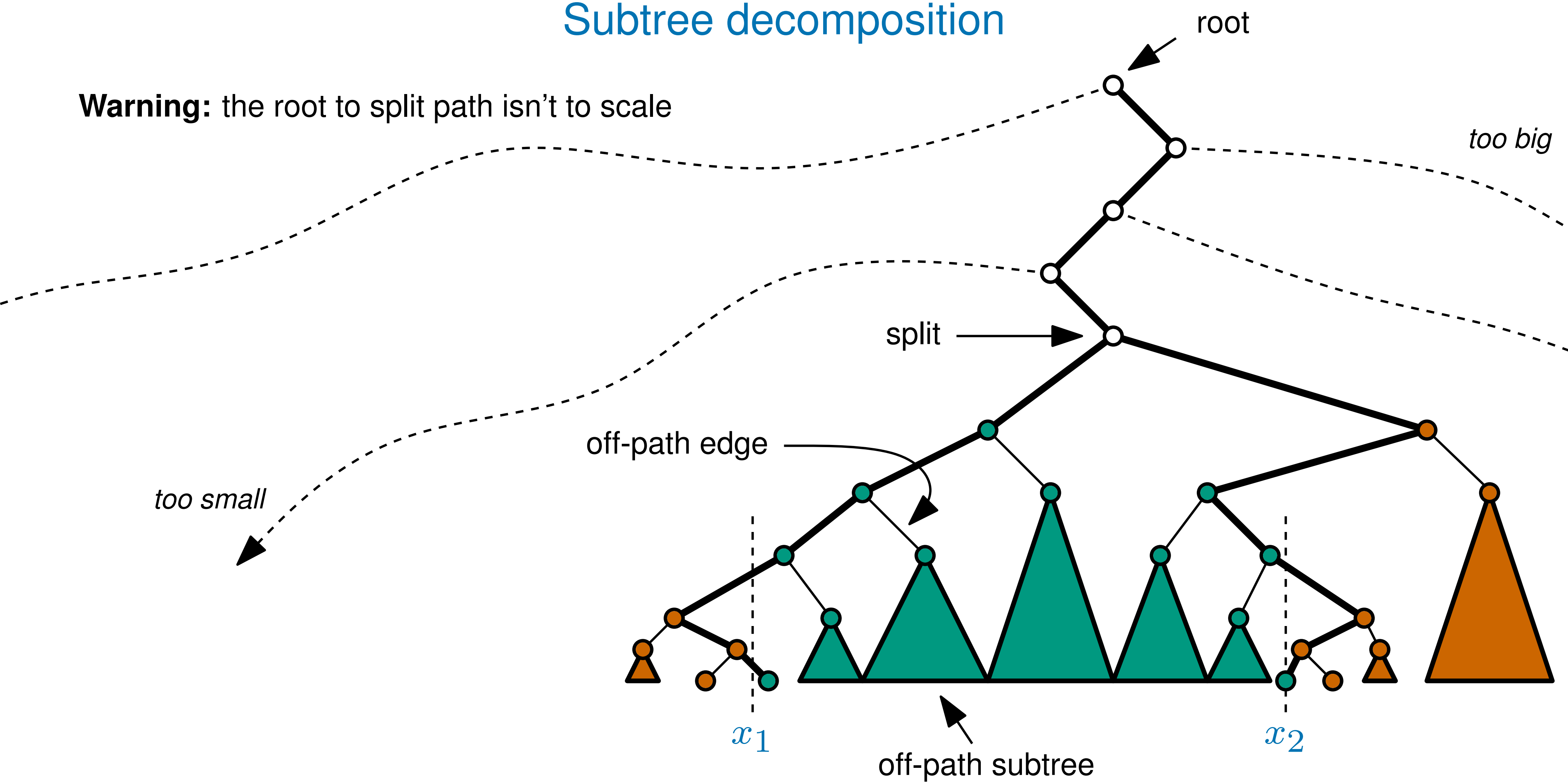


after the paths to x_1 and x_2 split. . .

any off-path subtree is either *in or out*

Subtree decomposition

Warning: the root to split path isn't to scale



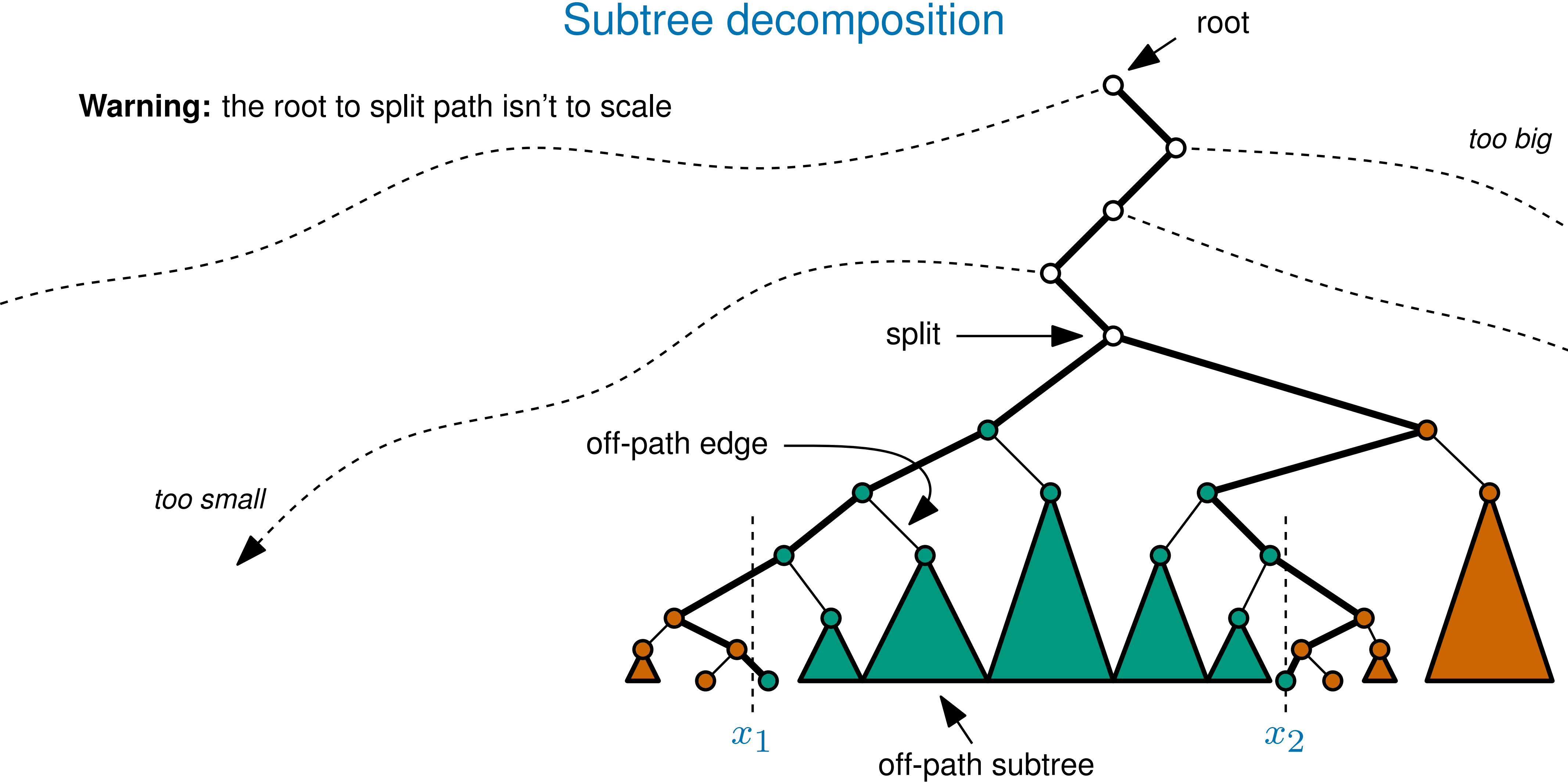
after the paths to x_1 and x_2 split...

any off-path subtree is either *in* or *out*

i.e. every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or none has

Subtree decomposition

Warning: the root to split path isn't to scale



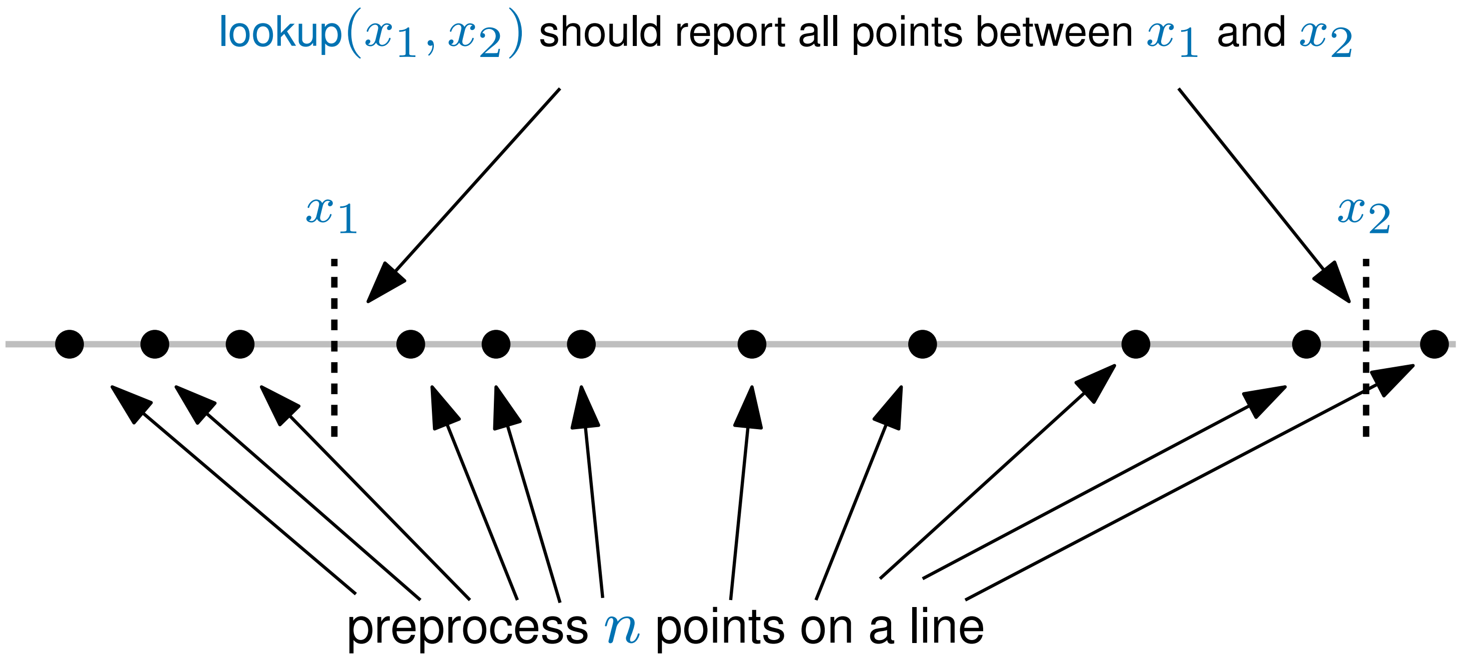
after the paths to x_1 and x_2 split...

any off-path subtree is either *in* or *out*

i.e. every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or none has

this will be useful for 2D range searching

1D range searching summary



$O(n \log n)$ prep time

$O(n)$ space

$O(\log n + k)$ lookup time

where k is the number of points reported

(this is known as being output sensitive)

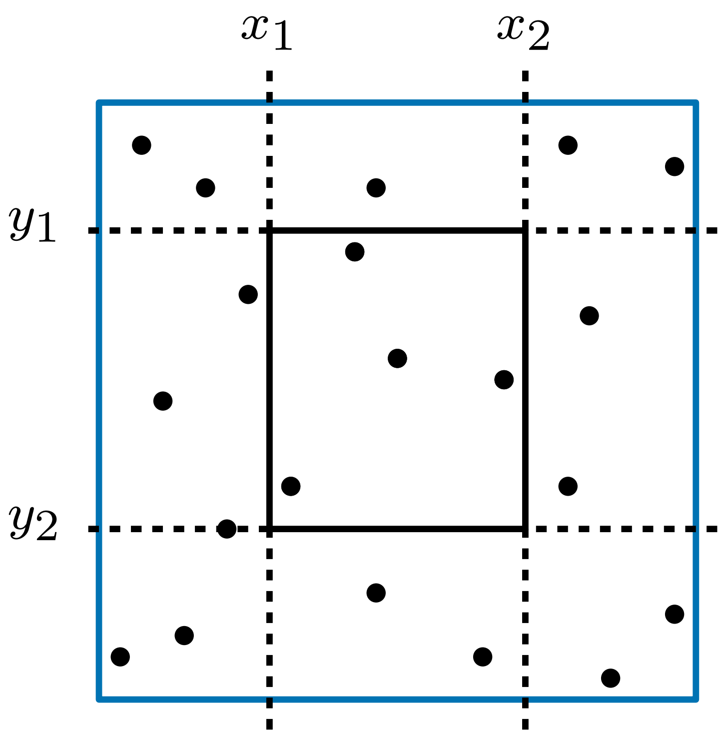
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



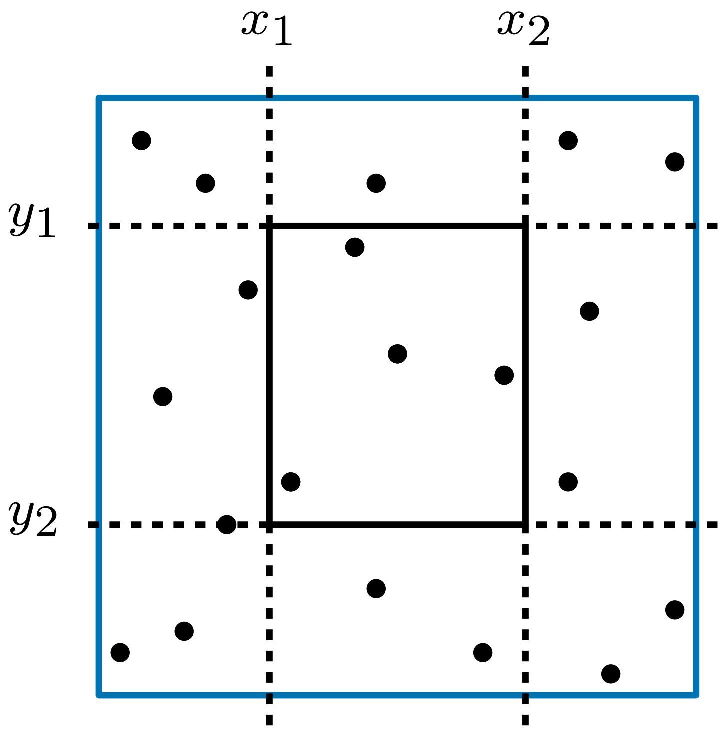
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

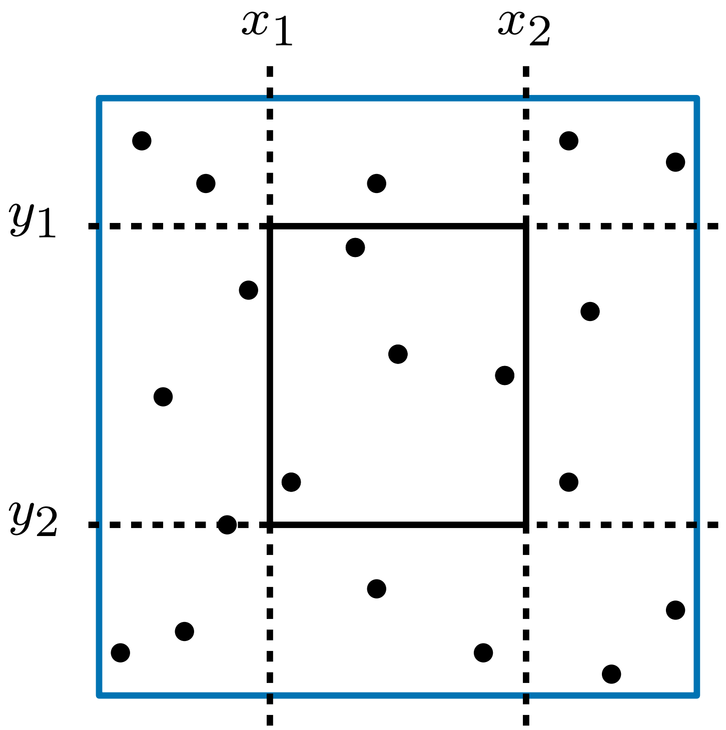
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$

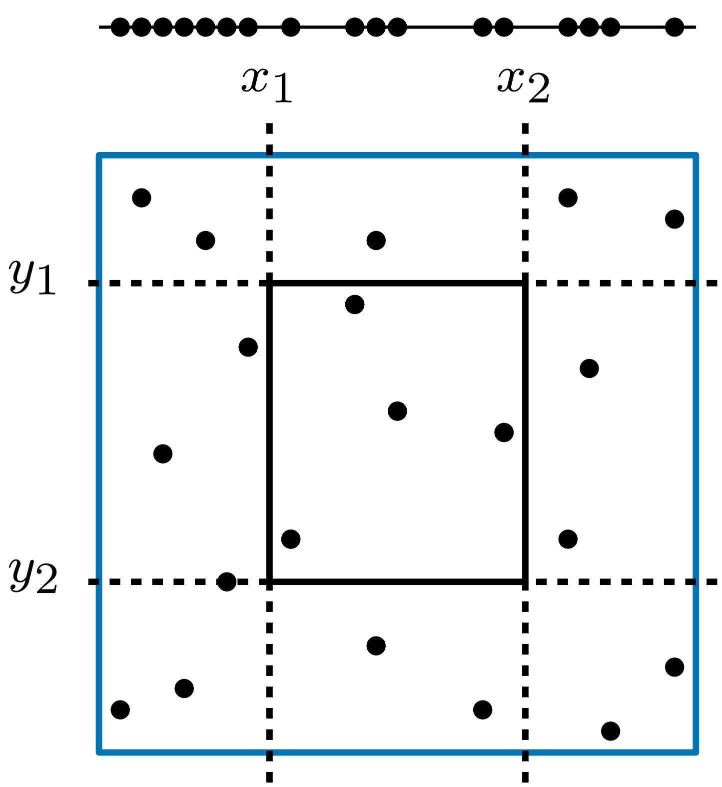
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$

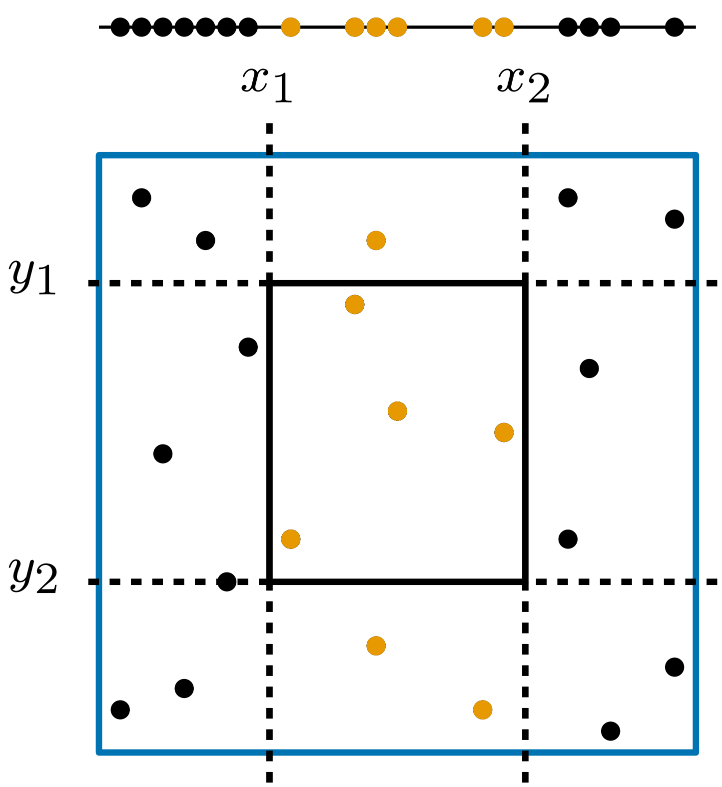
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$

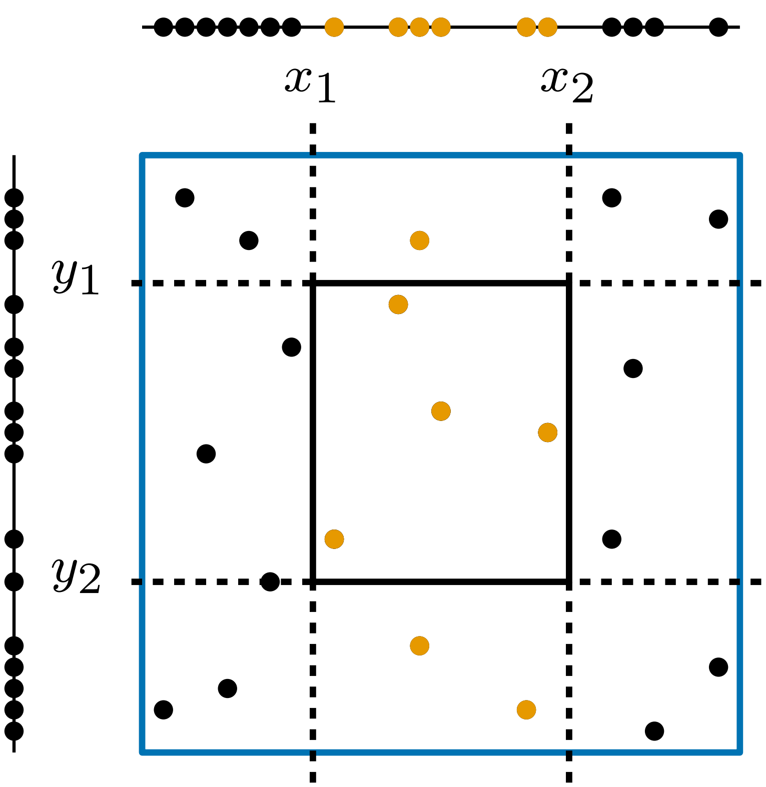
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$

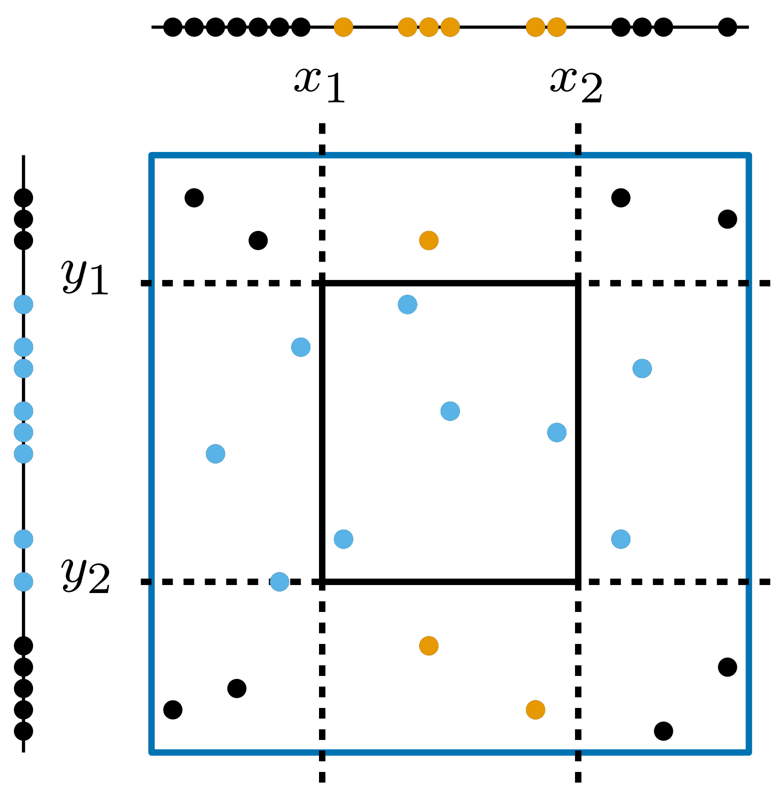
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$

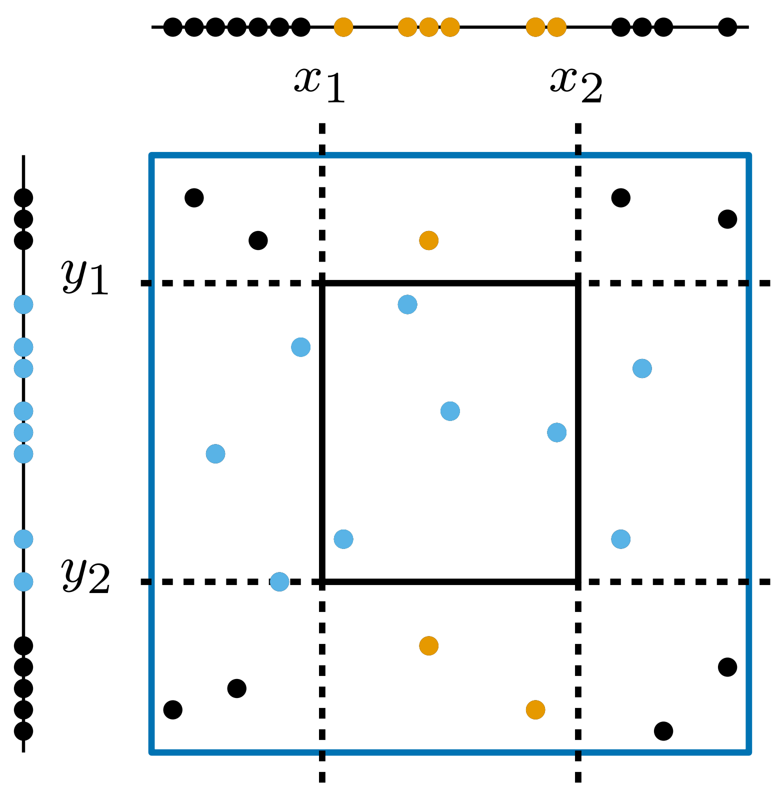
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

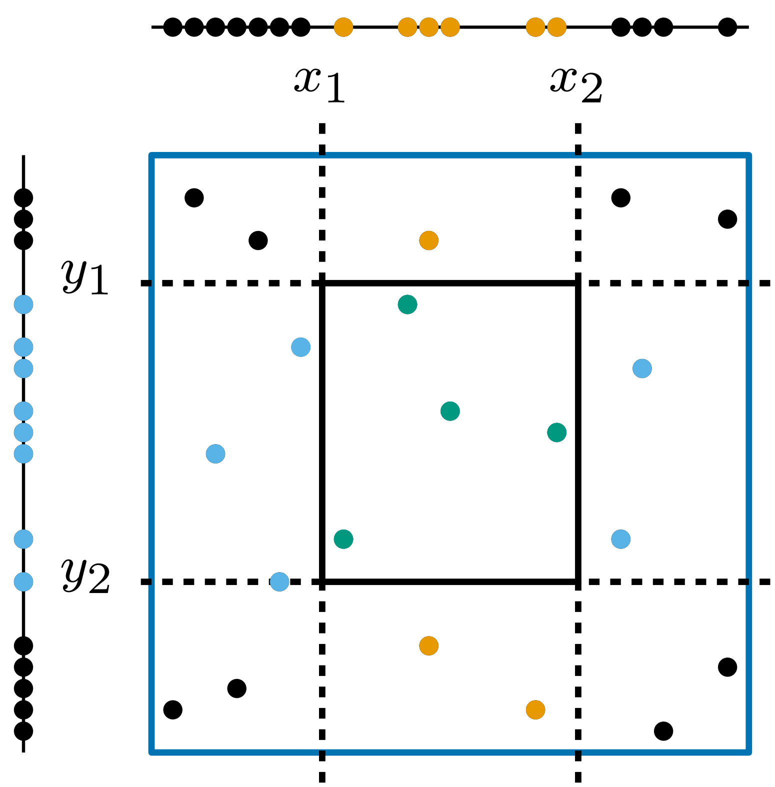
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

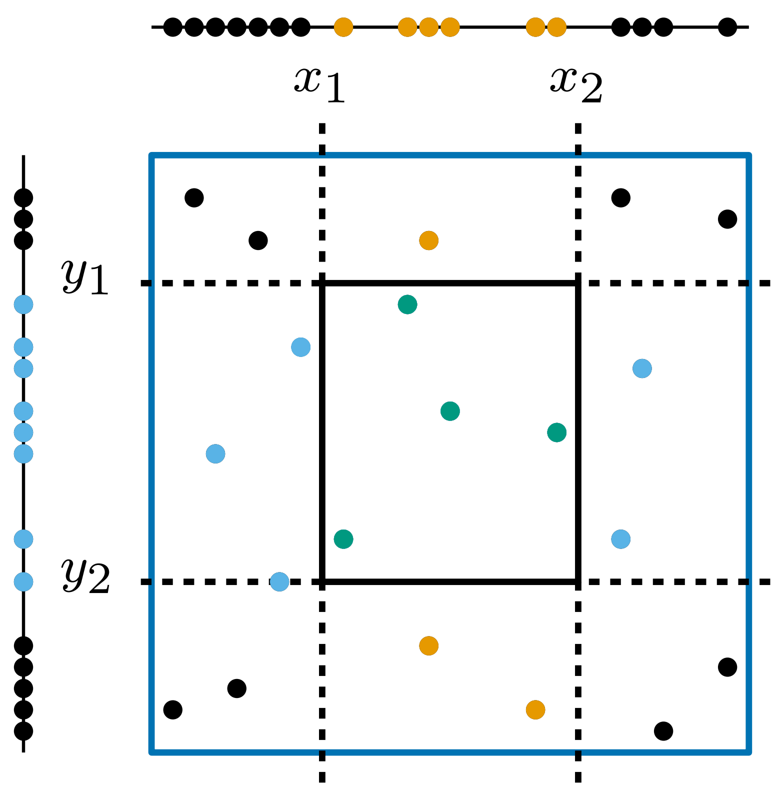
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

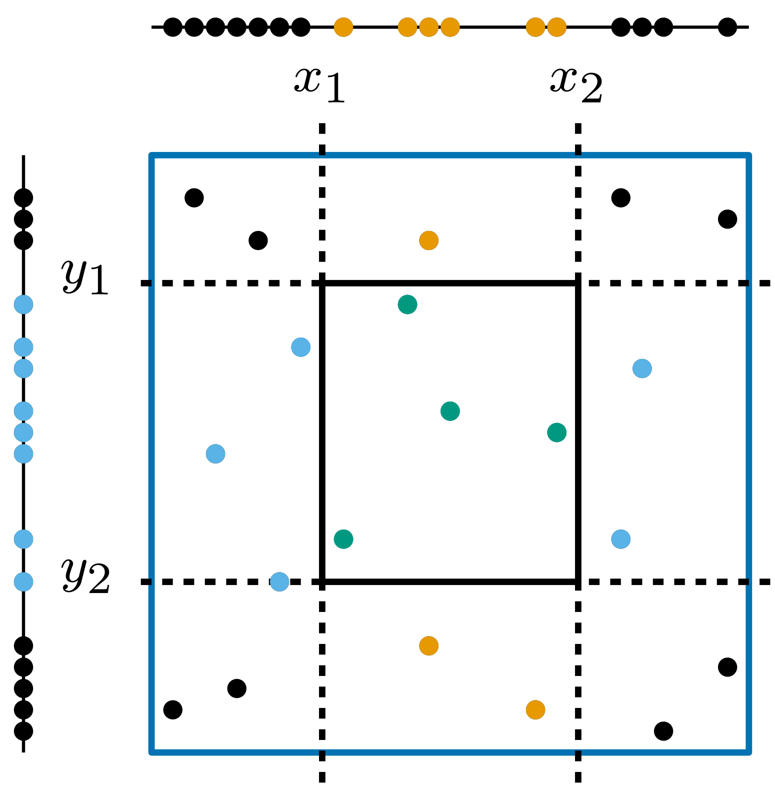
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k) + O(\log n + k) + O(k)$$

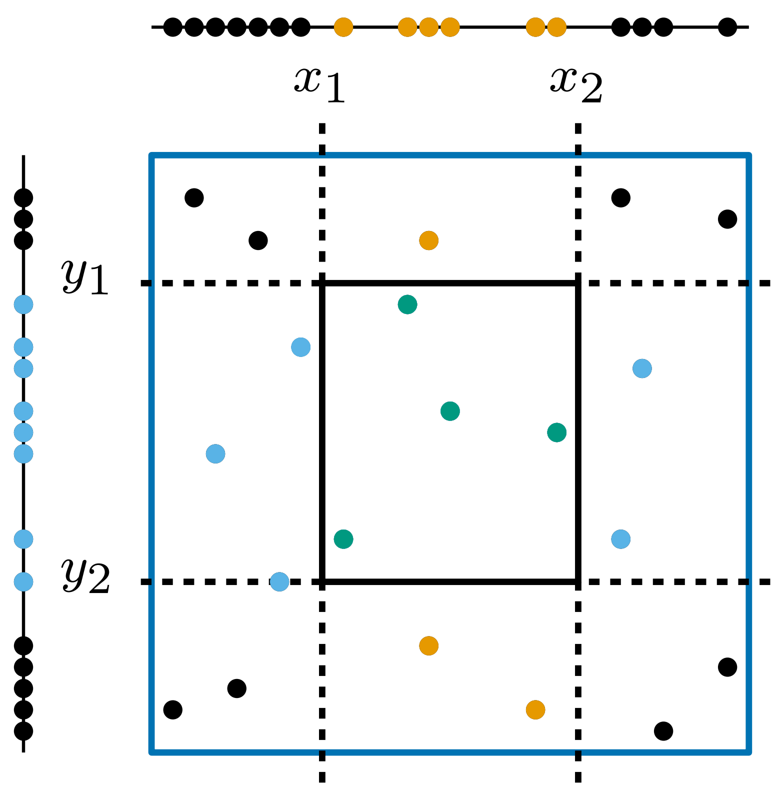
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k) + O(\log n + k) + O(k)$$
$$= O(\log n + k)$$

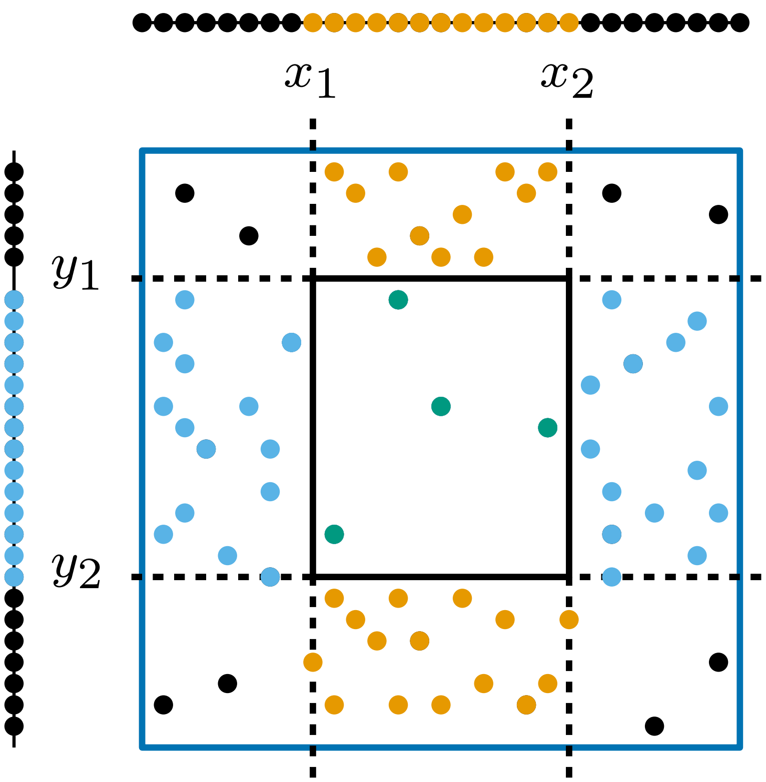
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k) + O(\log n + k) + O(k)$$
$$= O(\log n + k)$$

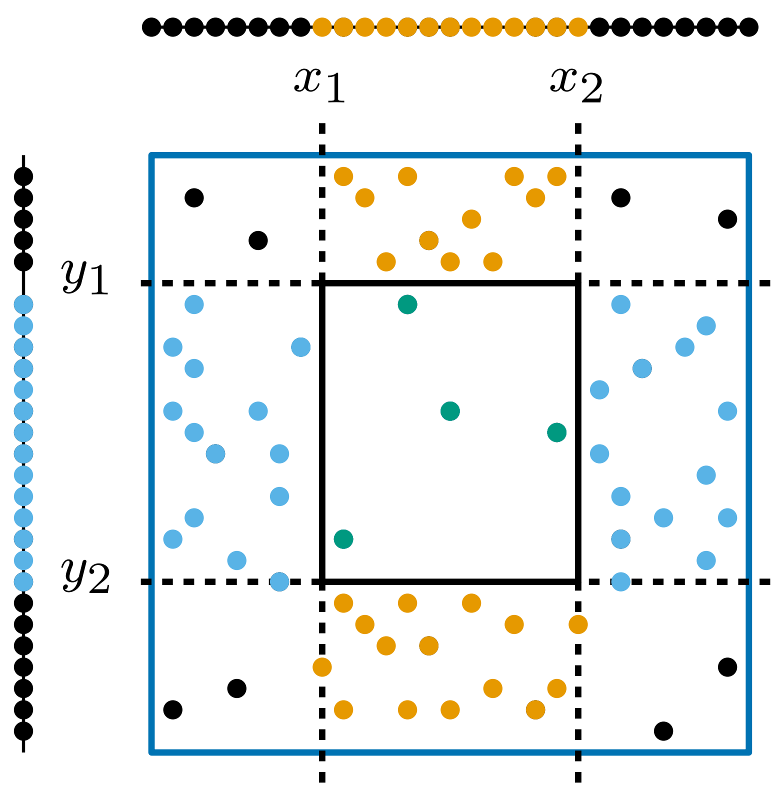
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k) + O(\log n + k) + O(k)$$
$$= O(\log n + k)$$

???

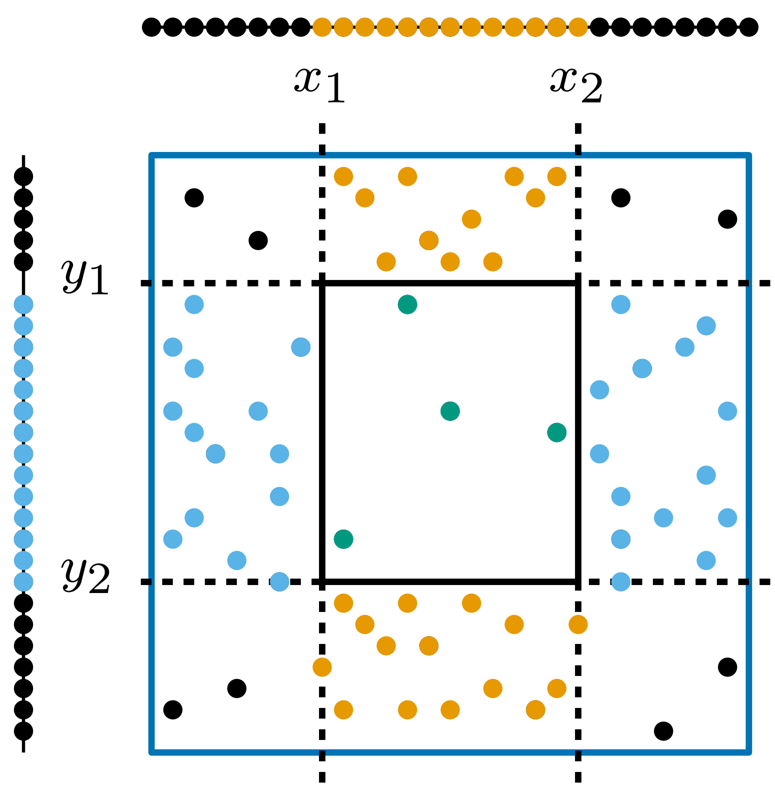
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k) + O(\log n + k) + O(k) = O(\log n + k)$$

???

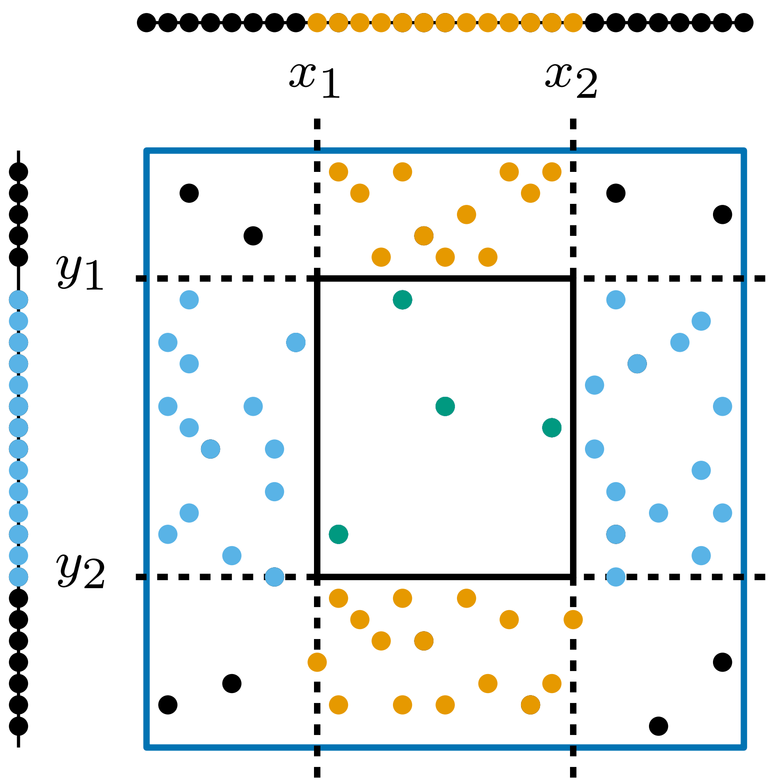
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k_x) + O(\log n + k_y) + O(k_x + k_y)$$
$$= O(\log n + k_x + k_y)$$

here k_x is the number of points with $x_1 \leqslant x \leqslant x_2$ (respectively for k_y)

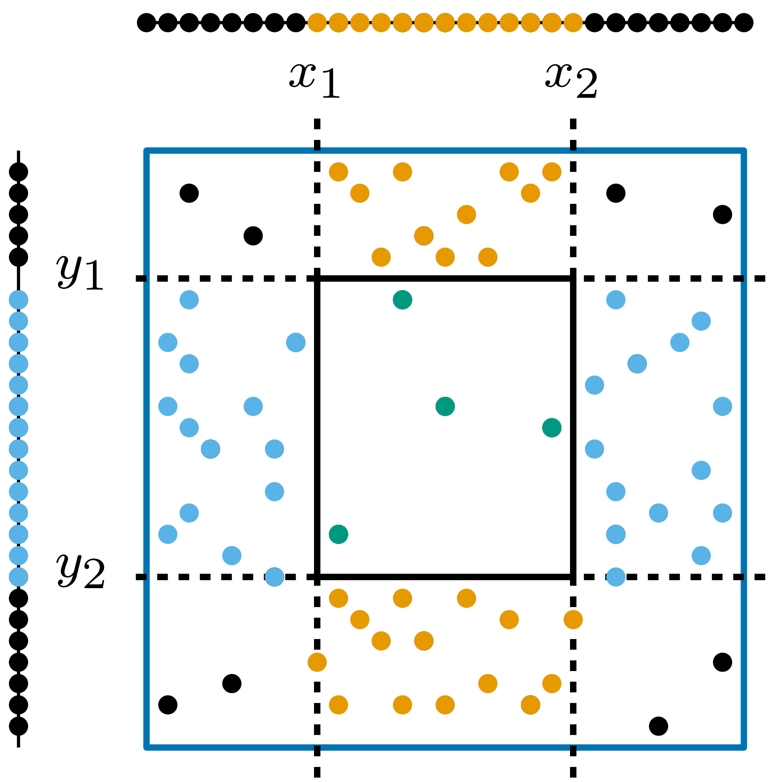
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Attempt one:

- Find all the points with $x_1 \leqslant x \leqslant x_2$
- Find all the points with $y_1 \leqslant y \leqslant y_2$
- Find all the points in both lists

How long does this take?

$$O(\log n + k_x) + O(\log n + k_y) + O(k_x + k_y)$$
$$= O(\log n + k_x + k_y)$$

these could be huge in comparison with k

here k_x is the number of points with $x_1 \leqslant x \leqslant x_2$ (respectively for k_y)

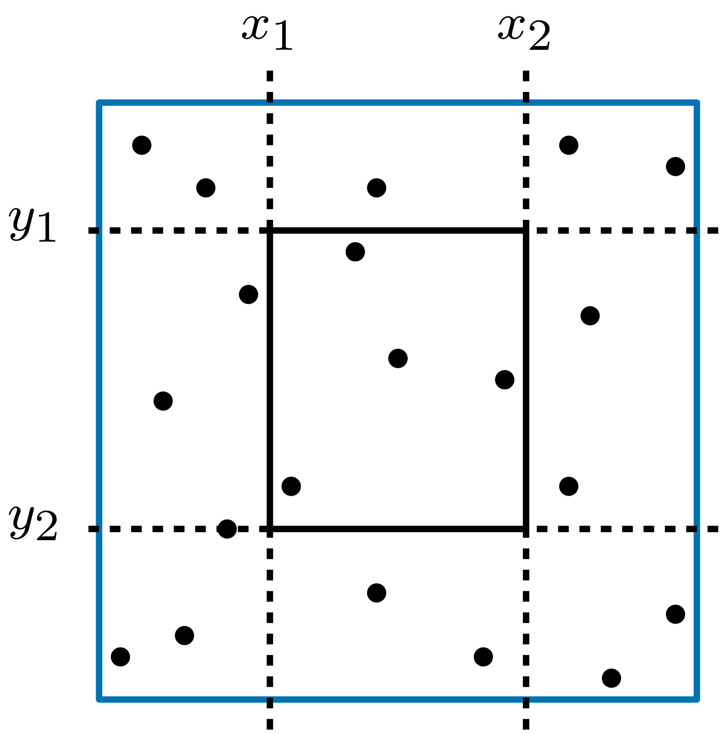
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



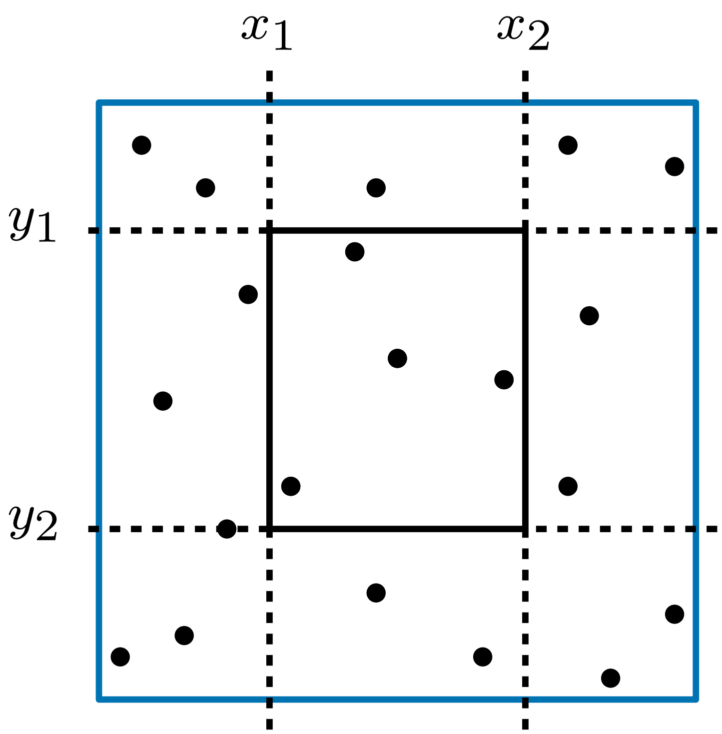
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

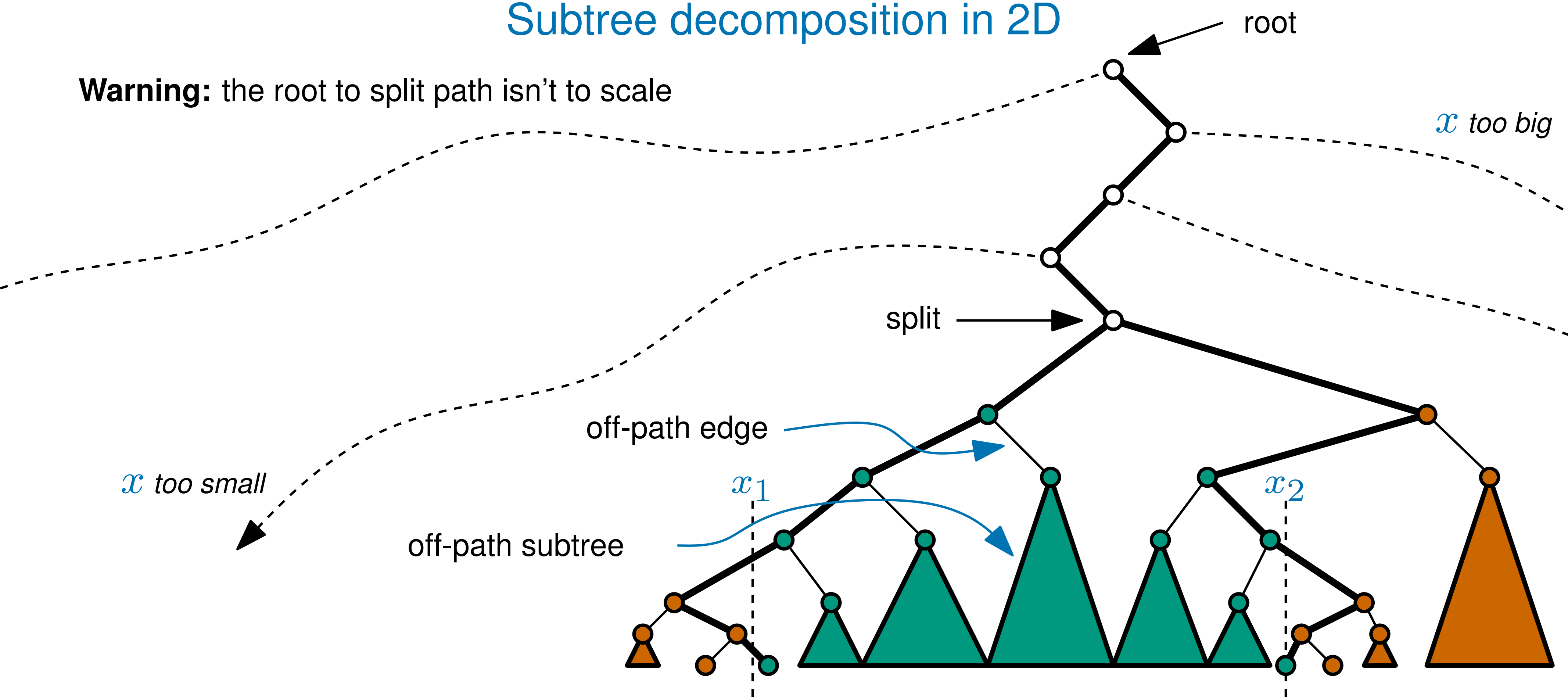
  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



how can we do better?

Subtree decomposition in 2D

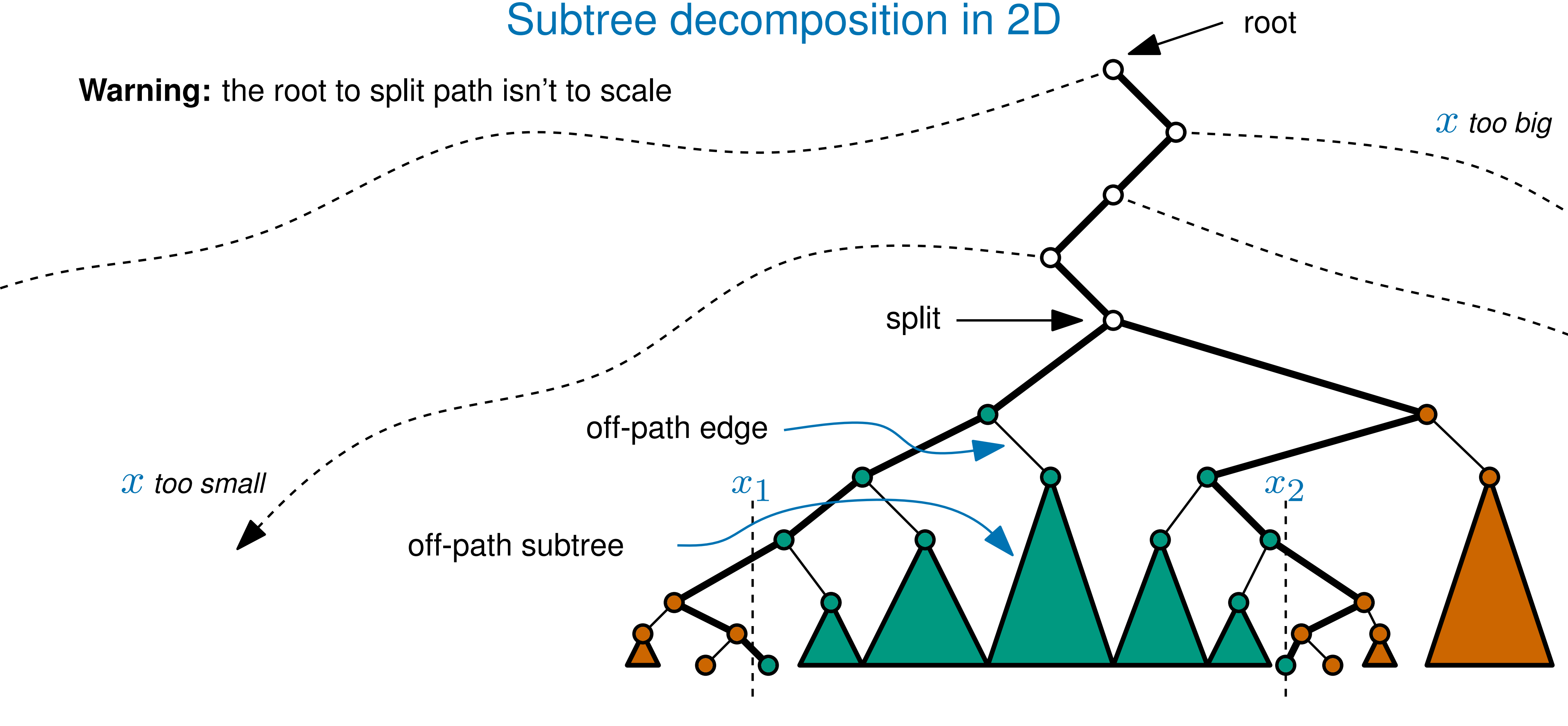
Warning: the root to split path isn't to scale



(during preprocessing) build a balanced binary tree using the x-coordinates

Subtree decomposition in 2D

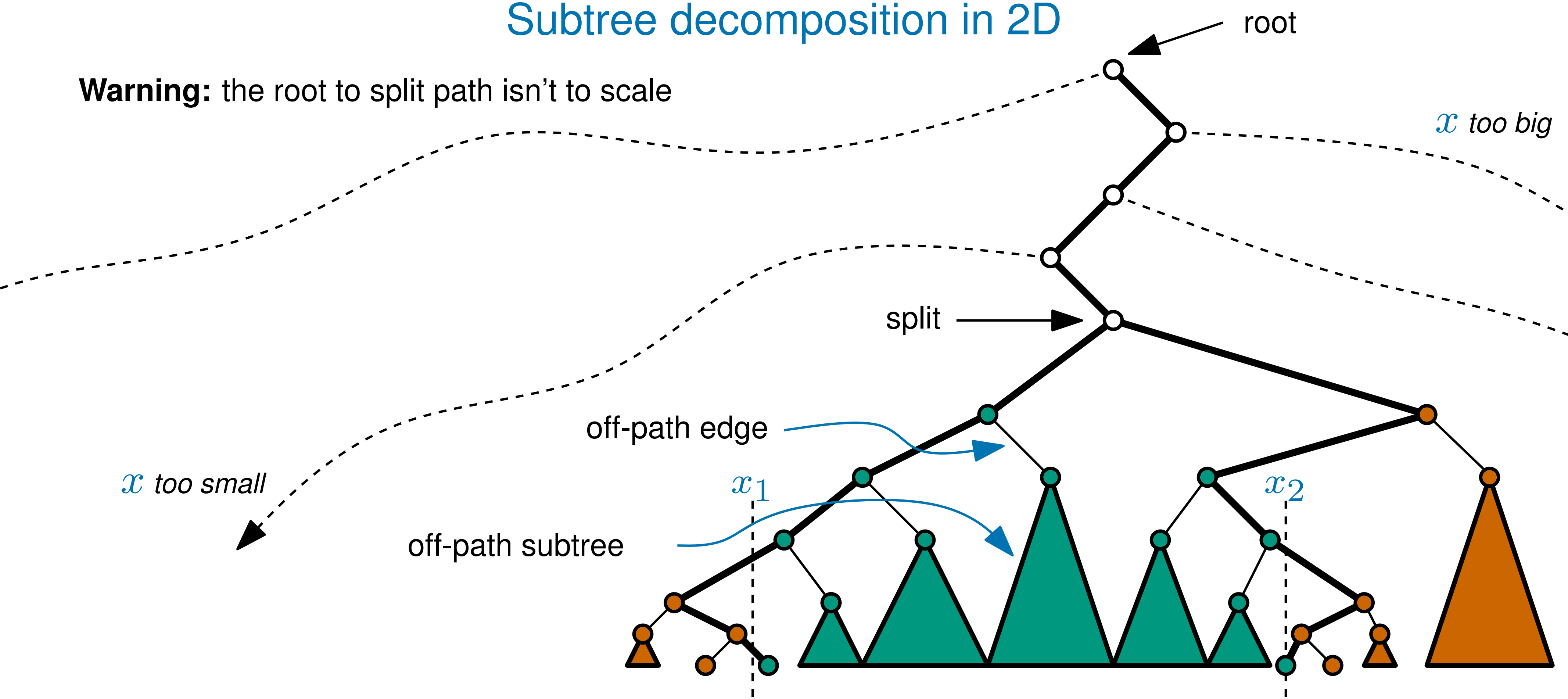
Warning: the root to split path isn't to scale



(during preprocessing) build a balanced binary tree using the x-coordinates

to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

Subtree decomposition in 2D



(during preprocessing) build a balanced binary tree using the x-coordinates

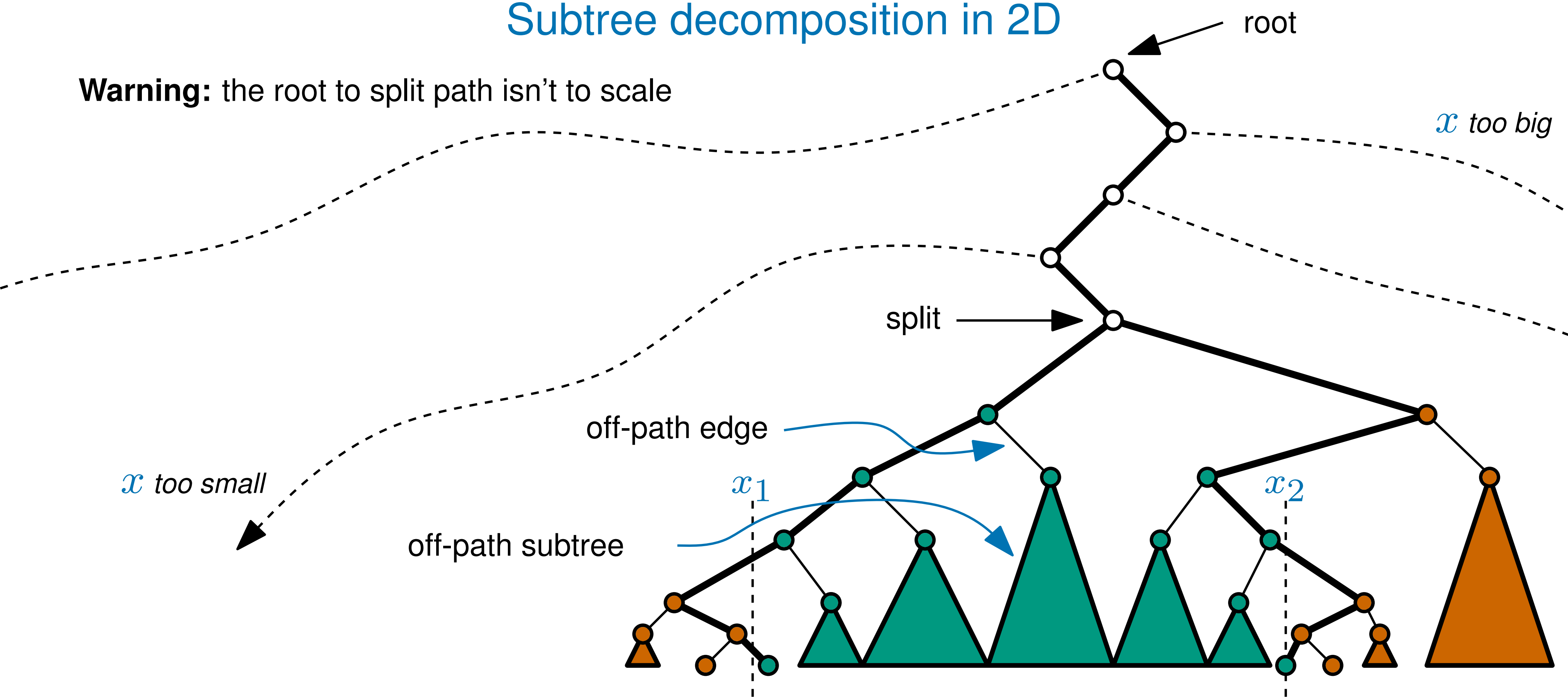
to perform a lookup(x_1, x_2, y_1, y_2) follow the paths to x_1 and x_2 as before

for any off-path subtree. . .

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Subtree decomposition in 2D

Warning: the root to split path isn't to scale



(during preprocessing) build a balanced binary tree using the x-coordinates

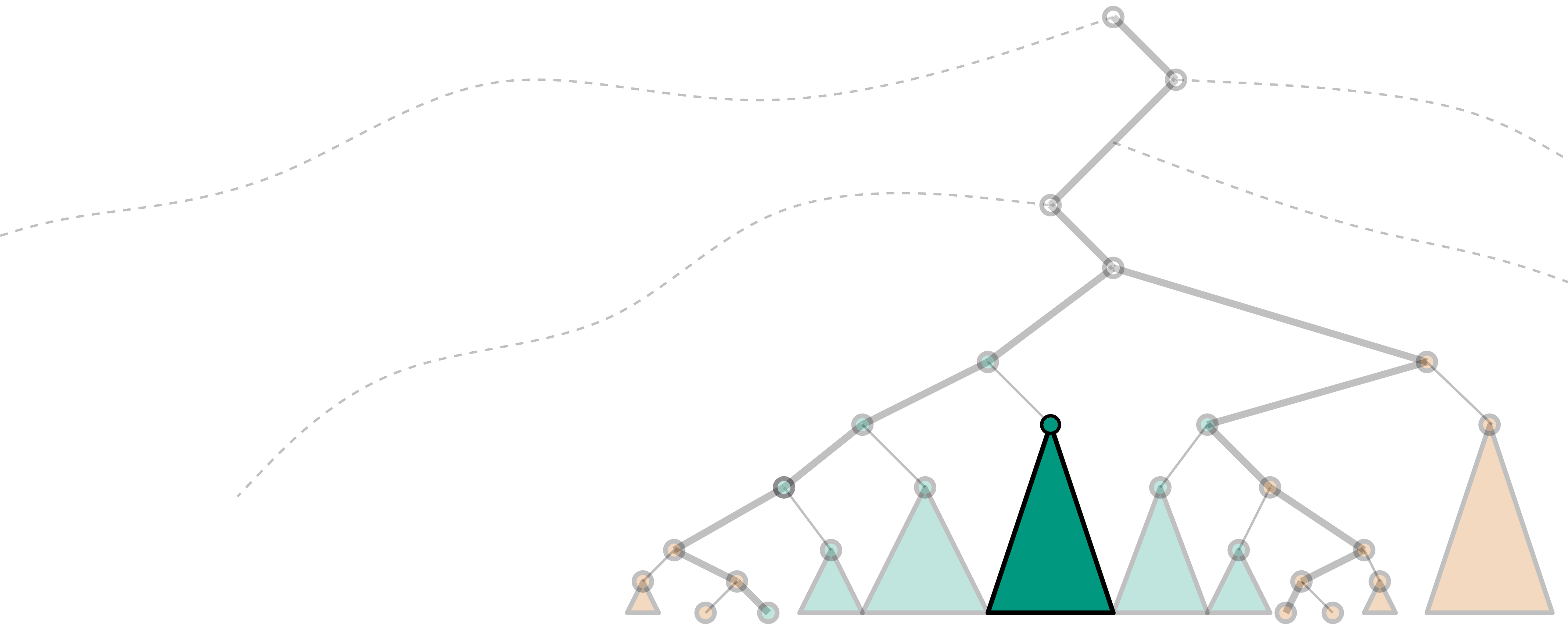
to perform a lookup(x_1, x_2, y_1, y_2) follow the paths to x_1 and x_2 as before

for any off-path subtree. . .

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D



(during preprocessing) build a balanced binary tree using the x-coordinates

to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

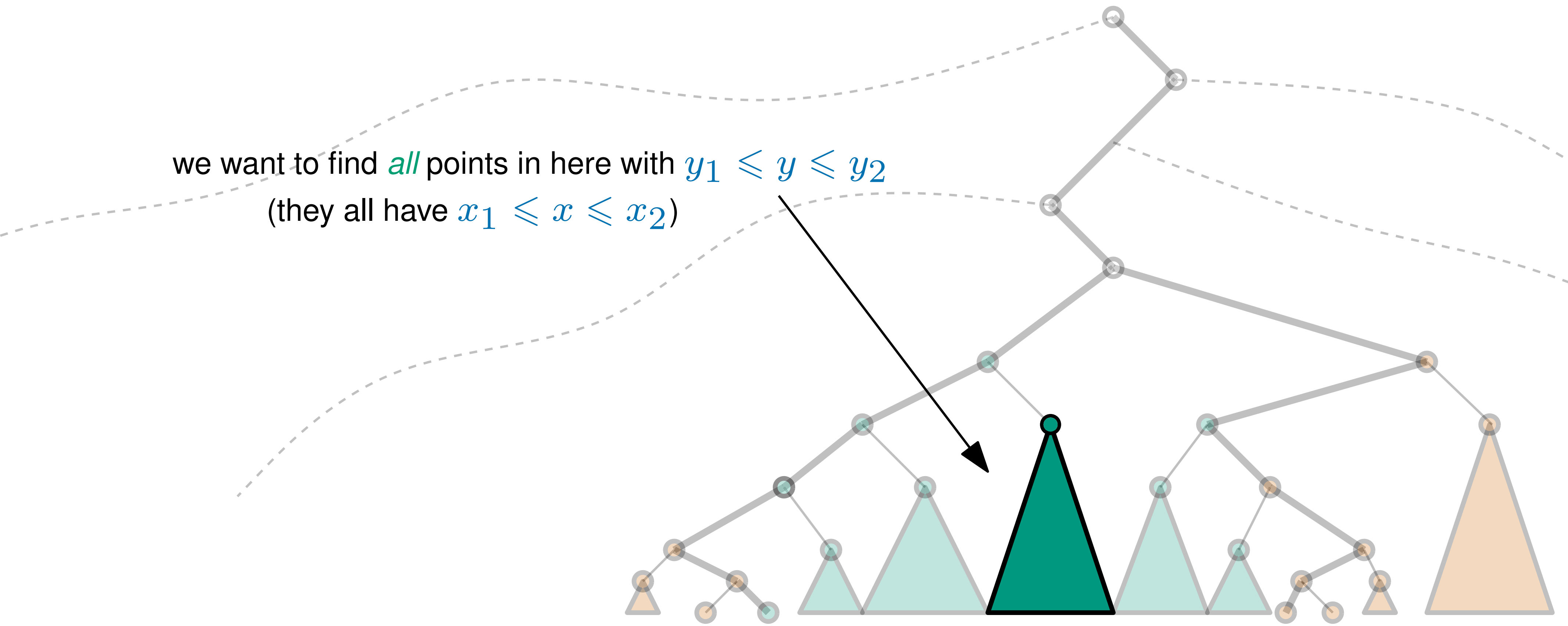
for any off-path subtree. . .

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D

we want to find *all* points in here with $y_1 \leqslant y \leqslant y_2$
(they all have $x_1 \leqslant x \leqslant x_2$)



(during preprocessing) build a balanced binary tree using the x-coordinates

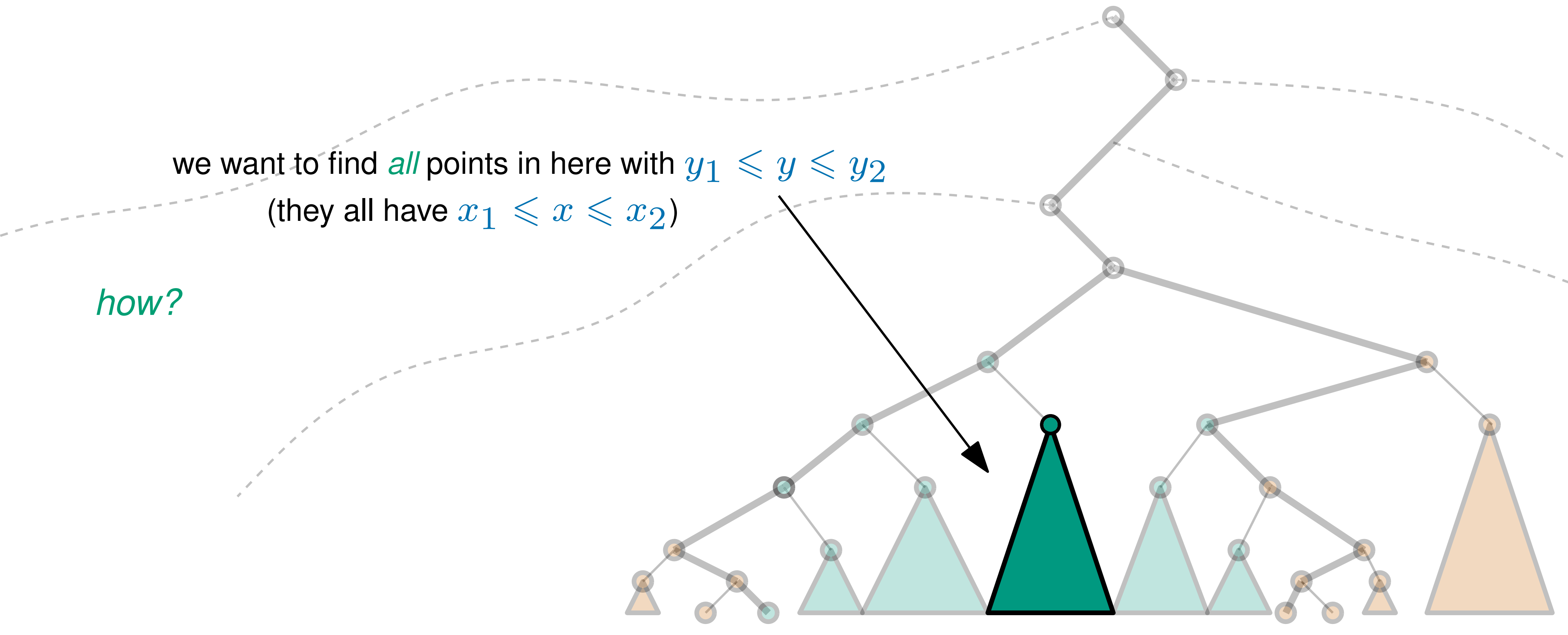
to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

for any off-path subtree...

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D



(during preprocessing) build a balanced binary tree using the x-coordinates

to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

for any off-path subtree...

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D

we want to find *all* points in here with $y_1 \leqslant y \leqslant y_2$

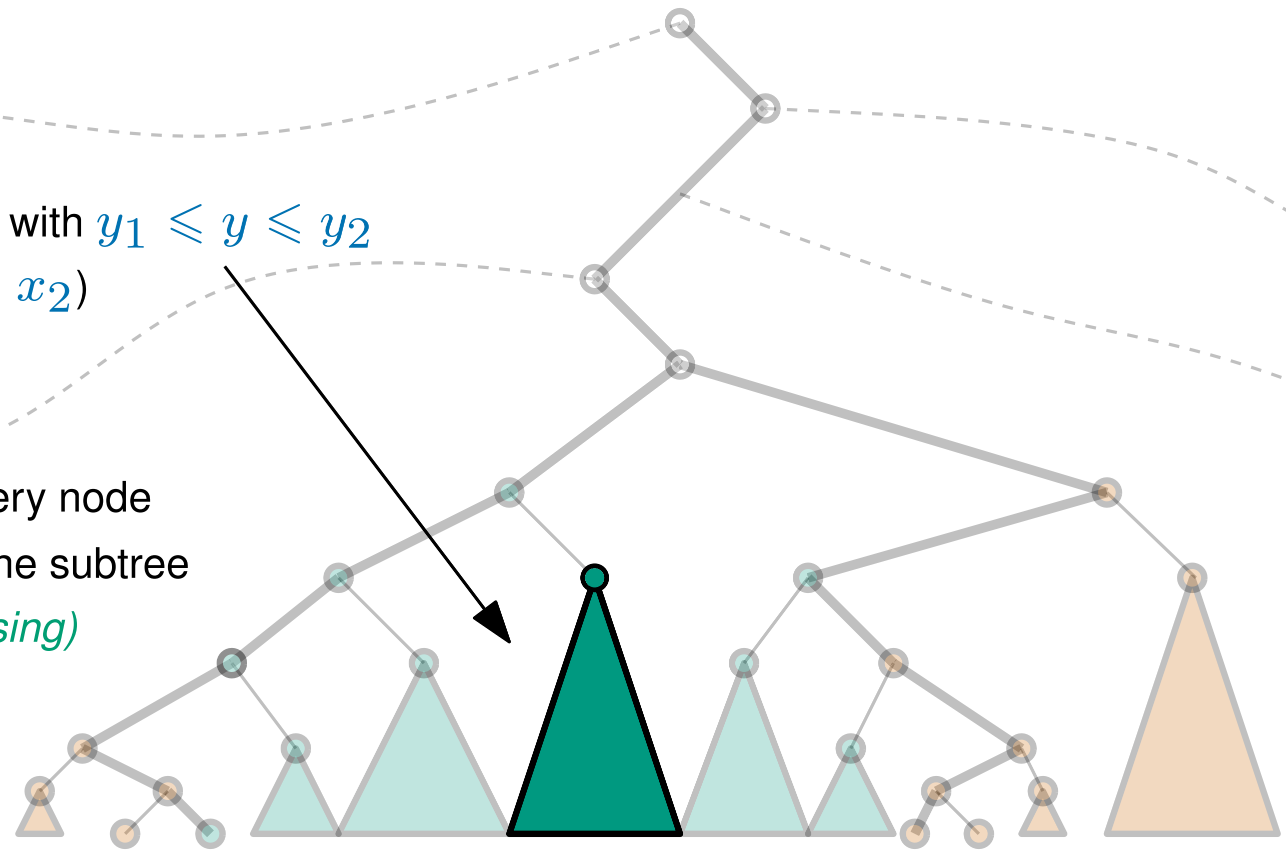
(they all have $x_1 \leqslant x \leqslant x_2$)

how?

build a *1D range searching structure* at every node

on the y-coordinates of the points in the subtree

(during preprocessing)



(during preprocessing) build a balanced binary tree using the x-coordinates

to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

for any off-path subtree. . .

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

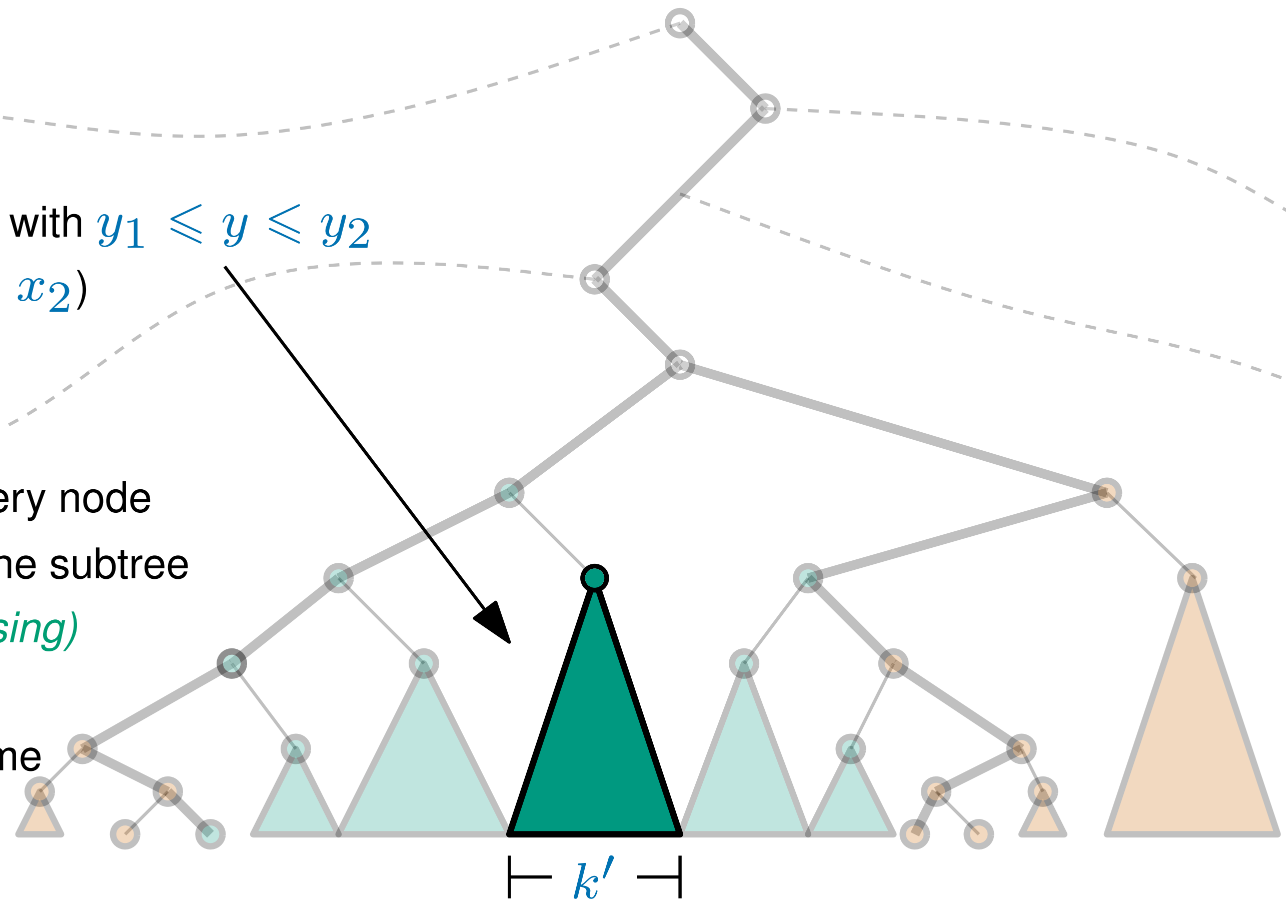
Subtree decomposition in 2D

we want to find *all* points in here with $y_1 \leqslant y \leqslant y_2$
(they all have $x_1 \leqslant x \leqslant x_2$)

how?

build a *1D range searching structure* at every node
on the y-coordinates of the points in the subtree
(during preprocessing)

a 1D lookup takes $O(\log n + k')$ time



(during preprocessing) build a balanced binary tree using the x-coordinates

to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

for any off-path subtree...
every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D

we want to find *all* points in here with $y_1 \leqslant y \leqslant y_2$

(they all have $x_1 \leqslant x \leqslant x_2$)

how?

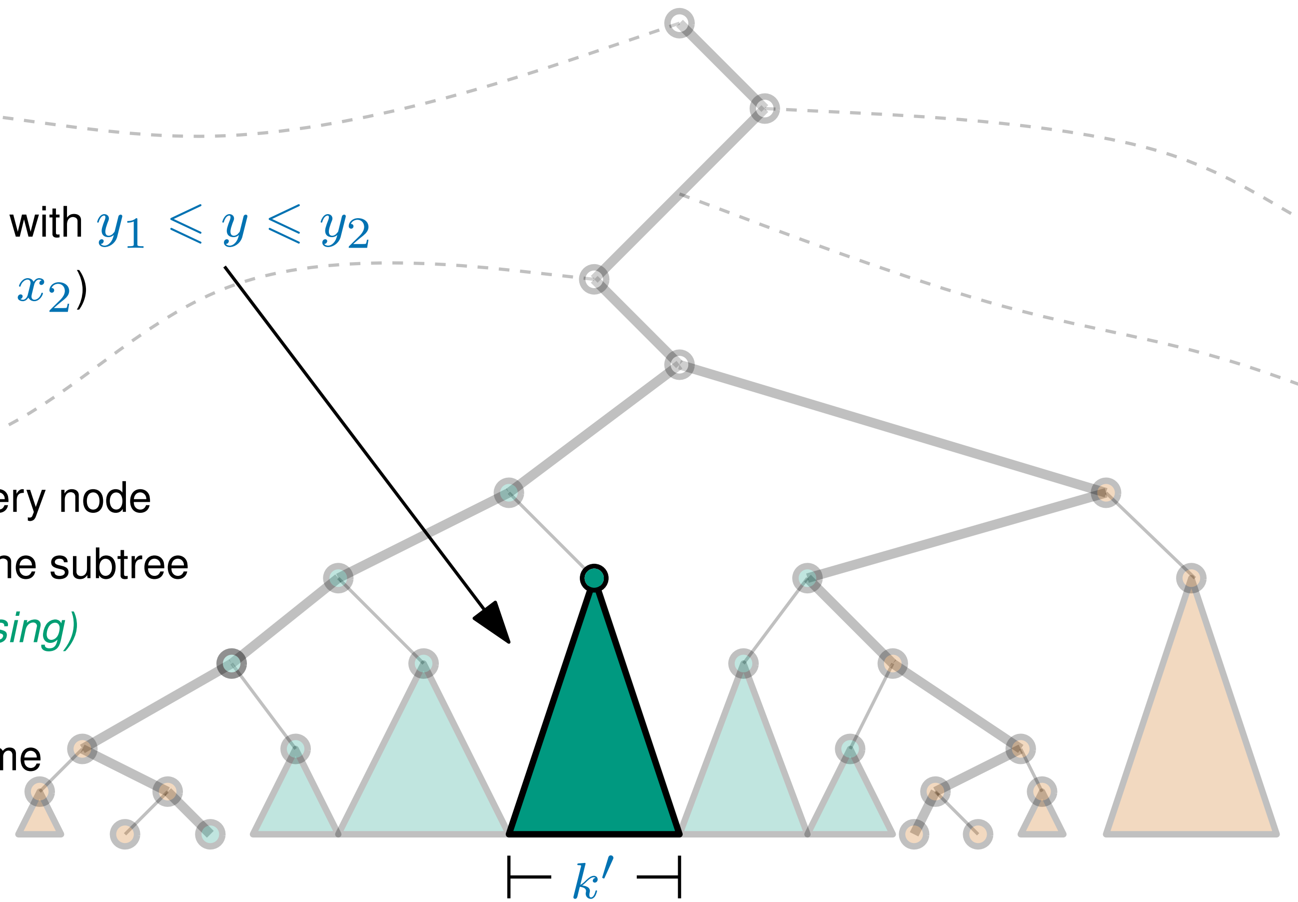
build a *1D range searching structure* at every node

on the y-coordinates of the points in the subtree

(during preprocessing)

a 1D lookup takes $O(\log n + k')$ time

and only returns points we want



(during preprocessing) build a balanced binary tree using the x-coordinates

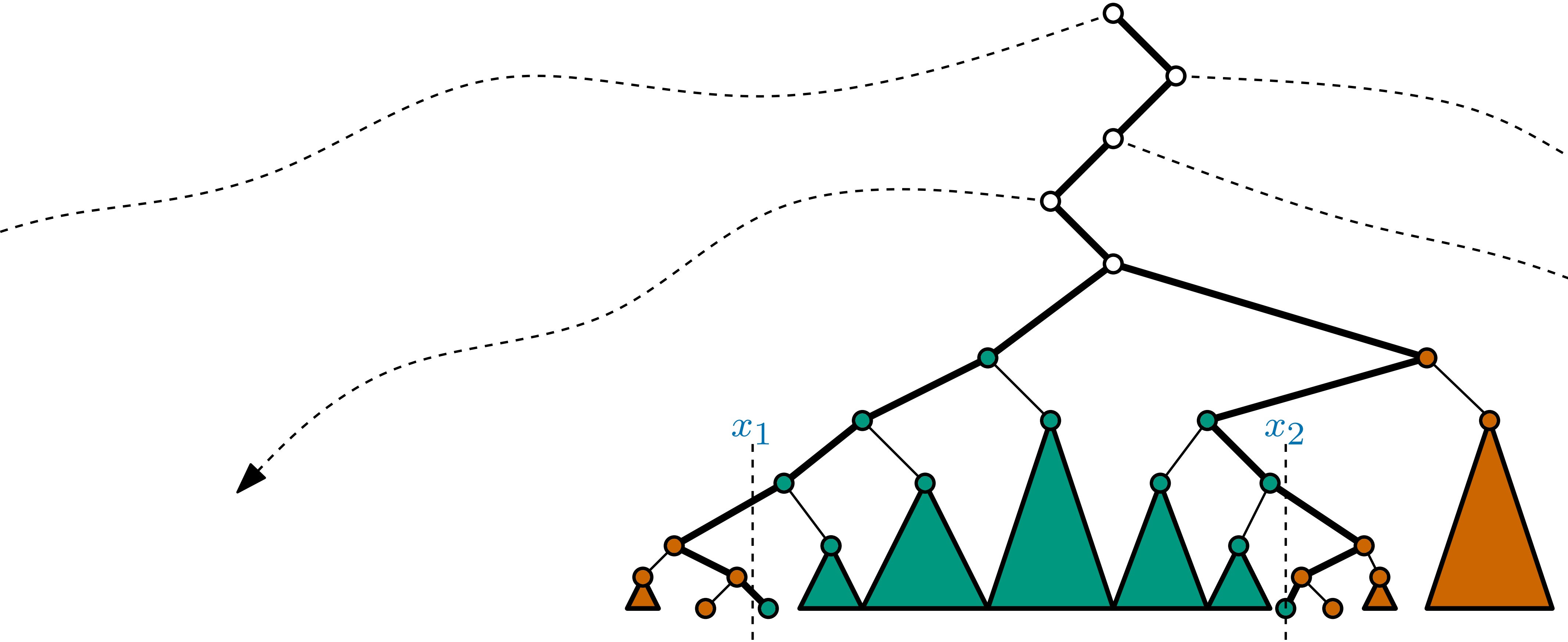
to perform a lookup(x_1, x_2, y_1, y_2) *follow the paths to* x_1 *and* x_2 *as before*

for any off-path subtree...

every point in the subtree has $x_1 \leqslant x \leqslant x_2$ or no point has

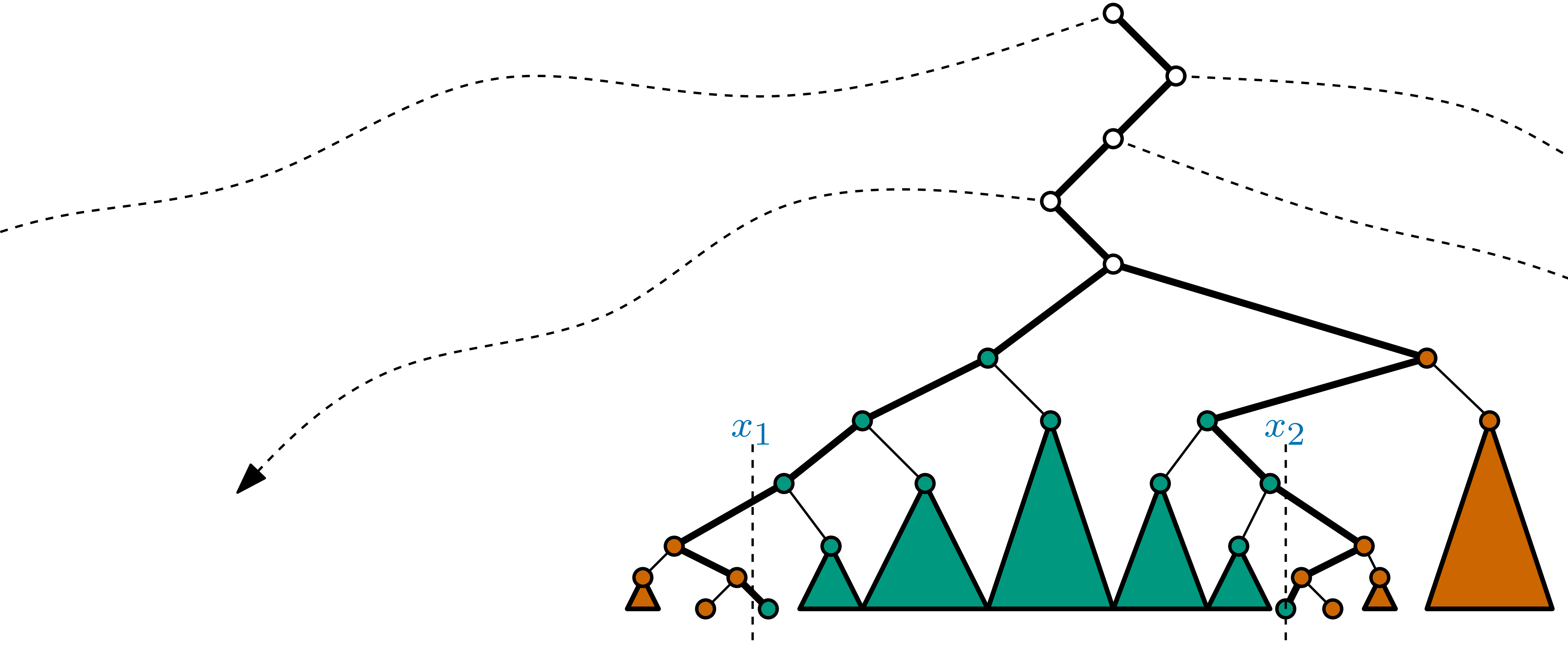
Idea: filter these subtrees by y-coordinate

Subtree decomposition in 2D



Query summary

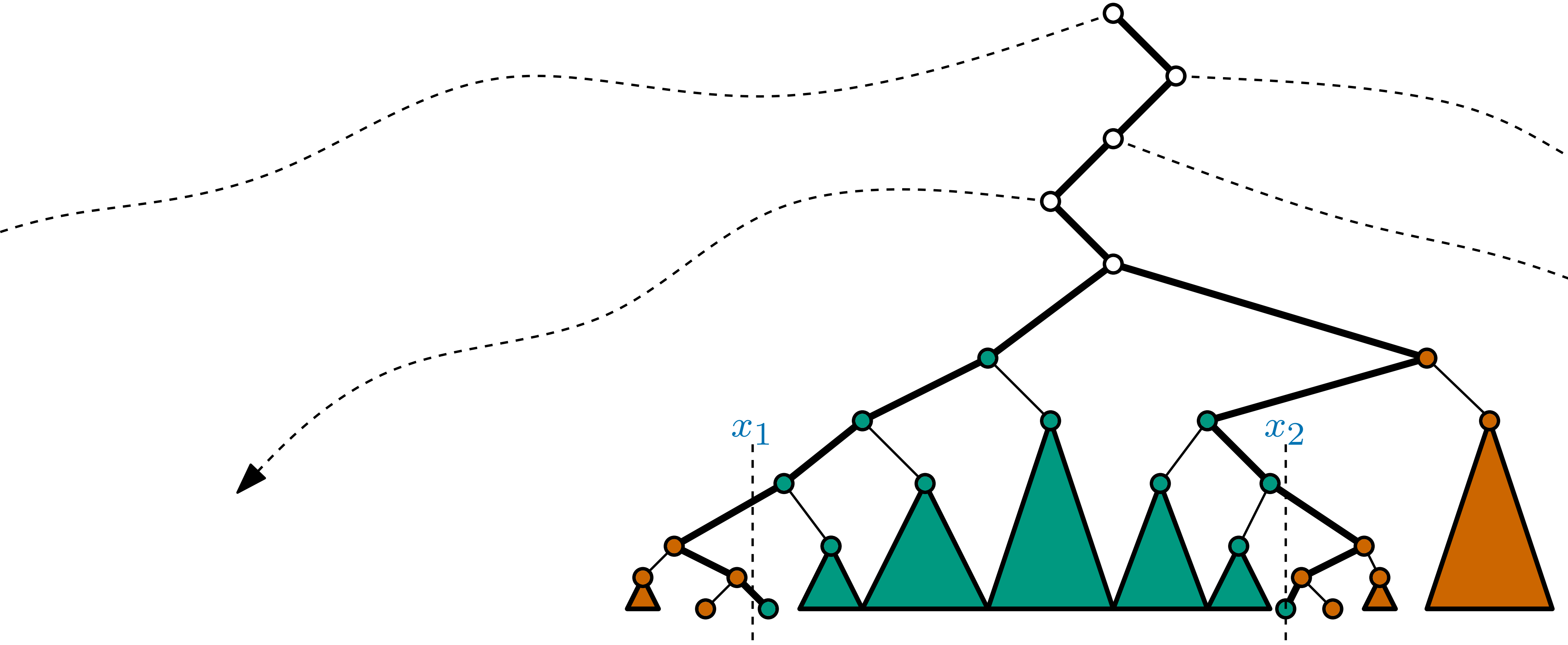
Subtree decomposition in 2D



Query summary

1. Follow the paths to x_1 and x_2

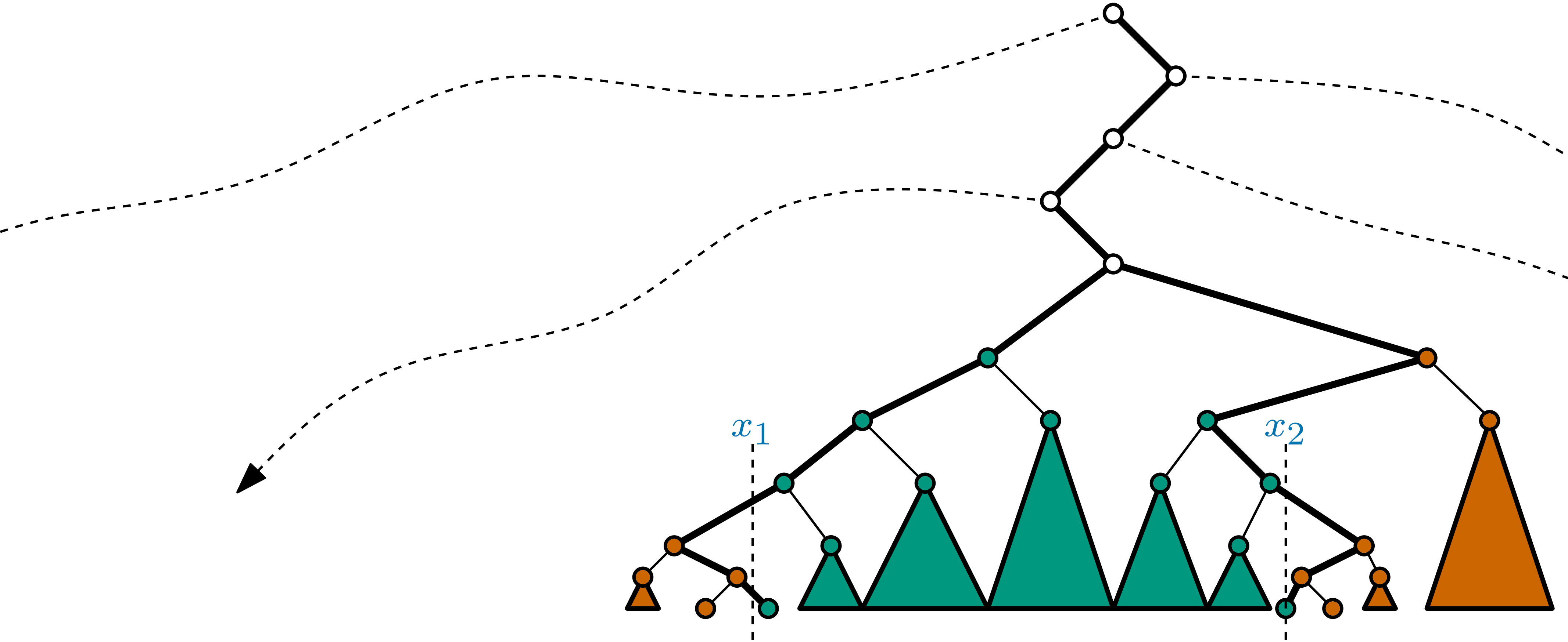
Subtree decomposition in 2D



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)

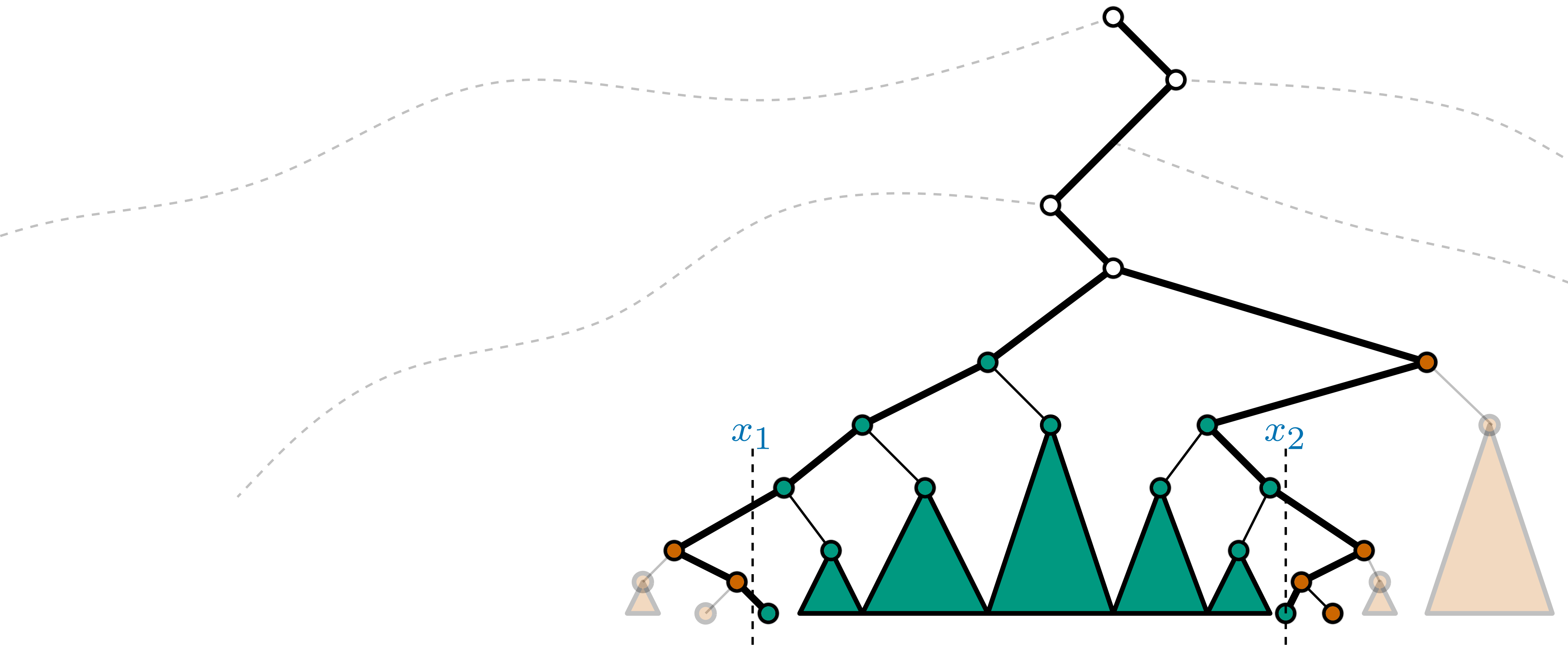
Subtree decomposition in 2D



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*

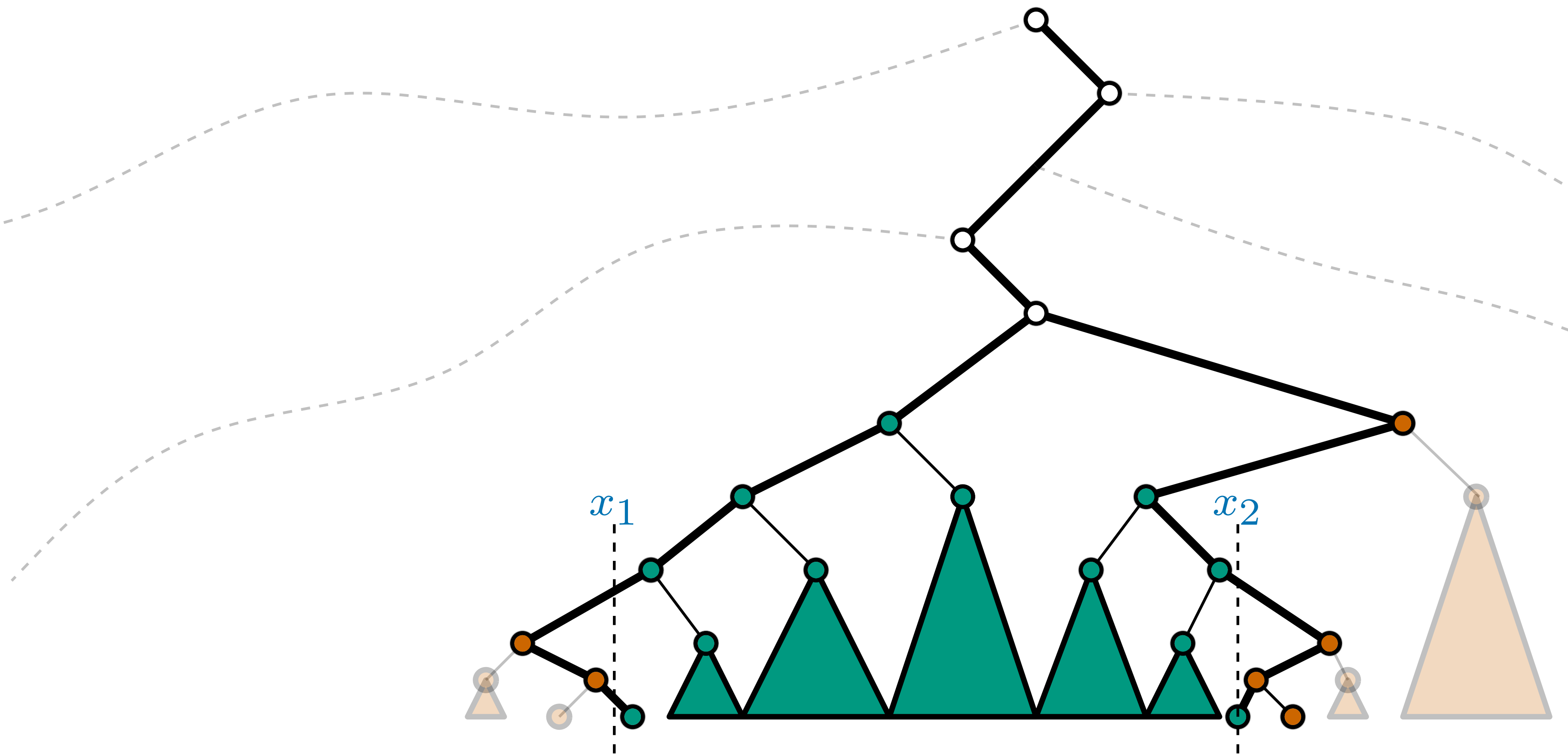
Subtree decomposition in 2D



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*

Subtree decomposition in 2D



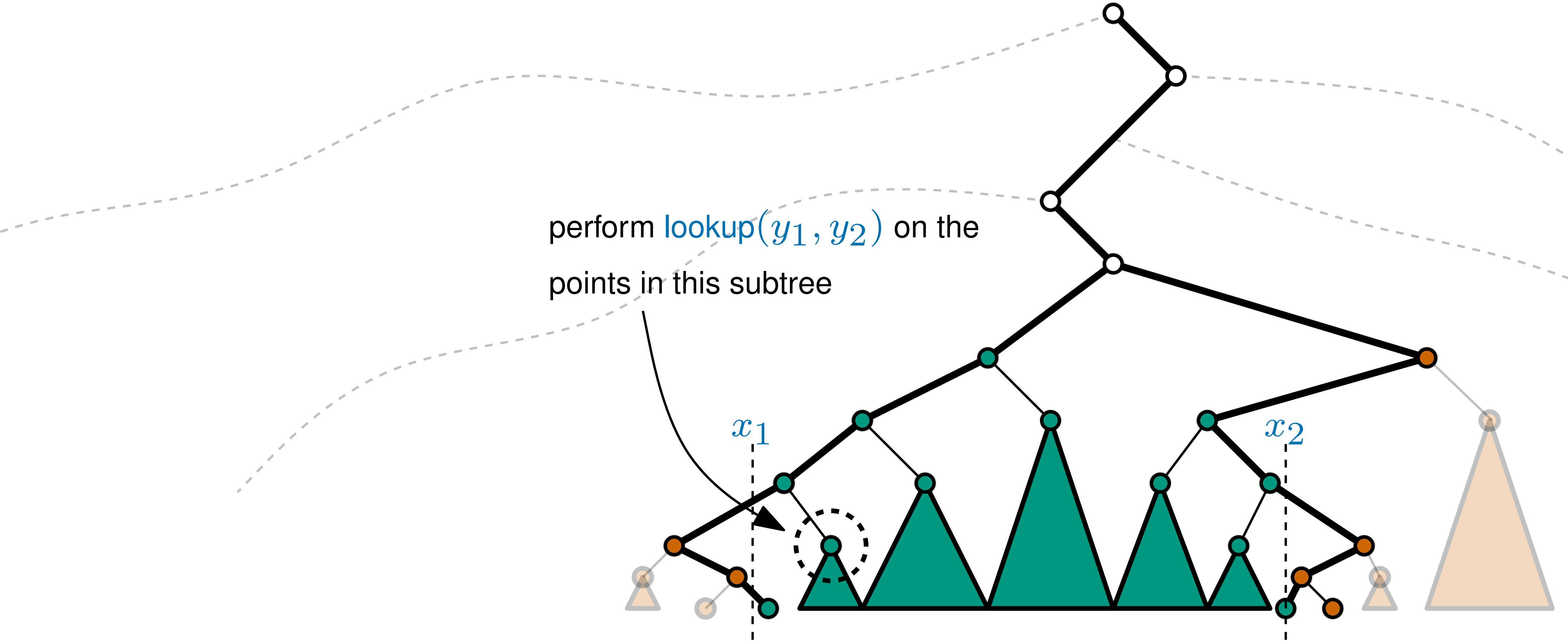
Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

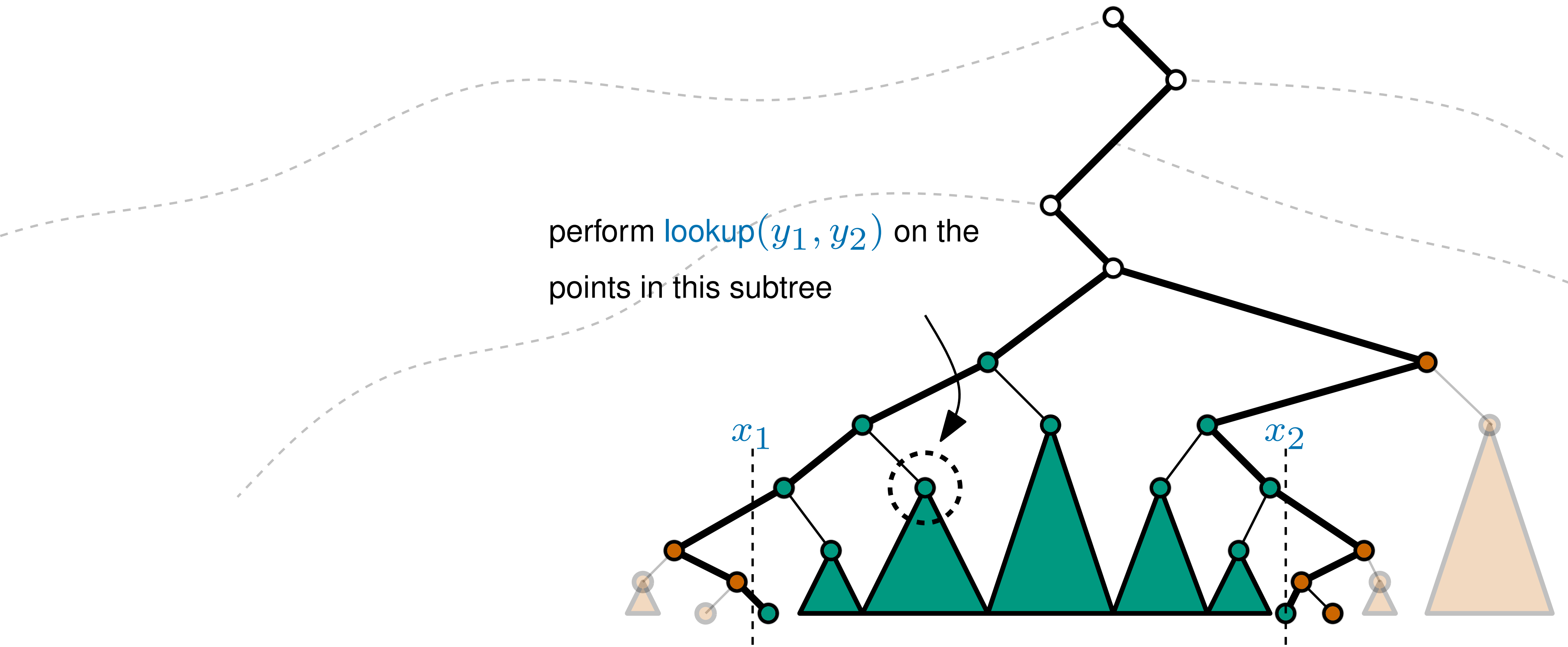


Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree to filter the y coordinates

Subtree decomposition in 2D

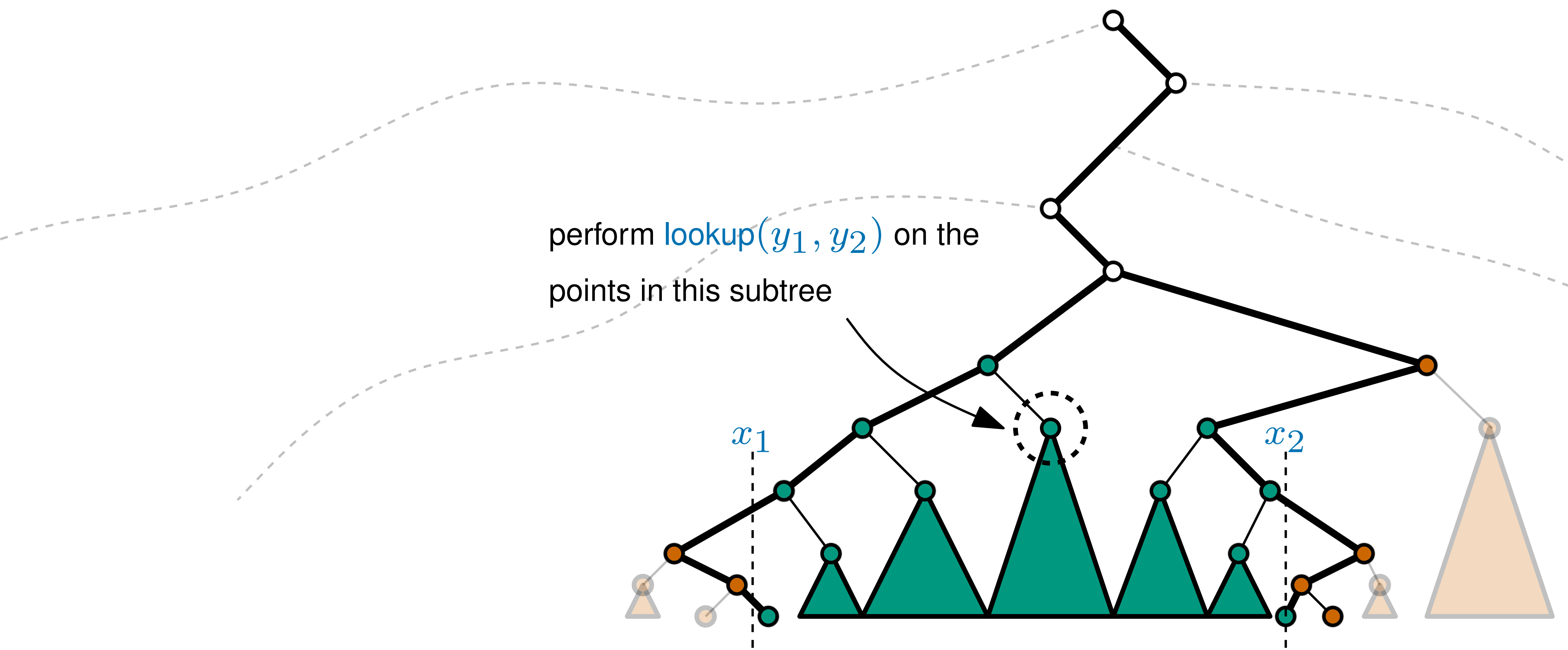


Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range. . .

use the 1D range structure for that subtree to filter the y coordinates

Subtree decomposition in 2D



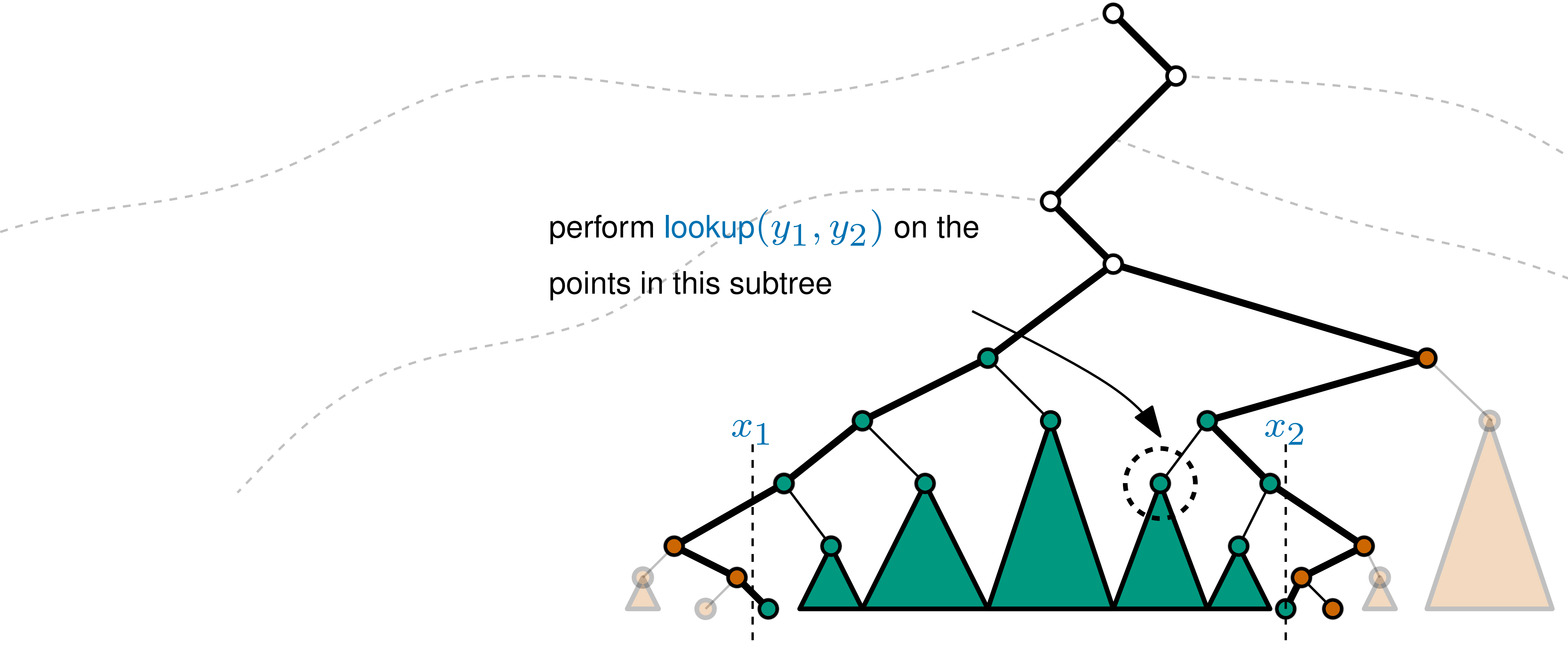
Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

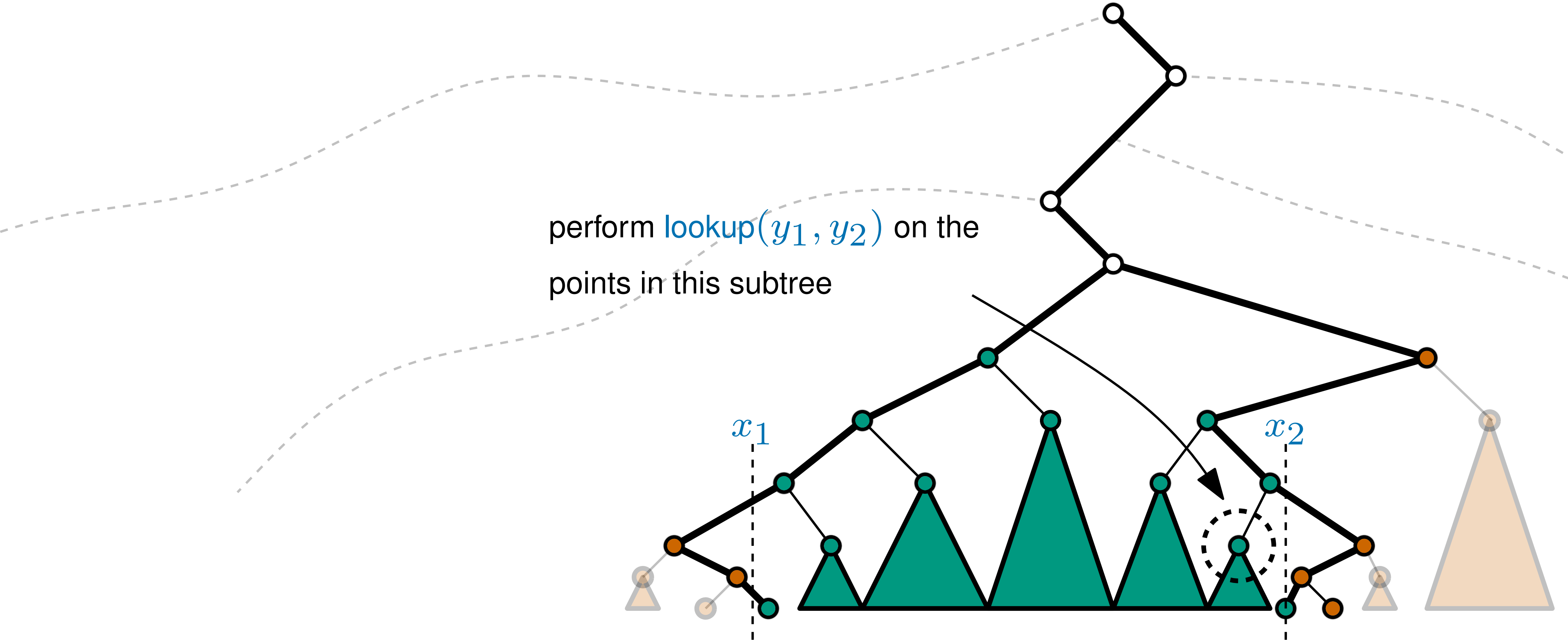


Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

   use the 1D range structure for that subtree to filter the y coordinates

Subtree decomposition in 2D



Query summary

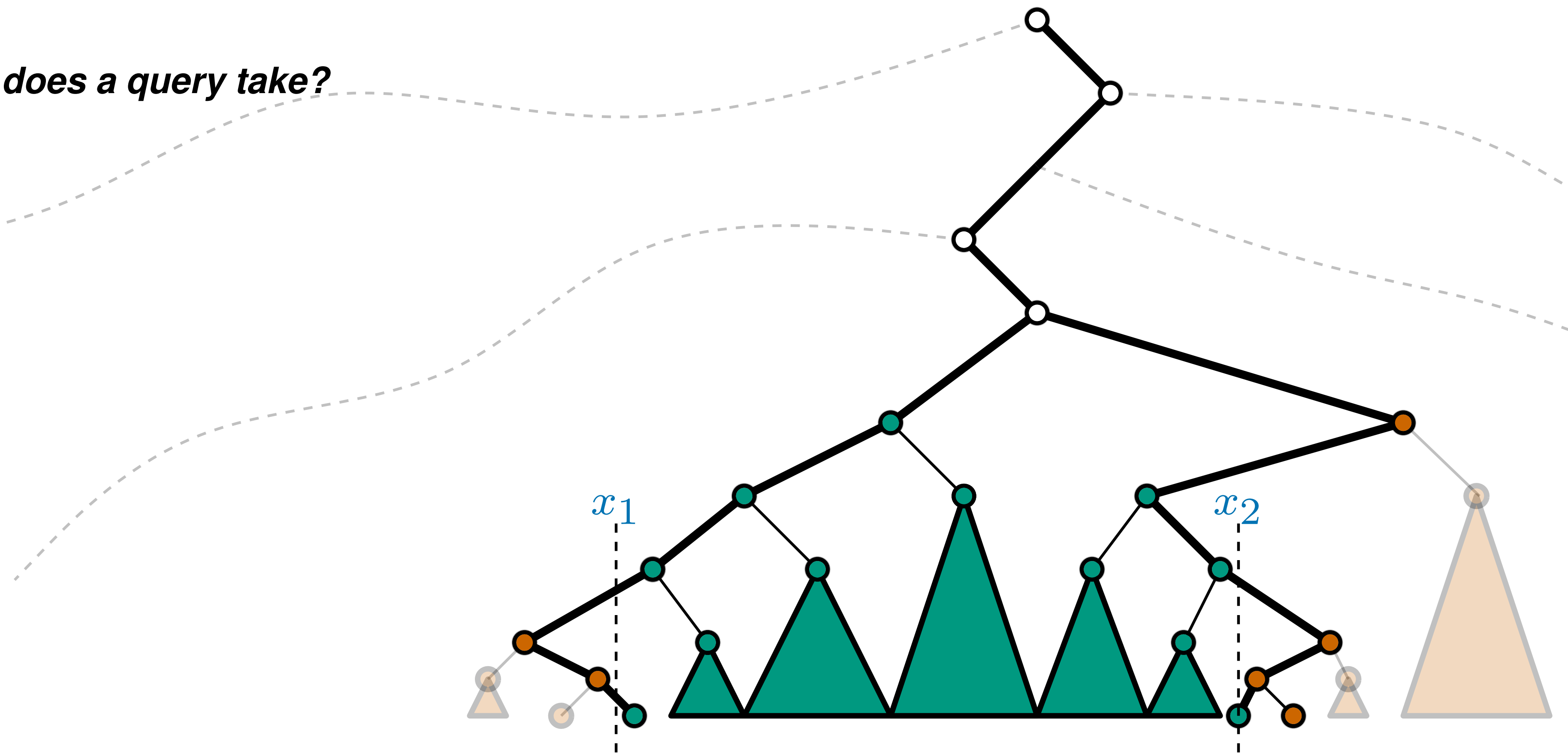
1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

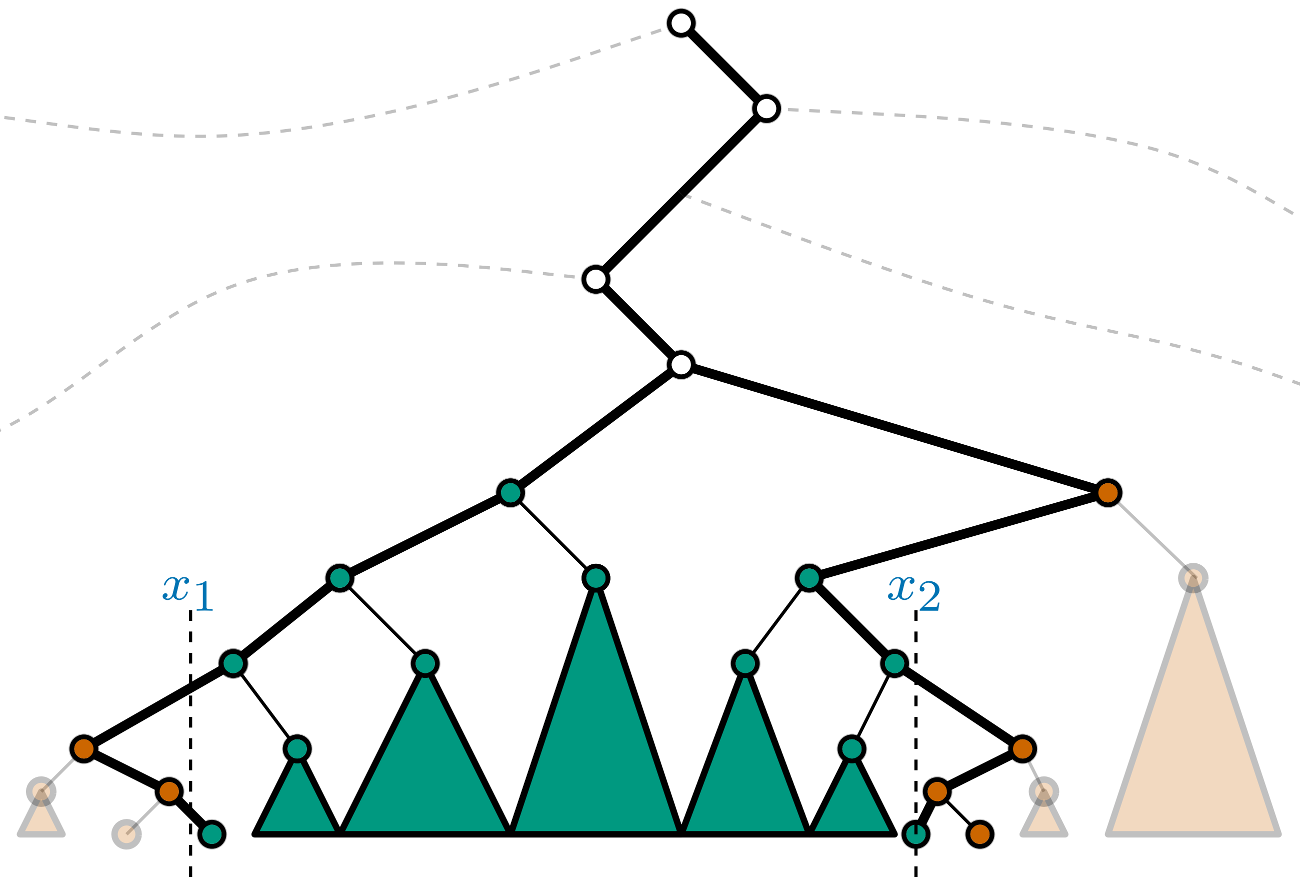
   use the 1D range structure for that subtree

   to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?

The paths have length $O(\log n)$



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

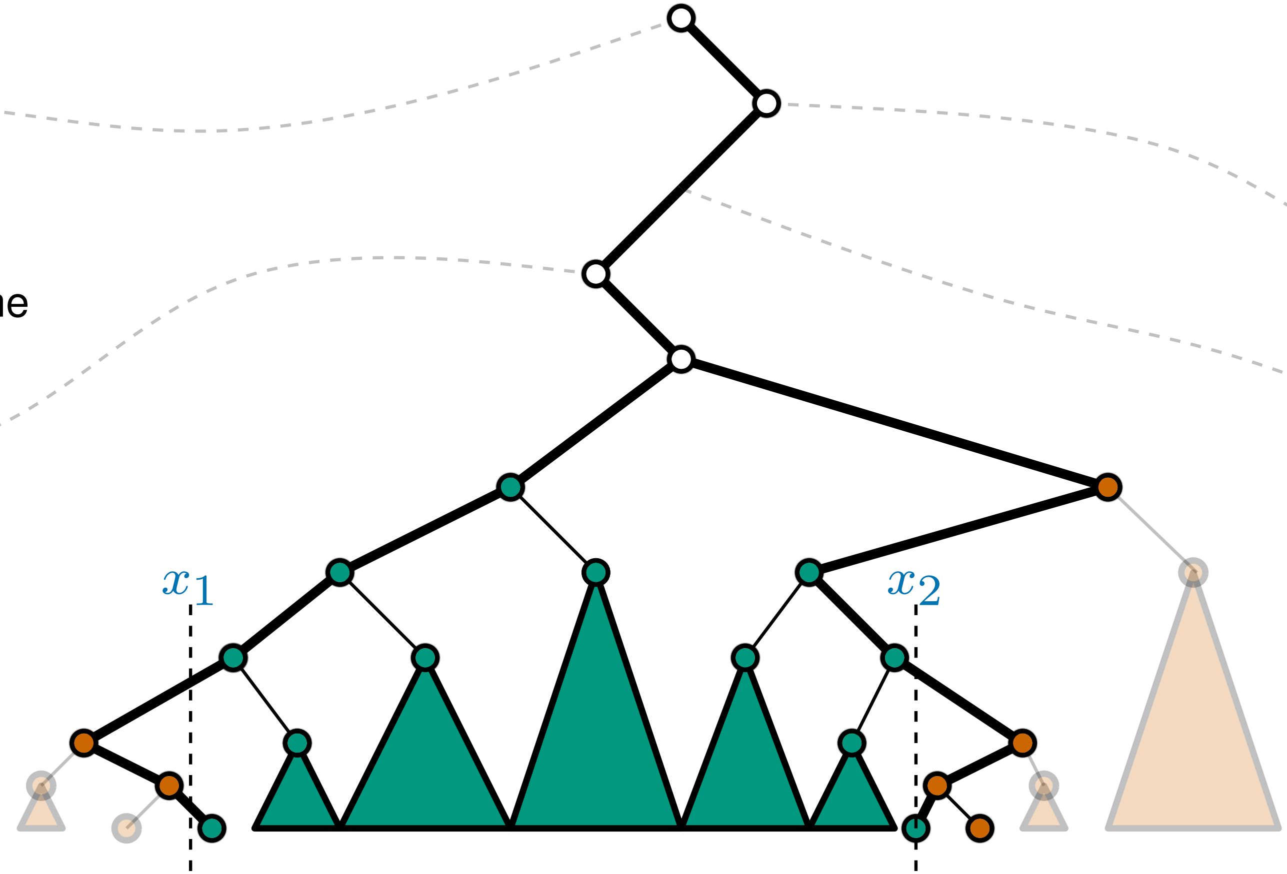
to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

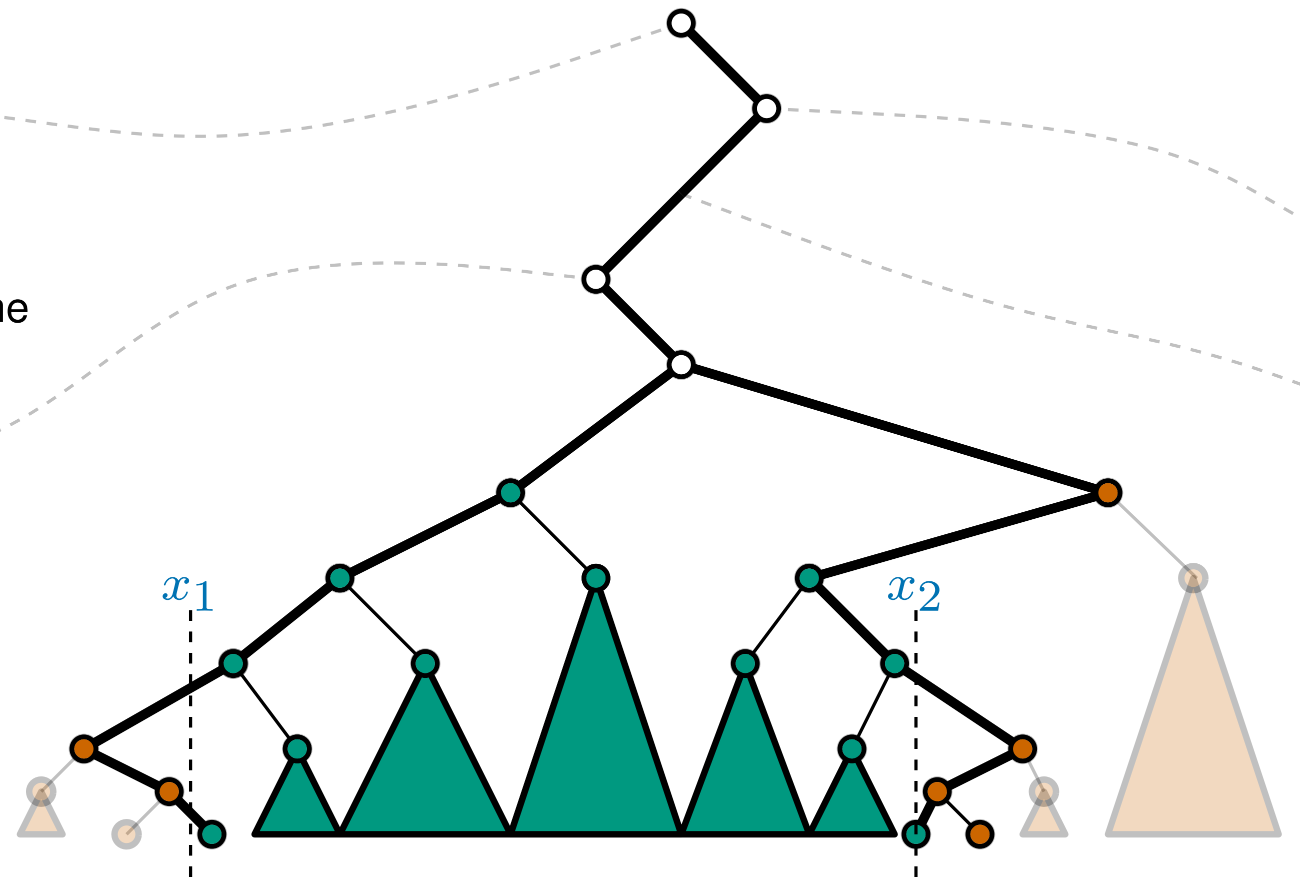
Subtree decomposition in 2D

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

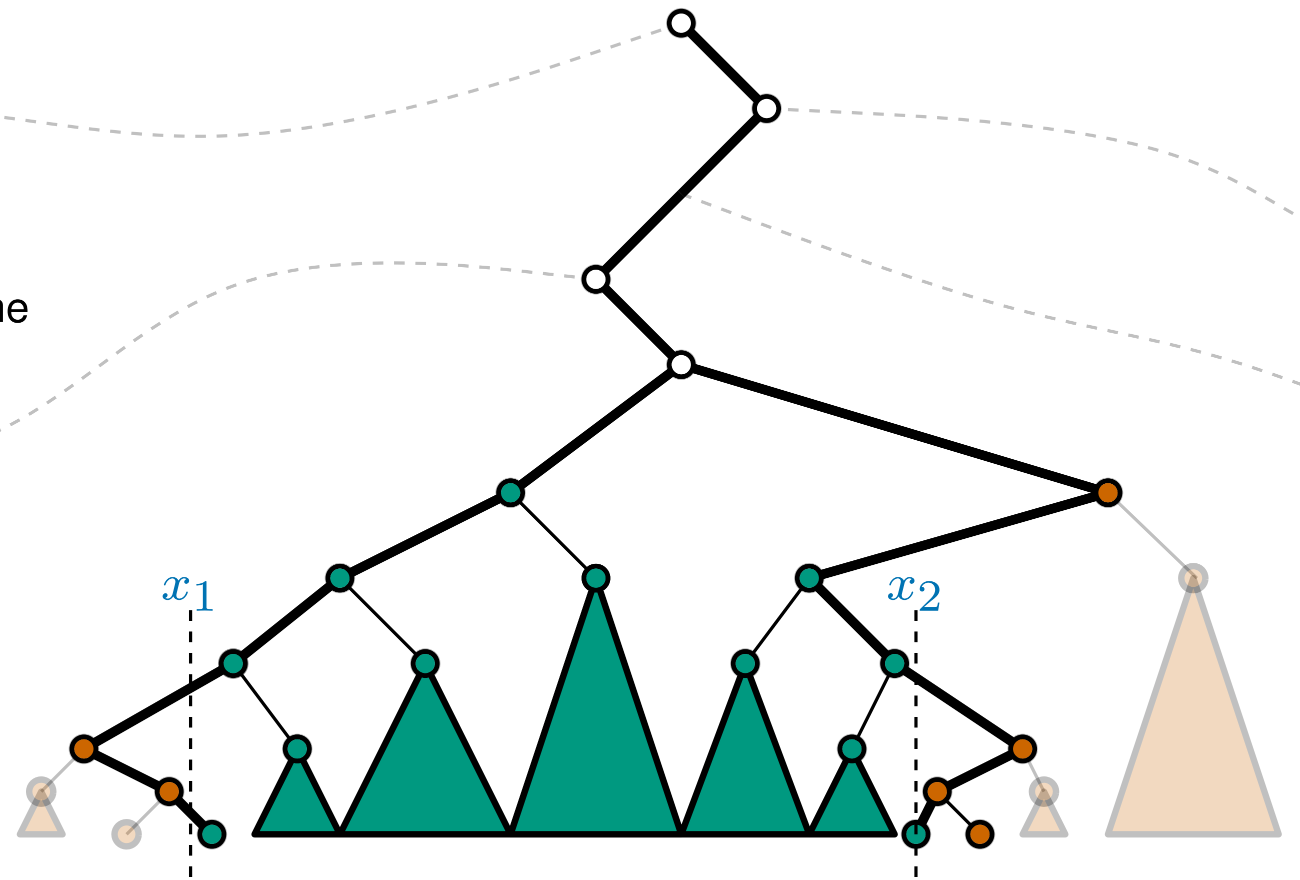
How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

We do $O(\log n)$ 1D lookups. . .



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range. . .

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?

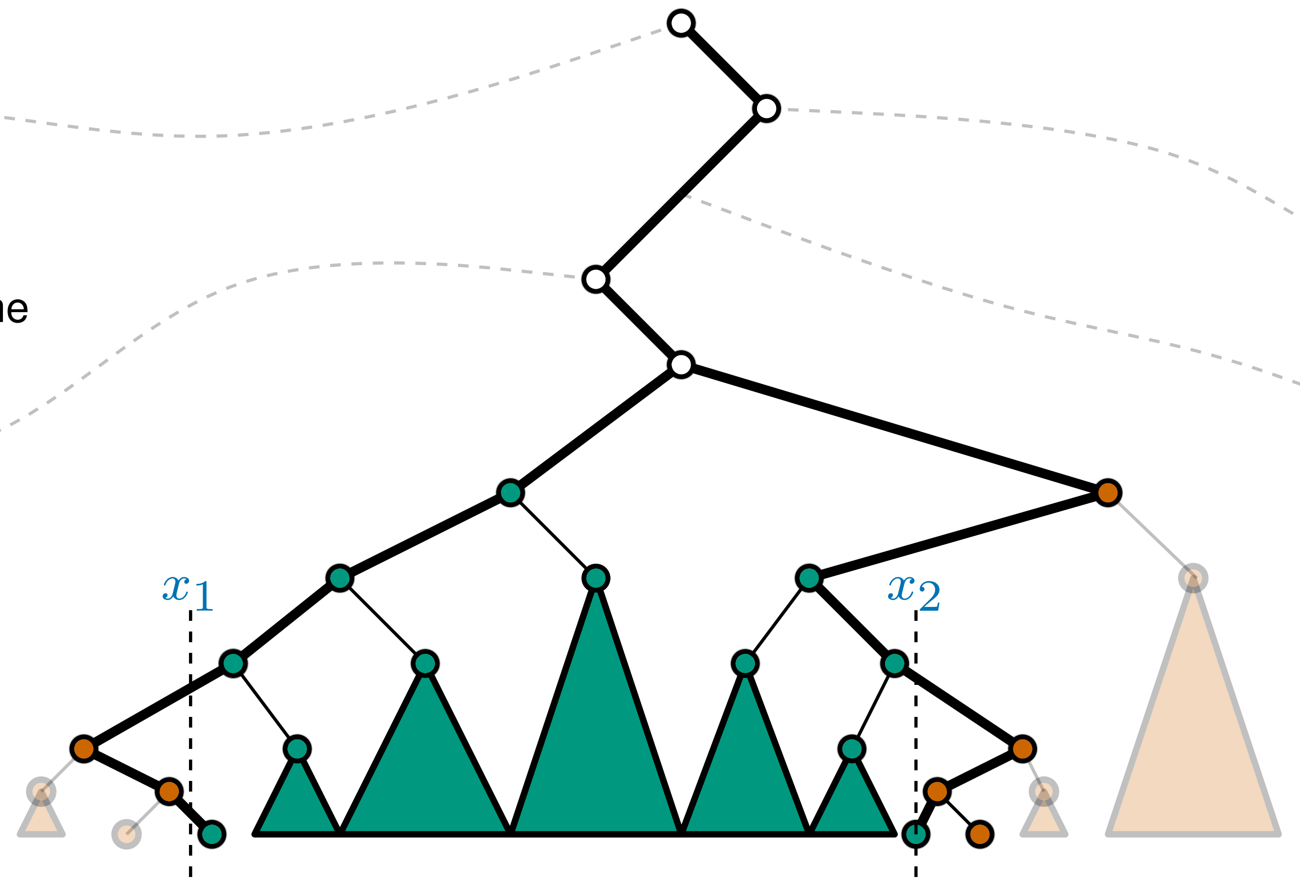
The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

We do $O(\log n)$ 1D lookups...

Each takes $O(\log n + k')$ time



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

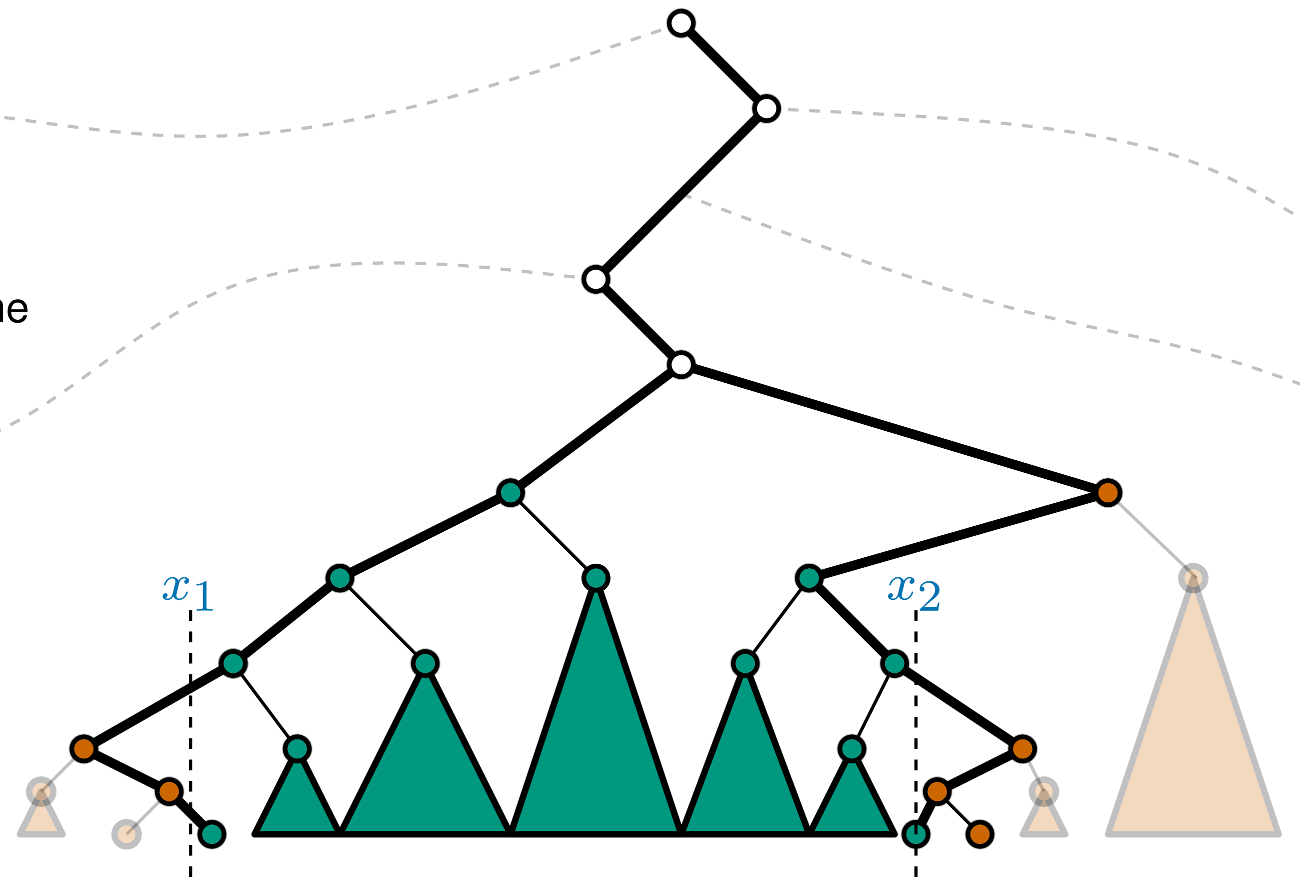
As for step 3,

We do $O(\log n)$ 1D lookups...

Each takes $O(\log n + k')$ time

This sums to...

$O(\log^2 n + k)$



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

Subtree decomposition in 2D

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

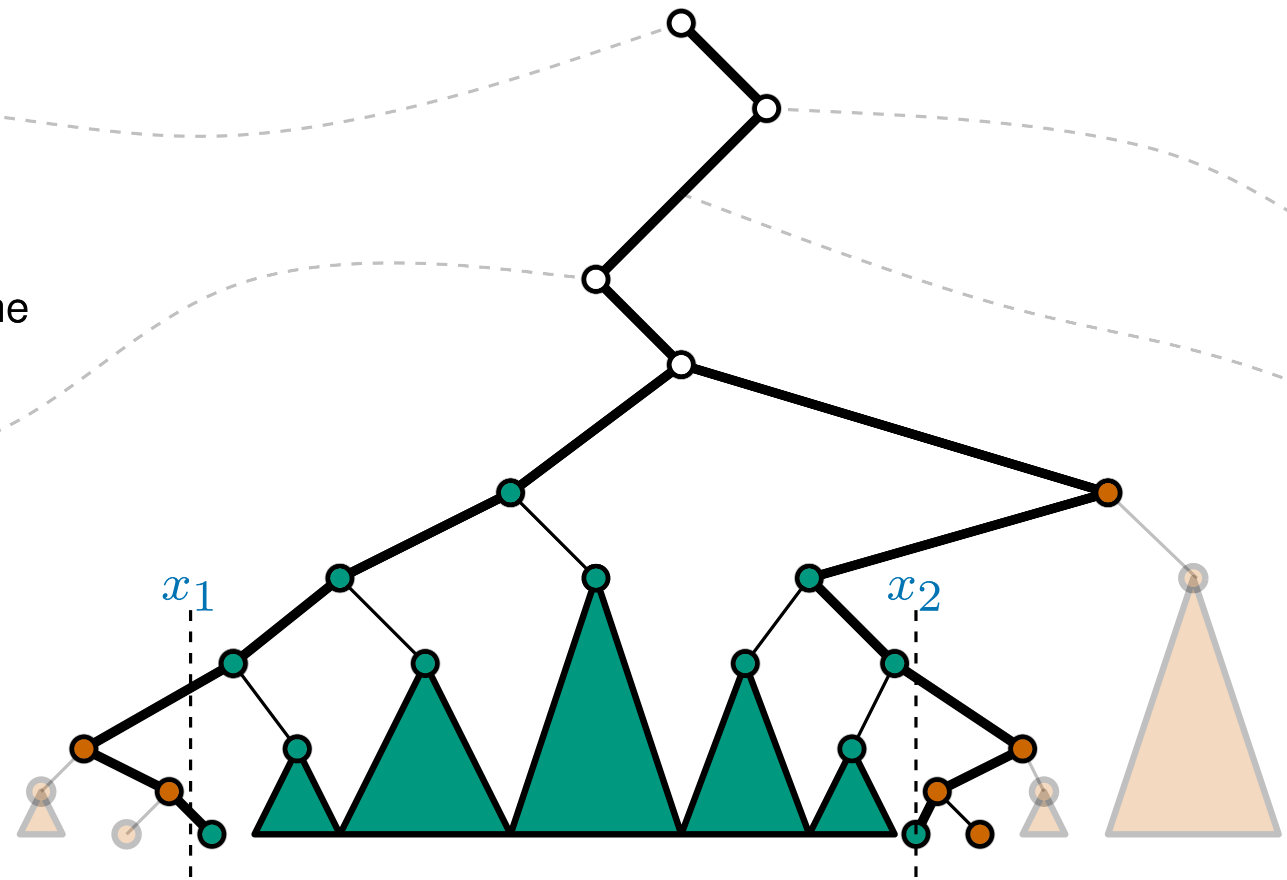
We do $O(\log n)$ 1D lookups...

Each takes $O(\log n + k')$ time

This sums to...

$$O(\log^2 n + k)$$

because the 1D lookups are disjoint



Query summary

1. Follow the paths to x_1 and x_2 (inspecting the points on the path as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree
to filter the y coordinates

Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space. . .*

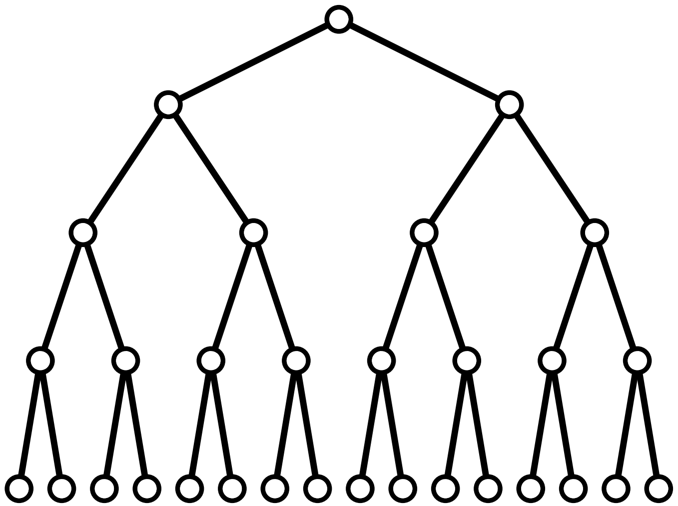
but we added some stuff

at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)

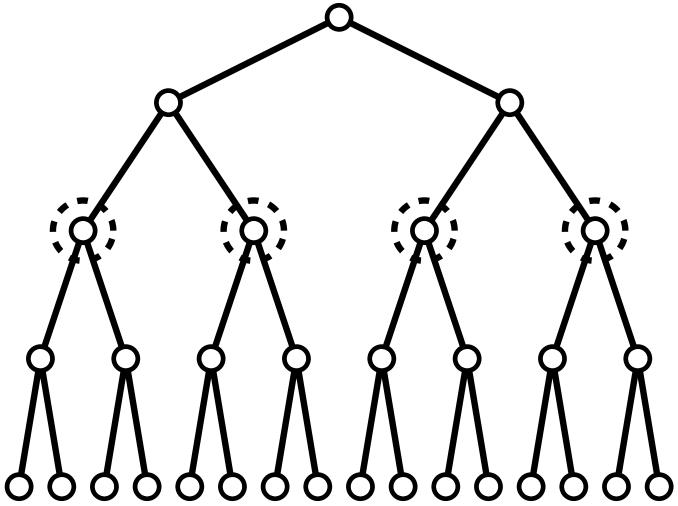


Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space...*

but we added some stuff



at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)

look at any level in the tree

i.e. all nodes at the same distance from the root

Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space...*

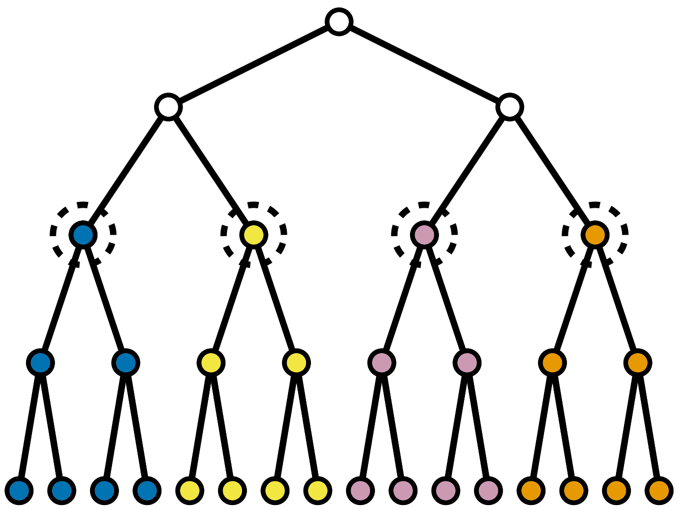
but we added some stuff

at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)



look at any level in the tree

i.e. all nodes at the same distance from the root

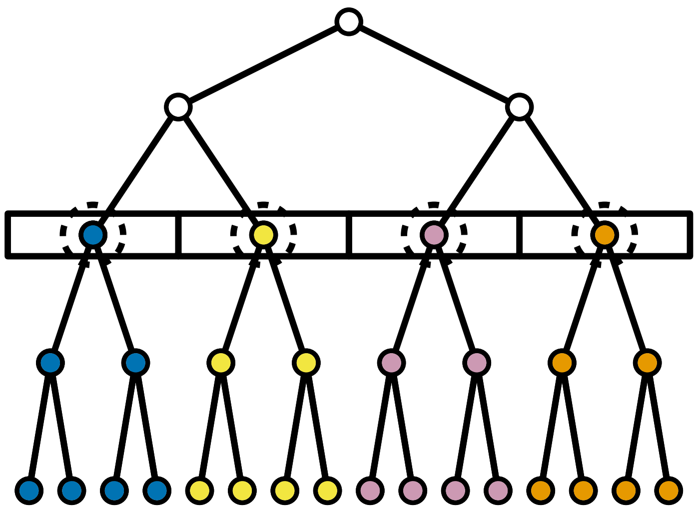
the points in these subtrees are disjoint

Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space...*

but we added some stuff



at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)

look at any level in the tree

i.e. all nodes at the same distance from the root

the points in these subtrees are disjoint

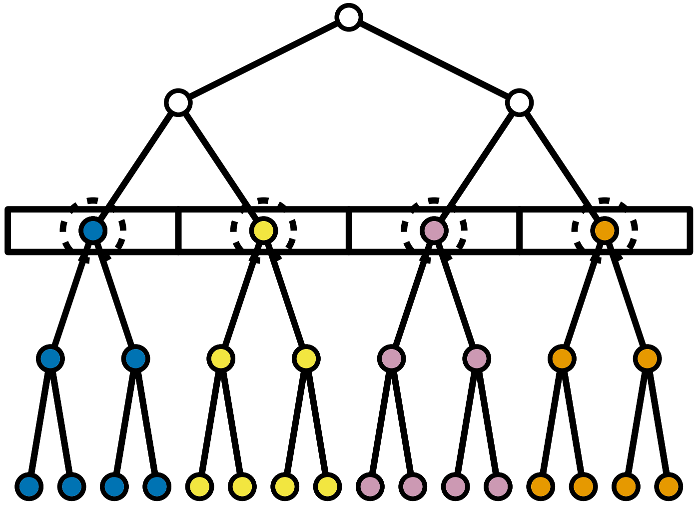
so the sizes of the arrays add up to n

Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space...*

but we added some stuff



at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)

look at any level in the tree

i.e. all nodes at the same distance from the root

the points in these subtrees are disjoint

so the sizes of the arrays add up to n

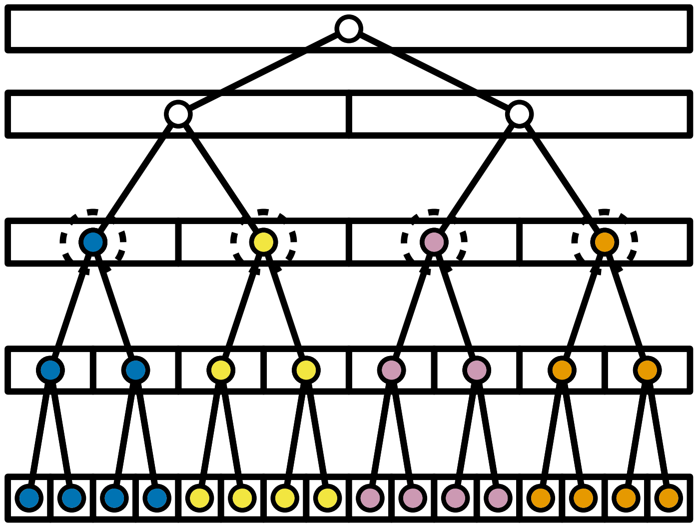
As the tree has depth $O(\log n)$...

Space Usage

How much space does our 2D range structure use?

the original (1D) structure used $O(n)$ *space...*

but we added some stuff



at each node we store an array

containing the points in its subtree

the array is sorted by y coordinate

(this gives us a 1D range data structure)

look at any level in the tree

i.e. all nodes at the same distance from the root

the points in these subtrees are disjoint

so the sizes of the arrays add up to n

As the tree has depth $O(\log n)$...

the total space used is $O(n \log n)$

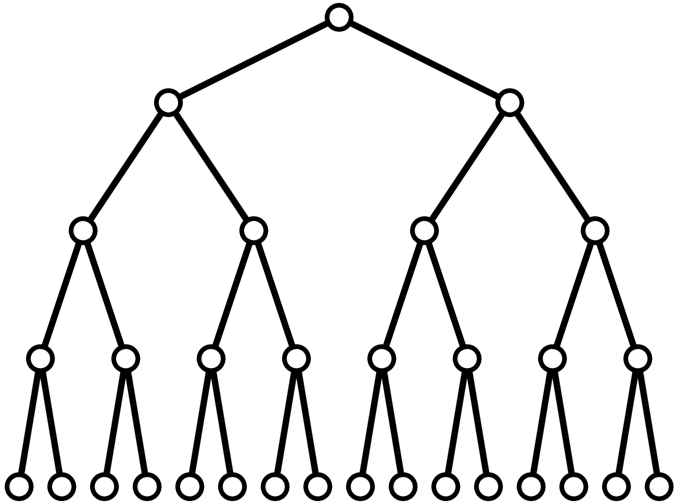
Preprocessing time

How much prep time does our 2D range structure take?

the original (1D) structure used $O(n \log n)$ *prep time. . .*

but we added some stuff

How long does it take to build the arrays at the nodes?

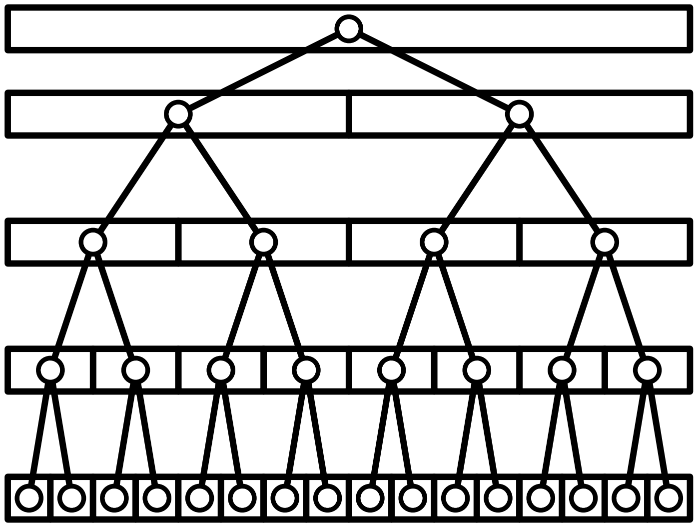


Preprocessing time

How much prep time does our 2D range structure take?

the original (1D) structure used $O(n \log n)$ *prep time...*

but we added some stuff



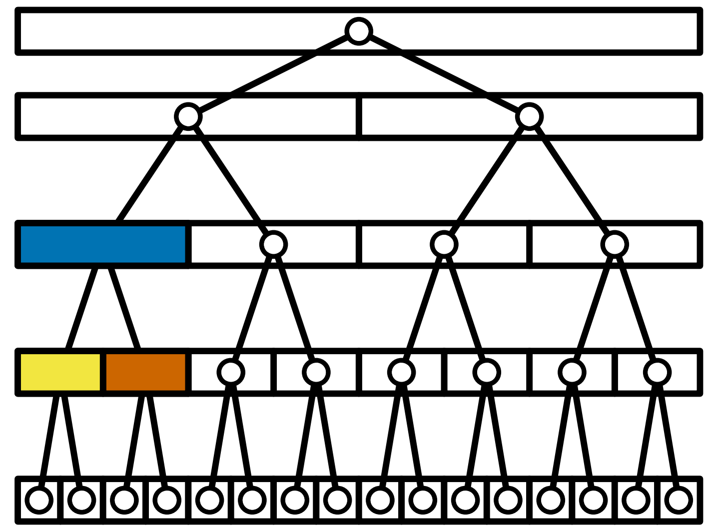
How long does it take to build the arrays at the nodes?

Preprocessing time

How much prep time does our 2D range structure take?

the original (1D) structure used $O(n \log n)$ *prep time. . .*

but we added some stuff



How long does it take to build the arrays at the nodes?

is just  merged with 

as  and  are already sorted,

merging them takes $O(\ell)$ time

Therefore the total time is $O(n \log n)$

(which is the sum of the lengths of the arrays)

University of BRISTOL

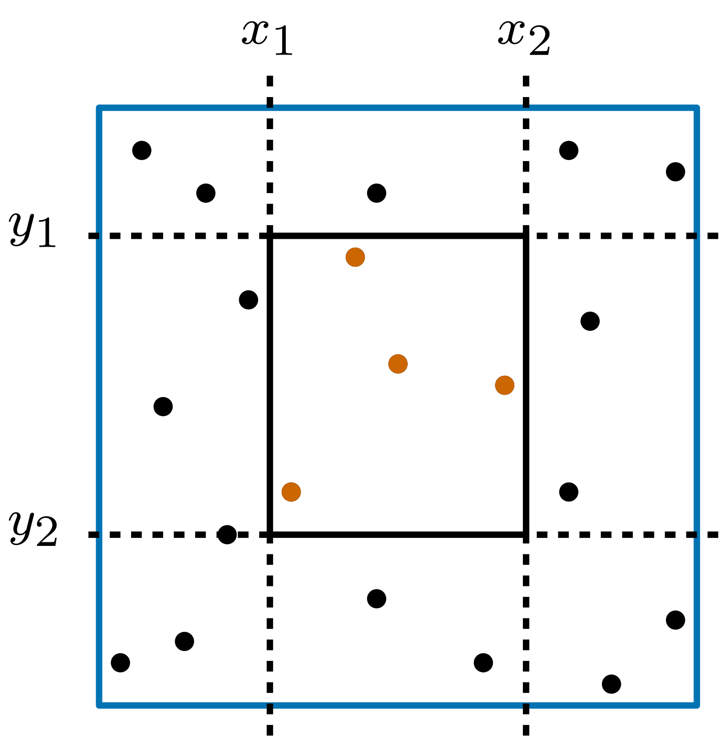
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Summary

$O(n \log n)$ prep time

$O(n \log n)$ space

$O(\log^2 n + k)$ lookup time

where k is the number of points reported

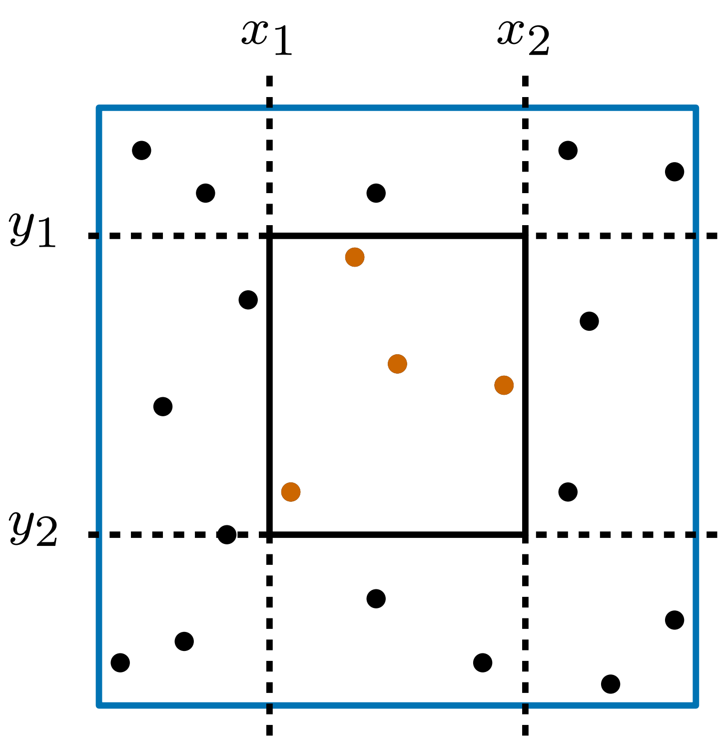
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Summary

$O(n \log n)$ prep time

$O(n \log n)$ space

$O(\log^2 n + k)$ lookup time

where k is the number of points reported

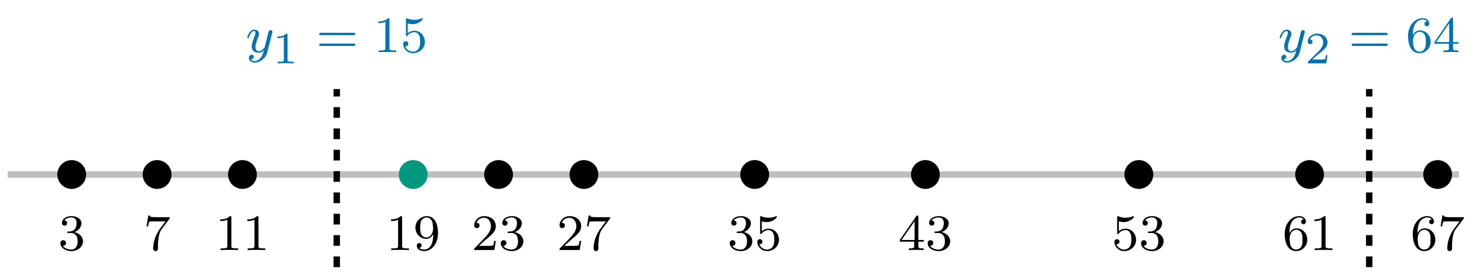
actually we can improve this :)

Improving the query time

when we do a 2D look-up we do $O(\log n)$ 1D lookups...

all with the same y_1 and y_2

(but on different point sets)



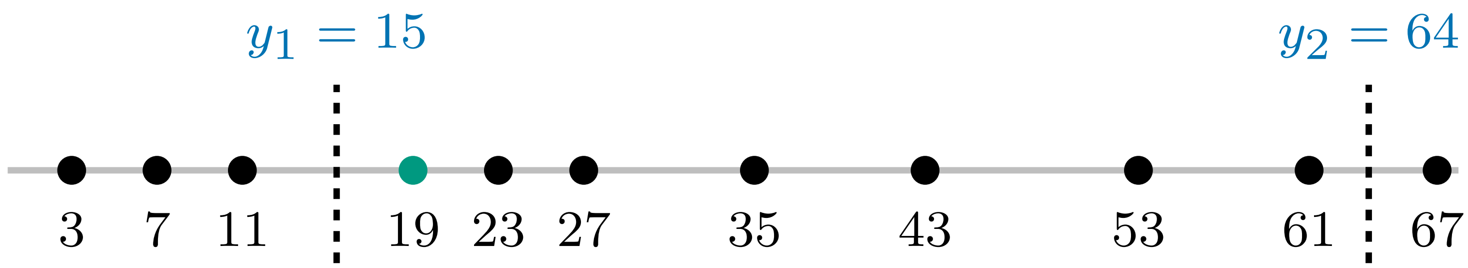
3	7	11	19	23	27	35	43	53	61	67

Improving the query time

when we do a 2D look-up we do $O(\log n)$ 1D lookups...

all with the same y_1 and y_2

(but on different point sets)



3	7	11	19	23	27	35	43	53	61	67

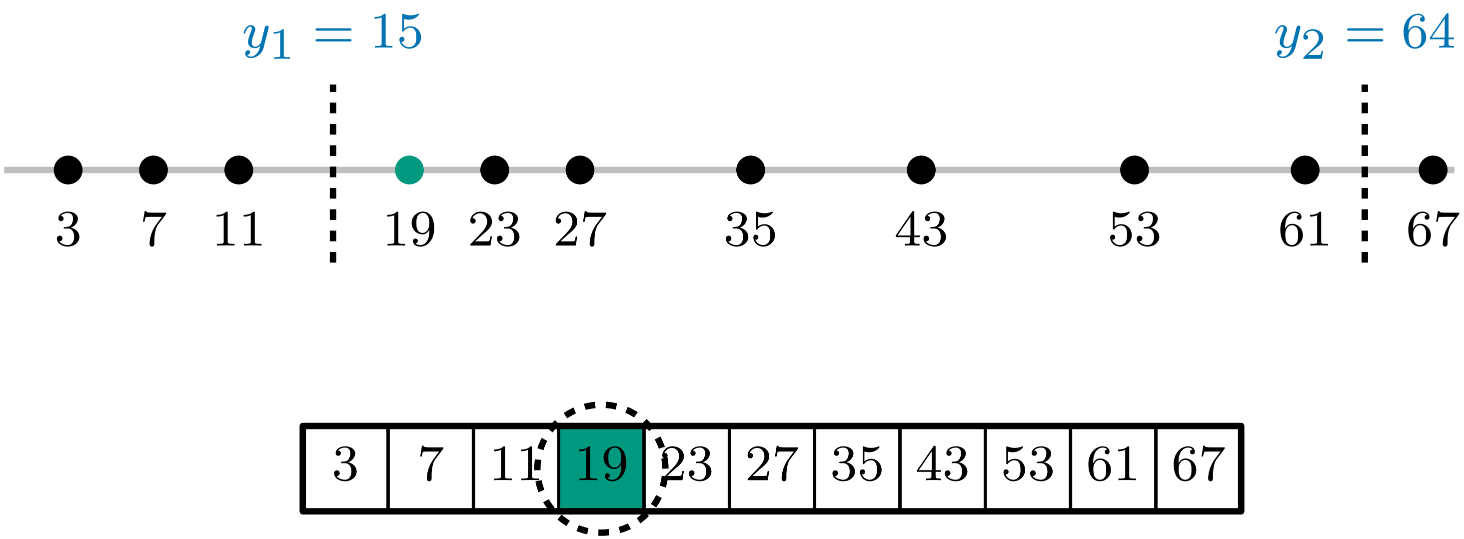
The *slow* part is finding the successor of y_1

Improving the query time

when we do a 2D look-up we do $O(\log n)$ 1D lookups. . .

all with the same y_1 and y_2

(but on different point sets)

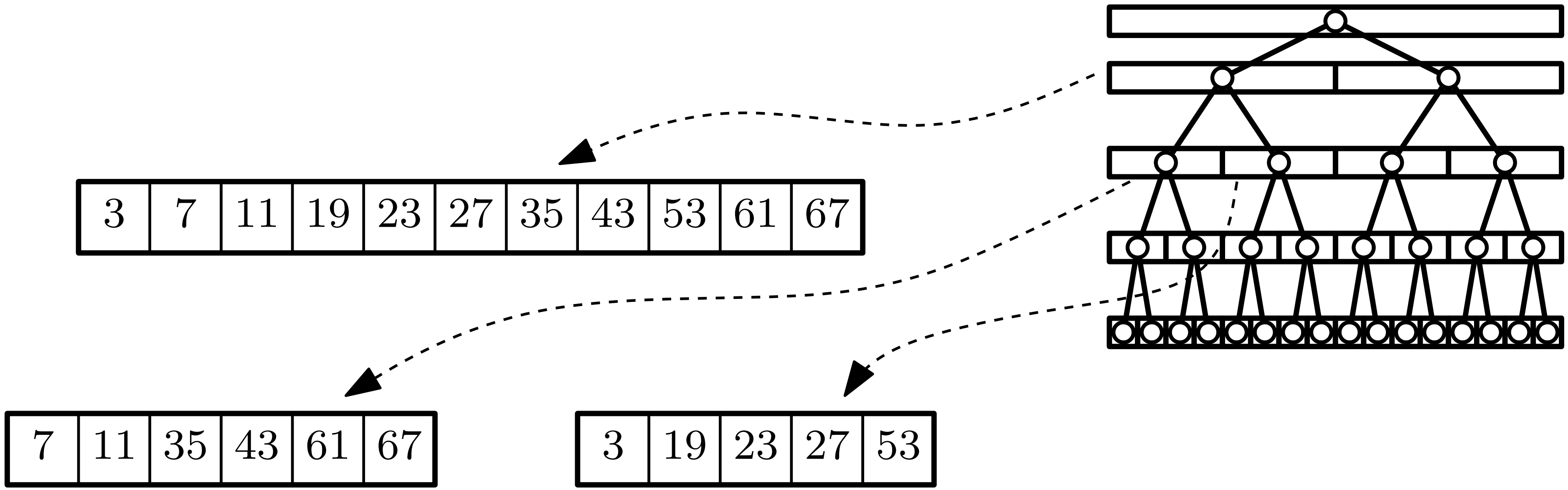


The *slow* part is finding the successor of y_1

If I told you where this point was, a 1D lookup would only take $O(k')$ time

(where k' is the number of points between y_1 and y_2)

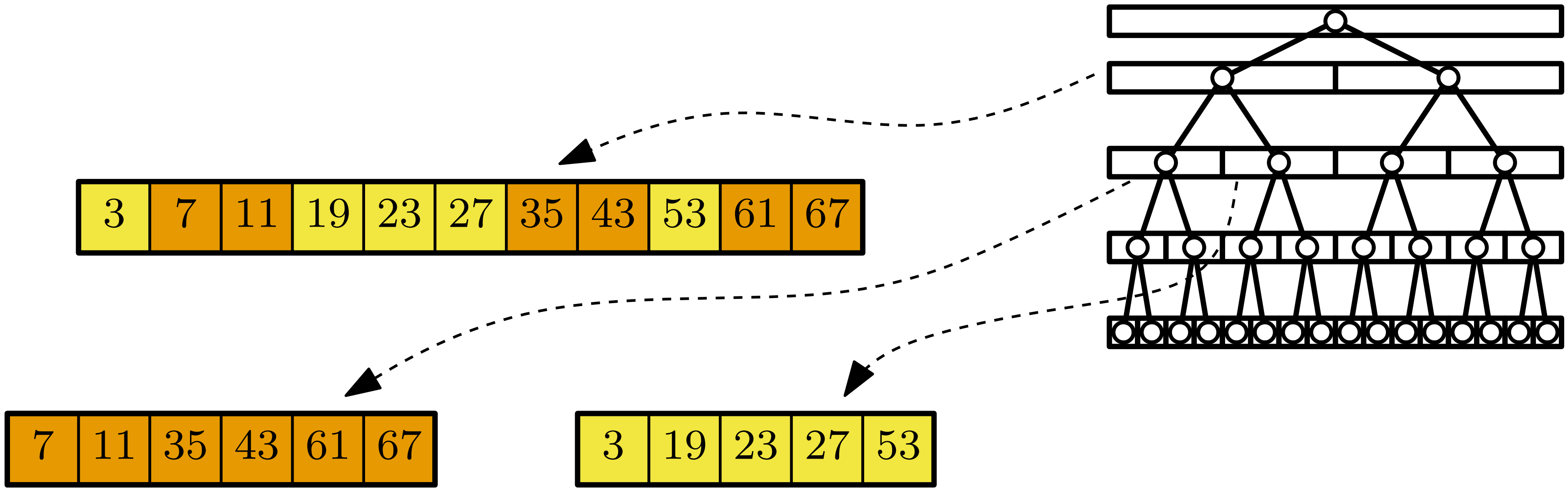
Improving the query time



The arrays of points at the children

partition the array of the parent

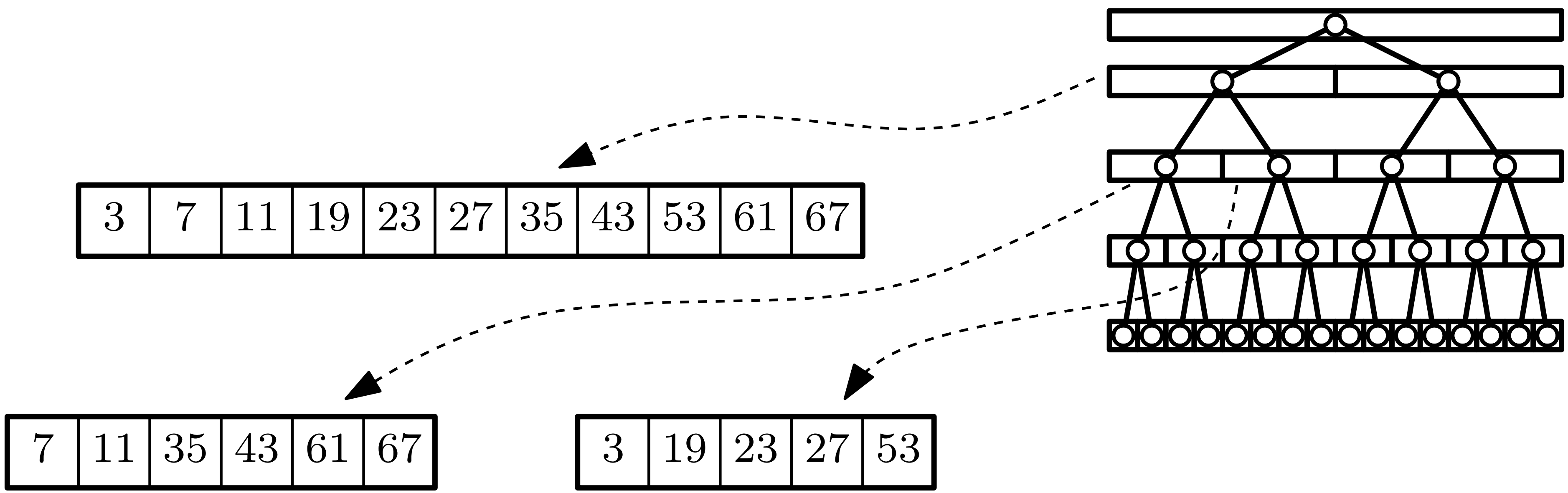
Improving the query time



The arrays of points at the children

partition the array of the parent

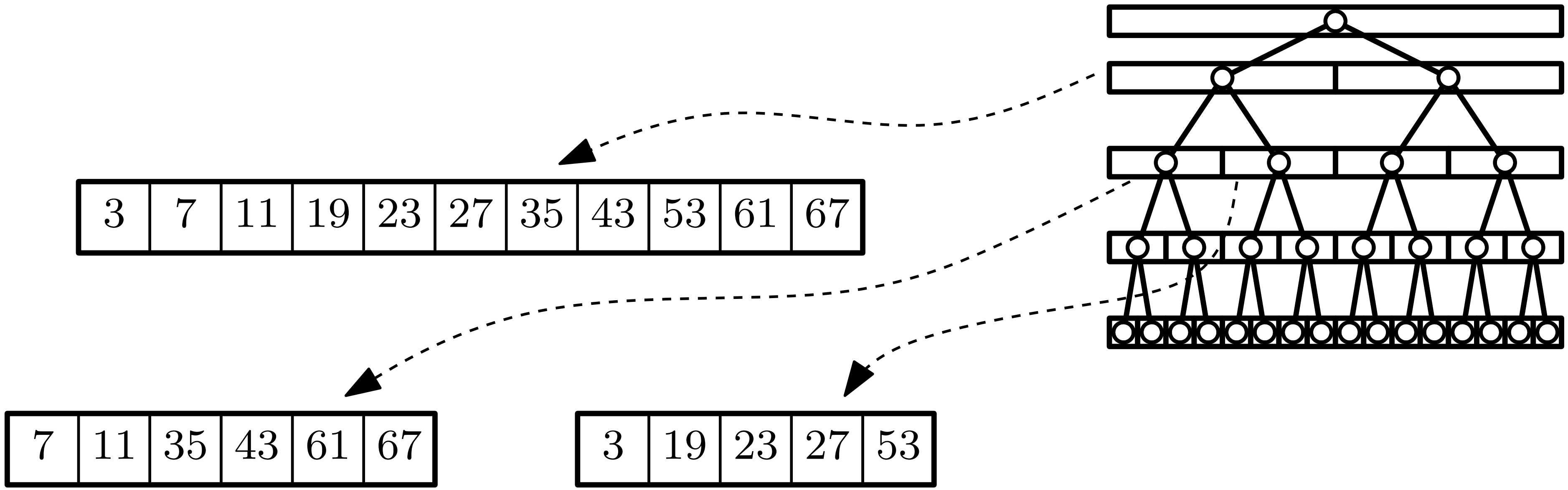
Improving the query time



The arrays of points at the children

partition the array of the parent

Improving the query time



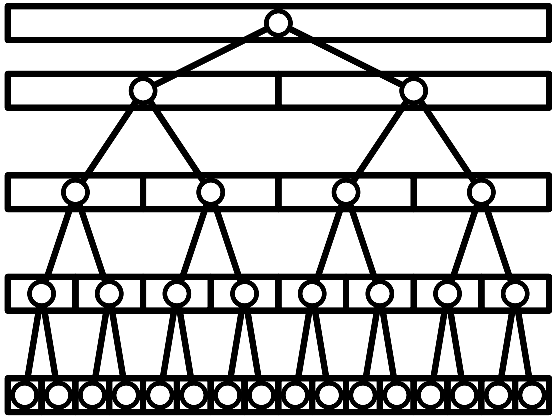
The arrays of points at the children

partition the array of the parent

The child arrays are sorted by y coordinate

(but have been partitioned by x coordinate)

Improving the query time



3	7	11	19	23	27	35	43	53	61	67

7	11	35	43	61	67

3	19	23	27	53

The arrays of points at the children

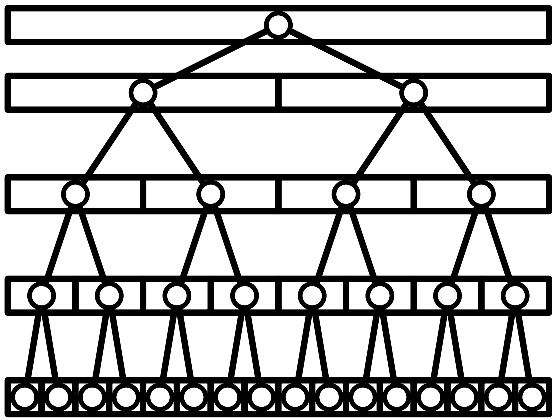
partition the array of the parent

The child arrays are sorted by y coordinate

(but have been partitioned by x coordinate)

Consider a point in the parent array...

Improving the query time



3	7	11	19	23	27	35	43	53	61	67

7	11	35	43	61	67

3	19	23	27	53

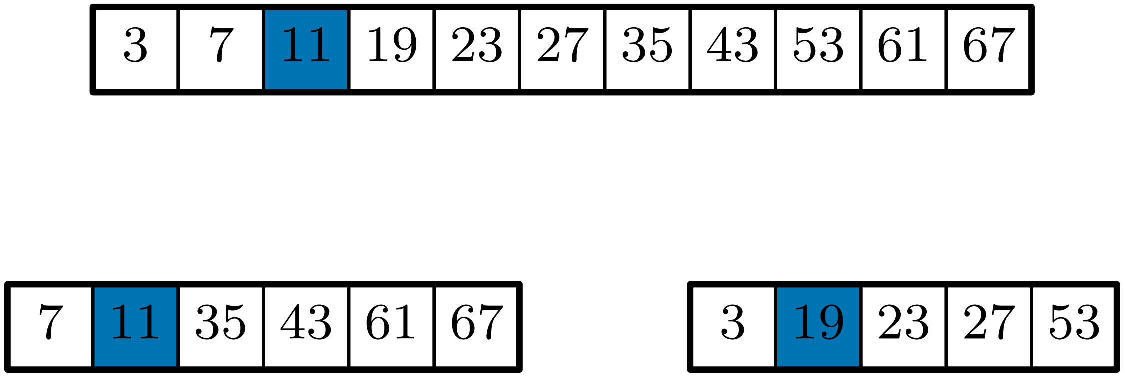
The arrays of points at the children
partition the array of the parent

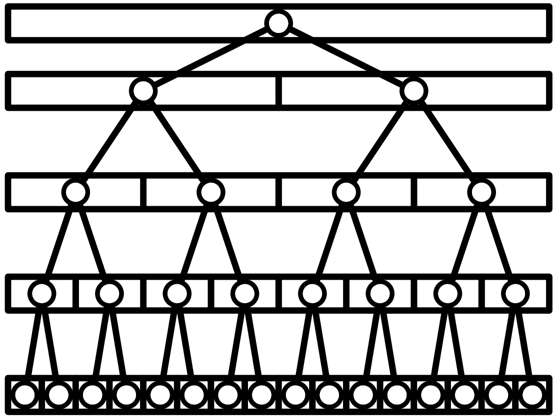
The child arrays are sorted by y coordinate

(but have been partitioned by x coordinate)

Consider a point in the parent array. . .

Improving the query time





The arrays of points at the children

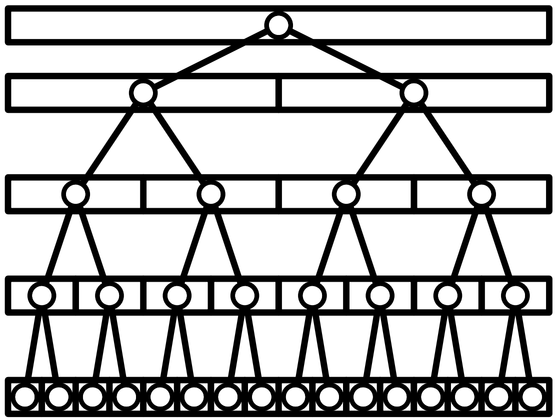
partition the array of the parent

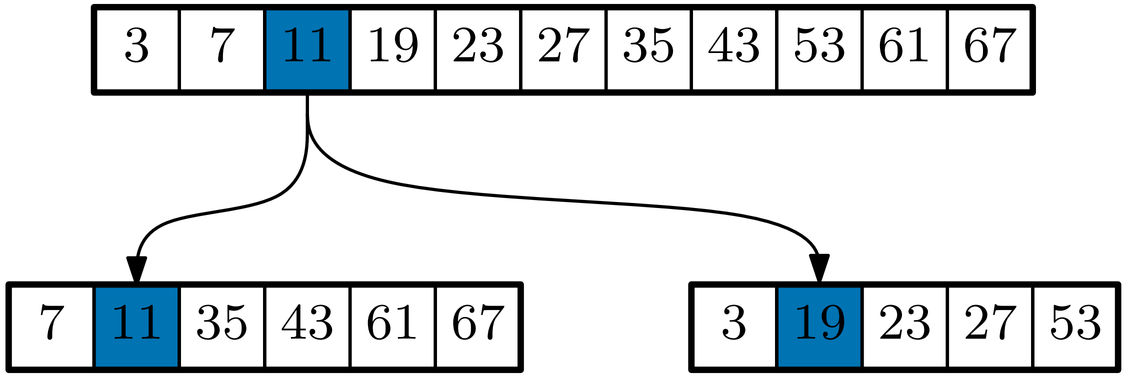
The child arrays are sorted by y coordinate

(but have been partitioned by x coordinate)

Consider a point in the parent array. . .

Improving the query time





The arrays of points at the children

partition the array of the parent

The child arrays are sorted by y coordinate

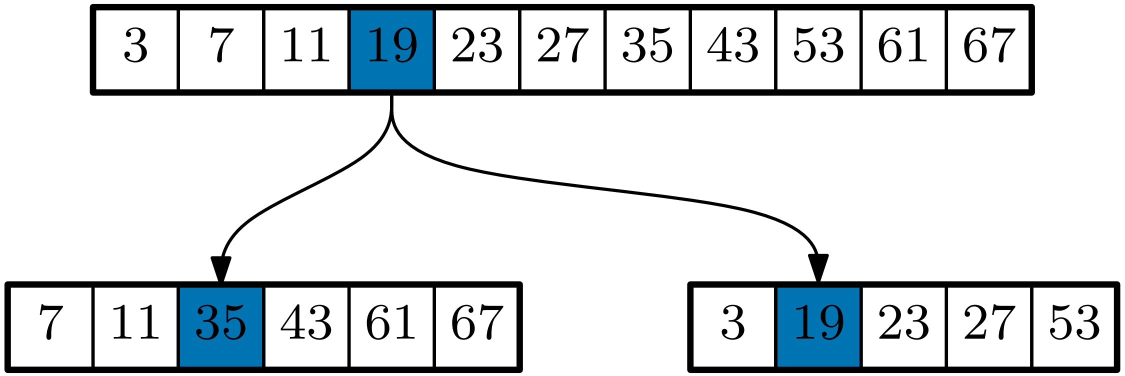
(but have been partitioned by x coordinate)

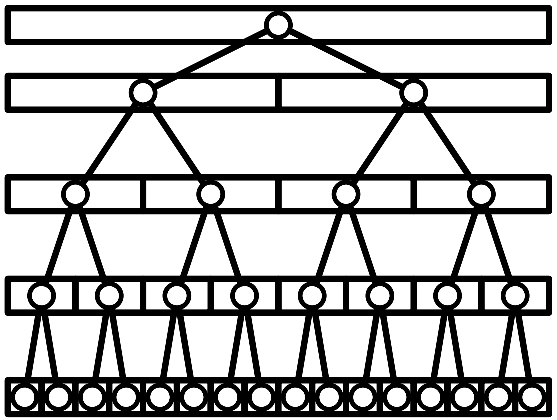
Consider a point in the parent array...

we add a link to its successor in both child arrays

(we do this for every point during preprocessing)

Improving the query time





The arrays of points at the children

partition the array of the parent

The child arrays are sorted by y coordinate

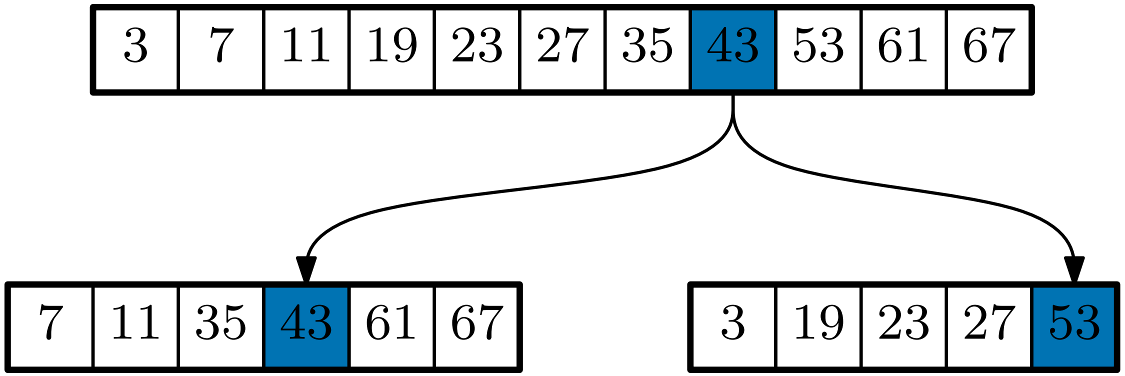
(but have been partitioned by x coordinate)

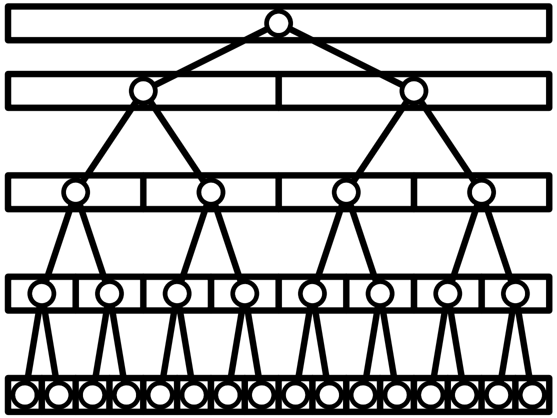
Consider a point in the parent array...

we add a link to its successor in both child arrays

(we do this for every point during preprocessing)

Improving the query time





The arrays of points at the children

partition the array of the parent

The child arrays are sorted by y coordinate

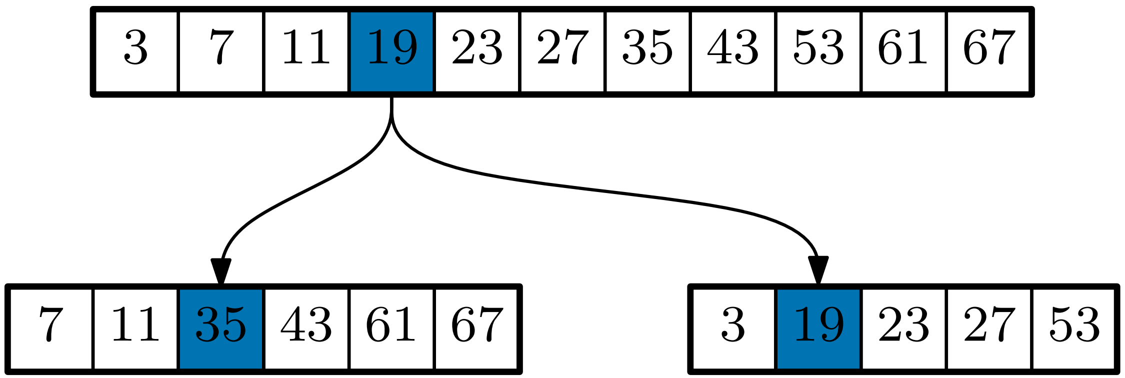
(but have been partitioned by x coordinate)

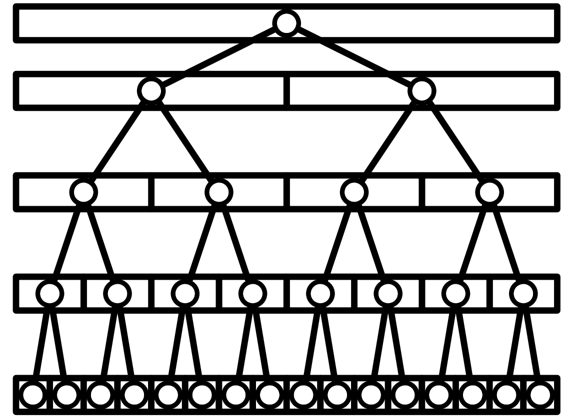
Consider a point in the parent array...

we add a link to its successor in both child arrays

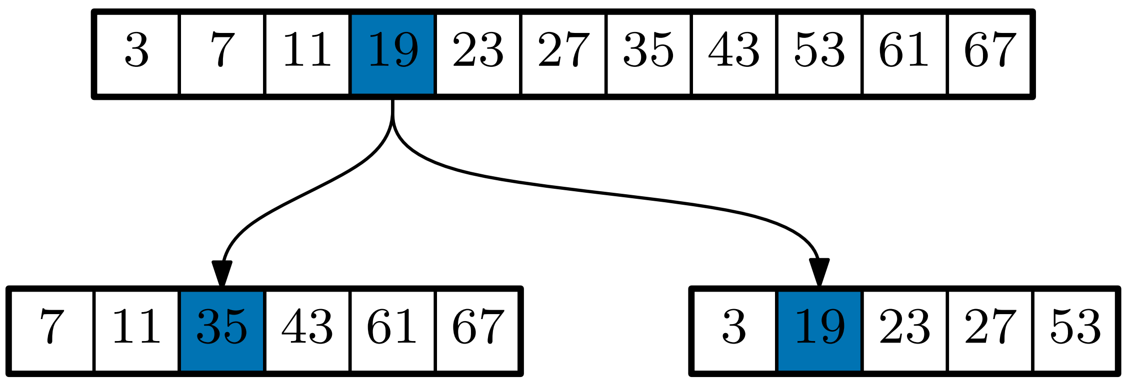
(we do this for every point during preprocessing)

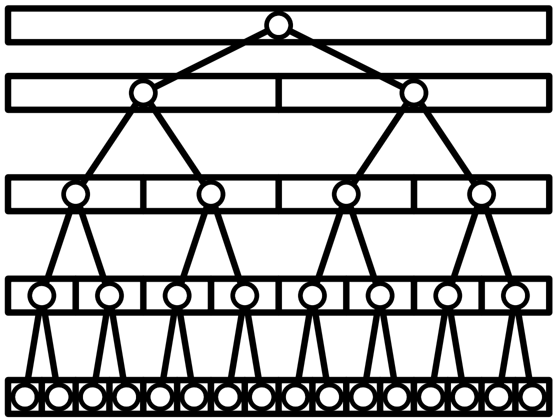
Improving the query time





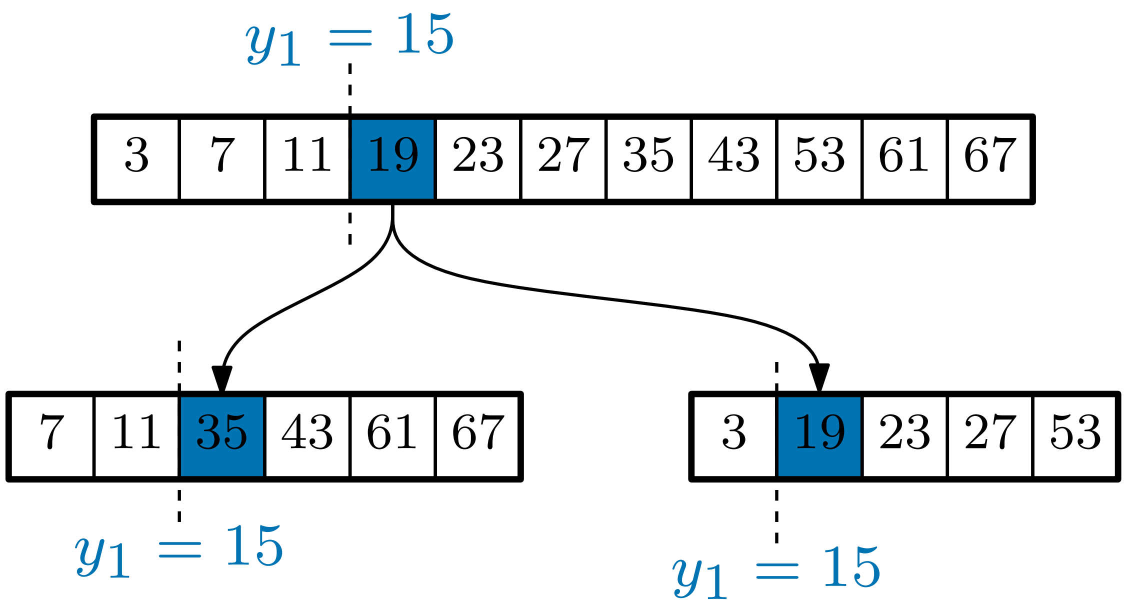
Improving the query time

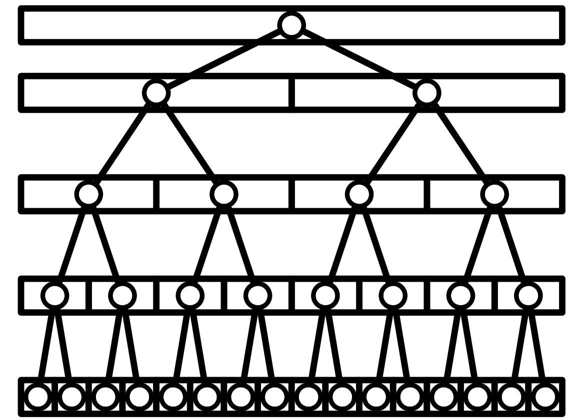




Observation if we know where the successor of y_1 is in the parent, can find the successor in either child in $O(1)$ time

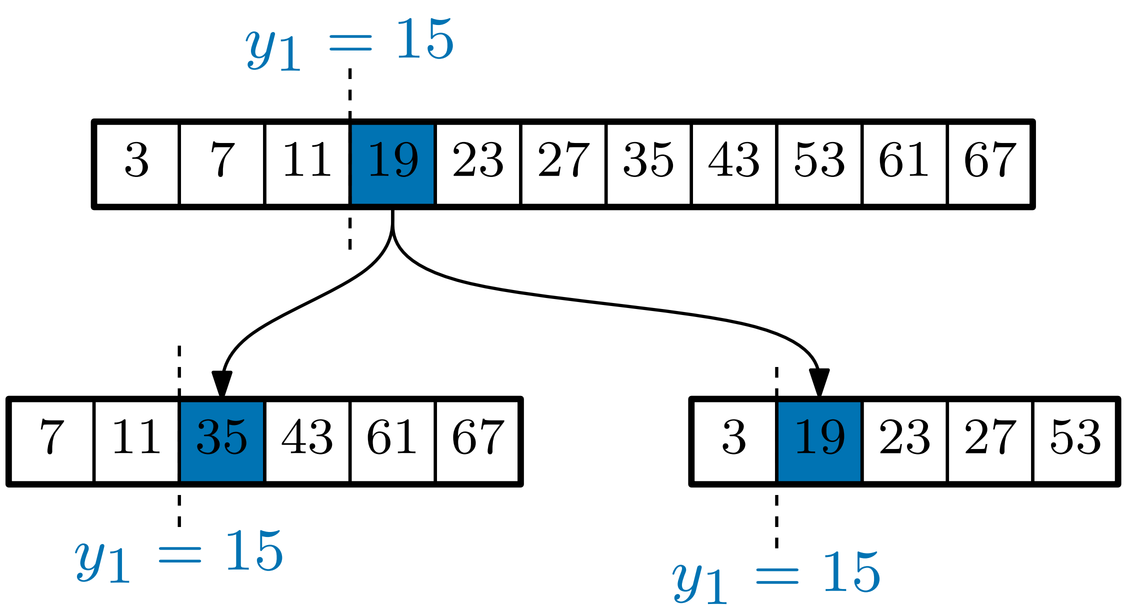
Improving the query time

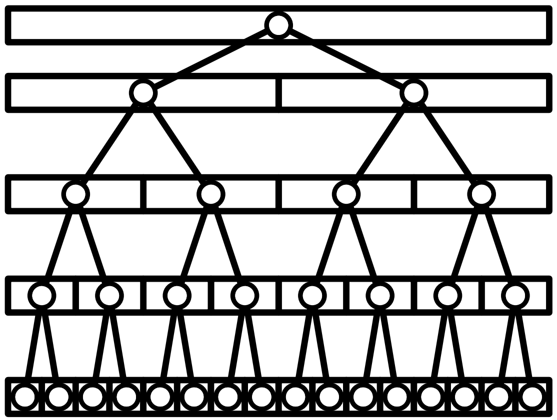




Observation if we know where the successor of y_1 is in the parent, can find the successor in either child in $O(1)$ time

Improving the query time



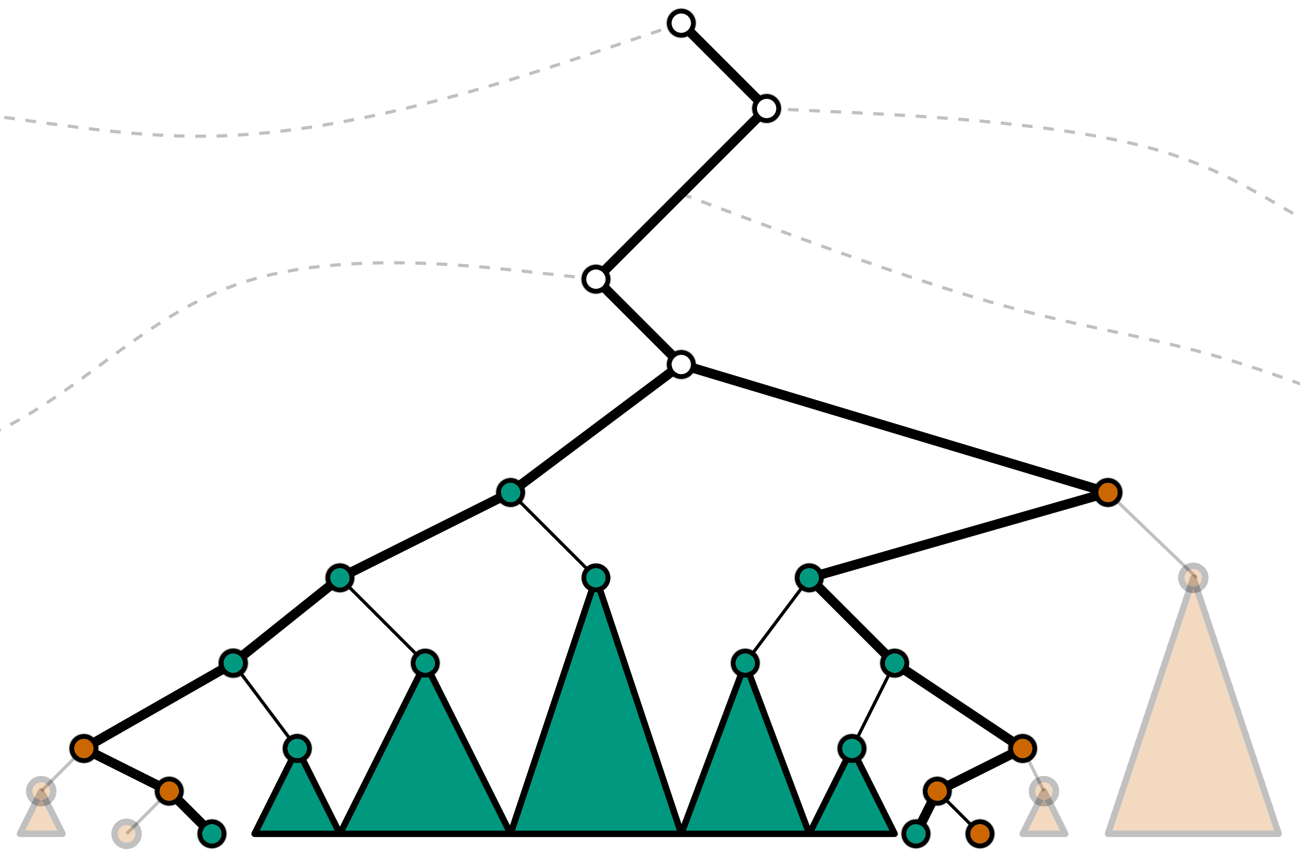


Observation if we know where the successor of y_1 is in the parent, can find the successor in either child in $O(1)$ time

adding these links doesn't increase the space or the prep time

The improved query time

How long does a query take?



Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range. . .

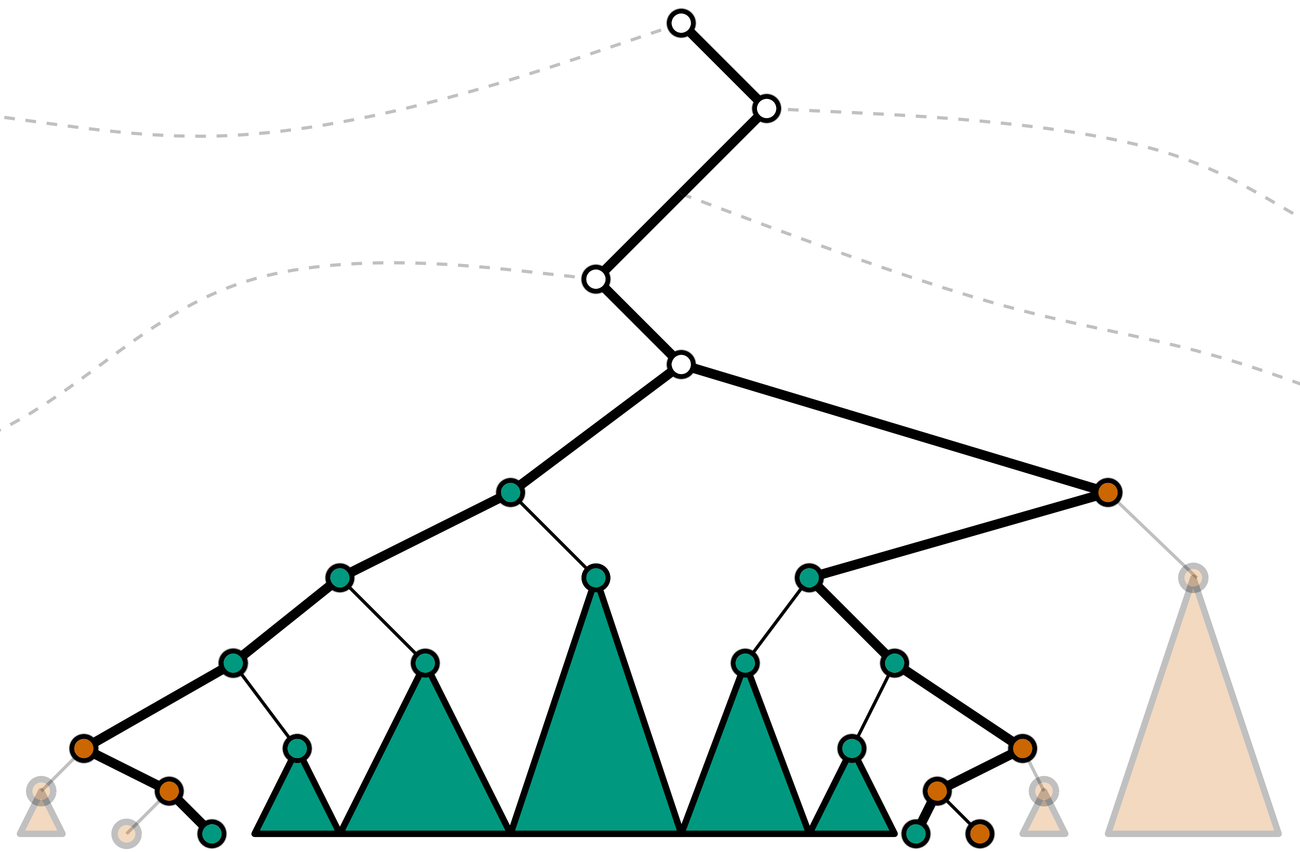
   use the 1D range structure for that subtree

   to filter the y coordinates

The improved query time

How long does a query take?

The paths have length $O(\log n)$



Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

   use the 1D range structure for that subtree

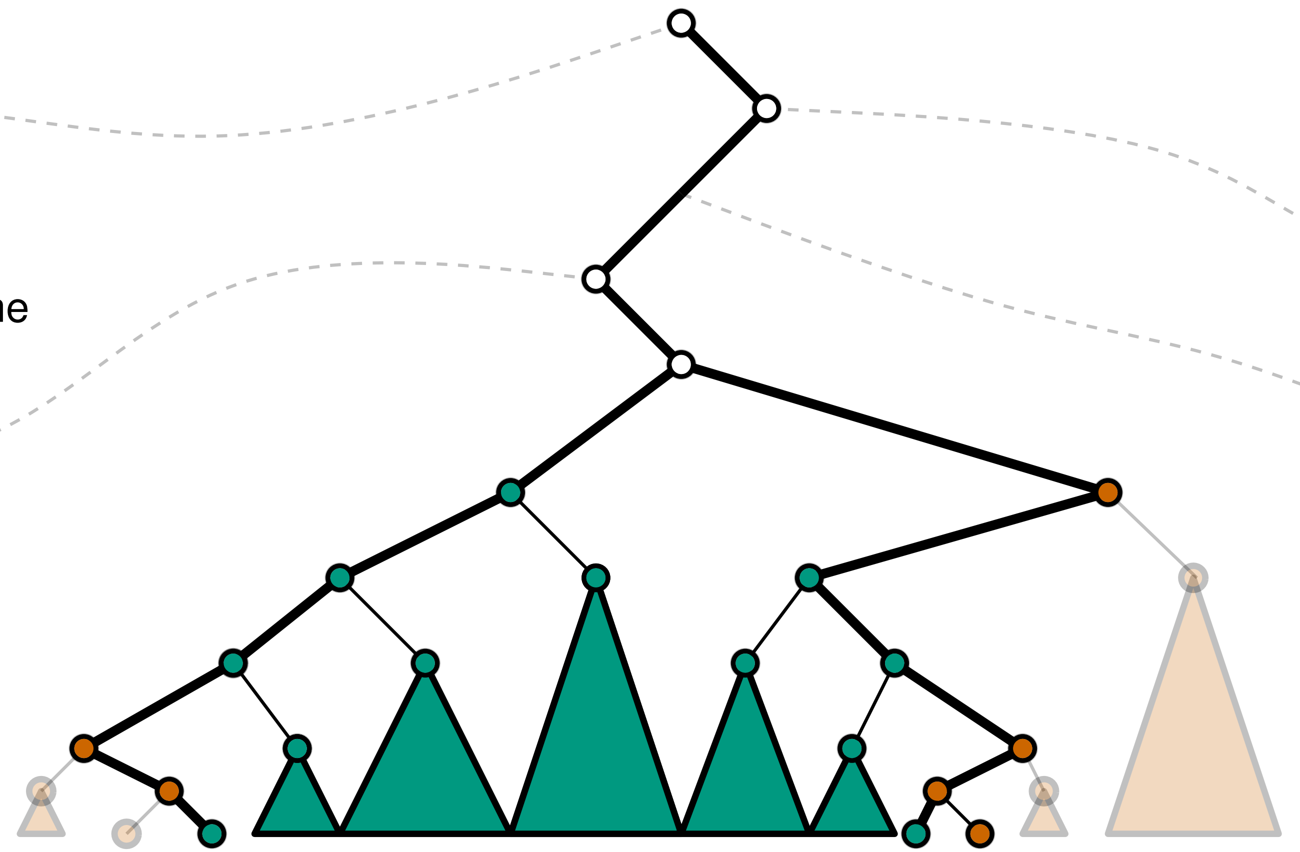
   to filter the y coordinates

The improved query time

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time



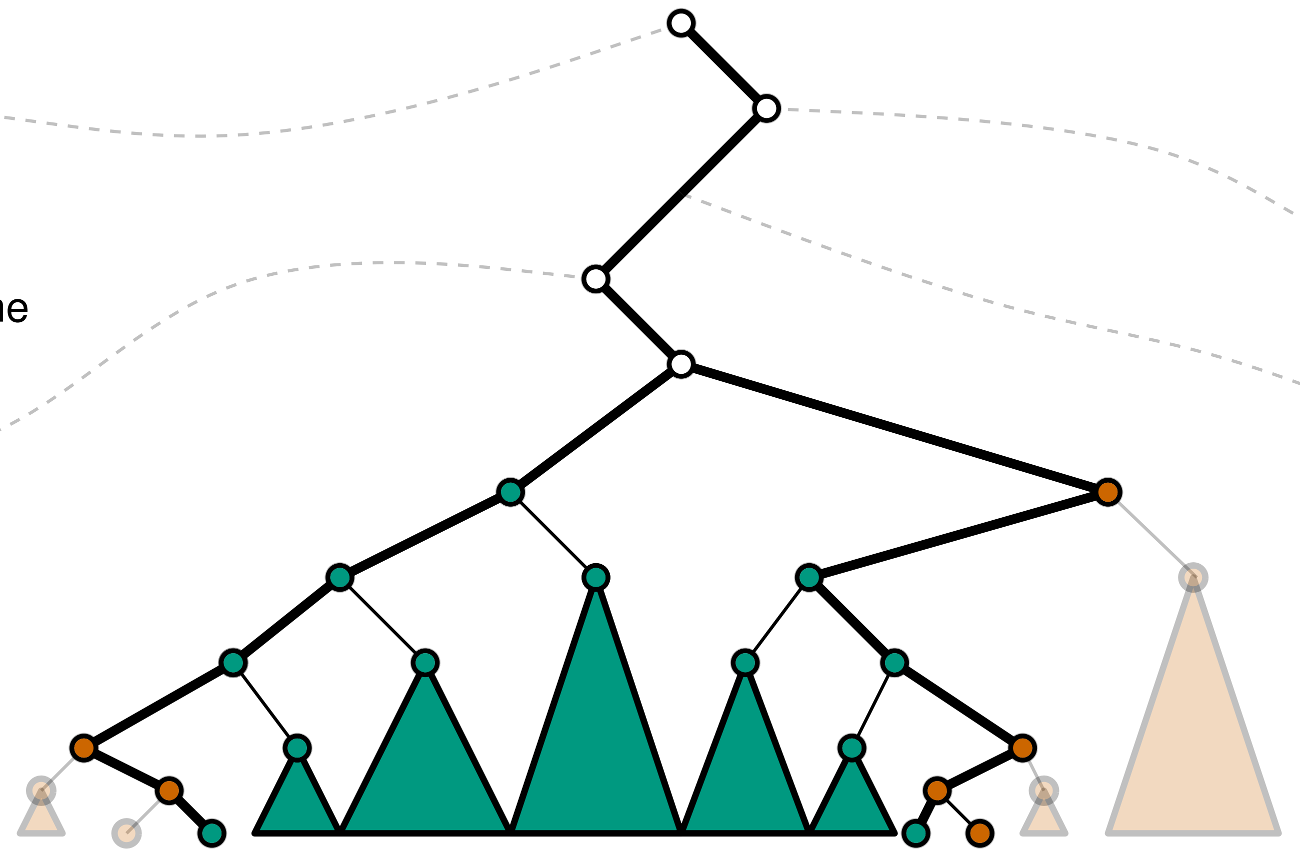
Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

   use the 1D range structure for that subtree

   to filter the y coordinates

The improved query time



How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

   use the 1D range structure for that subtree

   to filter the y coordinates

The improved query time

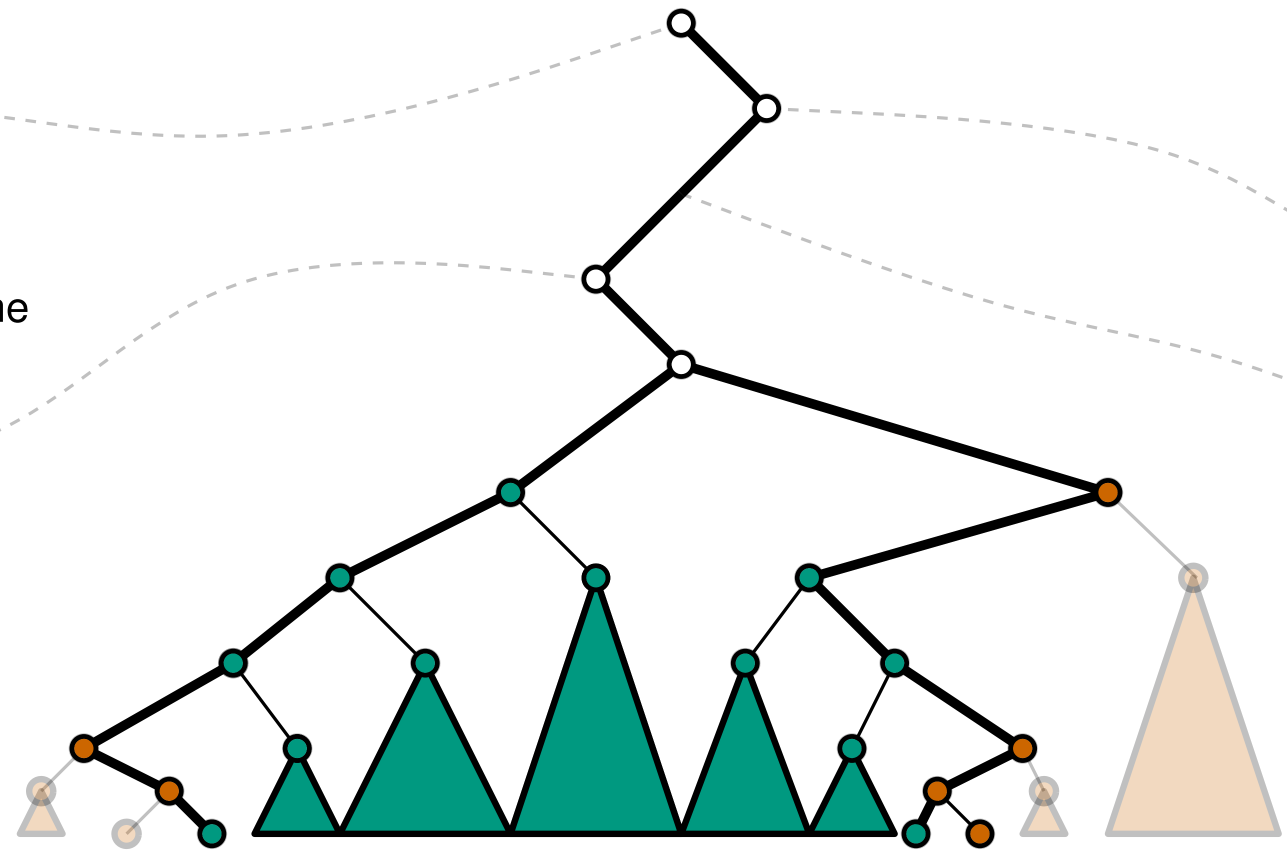
How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

We do $O(\log n)$ 1D lookups...



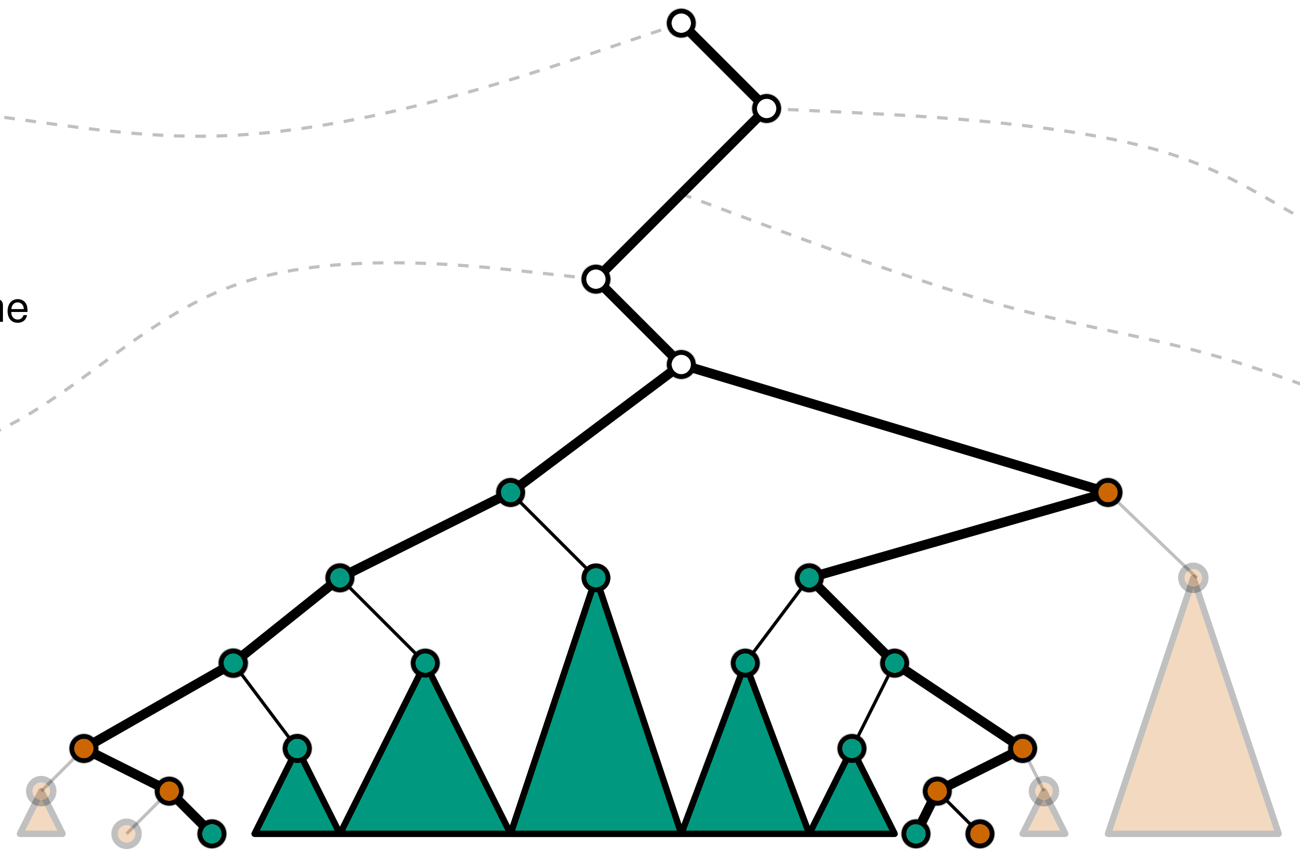
Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

   use the 1D range structure for that subtree

   to filter the y coordinates

The improved query time



How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

As for step 3,

We do $O(\log n)$ 1D lookups...

Each takes $O(k')$ time

Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

The improved query time

How long does a query take?

The paths have length $O(\log n)$

So steps 1. and 2. take $O(\log n)$ time

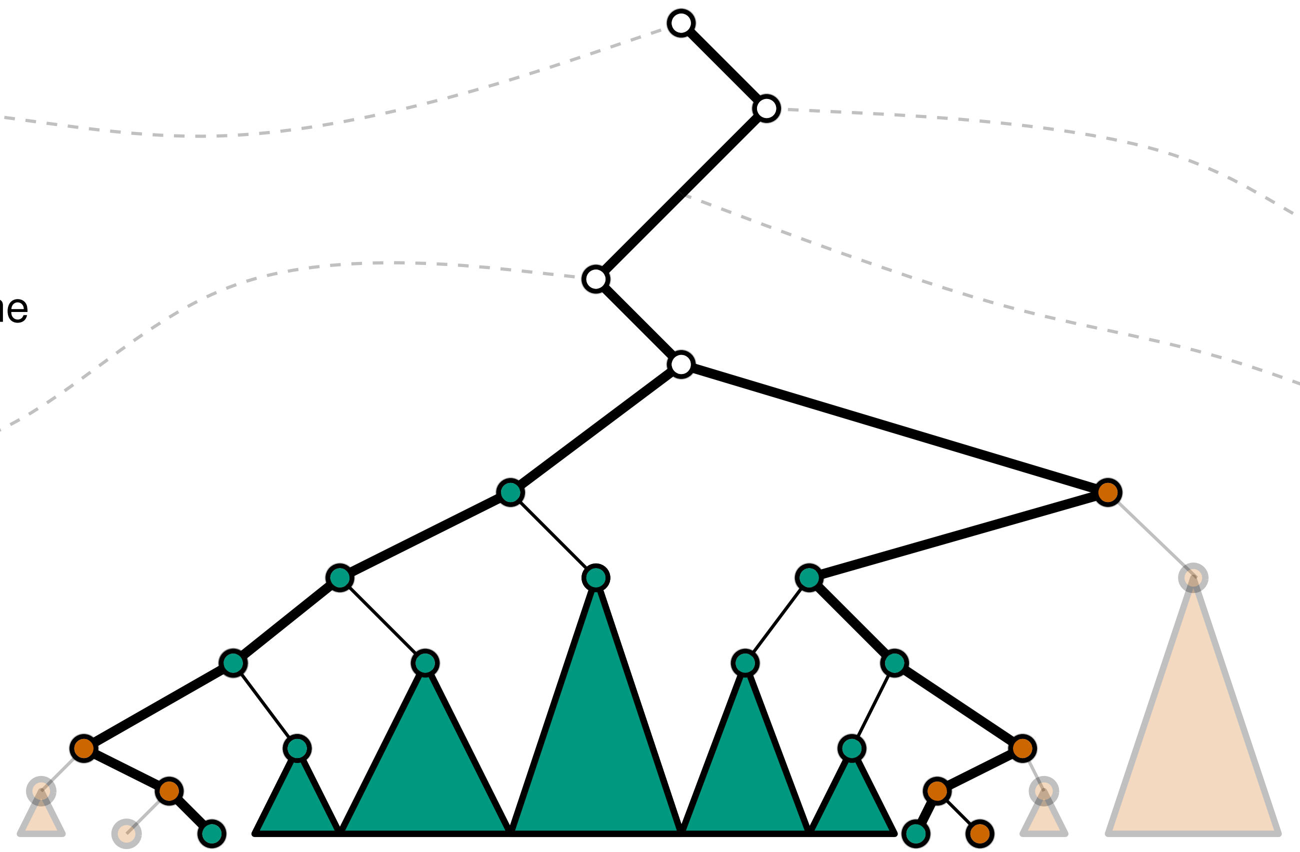
As for step 3,

We do $O(\log n)$ 1D lookups...

Each takes $O(k')$ time

This sums to...

$O(\log n + k)$



Query summary

1. Follow the paths to x_1 and x_2 (updating the successor to y_1 as you go)
2. Discard off-path subtrees where the x coordinates are *too large* or *too small*
3. For each off-path subtree where the x coordinates are in range...

use the 1D range structure for that subtree

to filter the y coordinates

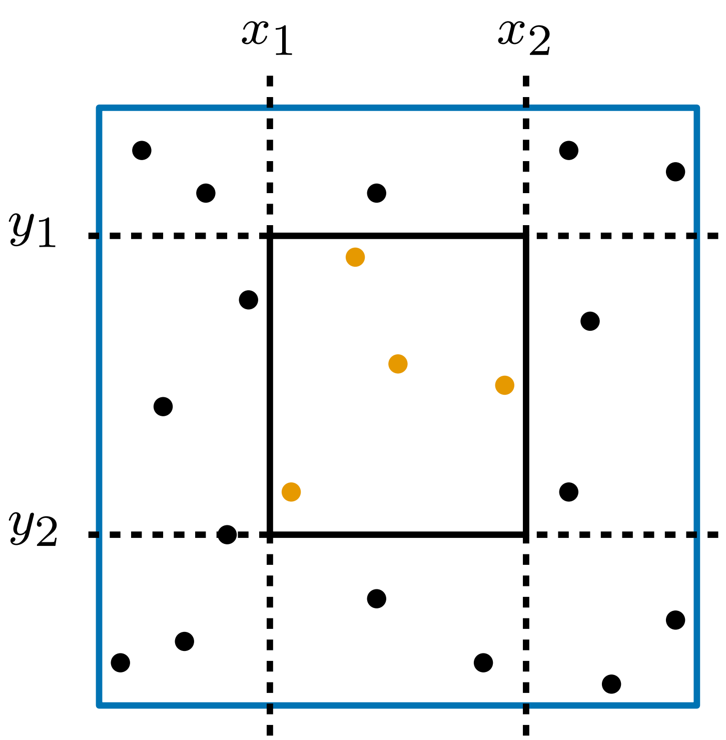
2D range searching

- A **2D range searching data structure** stores n distinct (x, y)-pairs and supports:

  the lookup(x_1, x_2, y_1, y_2) operation

  which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

  i.e. every (x, y) with $x_1 \leqslant x \leqslant x_2$ and $y_1 \leqslant y \leqslant y_2$.



Summary

$O(n \log n)$ prep time

$O(n \log n)$ space

$O(\log n + k)$ lookup time

where k is the number of points reported

we improved this :)

using fractional cascading