DYNAMIC PLANAR *POINT LOCATION*
WITH
SUB-LOGARITHMIC *LOCAL UPDATES*

Maarten Löffler *Utrecht University*

Joe Simons *University of California, Irvine*

Darren Strash *Intel Oregon*

DYNAMIC PLANAR *POINT LOCATION*
WITH
SUB-LOGARITHMIC *LOCAL UPDATES*

Maarten Löffler *Utrecht University*

Joe Simons *University of California, Irvine*

Darren Strash *Intel Oregon*

DYNAMIC PLANAR *POINT LOCATION*
WITH
SUB-LOGARITHMIC *LOCAL UPDATES*

Maarten Löffler *Utrecht University*

Joe Simons *University of California, Irvine*

Darren Strash *Intel Oregon*

DYNAMIC PLANAR *POINT LOCATION*

WITH

SUB-LOGARITHMIC *LOCAL UPDATES*

Maarten Löffler *Utrecht University*

Joe Simons *University of California, Irvine*

Darren Strash *Intel Oregon*

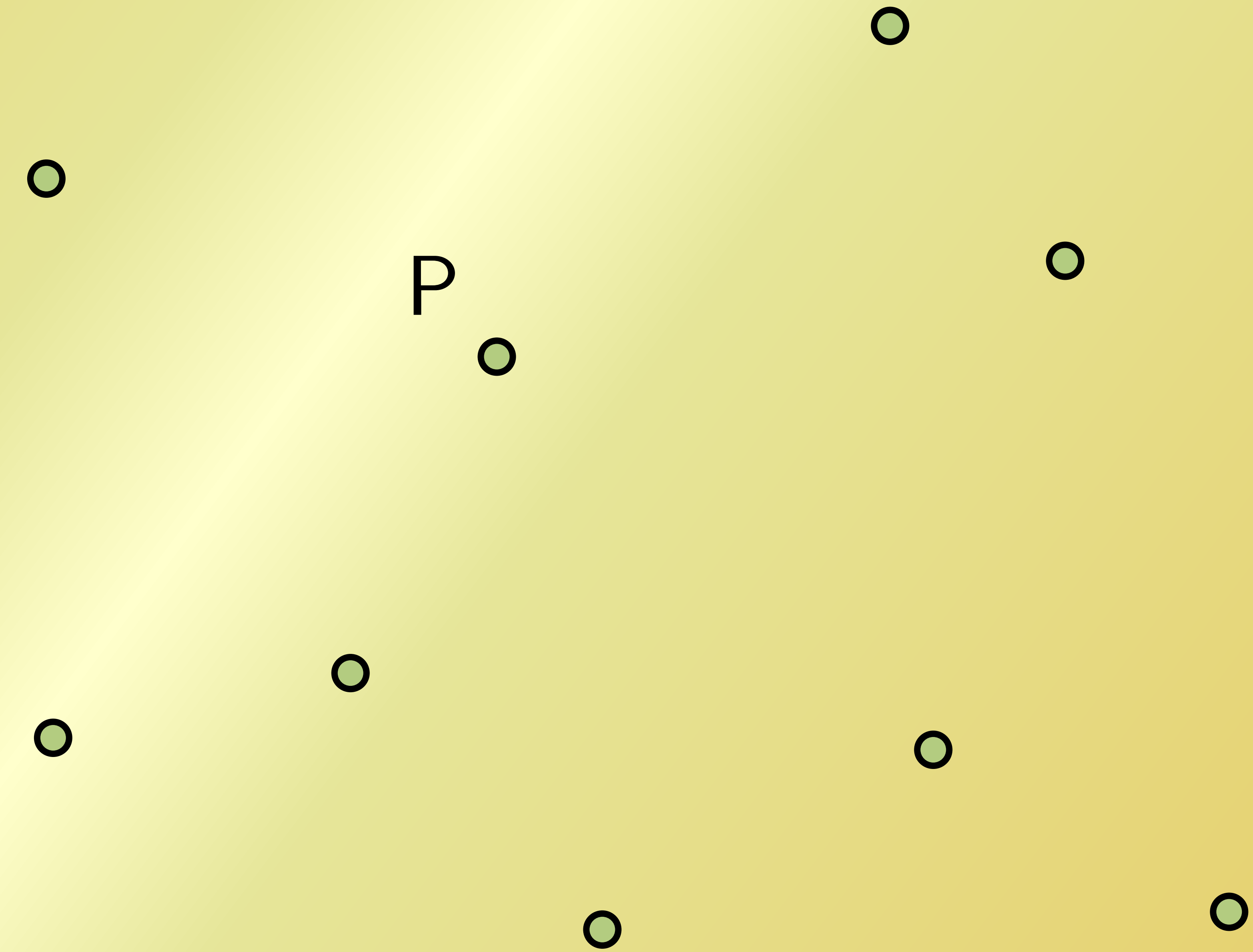
Let P be a set of n points in the plane.

Let P be a set of n points in the plane.

P

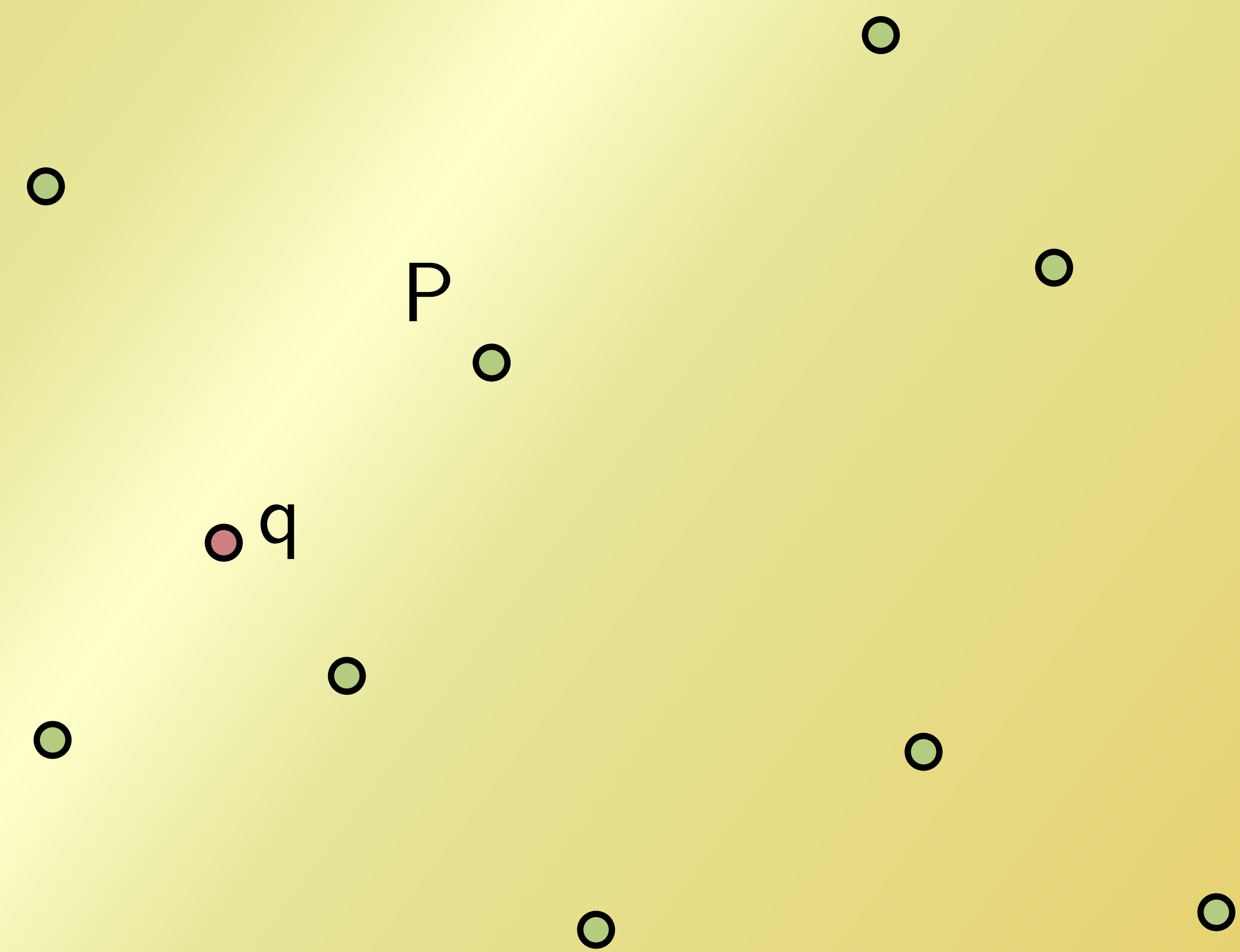
Let P be a set of n points in the plane.

Let q be one more point.



Let P be a set of n points in the plane.

Let q be one more point.

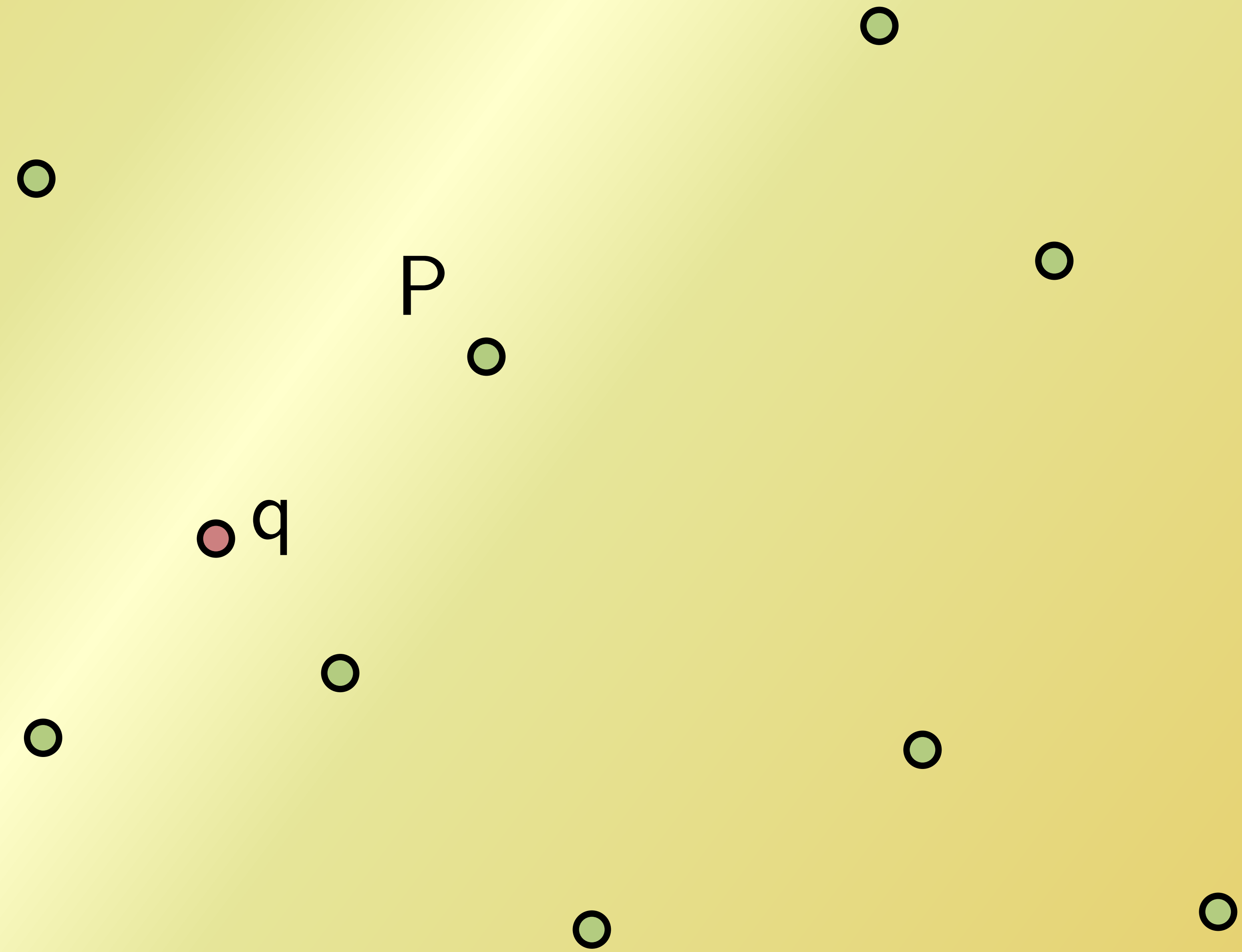


Let P be a set of n points in the plane.

Let q be one more point.

QUESTION

Is q an element of P?



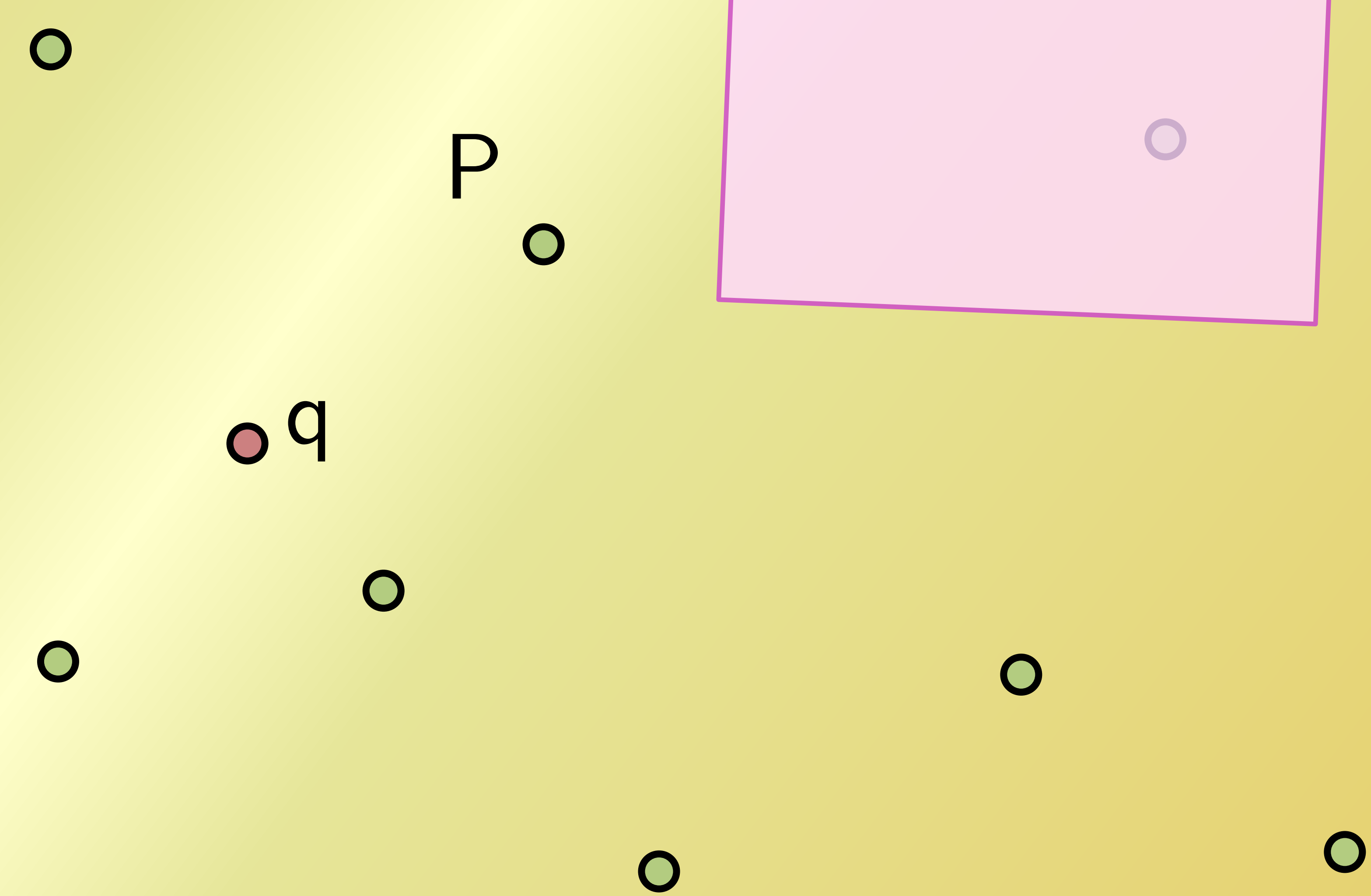
Let P be a set of n points in the plane.

Let q be one more point.

QUESTION

Is q an element of P?

Two possible answers: *yes* or *no*.



Let P be a set of n points in the plane.

Let q be one more point.

QUESTION

Is q an element of P?

Two possible answers: *yes* or *no*.

We can answer the question in linear time...

Let P be a set of n points in the plane.

Let q be one more point.

QUESTION
Is q an element of P?

P

q

Two possible answers: *yes* or *no*.

We can answer the question in linear time...

Or, after preprocessing P, in $O(\log n)$ time!

Let P be a set of n points in the plane.

Let q be one more point.

QUESTION

Is q an element of P?

Two possible answers: *yes* or *no*.

We can answer the question in linear time...

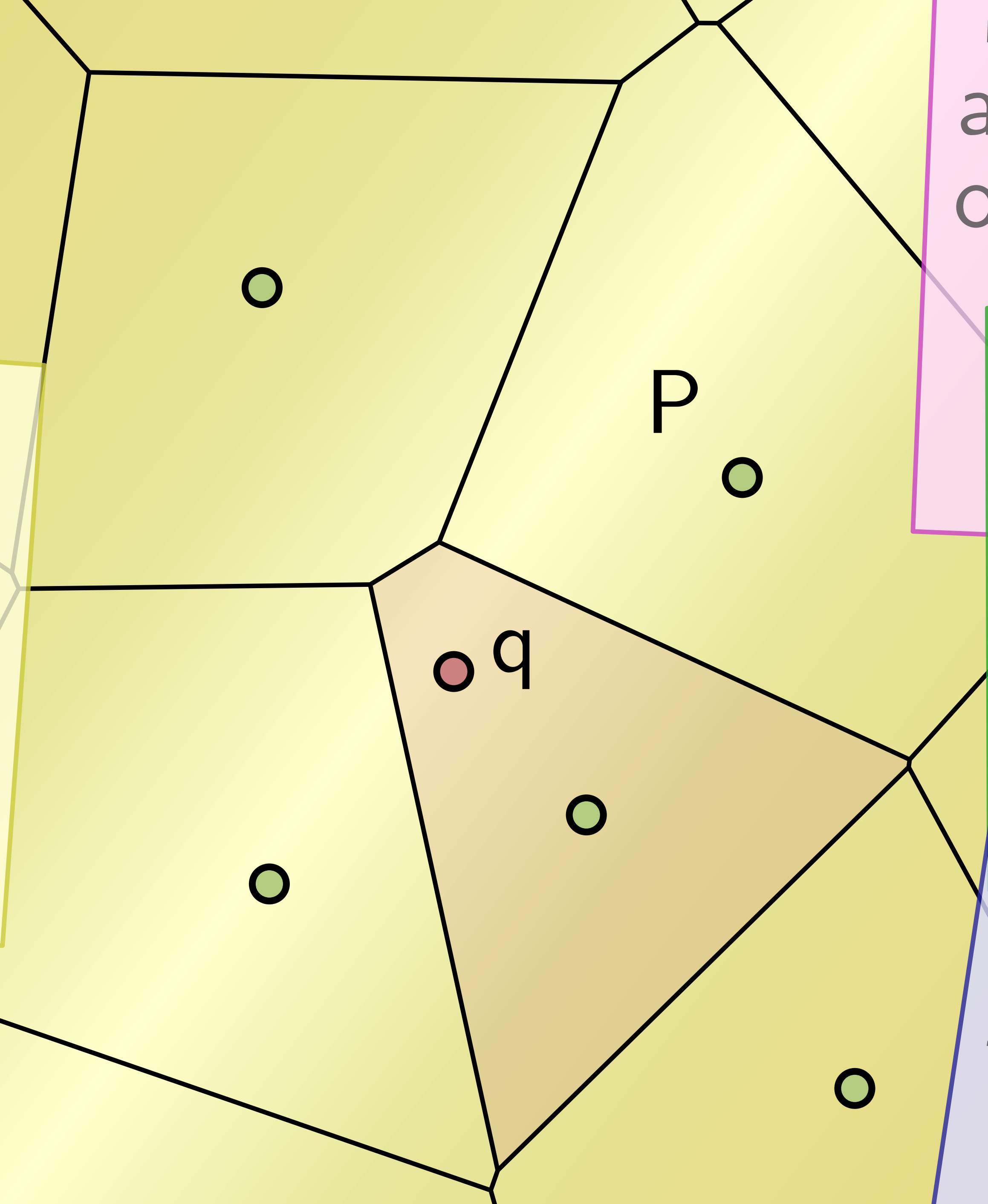
Or, after preprocessing P, in $O(\log n)$ time!

Let P be a set of n points in the plane.

Let q be one more point.

QUESTION

Is q an element of P?



Two possible answers: *yes* or *no*.

We can answer the question in linear time...

Or, after preprocessing P, in $O(\log n)$ time!

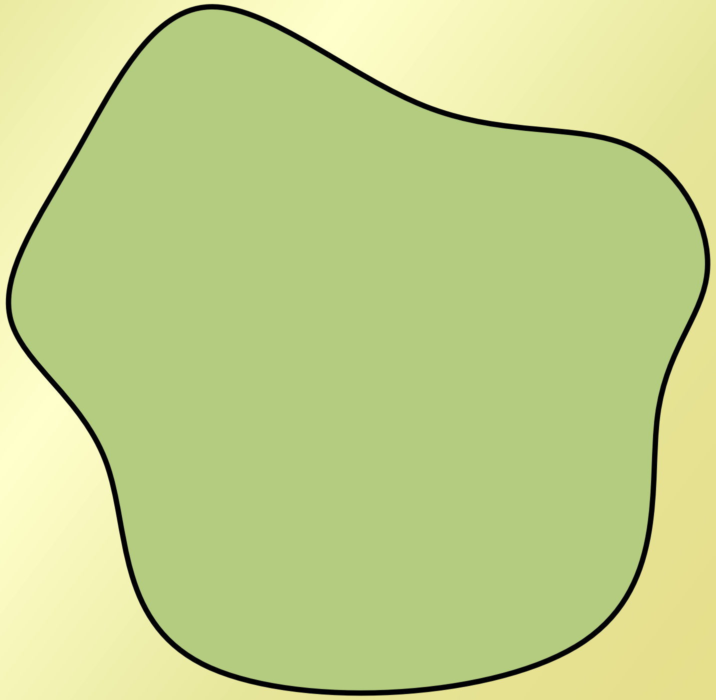
Now, suppose our points are *imprecise*.

Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

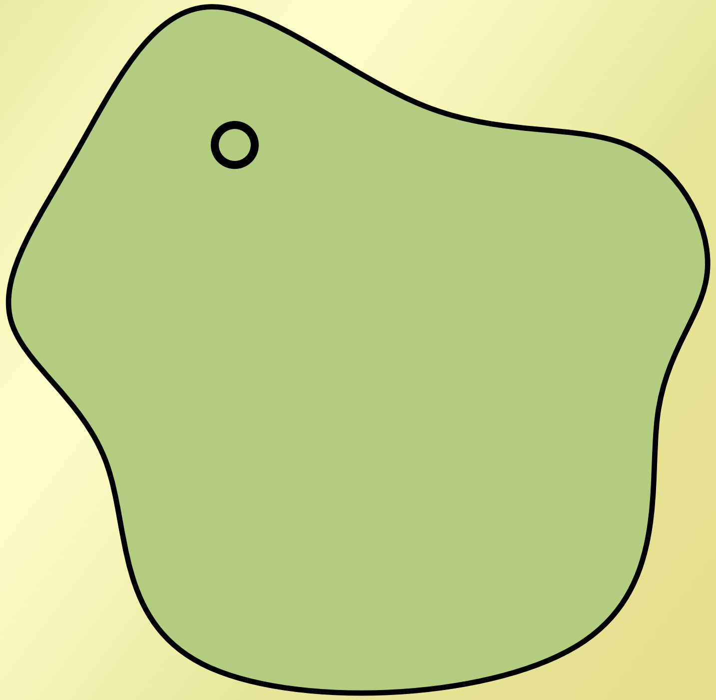
Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.



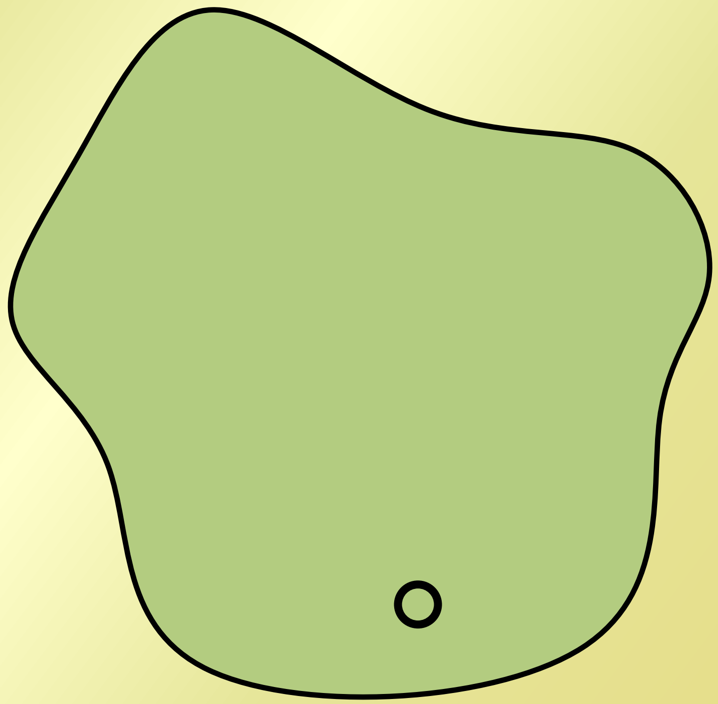
Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.



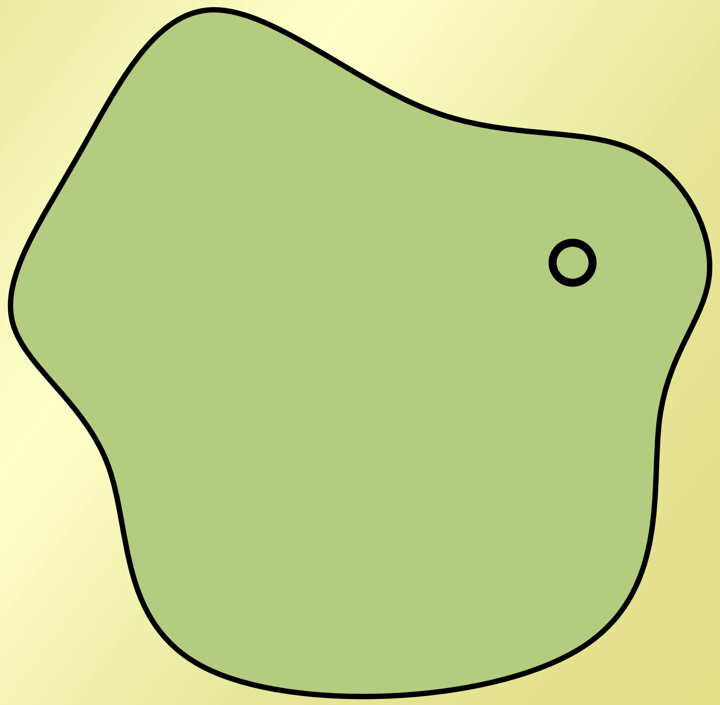
Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.



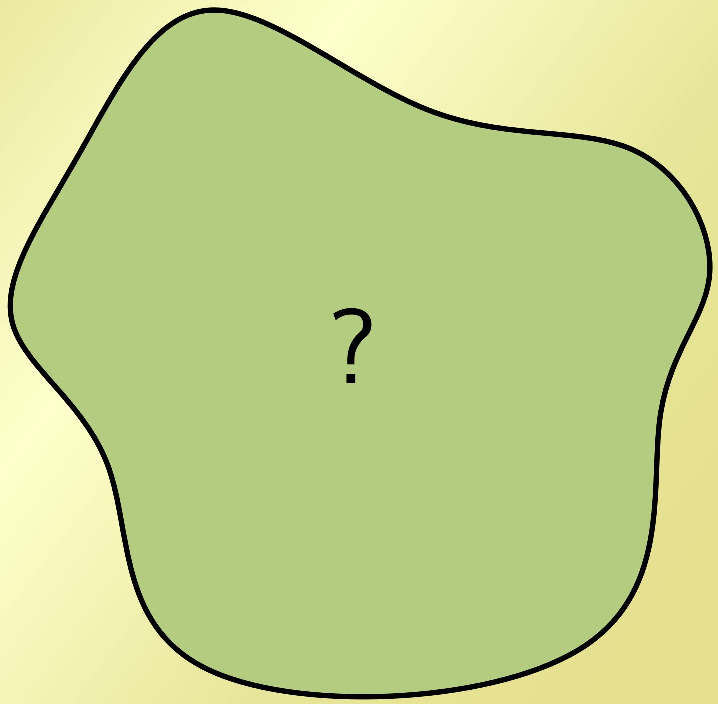
Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.



Now, suppose our points are *imprecise*.

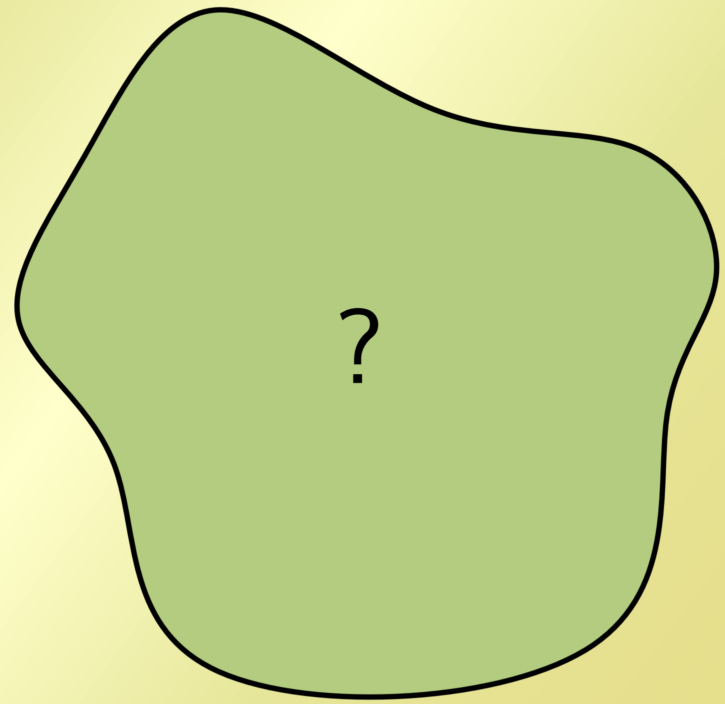
That is, each point is given as a region of *potential* locations.



Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

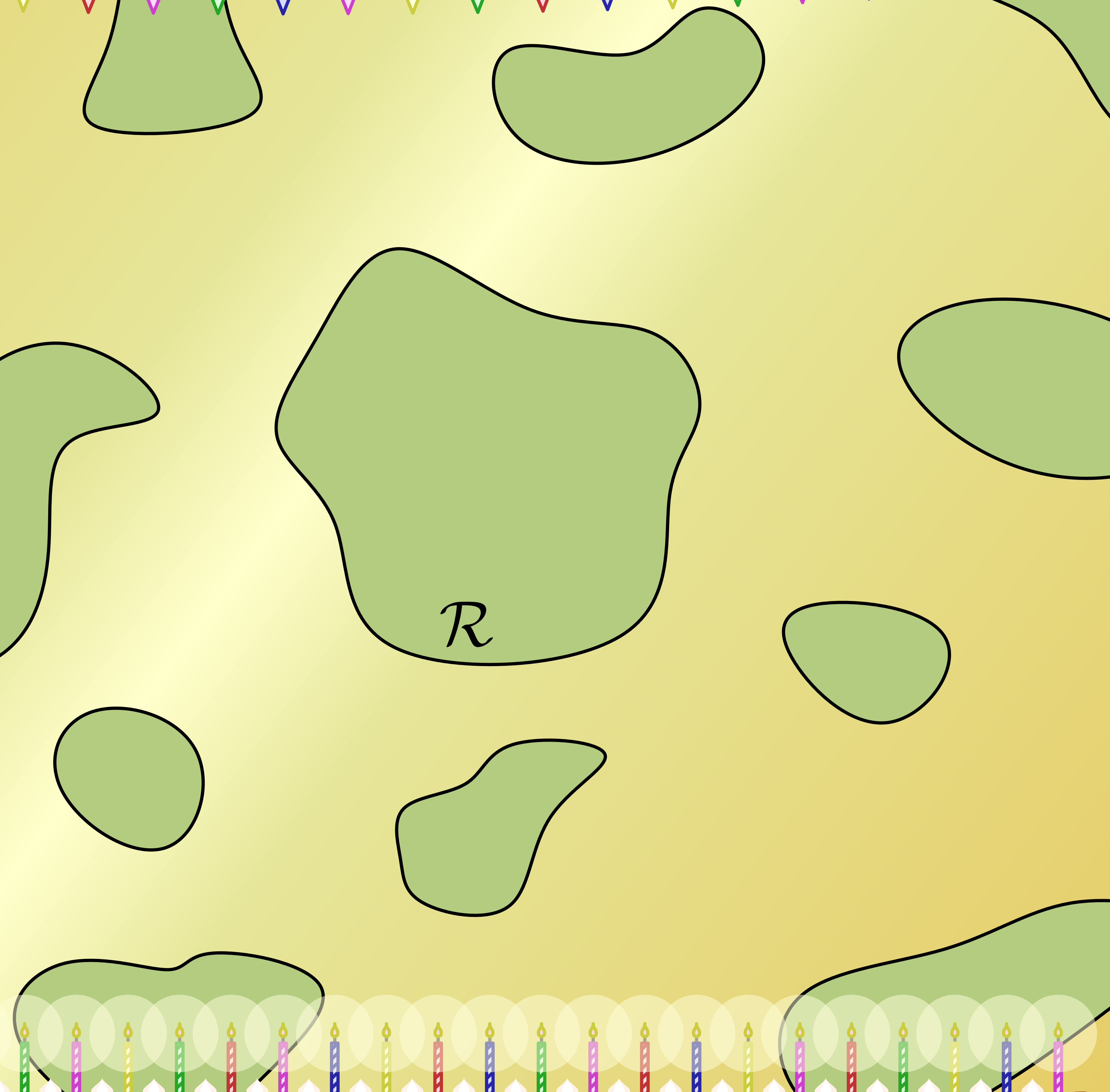
Let $\mathcal{R}$ be a set of n such regions in the plane.



Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

Let $\mathcal{R}$ be a set of **n** such regions in the plane.



Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

Let $\mathcal{R}$ be a set of **n** such regions in the plane.

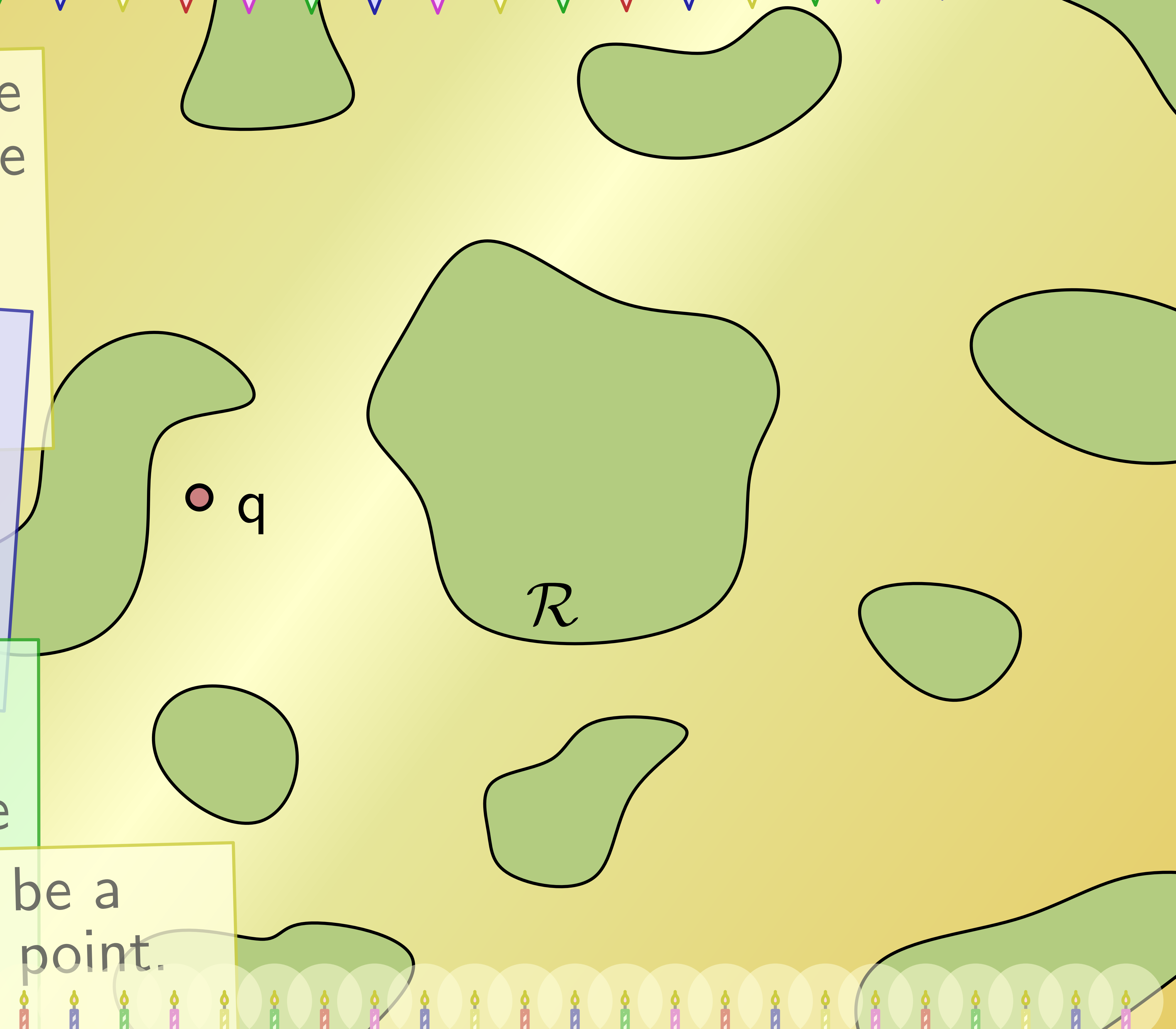
Let **q** be a query point.

Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

Let $\mathcal{R}$ be a set of $\mathbf{n}$ such regions in the plane.

Let $\mathbf{q}$ be a query point.



Now, suppose our points are *imprecise*.

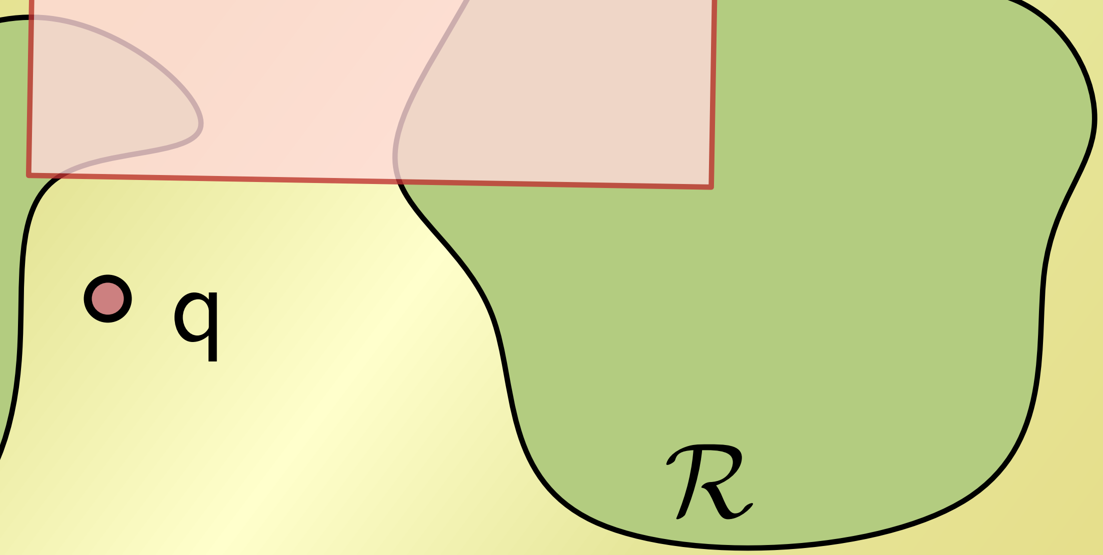
That is, each point is given as a region of *potential* locations.

Let $\mathcal{R}$ be a set of **n** such regions in the plane.

Let **q** be a query point.

QUESTION

Is **q** an element of P?



Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

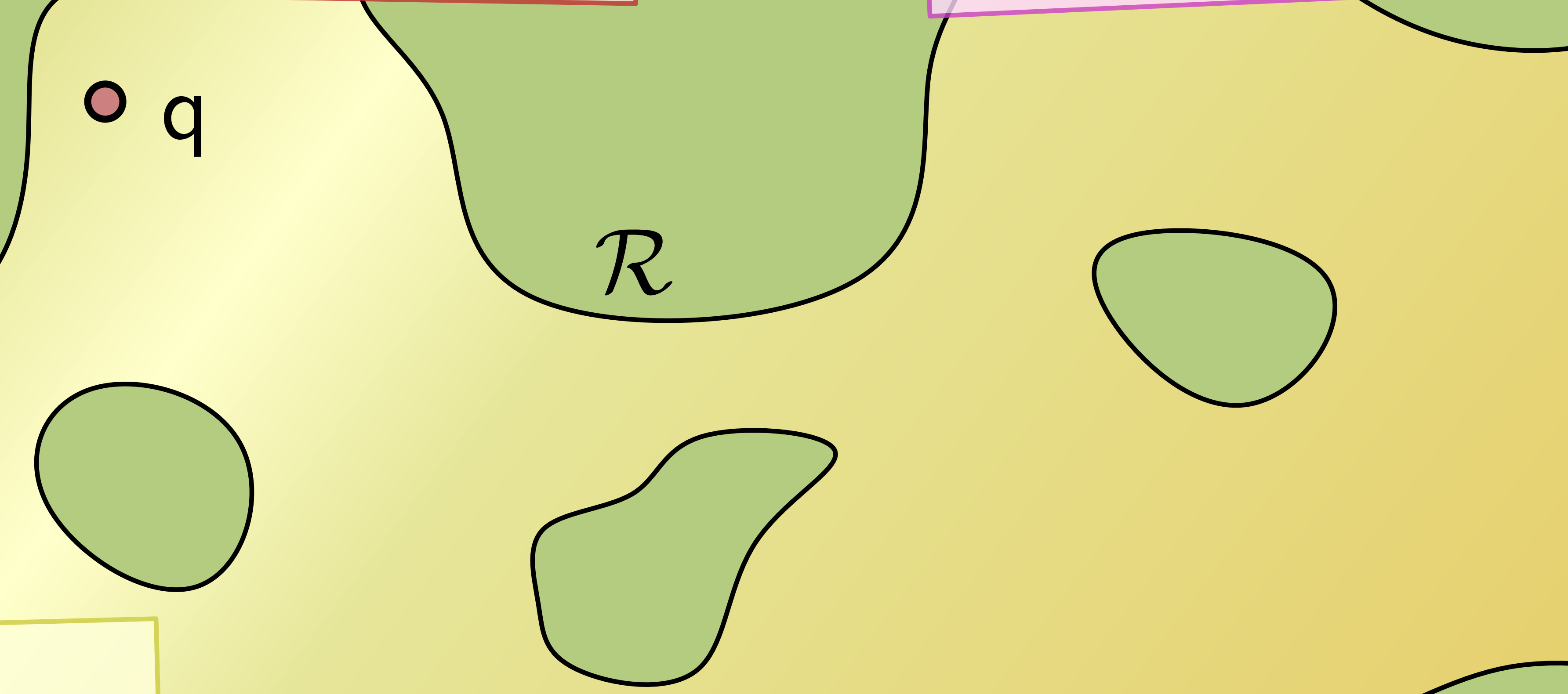
Let $\mathcal{R}$ be a set of n such regions in the plane.

Let q be a query point.

QUESTION

Is q an element of P?

Two possible answers: *maybe* or *no*.



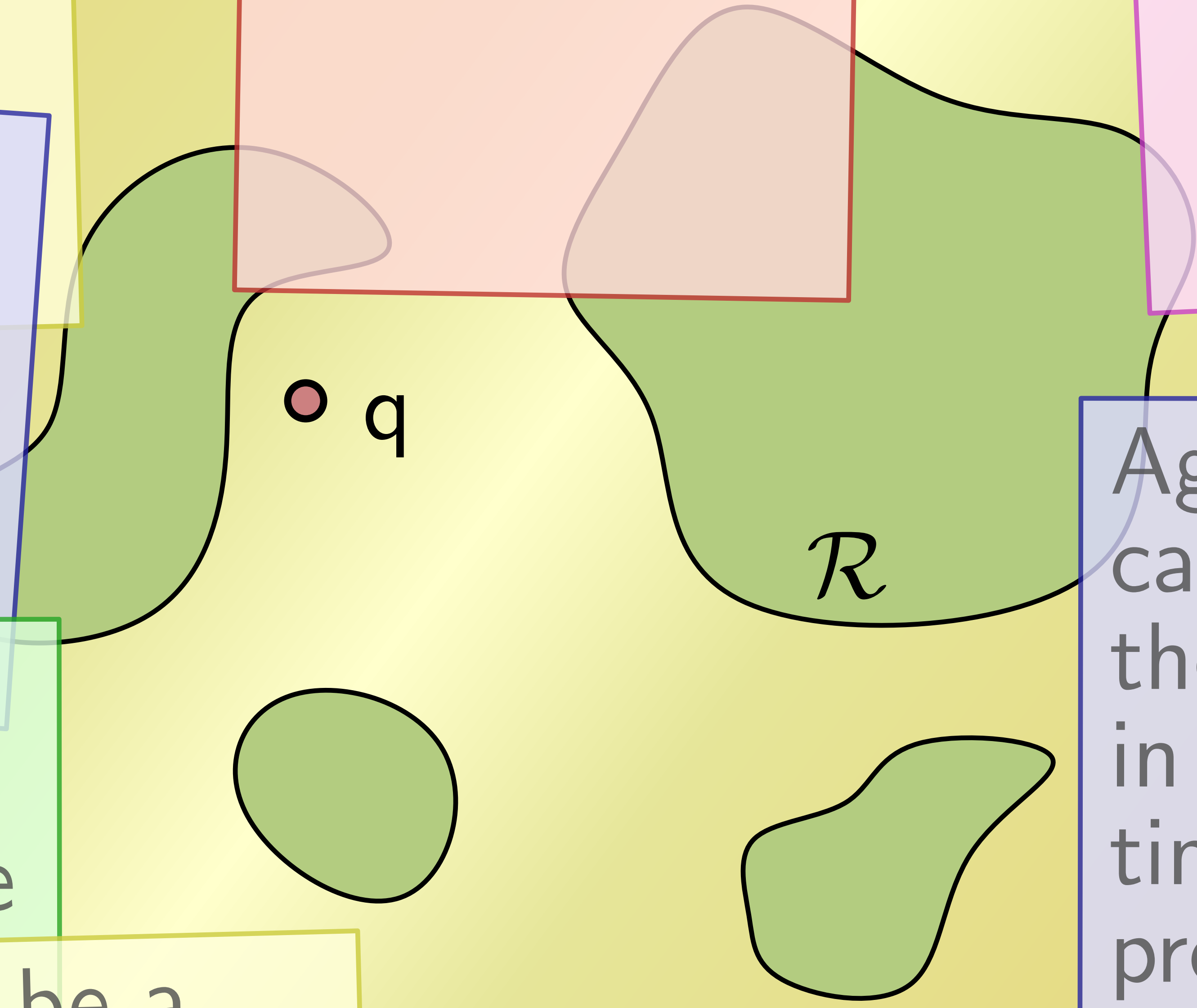
Now, suppose our points are *imprecise*.

That is, each point is given as a region of *potential* locations.

Let $\mathcal{R}$ be a set of $\mathbf{n}$ such regions in the plane.

Let $\mathbf{q}$ be a query point.

QUESTION
Is $\mathbf{q}$ an element of P?



Two possible answers: *maybe* or *no*.

Again, we can answer the question in logarithmic time after preprocessing

Suppose furthermore that our points are *dynamic*.

Suppose furthermore that our points are *dynamic*.

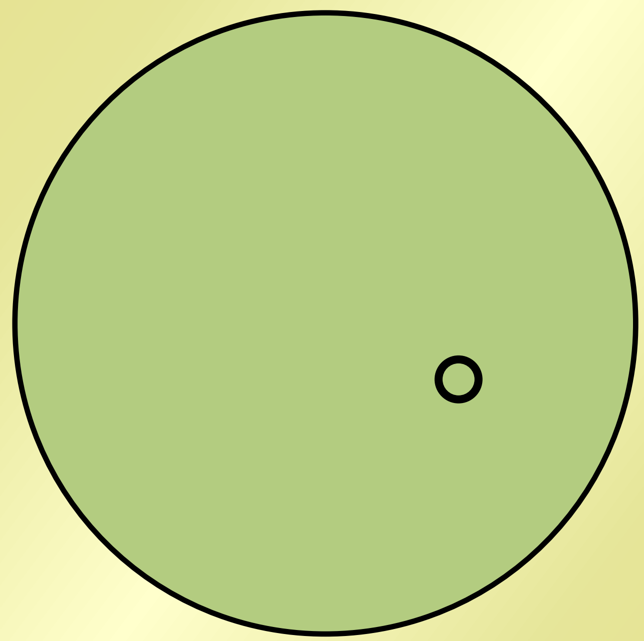
Our estimate of a point's location may change. . .

Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

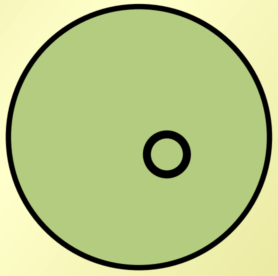
Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .



Suppose furthermore that our points are *dynamic*.

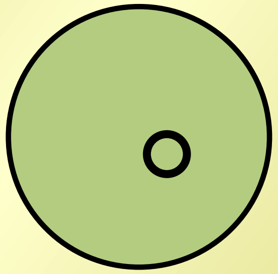
Our estimate of a point's location may change…



Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

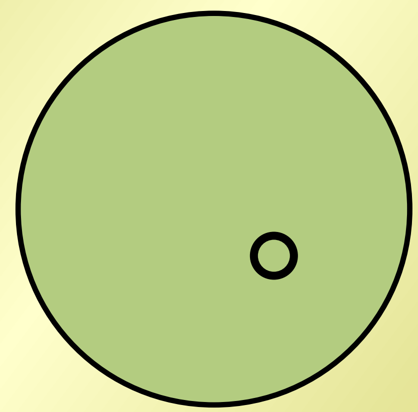
. . . or the true location itself of a point may change.



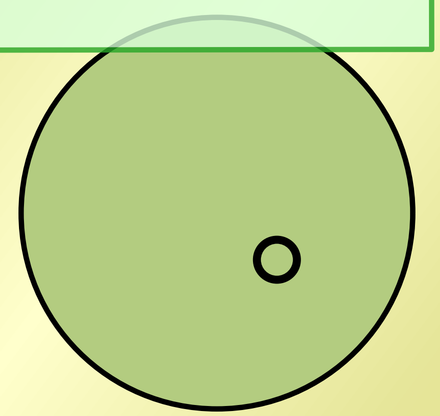
Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.



Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

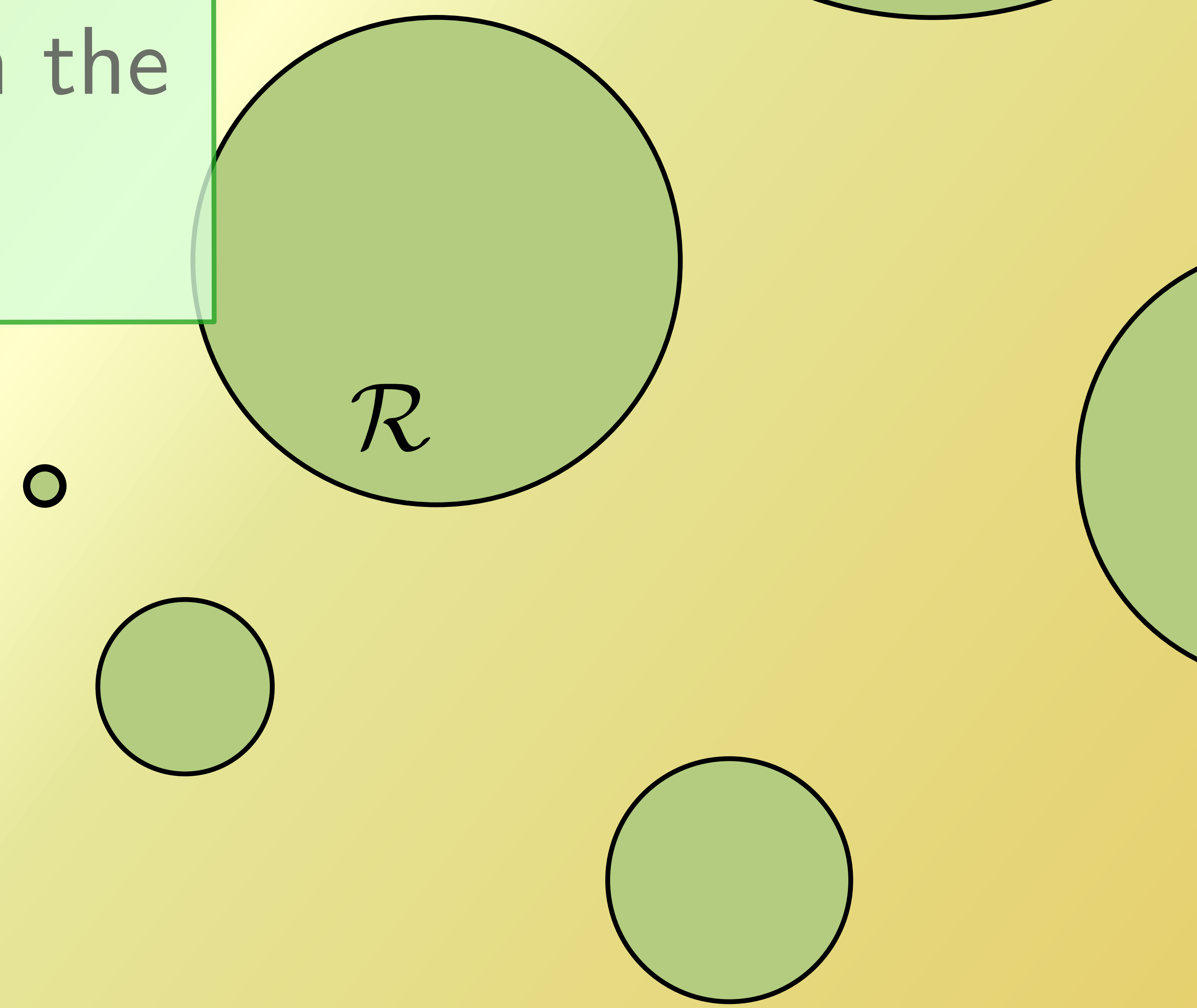


Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.



Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

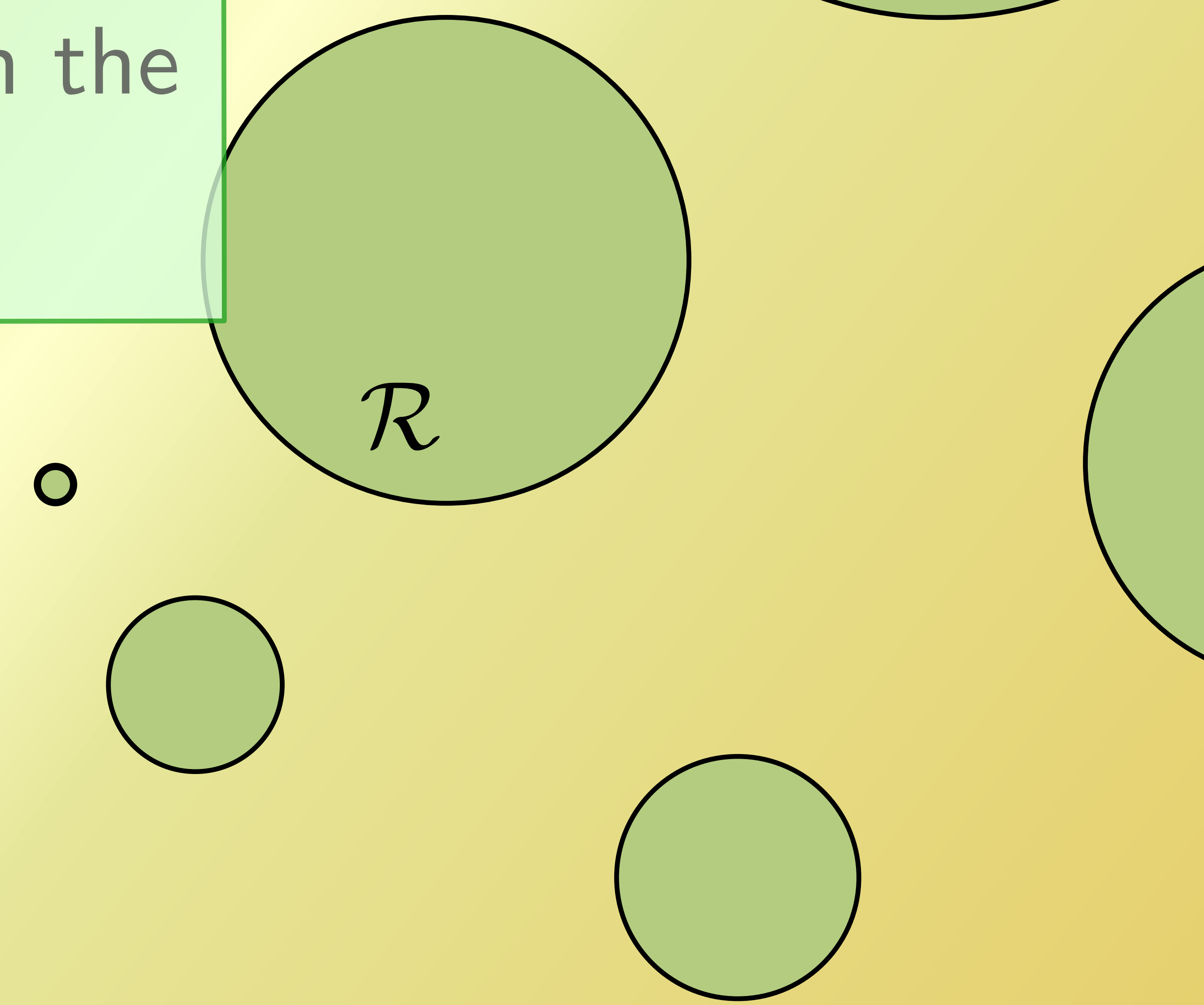
. . . or the true location itself of a point may change.

Suppose furthermore that our points are *dynamic*.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

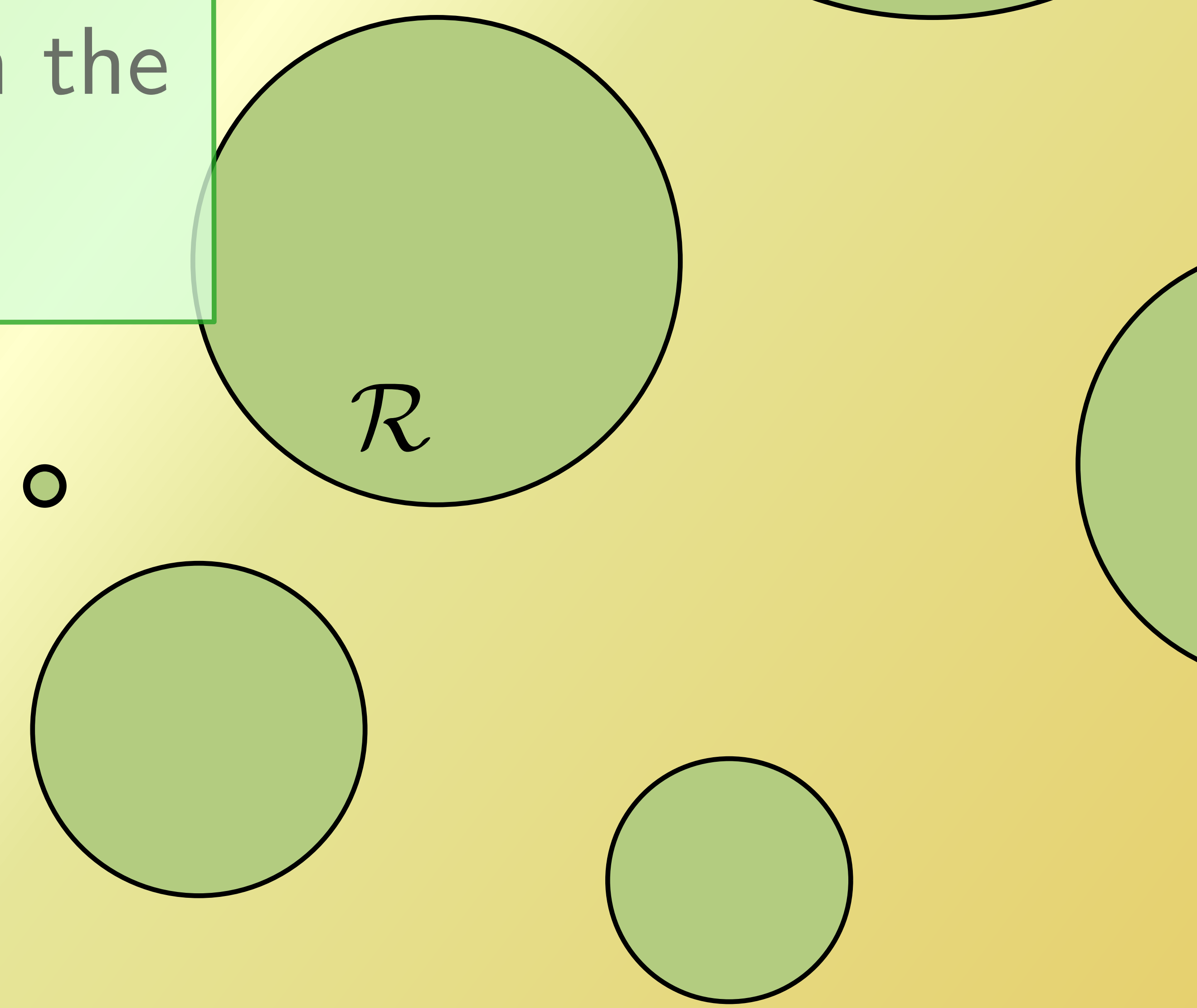


Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.



Suppose furthermore that our points are *dynamic*.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

Let q be a query point.

$\mathcal{R}$

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Let q be a query point.

Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

Suppose furthermore that our points are *dynamic*.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Let q be a query point.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

QUESTION

Is q an element of P?

Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Let q be a query point.

QUESTION

Is q an element of P?

We can still answer the question in logarithmic time after preprocessing

Suppose furthermore that our points are *dynamic*.

Our estimate of a point's location may change. . .

. . . or the true location itself of a point may change.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

Let q be a query point.

q

QUESTION
Is q an element of P?

We can still answer the question in logarithmic time after preprocessing

$\mathcal{R}$

But now we also need to respond to changes in $\mathcal{R}$

Suppose furthermore that our points are *dynamic*.

Let $\mathcal{R}$ be a set of n dynamic regions in the plane.

We can still answer the question in logarithmic time after preprocessing

Our estimate of a point's location may change. . .

q

Let q be a query point.

$\mathcal{R}$

But now we also need to respond to changes in $\mathcal{R}$

. . . or the true location itself of a point may change.

QUESTION
Is q an element of P?

We want to also handle *updates* efficiently.

What is known about dynamic planar point location?

$O(\log^2 n)$ queries with $O(\log n)$ updates.

[Cheng & Janardan, 1992]

What is known about dynamic planar point location?

$O(\log n)$ queries with $O(\log^{1+\varepsilon} n)$ updates.

[Arge *et al.*, 2006]

$O(\log^2 n)$ queries with $O(\log n)$ updates.

[Cheng & Janardan, 1992]

What is known about dynamic planar point location?

$O(\log n)$ queries with $O(\log^{1+\varepsilon} n)$ updates.

[Arge *et al.*, 2006]

In special cases, $O(\log n)$ queries *and* updates is possible ...

$O(\log^2 n)$ queries with $O(\log n)$ updates.

[Cheng & Janardan, 1992]

What is known about dynamic planar point location?

What is known about dynamic planar point location?

$O(\log^2 n)$ queries with $O(\log n)$ updates.

[Cheng & Janardan, 1992]

$O(\log n)$ queries with $O(\log^{1+\varepsilon} n)$ updates.

[Arge *et al.*, 2006]

In special cases, $O(\log n)$ queries *and* updates is possible . . .

. . . such as in *monotone* subdivisions . . .

[Goodrich & Tamassia, 1998]

$O(\log n)$ queries with $O(\log^{1+\varepsilon} n)$ updates.

[Arge *et al.*, 2006]

In special cases, $O(\log n)$ queries *and* updates is possible . . .

. . . such as in *monotone* subdivisions . . .

[Goodrich & Tamassia, 1998]

$O(\log^2 n)$ queries with $O(\log n)$ updates.

[Cheng & Janardan, 1992]

What is known about dynamic planar point location?

. . . or in *rectilinear* subdivisions.

[Blelloch, 2008]
[Giora & Kaplan, 2009]

Updates always take at least logarithmic time.

Updates always take at least logarithmic time.

The bottleneck is often point location itself.

Updates always take at least logarithmic time.

The bottleneck is often point location itself.

However, in our application, updates are *local*.

Updates always take at least logarithmic time.

The bottleneck is often point location itself.

However, in our application, updates are *local*.

QUESTION

Is is possible to break the $\log n$ barrier in this case?

PROBLEM STATEMENT & RESULTS

PROBLEM

Maintain n regions in the plane such that . . .

PROBLEM

Maintain n regions in the plane such that . . .

PROBLEM

Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

PROBLEM

Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

PROBLEM

Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

. . . we can *delete* any region in $\log n$ time . . .

PROBLEM
Maintain n regions in the plane such that …

… we can *insert* a new region of any size in $\log n$ time …

… we can *delete* any region in $\log n$ time …

PROBLEM
Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

. . . we can *delete* any region in $\log n$ time . . .

. . . we can locally alter, or *update*, a region in less than $\log n$ time . . .

PROBLEM
Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

. . . we can *delete* any region in $\log n$ time . . .

. . . we can locally alter, or *update*, a region in less than $\log n$ time . . .

PROBLEM

Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

. . . we can *delete* any region in $\log n$ time . . .

. . . we can locally alter, or *update*, a region in less than $\log n$ time . . .

. . . and we can answer point location queries in $\log n$ time

PROBLEM
Maintain n regions in the plane such that . . .

. . . we can *insert* a new region of any size in $\log n$ time . . .

. . . we can *delete* any region in $\log n$ time . . .

. . . we can locally alter, or *update*, a region in less than $\log n$ time . . .

. . . and we can answer point location queries in $\log n$ time

Of course, this is not always possible.

Of course, this is not always possible.

When is an update *local*?

Of course, this is not always possible.

When is an update *local*?

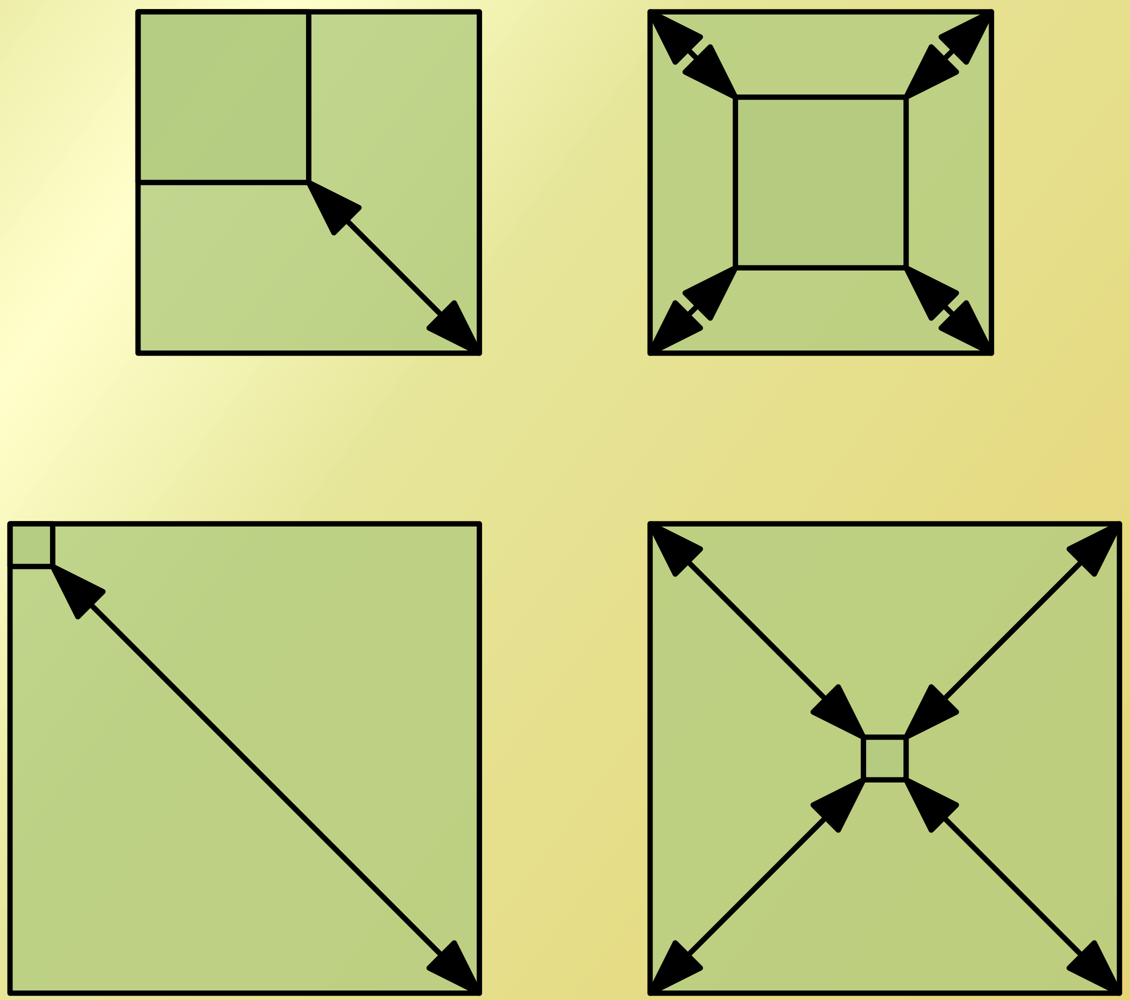
Regions can grow or shrink by at most a constant factor.

When is an update *local*?

Regions can grow or shrink by at most a constant factor.

Of course, this is not always possible.

GOOD



BAD

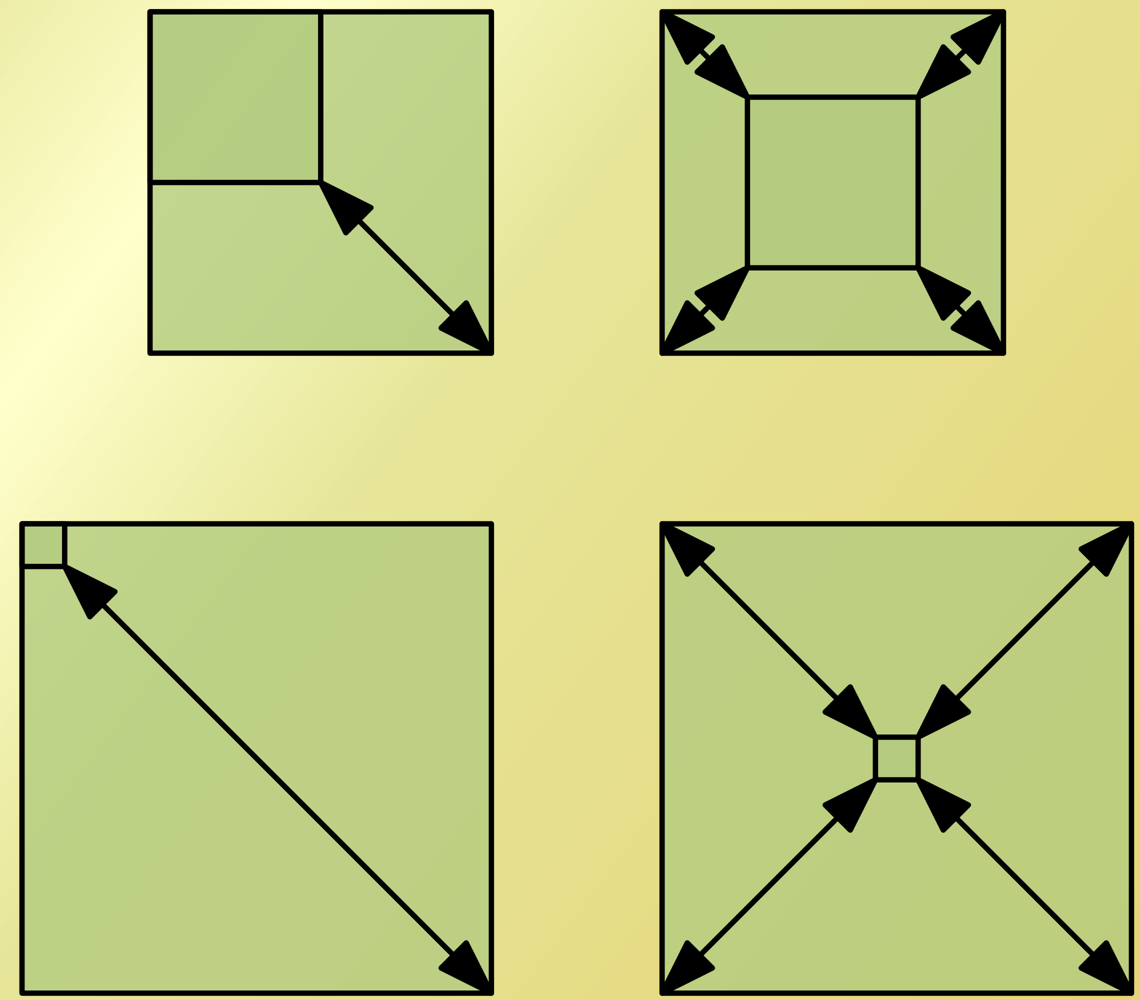
Of course, this is not always possible.

When is an update *local*?

Regions can grow or shrink by at most a constant factor.

Regions can move a constant times their current size.

GOOD



BAD

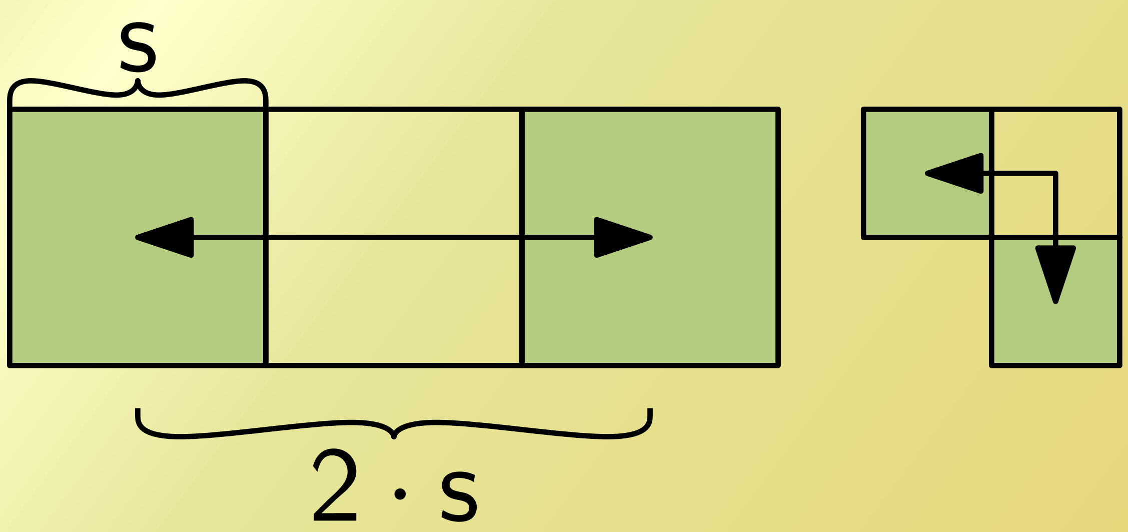
When is an update *local*?

Of course, this is not always possible.

Regions can grow or shrink by at most a constant factor.

Regions can move a constant times their current size.

GOOD



BAD

When is an update *local*?

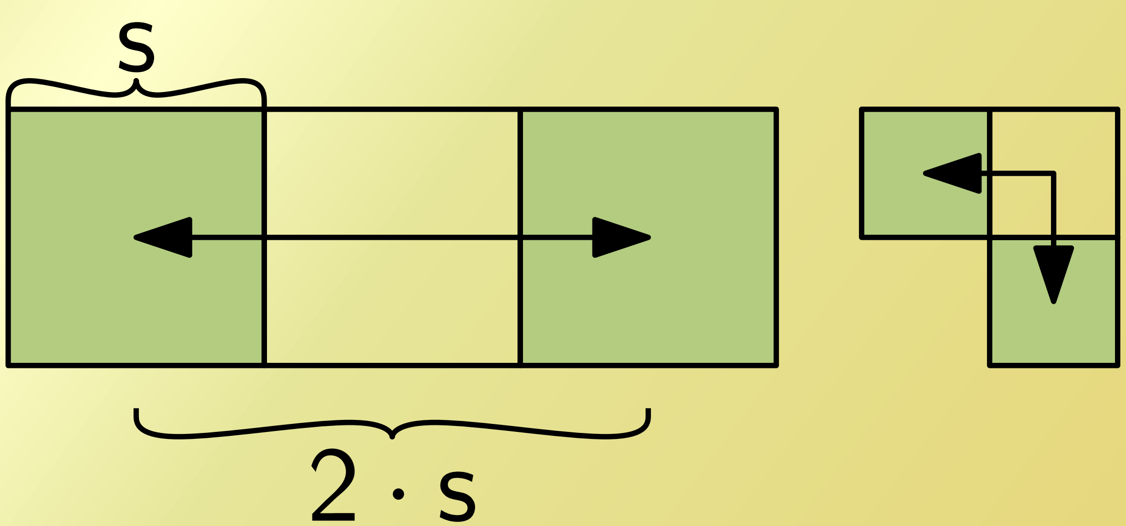
Regions can grow or shrink by at most a constant factor.

Regions can move a constant times their current size.

Regions can change their shape, as long as they stay *fat*.

Of course, this is not always possible.

GOOD



BAD

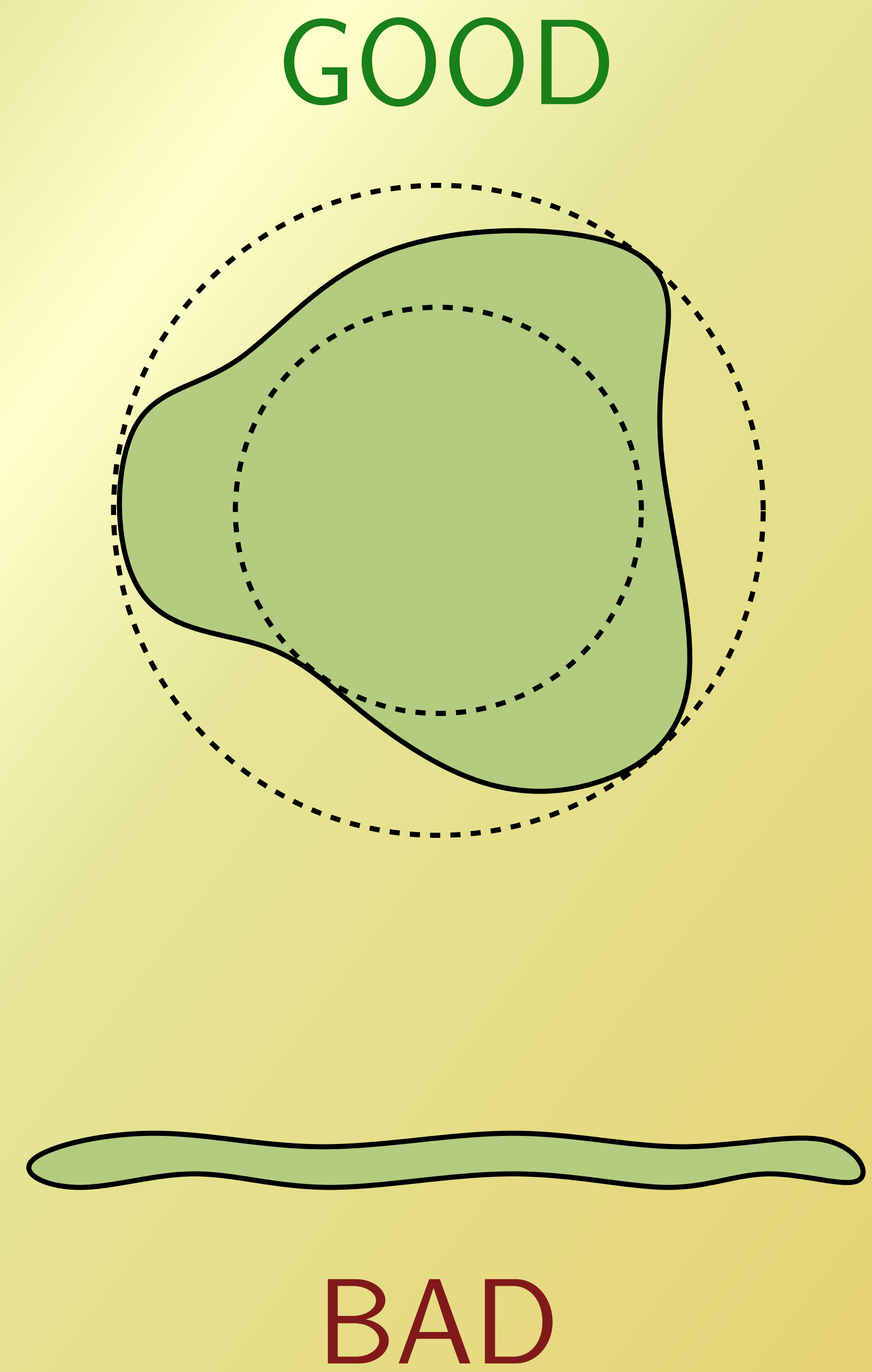
When is an update *local*?

Regions can grow or shrink by at most a constant factor.

Regions can move a constant times their current size.

Regions can change their shape, as long as they stay *fat*.

Of course, this is not always possible.



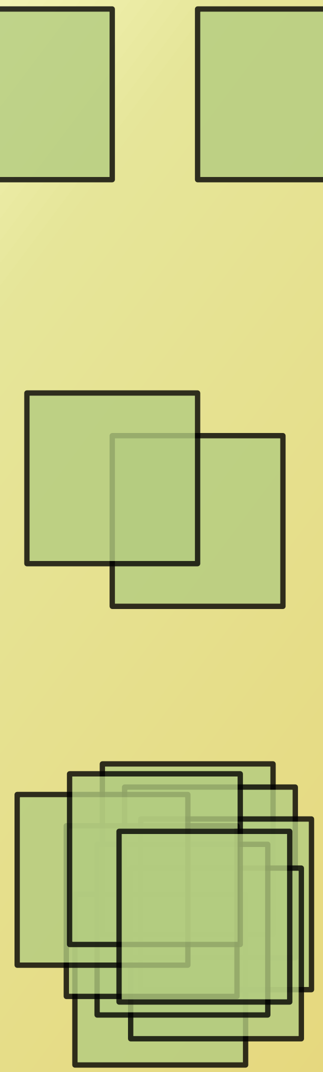
One more assumption: the regions are and stay disjoint!

One more assumption: the regions are and stay disjoint!

GOOD

BAD

UGLY



And the results are
. . .

And the results are . . .

1D:

And the results are . . .

1D: Queries: $O(\log n)$ time

And the results are . . .

1D:	Queries:	$O(\log n)$ time
	Insertions and deletions:	$O(\log n)$ time

And the results are . . .

1D:

Queries:	$O(\log n)$ time
Insertions and deletions:	$O(\log n)$ time
Local updates:	$O(1)$ time

And the results are . . .

1D:

Queries:	$O(\log n)$ time
Insertions and deletions:	$O(\log n)$ time
Local updates:	$O(1)$ time

2D:

And the results are . . .

1D:	Queries:	$O(\log n)$ time
	Insertions and deletions:	$O(\log n)$ time
	Local updates:	$O(1)$ time
2D:	Queries:	$O(\log n)$ time

And the results are . . .

1D:

Queries:	$O(\log n)$ time
Insertions and deletions:	$O(\log n)$ time
Local updates:	$O(1)$ time

2D:

Queries:	$O(\log n)$ time
Insertions and deletions:	$O(\log n)$ time

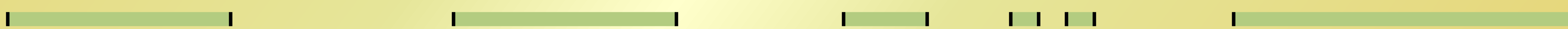
And the results are . . .

1D:	Queries:	$O(\log n)$ time
	Insertions and deletions:	$O(\log n)$ time
	Local updates:	$O(1)$ time
2D:	Queries:	$O(\log n)$ time
	Insertions and deletions:	$O(\log n)$ time
	Local updates:	$O(\log n / \log \log n)$ time

TECHNICAL DETAILS:
1 DIMENSION

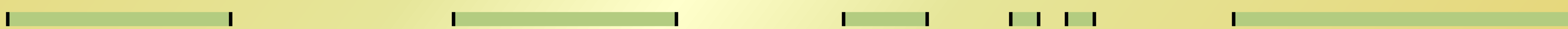
1-dimensional regions are intervals.

1-dimensional regions are intervals.



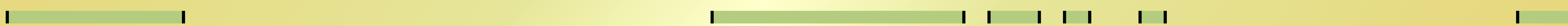
1-dimensional regions are intervals.

They move around on a line: big intervals are fast, small ones are slow.



1-dimensional regions are intervals.

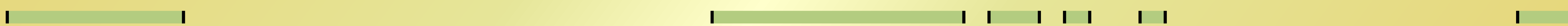
They move around on a line: big intervals are fast, small ones are slow.



1-dimensional regions are intervals.

They move around on a line: big intervals are fast, small ones are slow.

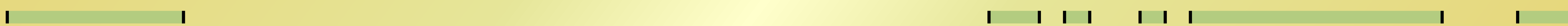
NOTE
Big intervals can *jump over* small ones!



1-dimensional regions are intervals.

They move around on a line: big intervals are fast, small ones are slow.

NOTE
Big intervals can *jump over* small ones!



We need a structure that provides quick access to “similar places”. . .

We need a structure that provides quick access to "similar places"...

...but also supports some sort of binary search.

We need a structure that provides quick access to "similar places" . . .

. . . but also supports some sort of binary search.

IDEA Let's maintain two trees.

We need a structure that provides quick access to “similar places”. . .

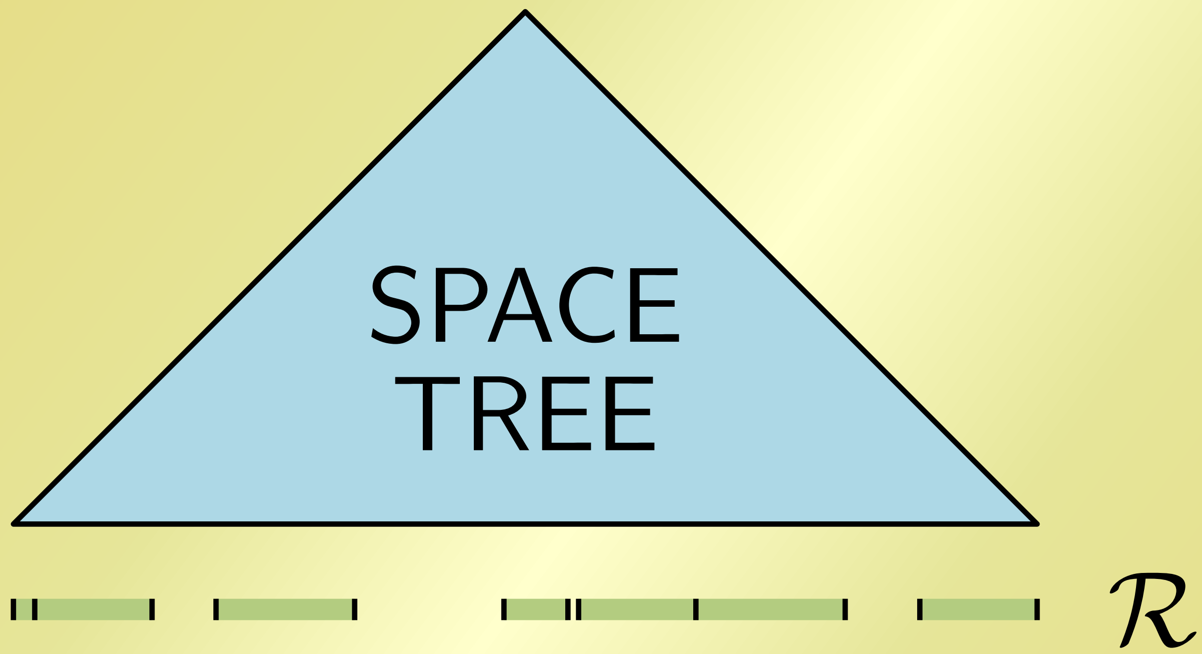
. . . but also supports some sort of binary search.

IDEA Let’s maintain two trees.

We need a structure that provides quick access to "similar places" . . .

. . . but also supports some sort of binary search.

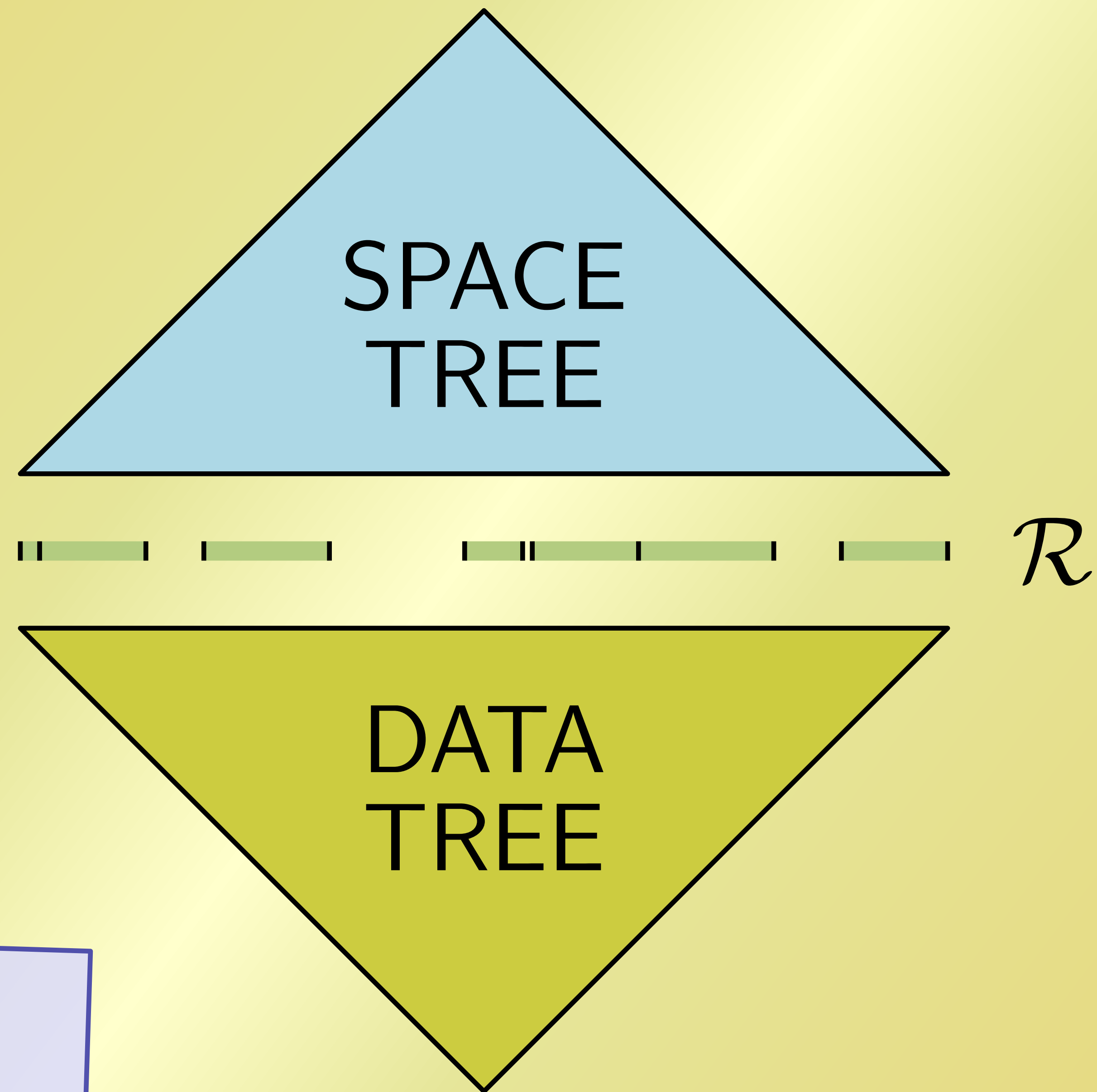
IDEA Let's maintain two trees.



We need a structure that provides quick access to "similar places" . . .

. . . but also supports some sort of binary search.

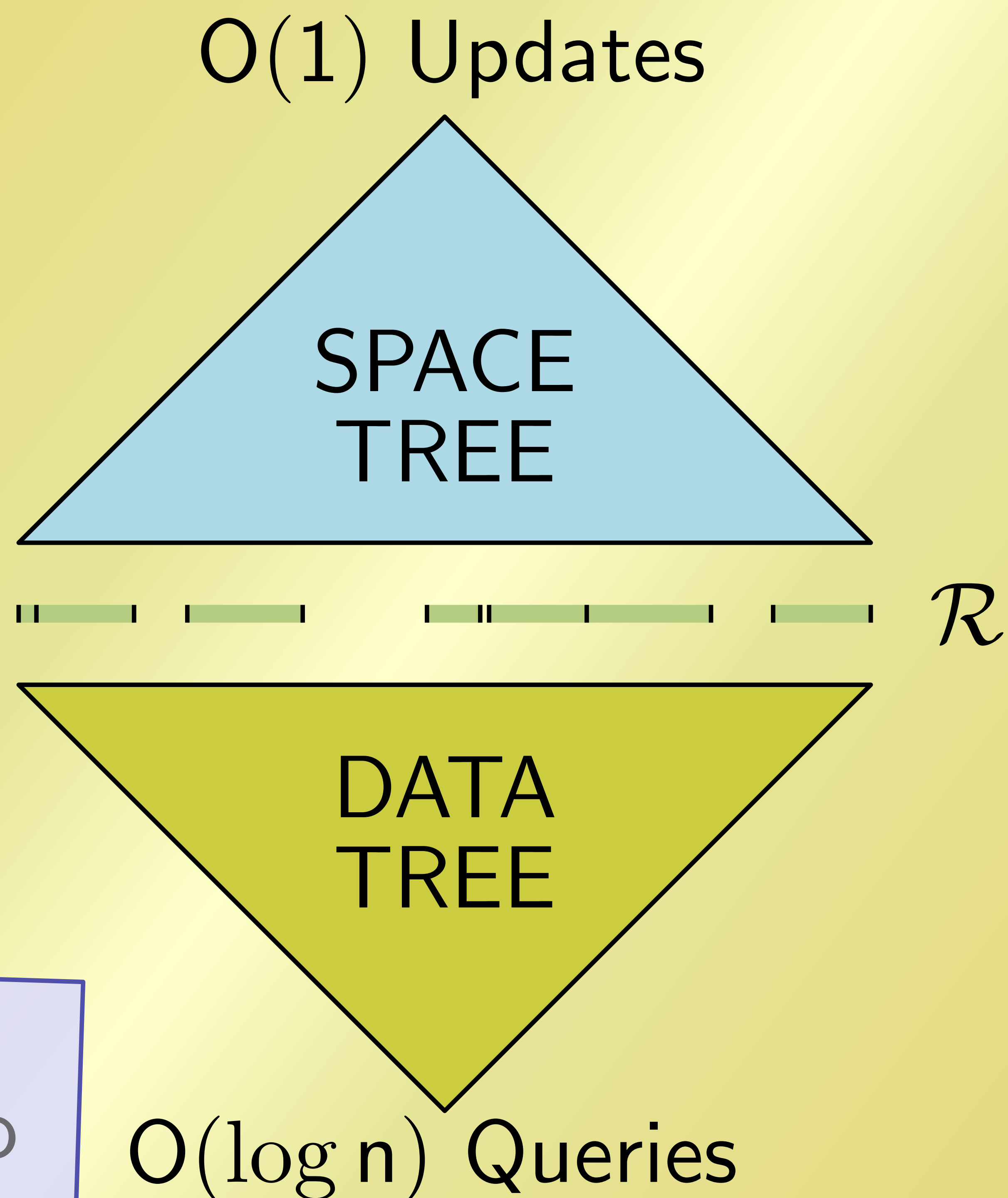
IDEA Let's maintain two trees.



We need a structure that provides quick access to "similar places"...

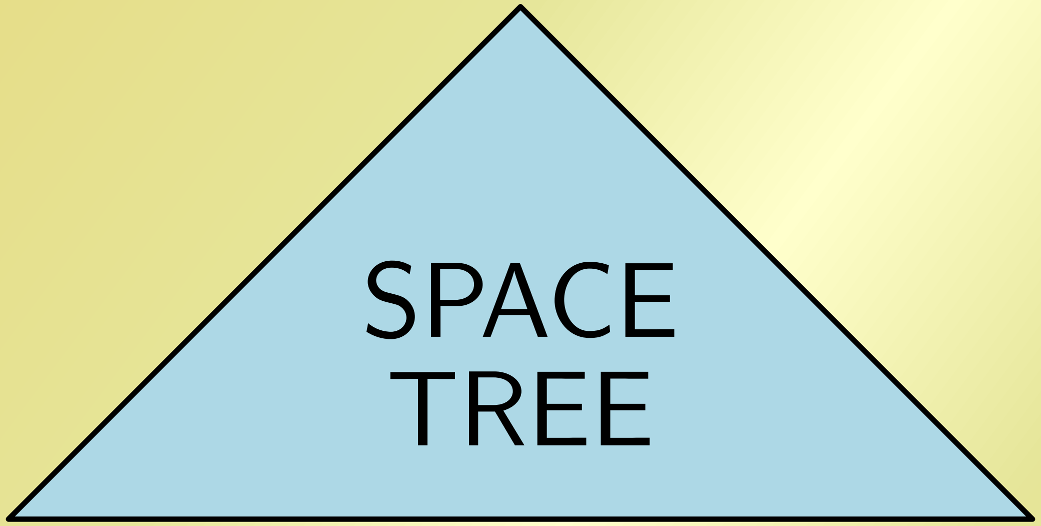
...but also supports some sort of binary search.

IDEA Let's maintain two trees.

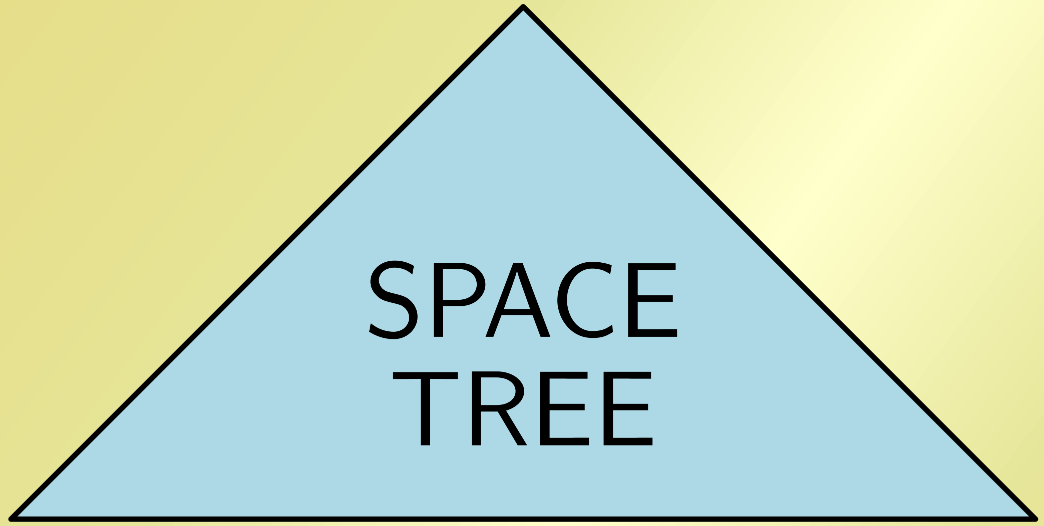


SPACE TREE

For the space tree we use a *quadtree*.

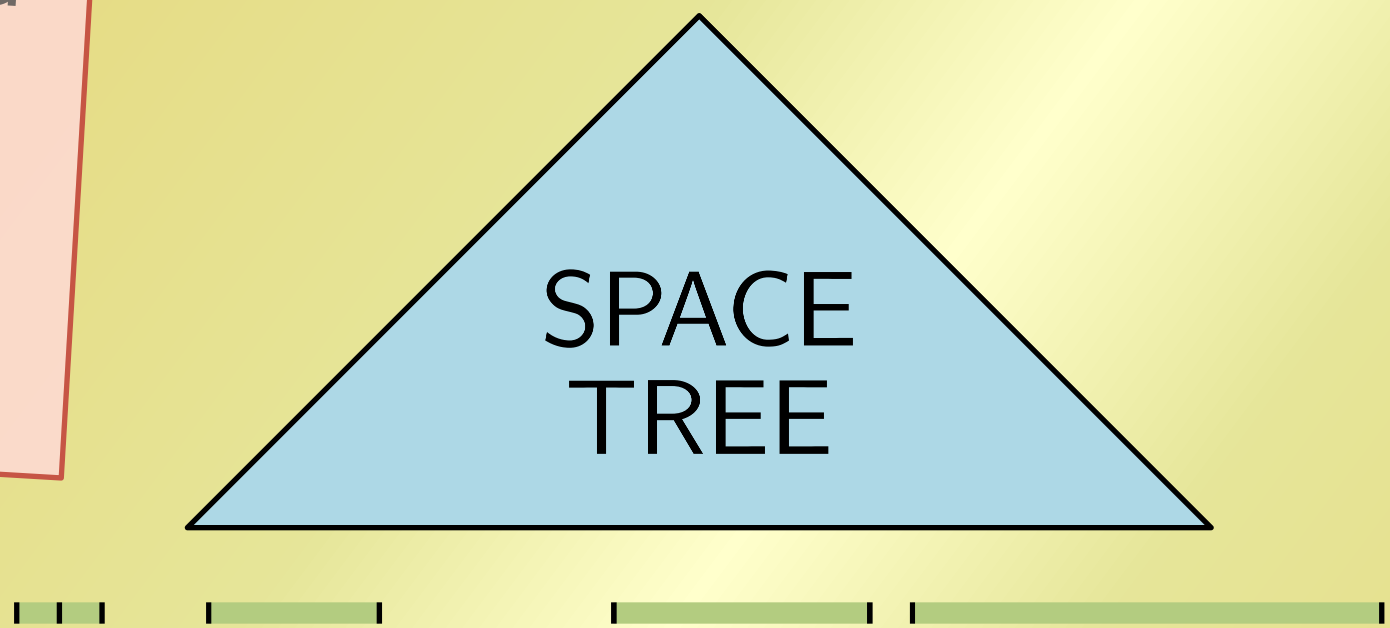


For the space tree we use a *quadtree*.



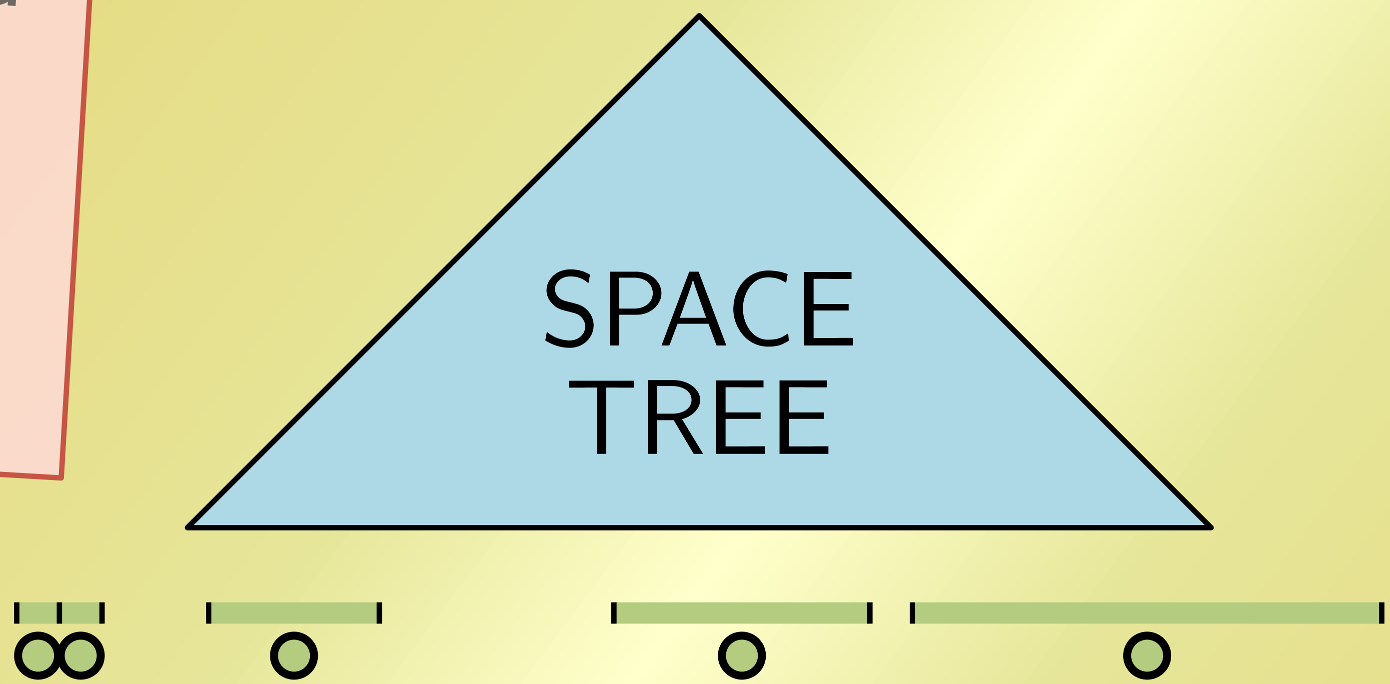
Consider the set P of midpoints of the intervals.

For the space tree we use a *quadtree*.



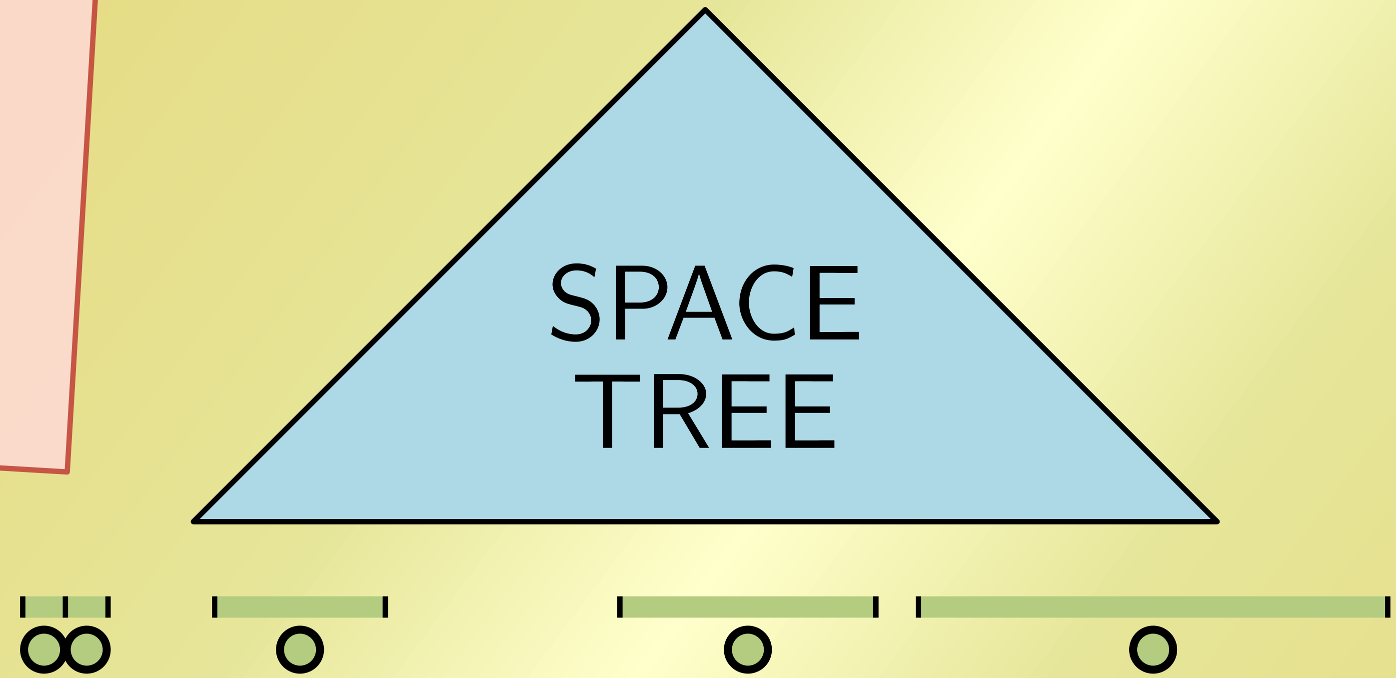
Consider the set P of midpoints of the intervals.

For the space tree we use a *quadtree*.



Consider the set P of midpoints of the intervals.

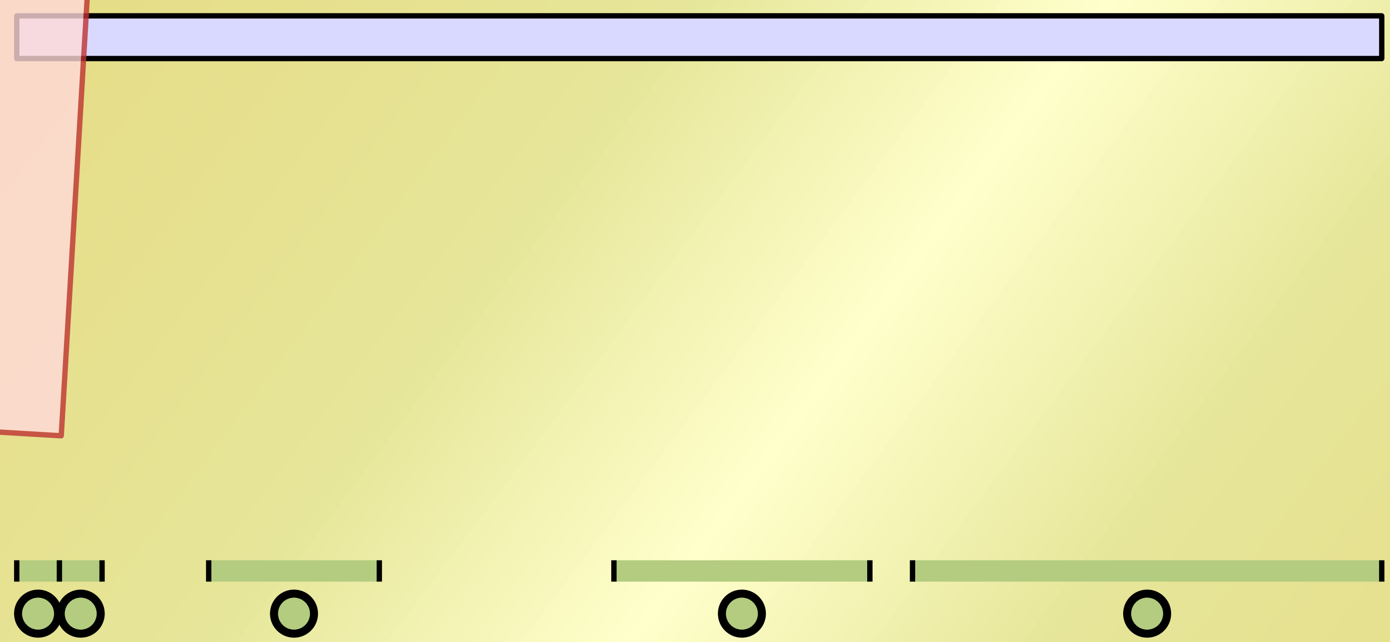
For the space tree we use a *quadtree*.



Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

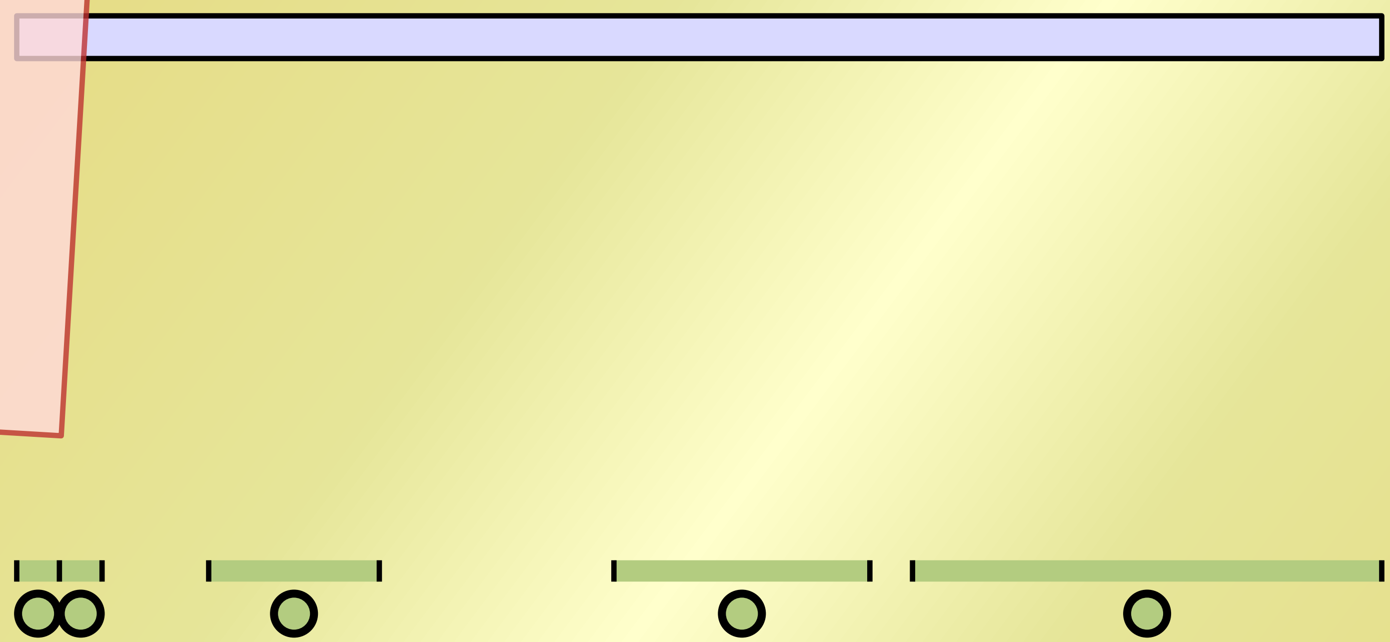
For the space tree we use a *quadtree*.



Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

For the space tree we use a *quadtree*.

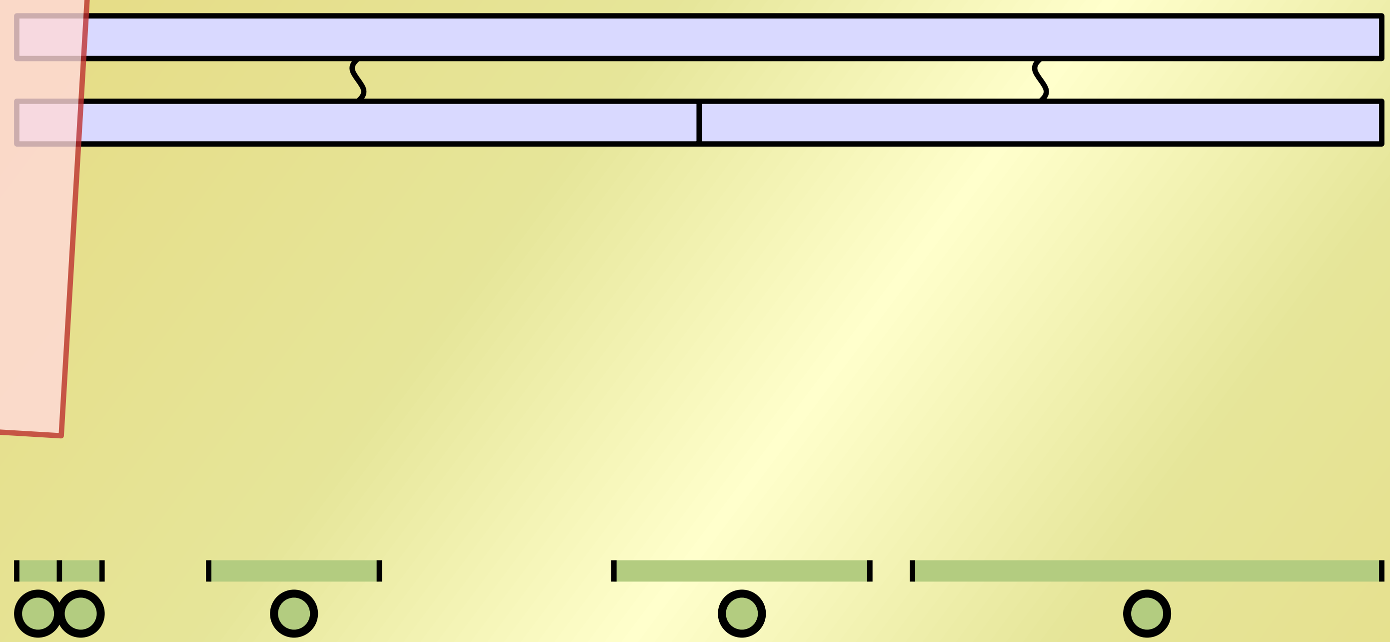


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.

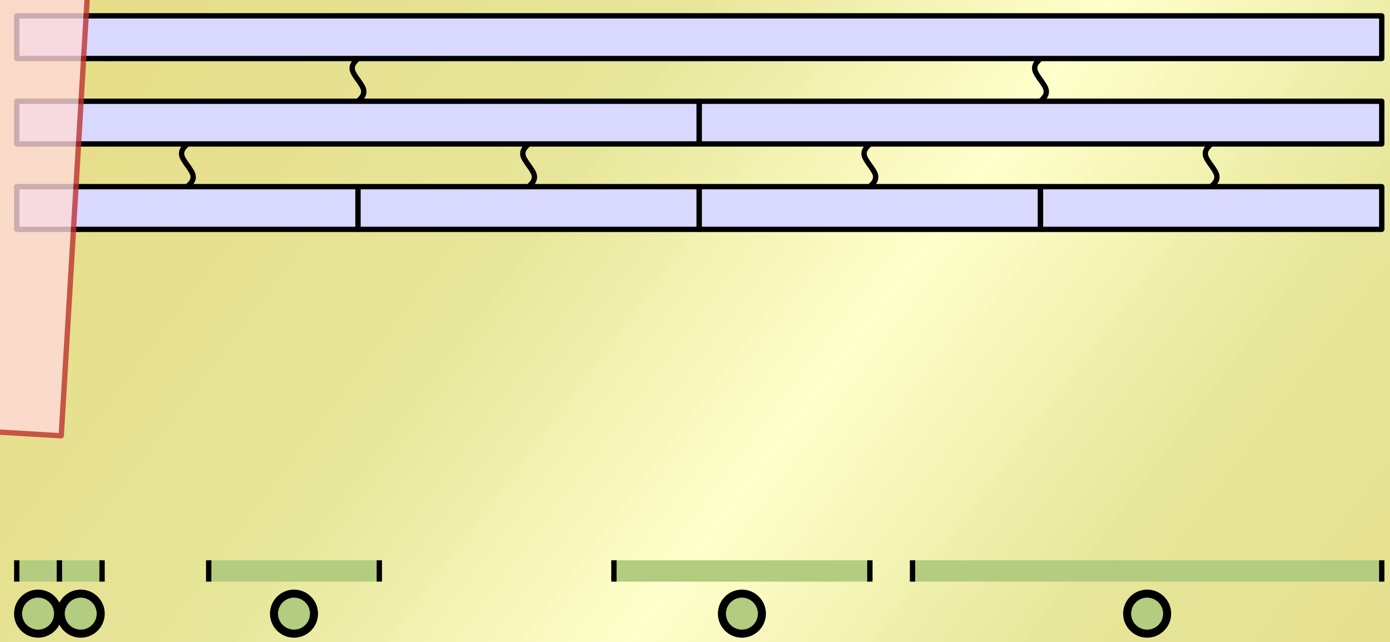


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.

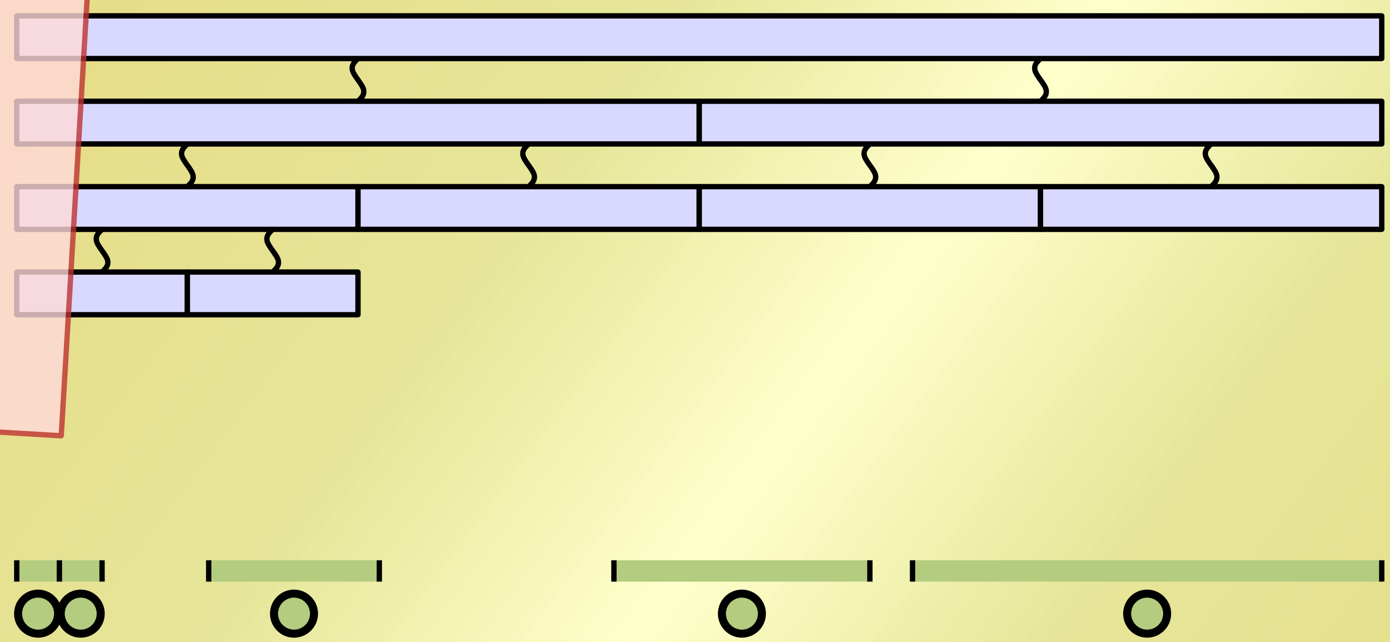


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.

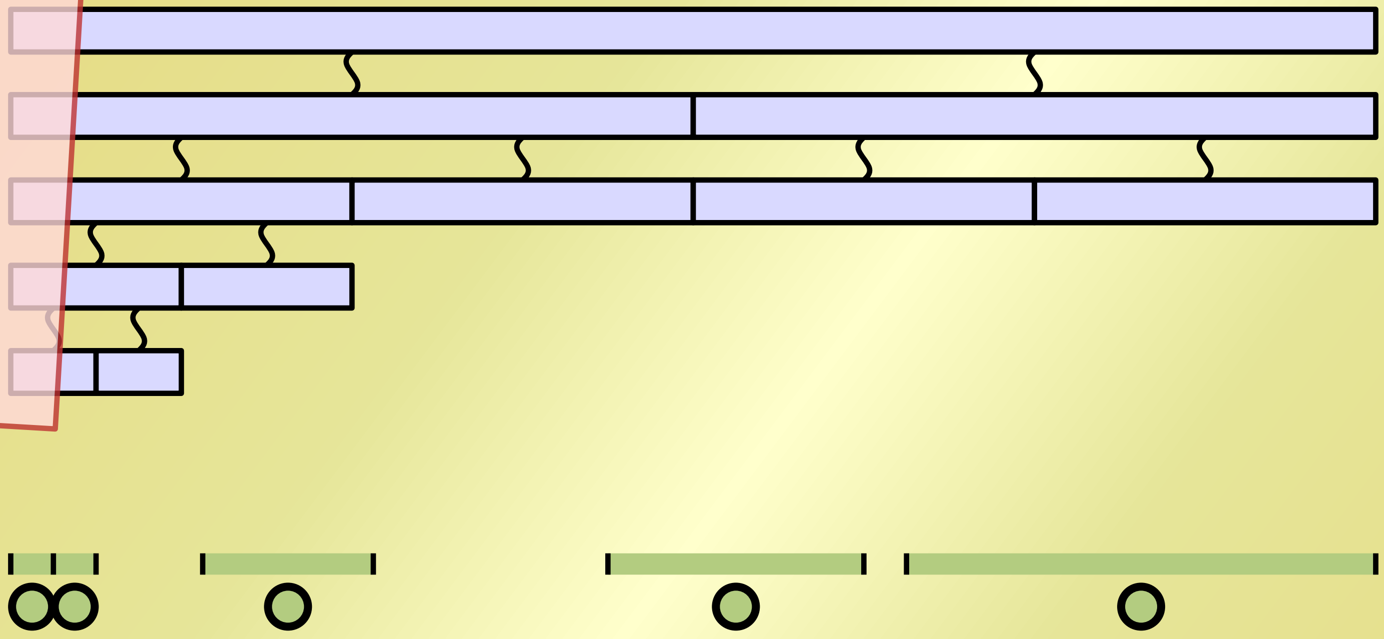


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.

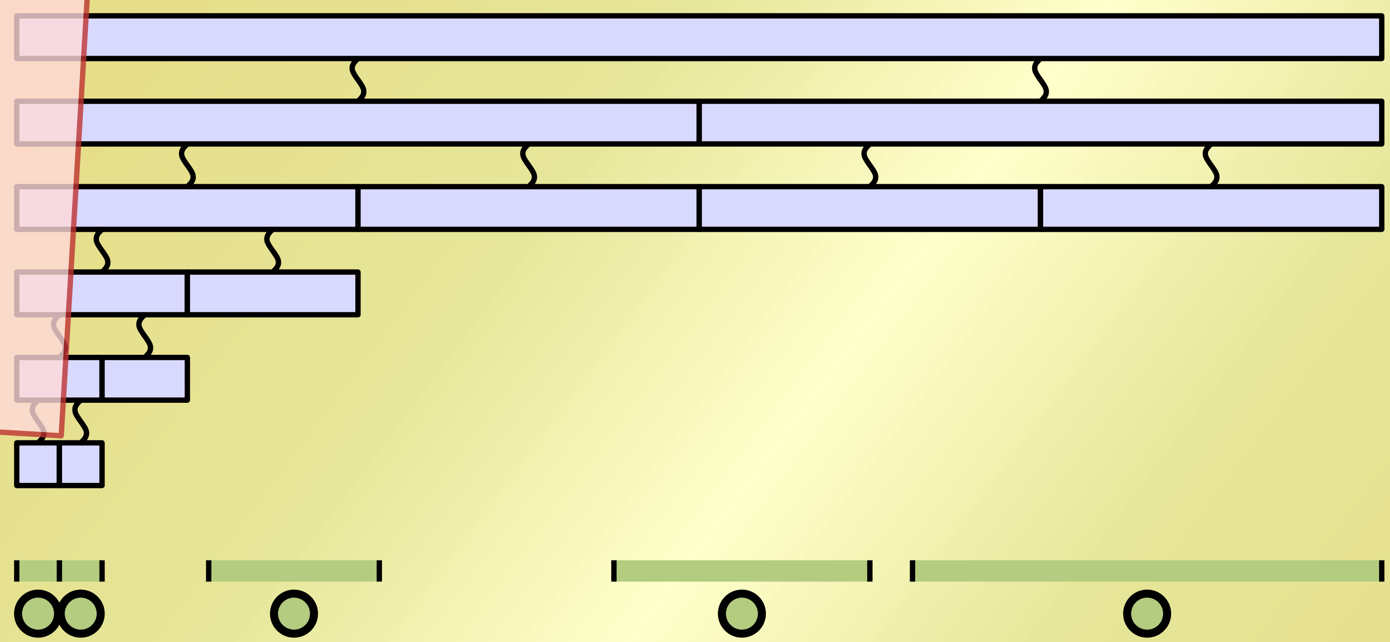


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.

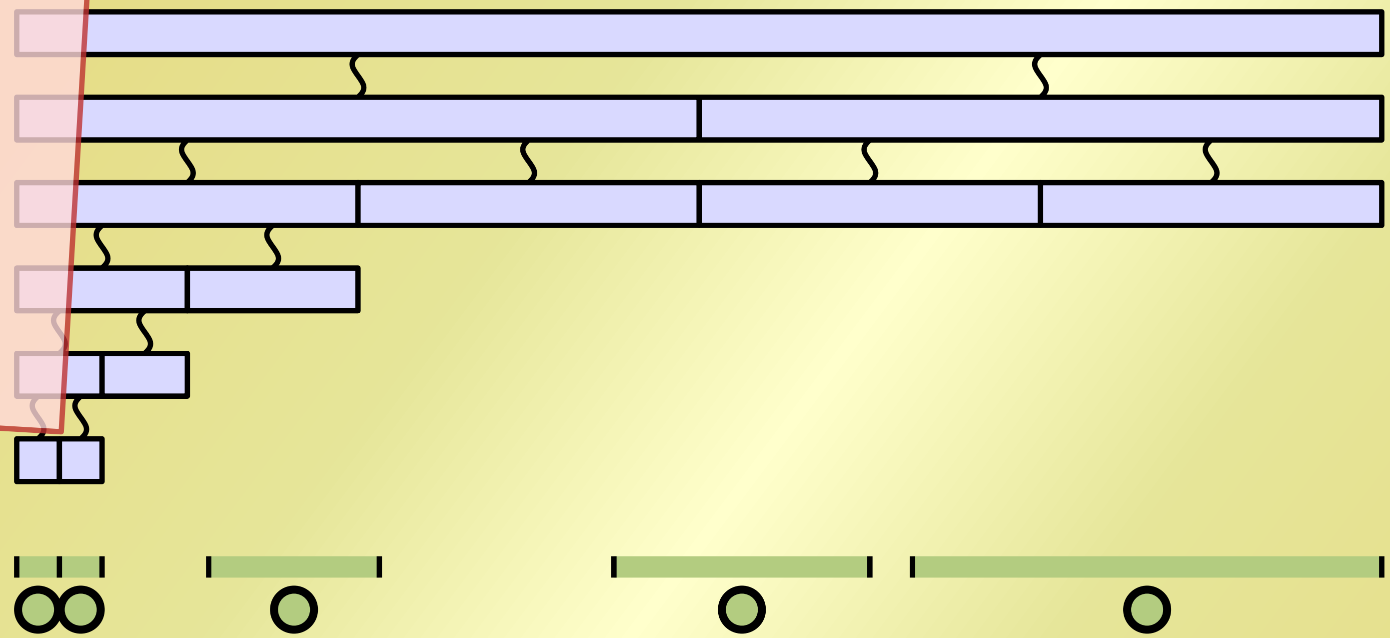


Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

For the space tree we use a *quadtree*.



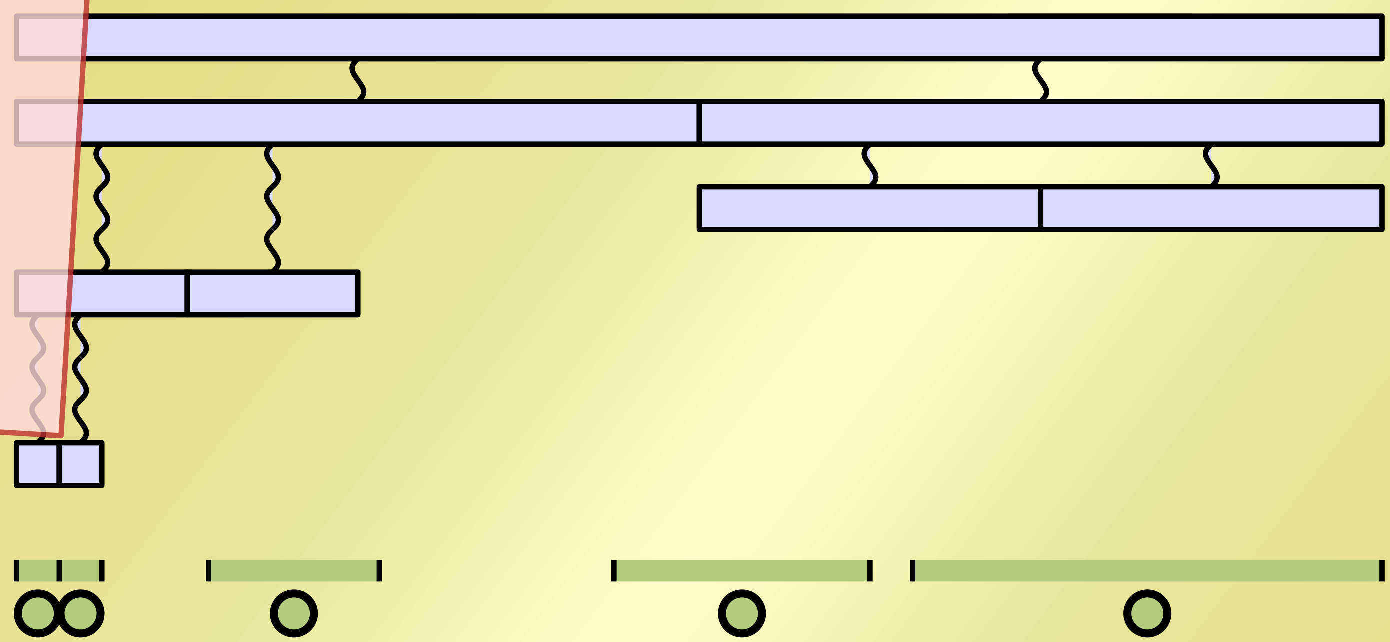
Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

Compress the tree by deleting long empty paths.

For the space tree we use a *quadtree*.



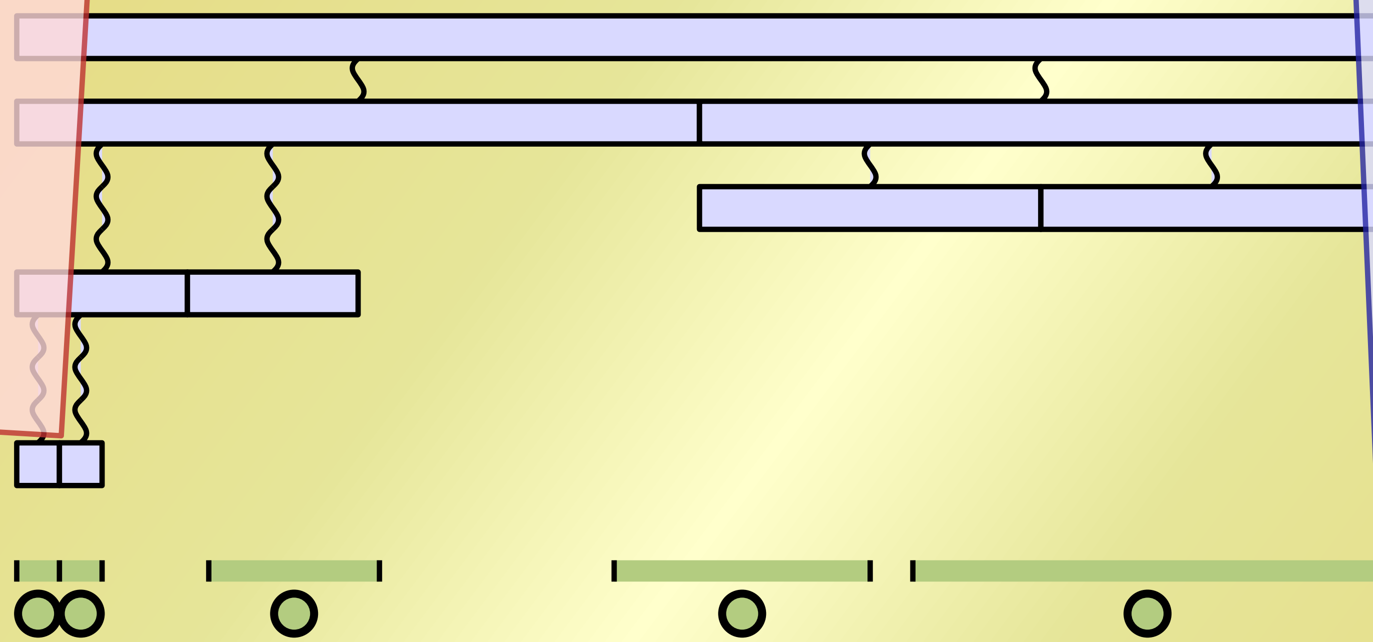
Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

Compress the tree by deleting long empty paths.

For the space tree we use a *quadtree*.



Finally, add pointers between neighbouring boxes of the same size.

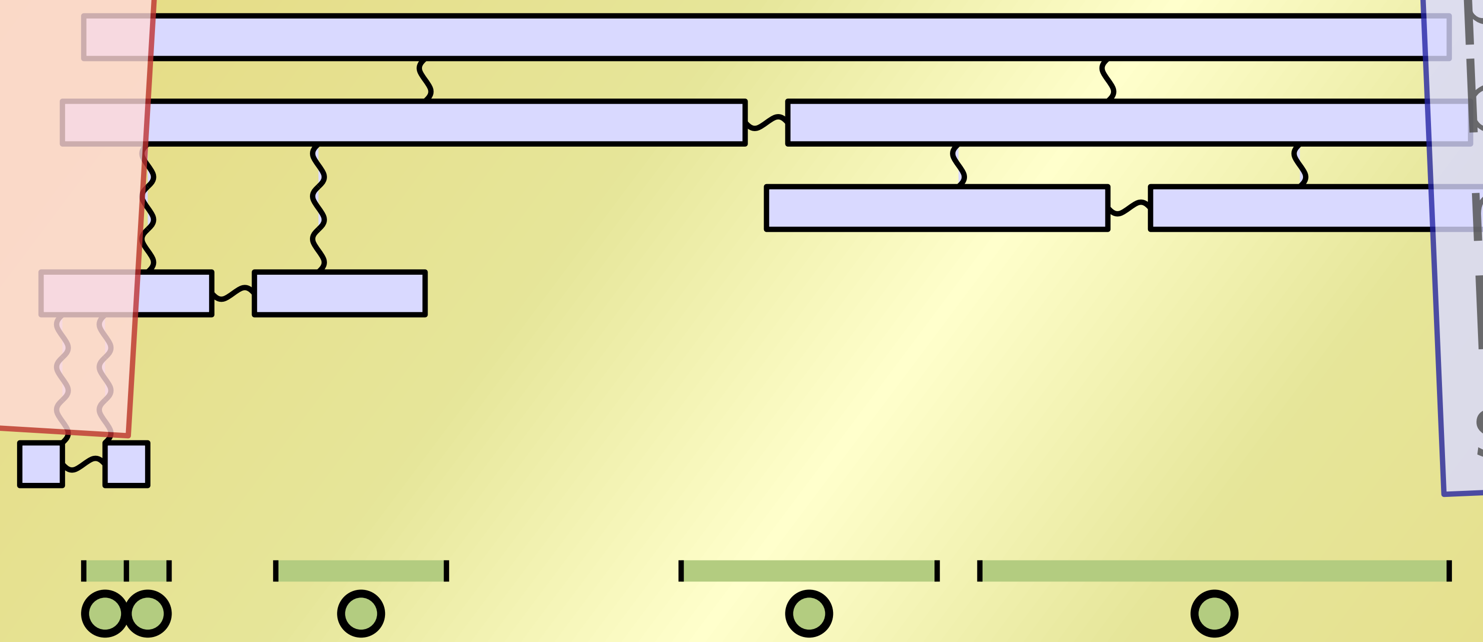
Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

Compress the tree by deleting long empty paths.

For the space tree we use a *quadtree*.



Finally, add pointers between neighbouring boxes of the same size.

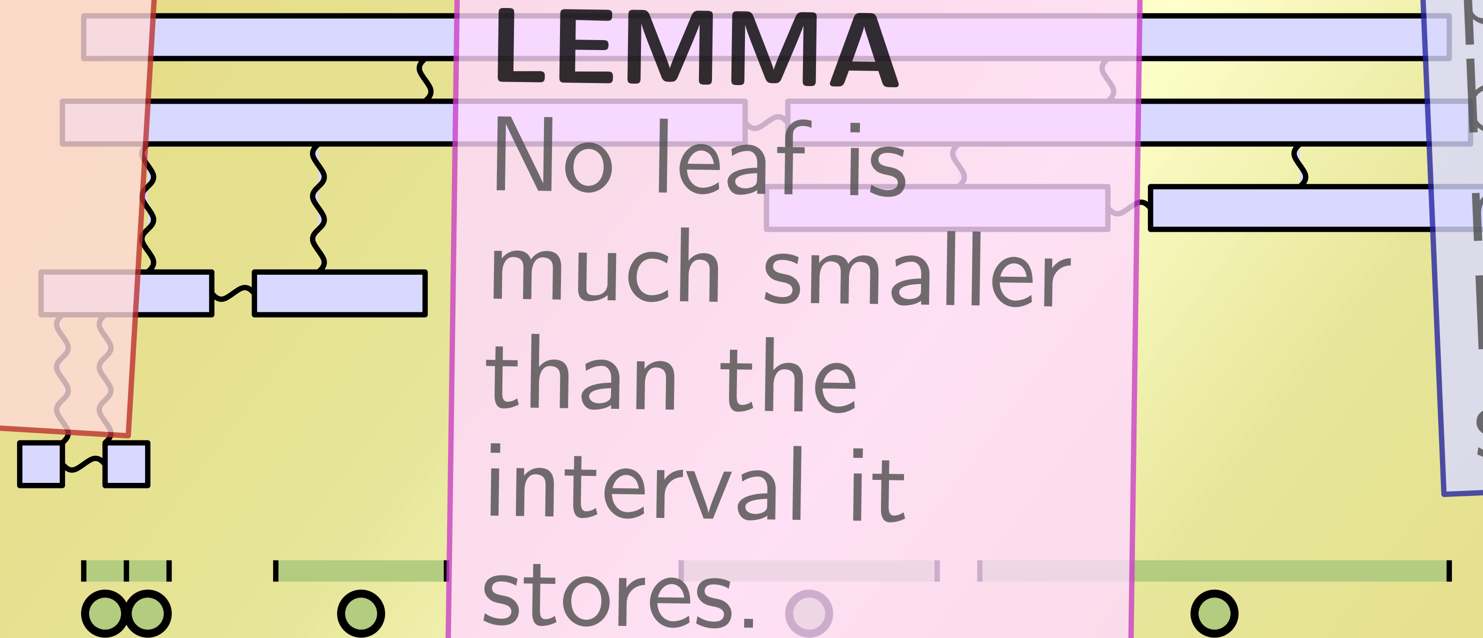
Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

Compress the tree by deleting long empty paths.

For the space tree we use a *quadtree*.



LEMMA
No leaf is much smaller than the interval it stores.

Finally, add pointers between neighbouring boxes of the same size.

Consider the set P of midpoints of the intervals.

Construct a *root* box containing all points of P.

Recursively split boxes that contain at least 2 points.

Compress the tree by deleting long empty paths.

DATA TREE

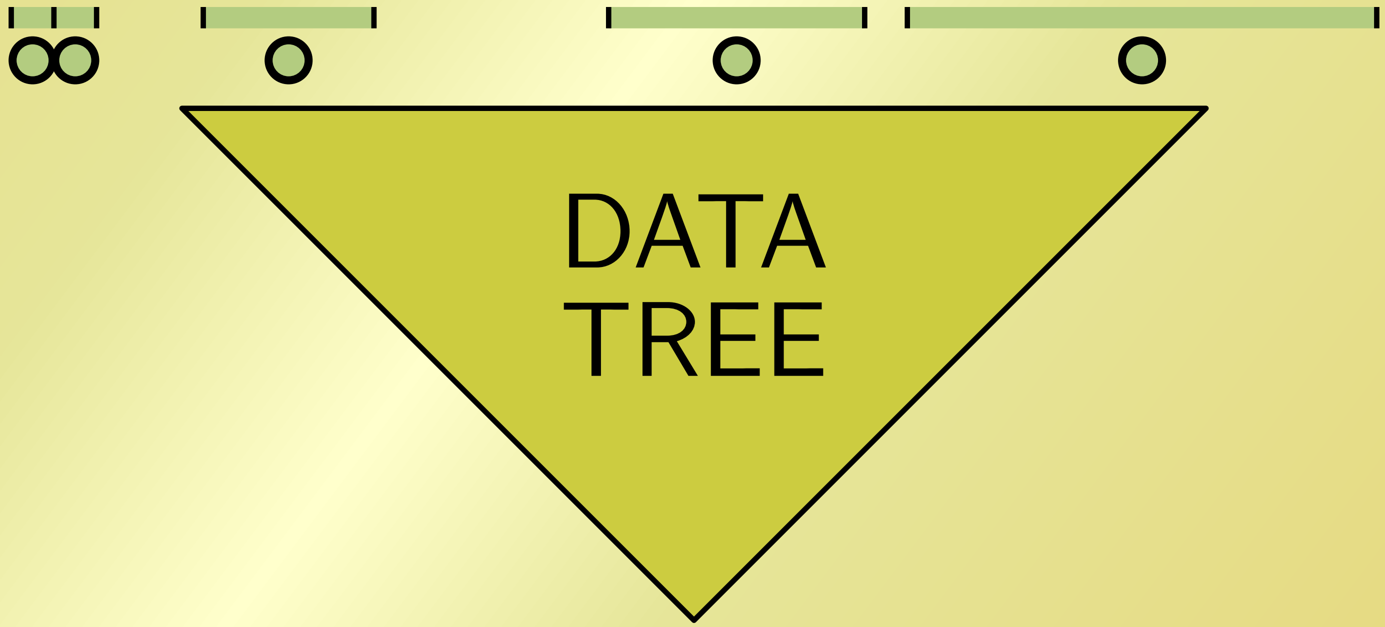
For the data tree we use a dynamic search tree.

For the data tree we use a dynamic search tree.

Again, consider the midpoints of the intervals.

For the data tree we use a dynamic search tree.

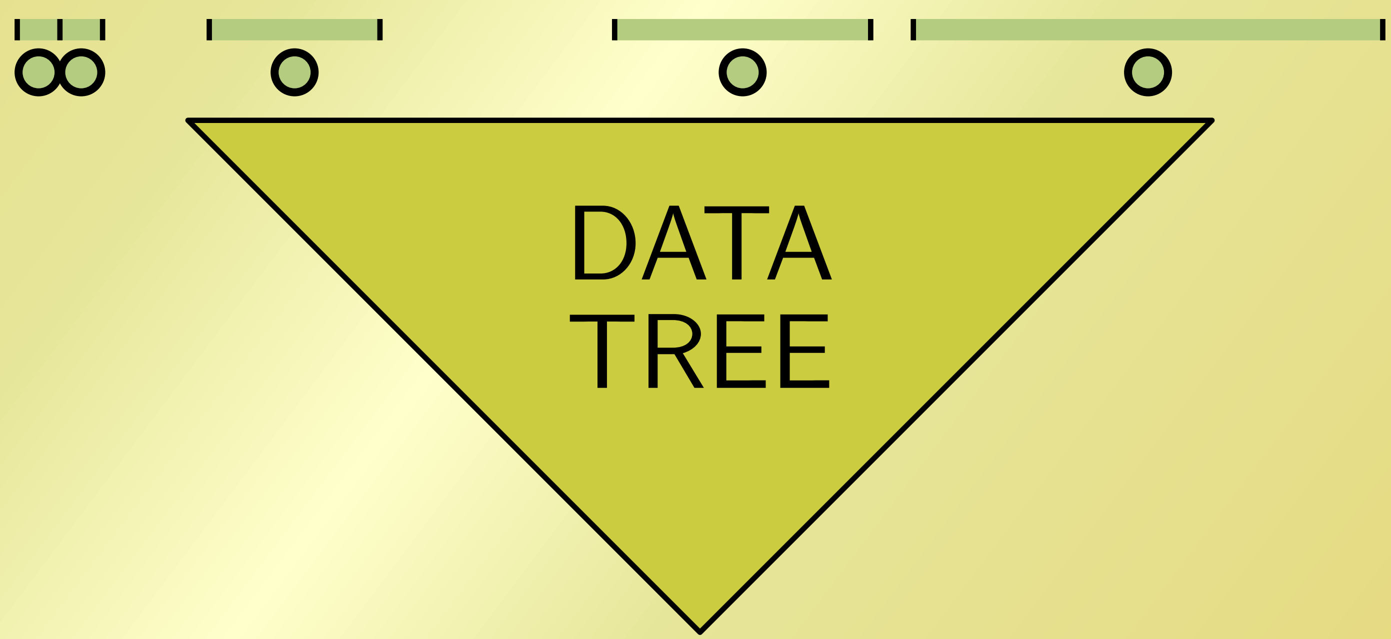
Again, consider the midpoints of the intervals.



For the data tree we use a dynamic search tree.

Again, consider the midpoints of the intervals.

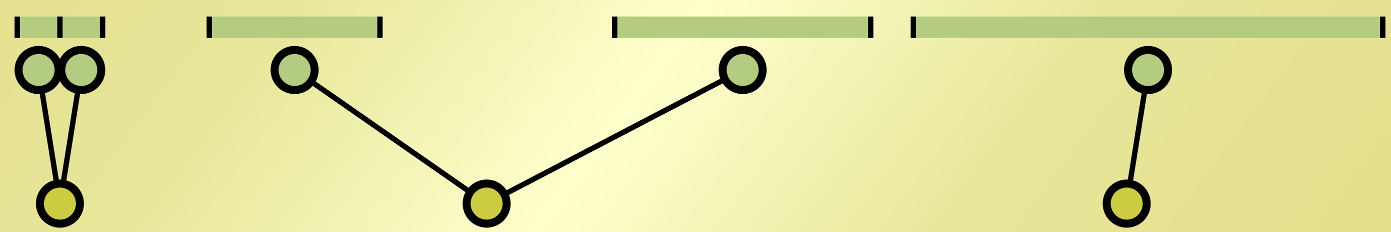
We now only care about their order, and build a *balanced* tree.



For the data tree we use a dynamic search tree.

Again, consider the midpoints of the intervals.

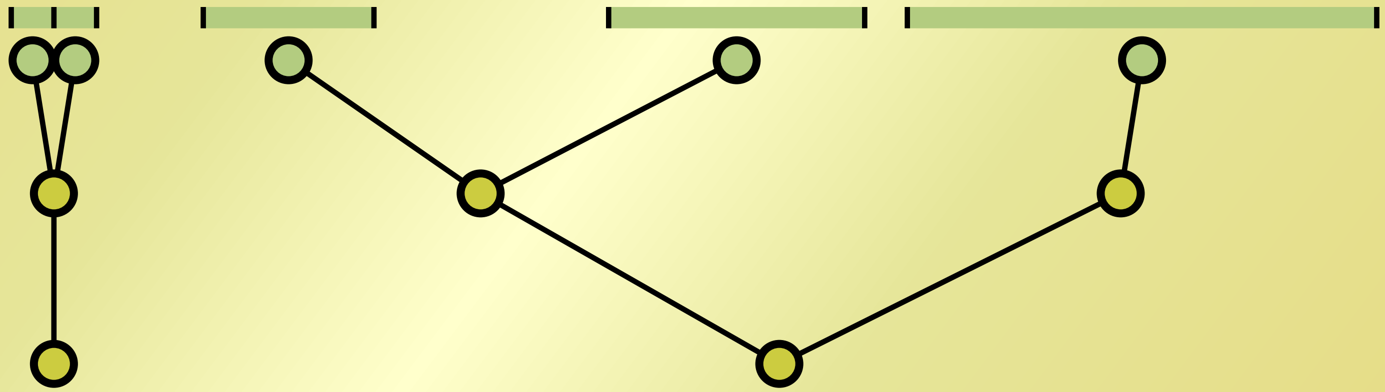
We now only care about their order, and build a *balanced* tree.



For the data tree we use a dynamic search tree.

Again, consider the midpoints of the intervals.

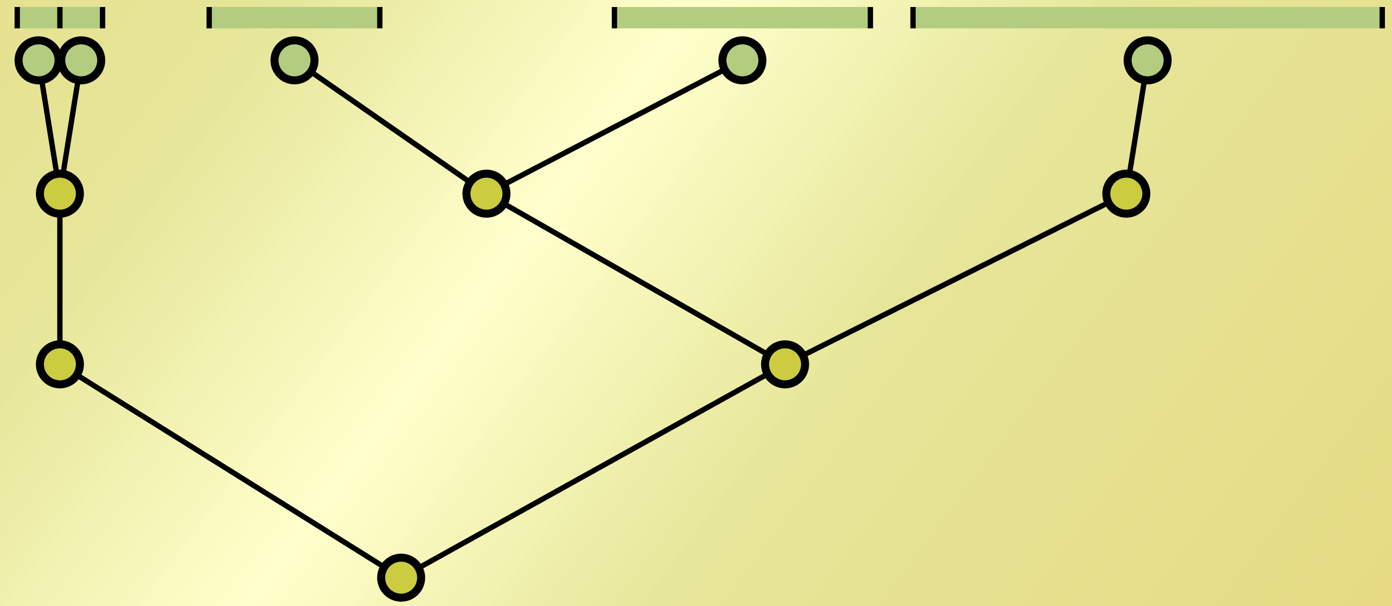
We now only care about their order, and build a *balanced* tree.



For the data tree we use a dynamic search tree.

Again, consider the midpoints of the intervals.

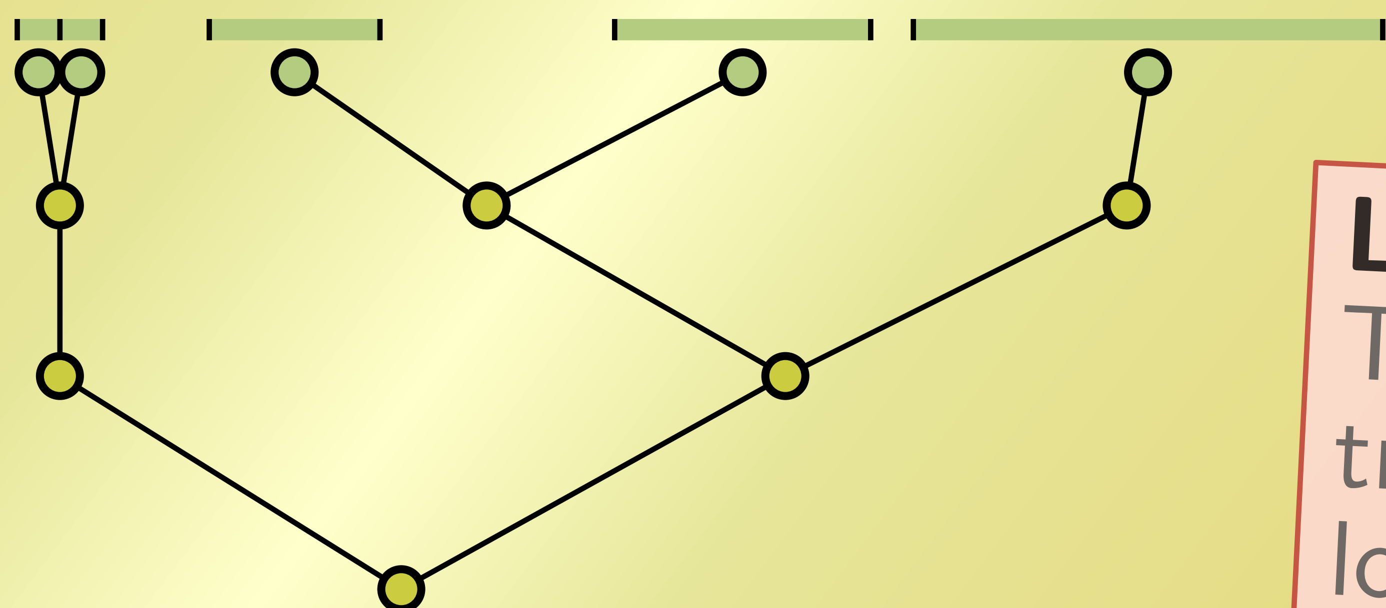
We now only care about their order, and build a *balanced* tree.



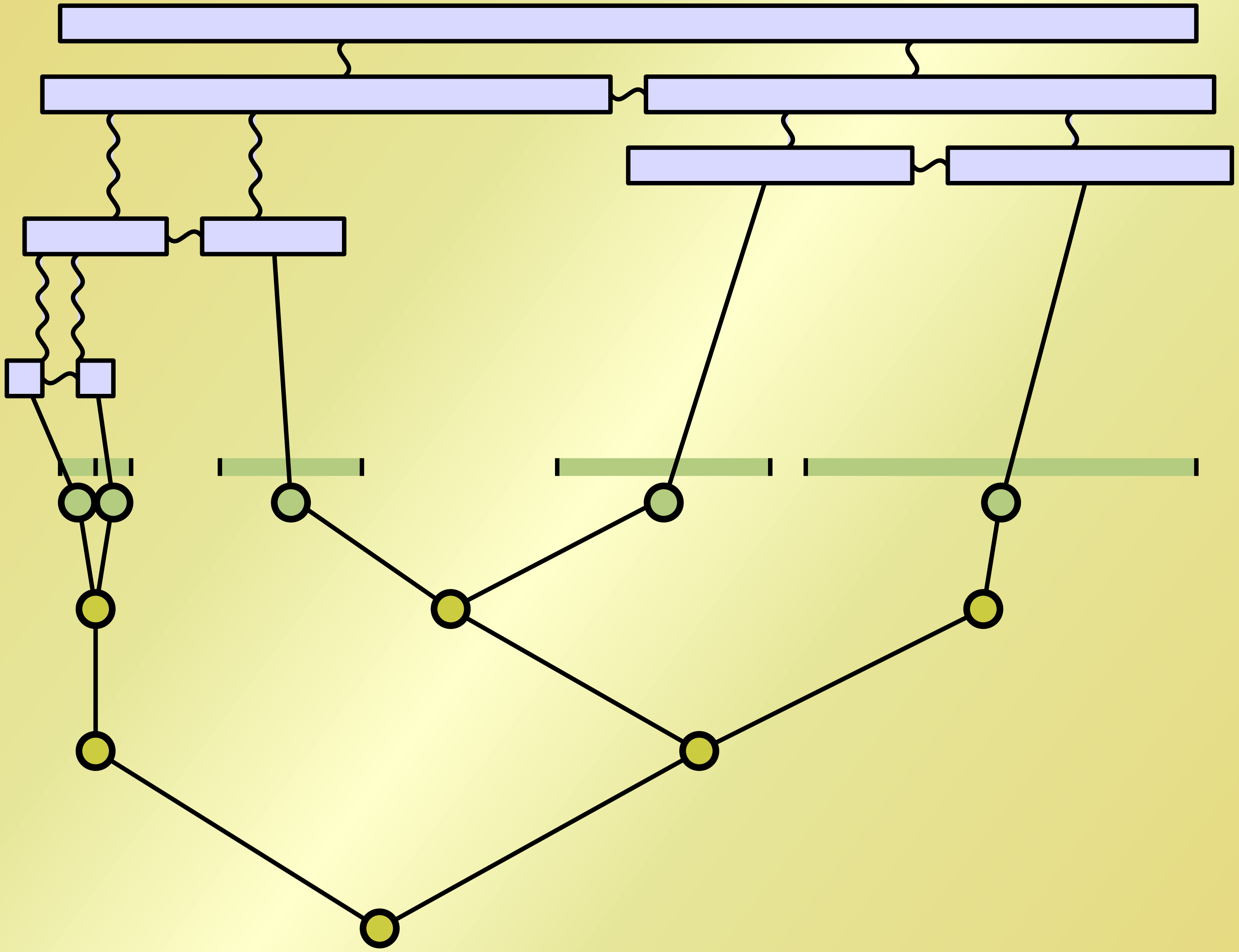
For the data tree we use a dynamic search tree.

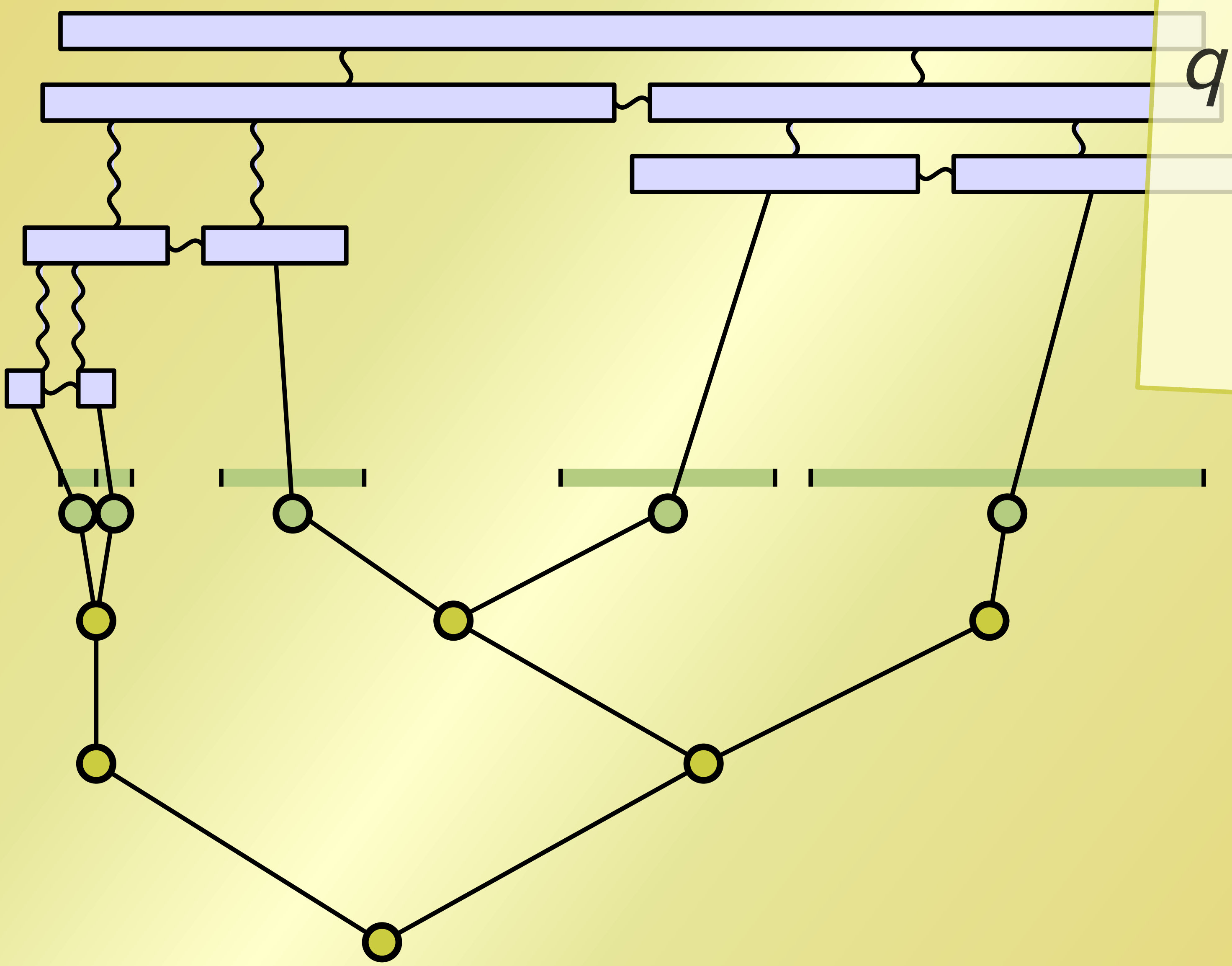
Again, consider the midpoints of the intervals.

We now only care about their order, and build a *balanced* tree.

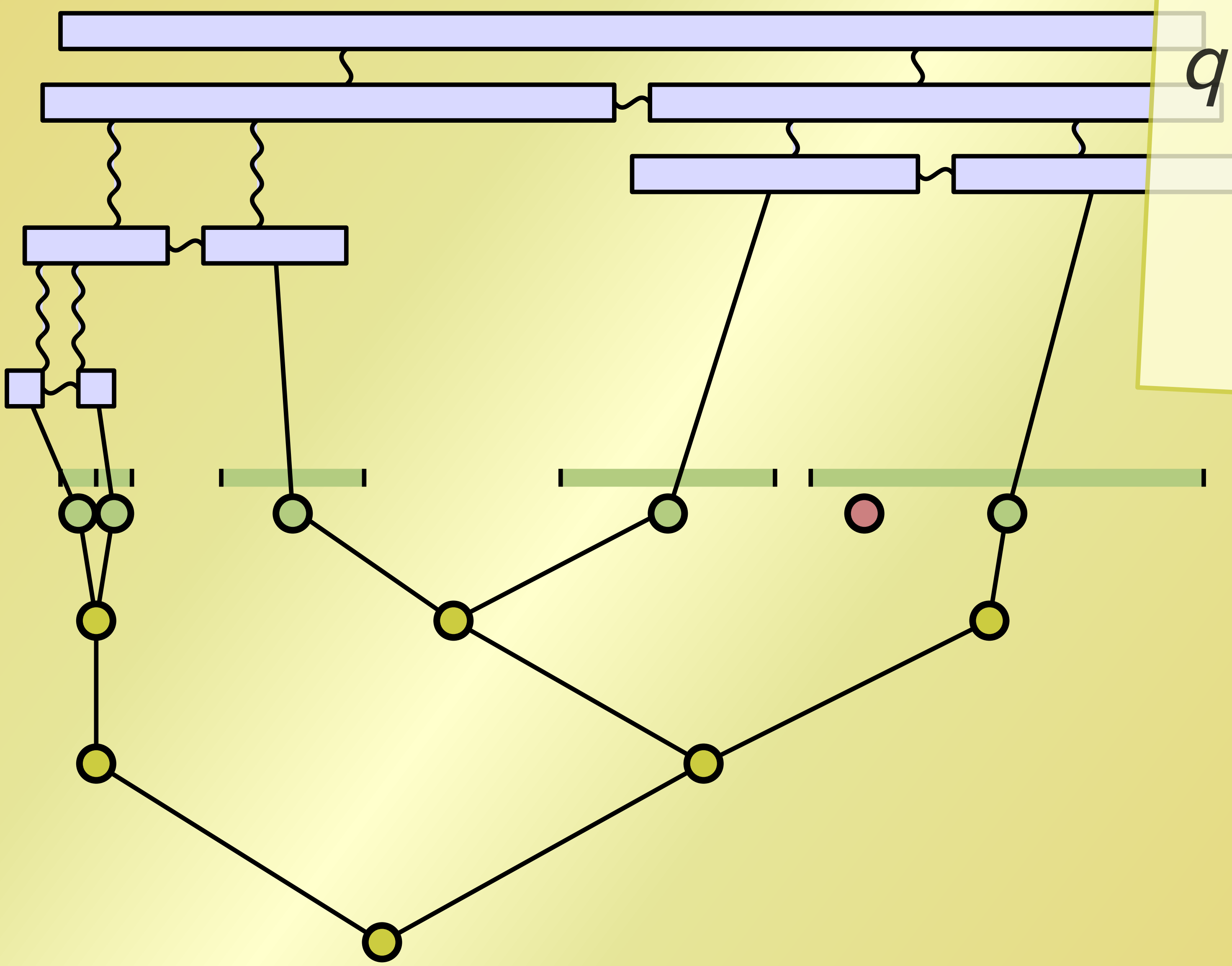


LEMMA
The search tree has logarithmic depth.

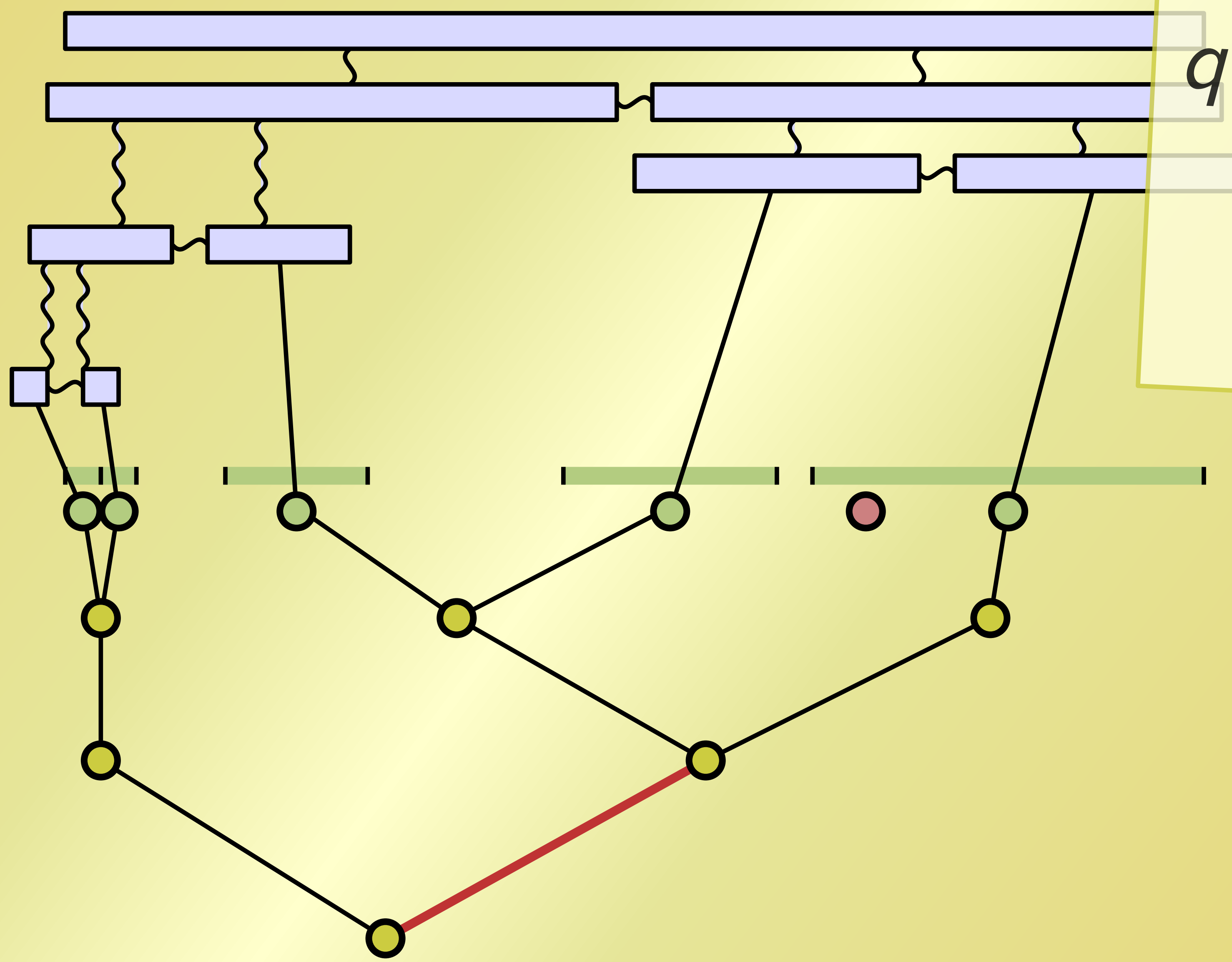




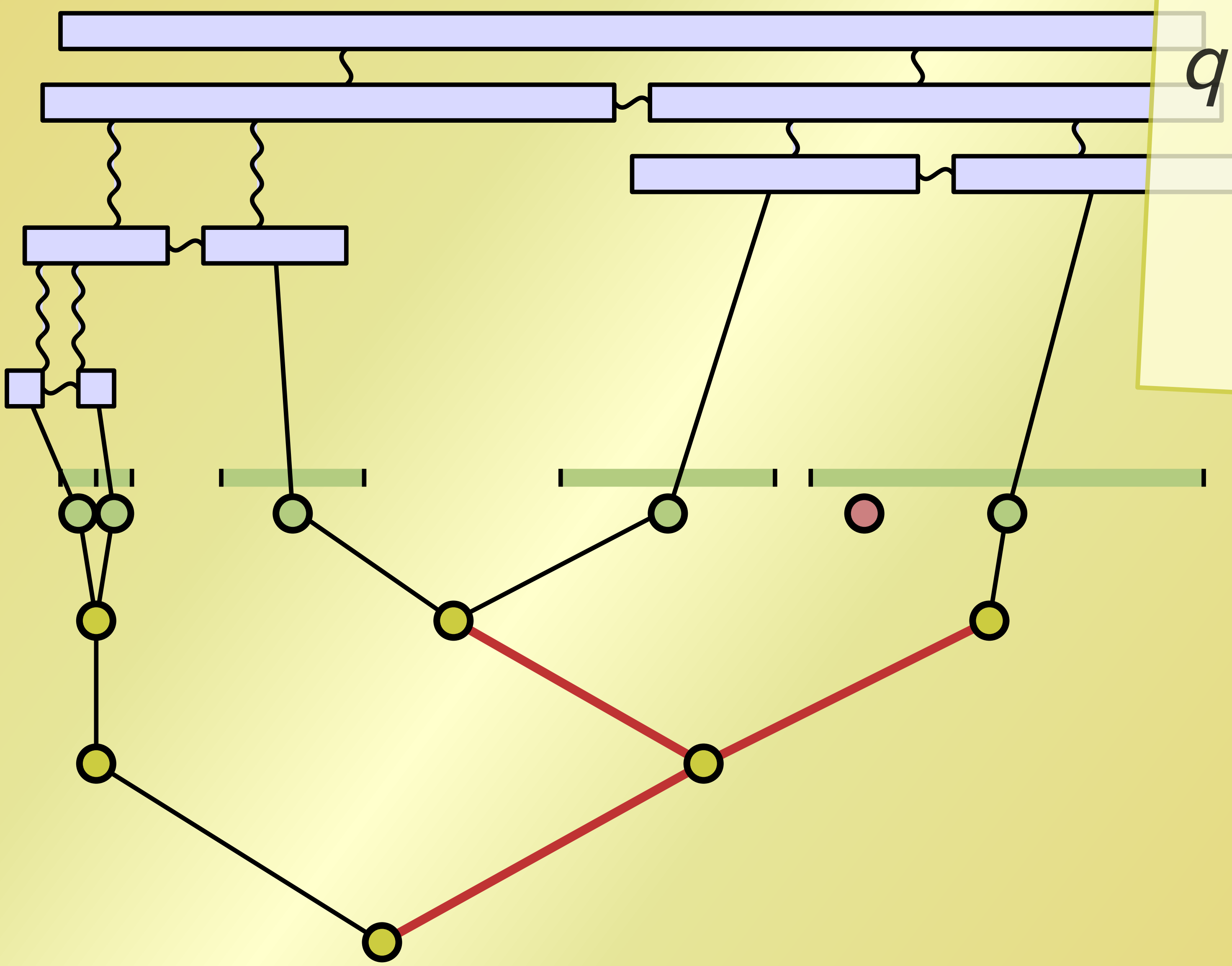
How do we handle a *query*?



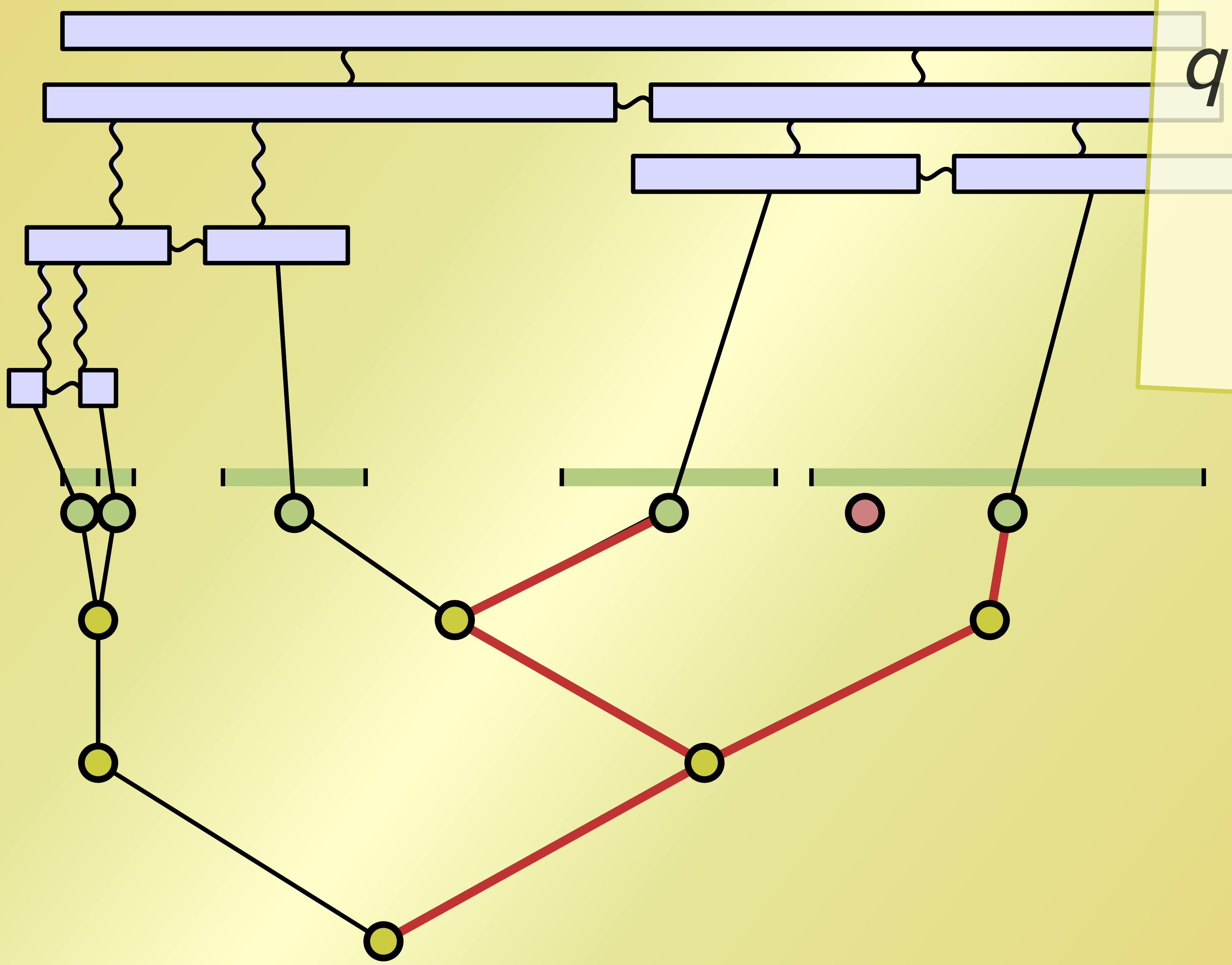
How do we handle a *query*?



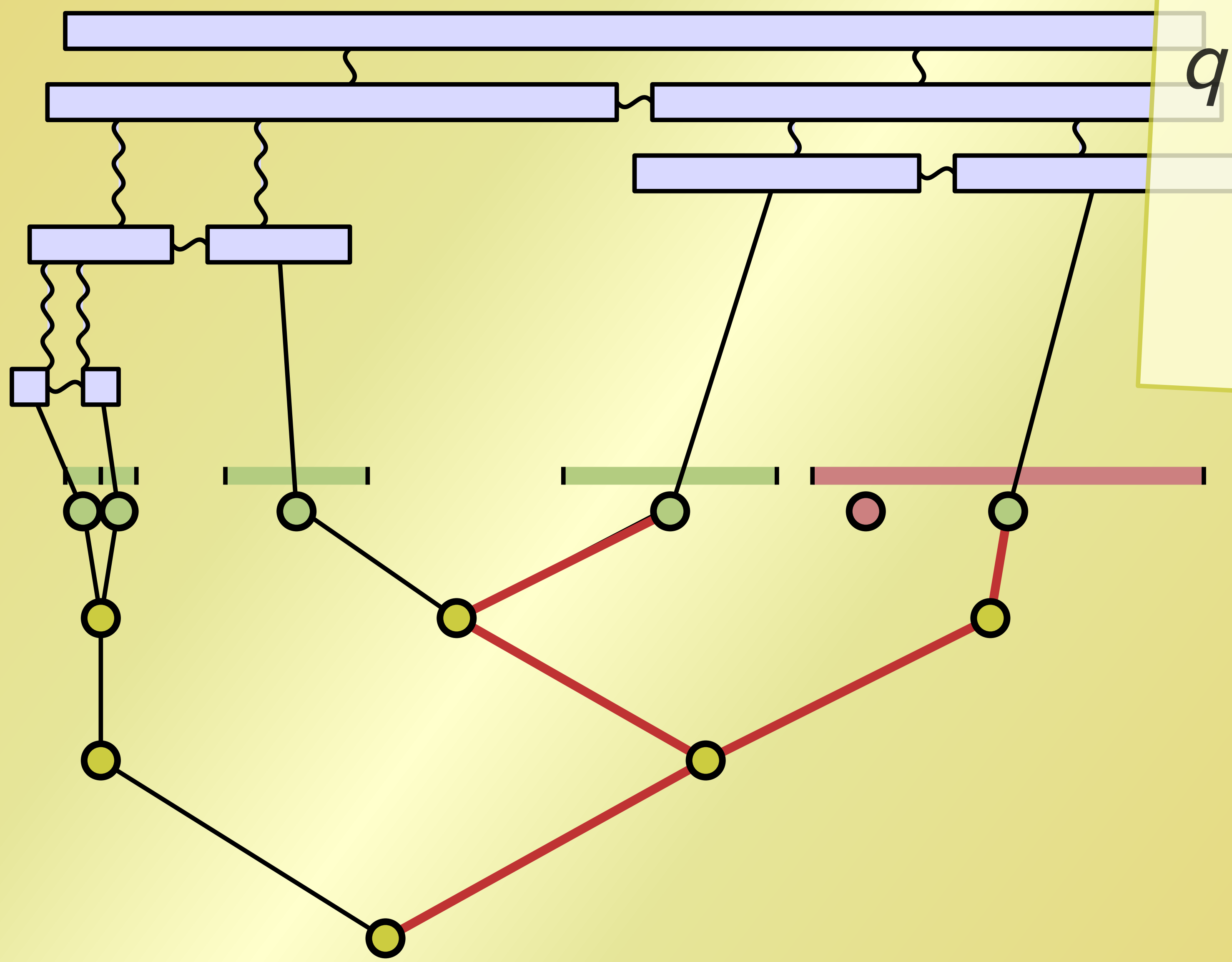
How do we handle a *query*?



How do we handle a *query*?

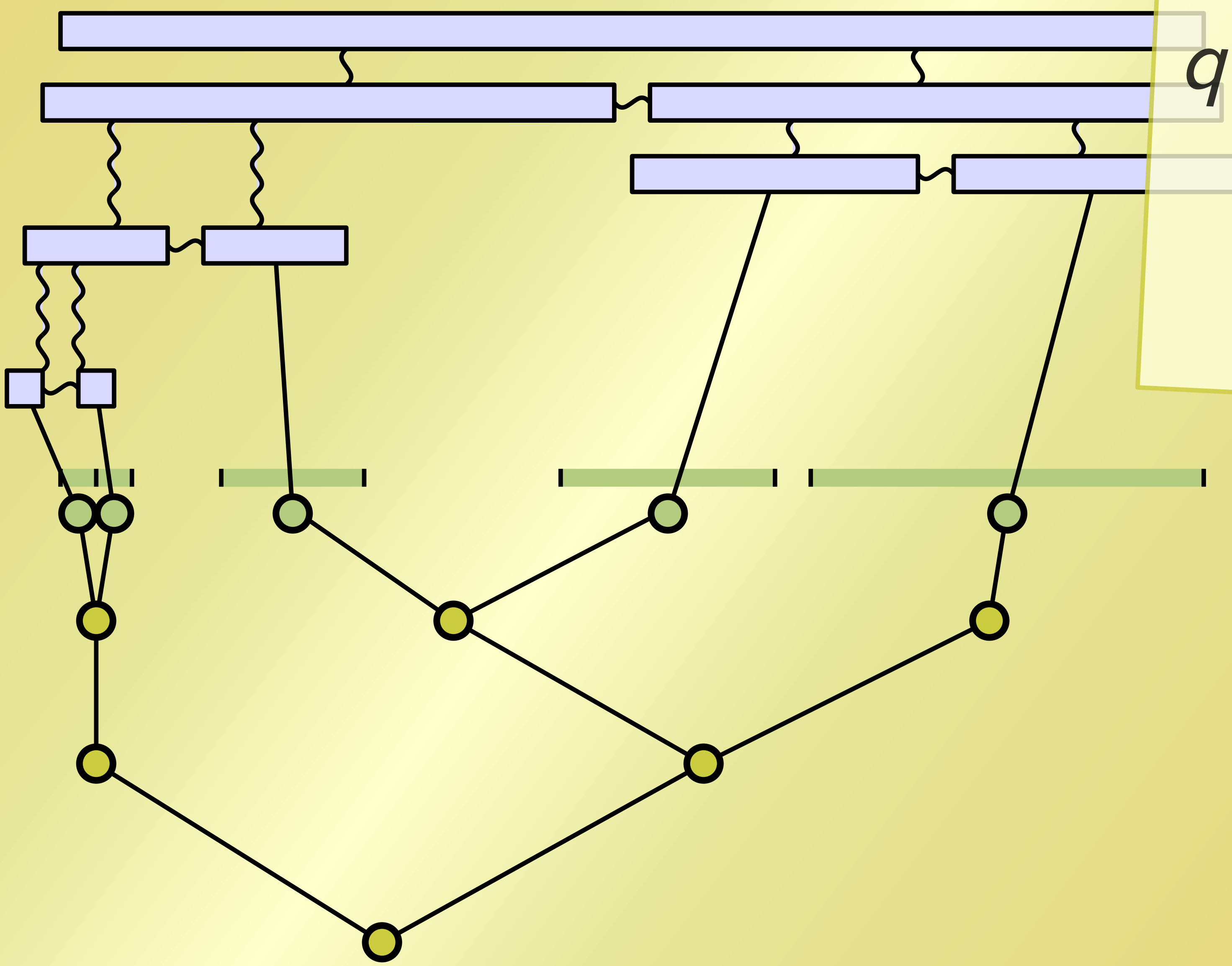


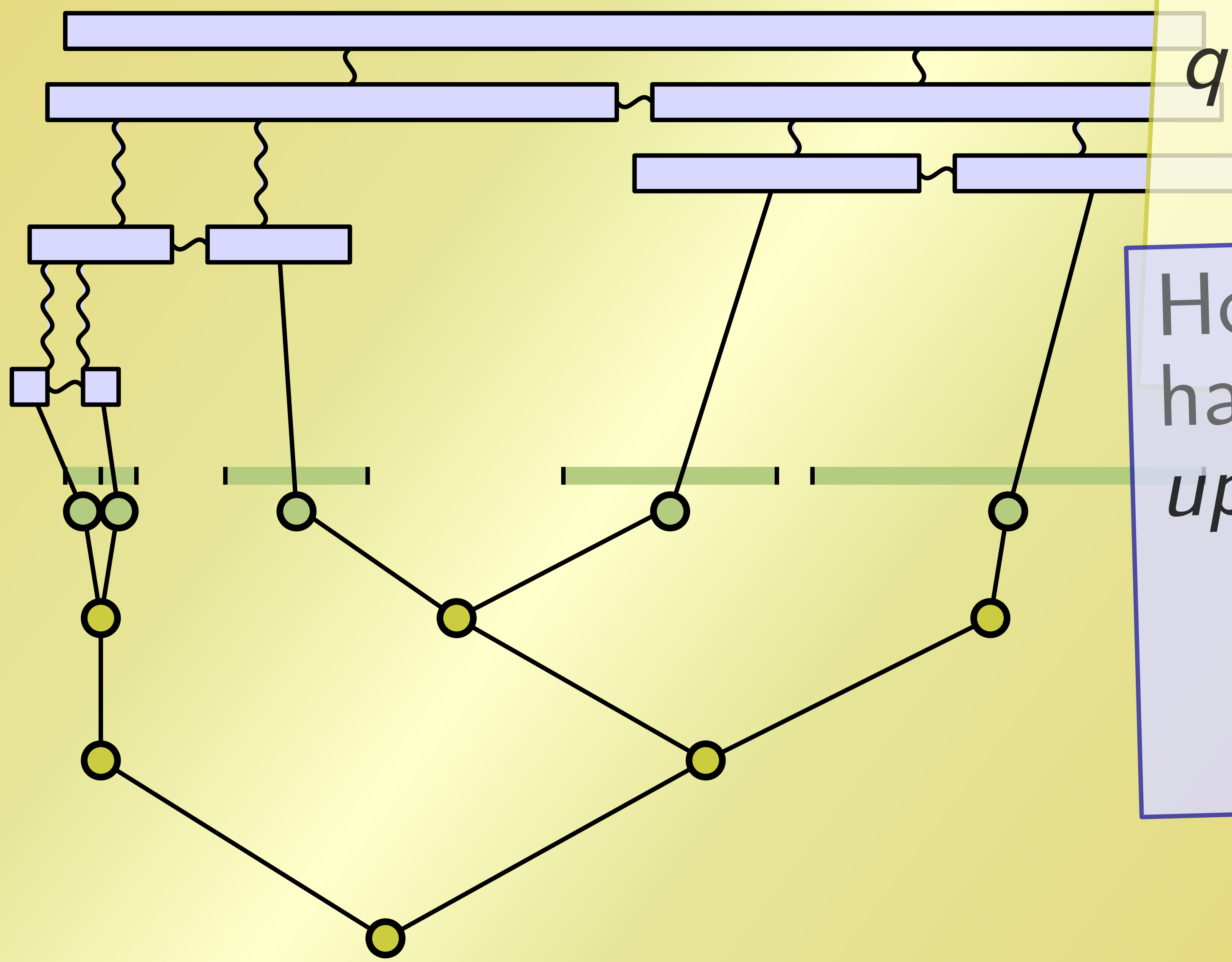
How do we handle a *query*?



How do we handle a *query*?

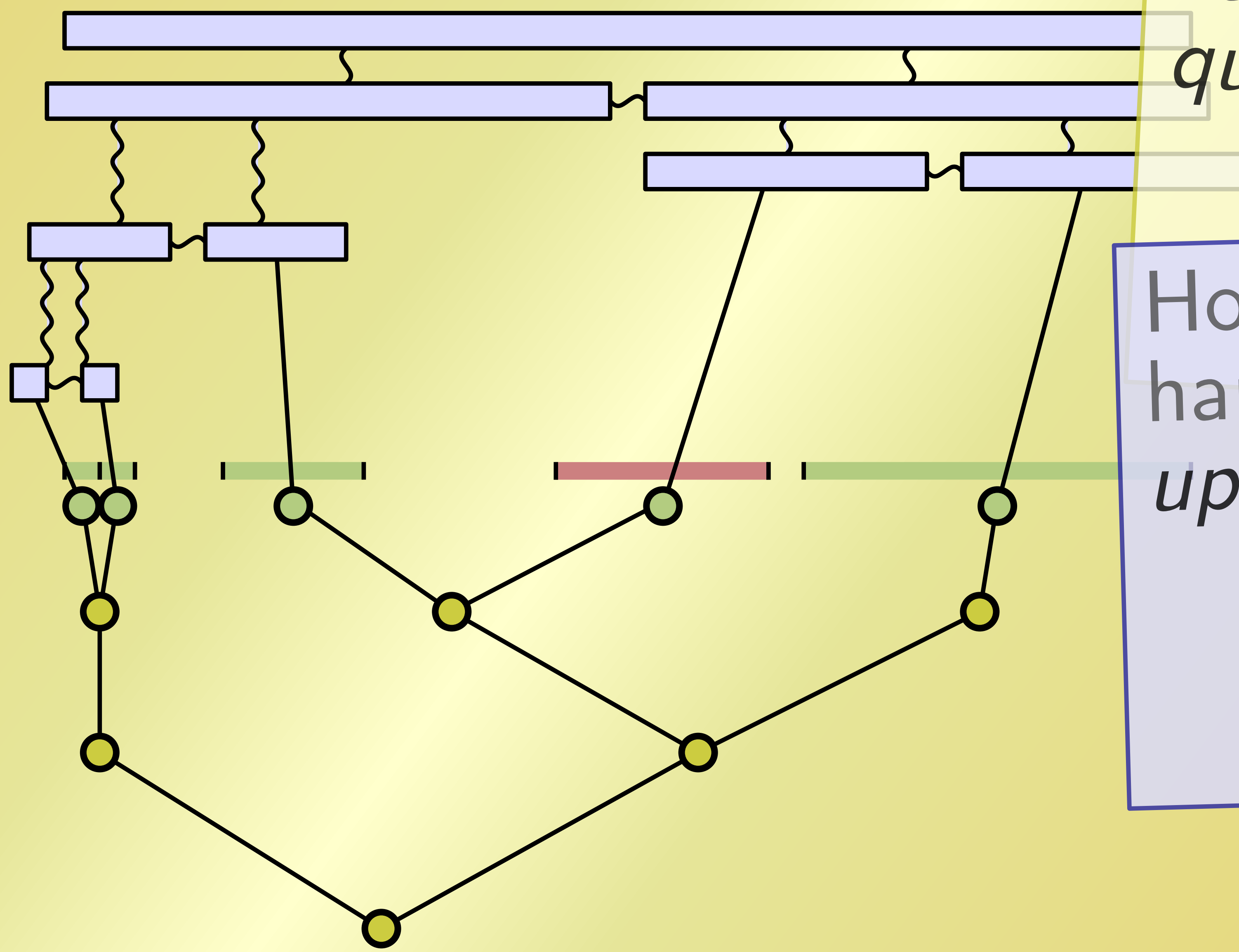
How do we handle a *query*?





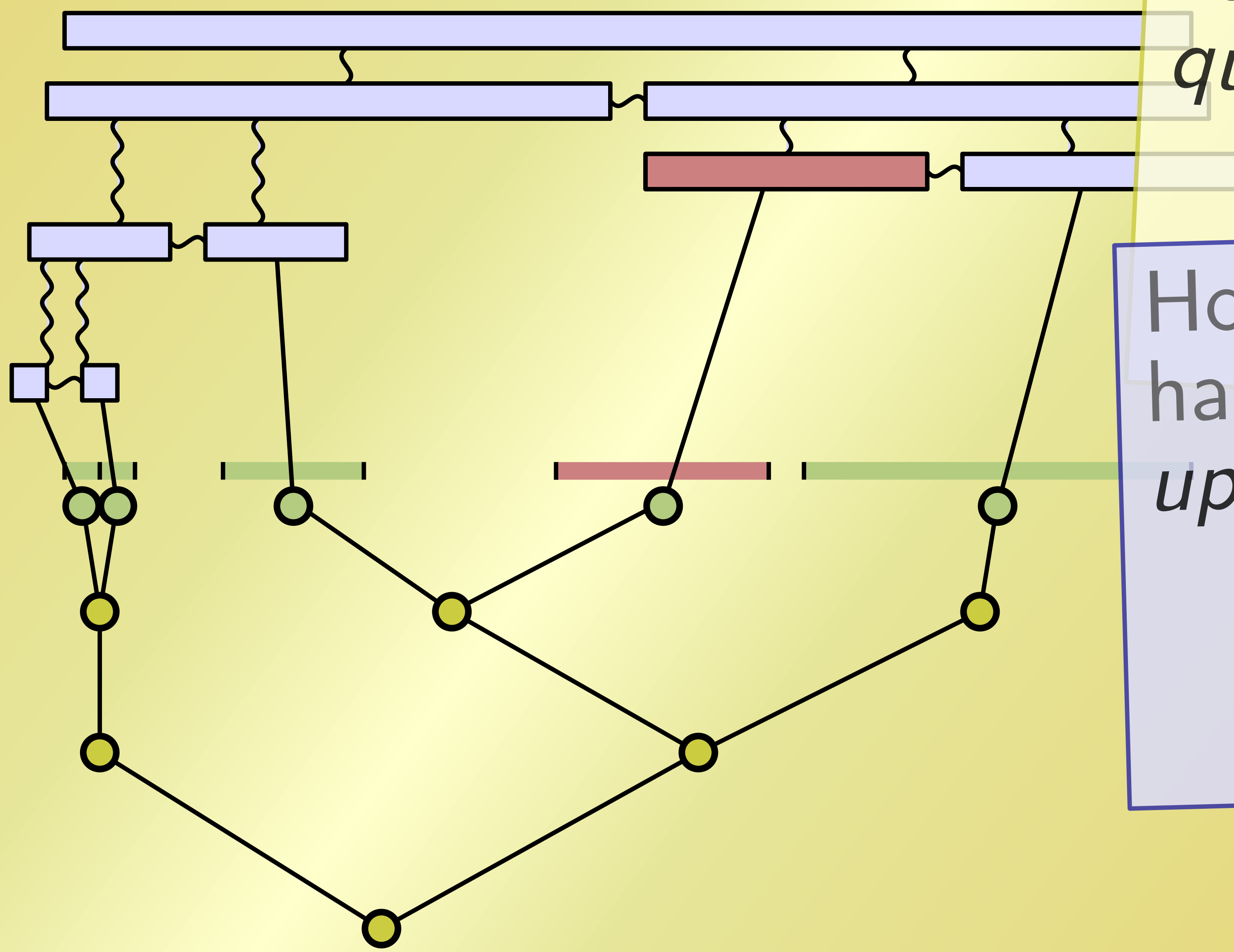
How do we handle a *query*?

How do we handle an *update*?



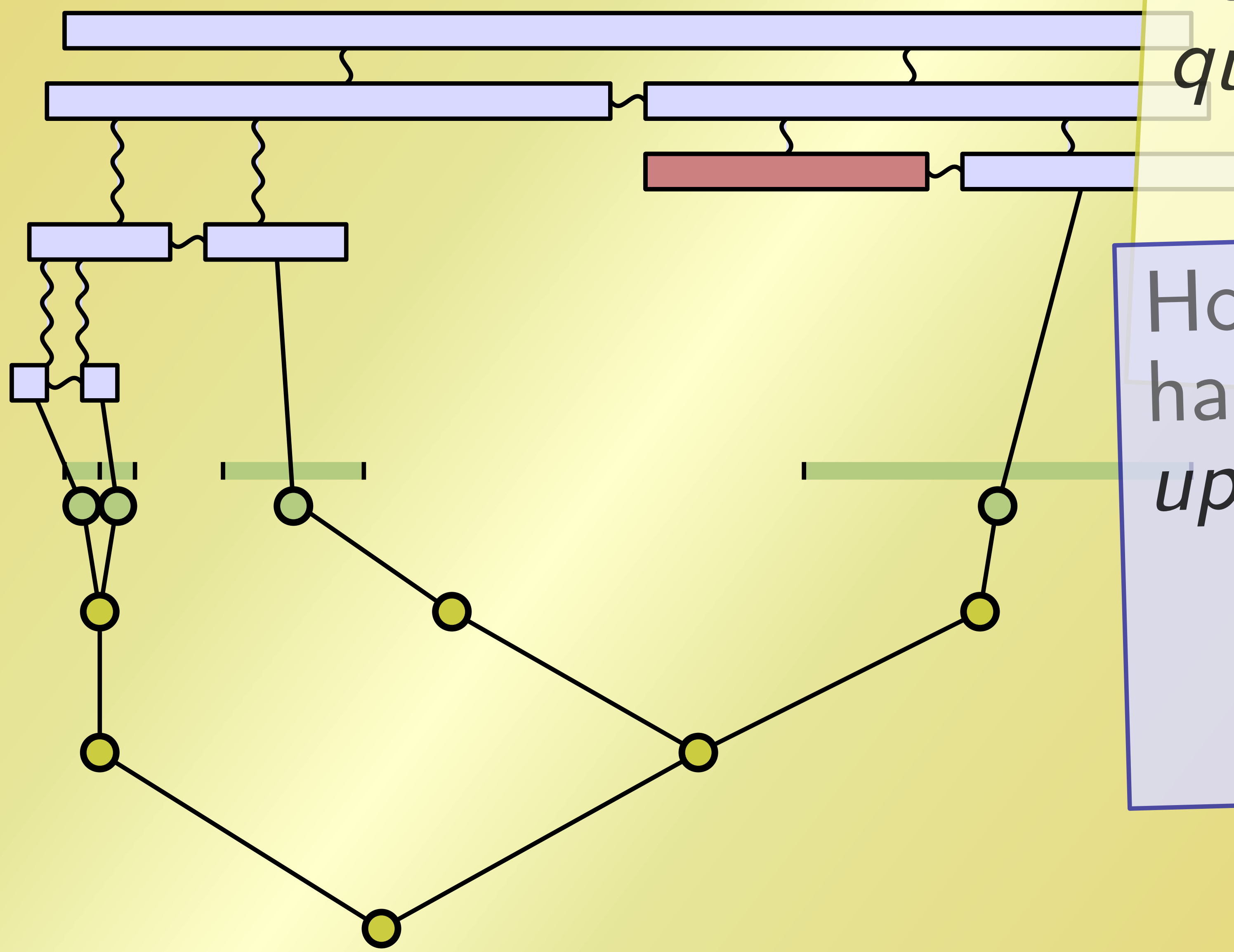
How do we handle a *query*?

How do we handle an *update*?



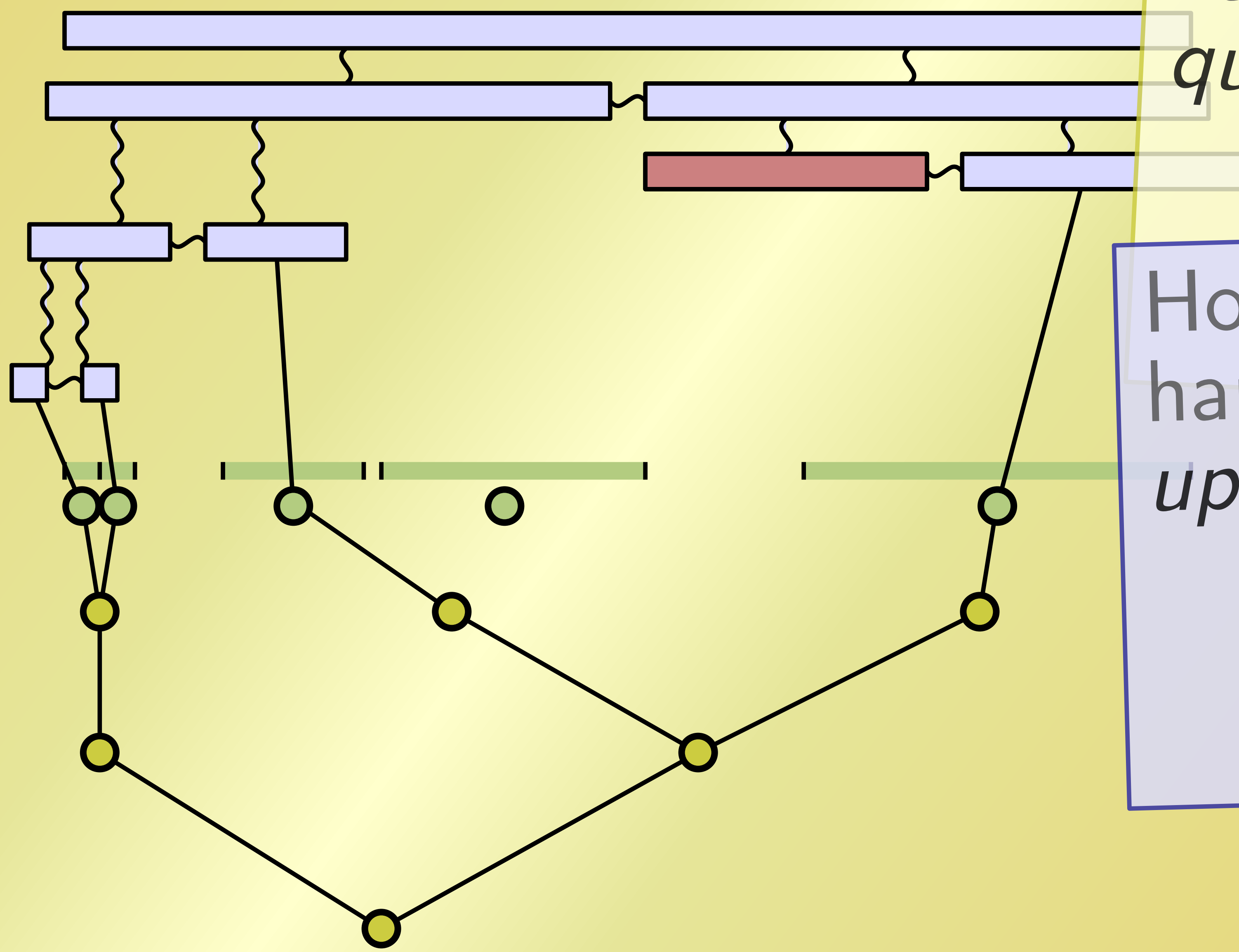
How do we handle a *query*?

How do we handle an *update*?



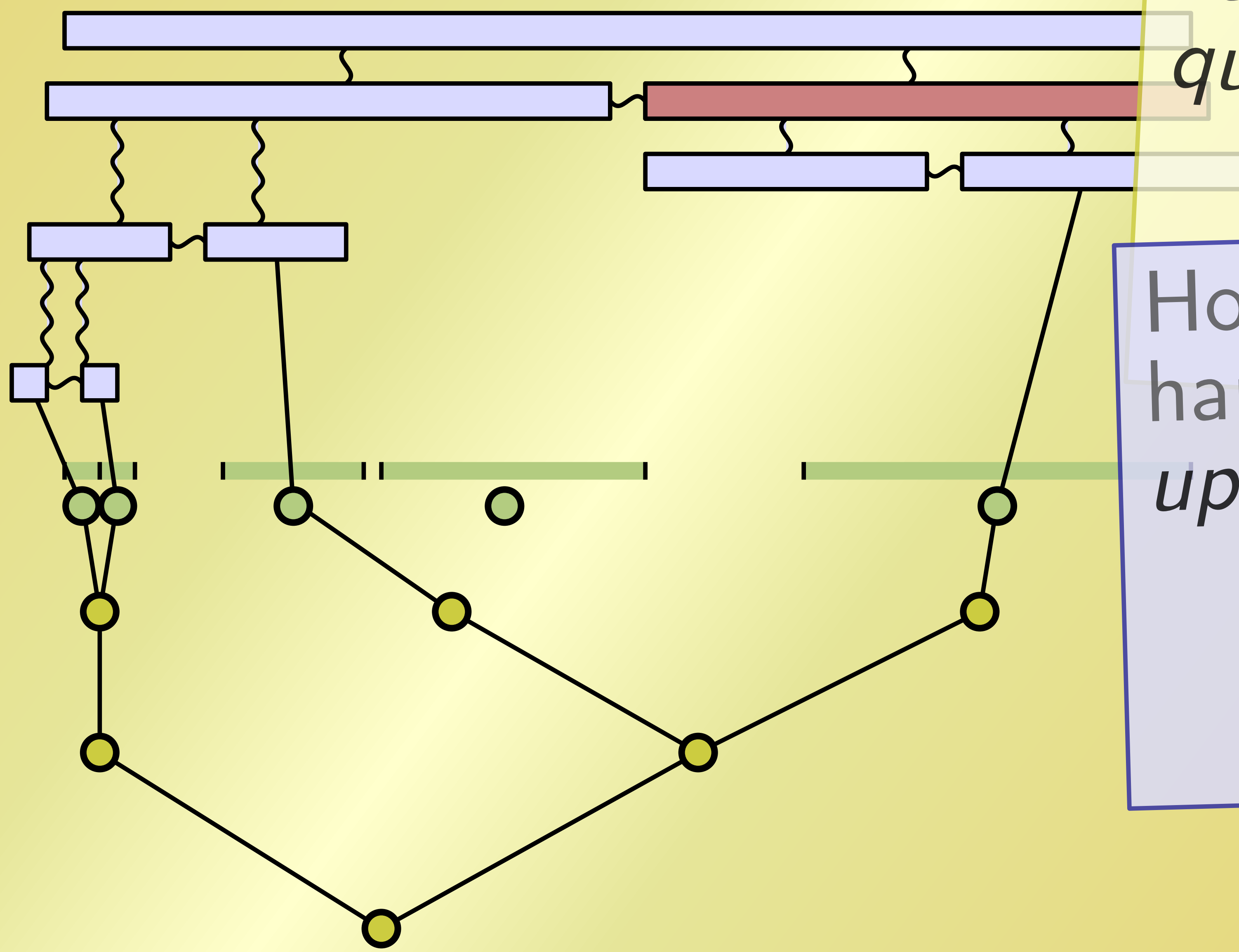
How do we handle a *query*?

How do we handle an *update*?



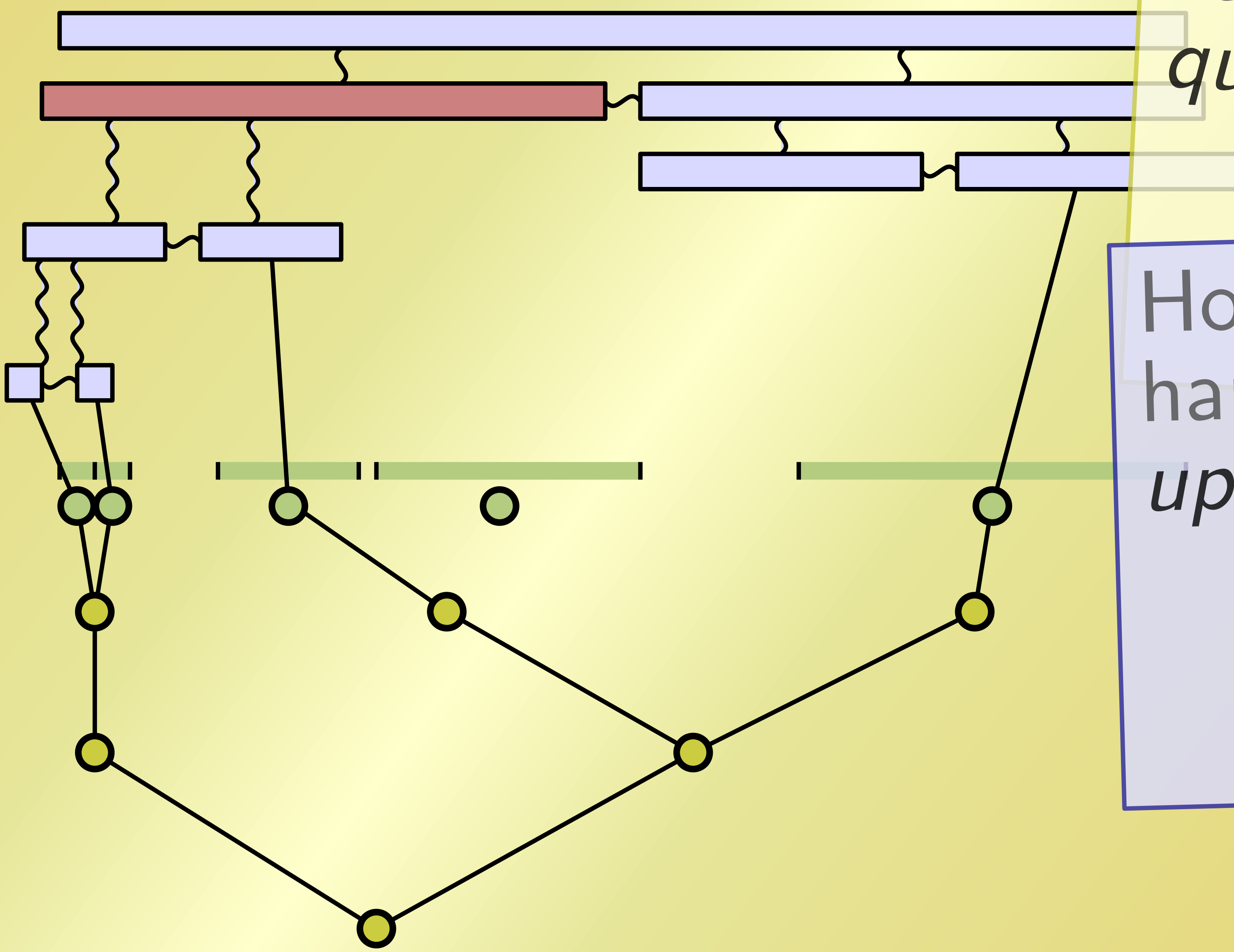
How do we handle a *query*?

How do we handle an *update*?



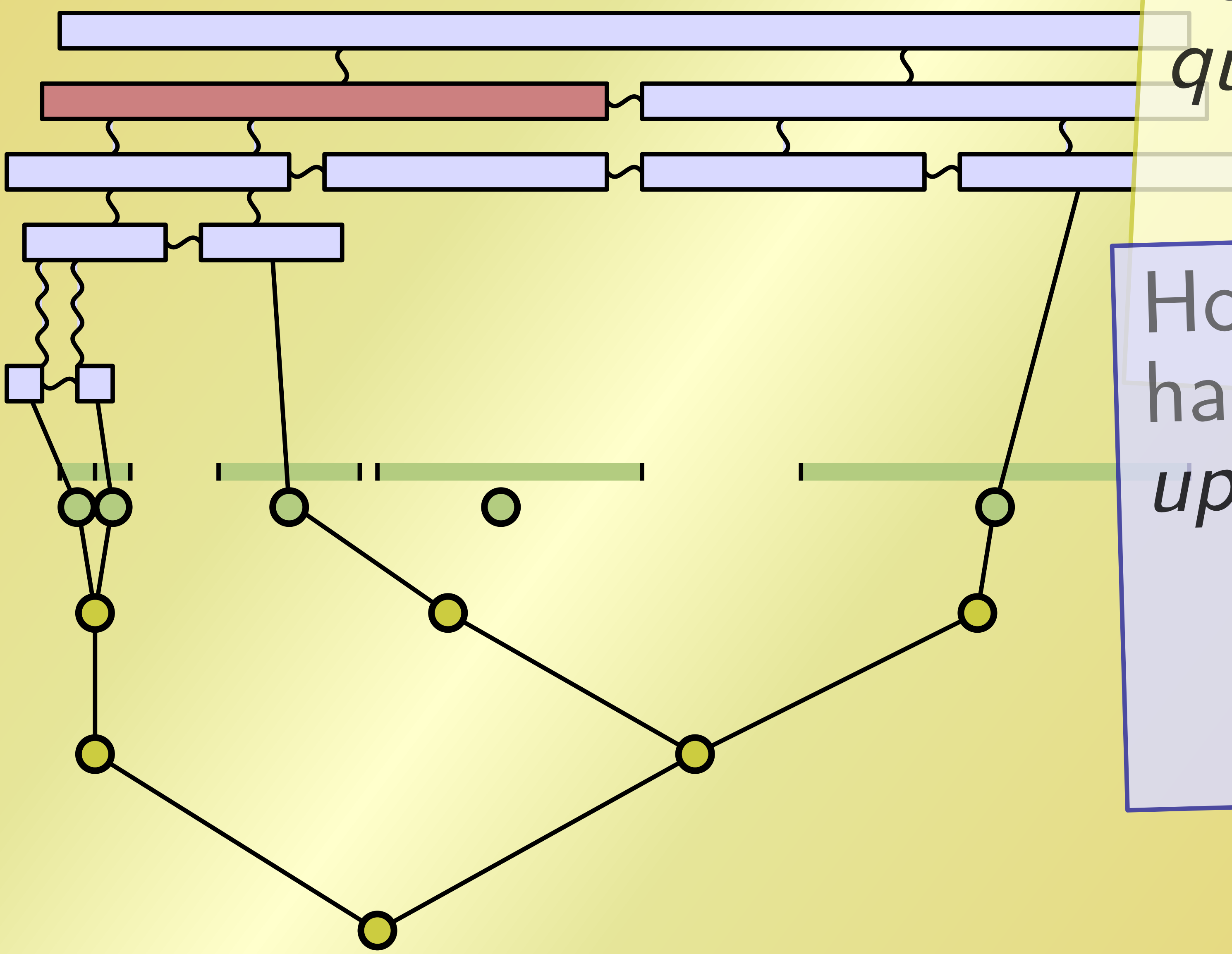
How do we handle a *query*?

How do we handle an *update*?



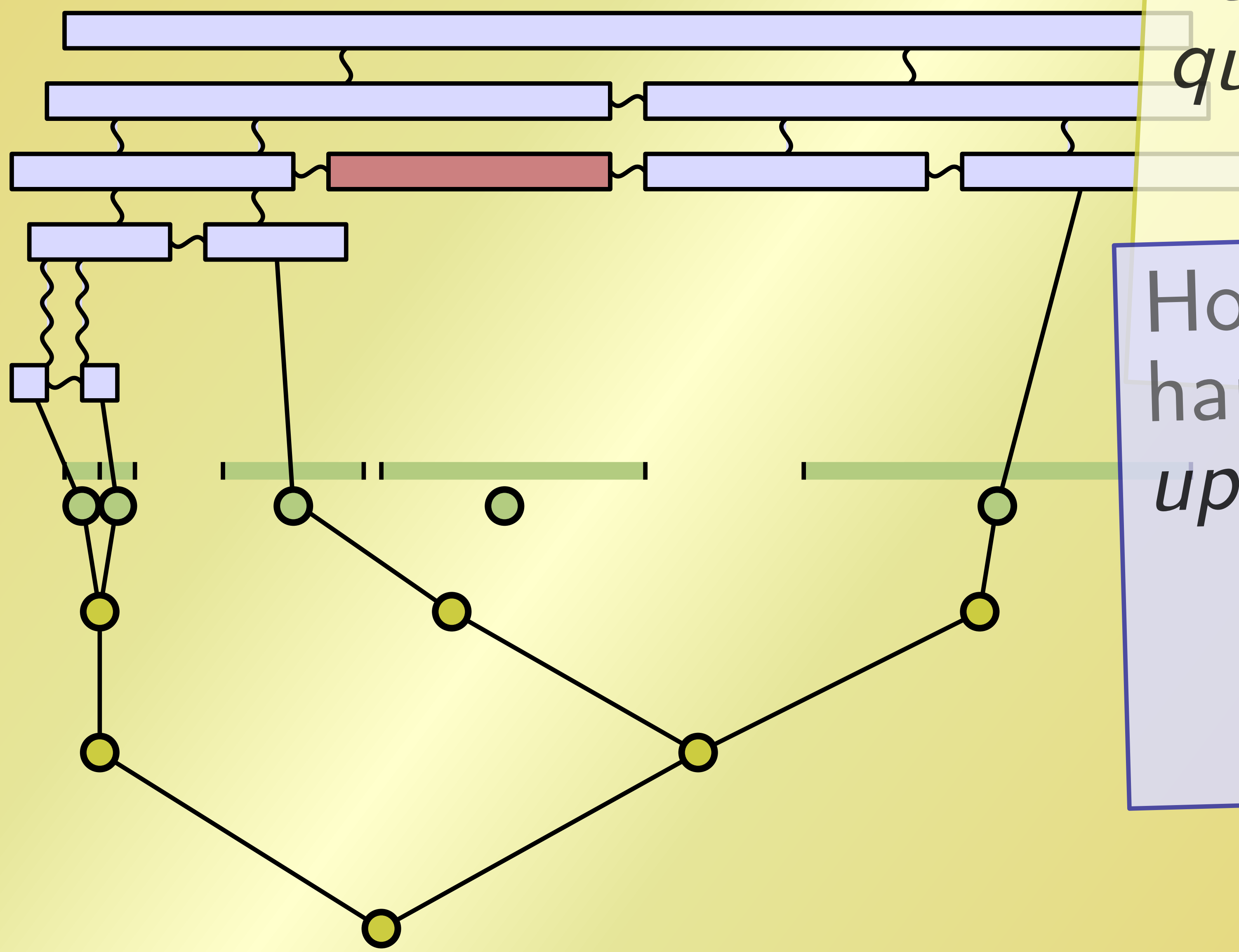
How do we handle a *query*?

How do we handle an *update*?



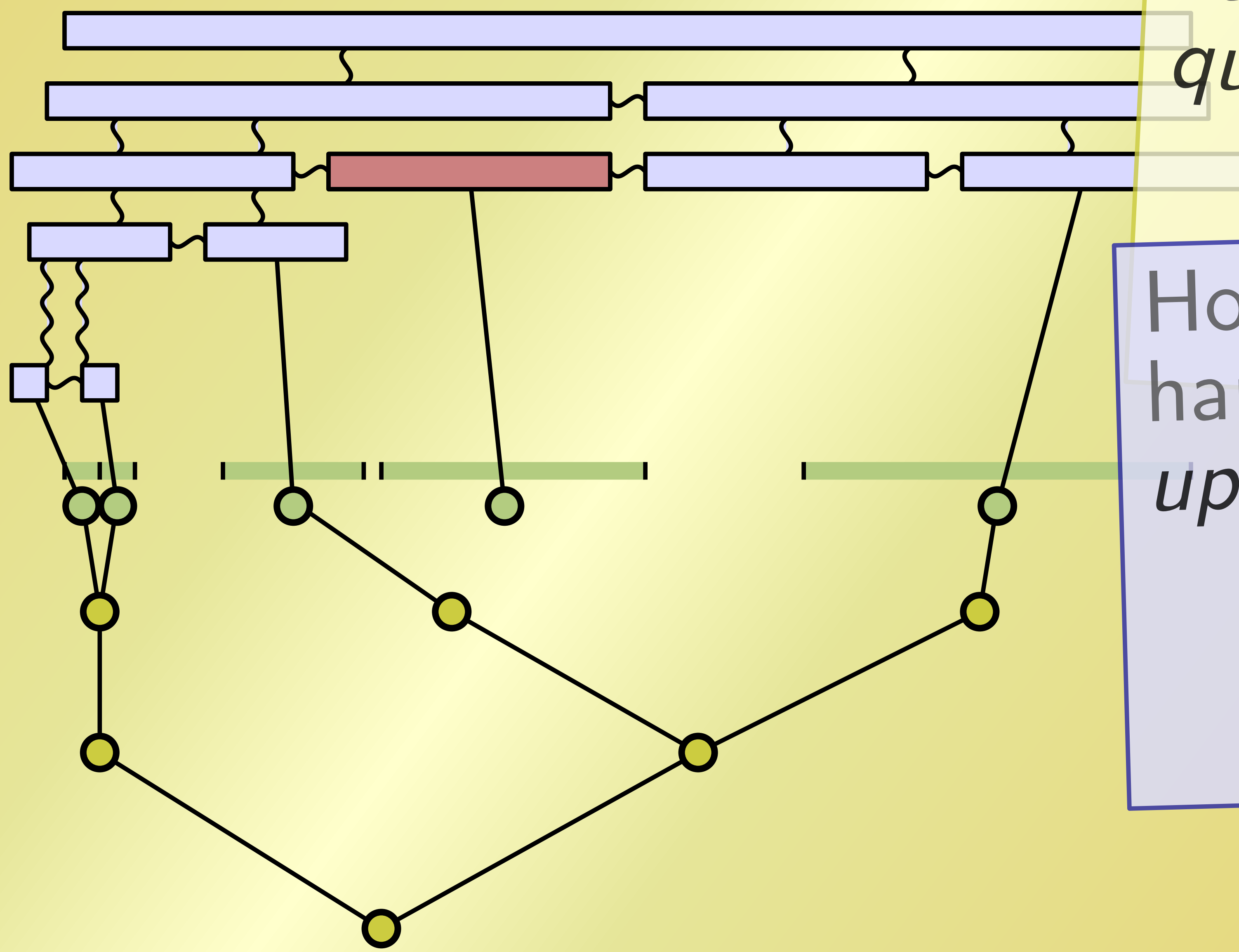
How do we handle a *query*?

How do we handle an *update*?



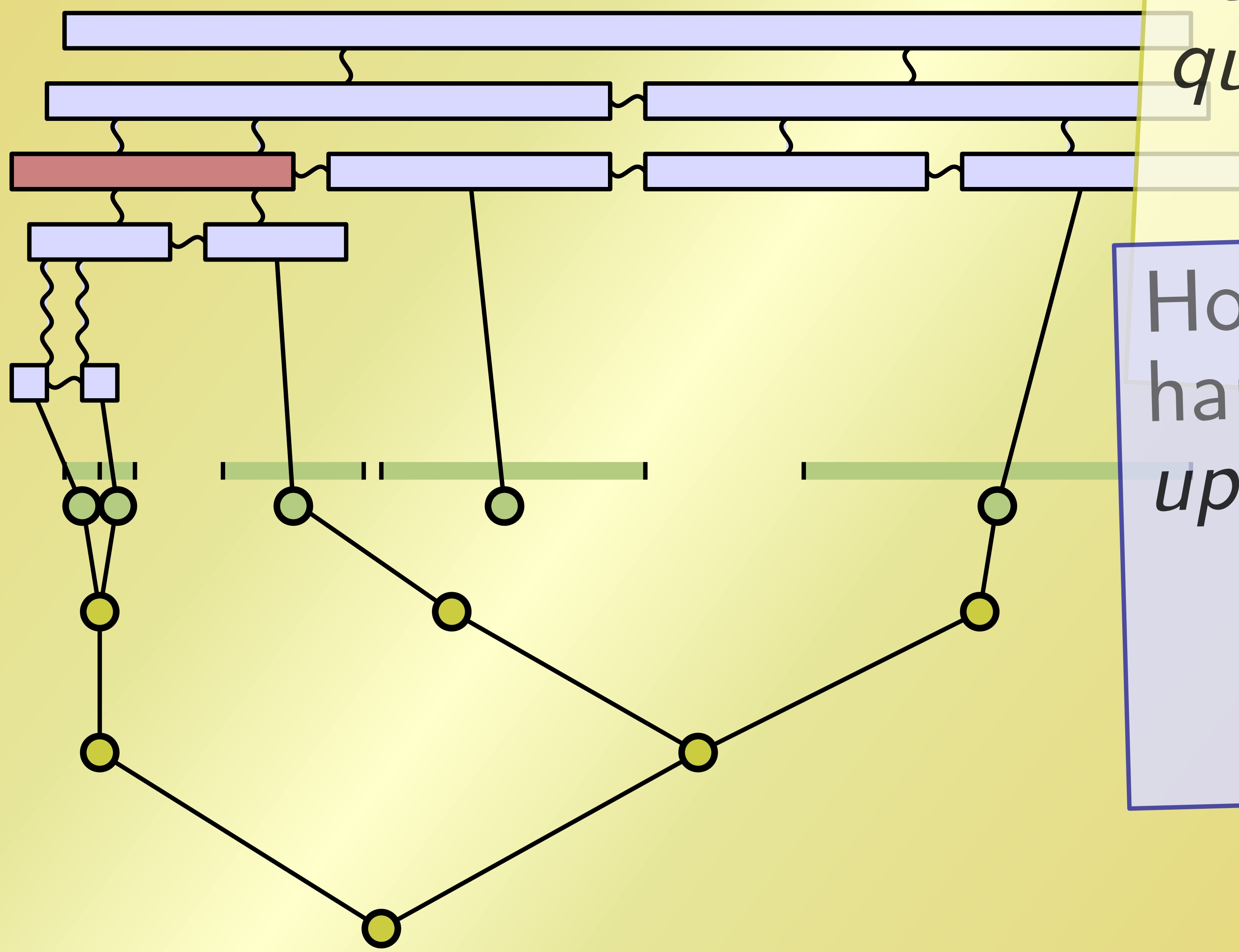
How do we handle a *query*?

How do we handle an *update*?



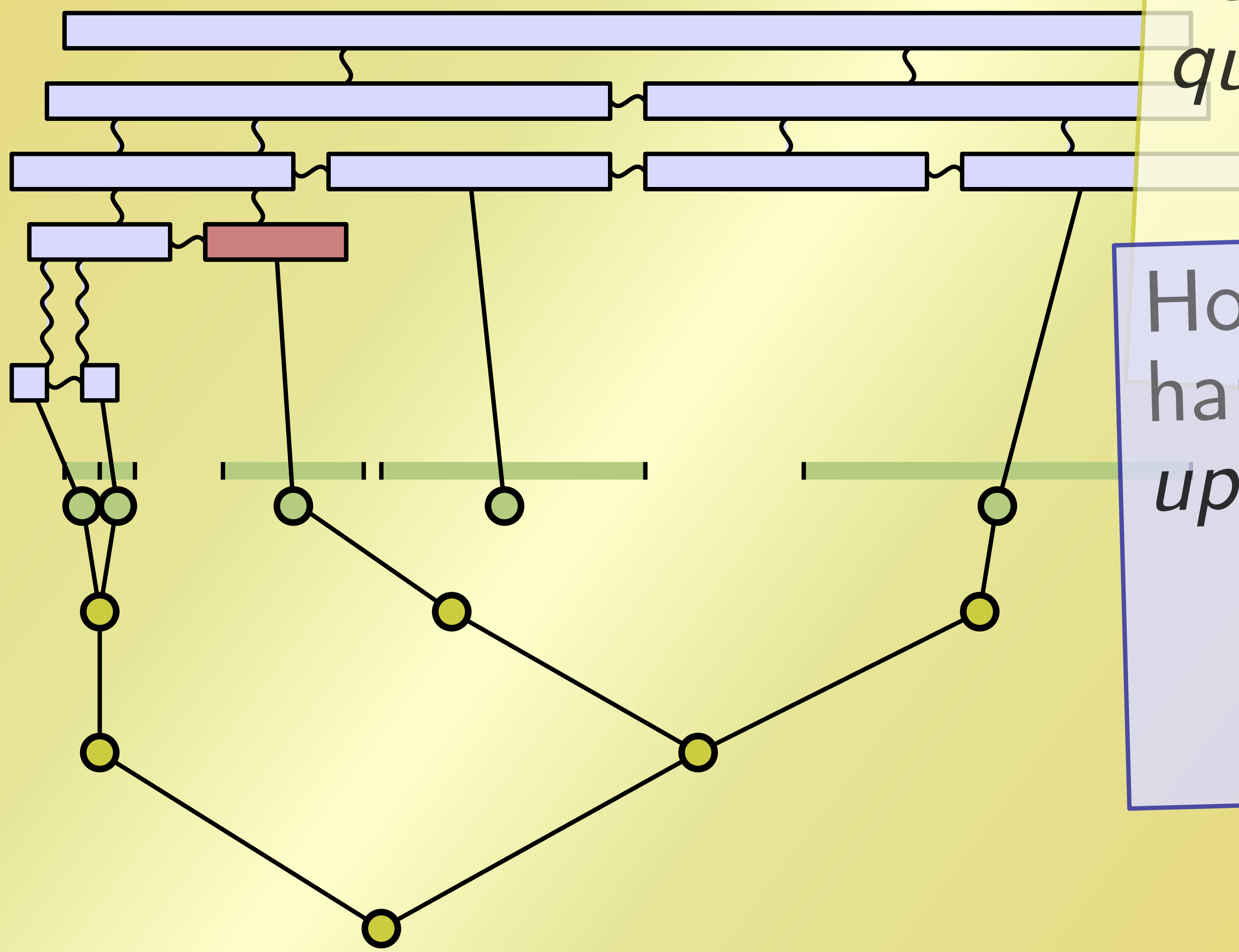
How do we handle a *query*?

How do we handle an *update*?



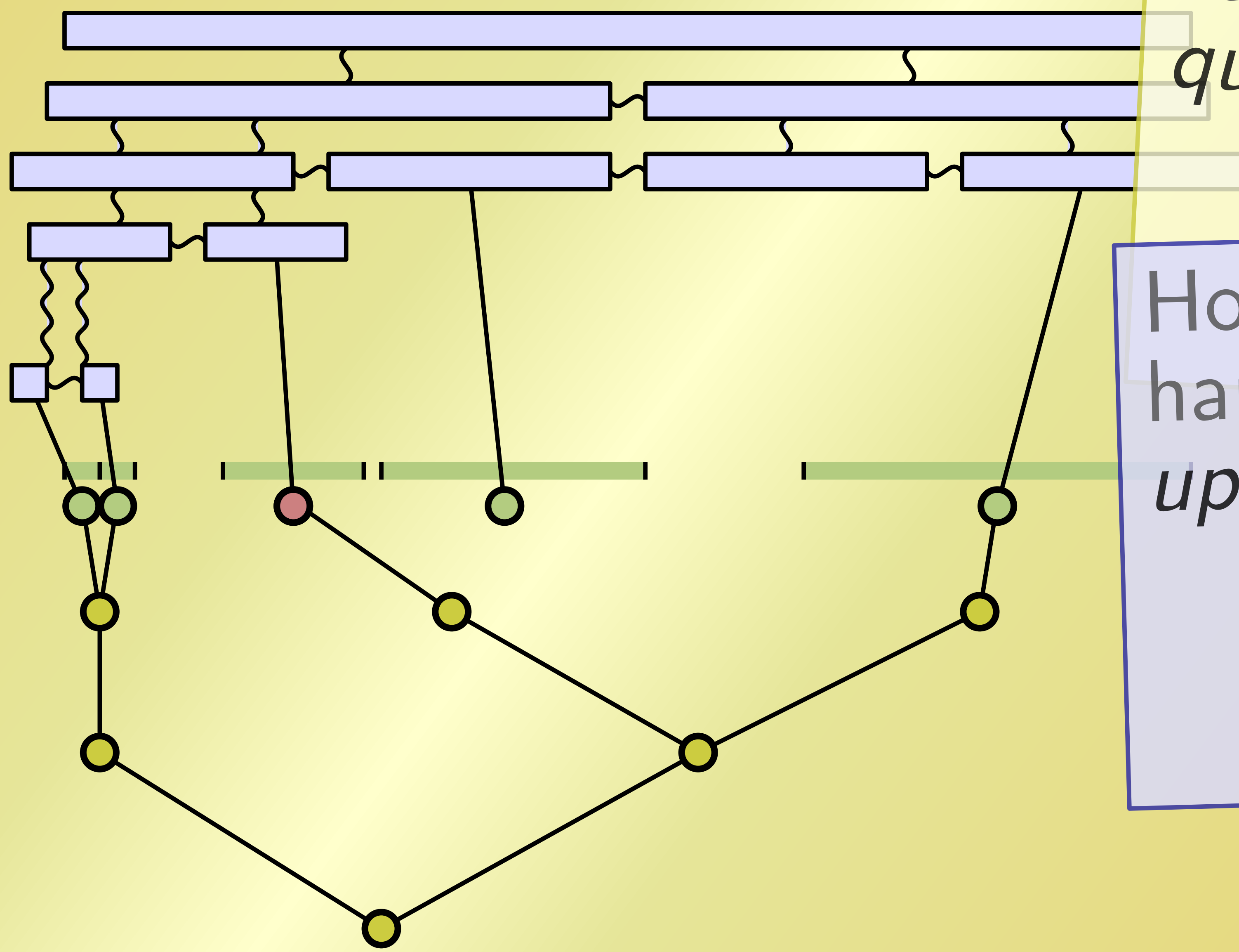
How do we handle a *query*?

How do we handle an *update*?



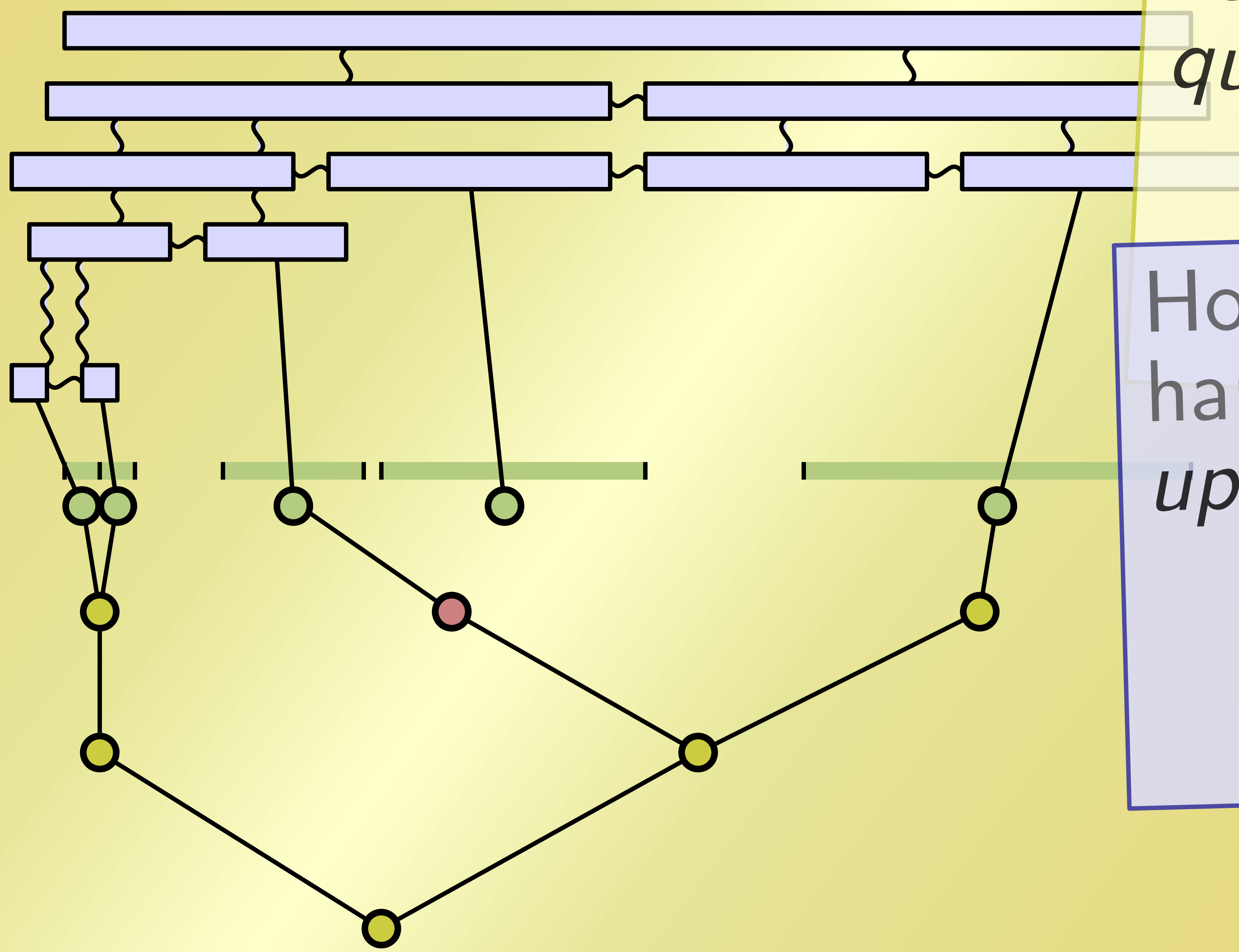
How do we handle a *query*?

How do we handle an *update*?



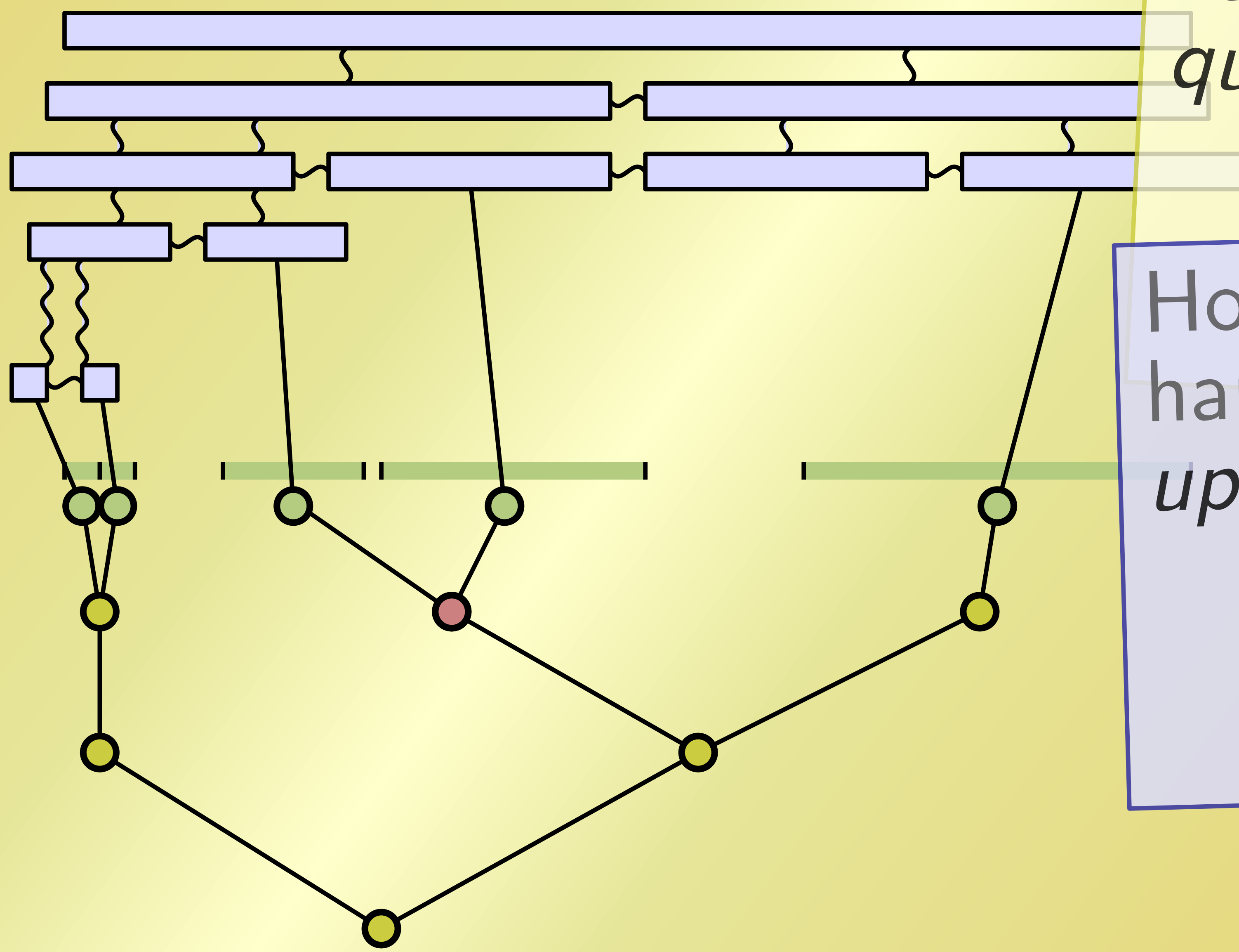
How do we handle a *query*?

How do we handle an *update*?



How do we handle a *query*?

How do we handle an *update*?



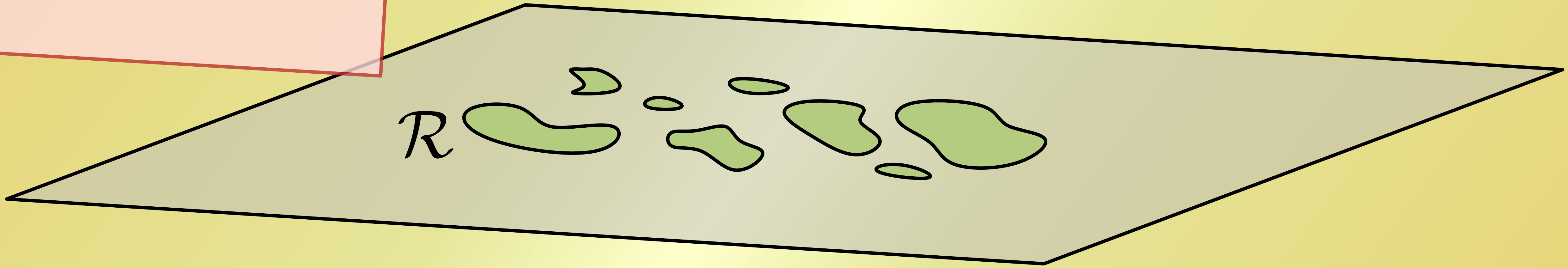
How do we handle a *query*?

How do we handle an *update*?

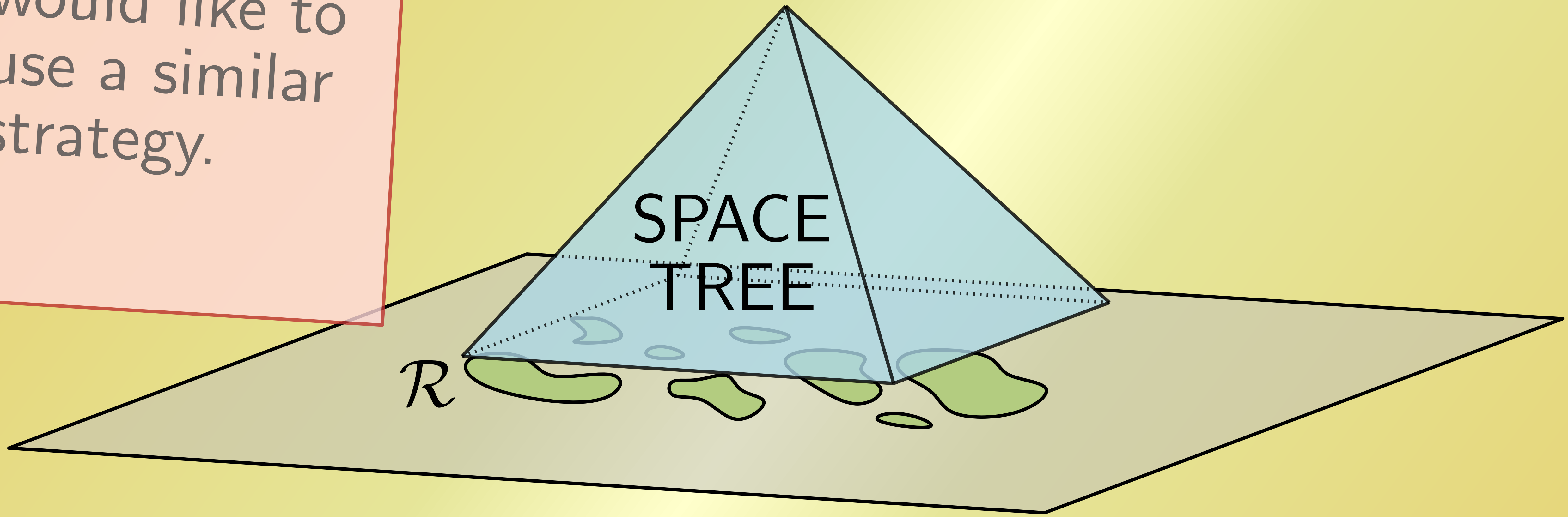
TECHNICAL DETAILS:
2 DIMENSIONS

In $\mathbb{R}^2$, we would like to use a similar strategy.

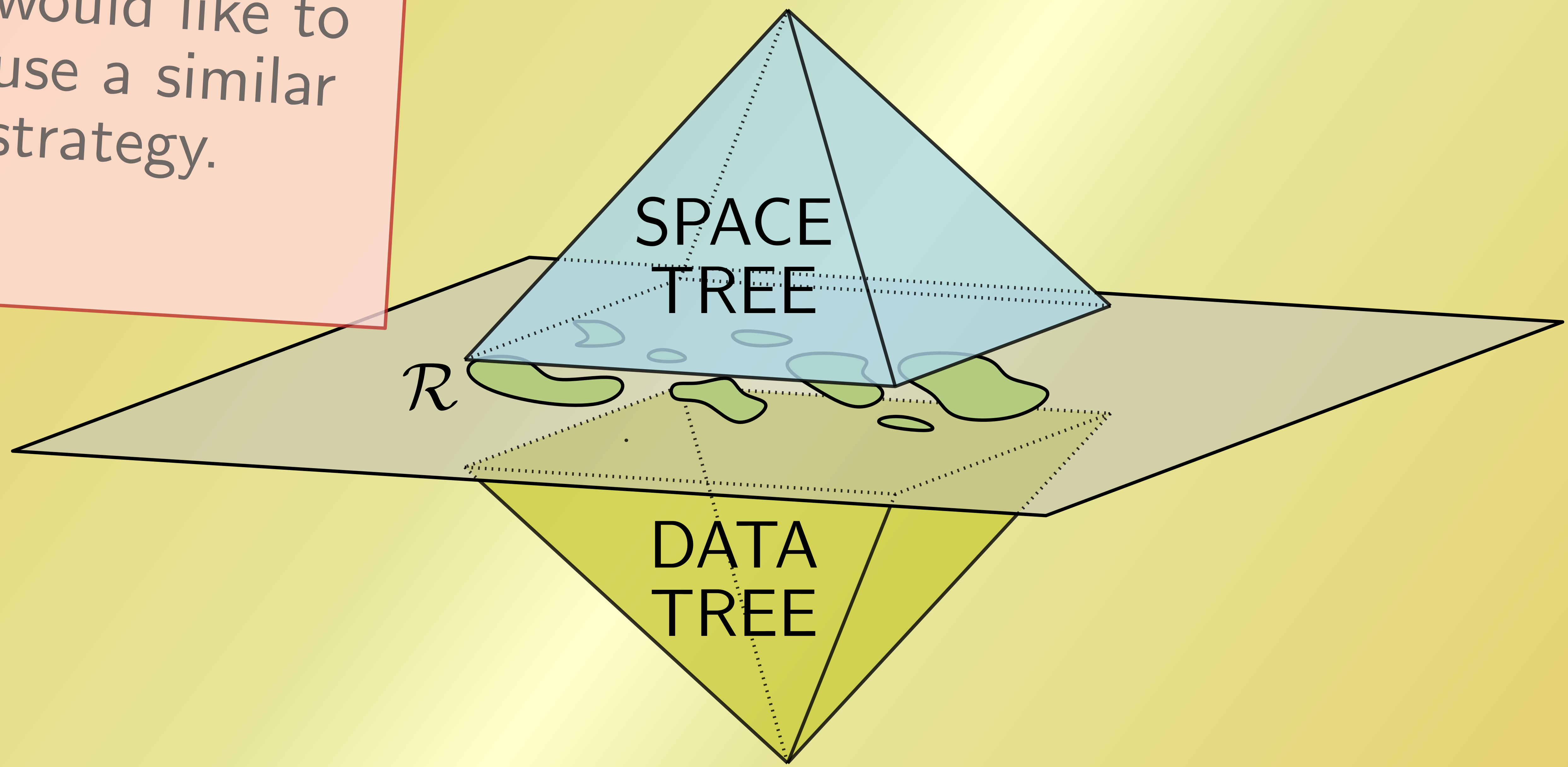
In $\mathbb{R}^2$, we would like to use a similar strategy.



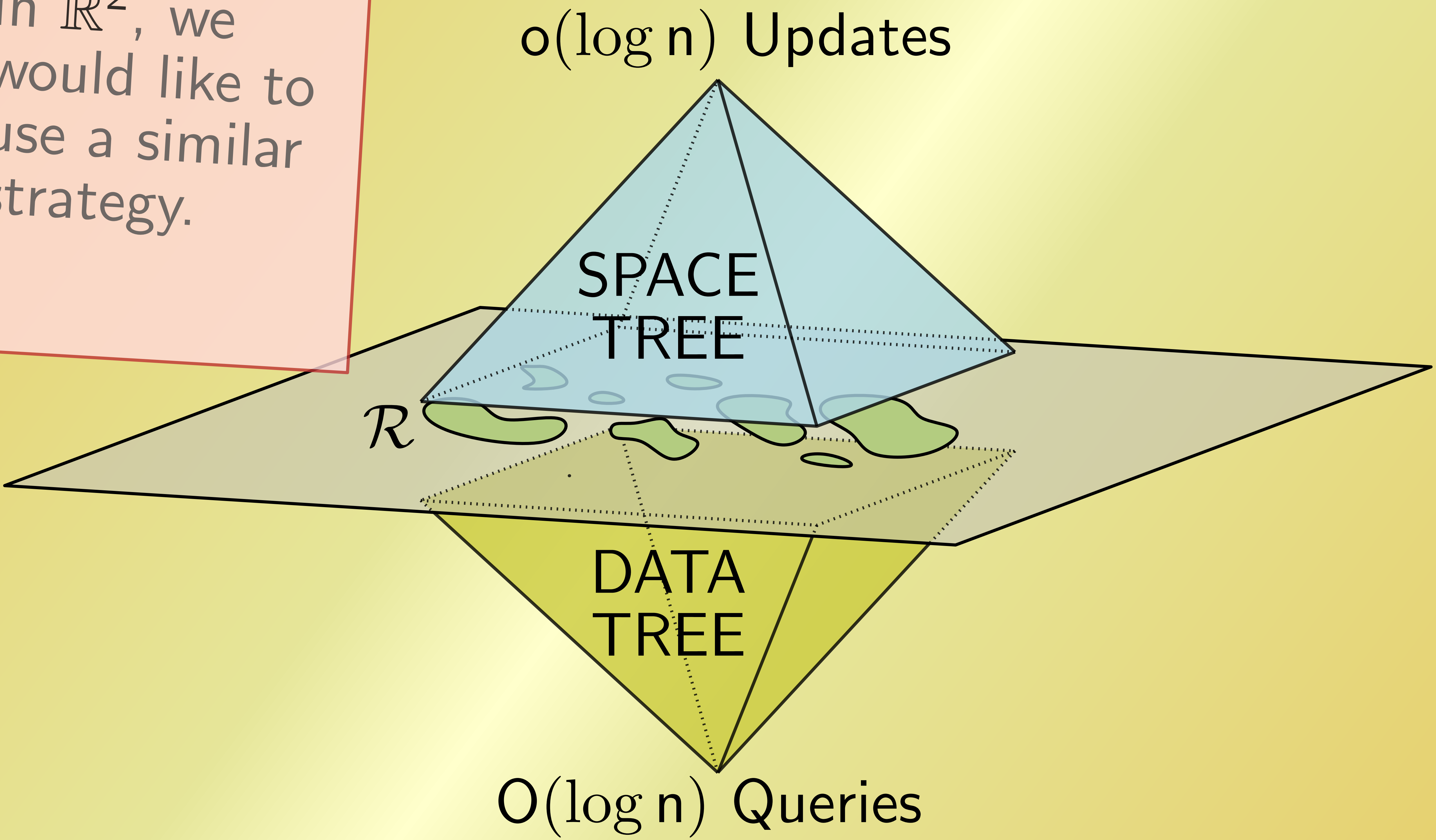
In $\mathbb{R}^2$, we would like to use a similar strategy.



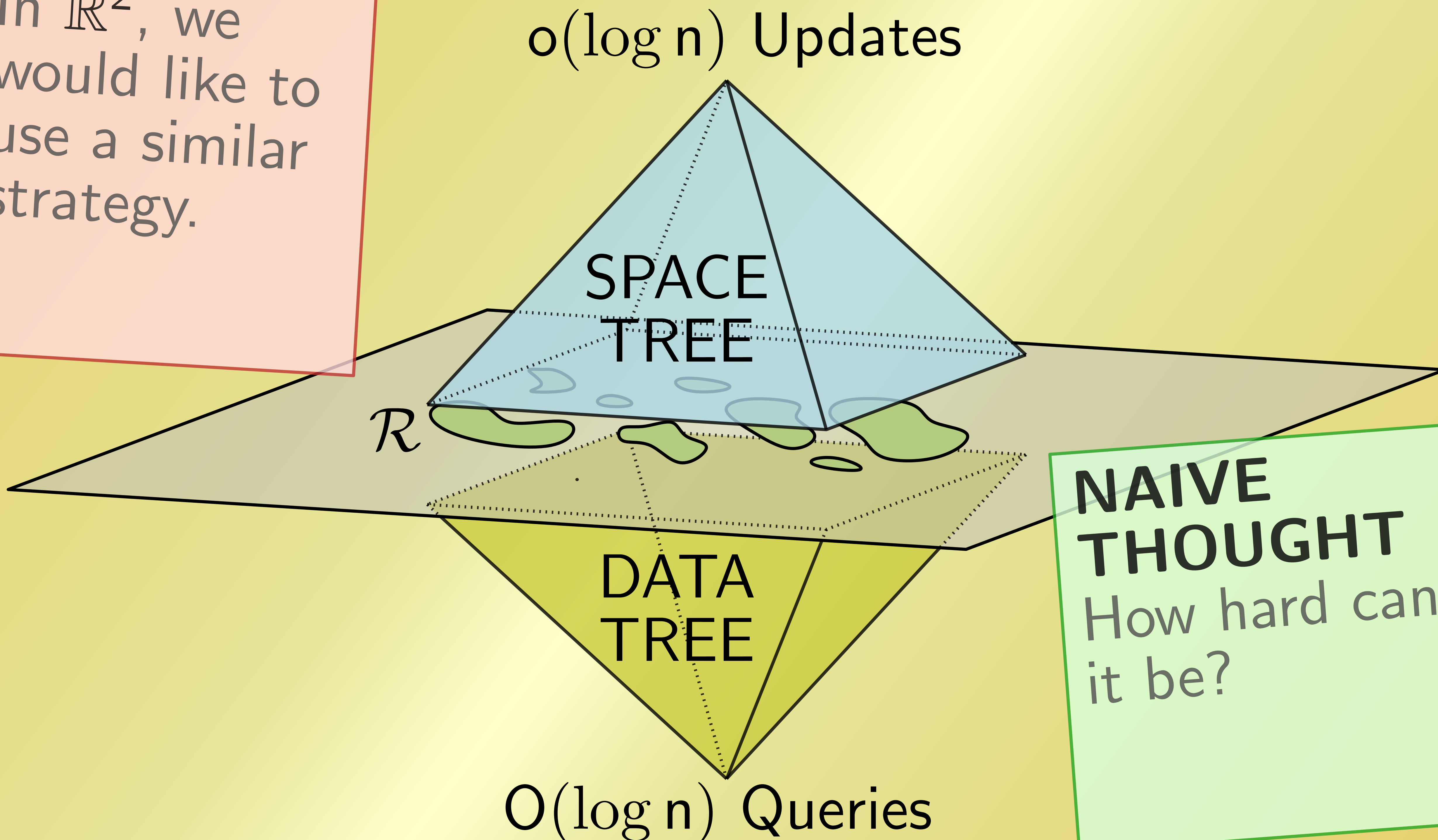
In $\mathbb{R}^2$, we would like to use a similar strategy.



In $\mathbb{R}^2$, we would like to use a similar strategy.

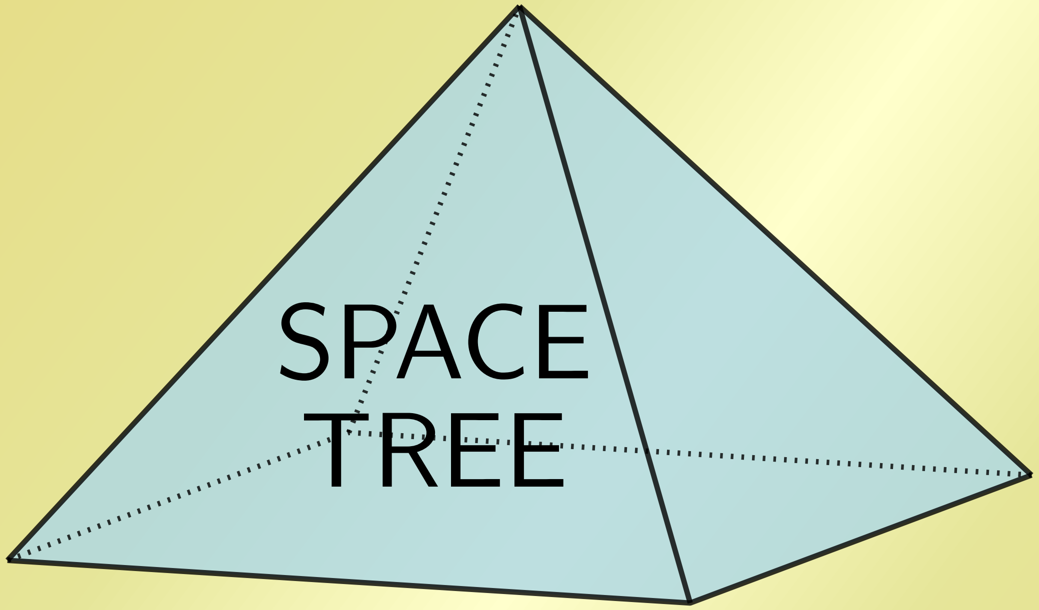


In $\mathbb{R}^2$, we would like to use a similar strategy.

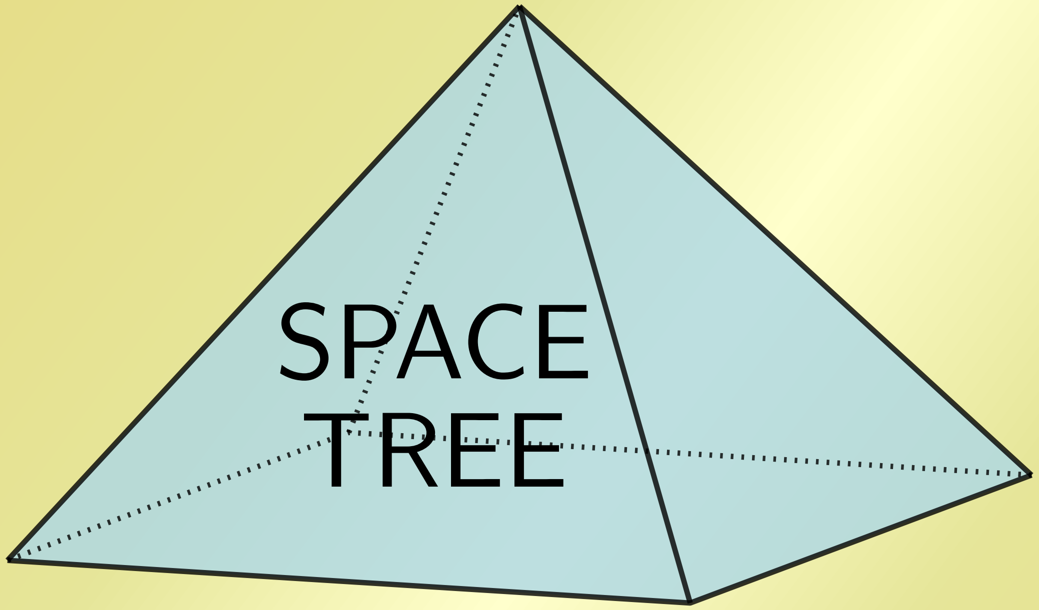


NAIVE THOUGHT

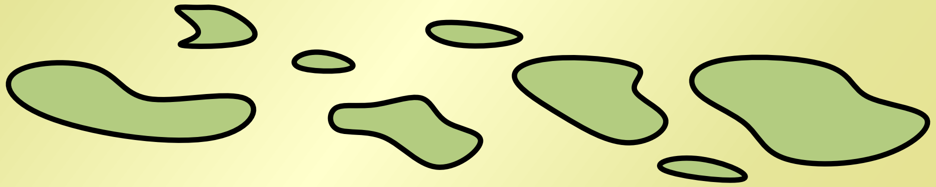
How hard can it be?


SPACE
TREE

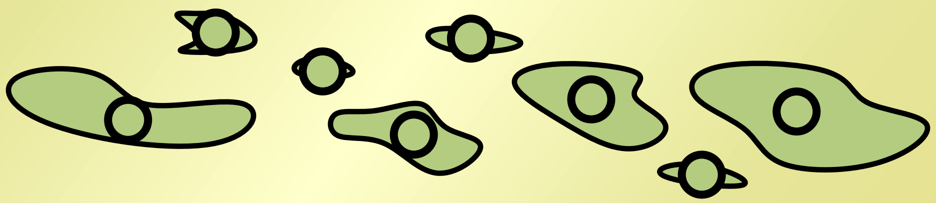
Quadtrees also exist in 2 dimensions!



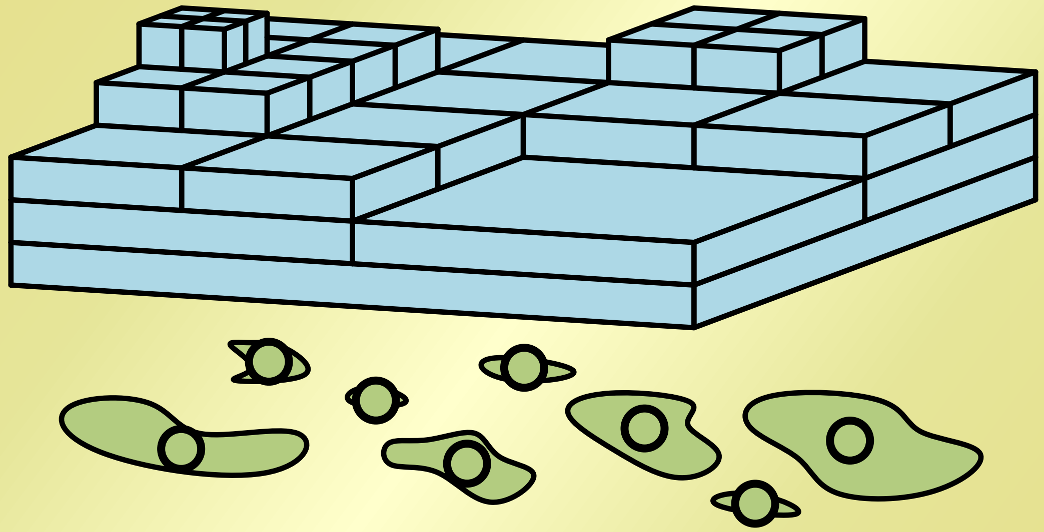
Quadtrees also exist in 2 dimensions!



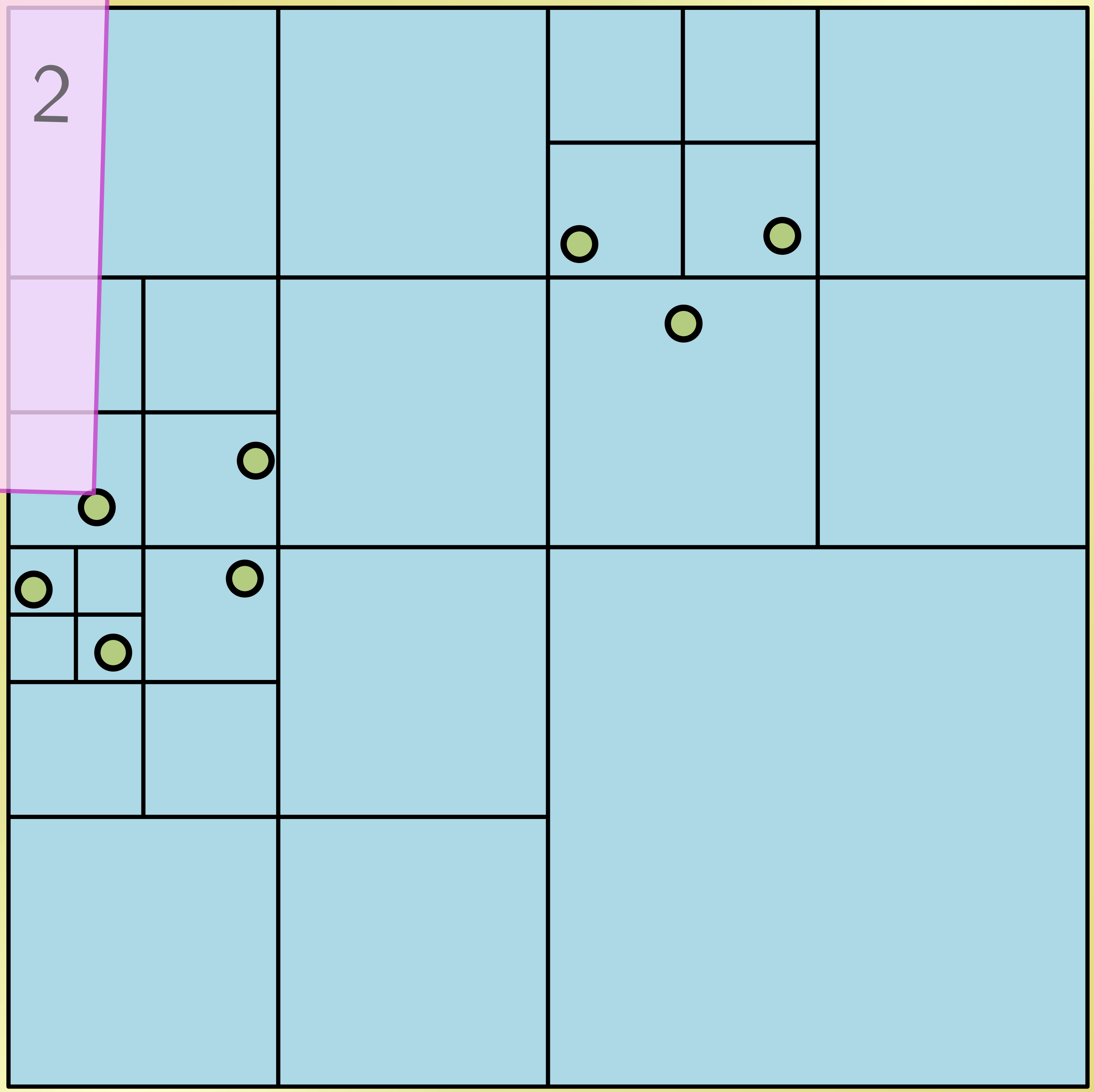
Quadtrees also exist in 2 dimensions!



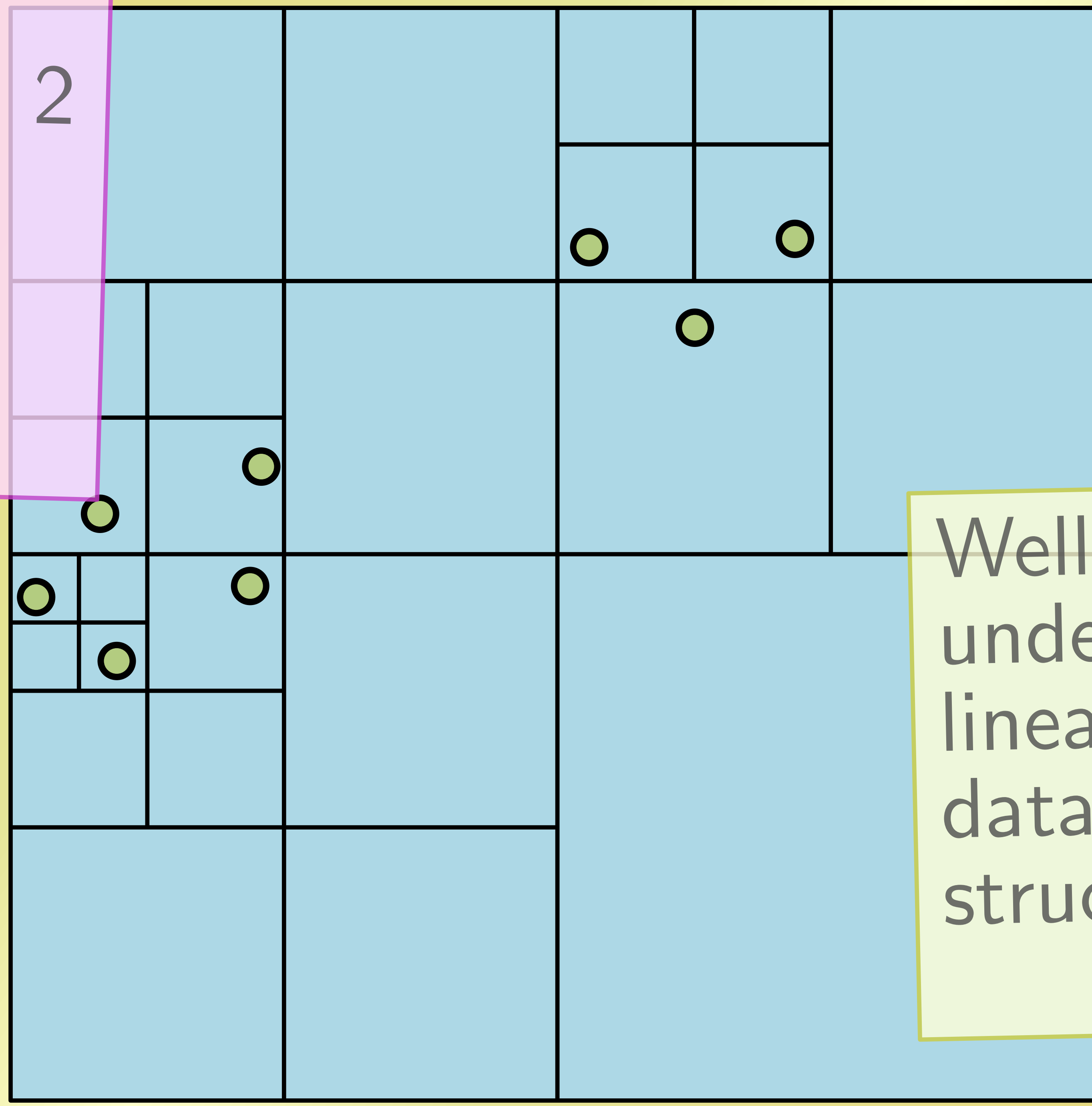
Quadtrees also exist in 2 dimensions!



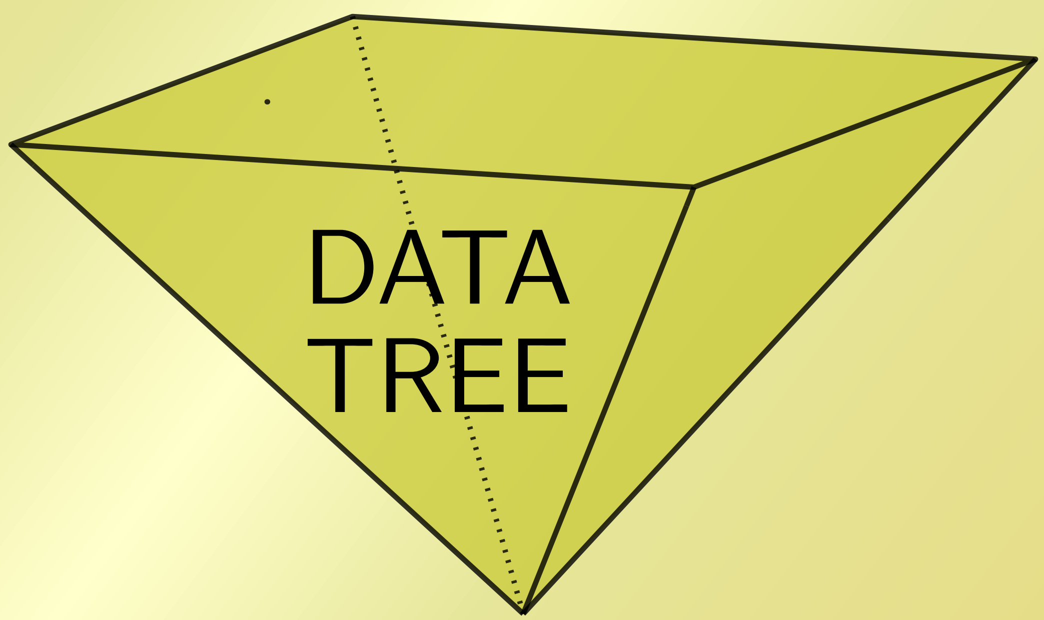
Quadtrees also exist in 2 dimensions!



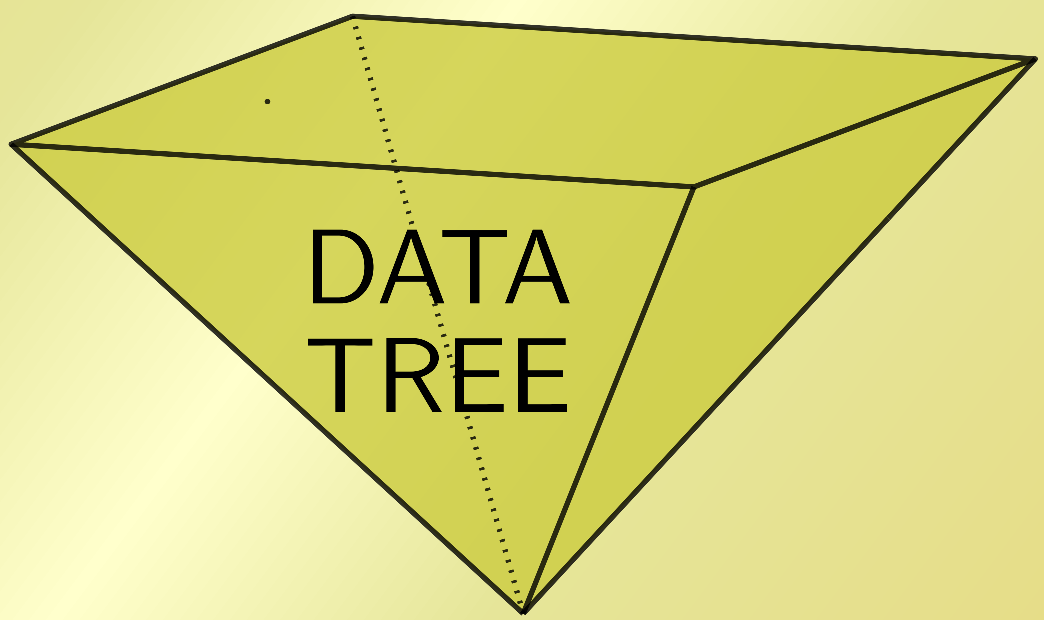
Quadtrees also exist in 2 dimensions!



Well understood, linear size data structure.

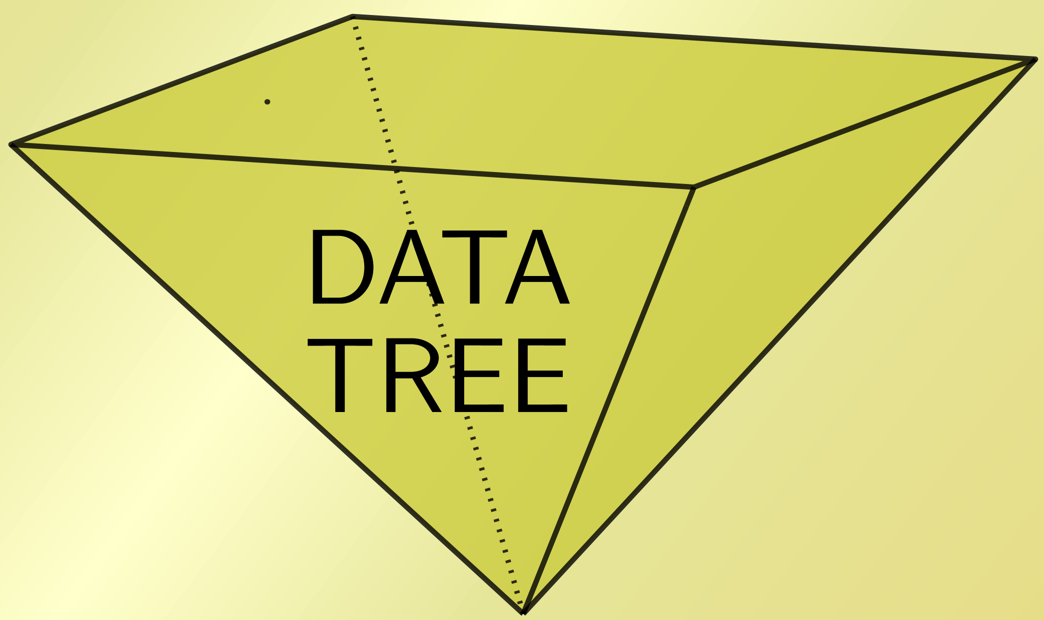

DATA
TREE

We still need something for the actual point location.



We still need something for the actual point location.

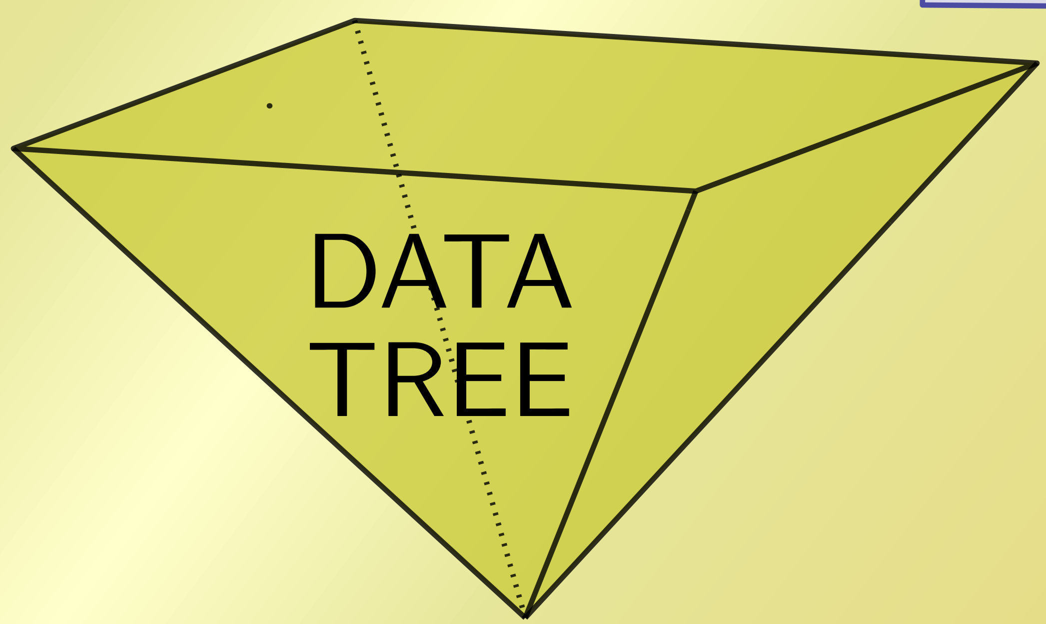
Build existing structure on regions, and use cross pointers as before?



We still need something for the actual point location.

Build existing structure on regions, and use cross pointers as before?

How do regions relate to the quadtree?

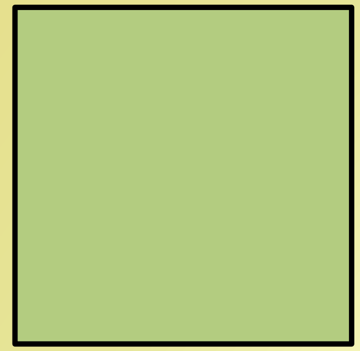


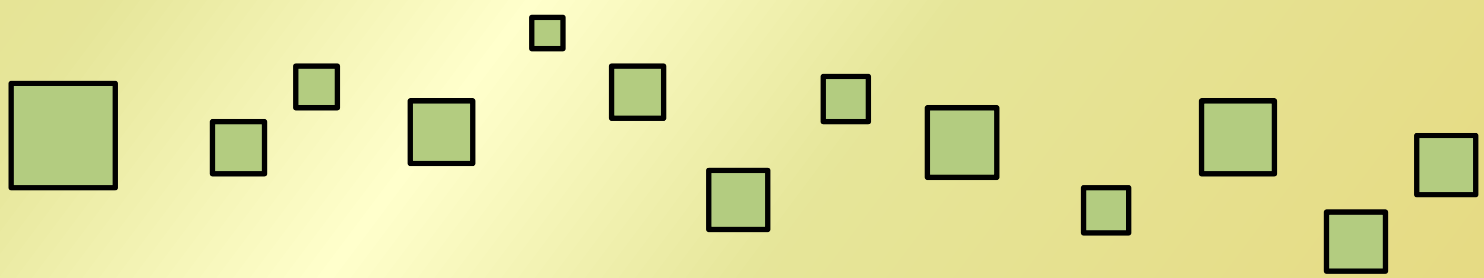
PROBLEM

We can't just use any search tree anymore.

PROBLEM

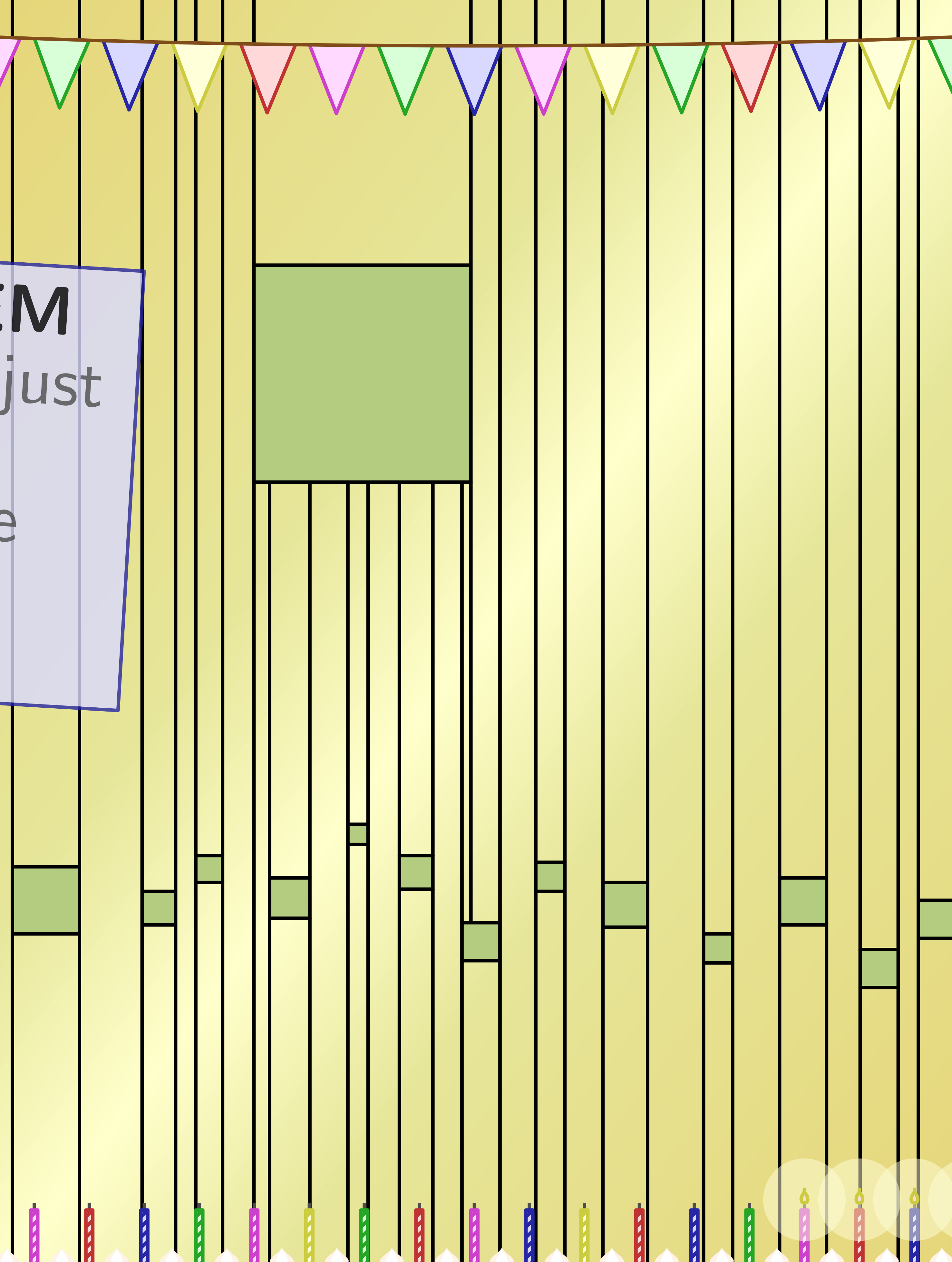
We can't just use any search tree anymore.





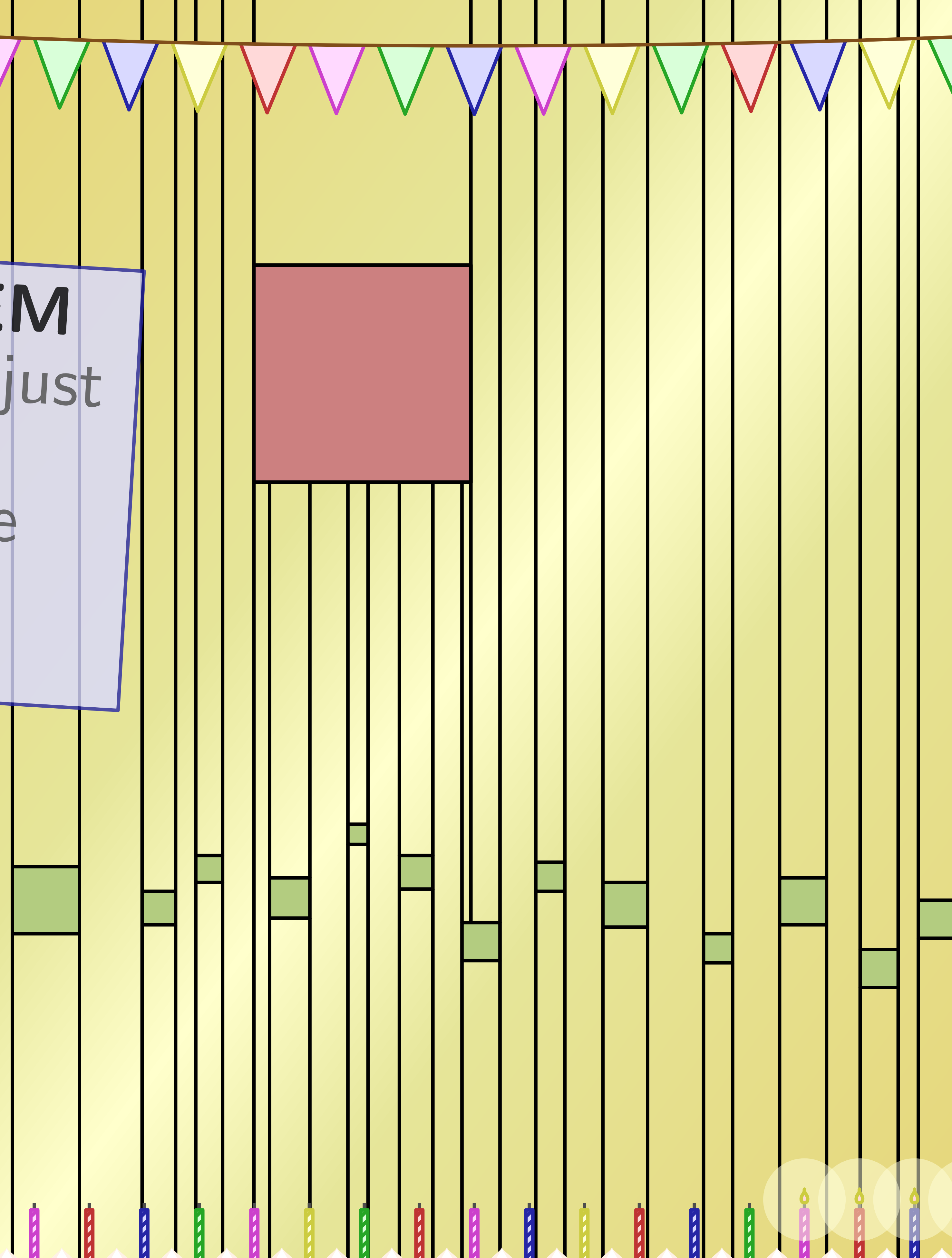
PROBLEM

We can't just use any search tree anymore.



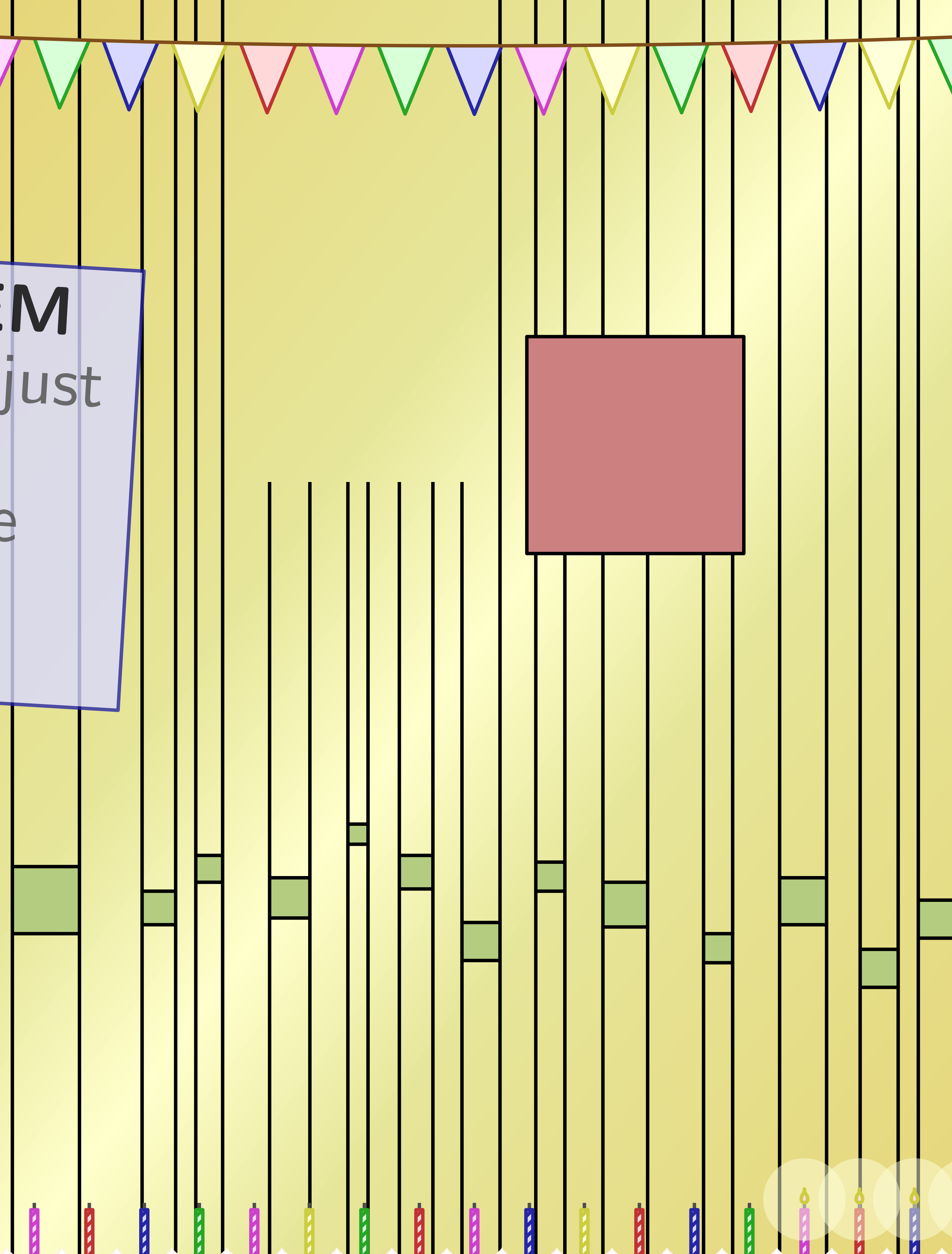
PROBLEM

We can't just use any search tree anymore.



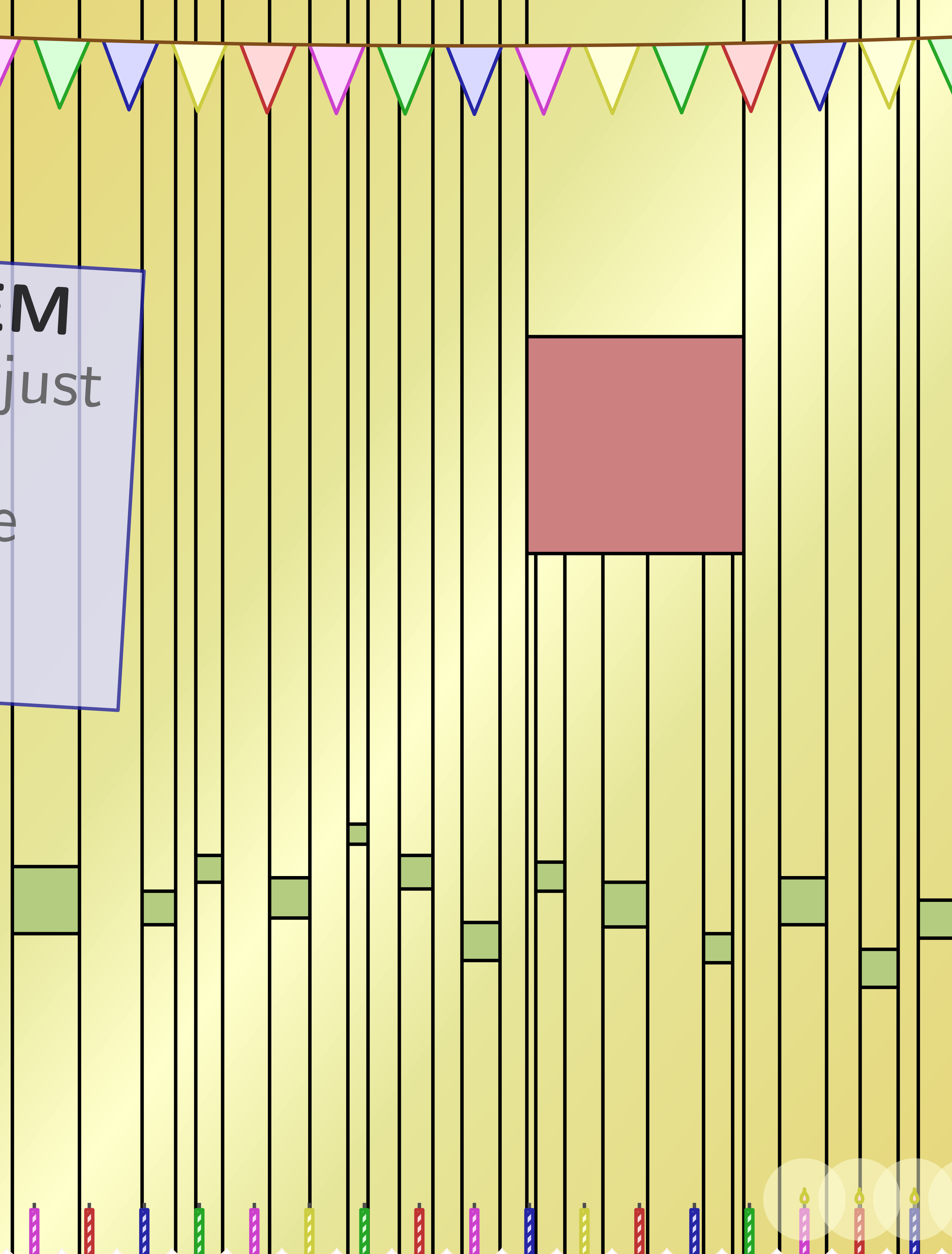
PROBLEM

We can't just use any search tree anymore.



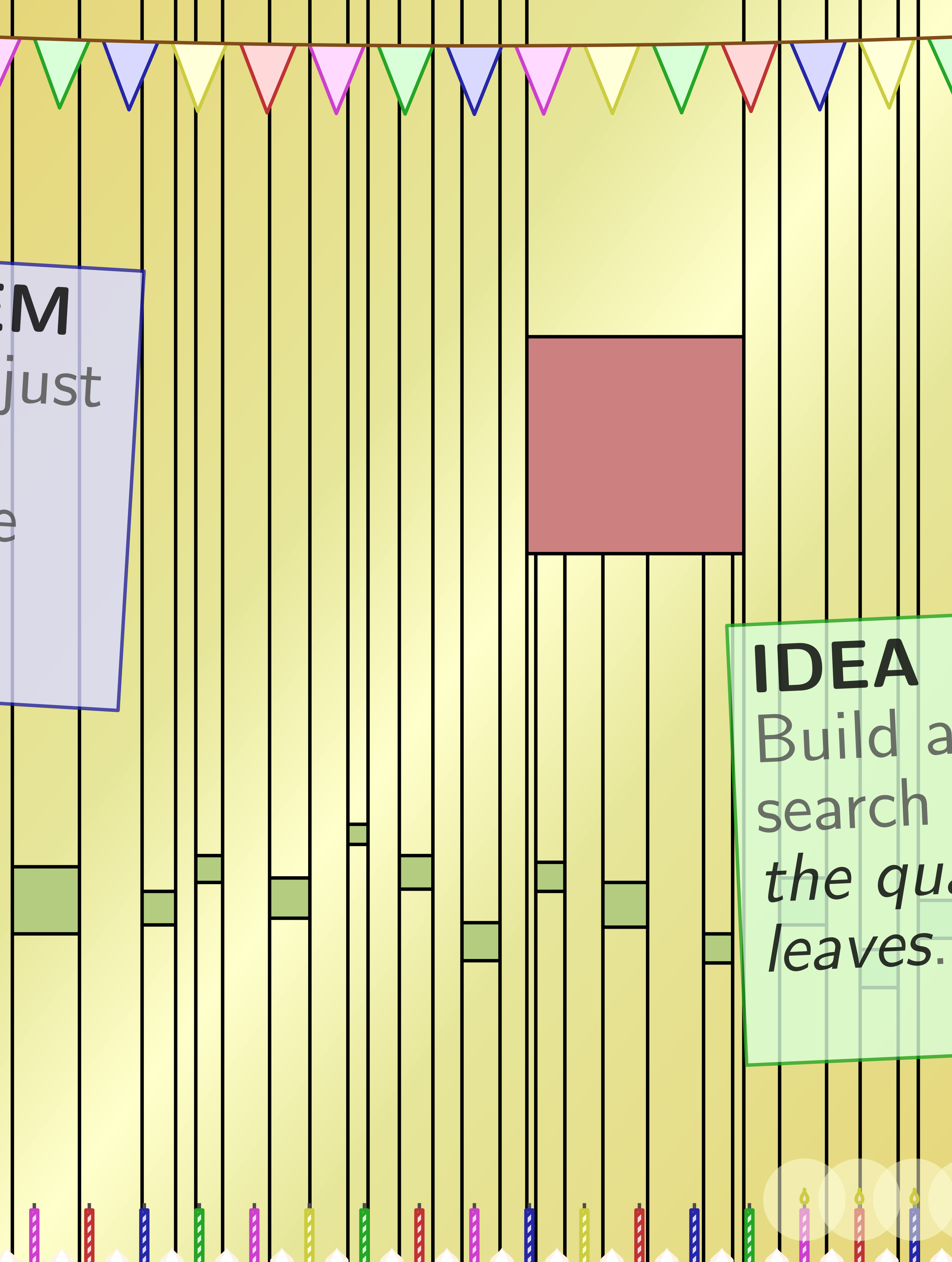
PROBLEM

We can't just use any search tree anymore.



PROBLEM

We can't just use any search tree anymore.

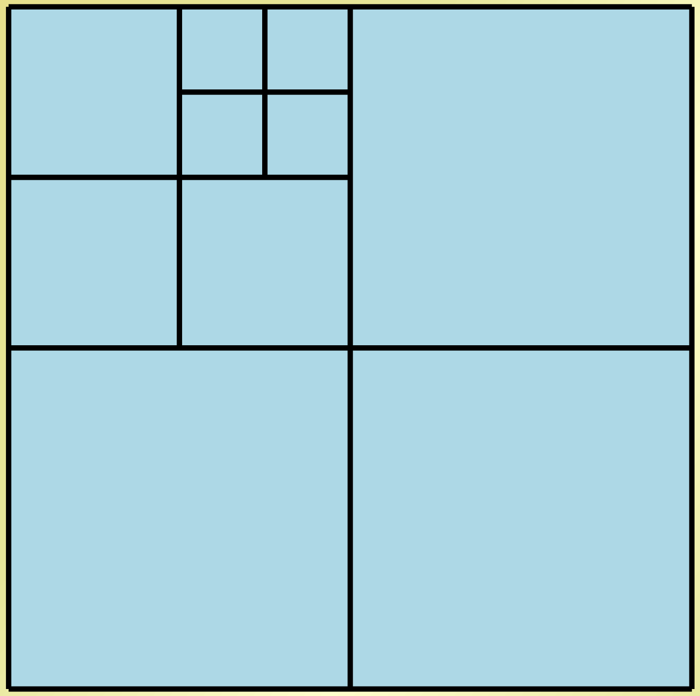


IDEA

Build a search tree *on the quadtree leaves*.

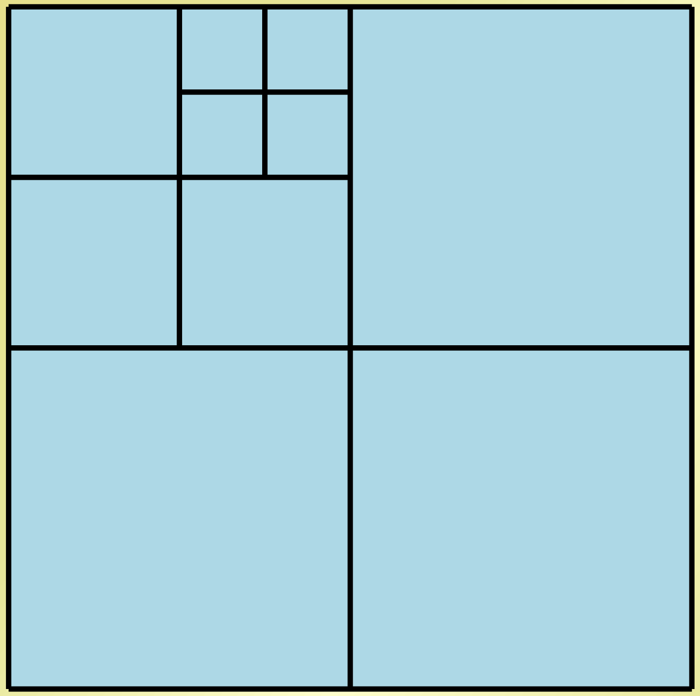
Take another look at a quadtree.

Take another look at a quadtree.

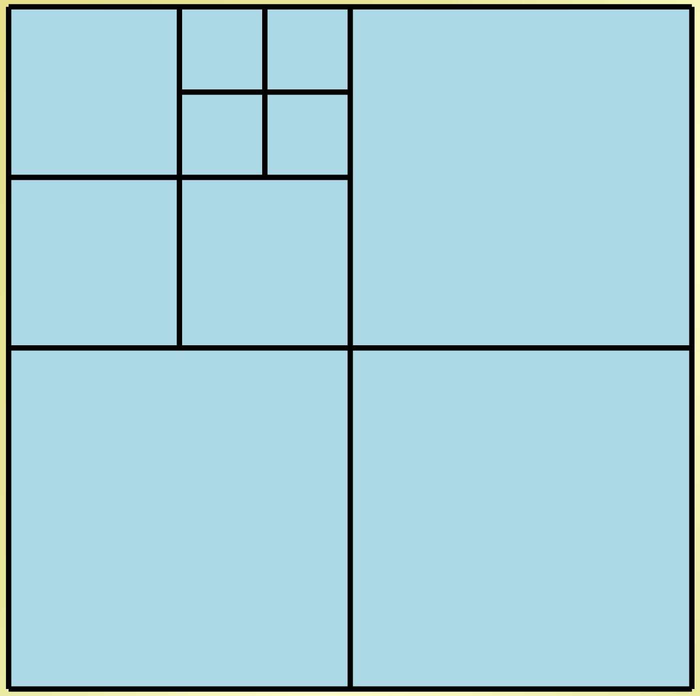


Take another look at a quadtree.

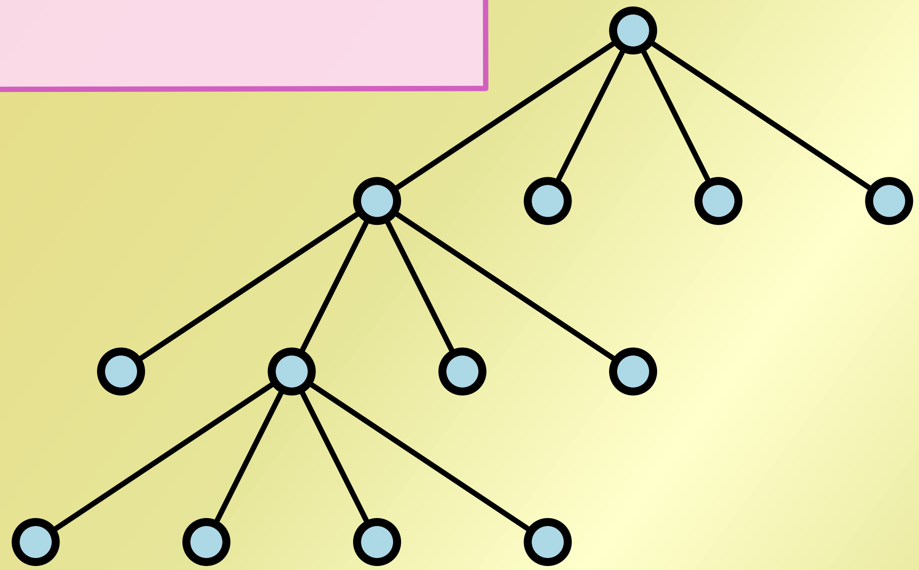
It's a degree 4 tree of potentially linear height.



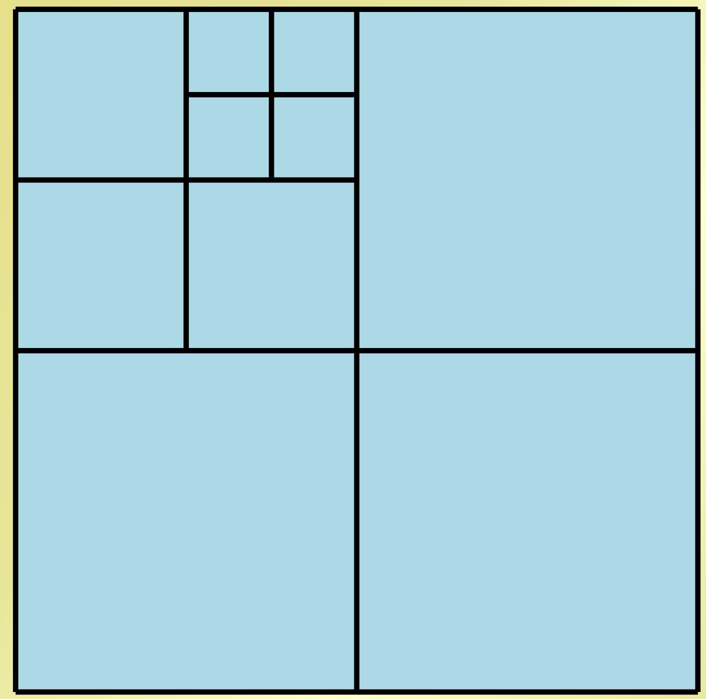
Take another look at a quadtree.



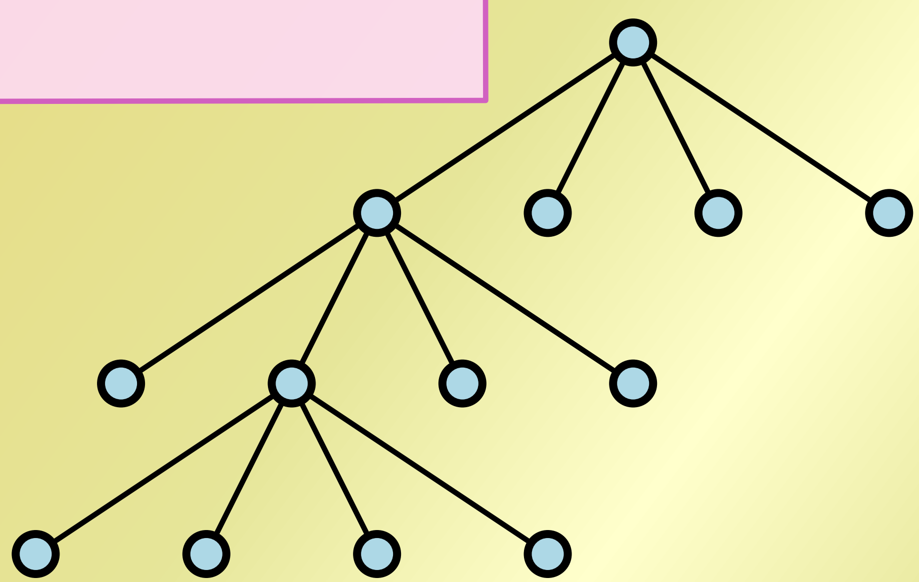
It's a degree 4 tree of potentially linear height.



Take another look at a quadtree.



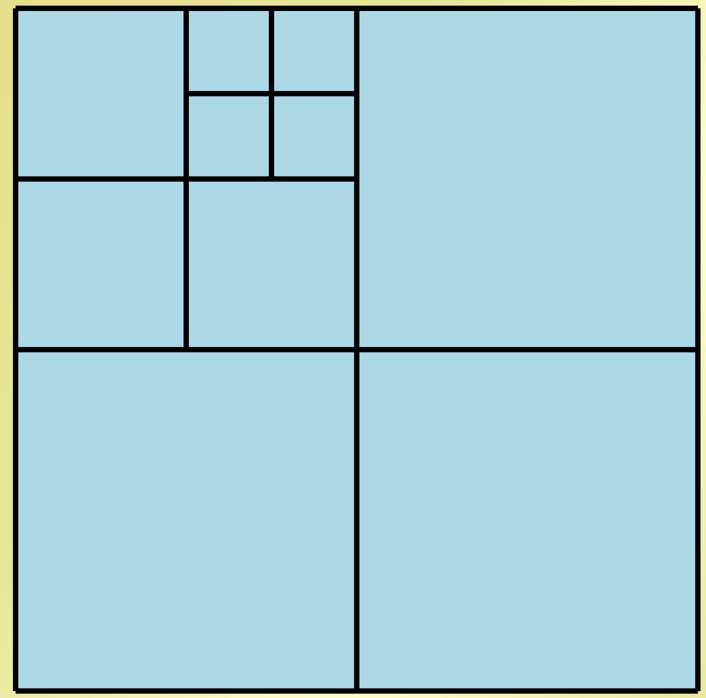
It's a degree 4 tree of potentially linear height.



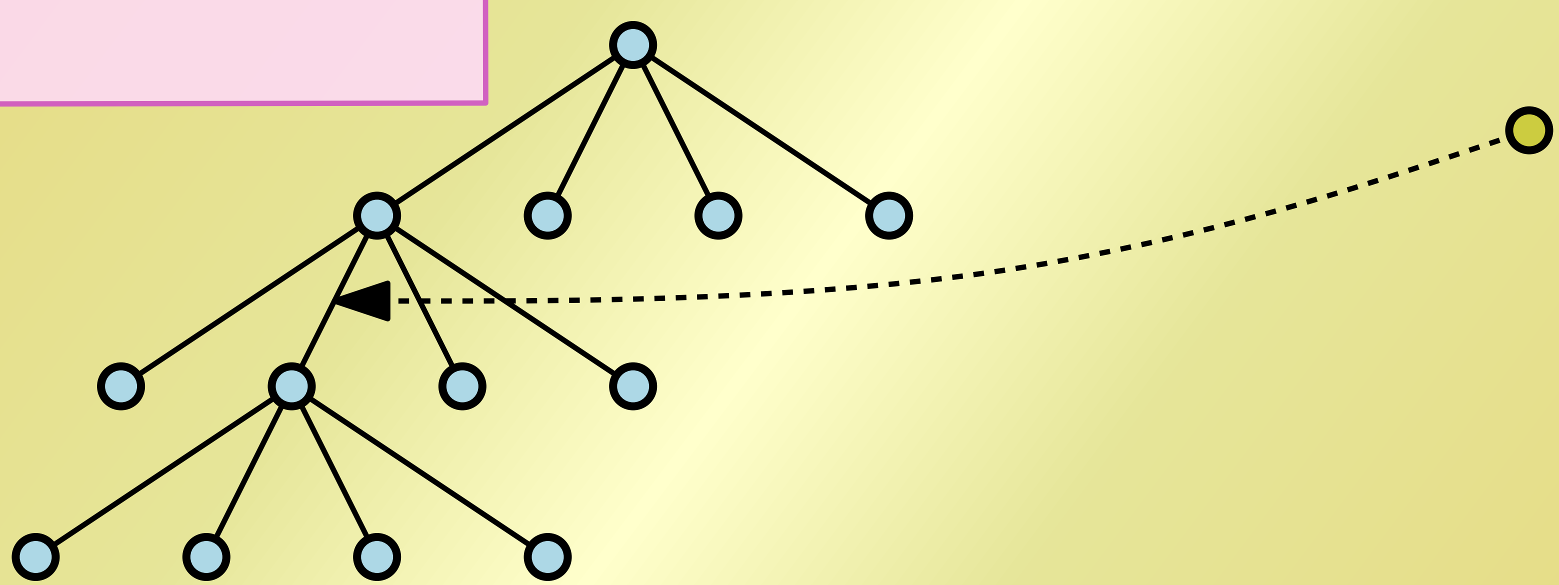
We build another tree, with a vertex for each edge of the quadtree.

Take another look at a quadtree.

It’s a degree 4 tree of potentially linear height.

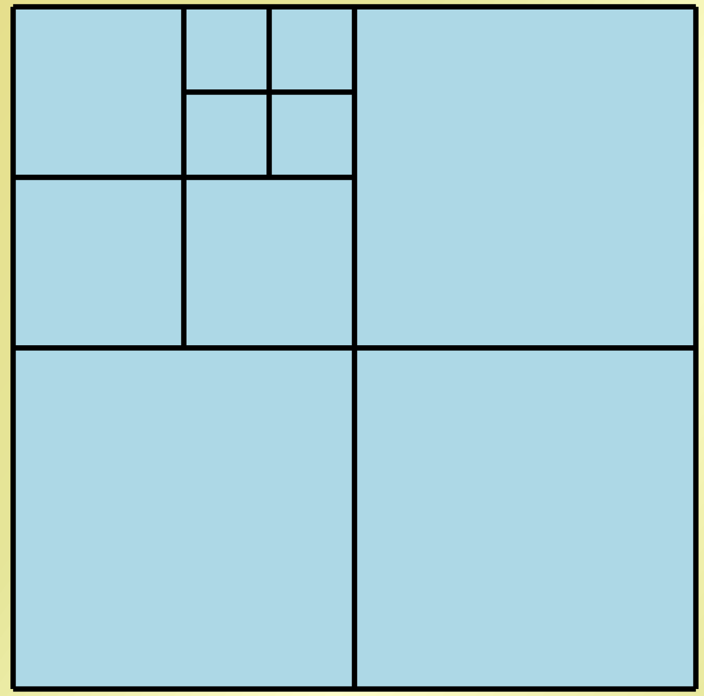


We build another tree, with a vertex for each edge of the quadtree.

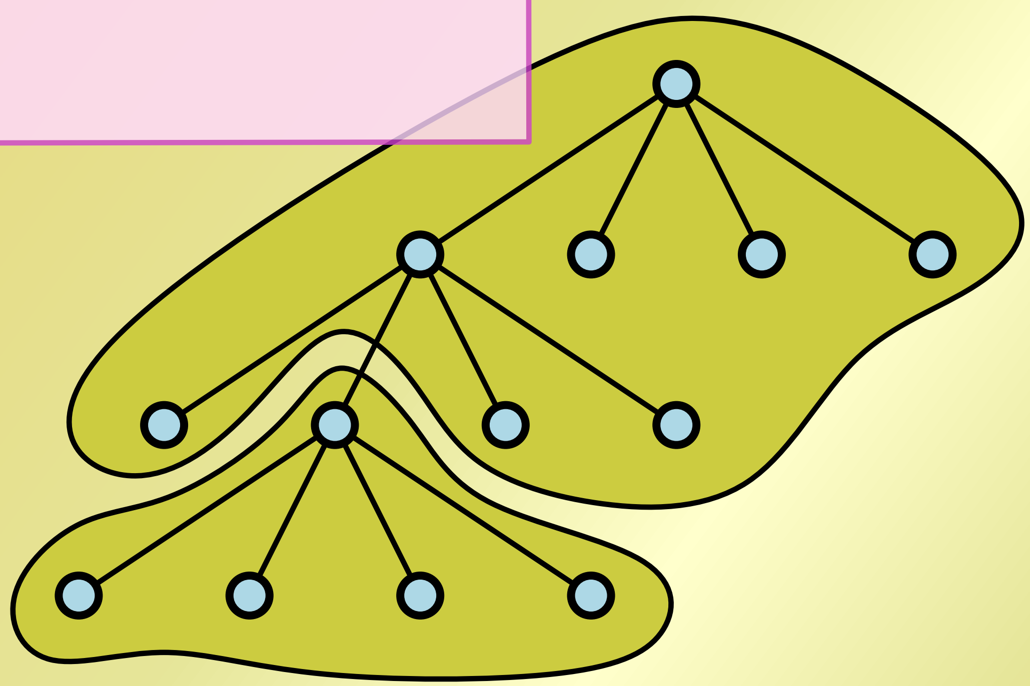


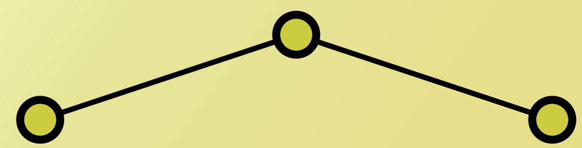
Take another look at a quadtree.

It's a degree 4 tree of potentially linear height.



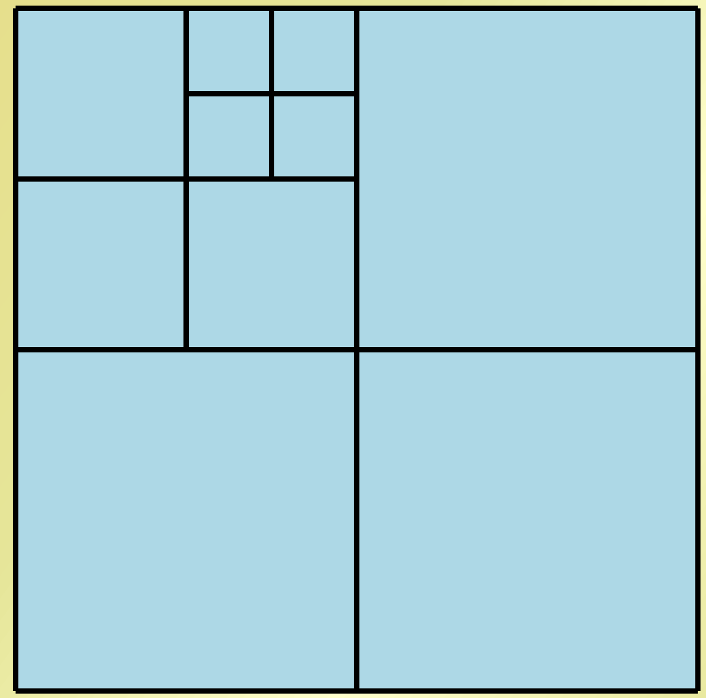
We build another tree, with a vertex for each edge of the quadtree.



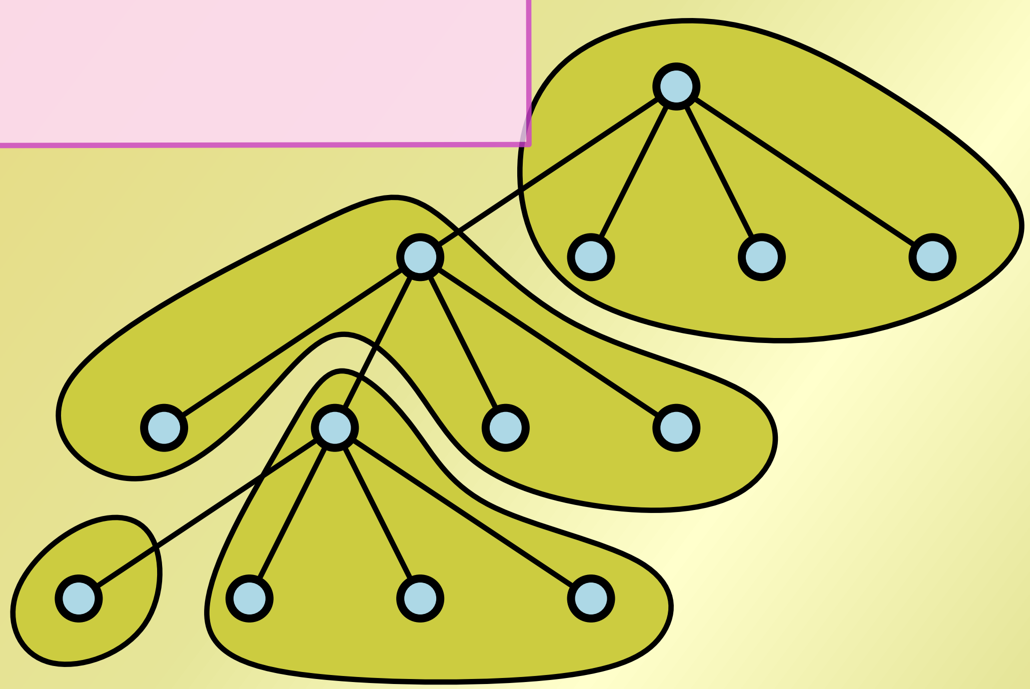


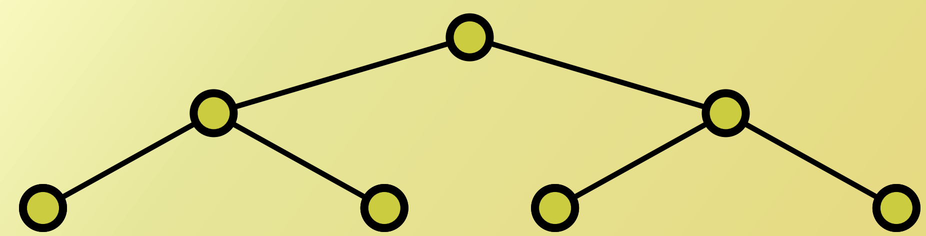
Take another look at a quadtree.

It's a degree 4 tree of potentially linear height.



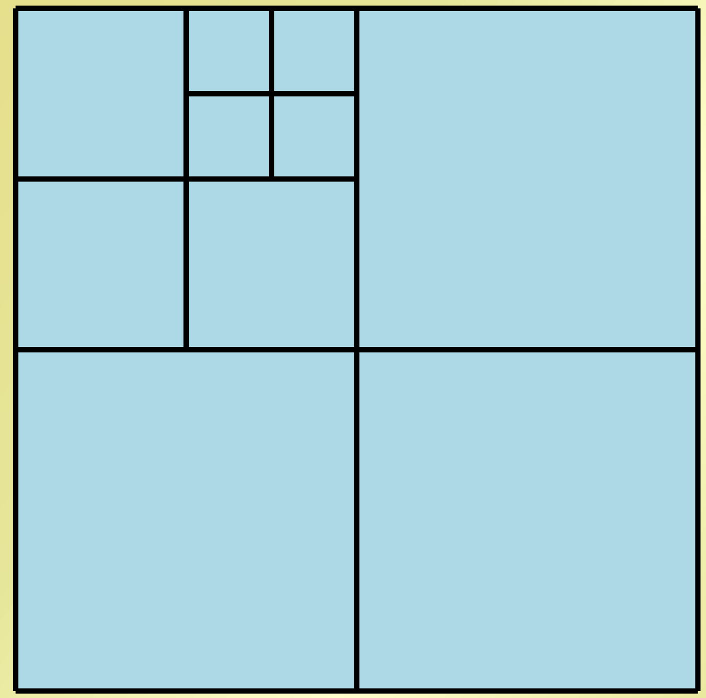
We build another tree, with a vertex for each edge of the quadtree.



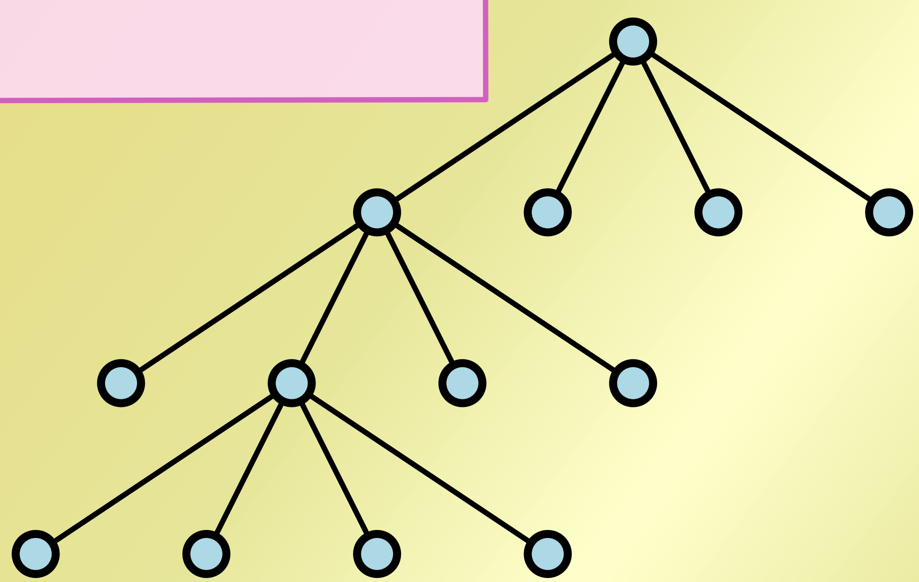


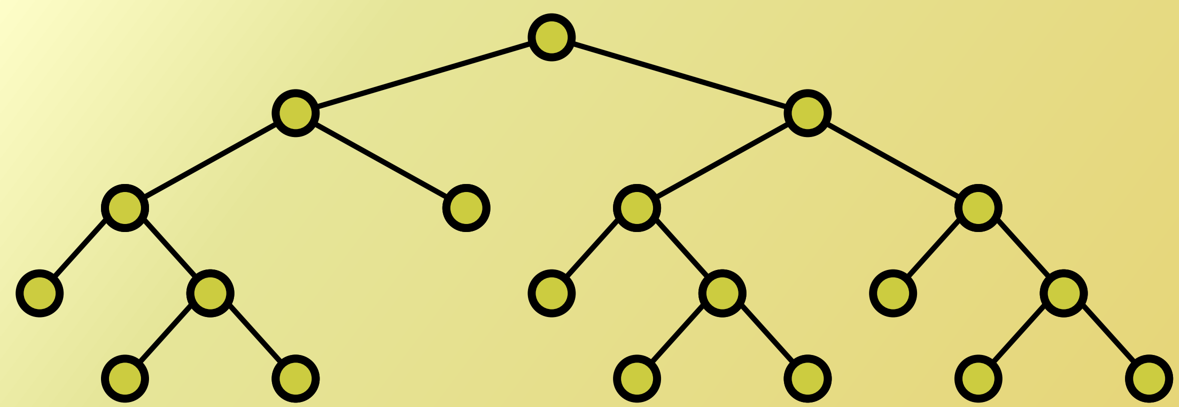
Take another look at a quadtree.

It's a degree 4 tree of potentially linear height.



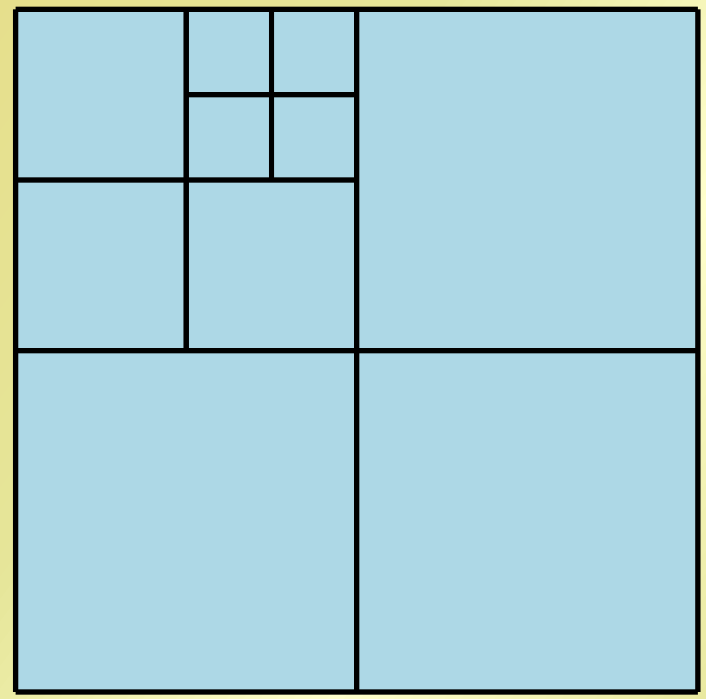
We build another tree, with a vertex for each edge of the quadtree.





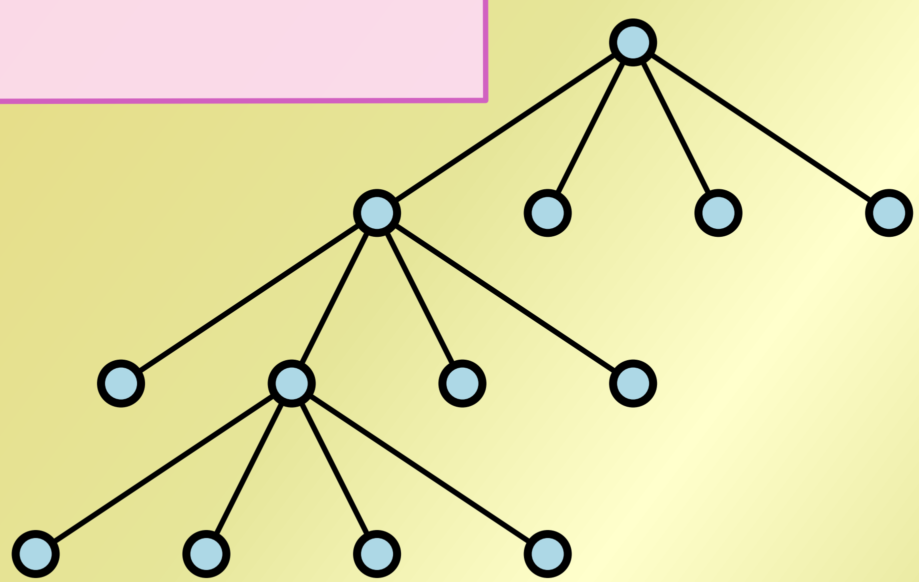
Take another look at a quadtree.

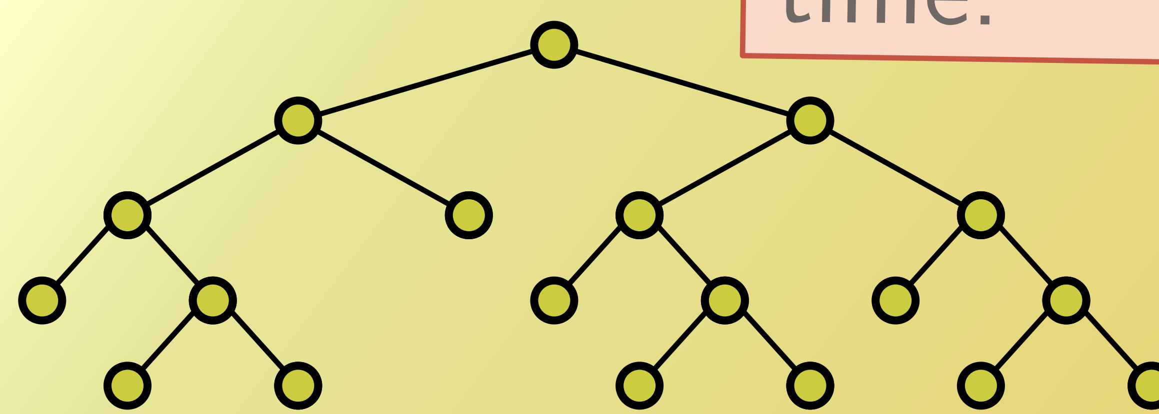
It's a degree 4 tree of potentially linear height.



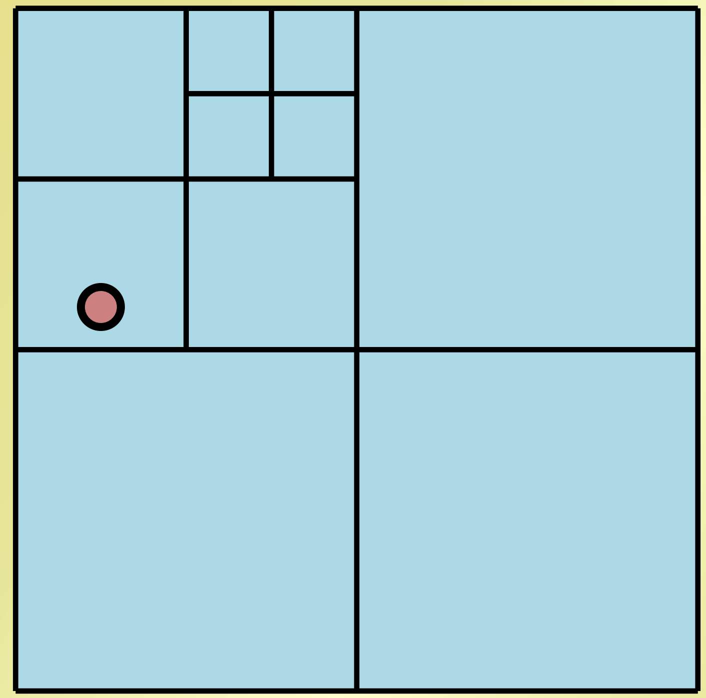
We build another tree, with a vertex for each edge of the quadtree.

Now we can locate points in the quadtree in $O(\log n)$ time.





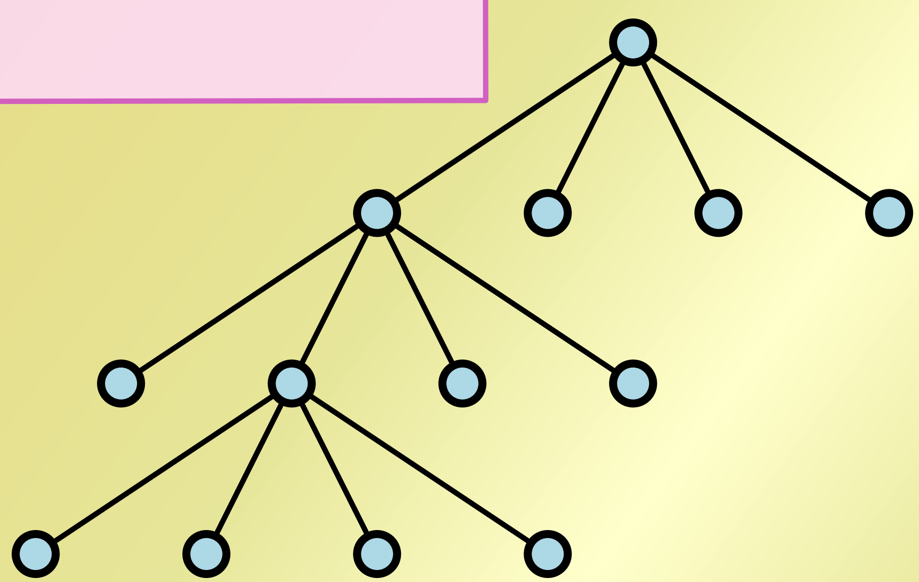
Take another look at a quadtree.

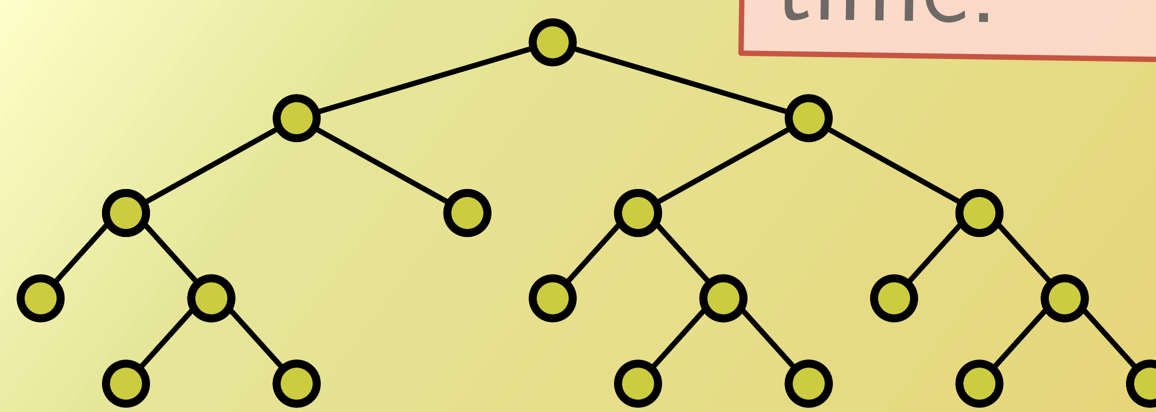


It's a degree 4 tree of potentially linear height.

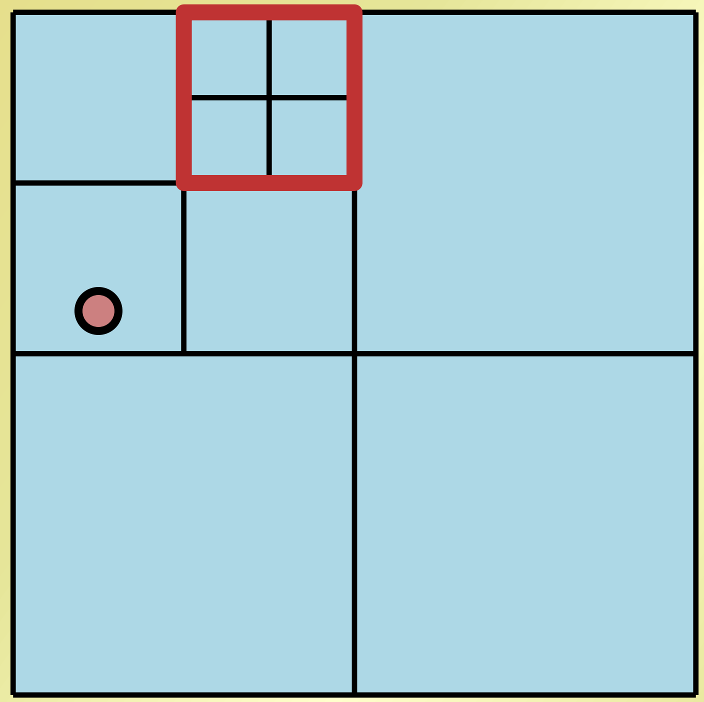
We build another tree, with a vertex for each edge of the quadtree.

Now we can locate points in the quadtree in $O(\log n)$ time.





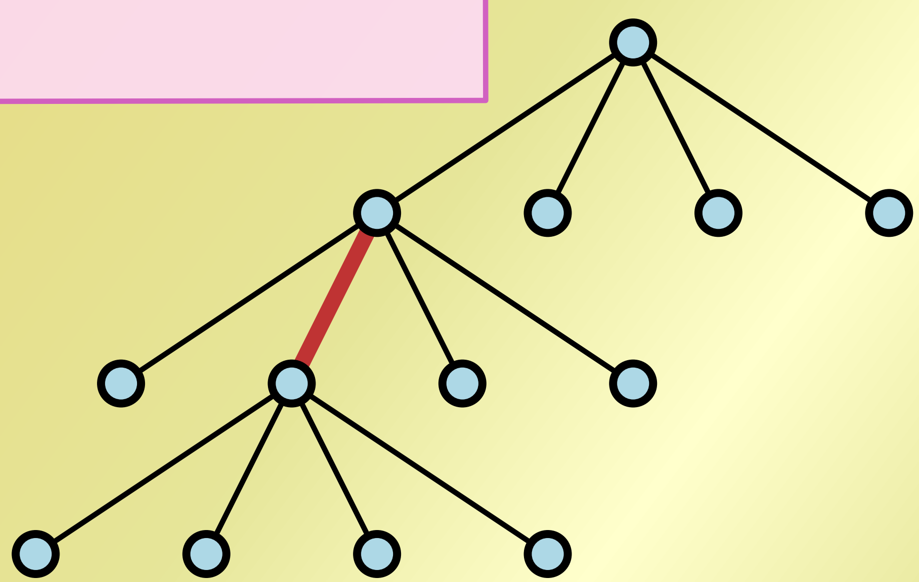
Take another look at a quadtree.

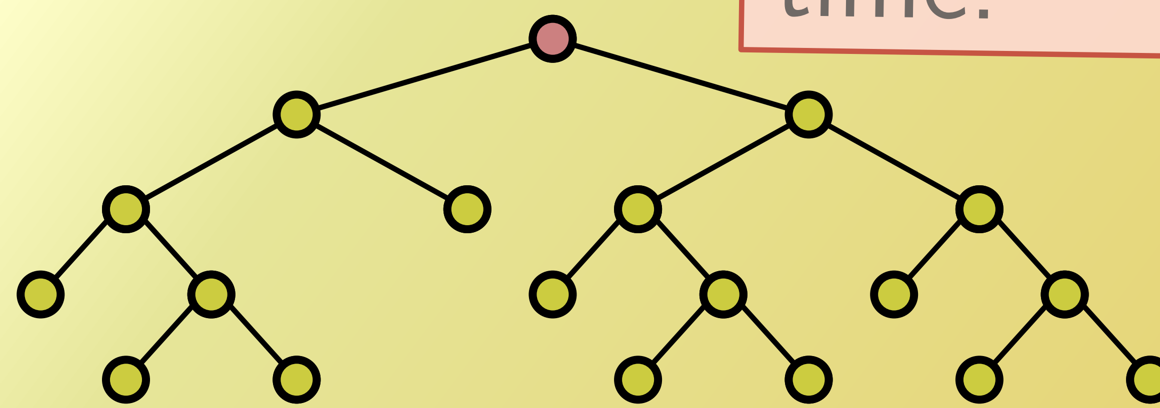


We build another tree, with a vertex for each edge of the quadtree.

It's a degree 4 tree of potentially linear height.

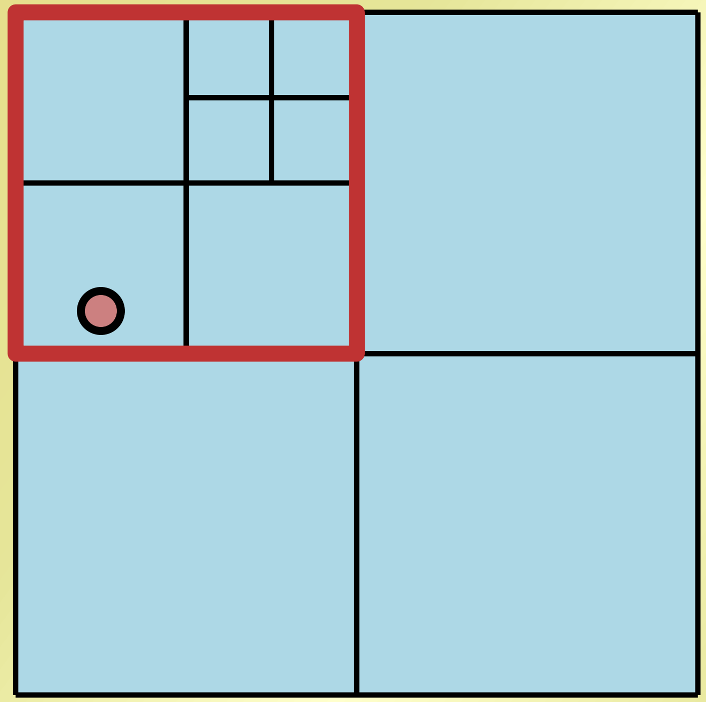
Now we can locate points in the quadtree in $O(\log n)$ time.





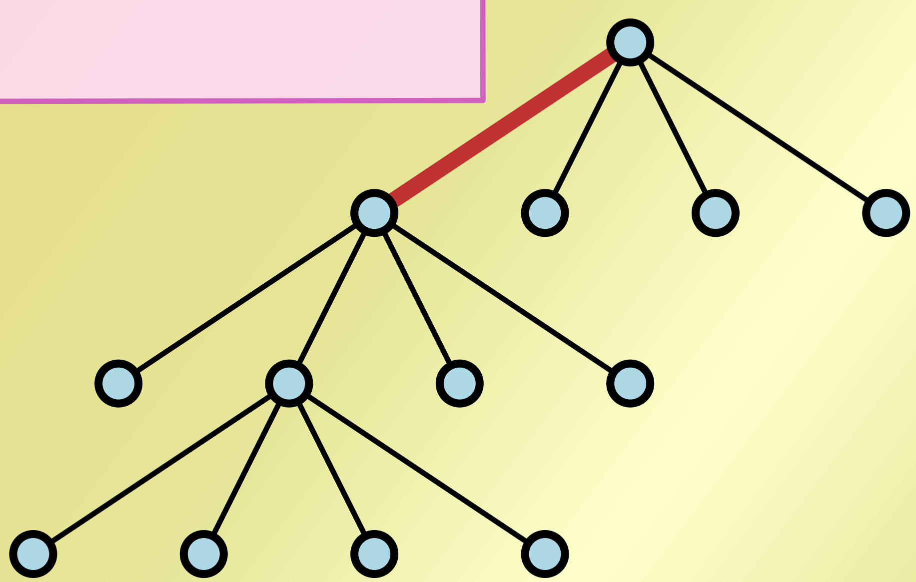
Take another look at a quadtree.

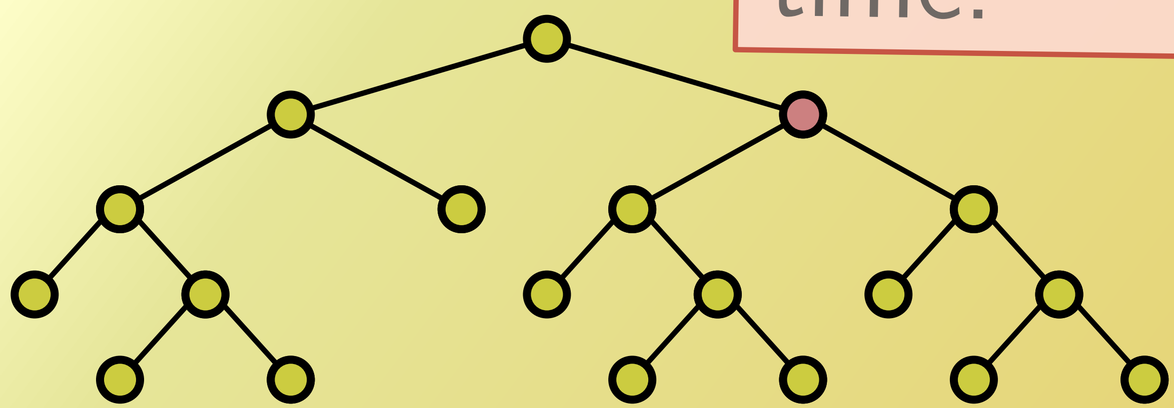
It's a degree 4 tree of potentially linear height.



We build another tree, with a vertex for each edge of the quadtree.

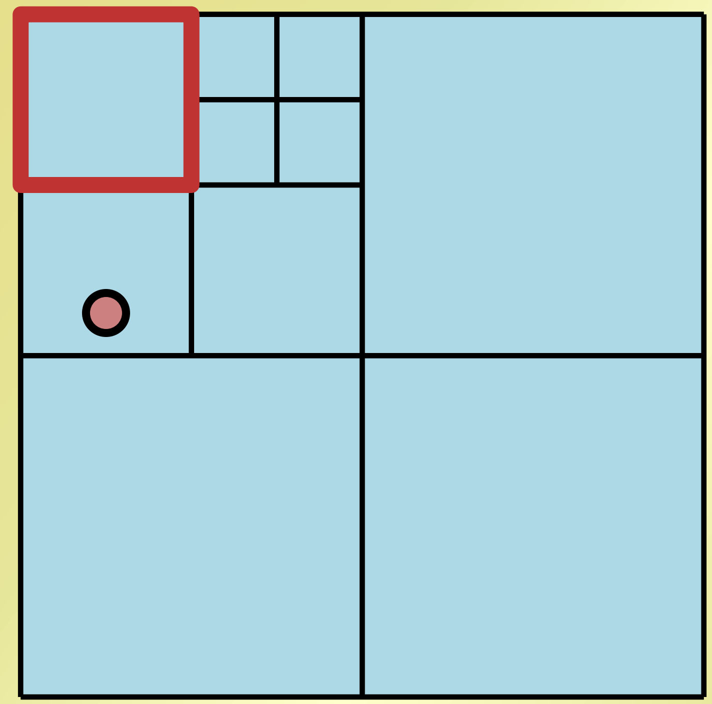
Now we can locate points in the quadtree in $O(\log n)$ time.





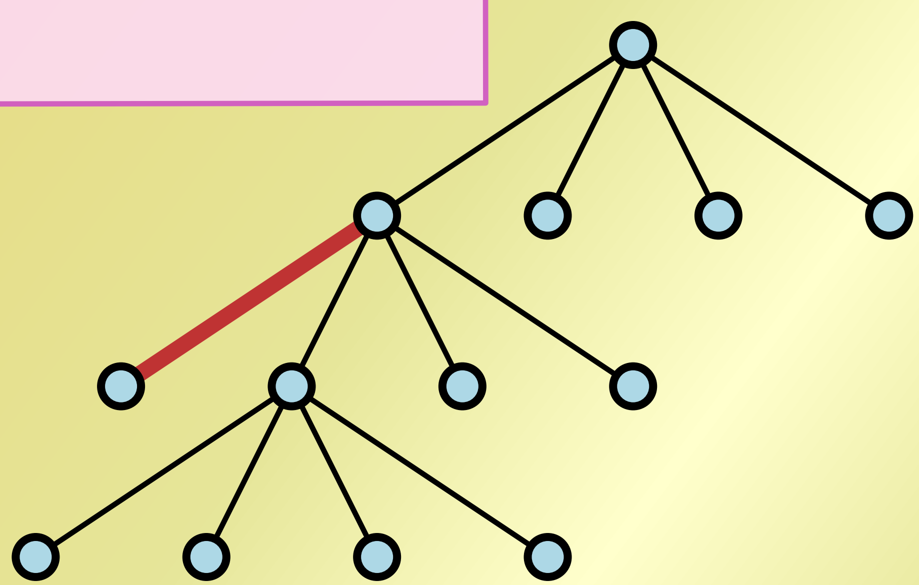
Take another look at a quadtree.

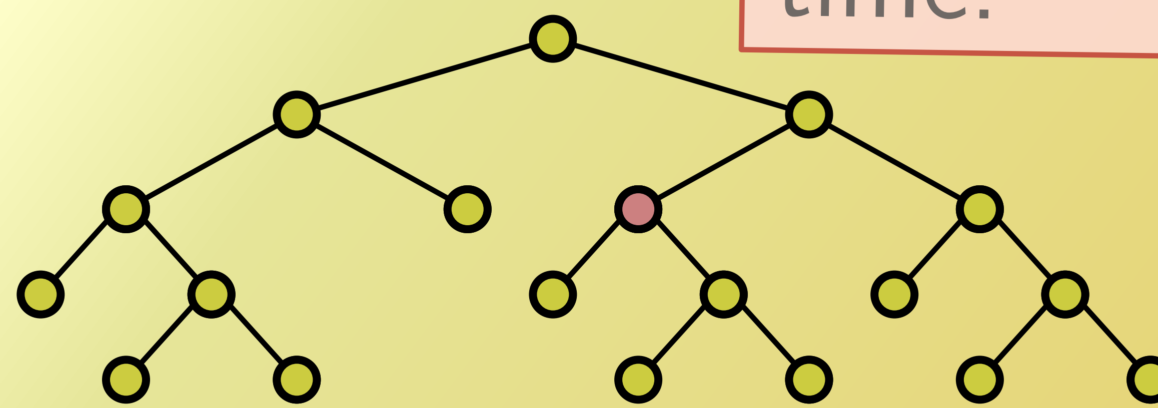
It's a degree 4 tree of potentially linear height.



We build another tree, with a vertex for each edge of the quadtree.

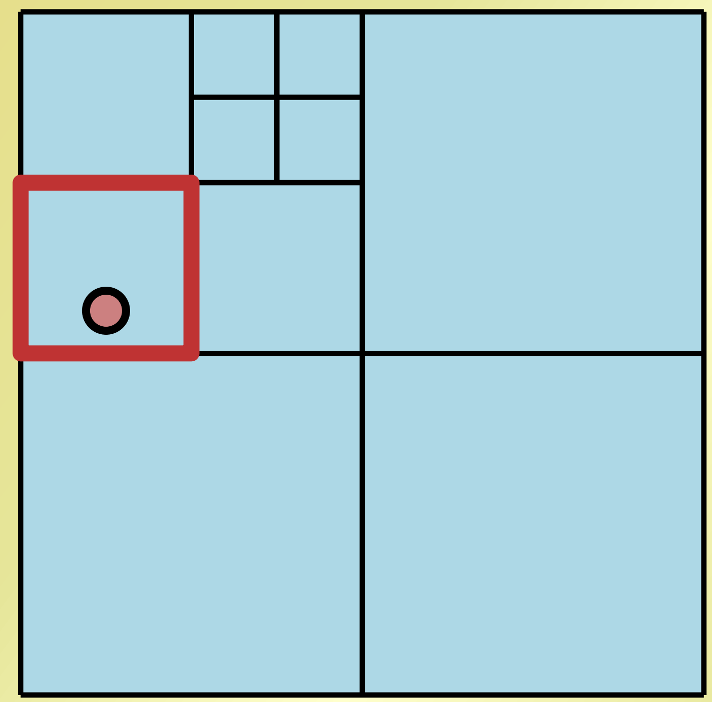
Now we can locate points in the quadtree in $O(\log n)$ time.





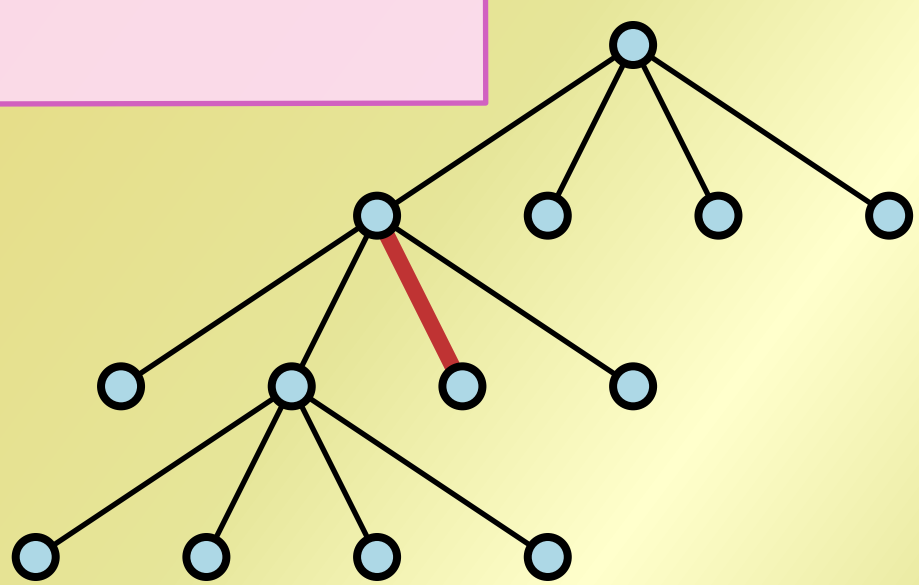
Take another look at a quadtree.

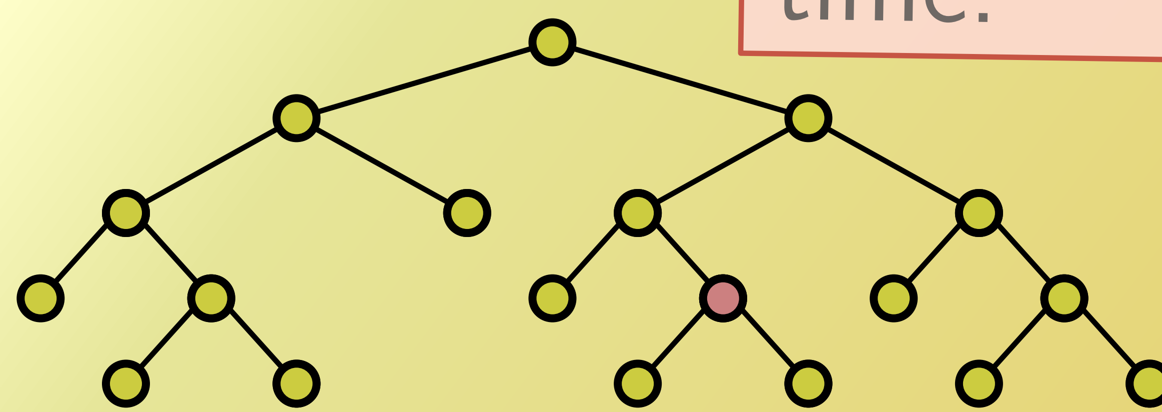
It's a degree 4 tree of potentially linear height.



We build another tree, with a vertex for each edge of the quadtree.

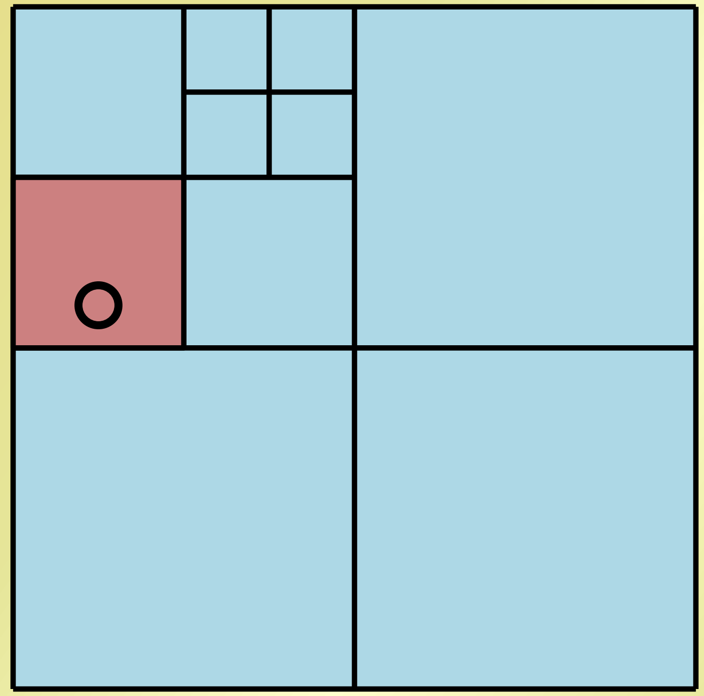
Now we can locate points in the quadtree in $O(\log n)$ time.





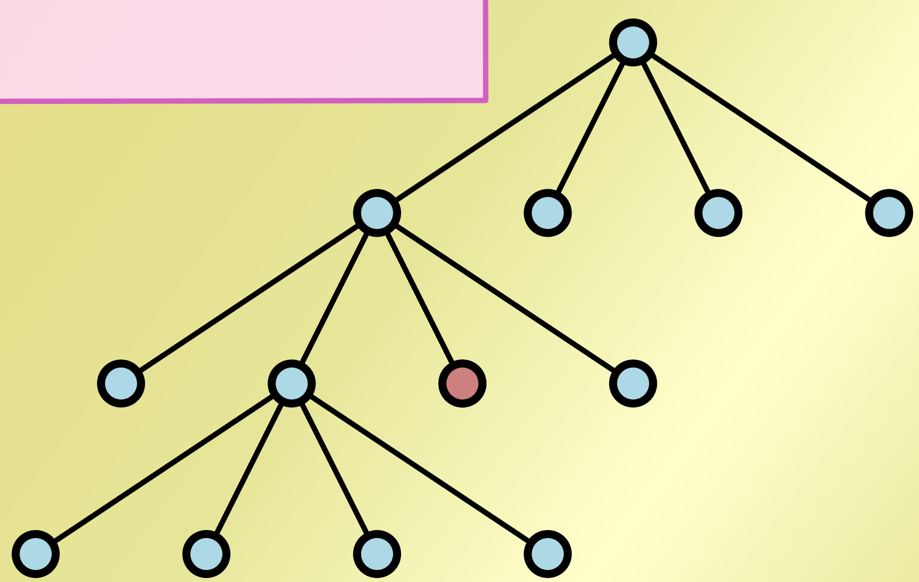
Take another look at a quadtree.

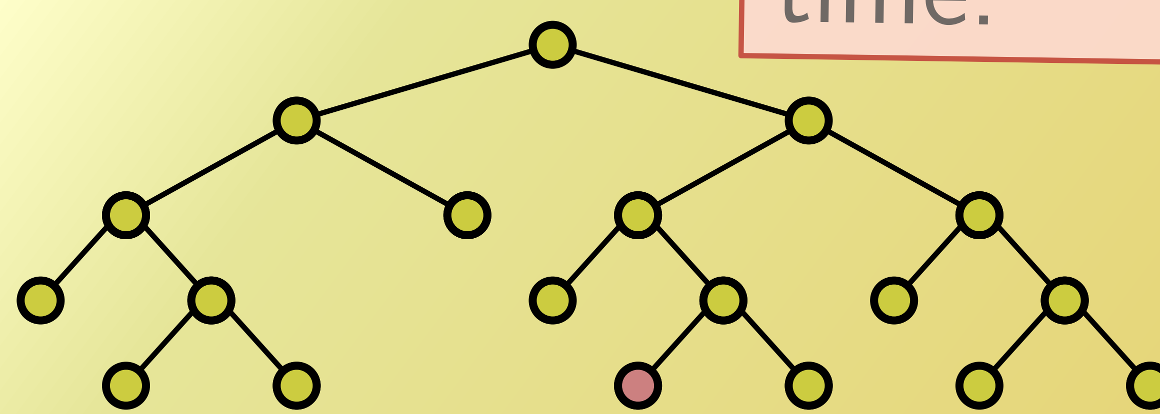
It's a degree 4 tree of potentially linear height.



We build another tree, with a vertex for each edge of the quadtree.

Now we can locate points in the quadtree in $O(\log n)$ time.



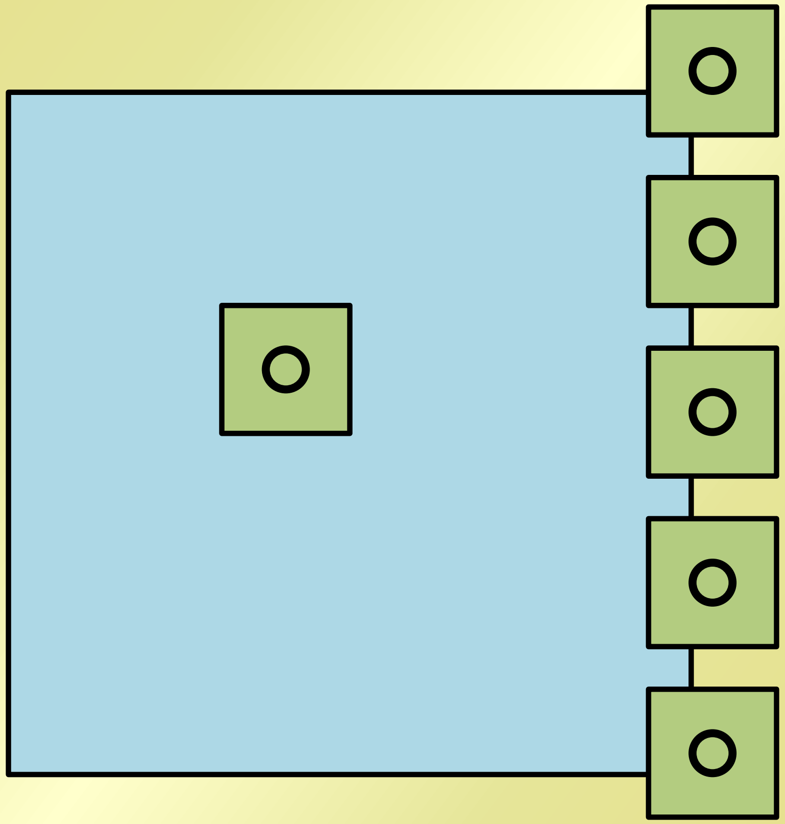


PROBLEM

The number of regions intersecting a quadtree leaf can be linear!

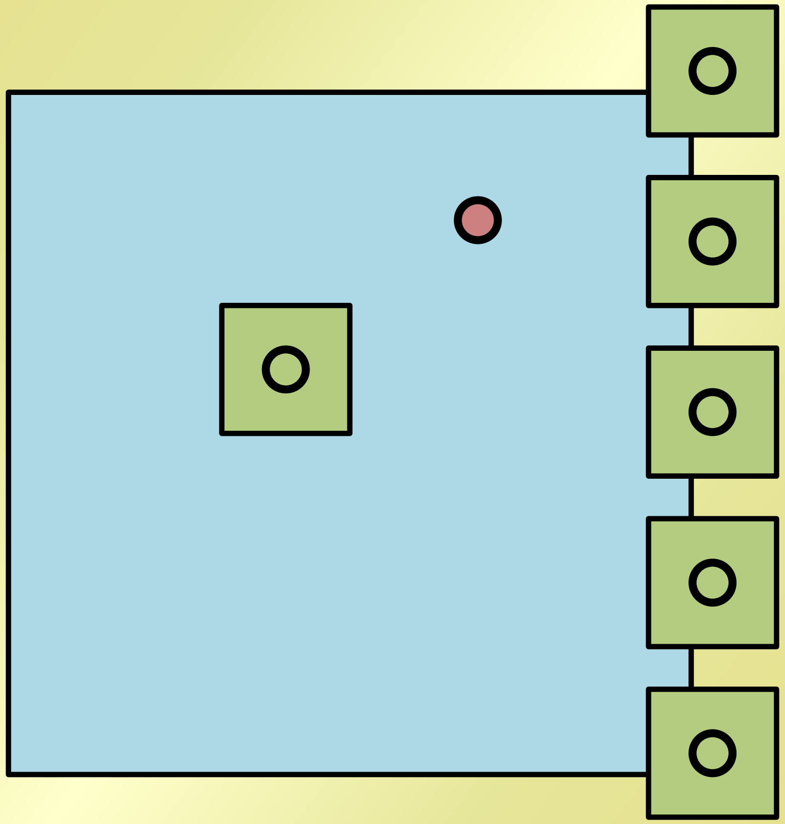
PROBLEM

The number of regions intersecting a quadtree leaf can be linear!



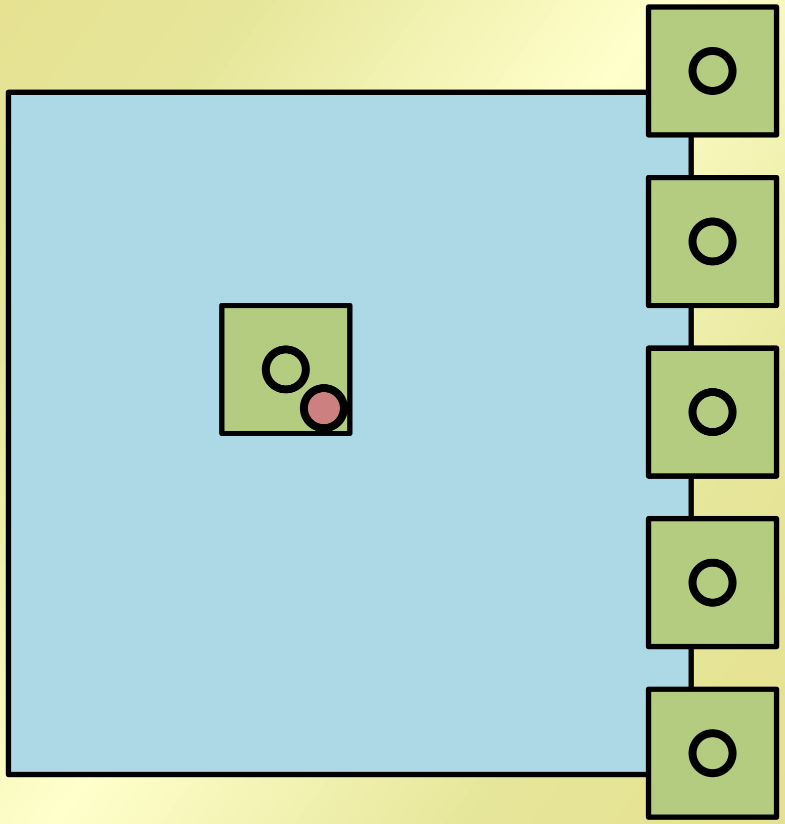
PROBLEM

The number of regions intersecting a quadtree leaf can be linear!



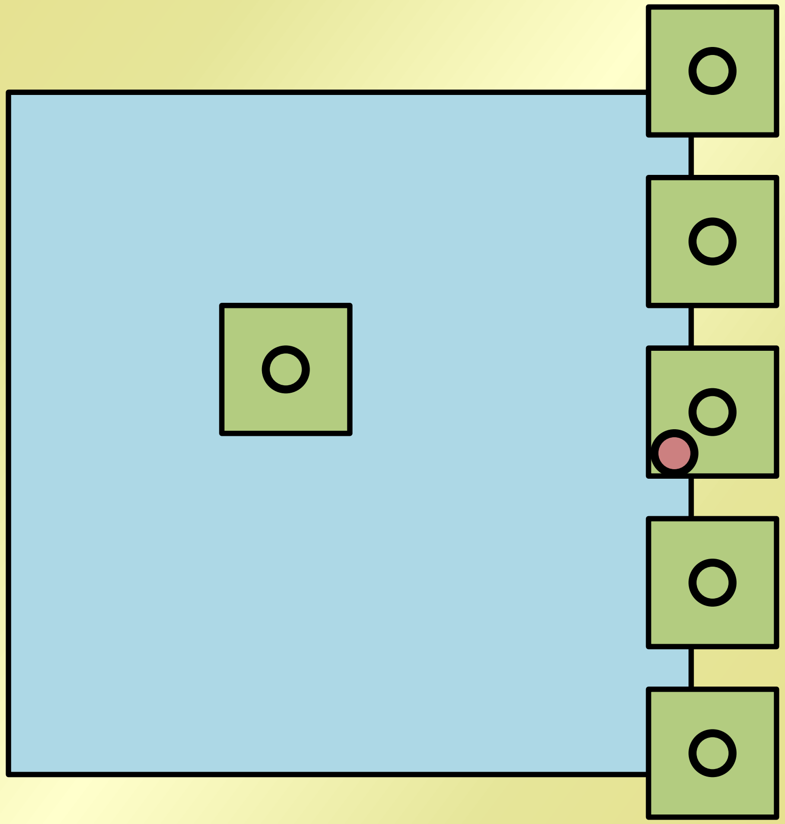
PROBLEM

The number of regions intersecting a quadtree leaf can be linear!



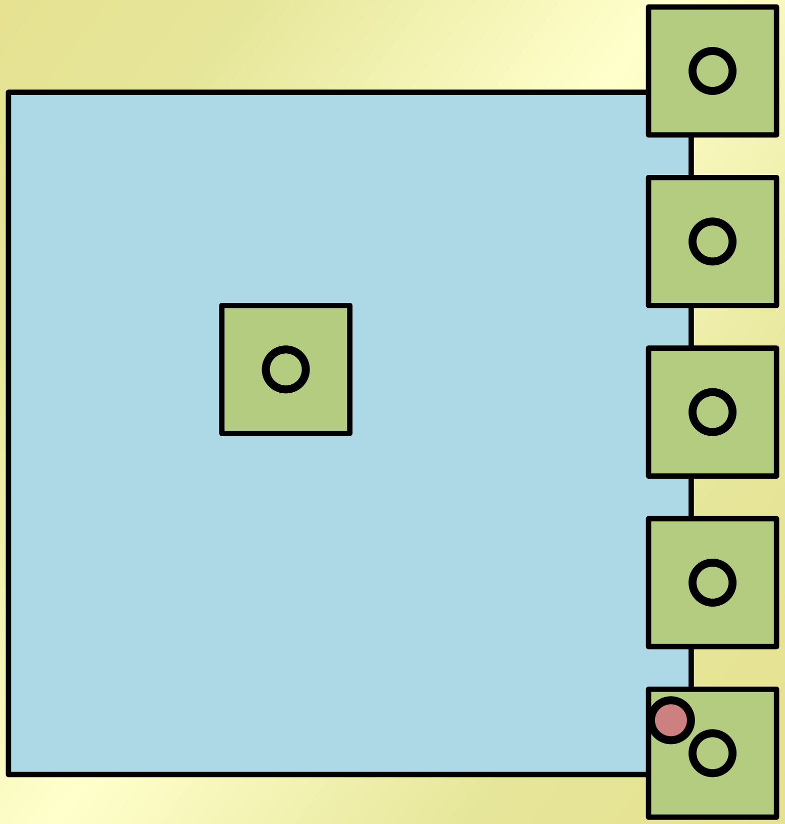
PROBLEM

The number of regions intersecting a quadtree leaf can be linear!



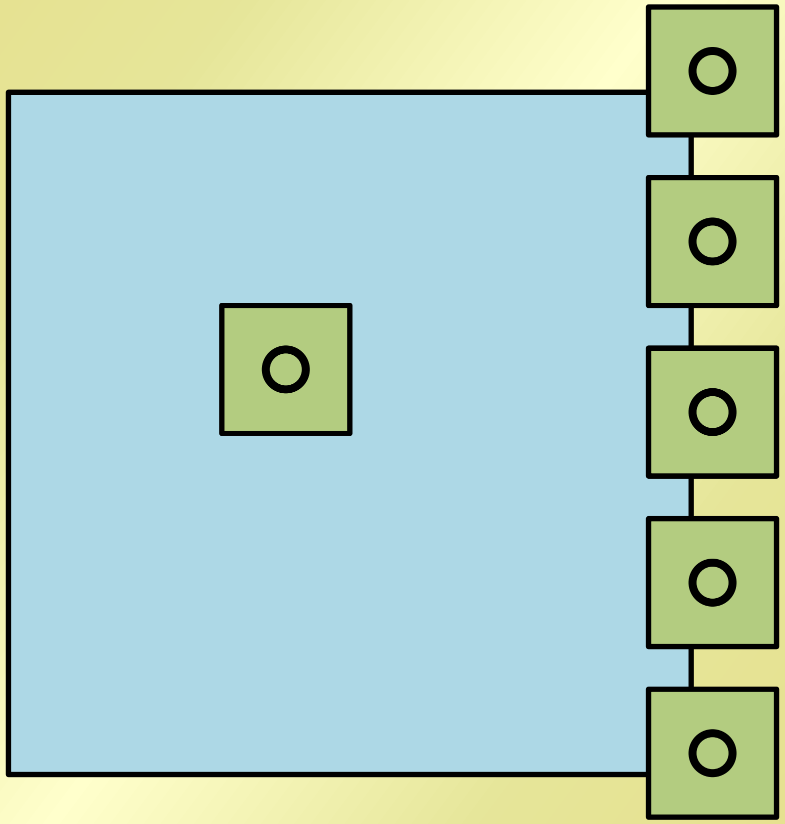
PROBLEM

The number of regions intersecting a quadtree leaf can be linear!



PROBLEM

The number of regions intersecting a quadtree leaf can be linear!

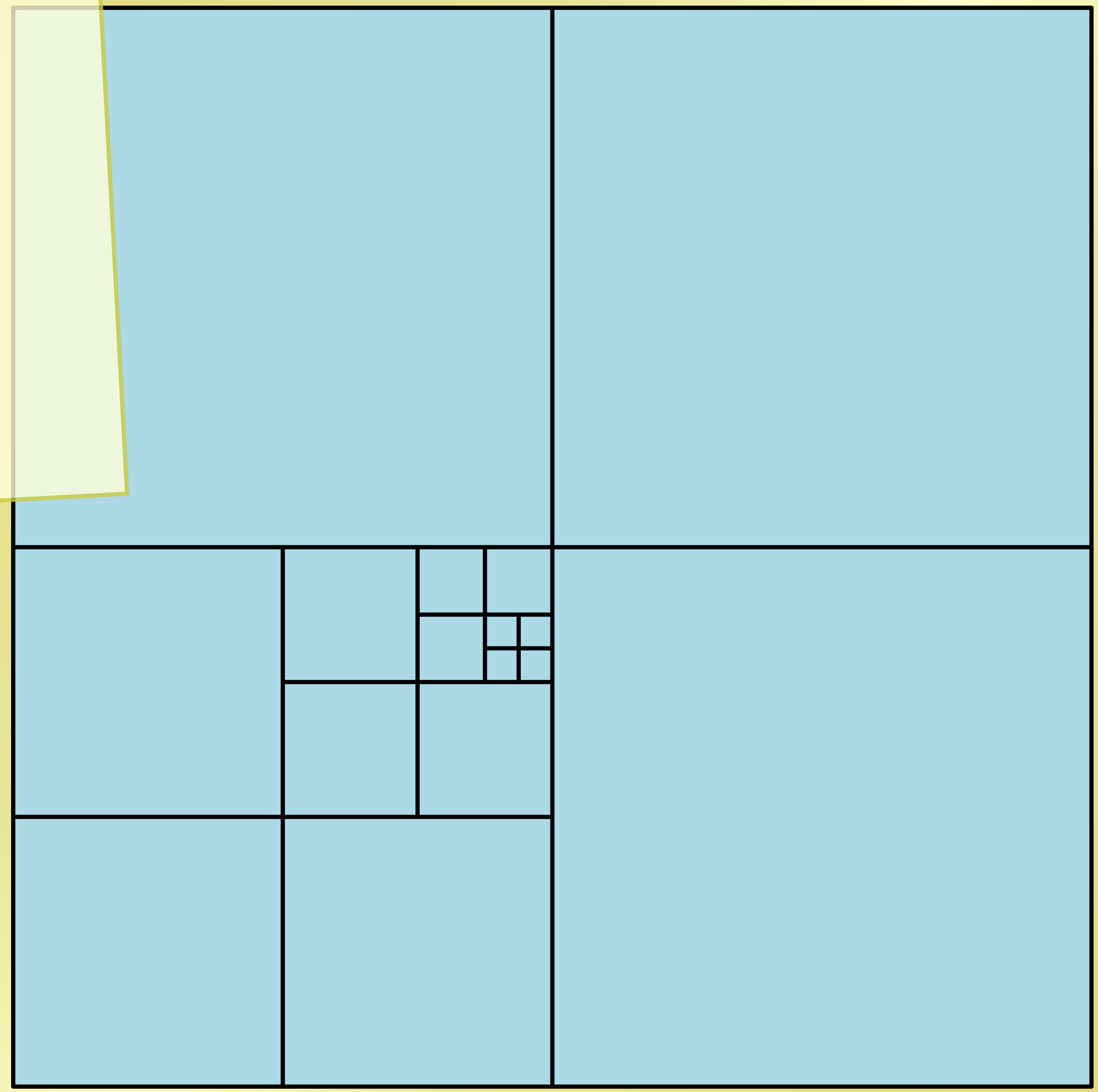


IDEA

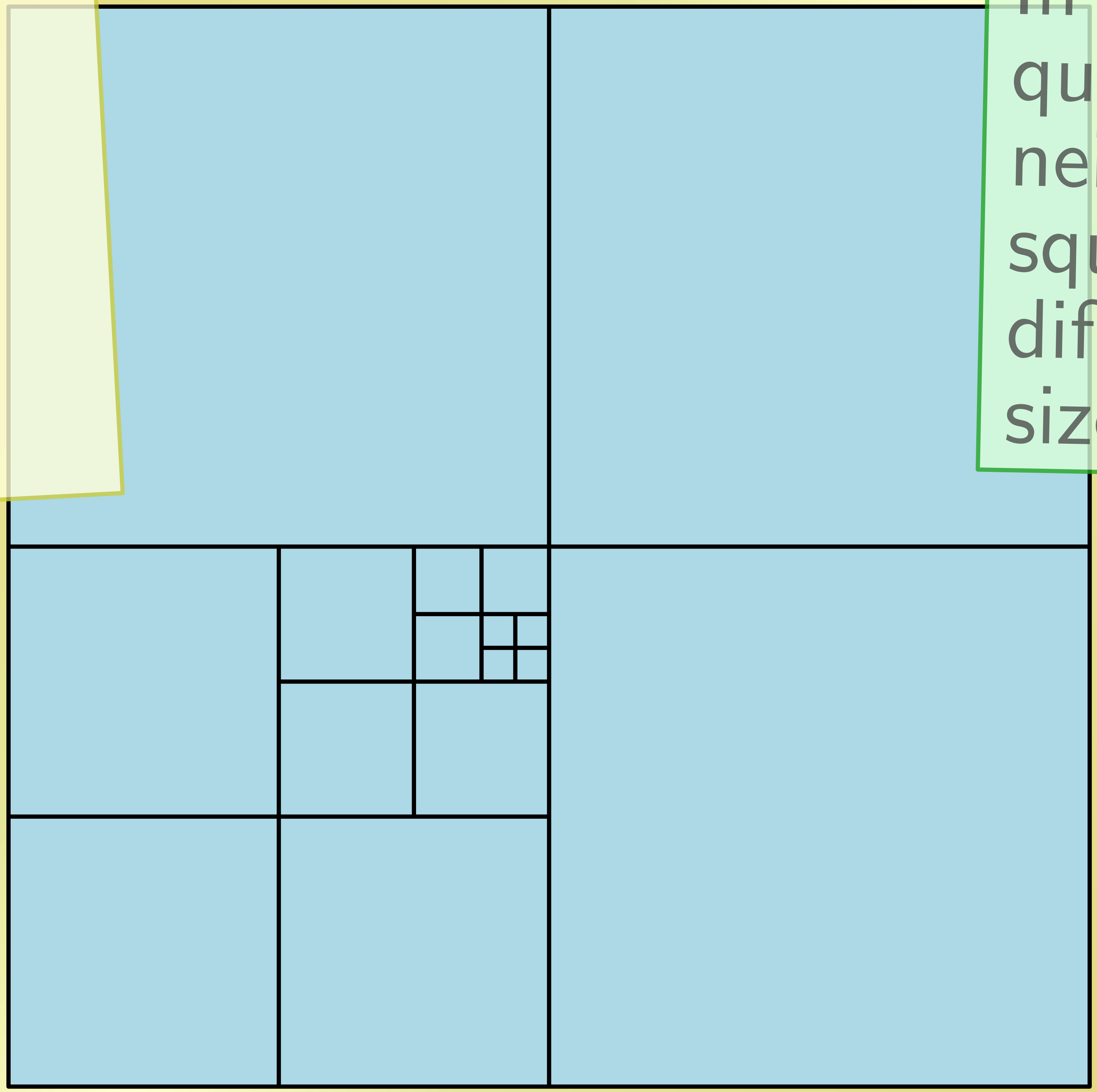
Maintain a *balanced* quadtree.

Consider a quadtree again.

Consider a quadtree again.

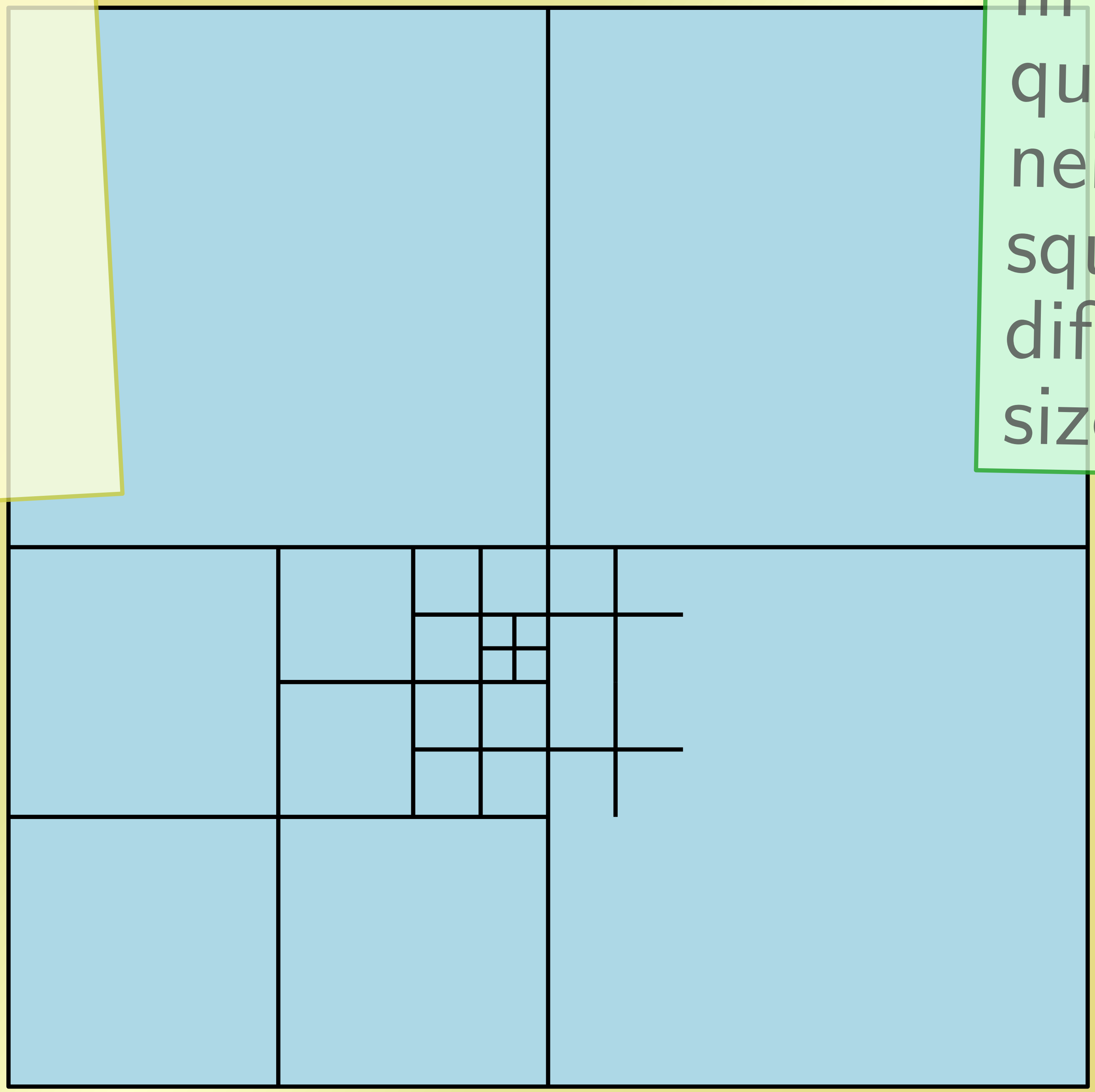


Consider a quadtree again.



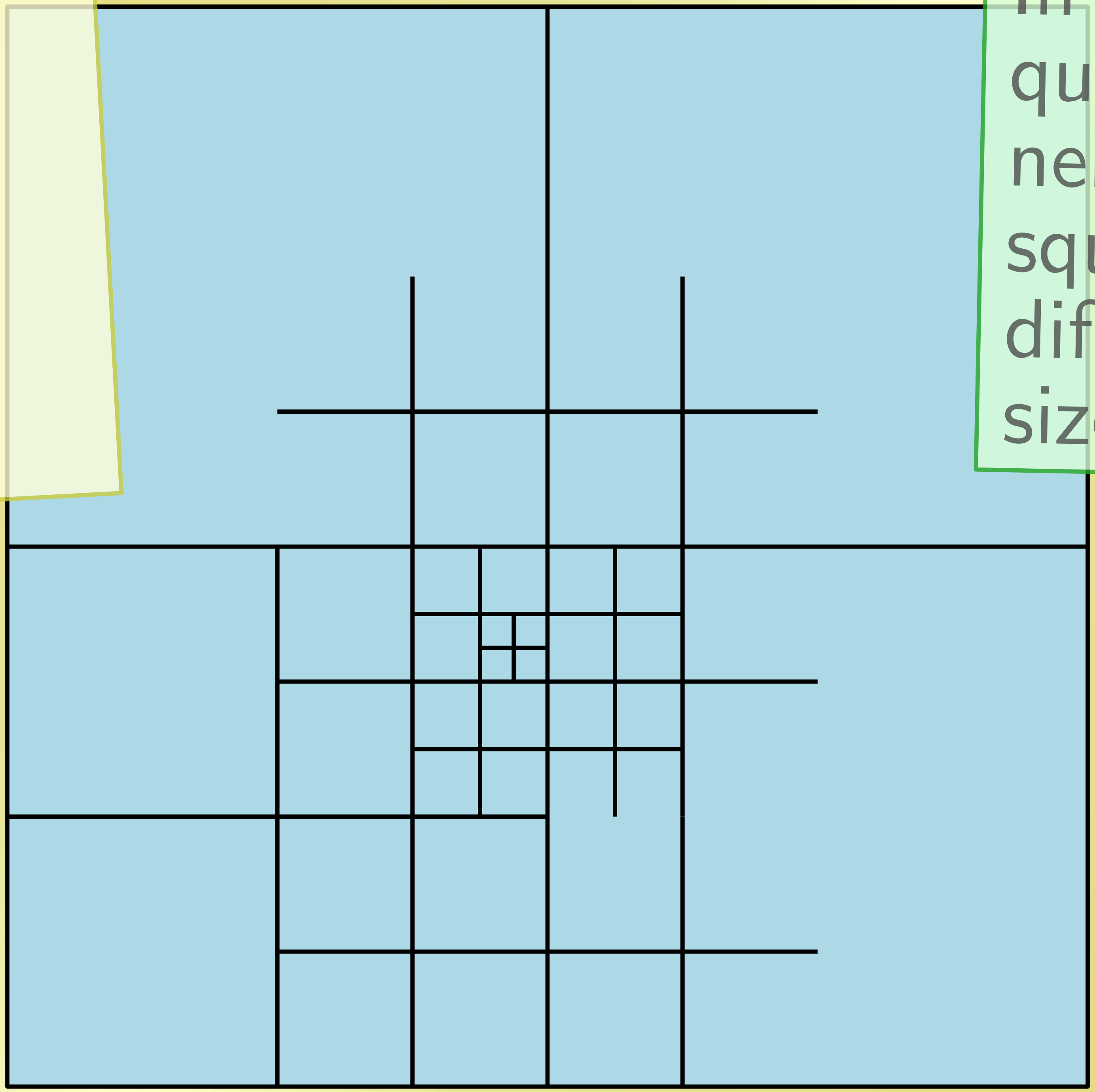
In a *balanced* quadtree, neighbouring squares don't differ much in size.

Consider a quadtree again.



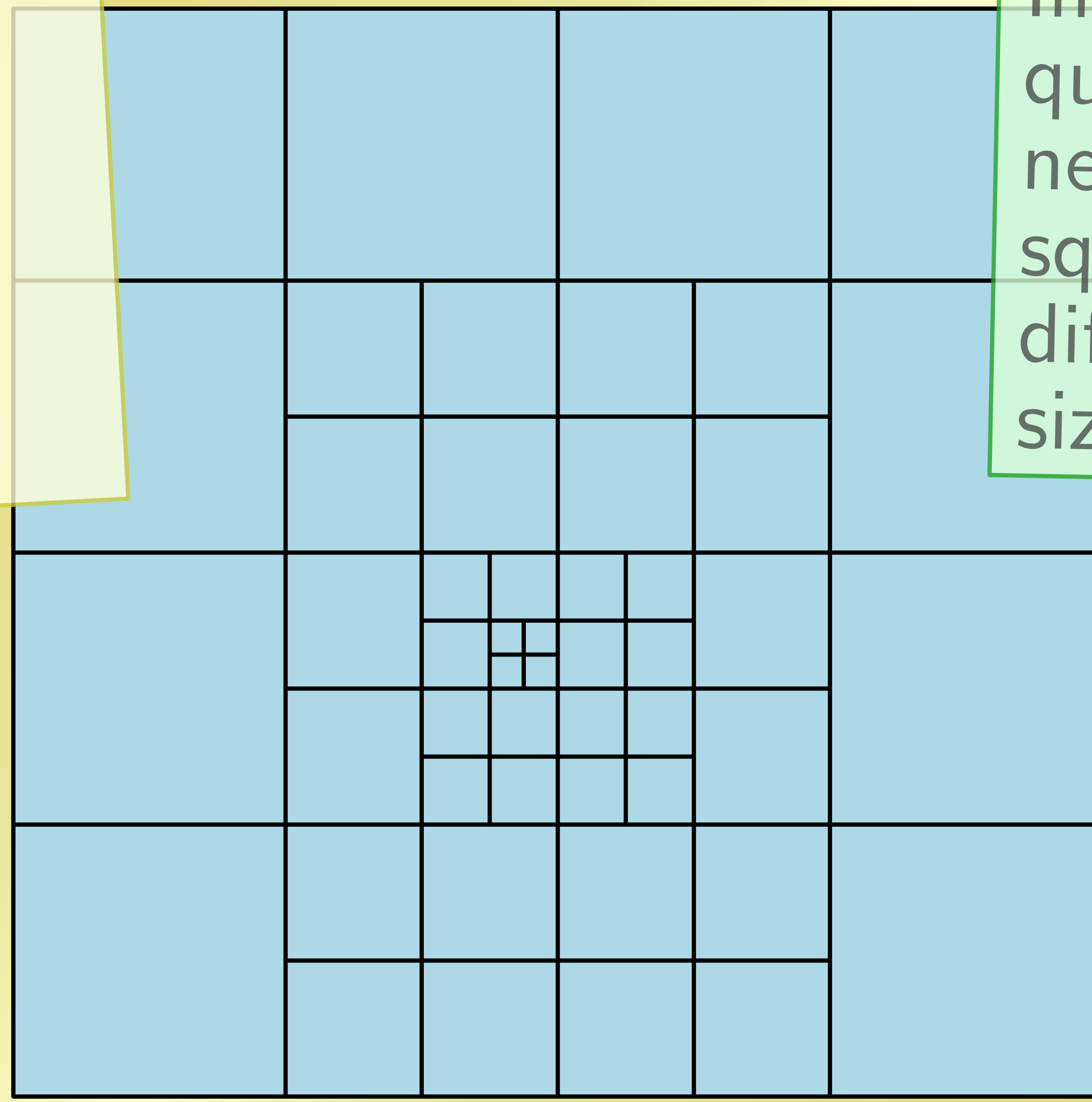
In a *balanced* quadtree, neighbouring squares don't differ much in size.

Consider a quadtree again.



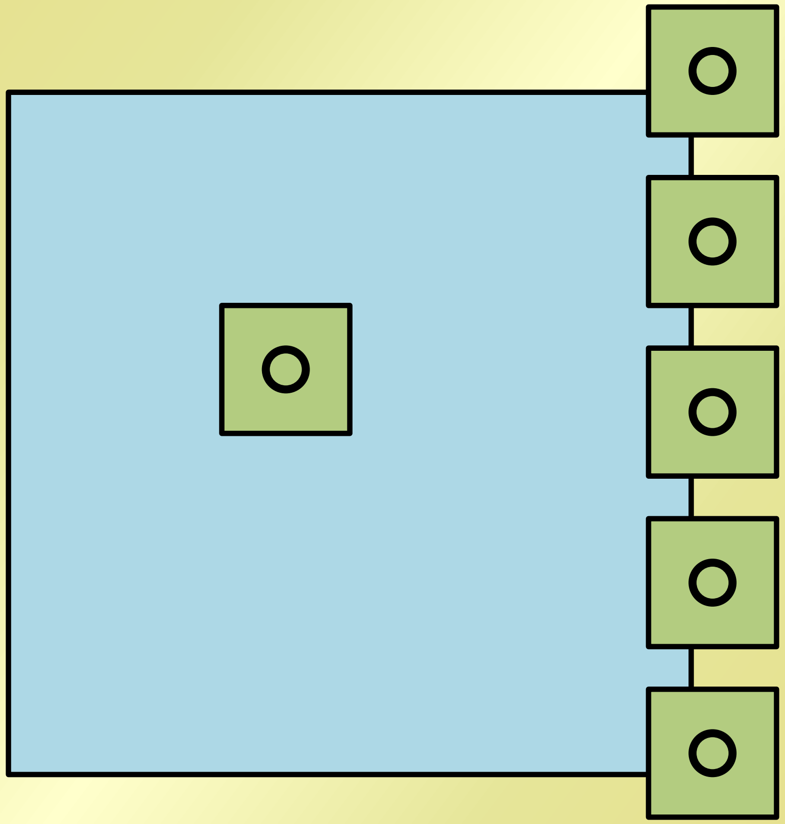
In a *balanced* quadtree, neighbouring squares don't differ much in size.

Consider a quadtree again.



In a *balanced* quadtree, neighbouring squares don't differ much in size.

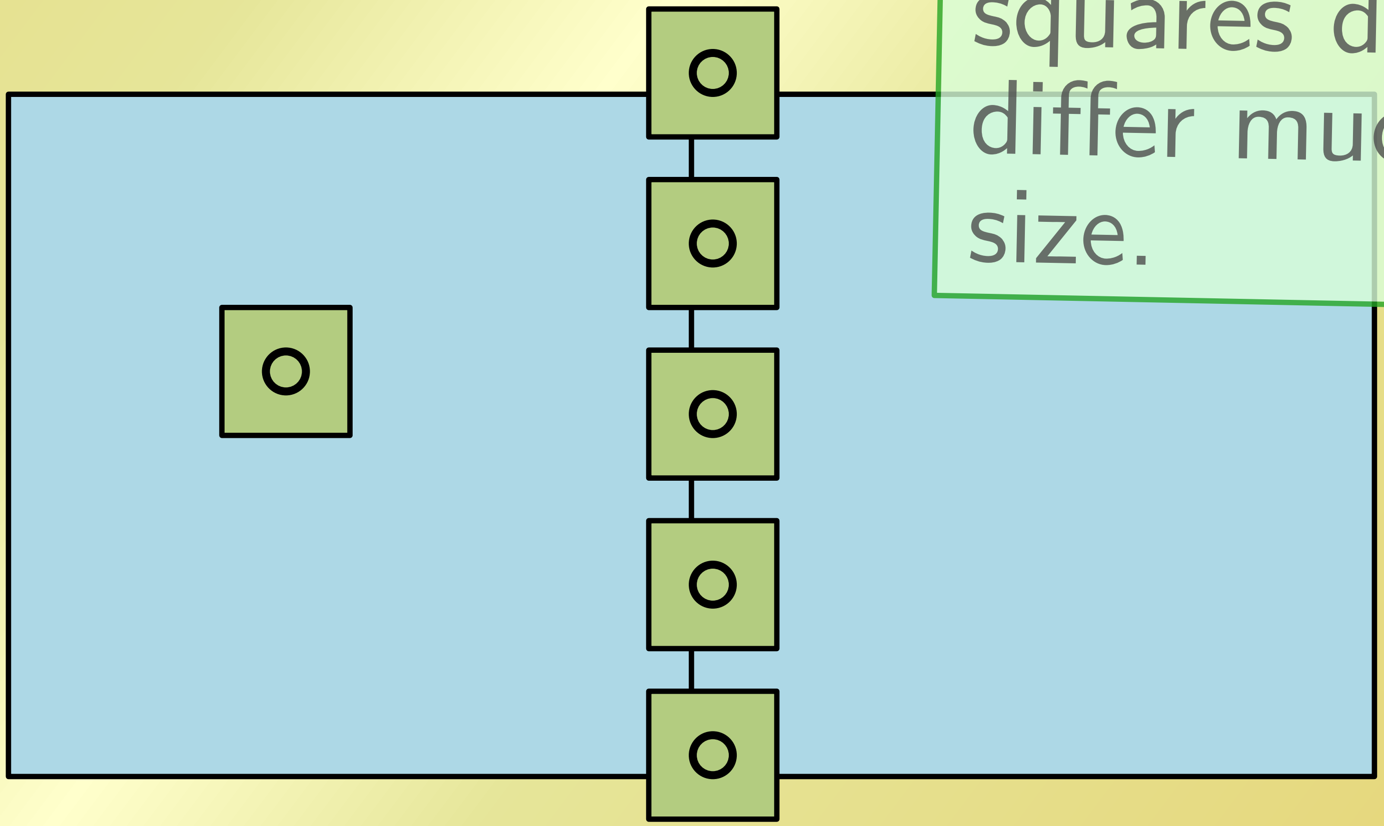
Consider a quadtree again.



In a *balanced* quadtree, neighbouring squares don't differ much in size.

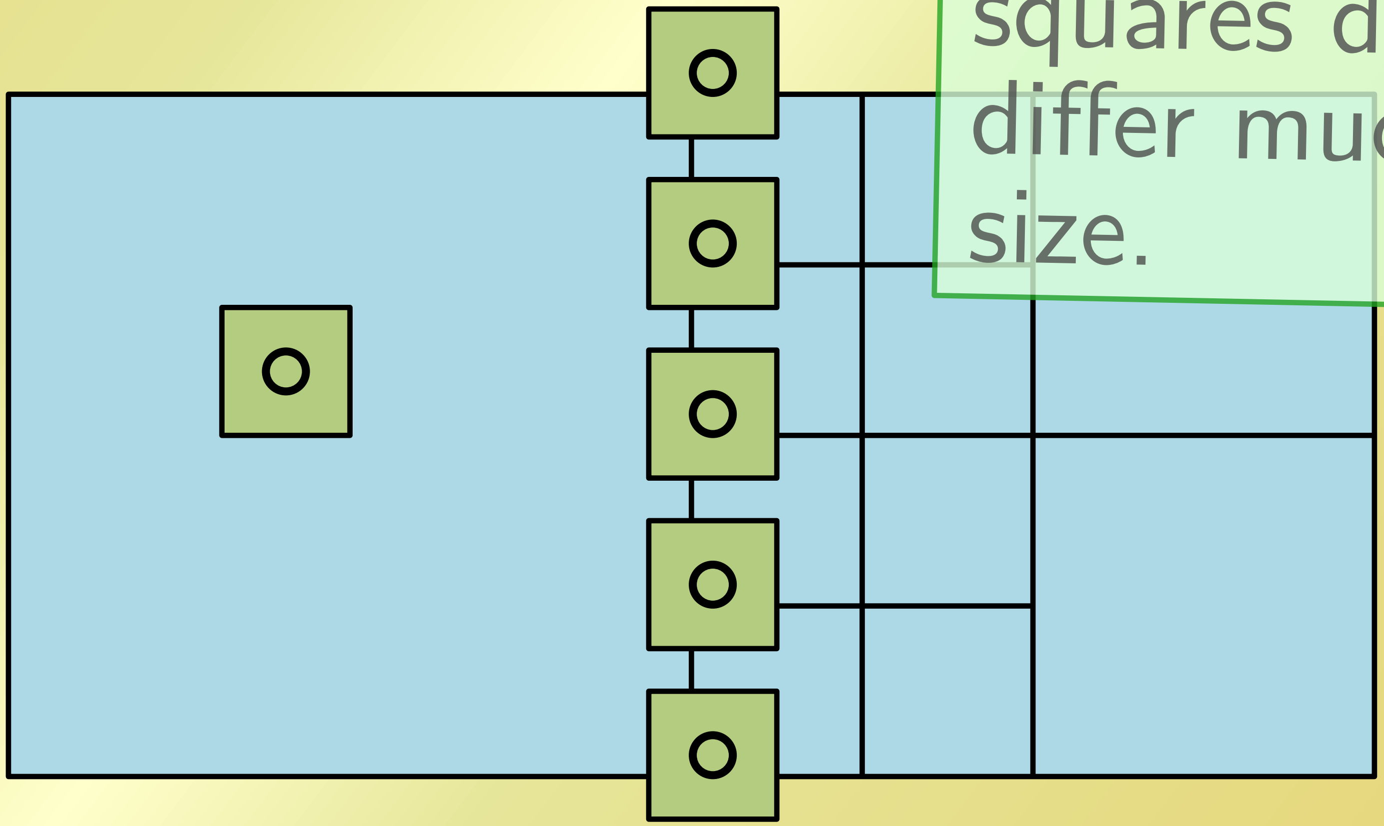
Consider a quadtree again.

In a *balanced* quadtree, neighbouring squares don't differ much in size.

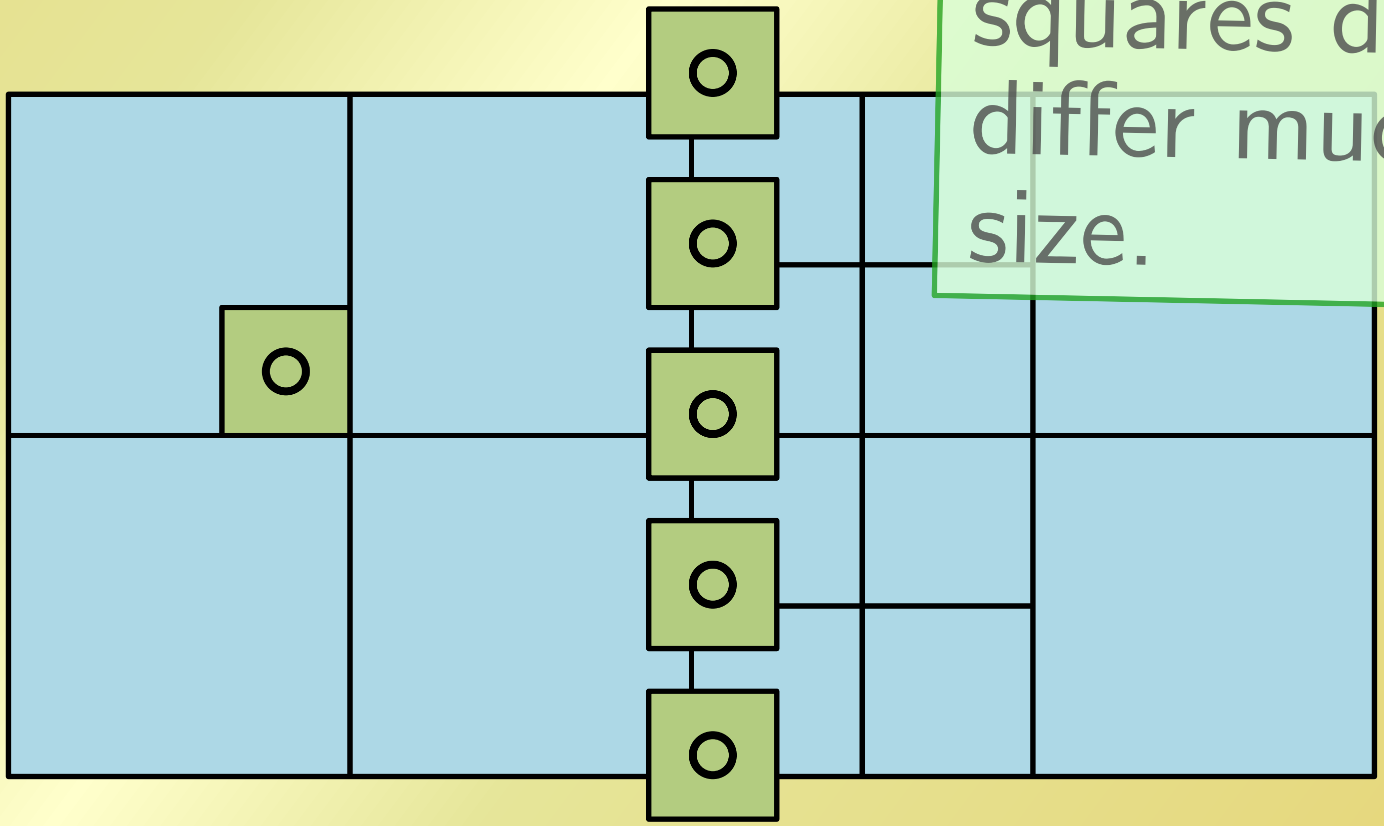


Consider a quadtree again.

In a *balanced* quadtree, neighbouring squares don't differ much in size.



Consider a quadtree again.

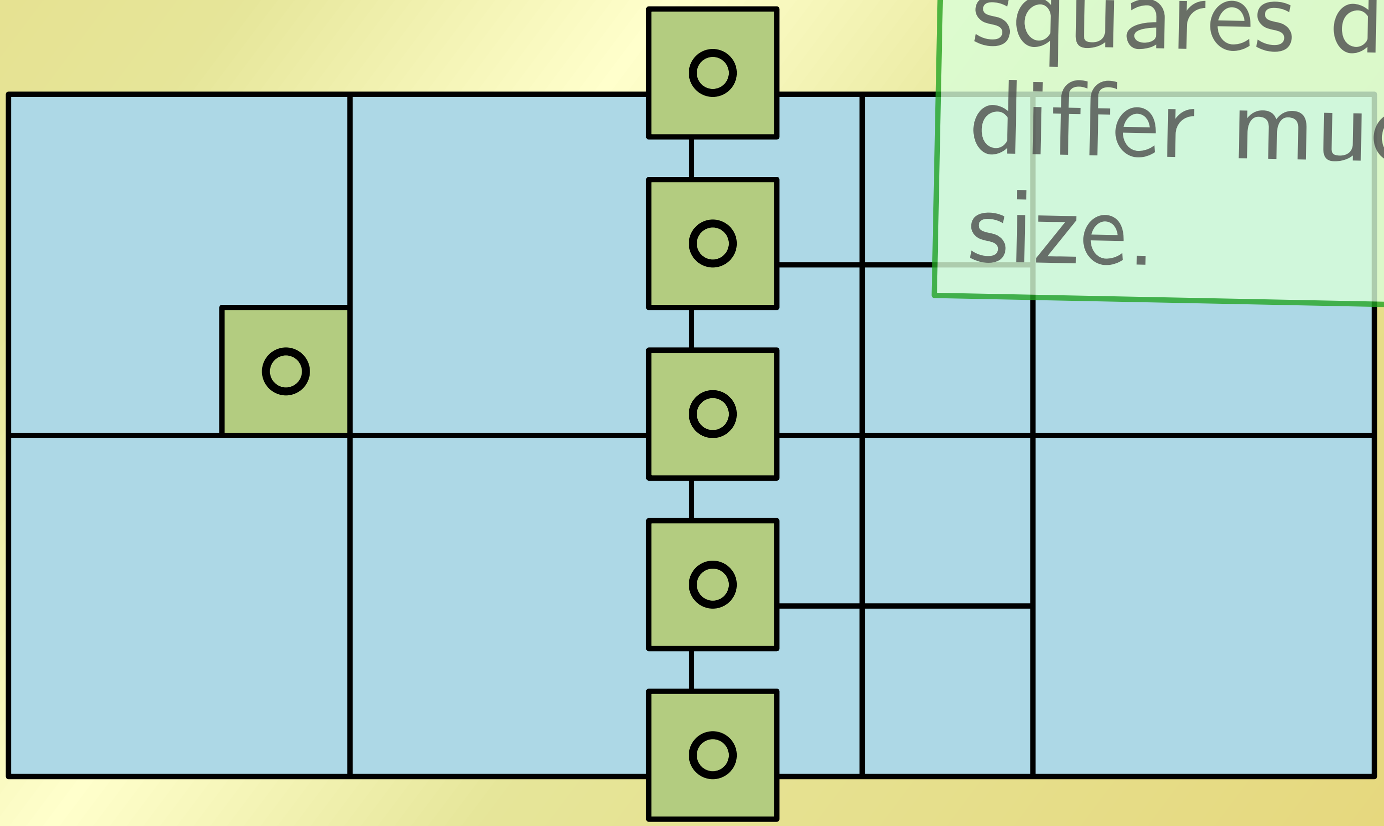


In a *balanced* quadtree, neighbouring squares don't differ much in size.

Consider a quadtree again.

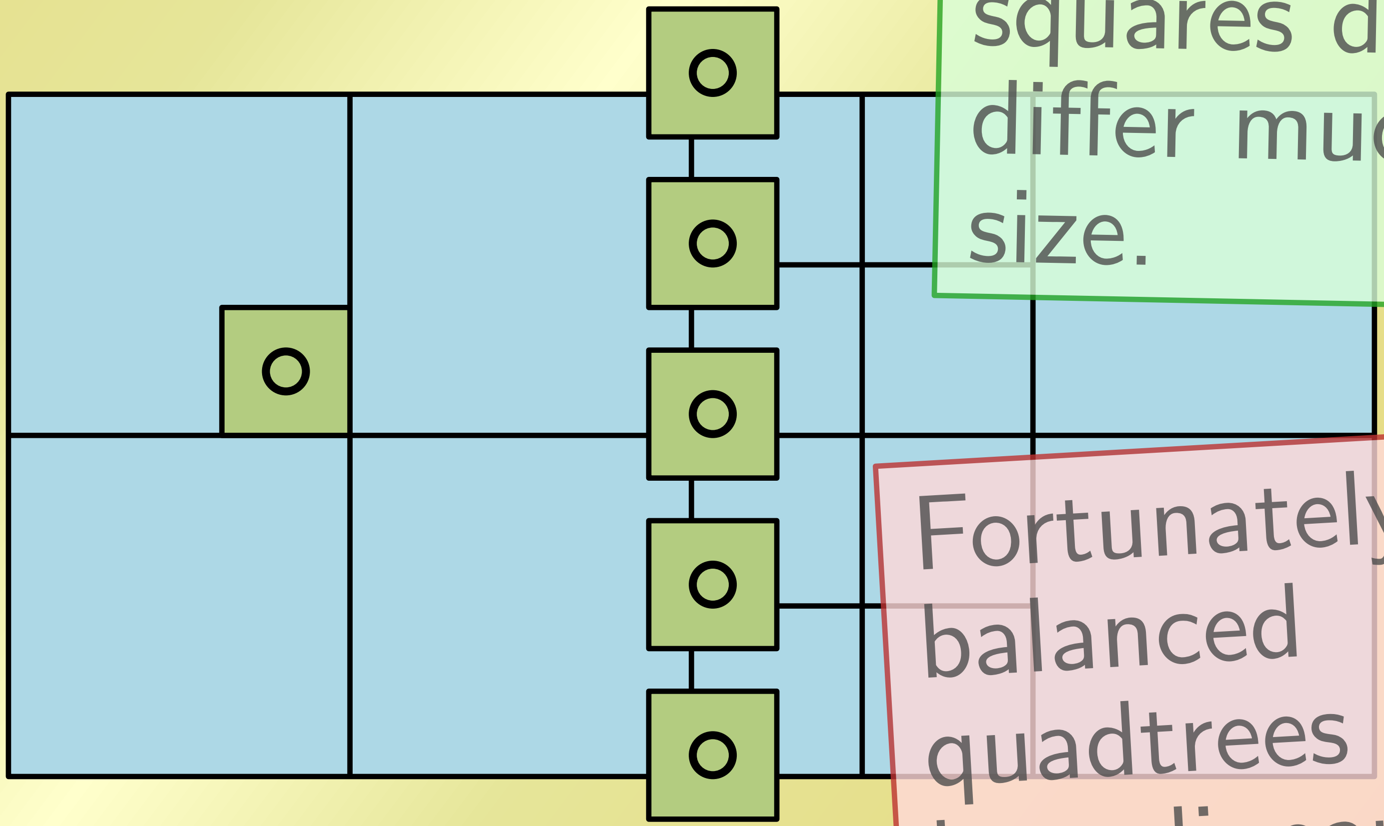
LEMMA
Now each leaf intersects at most $O(1)$ regions.

In a *balanced* quadtree, neighbouring squares don't differ much in size.



Consider a quadtree again.

In a *balanced* quadtree, neighbouring squares don't differ much in size.



LEMMA
Now each leaf intersects at most $O(1)$ regions.

Fortunately, balanced quadtrees still have linear size.

PROBLEM

But now we can't change the quadtree locally in $O(1)$ time!

PROBLEM

But now we can't change the quadtree locally in $O(1)$ time!

ADVICE

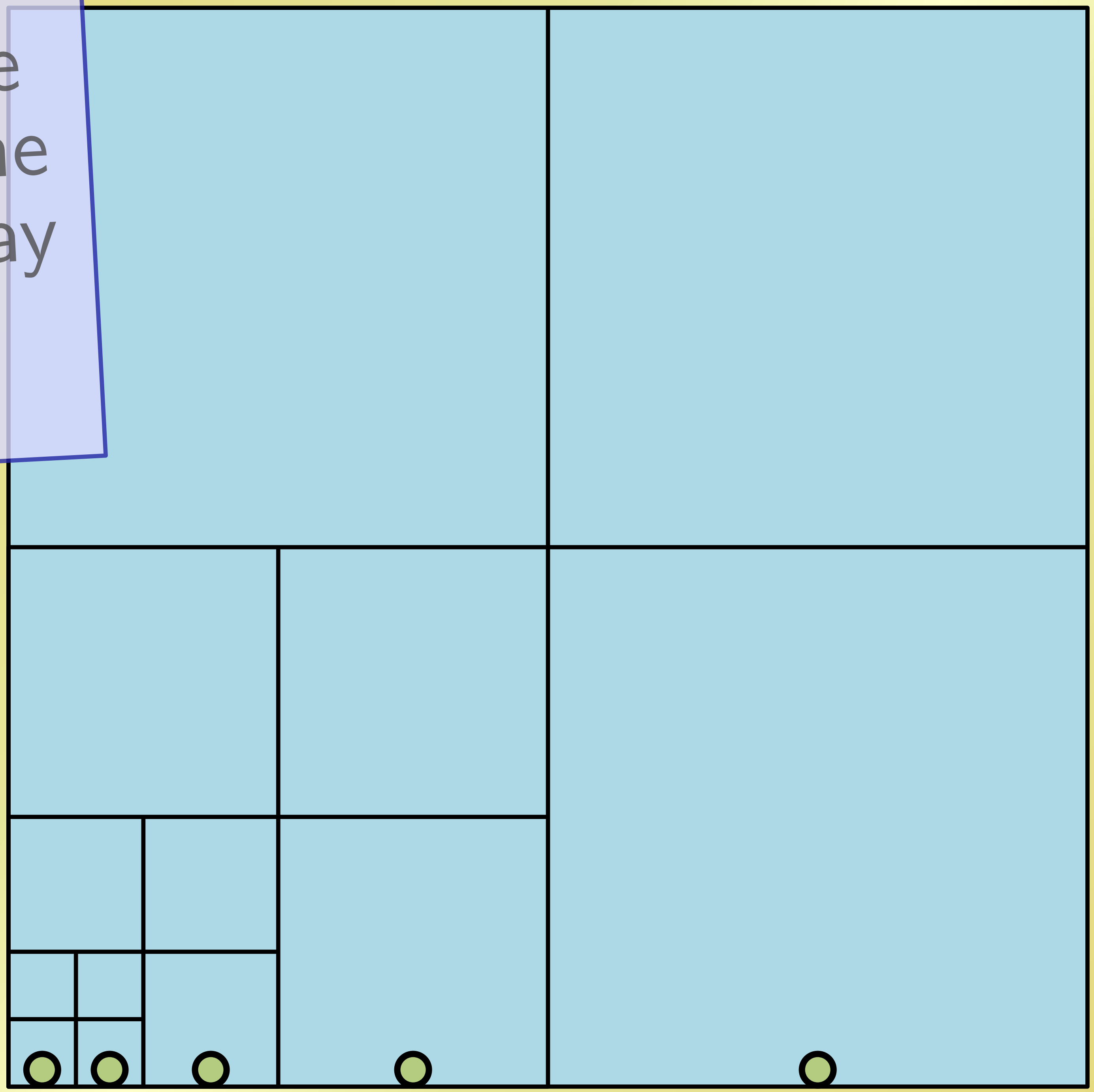
Don't worry, be happy!

PROBLEM

The distance from q to the right cell may be linear!

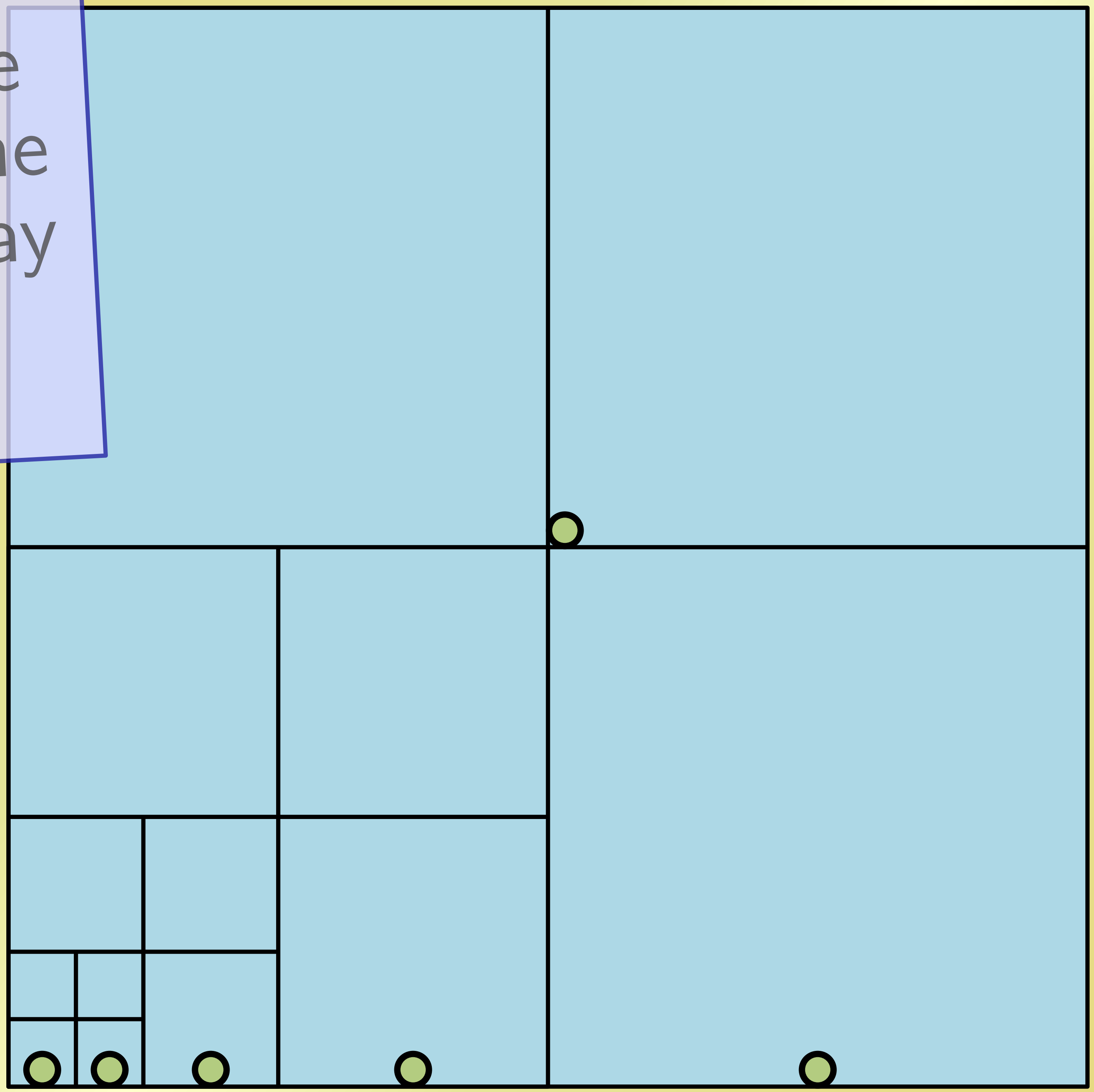
PROBLEM

The distance from q to the right cell may be linear!

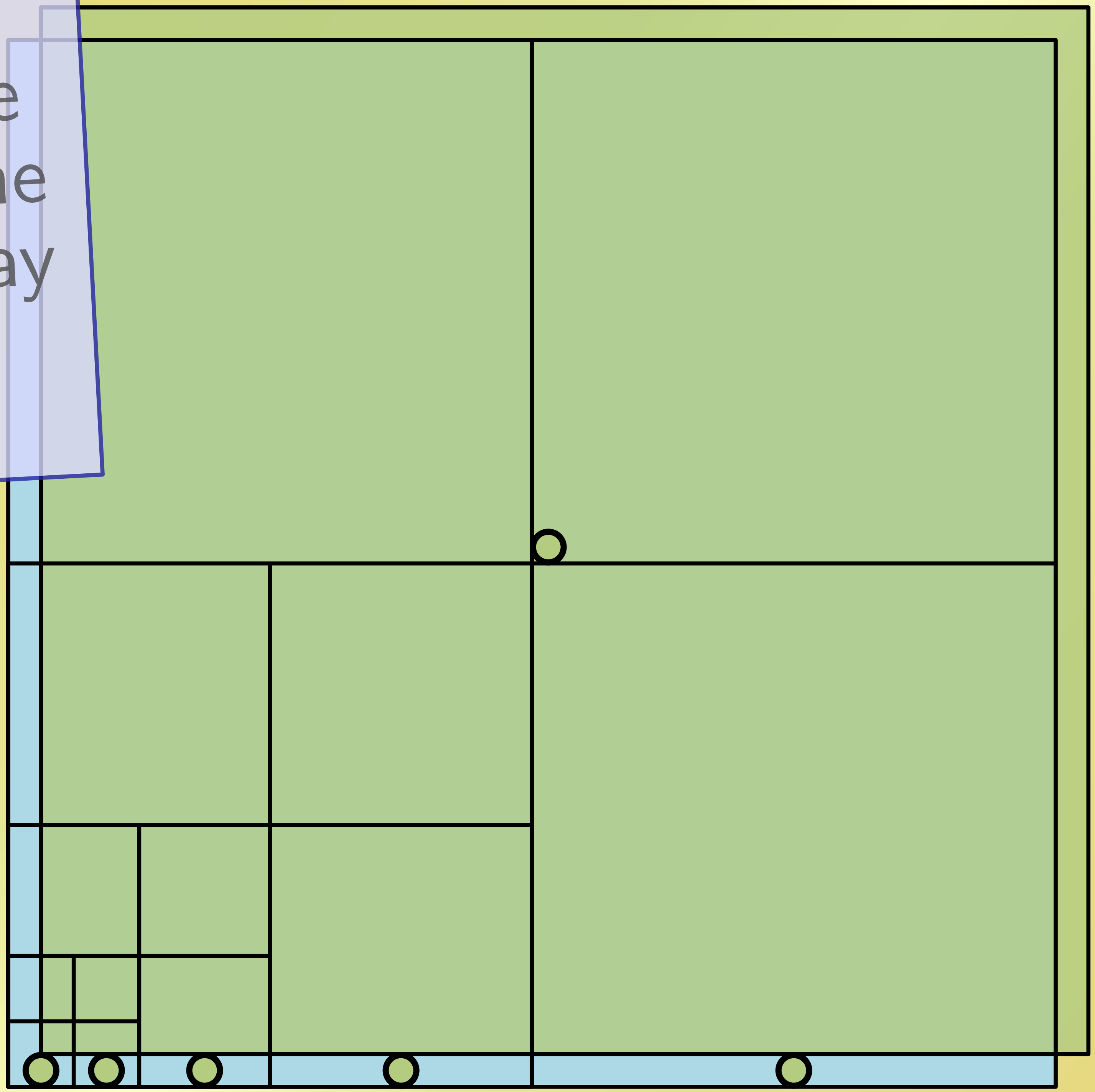


PROBLEM

The distance from q to the right cell may be linear!

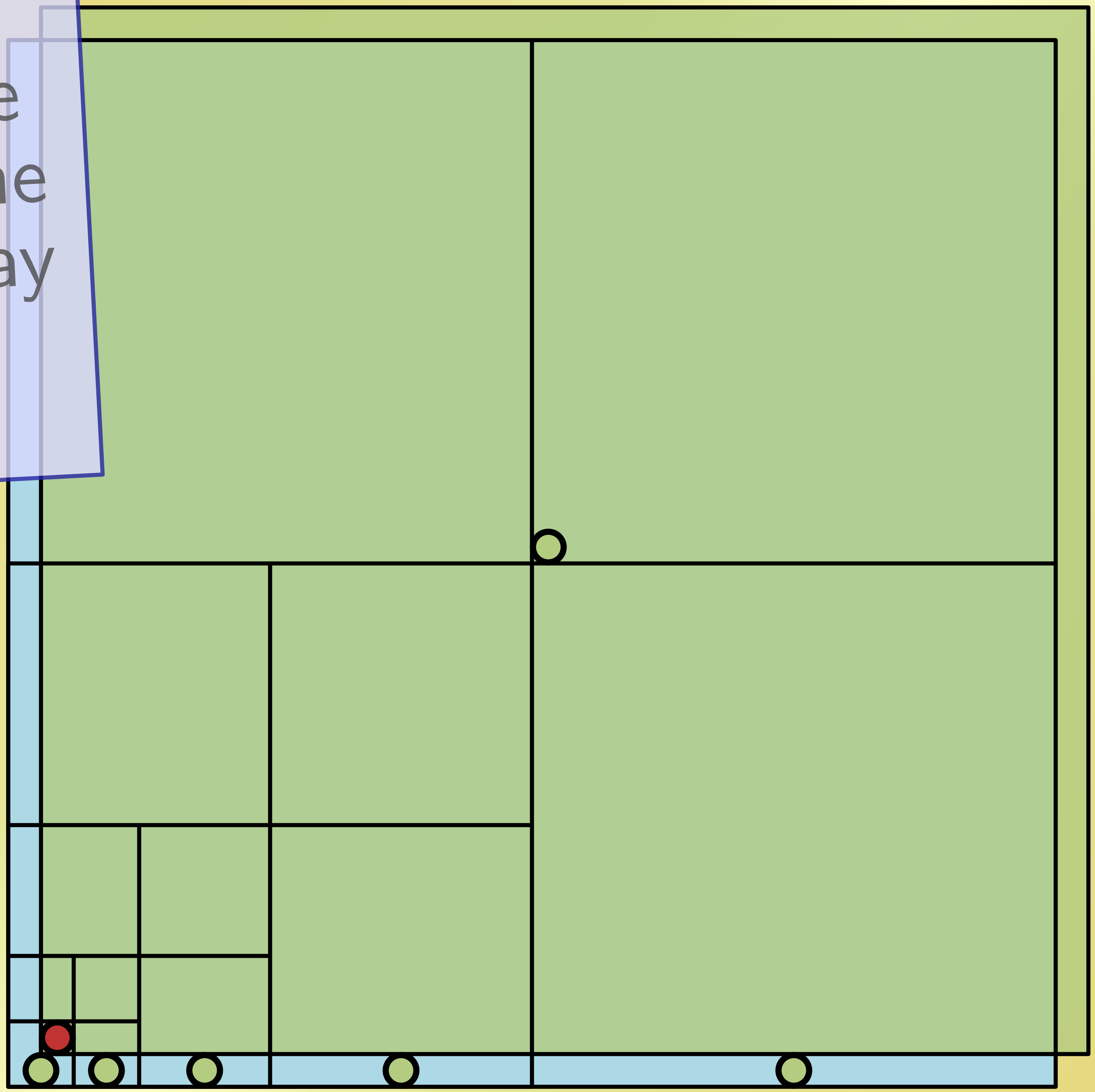


PROBLEM
The distance from q to the right cell may be linear!



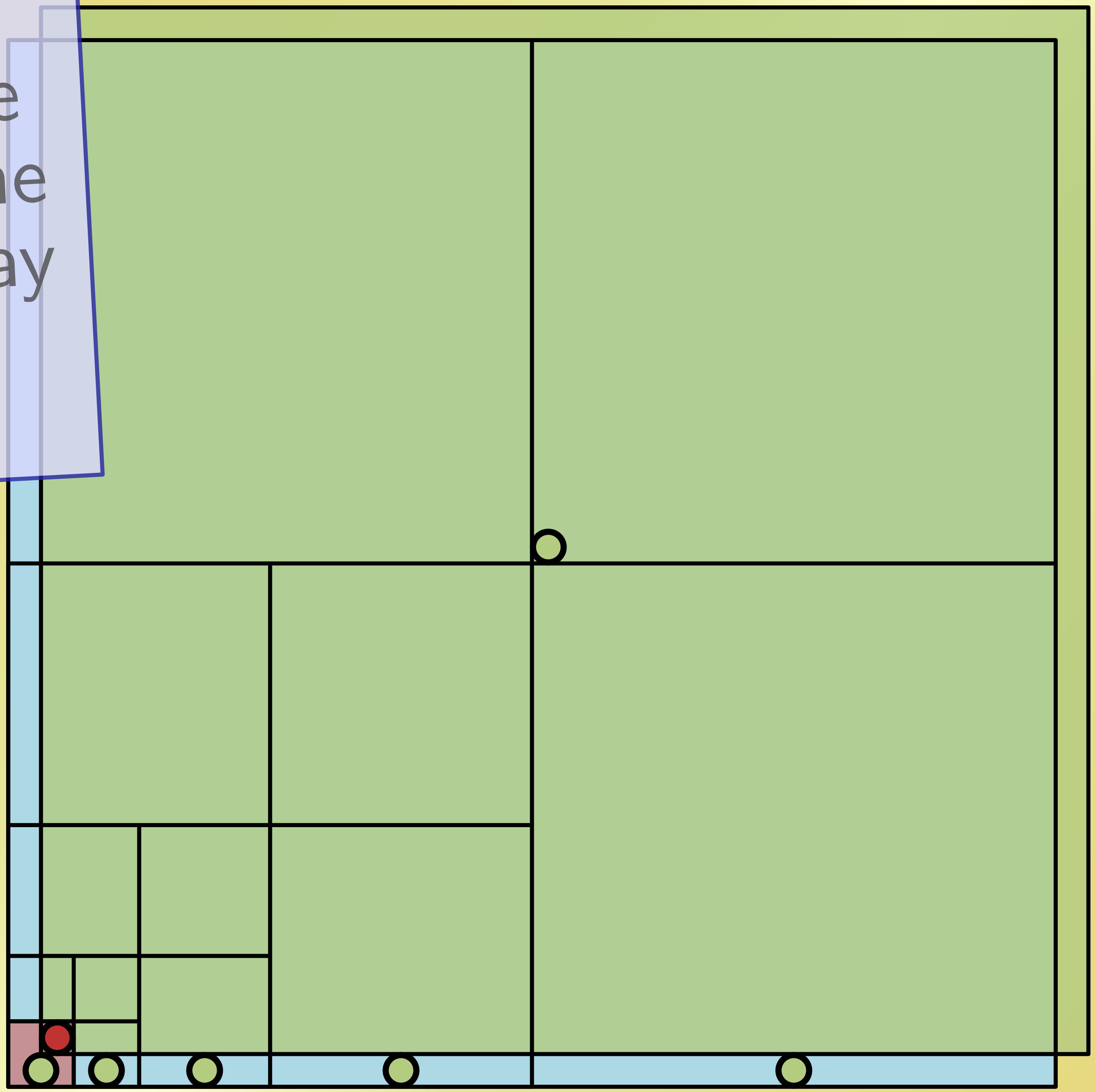
PROBLEM

The distance from q to the right cell may be linear!



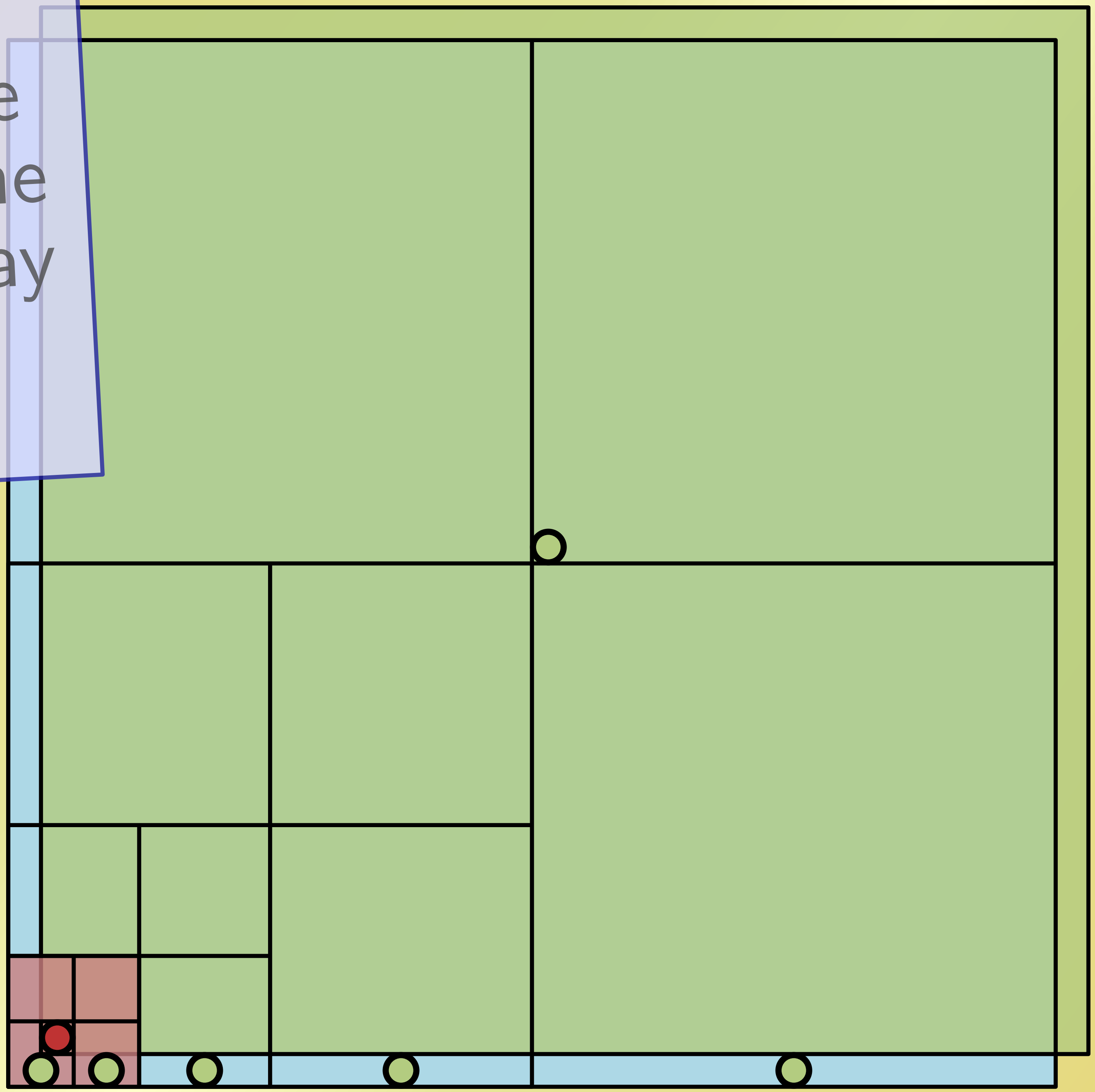
PROBLEM

The distance from q to the right cell may be linear!



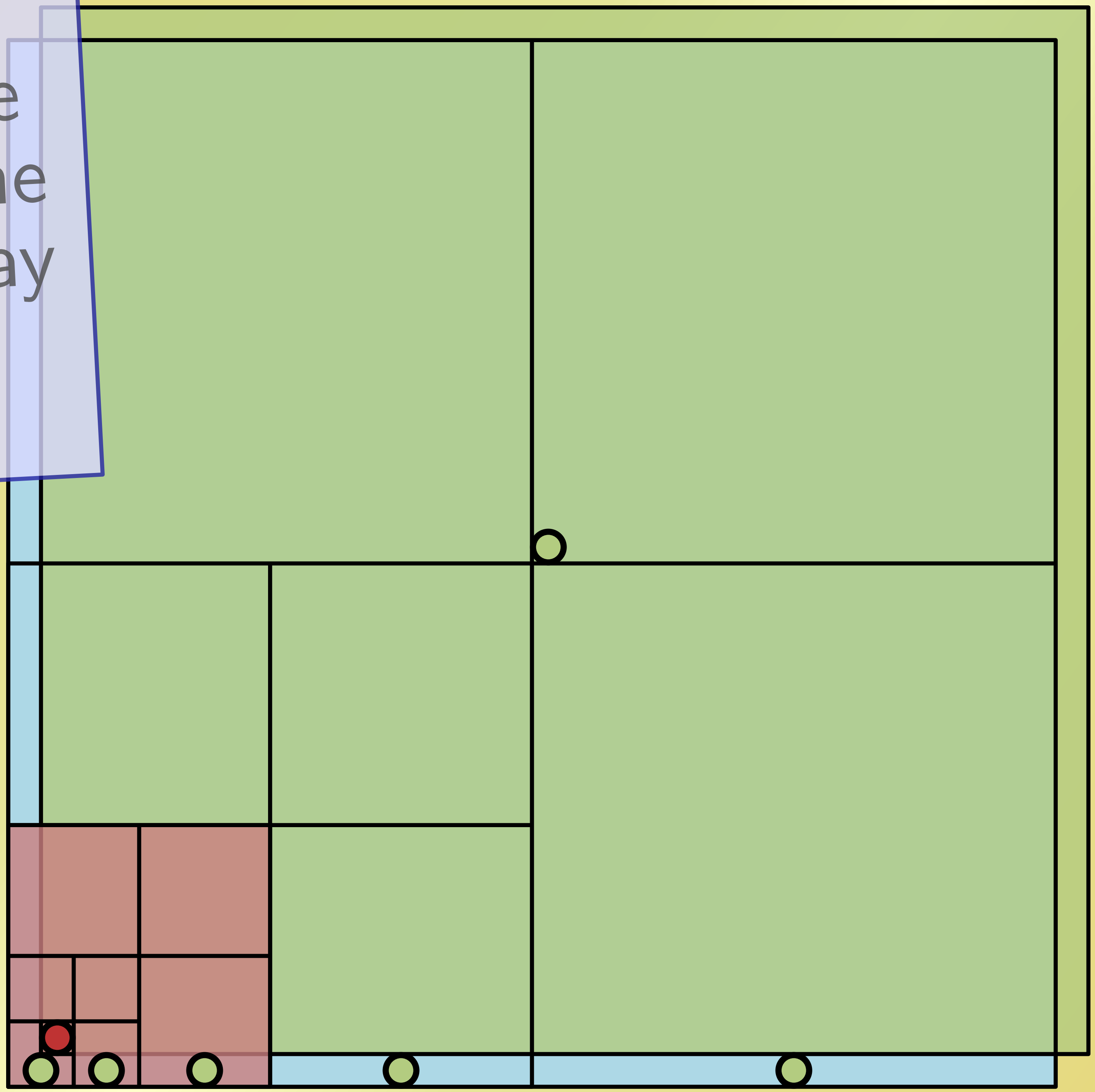
PROBLEM

The distance from q to the right cell may be linear!

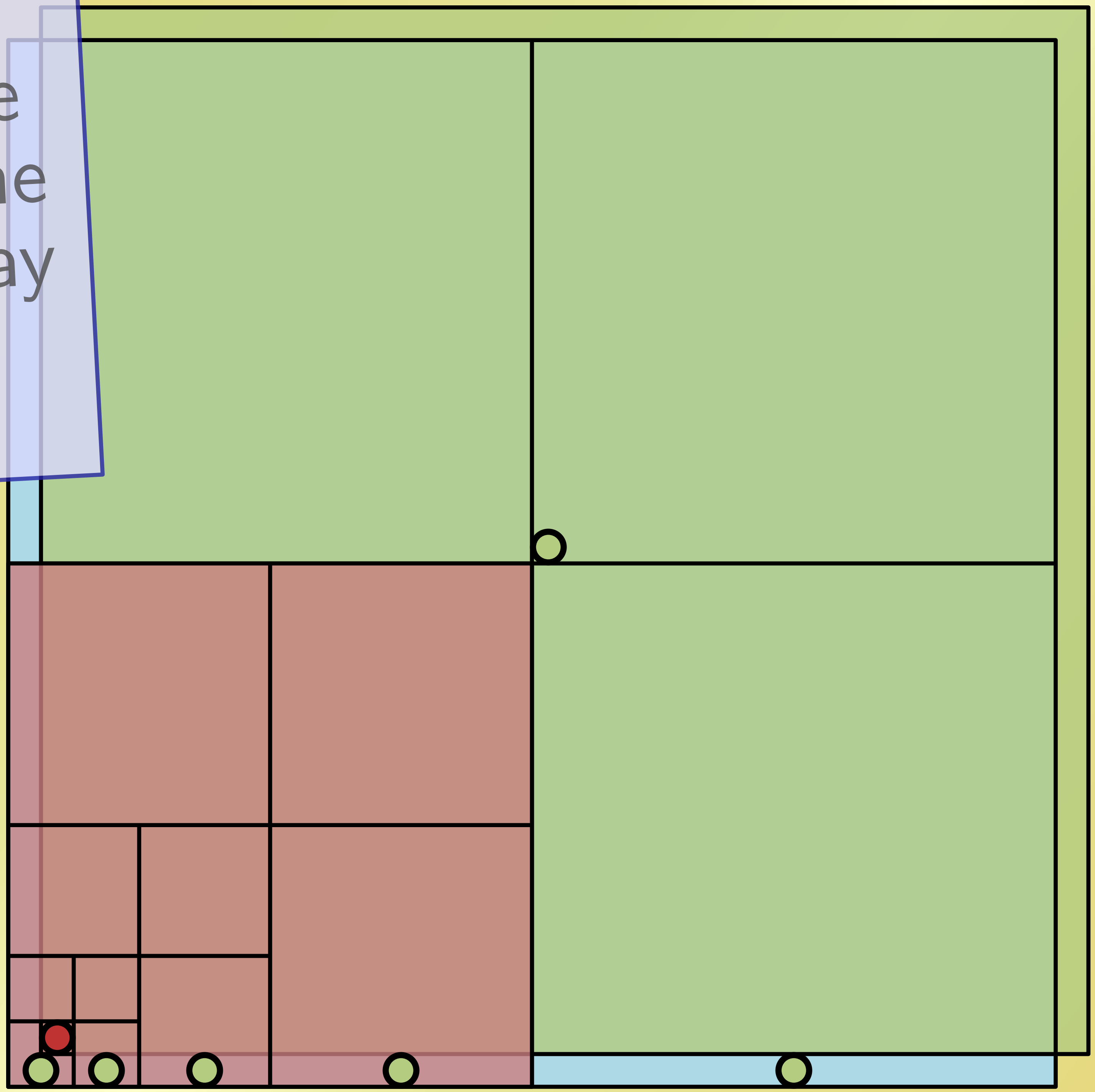


PROBLEM

The distance from q to the right cell may be linear!

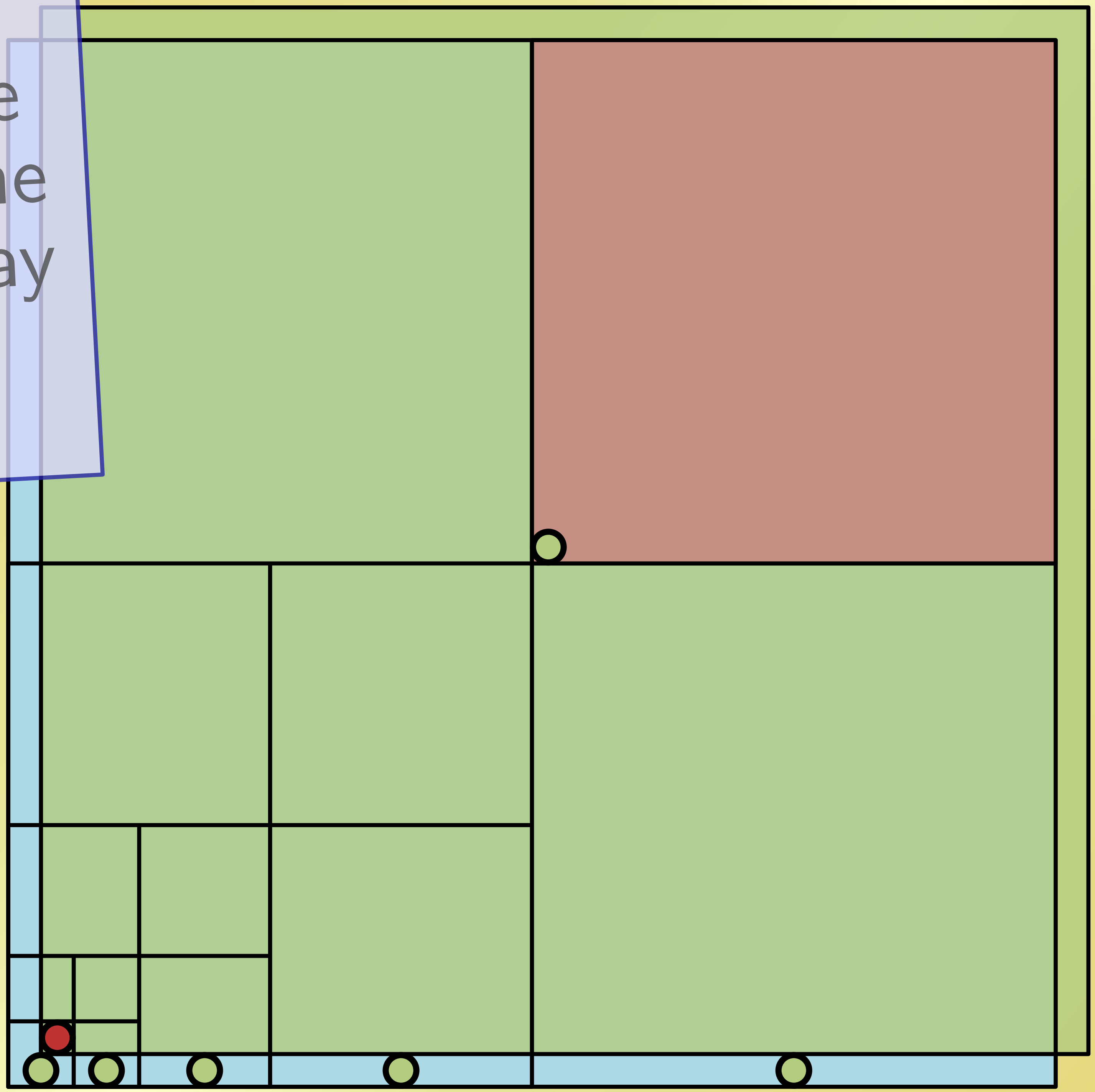


PROBLEM
The distance from q to the right cell may be linear!



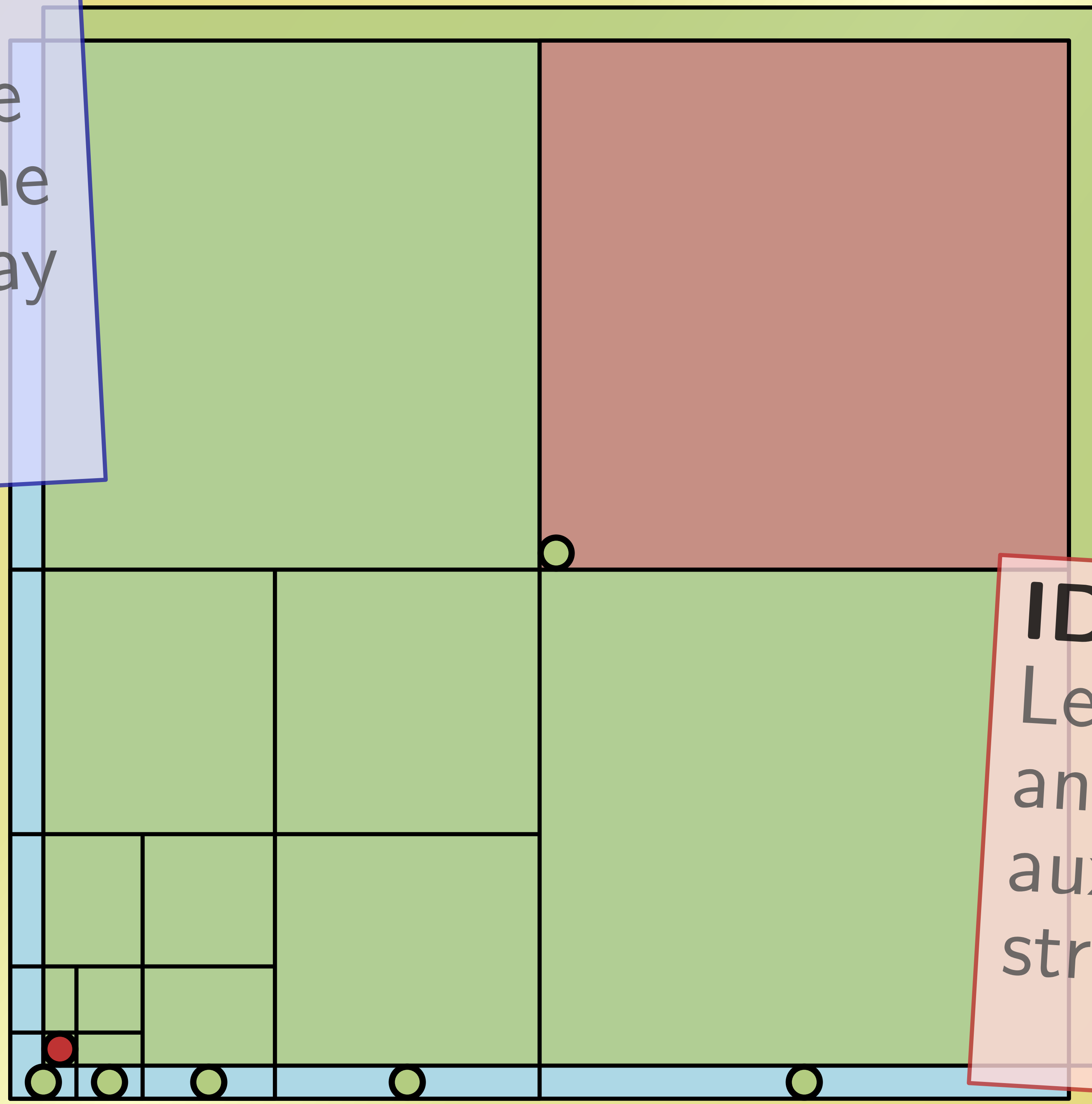
PROBLEM

The distance from q to the right cell may be linear!



PROBLEM

The distance from q to the right cell may be linear!



IDEA

Let's add yet another auxliliary structure!

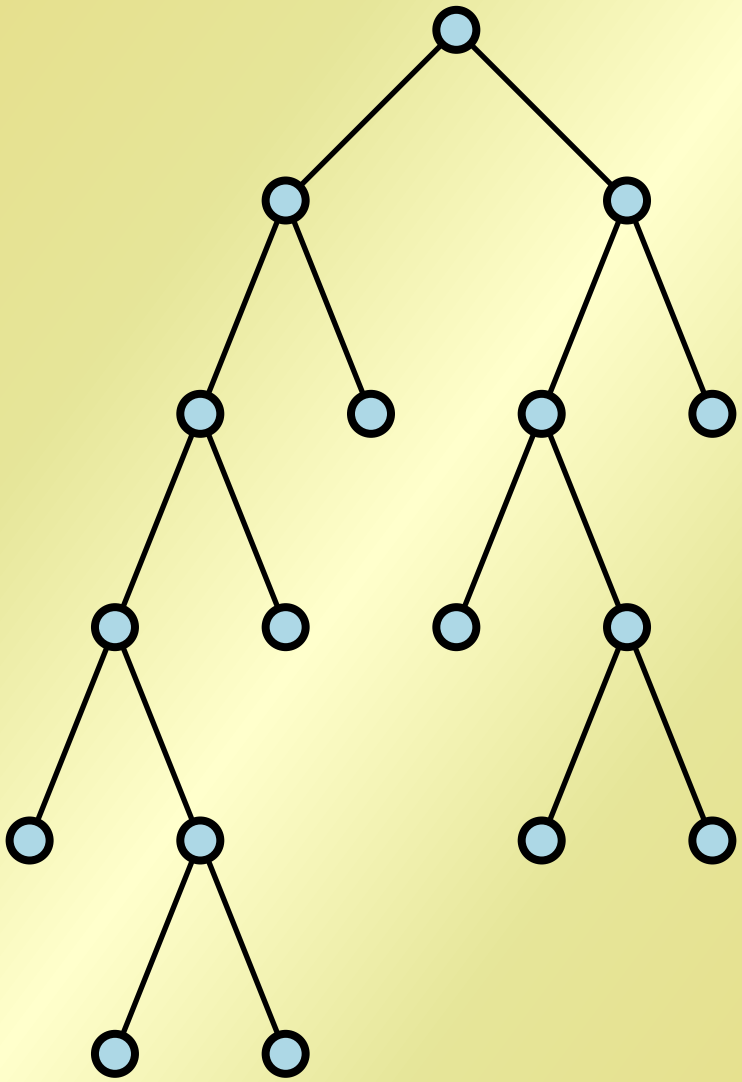
We use a *marked ancestor* data structure.

We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

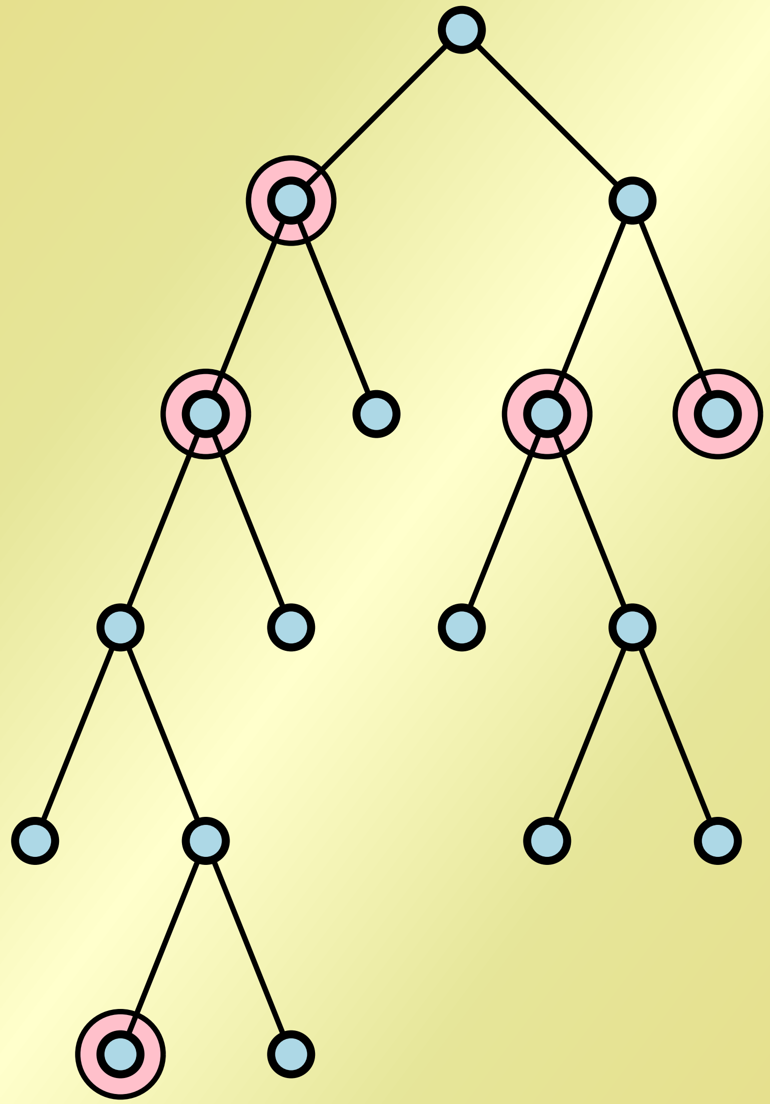
We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.



We use a *marked ancestor* data structure.

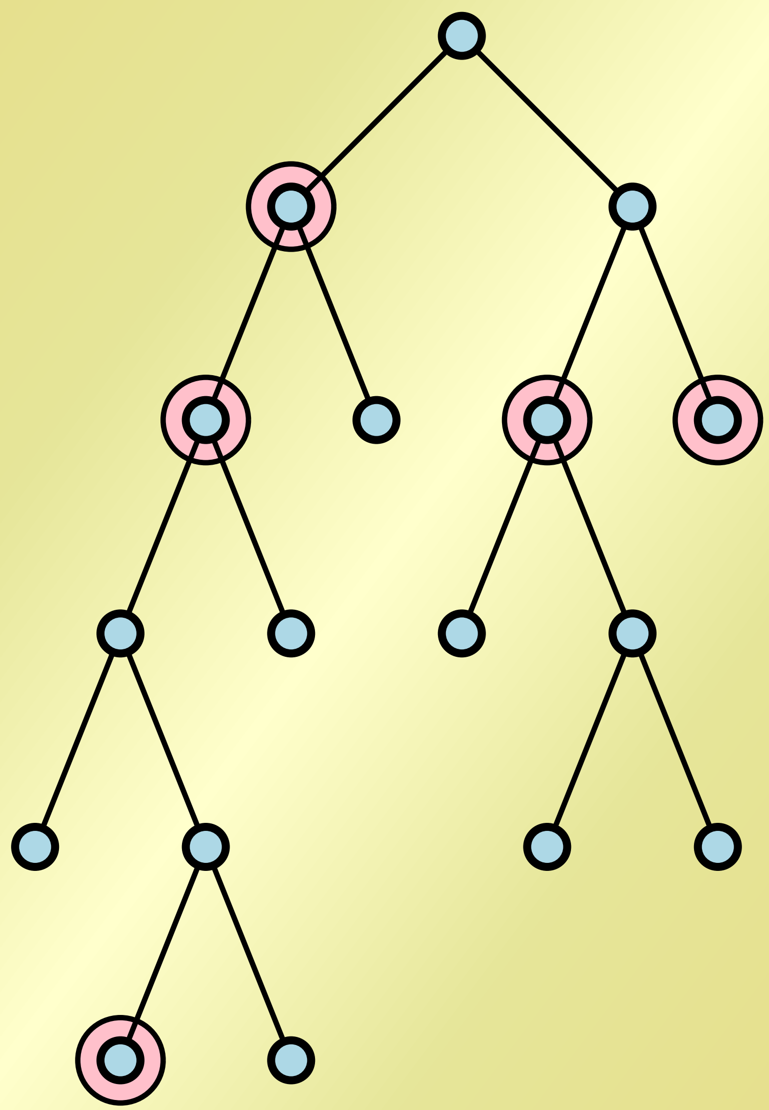
Consider a tree, where some nodes are *marked*.



We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

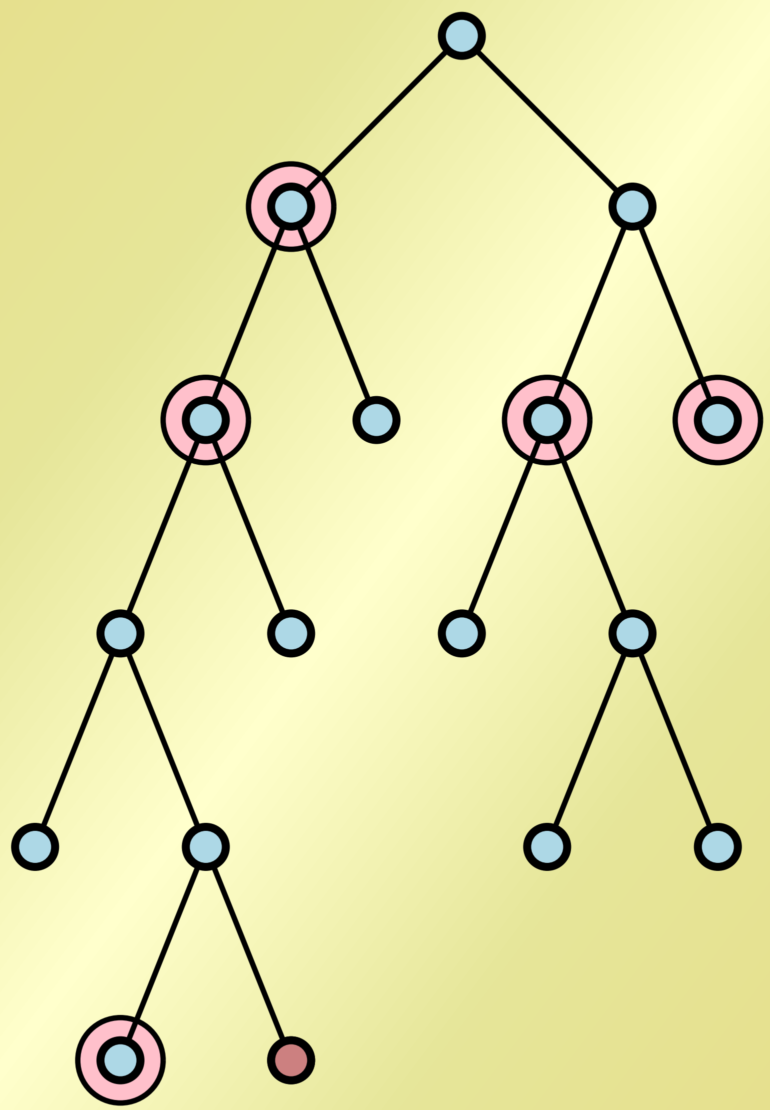
For a given query node, we wish to find the first marked ancestor.



We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

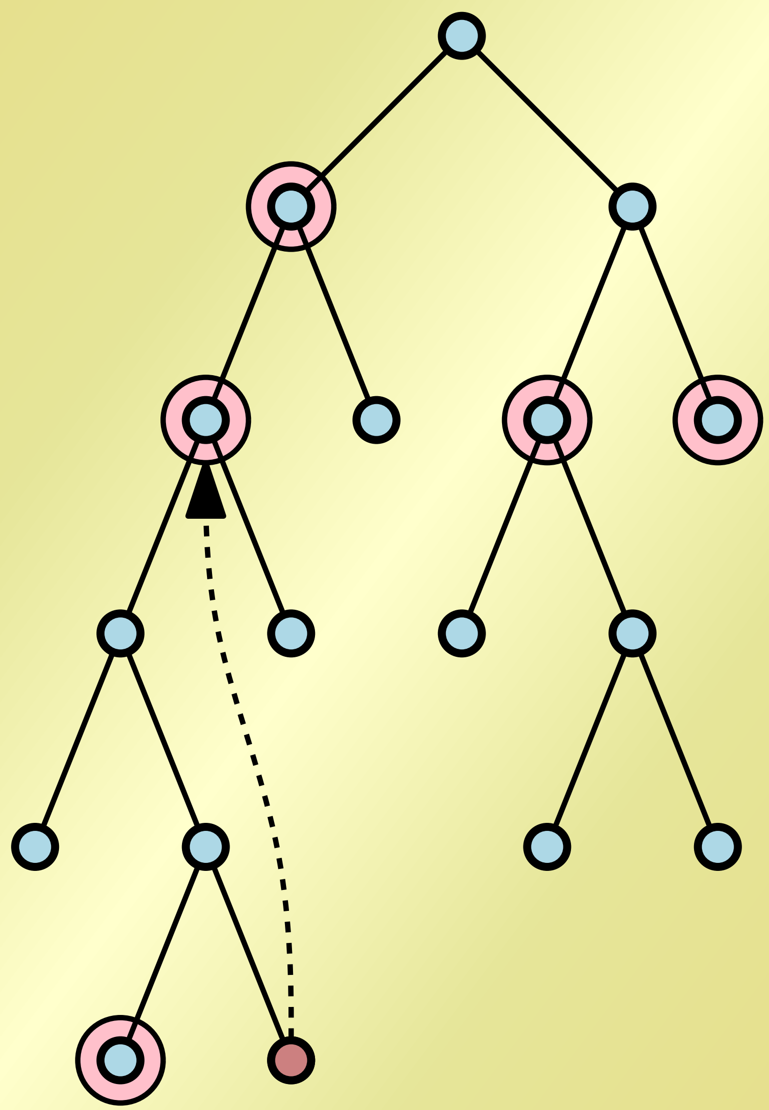
For a given query node, we wish to find the first marked ancestor.



We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

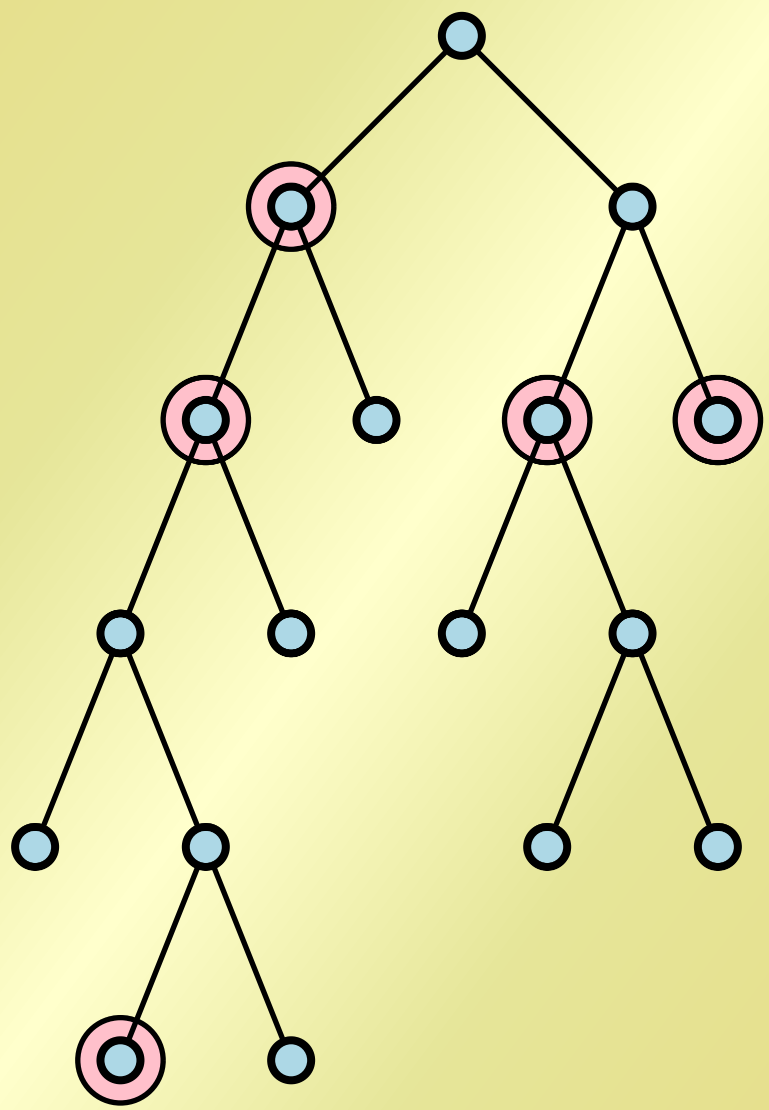
For a given query node, we wish to find the first marked ancestor.



We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

For a given query node, we wish to find the first marked ancestor.

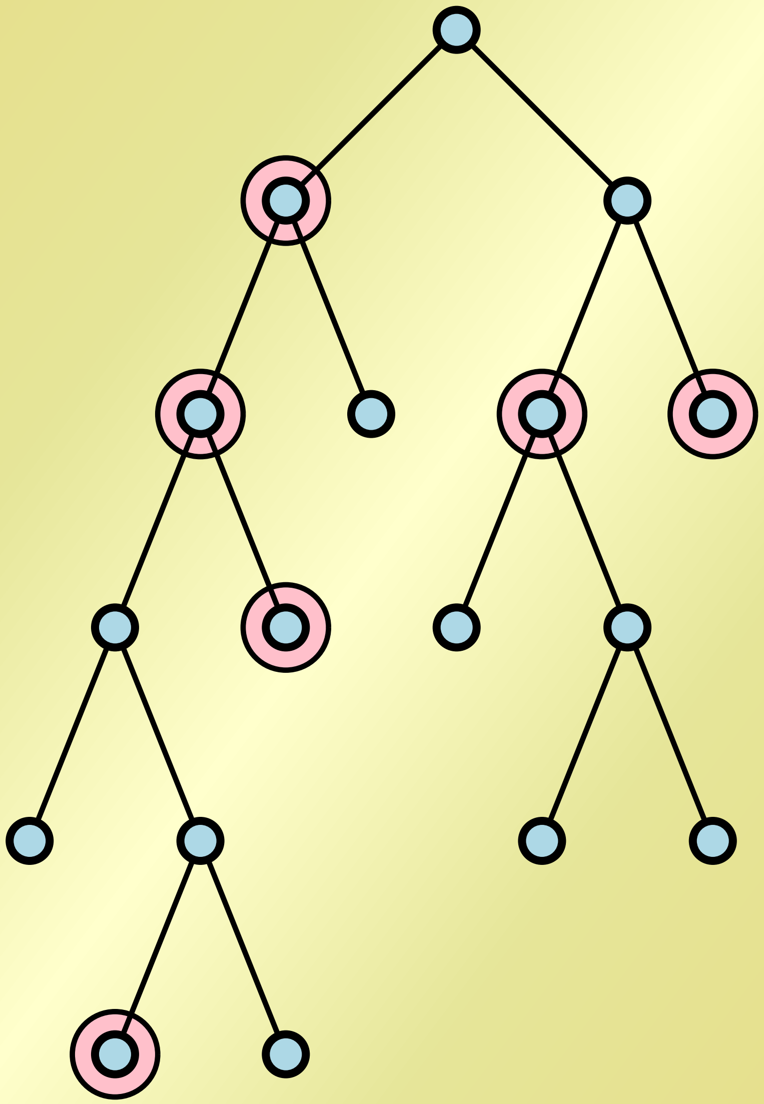


Also, we want to be able to mark and unmark nodes.

We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

For a given query node, we wish to find the first marked ancestor.

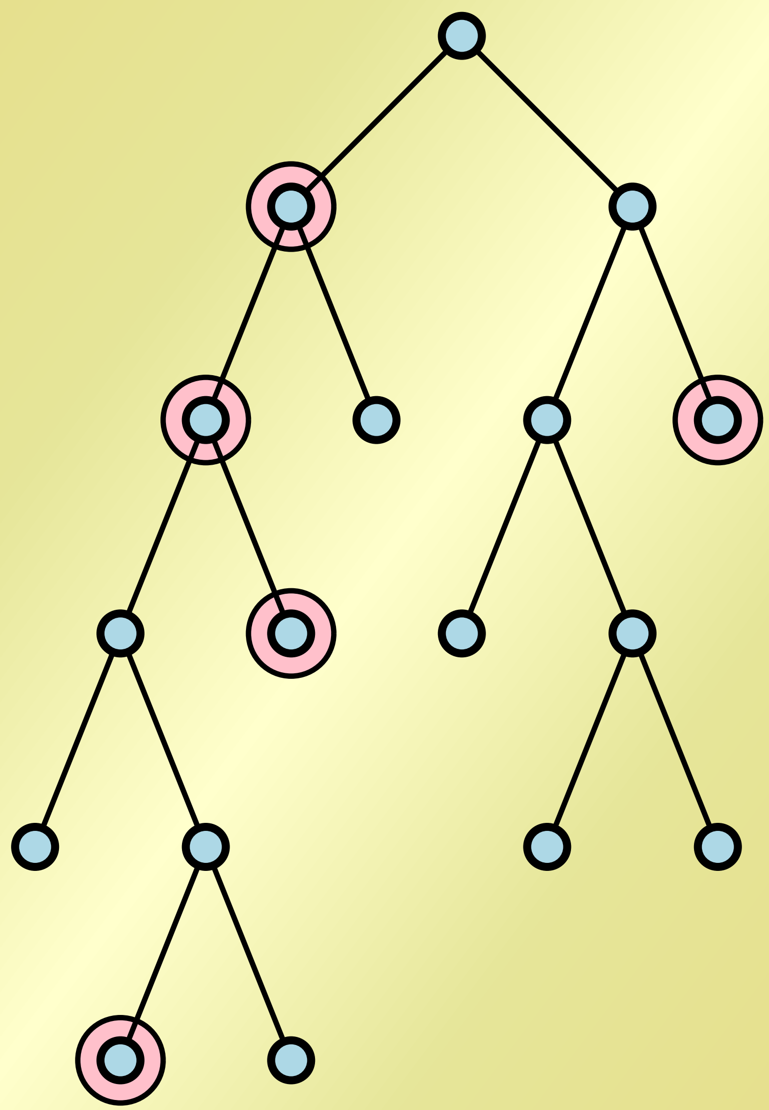


Also, we want to be able to mark and unmark nodes.

We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

For a given query node, we wish to find the first marked ancestor.

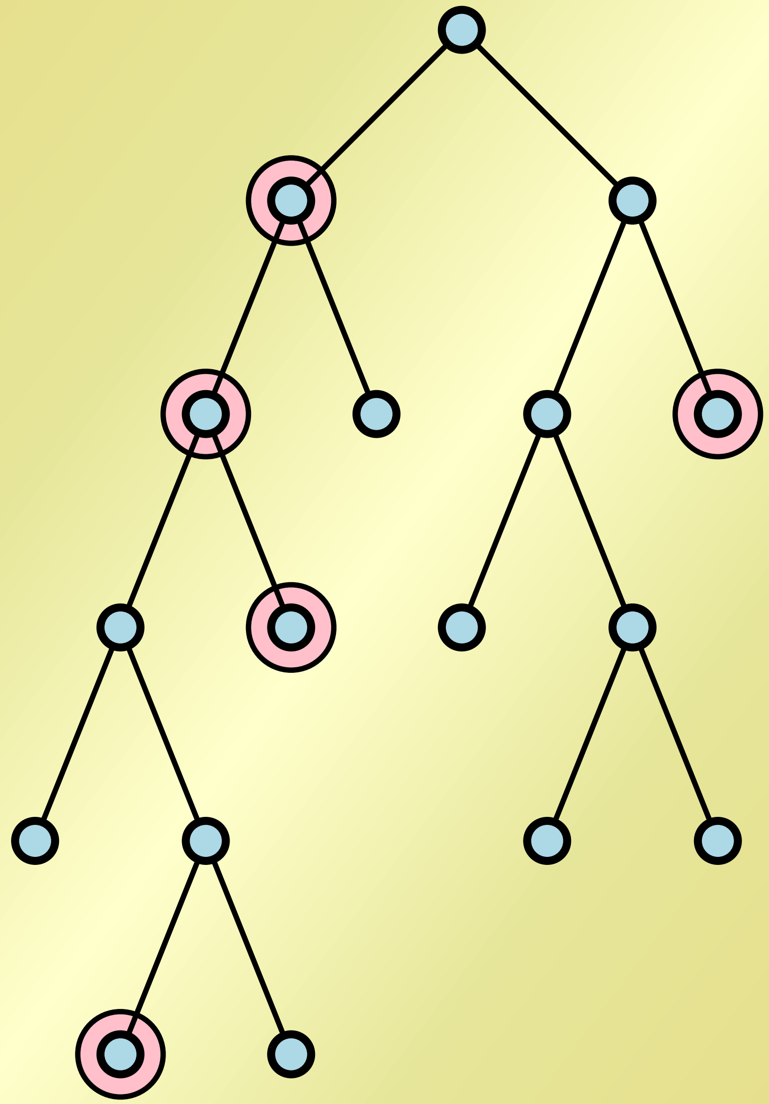


Also, we want to be able to mark and unmark nodes.

We use a *marked ancestor* data structure.

Consider a tree, where some nodes are *marked*.

For a given query node, we wish to find the first marked ancestor.



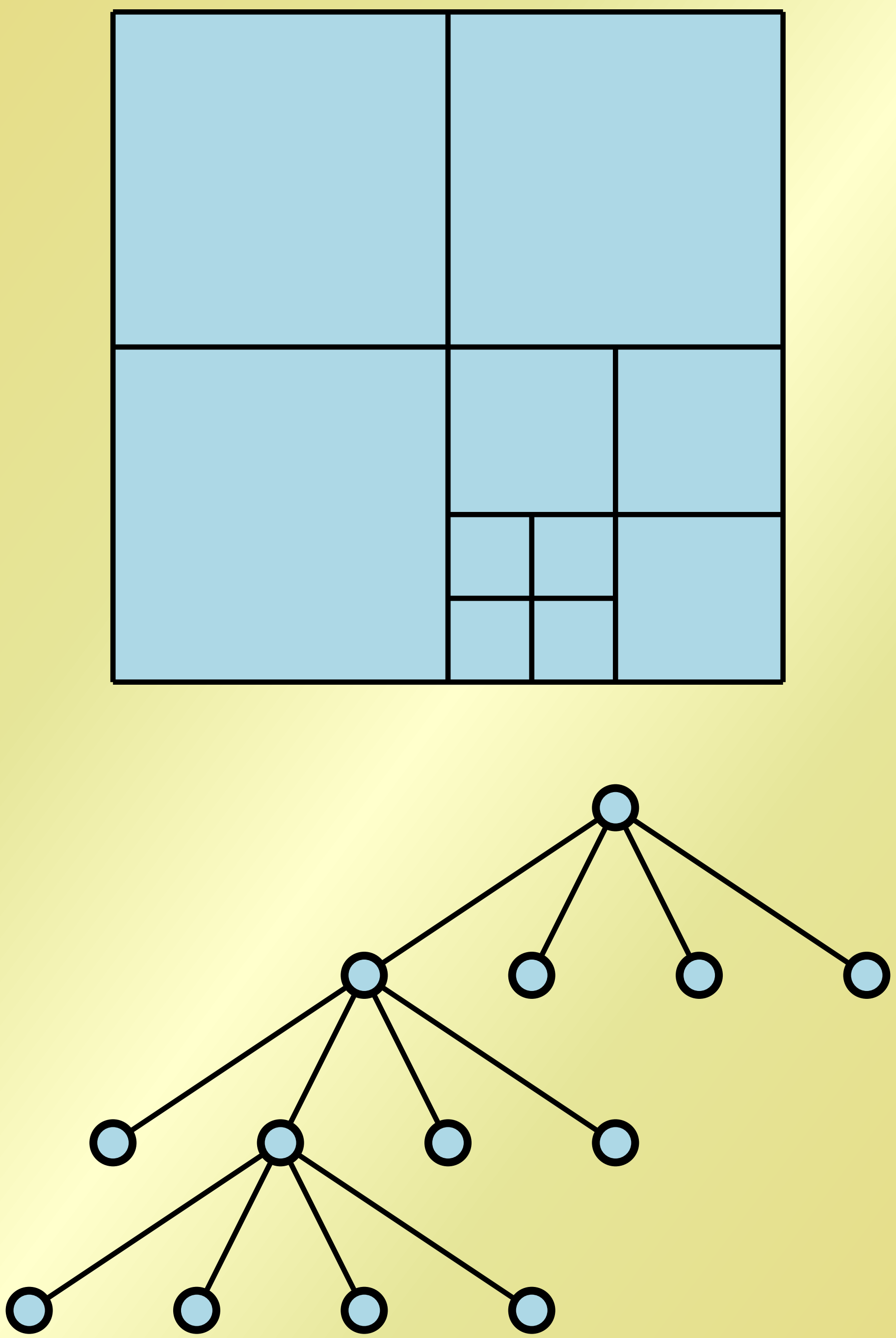
Also, we want to be able to mark and unmark nodes.

$O(\log \log n)$ (un)mark and $O(\frac{\log n}{\log \log n})$ queries is possible.

[Alstrup *et al.*, 1998]

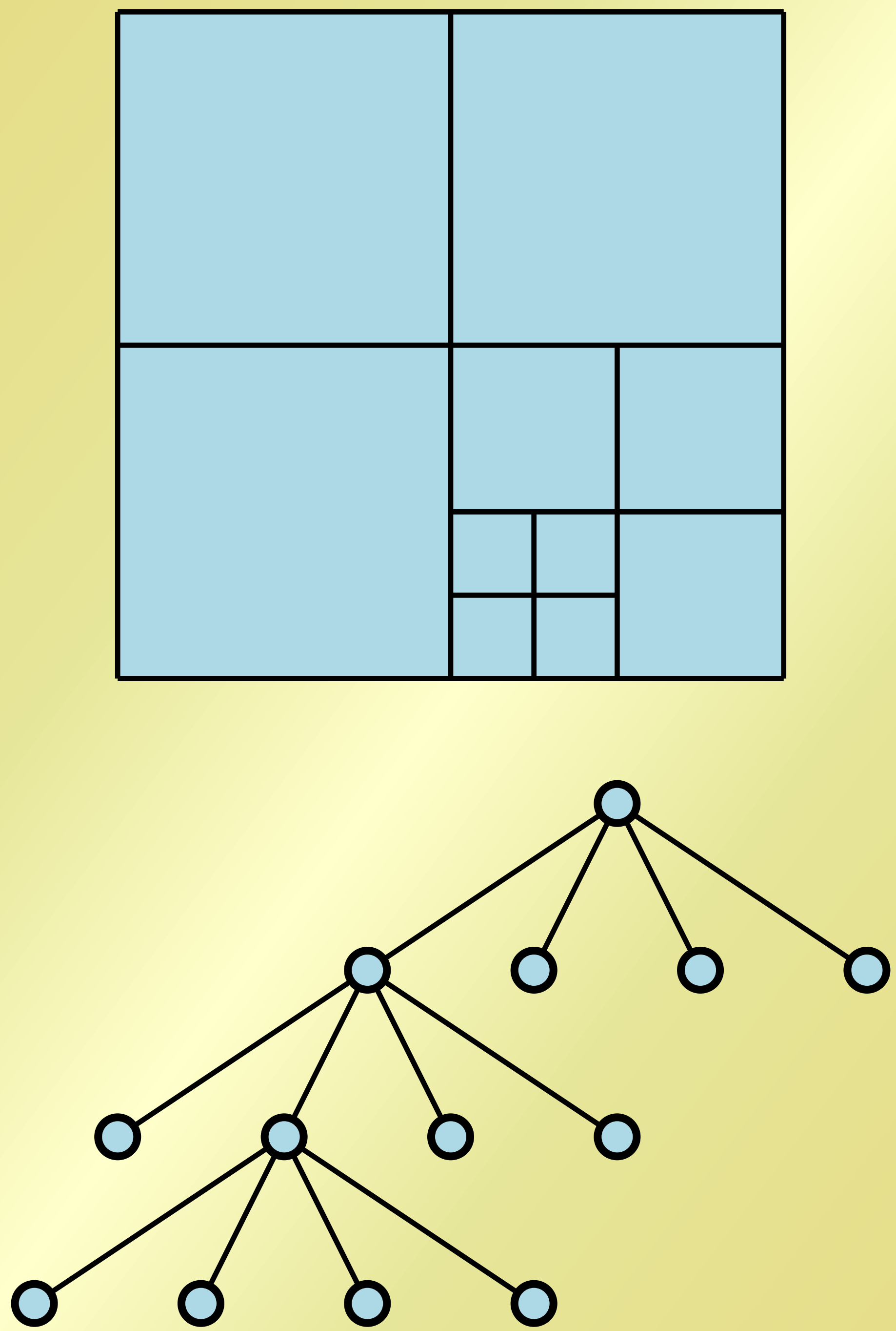
We build 4 MA trees on the quadtree: one for each corner.

We build 4 MA trees on the quadtree: one for each corner.



We build 4 MA trees on the quadtree: one for each corner.

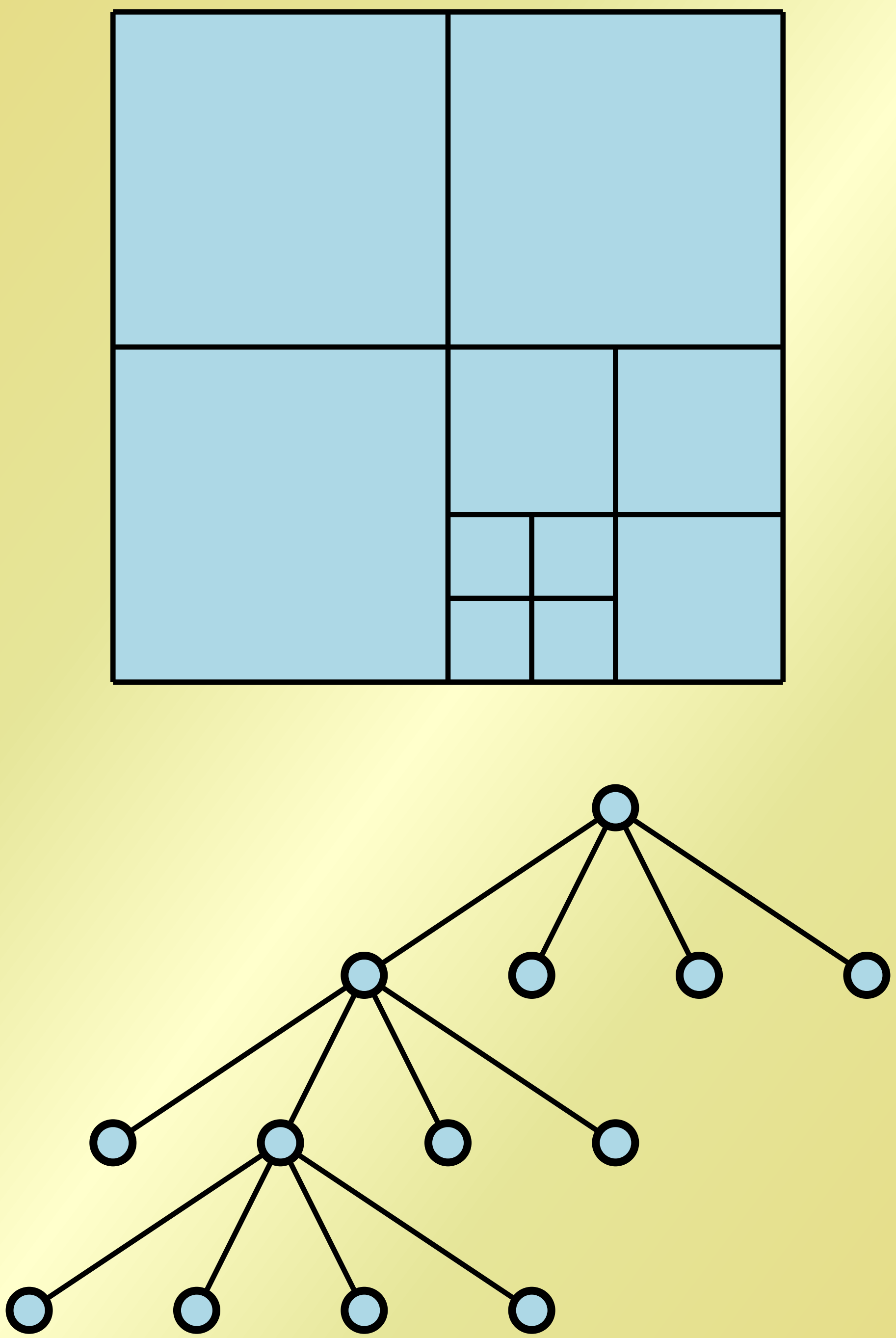
In the TL tree, we mark a cell of the quadtree if . . .



We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if …

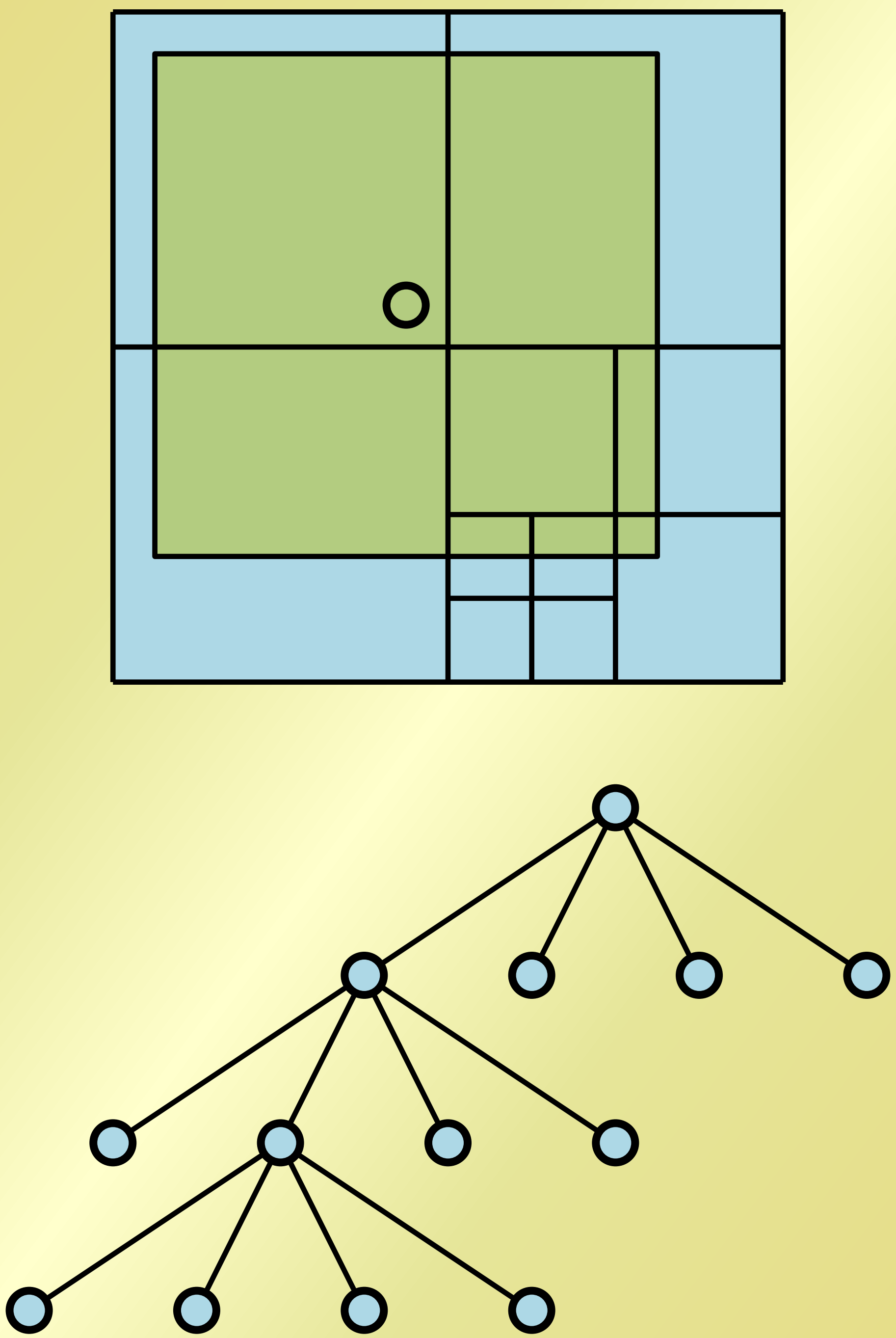
…its top left corner the center point of a region of size $\Theta(|C|)$.



We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if …

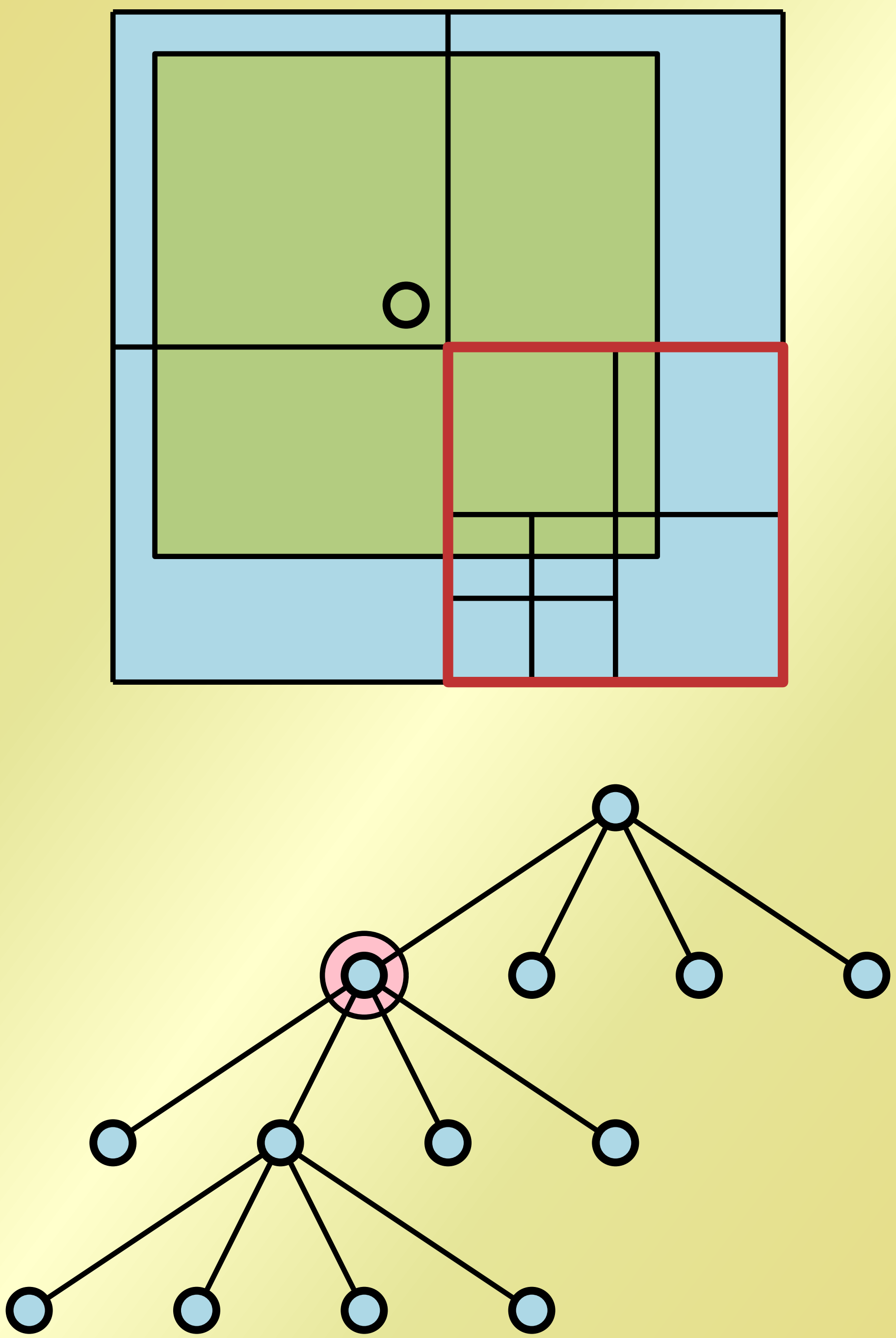
… its top left corner the center point of a region of size $\Theta(|C|)$.



We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

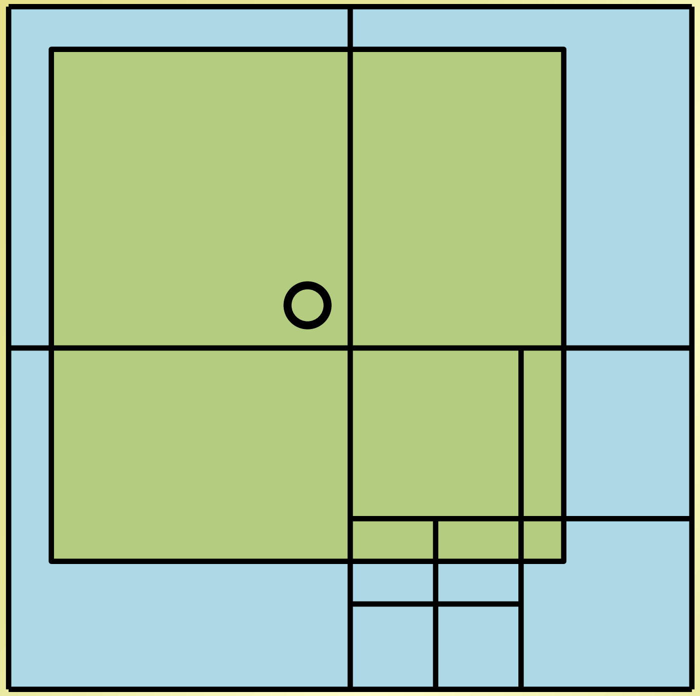
. . . its top left corner the center point of a region of size $\Theta(|C|)$.

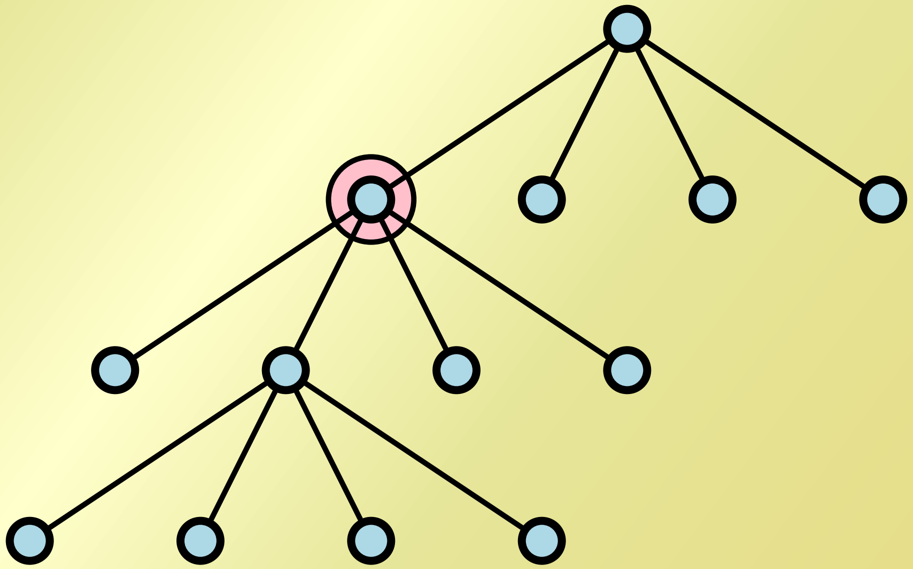


We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

. . . its top left corner the center point of a region of size $\Theta(|C|)$.



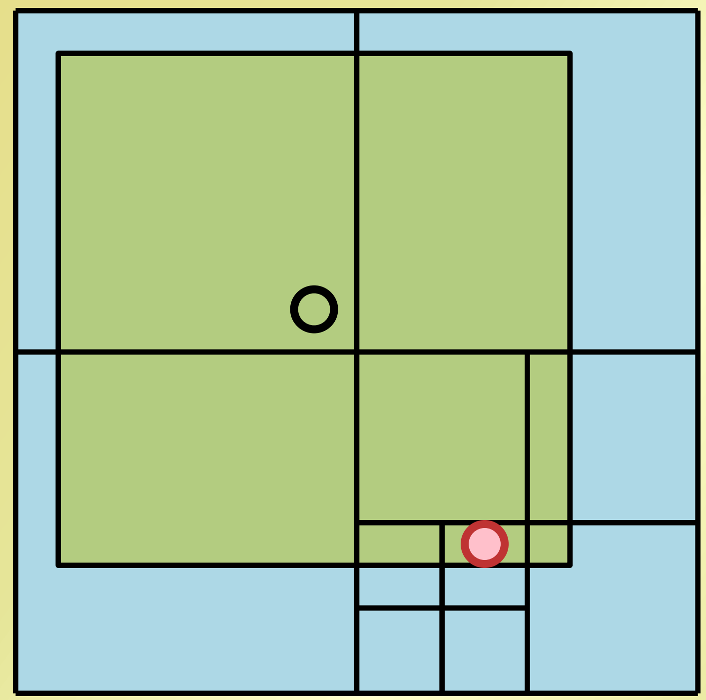


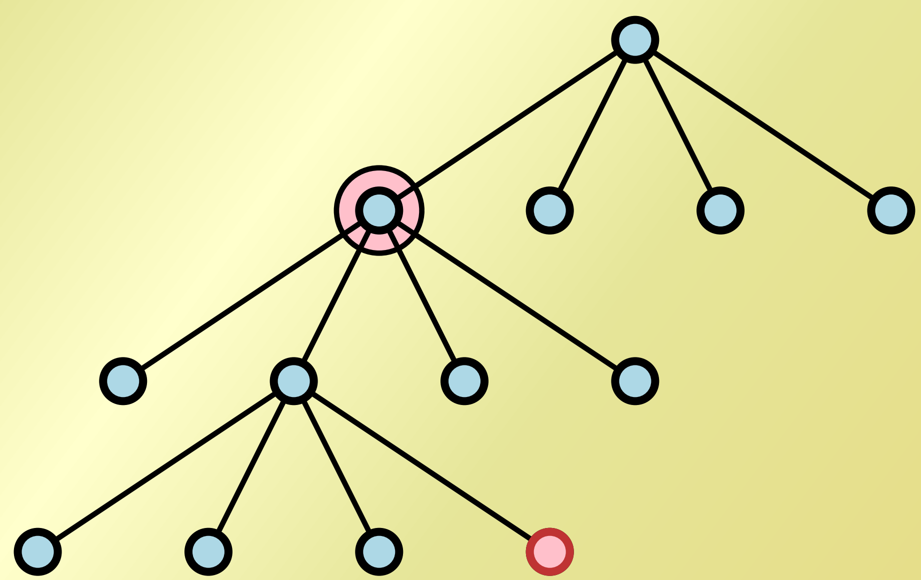
Now, given a query point in a small cell of the quadtree . . .

We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

. . . its top left corner the center point of a region of size $\Theta(|C|)$.



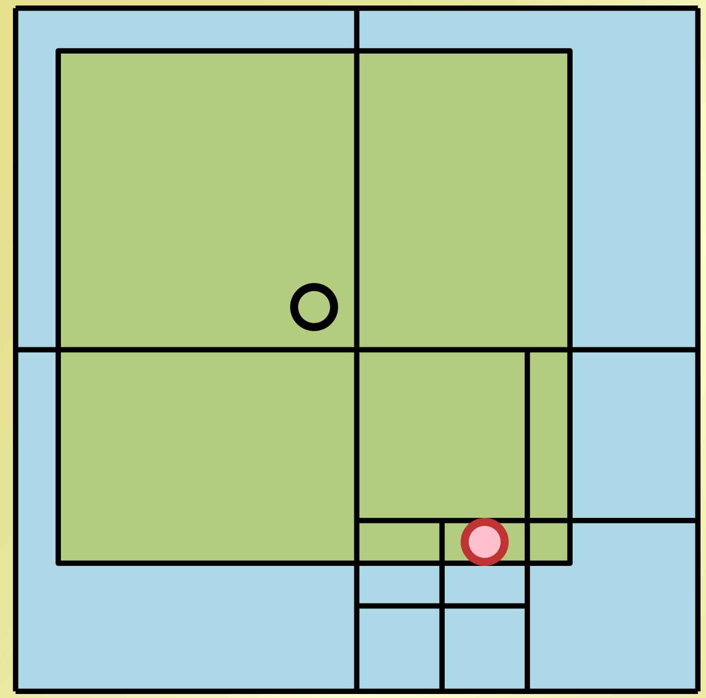


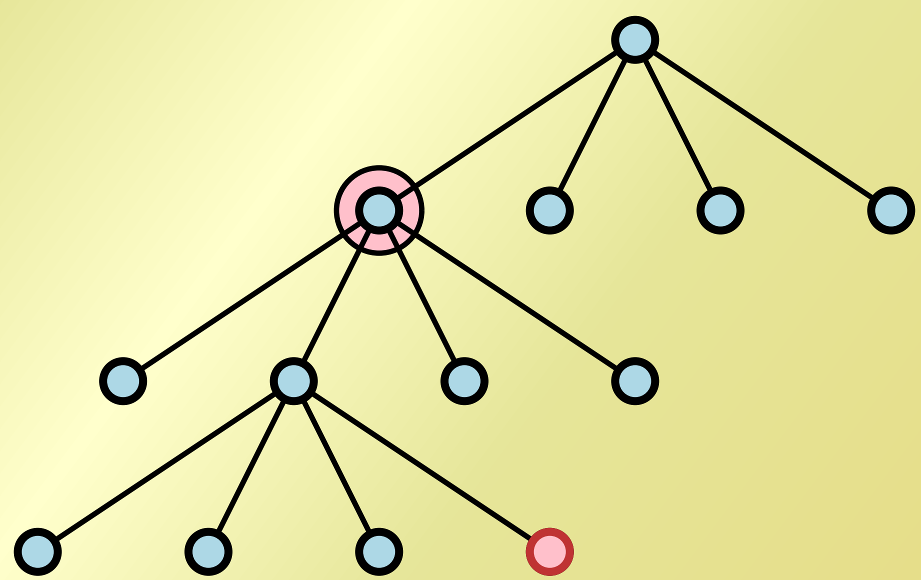
Now, given a query point in a small cell of the quadtree . . .

We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

. . . its top left corner the center point of a region of size $\Theta(|C|)$.





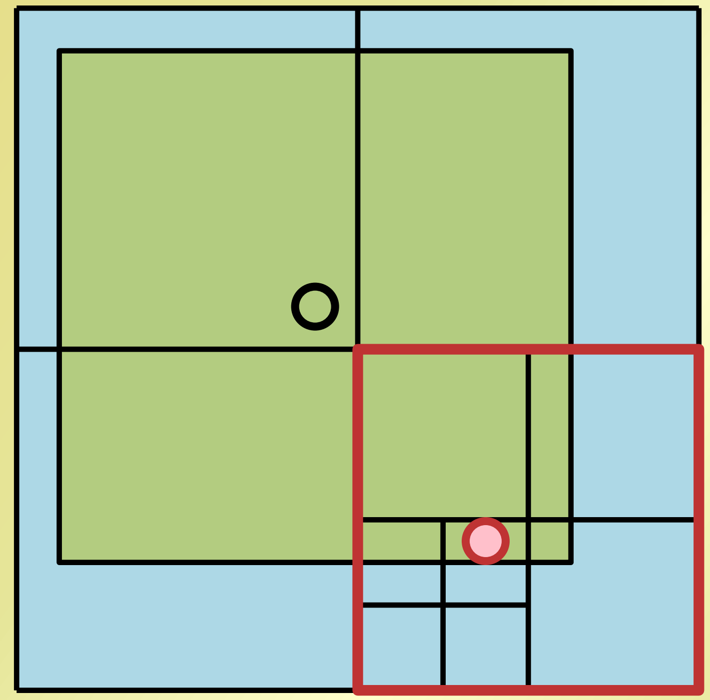
Now, given a query point in a small cell of the quadtree . . .

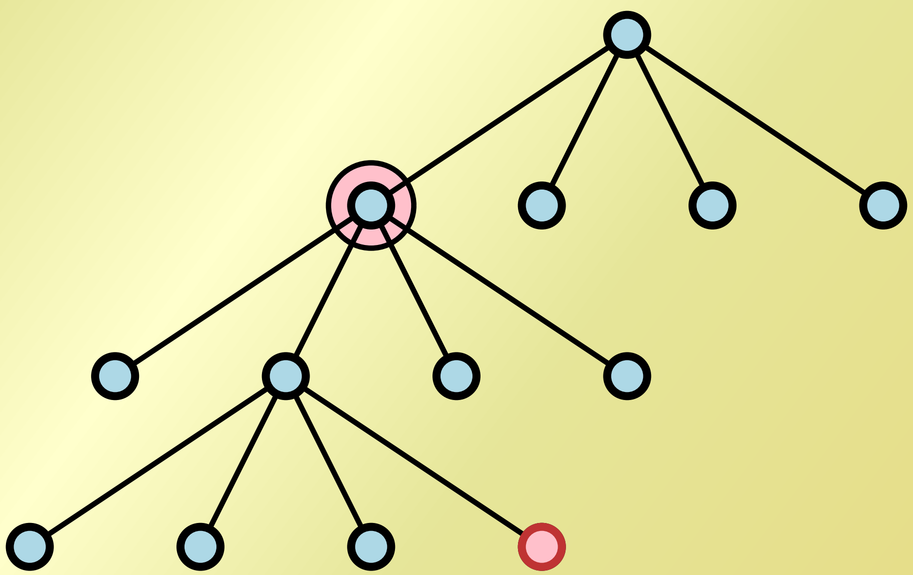
. . . we can quickly find its first marked ancestor.

We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

. . . its top left corner the center point of a region of size $\Theta(|C|)$.





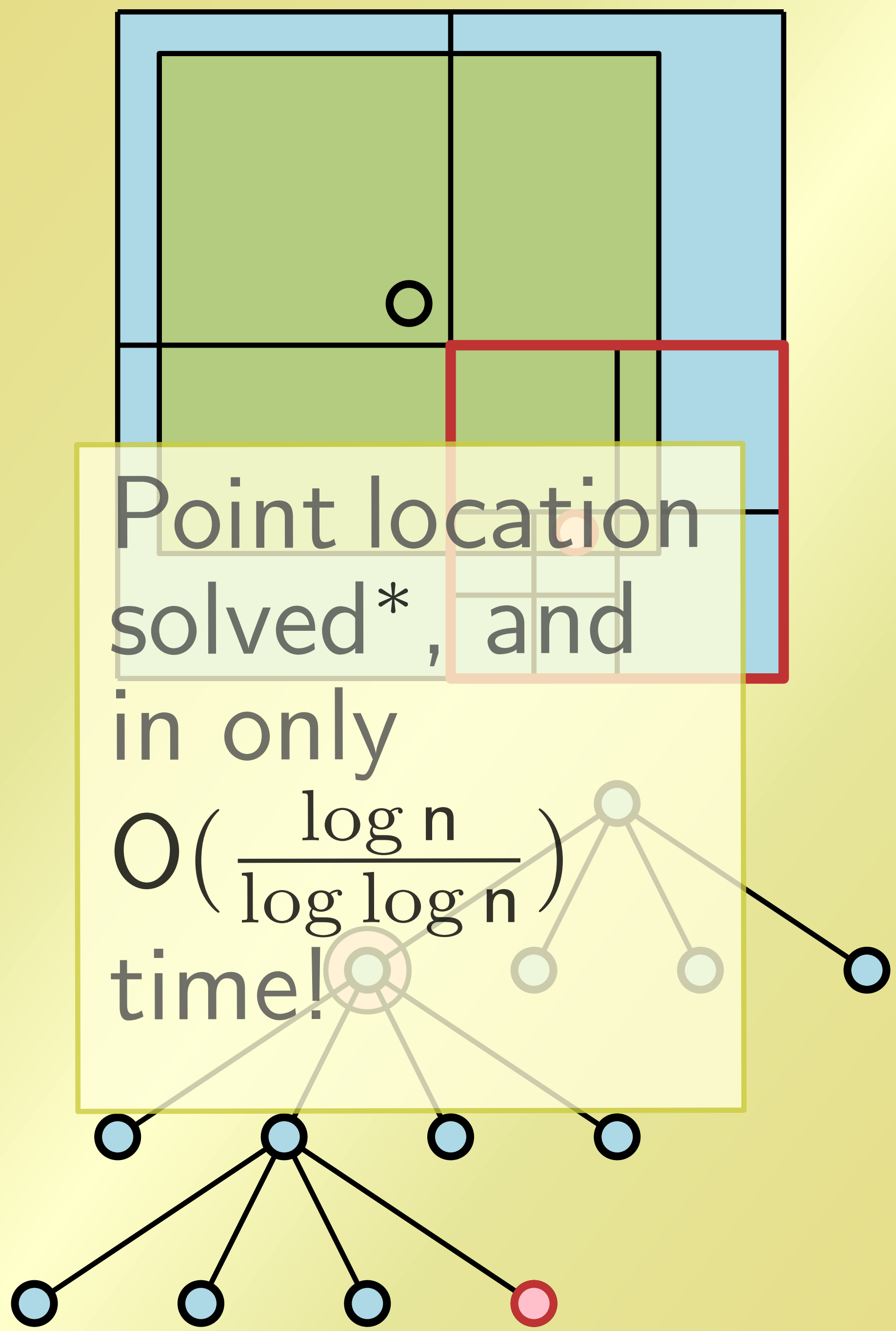
Now, given a query point in a small cell of the quadtree . . .

. . . we can quickly find its first marked ancestor.

We build 4 MA trees on the quadtree: one for each corner.

In the TL tree, we mark a cell of the quadtree if . . .

. . . its top left corner the center point of a region of size $\Theta(|C|)$.



Point location solved*, and in only $O(\frac{\log n}{\log \log n})$ time!

Now, given a query point in a small cell of the quadtree . . .

. . . we can quickly find its first marked ancestor.

* CAUTION! Many details have been swept under the rug. Be extremely careful not to trip when walking on the rug.

CONCLUSIONS

Sublogarithmic updates are indeed possible.

Sublogarithmic updates are indeed possible.

Disjointness seems to be important for our solution to work.

Sublogarithmic updates are indeed possible.

Disjointness seems to be important for our solution to work.

Much of the complexity comes from dealing with large spread.

Sublogarithmic updates are indeed possible.

Disjointness seems to be important for our solution to work.

Much of the complexity comes from dealing with large spread.

OPEN PROBLEM

Can the $O(\frac{\log n}{\log \log n})$ bound be improved?

Sublogarithmic updates are indeed possible.

Disjointness seems to be important for our solution to work.

Much of the complexity comes from dealing with large spread.

OPEN PROBLEM

Can the $O(\frac{\log n}{\log \log n})$ bound be improved?

OPEN PROBLEM

Can we deal with overlapping regions?

Sublogarithmic updates are indeed possible.

Disjointness seems to be important for our solution to work.

Much of the complexity comes from dealing with large spread.

OPEN PROBLEM

Can the $O(\frac{\log n}{\log \log n})$ bound be improved?

OPEN PROBLEM

Can we deal with overlapping regions?

OPEN PROBLEM

Do realistic input assumptions help?

THANKS!

THANKS!

THANKS!

THANKS!