Orthogonal Range Searches

Computational Geometry Week

- This week's lectures are all about problems that arise in computational geometry and how data structures can help solve them.
- This will be, by no means, an exhaustive survey of what's out there.
  - (Stanford's CS268 class goes into way more detail and covers far more ground.)
- However, I hope it gives you a sense of some of the beautiful ideas that arise when exploring this space.

Outline for This Week

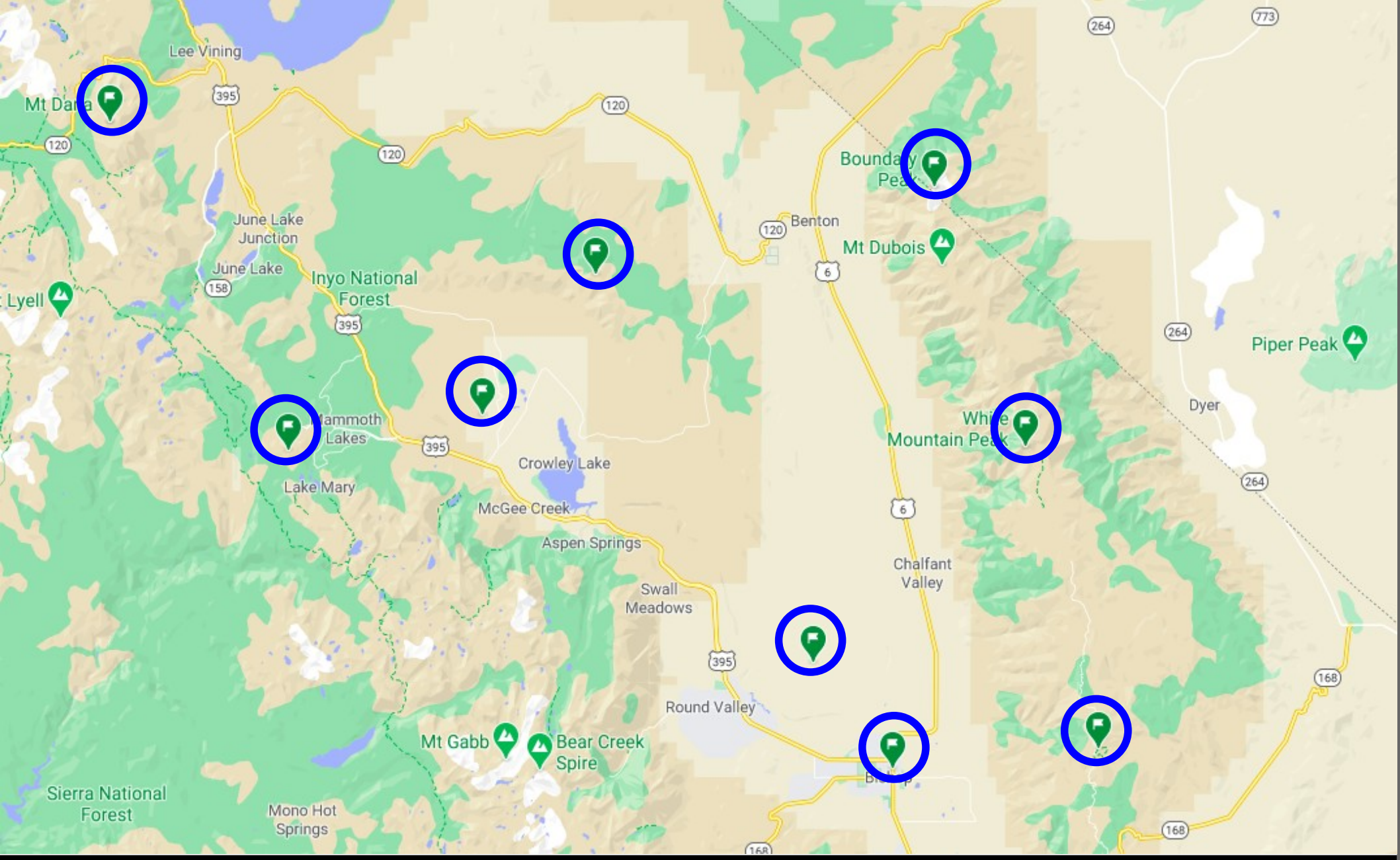
- ***Range Searching (Today)***
  - Finding what points are in a bounding box in 2D and higher.
- ***Point Location (Thursday)***
  - Figuring out which region of space a particular point is in.

Outline for Today

- ***2D Range Searching***
  - Rectangular queries in 2D space.
- ***Blocking Revisited***
  - Speeding up 2D range searches by chopping up space.
- ***Range Trees***
  - Taking blocking in a different direction.
- ***Geometric Cascading***
  - Reducing the costs of repeated binary searches.
- ***Higher Dimensional Searches***
  - Looking in 3D and higher.

2D Orthogonal Range Searching

How can we quickly find all points worth displaying?

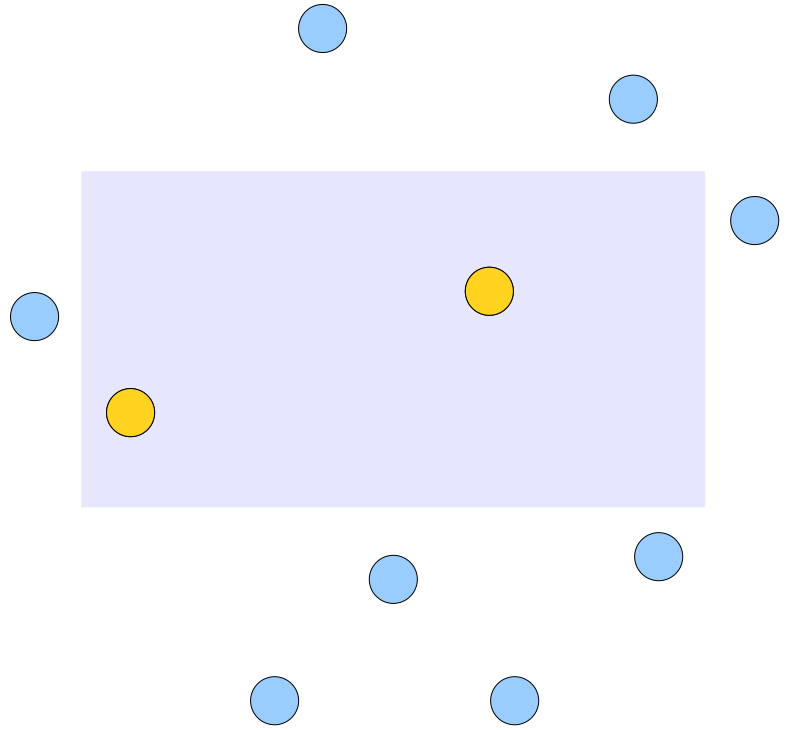


How can we quickly find all points worth displaying?

2D Orthogonal Range Searching

- We are given a collection of n points in 2D space, represented as (x, y) pairs.
- ***Goal:*** Preprocess the points to support queries of the form

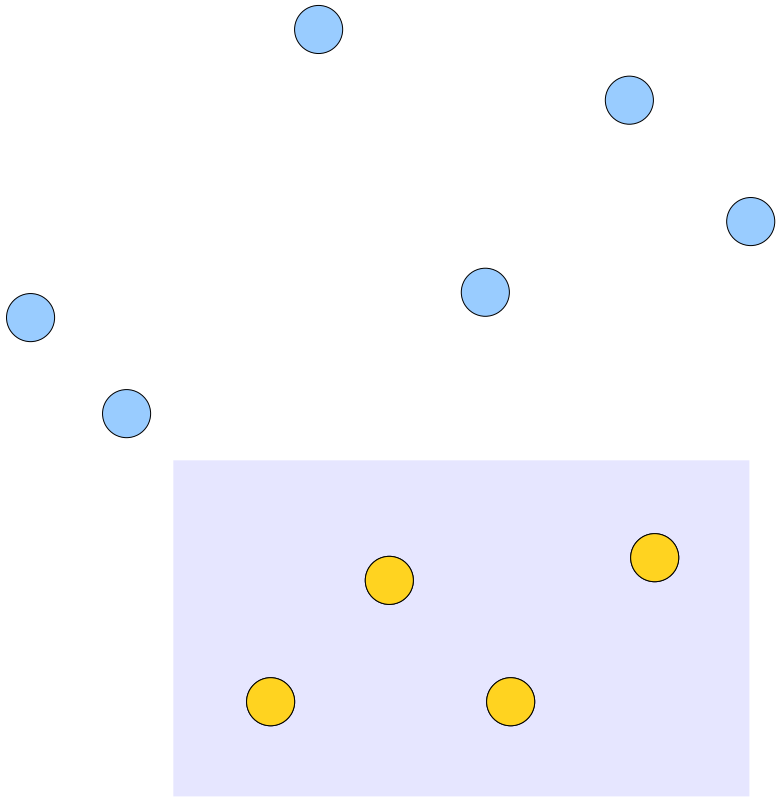
List all points whose x coordinate is between x_{min} and x_{max} and whose y coordinate is between y_{min} and y_{max}.



2D Orthogonal Range Searching

- We are given a collection of n points in 2D space, represented as (x, y) pairs.
- ***Goal:*** Preprocess the points to support queries of the form

List all points whose x coordinate is between x_{min} and x_{max} and whose y coordinate is between y_{min} and y_{max}.

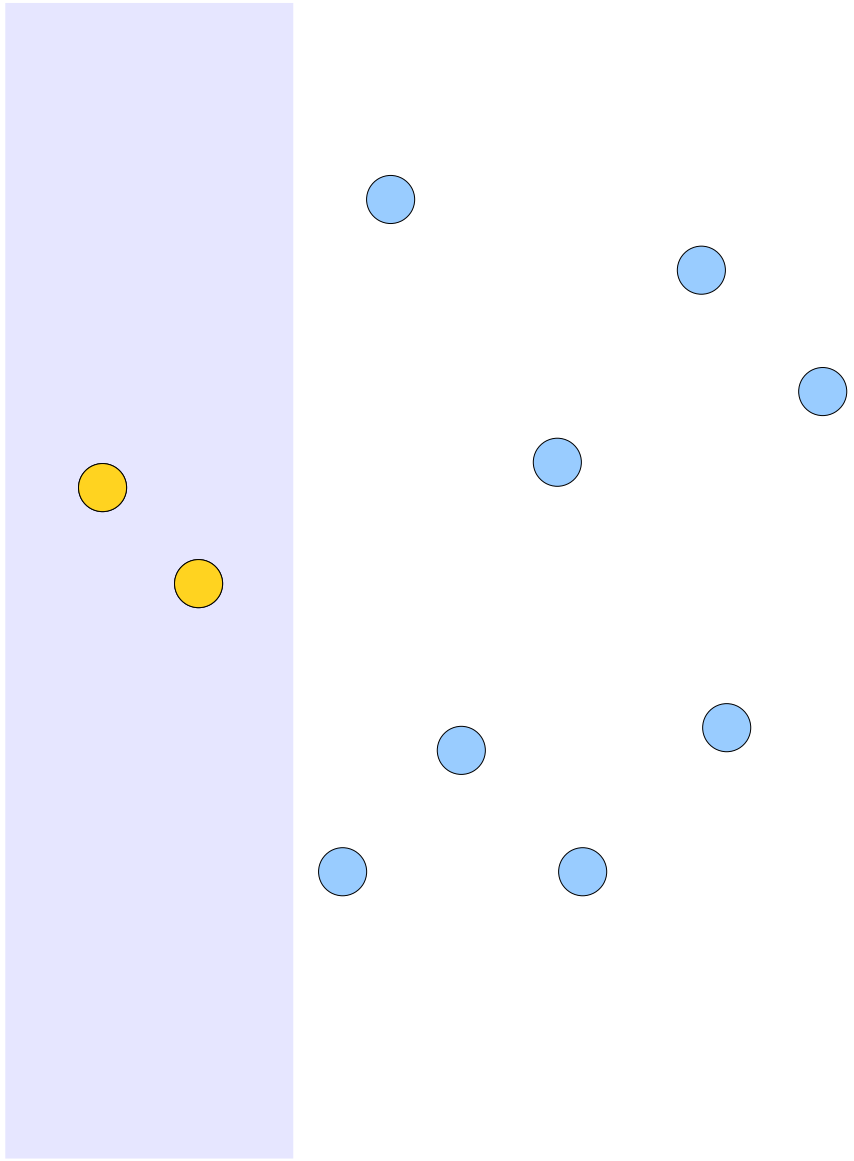


2D Orthogonal Range Searching

- We are given a collection of n points in 2D space, represented as (x, y) pairs.
- ***Goal:*** Preprocess the points to support queries of the form

  List all points whose x coordinate is between x_{min} and x_{max} and whose y coordinate is between y_{min} and y_{max}.

- As with RMQ, we expect to see some tradeoffs between preprocessing time and runtime.
- ***Question:*** How quickly can we solve this problem?



Tracking Our Solutions

- There are three separate resources we're going to keep track of today.
  - ***Preprocessing Time:*** How much up-front work do we have to do?
  - ***Query Time:*** How quickly can we query for the points in a space?
  - ***Space Usage:*** How much memory is required to store our data structure?
- That last one is new, but important to keep in mind today.

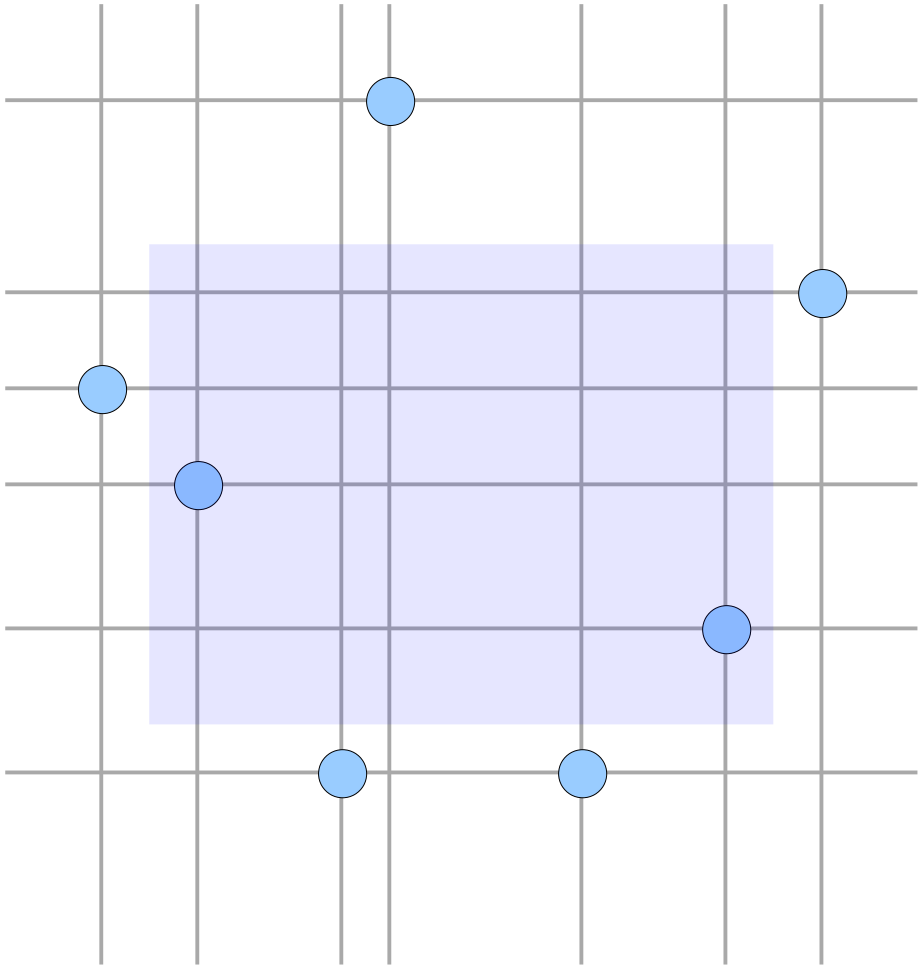
A Naive Solution

- Just for reference: what happens if we store the n points and iterate over all of them whenever there's a match?
  - Preprocessing time: O(1).
  - Query time: O(n).
  - Space usage: O(n).
- This is fine if n is small, but can't scale to, say, Google Maps.
- ***Question:*** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Linear Scan	O(1)	O(n)	O(n)

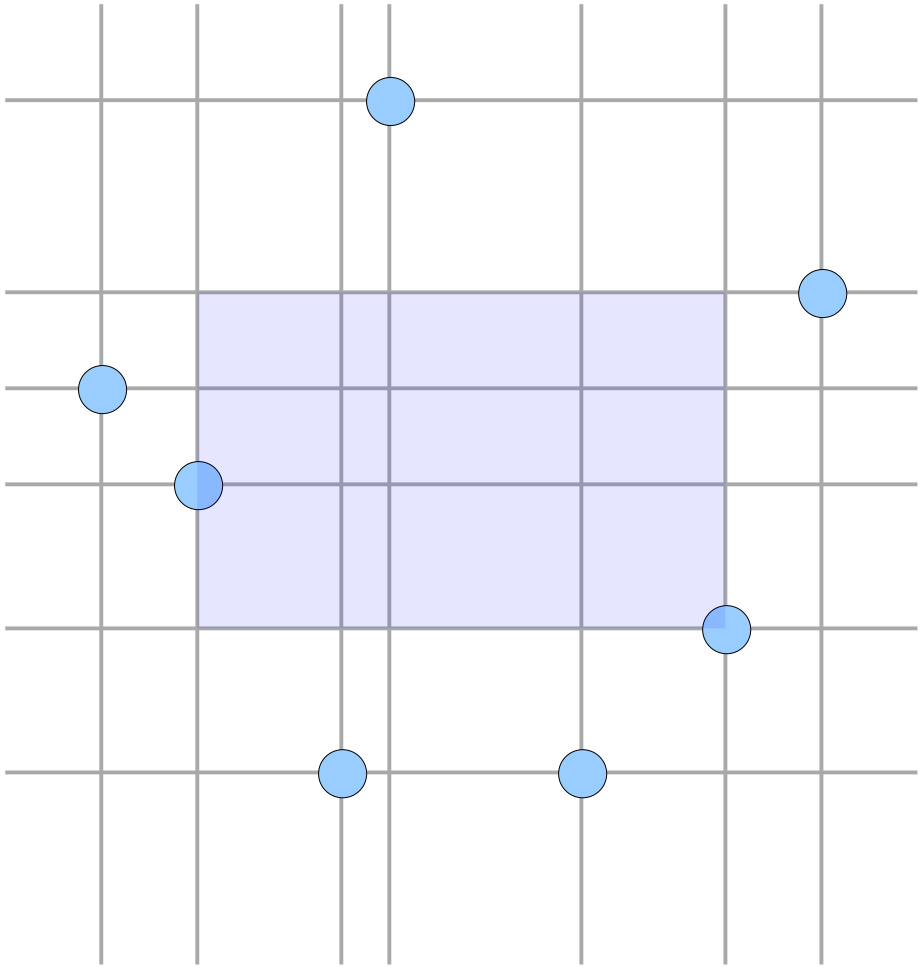
Another Naive Solution

- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.



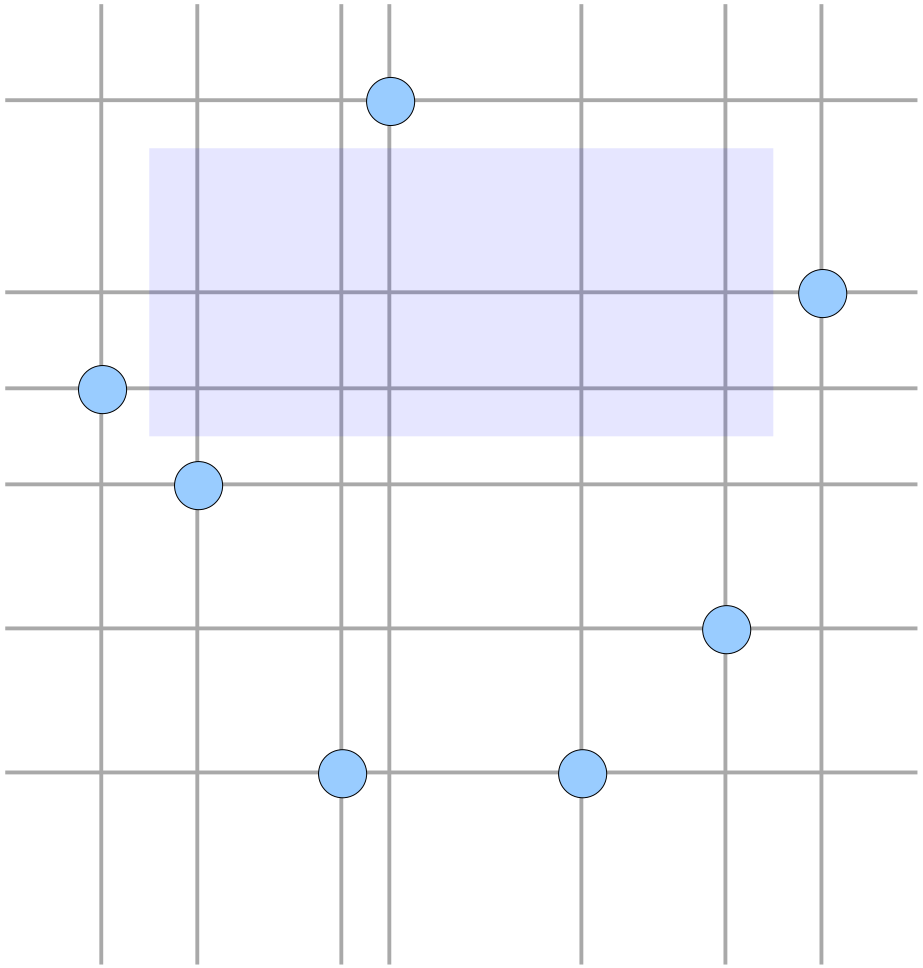
Another Naive Solution

- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.



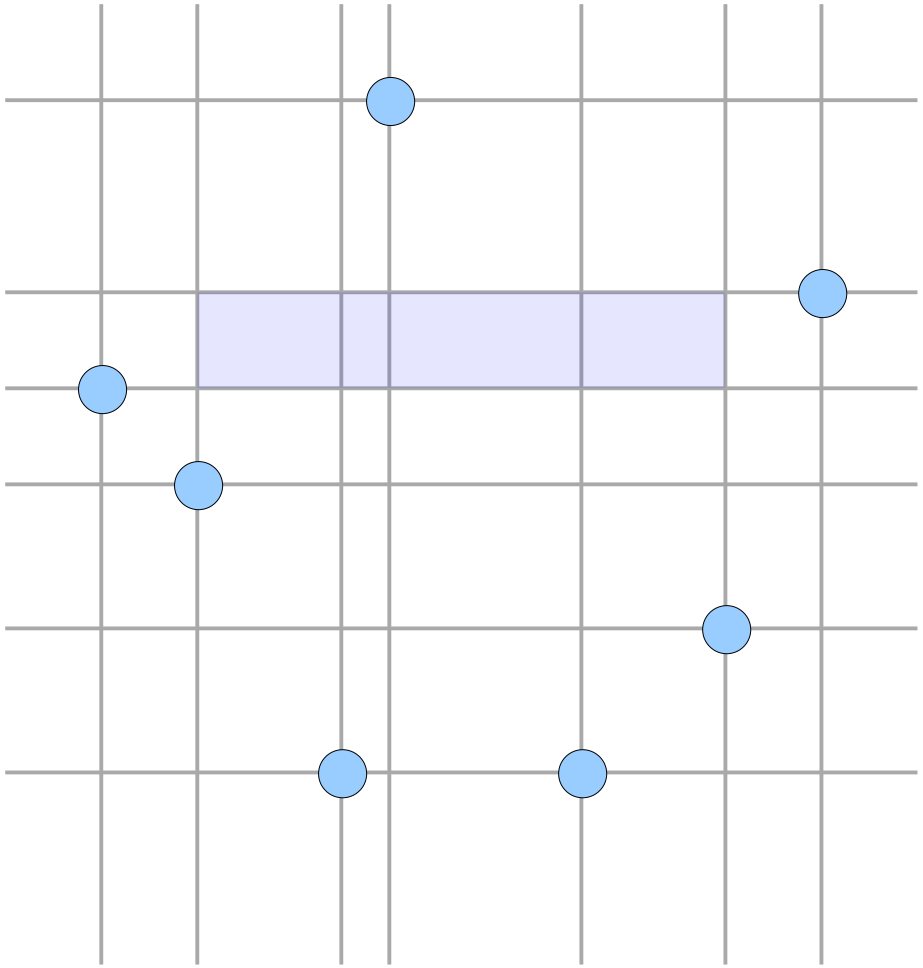
Another Naive Solution

- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.



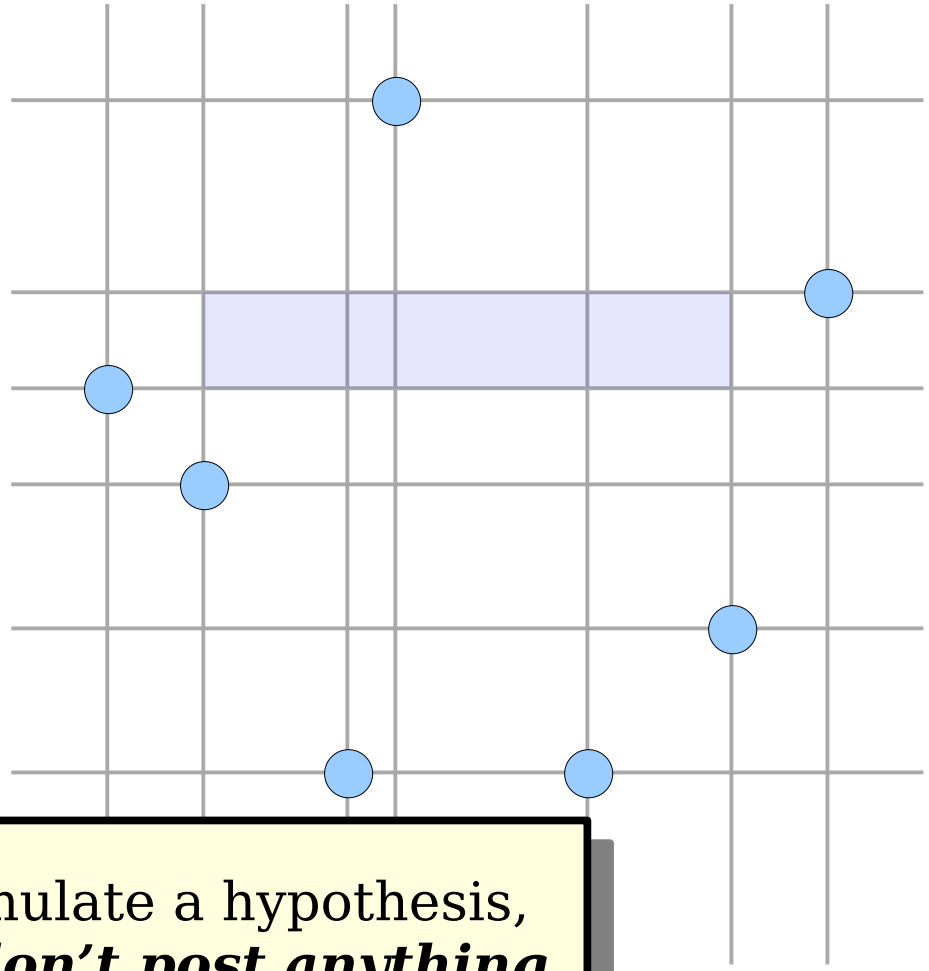
Another Naive Solution

- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.



Another Naive Solution

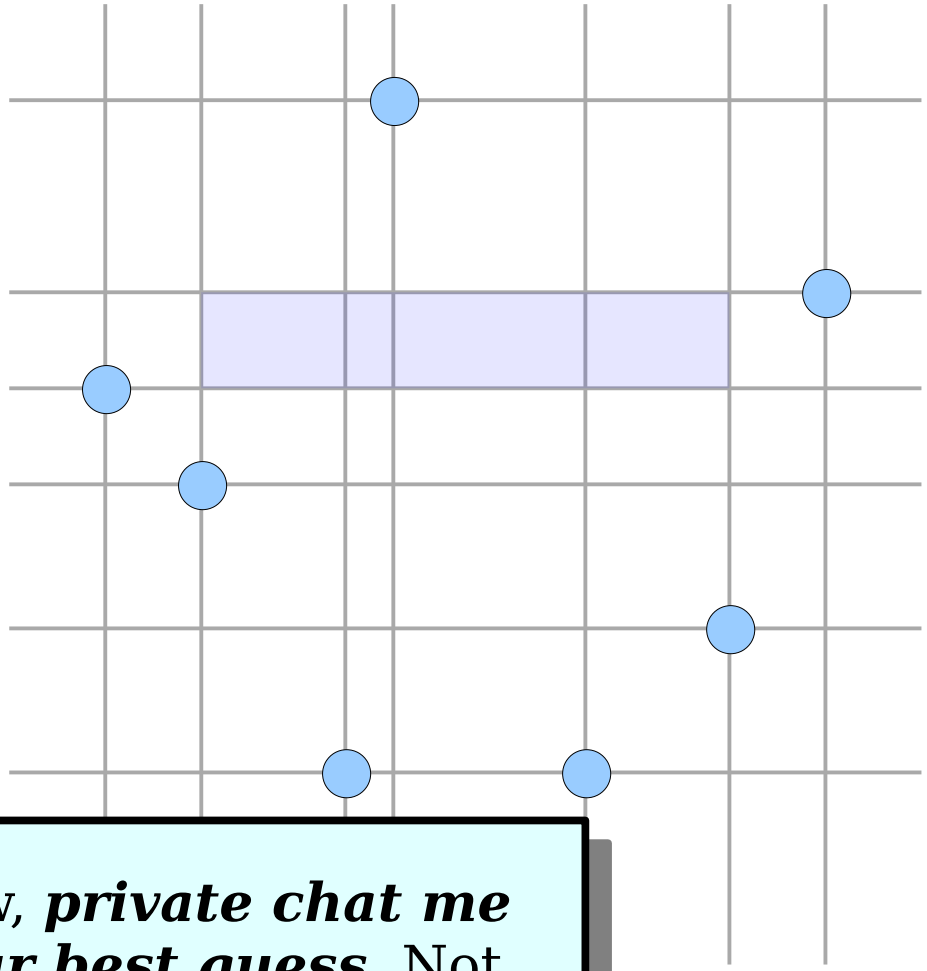
- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.
- ***Question:*** Asymptotically, how many rectangles are there, as a function of n?



Formulate a hypothesis, but ***don’t post anything in chat just yet***.

Another Naive Solution

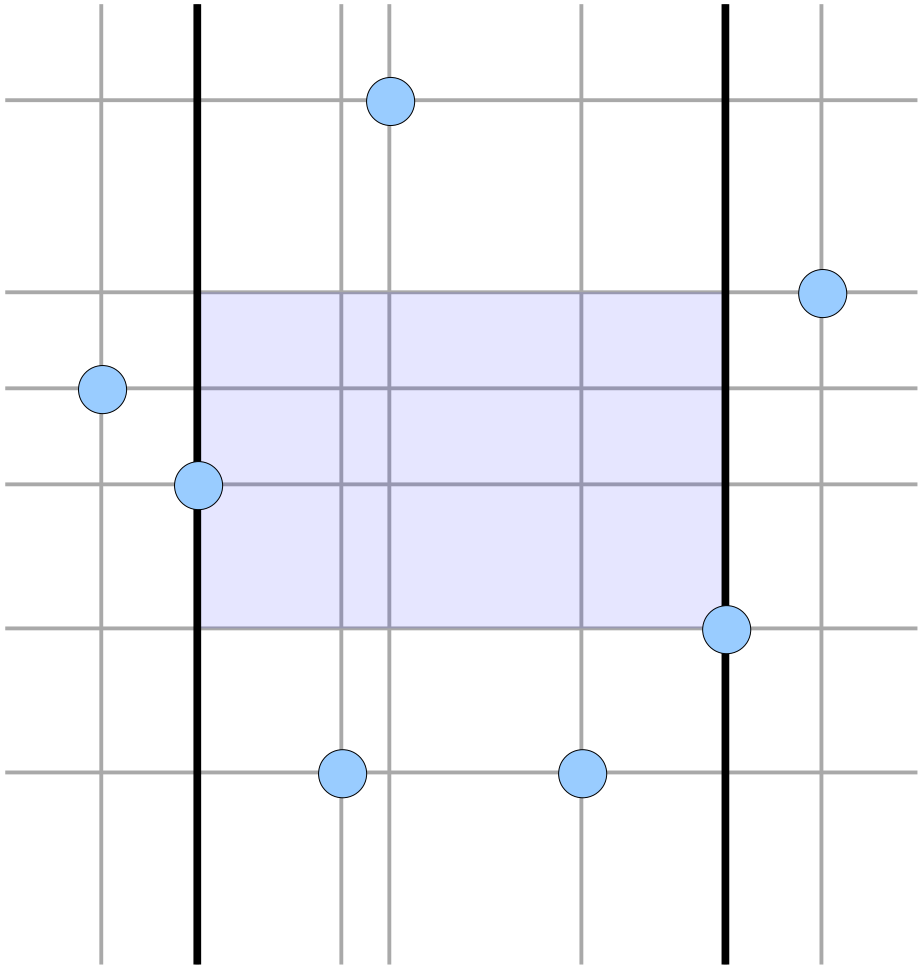
- What would a “precompute-all” solution look like?
- Suppose we have n points. They form an $n \times n$ grid with non-uniform spacing.
  - Could be smaller if there are duplicated x or y coordinates.
- Every query essentially corresponds to a rectangle in this space.
- ***Question:*** Asymptotically, how many rectangles are there, as a function of n?



Now, ***private chat me your best guess***. Not sure? Just answer “??”.

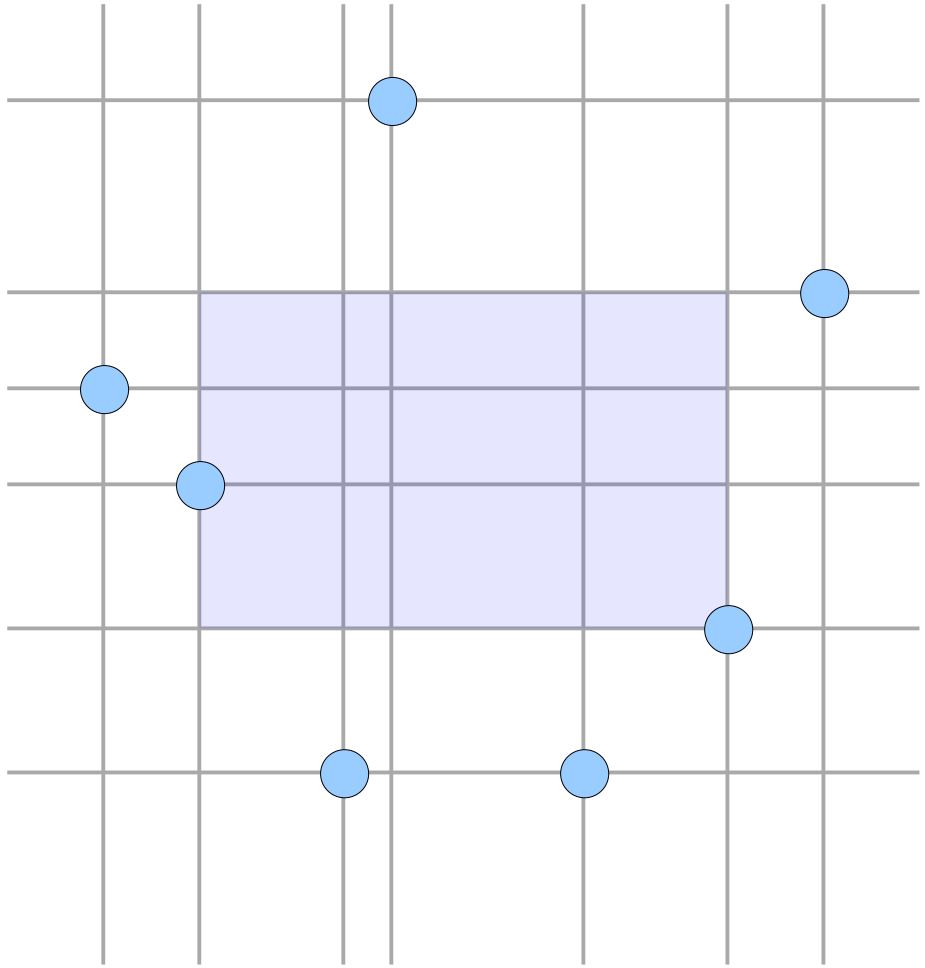
Another Naive Solution

- For now, just focus on the x axis.
- Any rectangle on this grid has a left and right endpoint.
- There are $O(n^2)$ ranges in the x direction, using reasoning analogous to what we saw with RMQ.
- Factoring in both dimensions, this gives us **$O(n^4)$** possible queries.



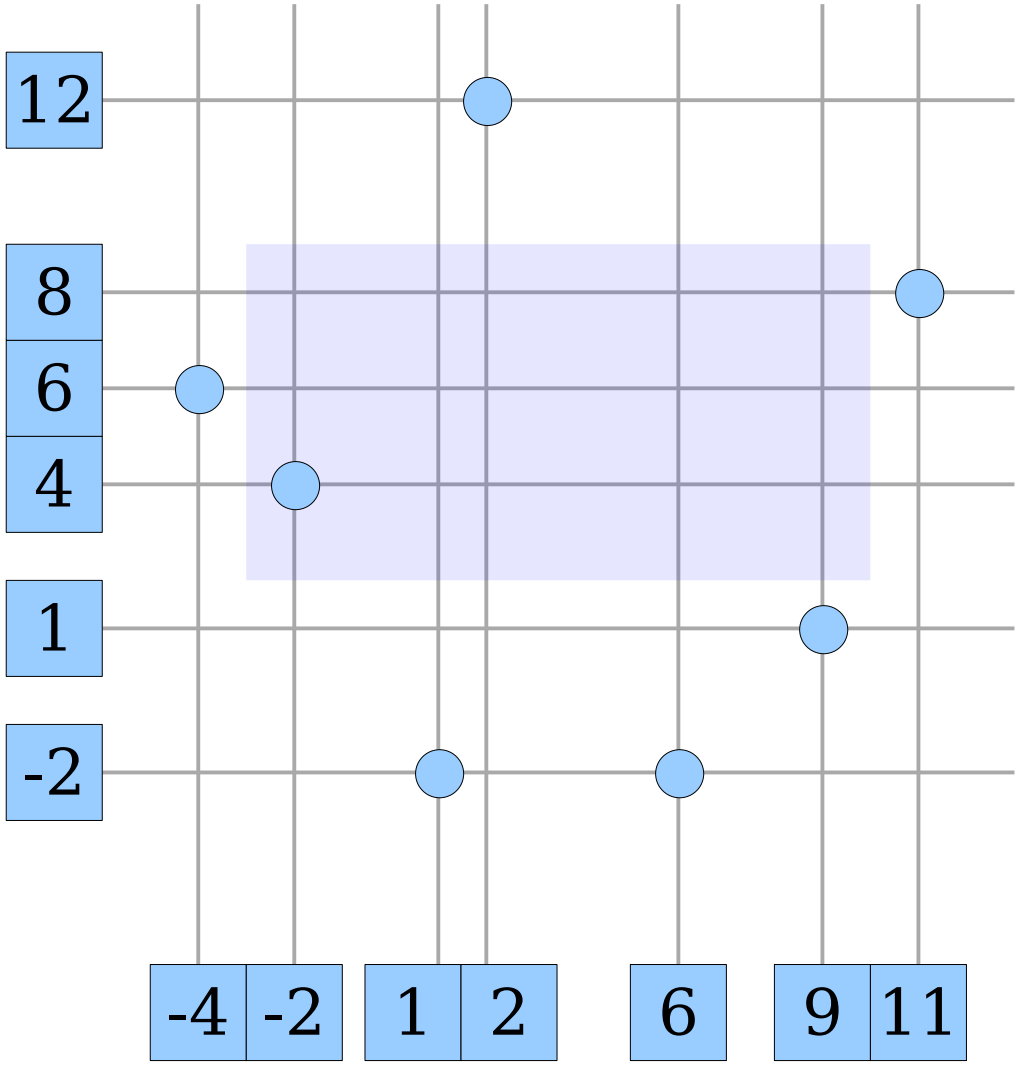
Another Naive Solution

- ***Idea:*** Construct a 4D table holding all possible queries.
- Each point would appear in $O(n^4)$ of those rectangles.
  - Do you see why?
- Total space usage: **$O(n^5)$**.
- We can populate that table in time $O(n^5)$.



Another Naive Solution

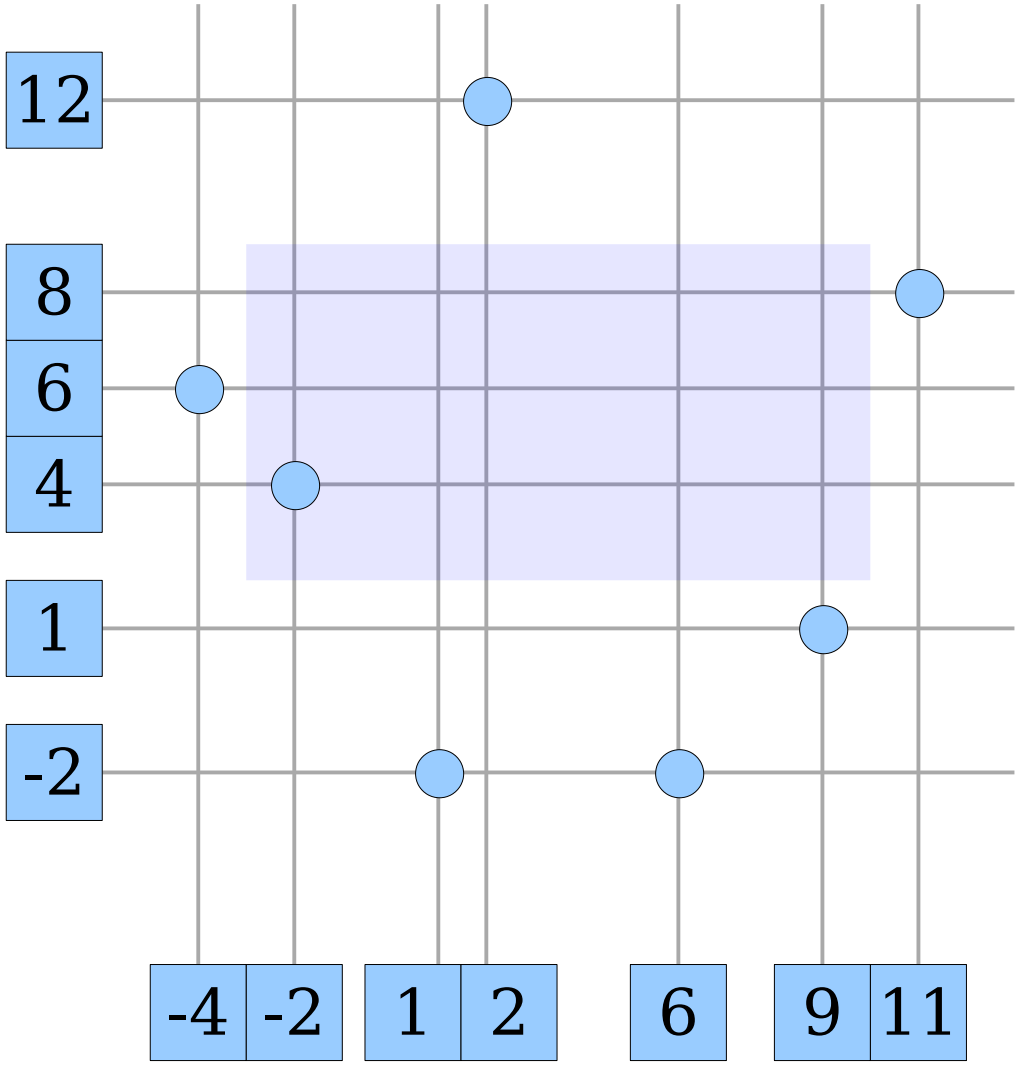
- How would we perform queries using this data structure?
- Write down the x and y coordinates of all the separators as sorted arrays.
  - Adds $O(n \log n)$ preprocessing time; that's nothing compared to our $O(n^5)$ work so far!
- Given a query, use four binary searches convert coordinates to grid indices.
- Look up the corresponding rectangle in our 4D table.



Another Naive Solution

- ***Recall:*** A lookup works by doing four binary searches on these arrays to snap to a grid, then returns the precomputed answer.
- ***Question:*** What is the cost of this approach?

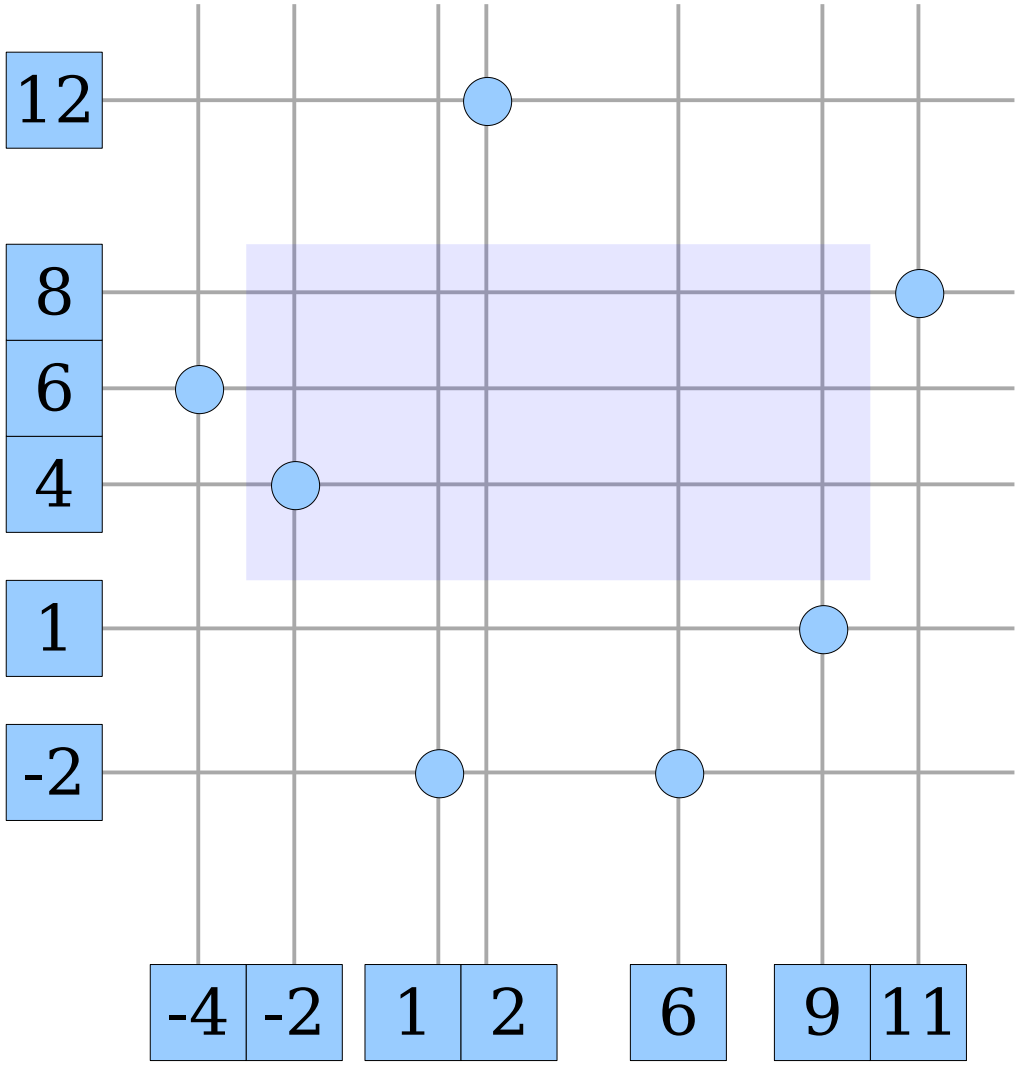
Formulate a hypothesis, but ***don't post anything in chat just yet***.



Another Naive Solution

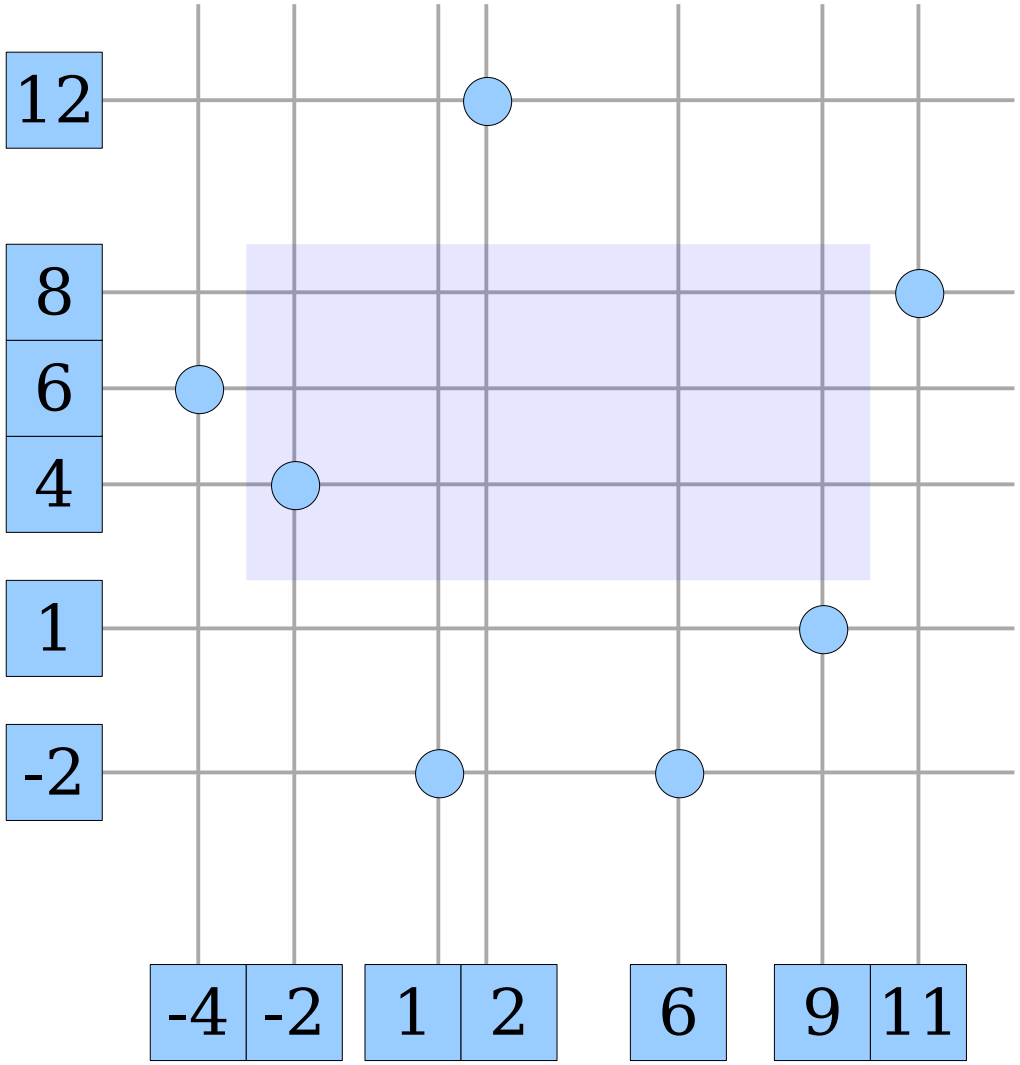
- ***Recall:*** A lookup works by doing four binary searches on these arrays to snap to a grid, then returns the precomputed answer.
- ***Question:*** What is the cost of this approach?

Now, ***private chat me your best guess***. Not sure? Just answer “??”.



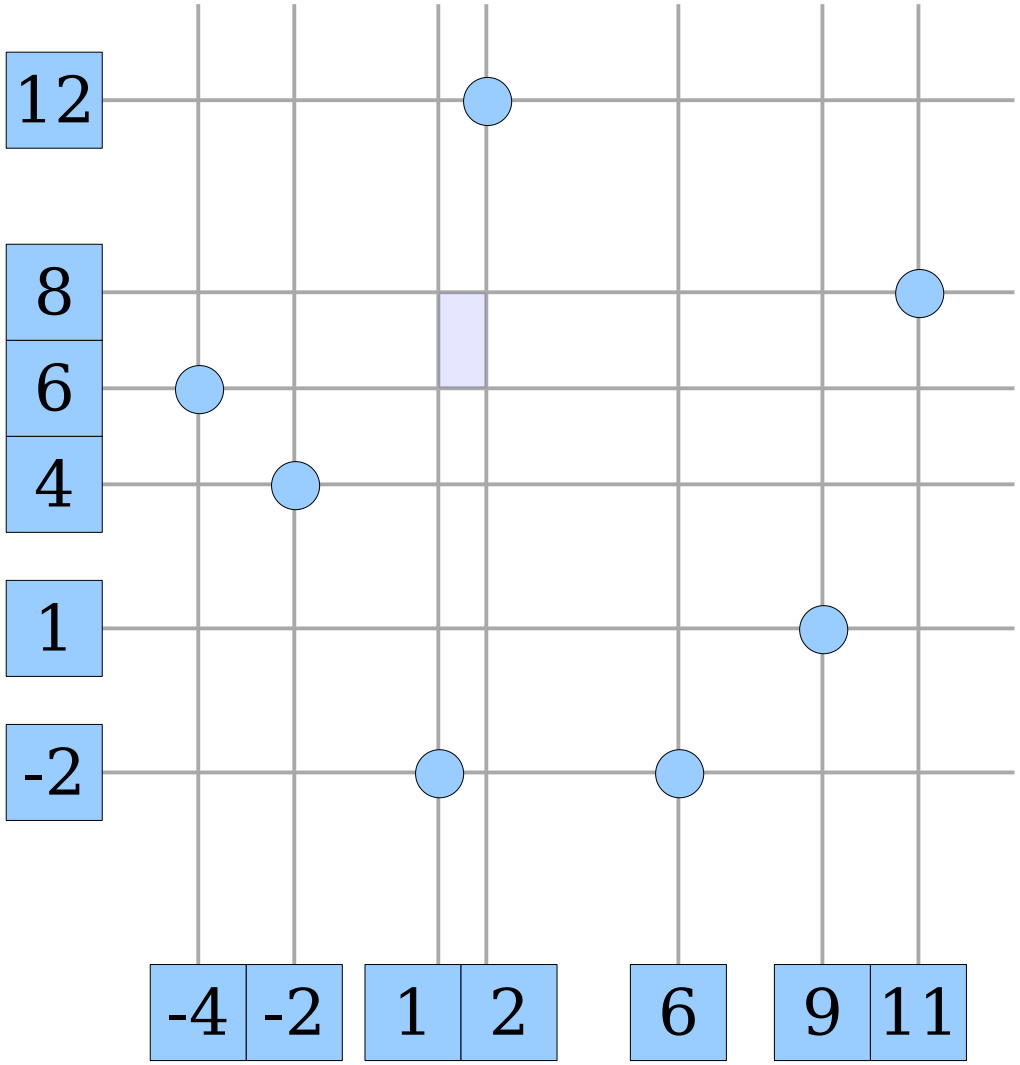
Another Naive Solution

- ***Recall:*** A lookup works by doing four binary searches on these arrays to snap to a grid, then returns the precomputed answer.
- ***Question:*** What is the cost of this approach?
- Each binary search takes time O(log *n*).
- However, the time required to return the answer varies based on how many points we return.



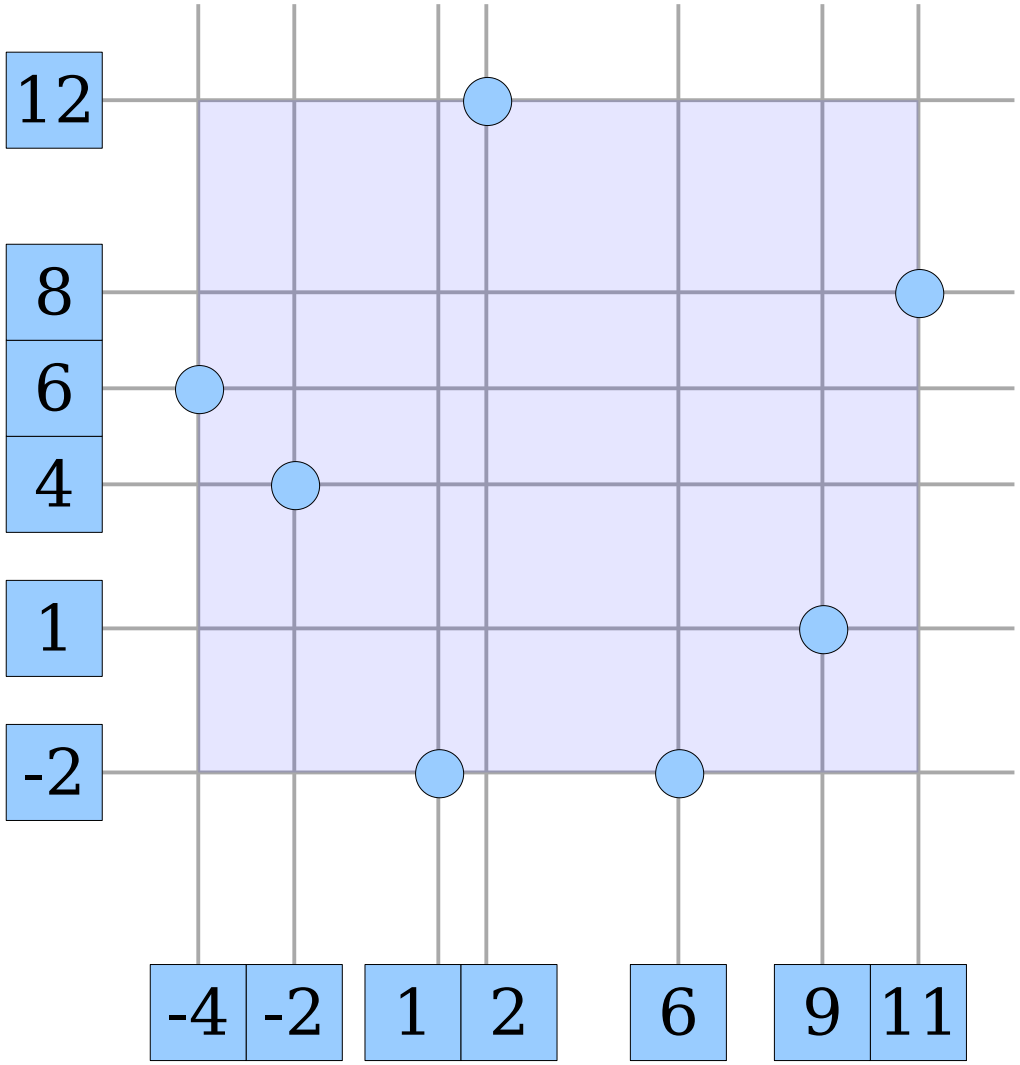
Another Naive Solution

- ***Recall:*** A lookup works by doing four binary searches on these arrays to snap to a grid, then returns the precomputed answer.
- ***Question:*** What is the cost of this approach?
- Each binary search takes time O(log *n*).
- However, the time required to return the answer varies based on how many points we return.



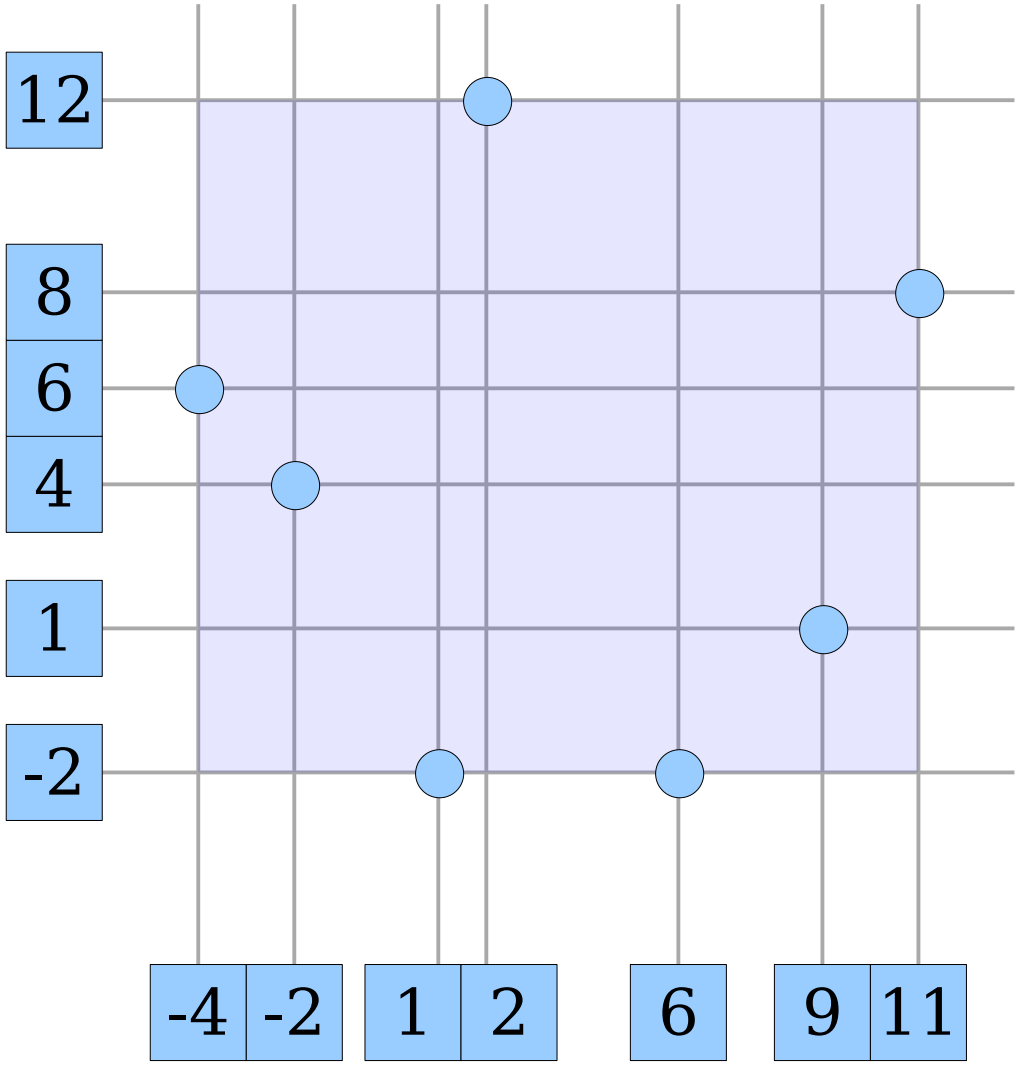
Another Naive Solution

- ***Recall:*** A lookup works by doing four binary searches on these arrays to snap to a grid, then returns the precomputed answer.
- ***Question:*** What is the cost of this approach?
- Each binary search takes time O(log *n*).
- However, the time required to return the answer varies based on how many points we return.



Another Naive Solution

- Range searches are ***output-sensitive algorithms***; they take longer to run based on the number of matches they report.
- Let ***k*** denote the number of matches returned by our algorithm.
- The cost of the lookup is then **O(log *n* + *k*)**.



Two Extremes

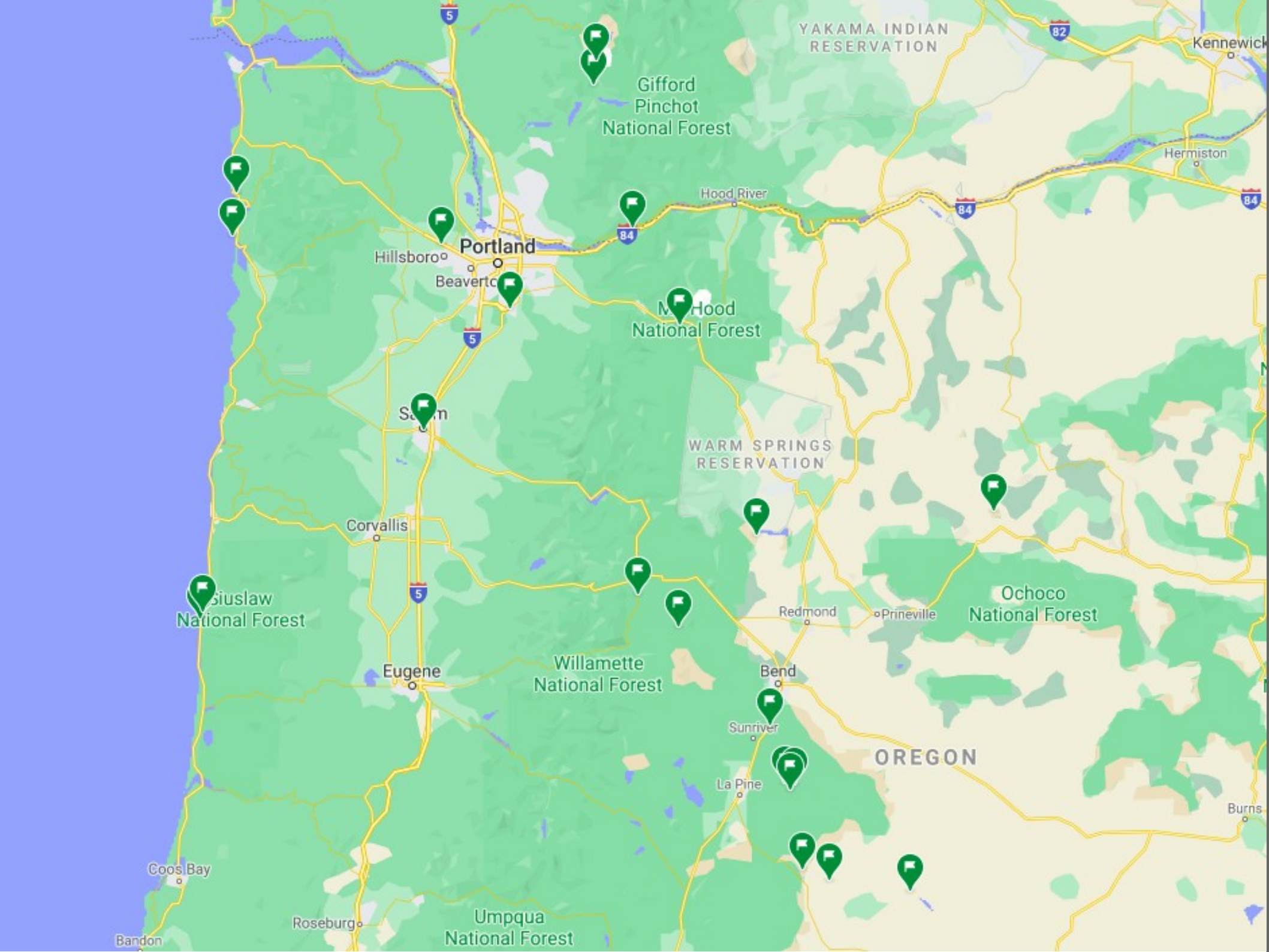
- Here's the scorecard so far.
- You know the drill from RMQ:

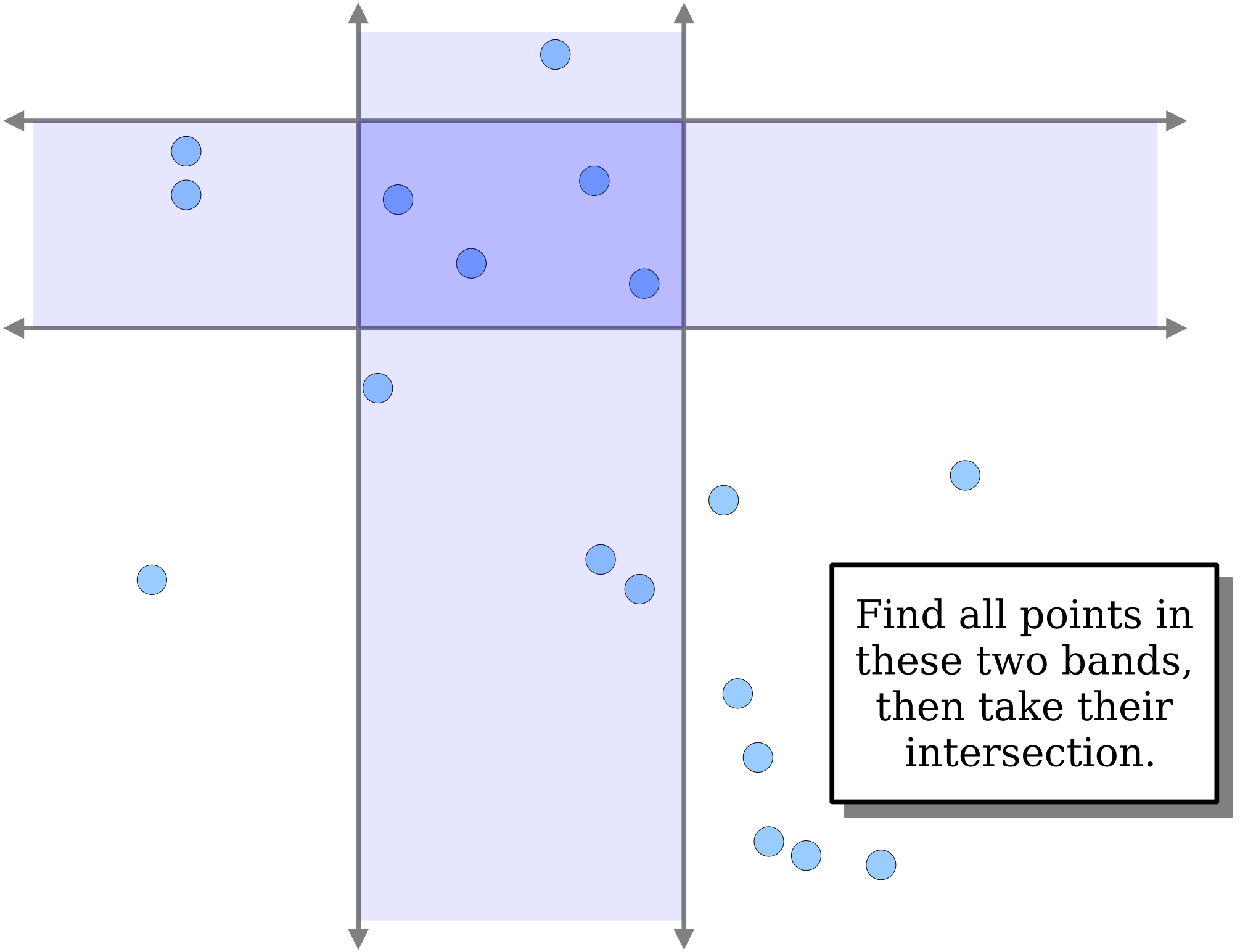
Is there a golden mean between these extremes?

- ***Goal:*** See how to make substantial reductions in preprocessing time and space usage.

	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$

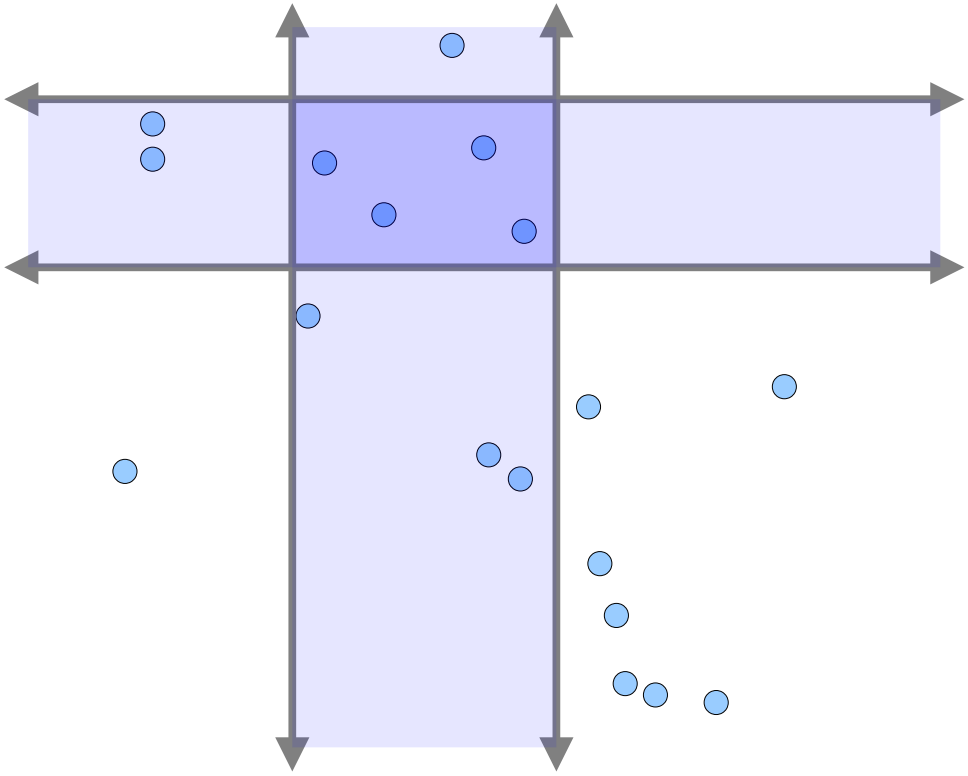
An Initial Approach: ***Band Searching***


YAKAMA INDIAN RESERVATION
Kennewick
Gifford Pinchot National Forest
Hermiston
Hood River
Portland
Hillsboro
Beaverton
Mt Hood National Forest
WARM SPRINGS RESERVATION
Corvallis
Siuslaw National Forest
Ochoco National Forest
Redmond
Prineville
Eugene
Willamette National Forest
Bend
Sunriver
OREGON
La Pine
Burns
Coos Bay
Roseburg
Umpqua National Forest
Bandon


Find all points in these two bands, then take their intersection.

2D Band Searching

- Store each point in two arrays, one sorted by x coordinates and one sorted by y coordinates.
- Use a binary search to find the first point in each array that's purely within the x and y bands.
- Use linear scans in each array to find all the points within the the x and y bands.
- Return the intersection of the points found this way.

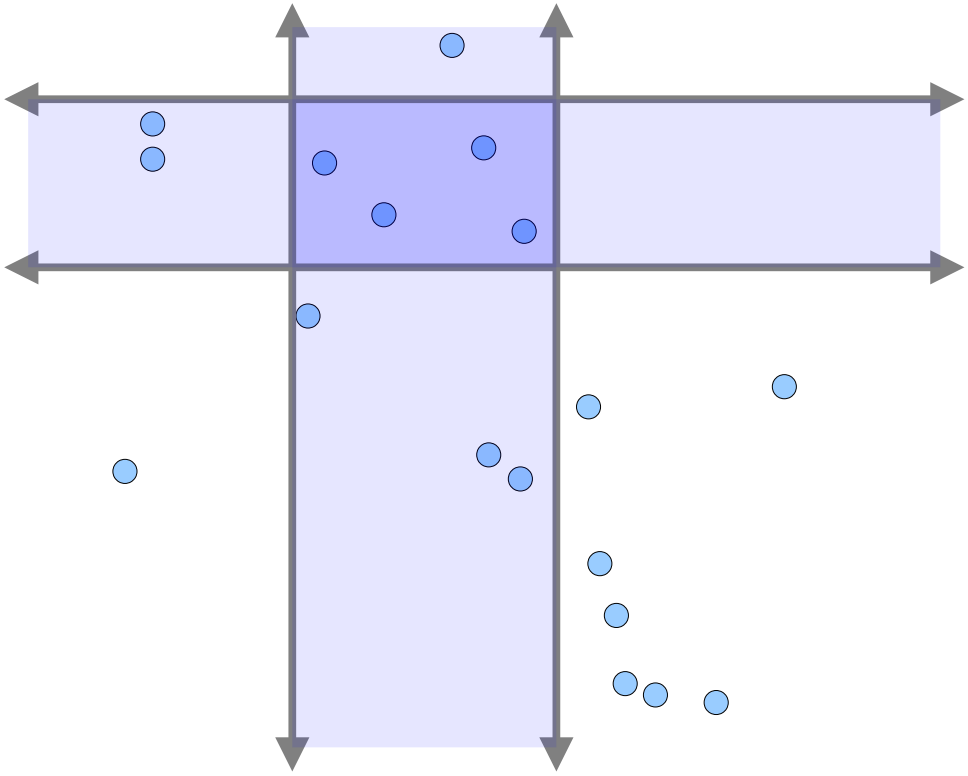


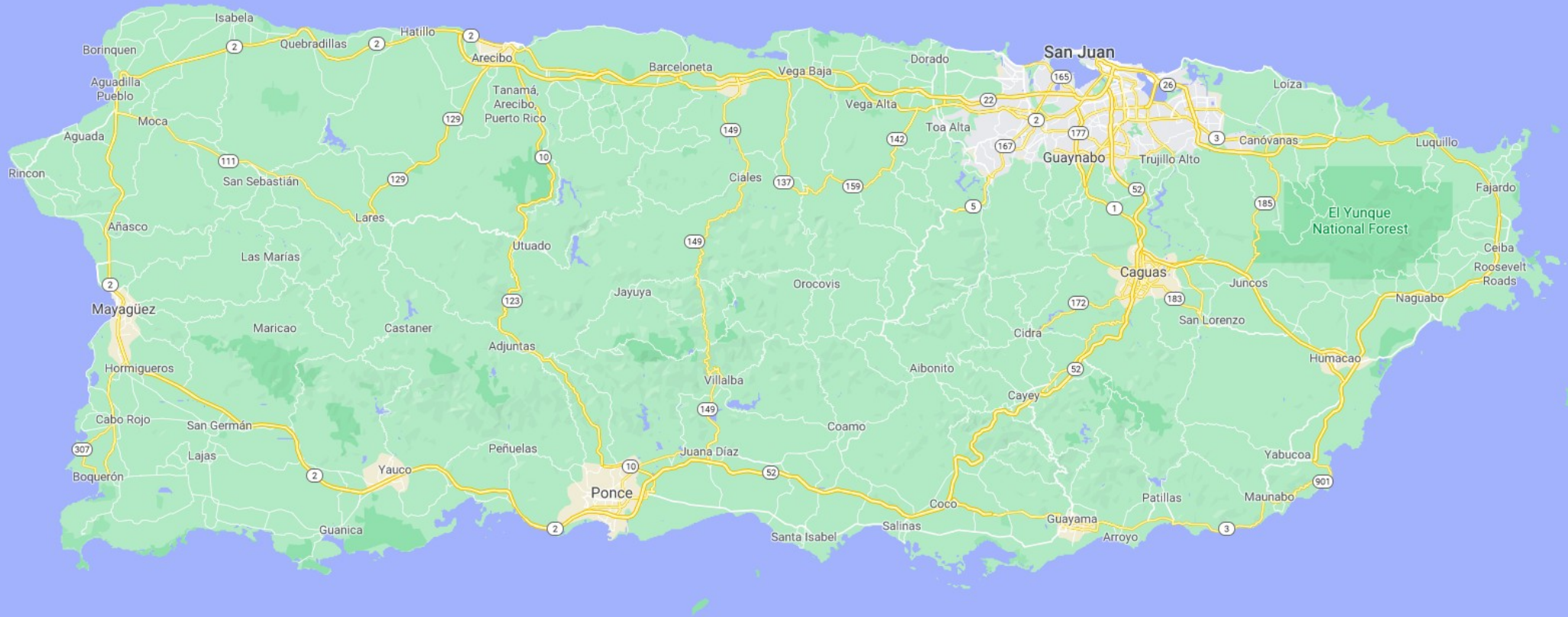
2D Band Searching

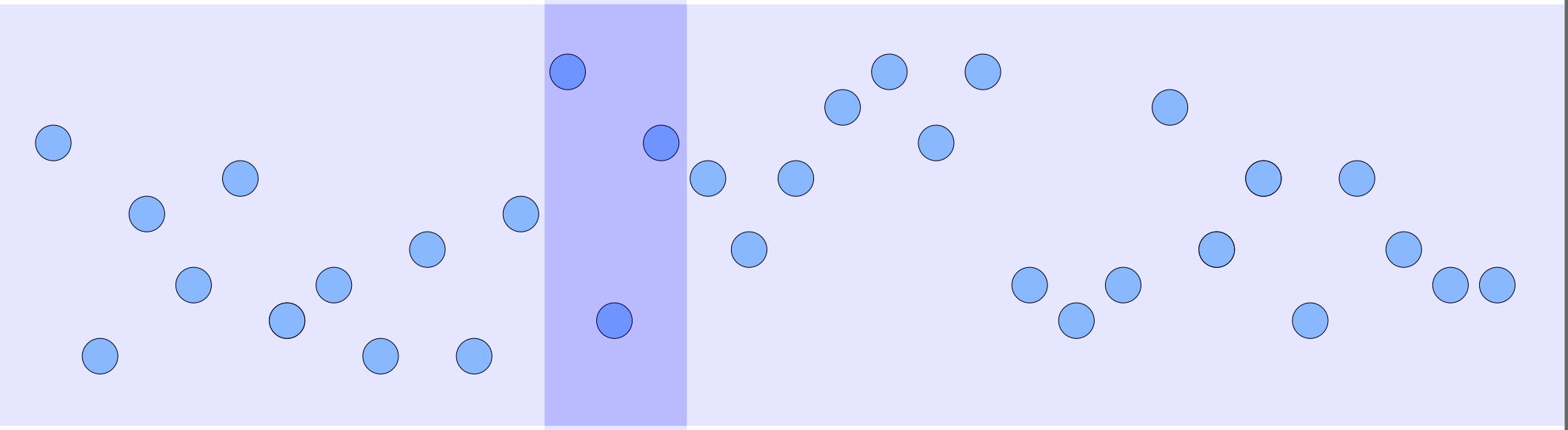
- How fast is this solution?
- Each binary search takes time O(log n).
- Suppose there are k_x points reported by the x-axis binary search and k_y points reported by the y-axis binary search.
- Total work done:

  O(log n + k_x + k_y).

- ***Question:*** Is that a good runtime?




Isabela
Borinquen
Quebradillas
Hatillo
Arecibo
San Juan
Dorado
Barceloneta
Vega Baja
Loíza
Aguadilla
Pueblo
Tanamá,
Arecibo,
Puerto Rico
Vega Alta
Moca
Toa Alta
Aguada
Canóvanas
Luquillo
Guaynabo
Trujillo Alto
Rincon
San Sebastián
Ciales
Fajardo
El Yunque
National Forest
Añasco
Lares
Utuado
Ceiba
Las Marías
Caguas
Roosevelt
Roads
Orocovis
Juncos
Jayuya
Mayagüez
Naguabo
San Lorenzo
Maricao
Castaner
Cidra
Adjuntas
Humacao
Hormigueros
Aibonito
Villalba
Cayey
Cabo Rojo
San Germán
Coamo
Peñuelas
Juana Díaz
Yabucoa
Lajas
Boquerón
Yauco
Ponce
Patillas
Maunabo
Coco
Guayama
Guanica
Salinas
Arroyo
Santa Isabel

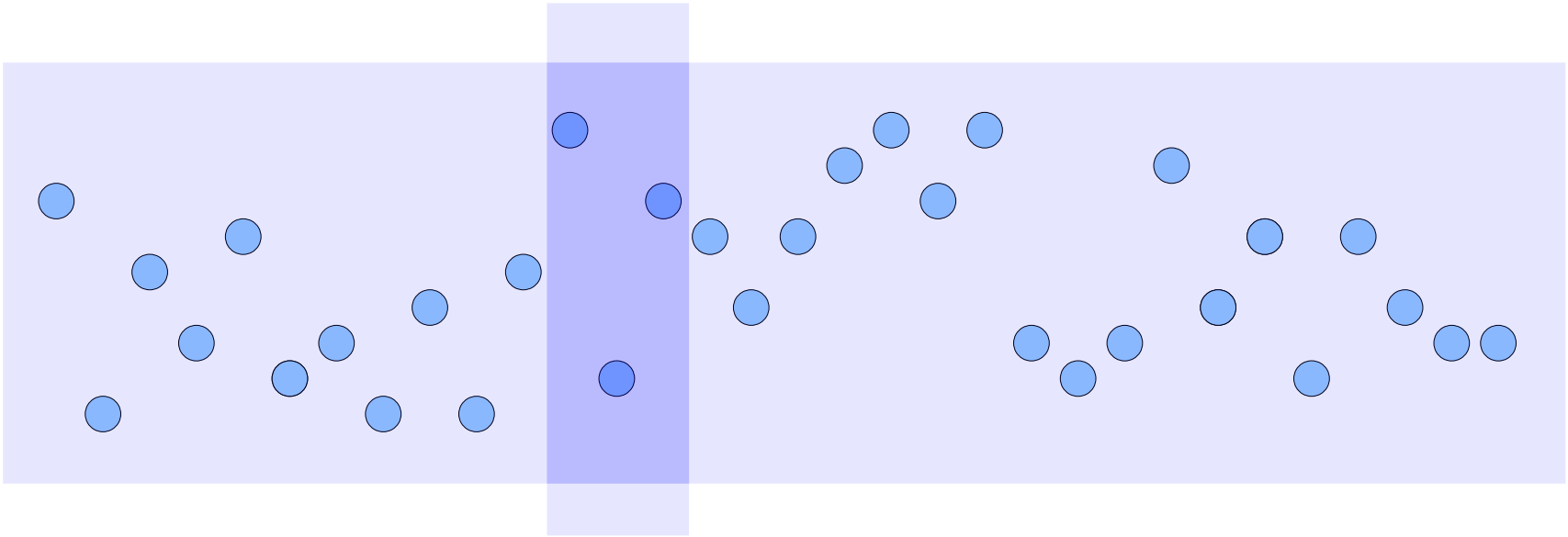


If our points are naturally squished along a narrow range, this approach can easily end up scanning over all the points.

If we're looking for worst-case efficiency, we're going to need another approach.

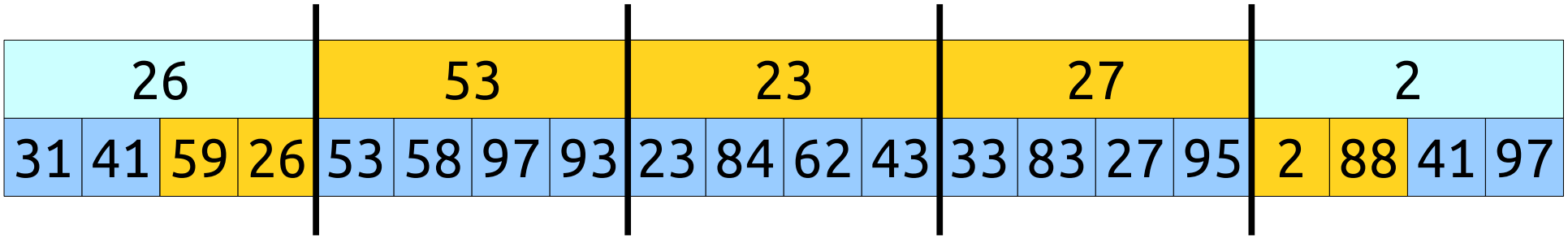
Why Is This Slow?

- Enumerating all the points in the correct *y* band is slow – many of them won't match.
- Testing all the points individually in the correct *y* band is slow; there are spatial correlations we aren't taking advantage of.
- ***Goal 1:*** Don't explicitly list all the points found in the *y* range. (Ideally, only list ones that match.)
- ***Goal 2:*** Don't explicitly test all the points found in the *y* range. (Ideally, use some kind of sorting to exclude many of them).



Goal 1: Don't explicitly list all the points found in the *y* range. (Ideally, only list ones that match.)

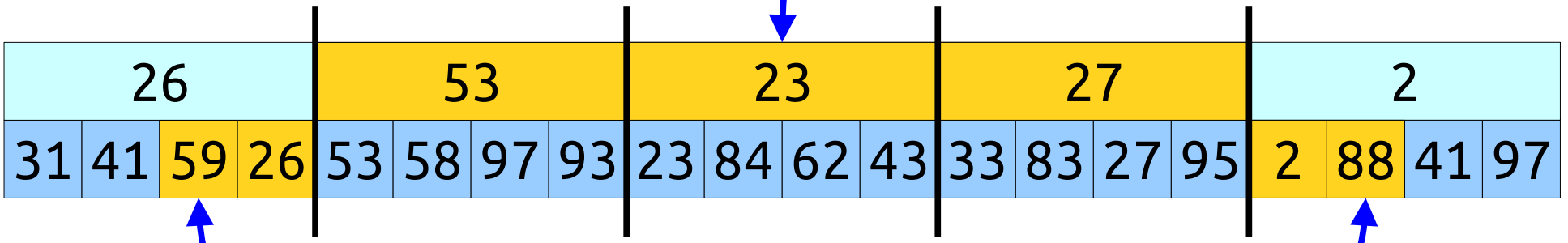
Goal 2: Don't explicitly test all the points found in the *y* range. (Ideally, use some kind of sorting to exclude many of them).



Goal 1: Don't explicitly list all the points found in the *y* range. (Ideally, only list ones that match.)

Goal 2: Don't explicitly test all the points found in the *y* range. (Ideally, use some kind of sorting to exclude many of them).

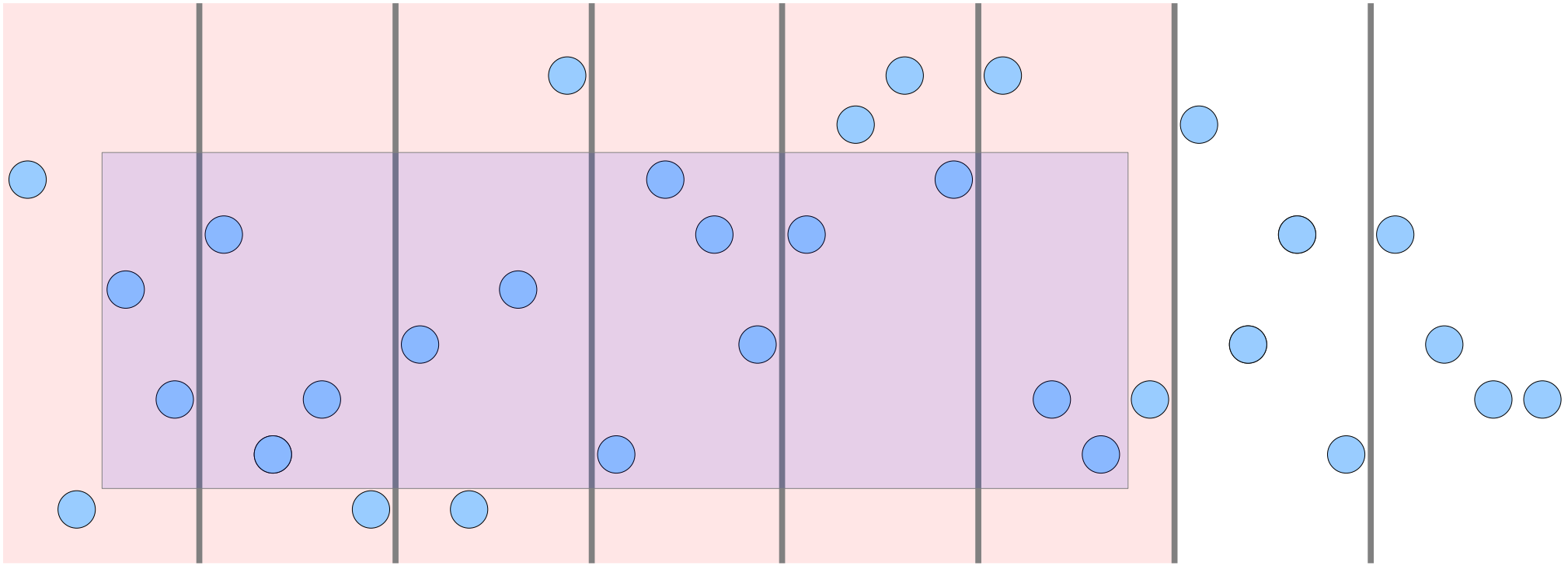
We don't need to list all items in blocks purely within the query range.



Items in boundary blocks need to be tested individually, but there aren't many of them.

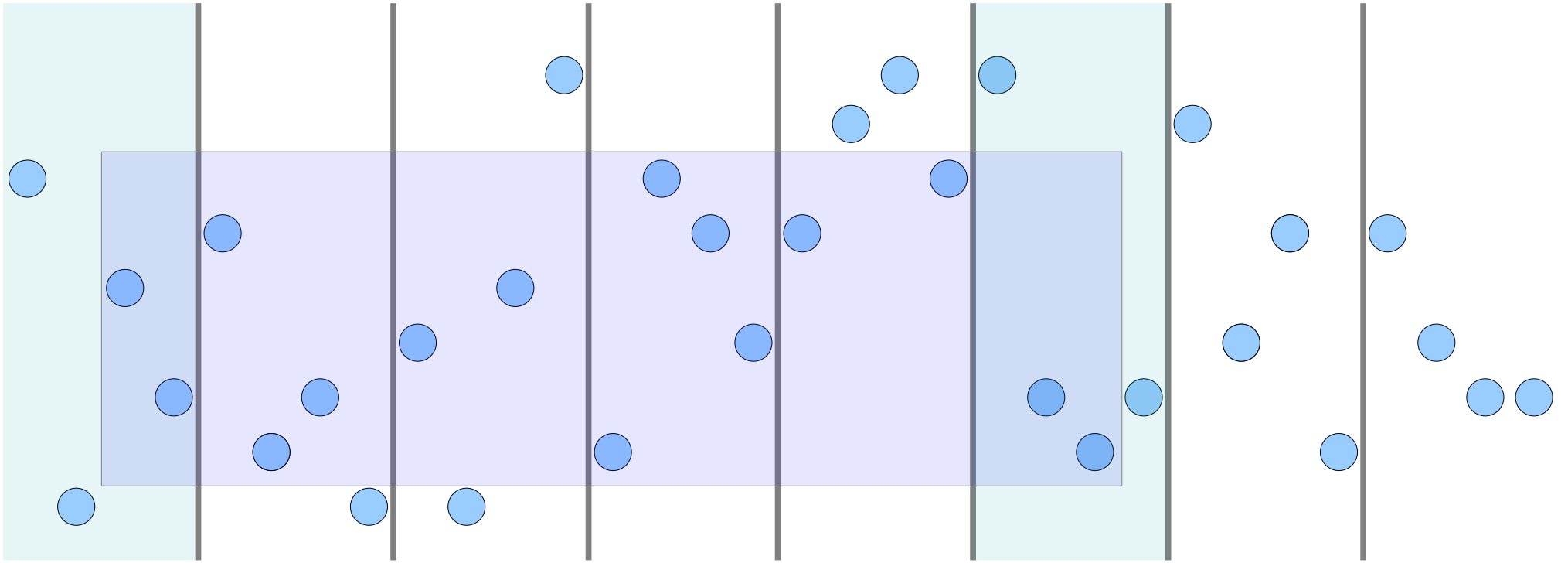
Split the points into blocks of size b by their x coordinates.

Binary search in the x coordinate to find which blocks overlap.



Split the points into blocks of size b by their x coordinates.

Binary search in the x coordinate to find which blocks overlap.

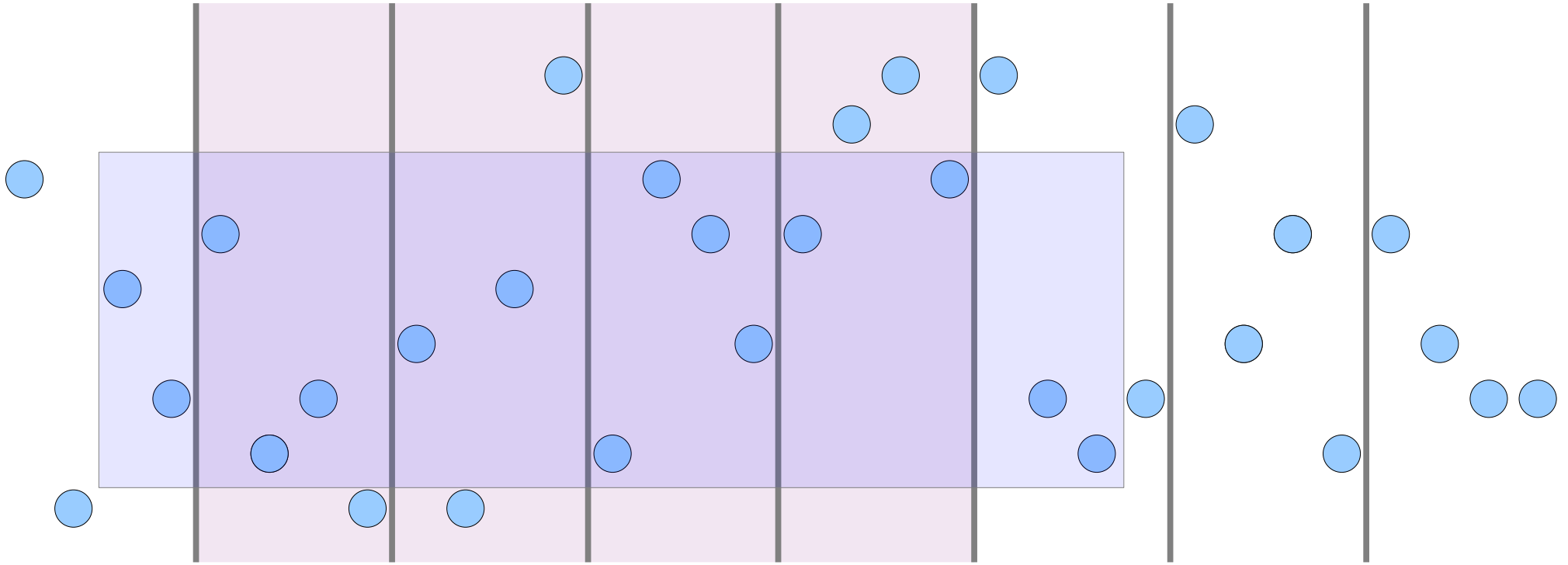


There are at most two partially-covered blocks. Points here may or may not be in the range.

Use a linear scan over these blocks to figure out which points are in the range.

Split the points into blocks of size b by their x coordinates.

Binary search in the x coordinate to find which blocks overlap.



All points in these blocks have x coordinates in range. We just need to check whether their y coordinates are valid.

Store the points within each block sorted by y coordinates. Use binary search to find all points in the correct y range.

Blocking Revisited

- Pick a block size b.
- Split the points into O(n / b) blocks of size b, grouped by x coordinates.
- Sort the points within each block by their y coordinates.
- To do a query:
  - Binary search in the x direction to find all blocks that overlap the query range.
  - Linear search over each partially-covered block.
  - Binary search over each fully-covered block.

Blocking Revisited

- Preprocessing step:
  - Sort points by their x coordinates.
  - Split the points into $O(n / b)$ blocks of size b.
  - Sort all the points in each block by their y coordinates.
- Total cost:

$$O(n \log n + (n / b)\, b \log b)$$

$$= \mathbf{O(n \log n)}.$$

Blocking Revisited

- Memory Usage:
  - Maintain a list of x coordinates delimiting where the $O(n / b)$ blocks start and end.
  - Each point is stored in exactly one block.
- Total memory usage: **$O(n)$**.

Blocking Revisited

- Query step:
  - Do a binary search in the x direction to see which blocks to check.
  - For each boundary block, do a linear scan within that block to find all points in the query box.
  - Within each internal block, do a binary search to find all points within the y range.
- Total time spent:

$$\begin{aligned} &O(\log (n / b) + b + (n / b) \log b + k) \\ &= O(\log n + b + (n / b) \log n + k) \\ &= \mathbf{O(b + (n / b) \log n + k)}. \end{aligned}$$

Picking the Block Size

- What choice of b minimizes this quantity?

$$\mathbf{O(}\boldsymbol{b} \mathbf{+ (}\boldsymbol{n} \mathbf{/} \boldsymbol{b}\mathbf{)\ log}\ \boldsymbol{n} \mathbf{+}\ \boldsymbol{k}\mathbf{)}$$

Formulate a hypothesis, but ***don't post anything in chat just yet***.

Picking the Block Size

- What choice of b minimizes this quantity?

$$\mathbf{O(}\boldsymbol{b} \mathbf{+ (}\boldsymbol{n} \mathbf{/} \boldsymbol{b}\mathbf{)\ log}\ \boldsymbol{n} \mathbf{+}\ \boldsymbol{k}\mathbf{)}$$

Now, ***private chat me your best guess***. Not sure? Just answer "??".

Picking the Block Size

- What choice of b minimizes this quantity?

$$\mathbf{O(}\boldsymbol{b} \mathbf{+ (}\boldsymbol{n} \mathbf{/} \boldsymbol{b}\mathbf{)\ log}\ \boldsymbol{n} \mathbf{+}\ \boldsymbol{k}\mathbf{)}$$

Picking the Block Size

- What choice of b minimizes this quantity?

$$\mathbf{O(}\boldsymbol{b}\mathbf{ + (}\boldsymbol{n}\mathbf{ / }\boldsymbol{b}\mathbf{) log }\boldsymbol{n}\mathbf{ + }\boldsymbol{k}\mathbf{)}$$

- ***Answer:*** pick $b = \sqrt{n \log n}$, giving an overall runtime of

$$\mathbf{O}\left(\sqrt{\boldsymbol{n}\mathbf{ log }\boldsymbol{n}} + \boldsymbol{k}\right)$$

- Whoa! I've never seen a runtime like that one before.

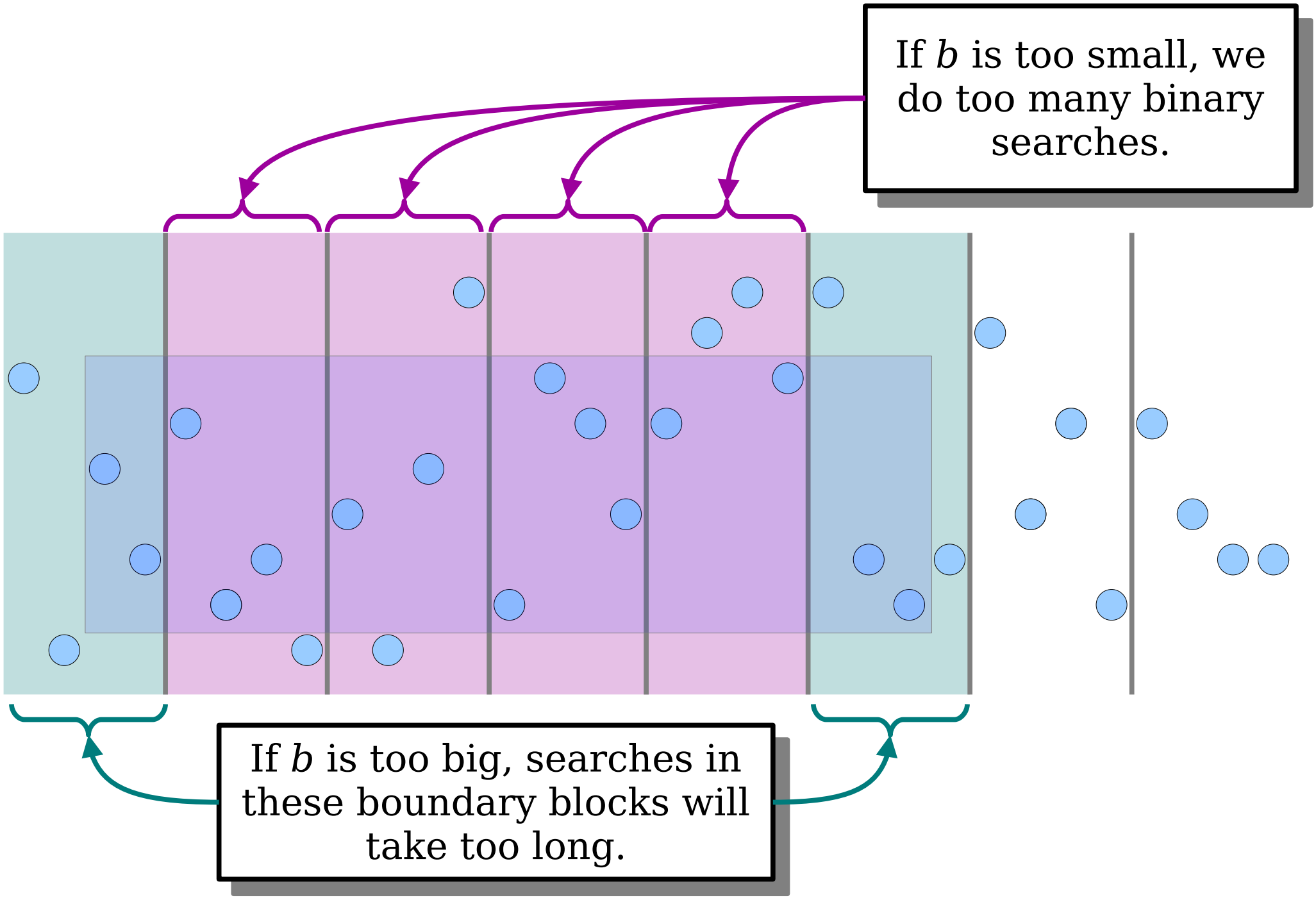
The Story So Far

- This new structure is a marked space improvement over precompute-all.
- It's also much better than the linear scan for pretty much all range sizes.
- ***Question:*** Can we do better?

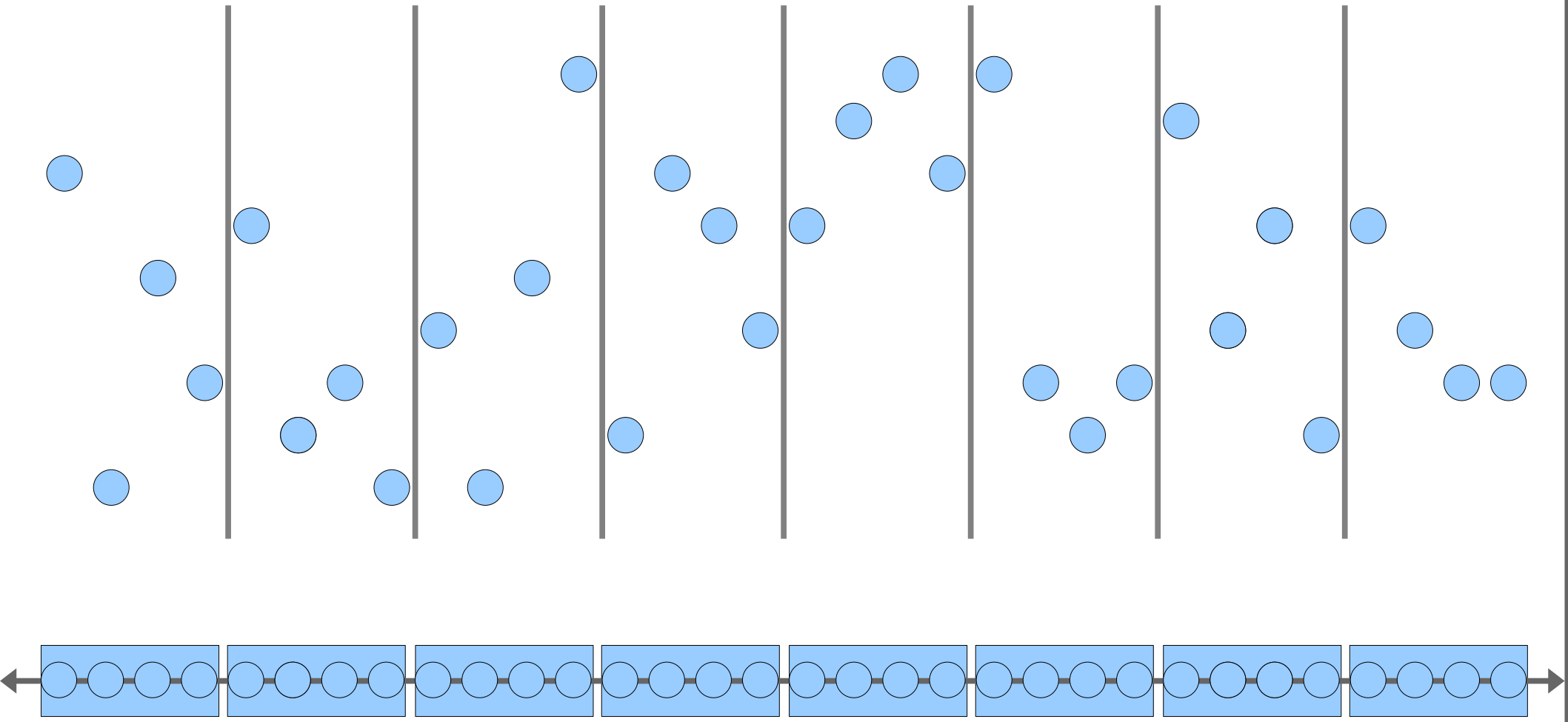
	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$

Speeding Things Up

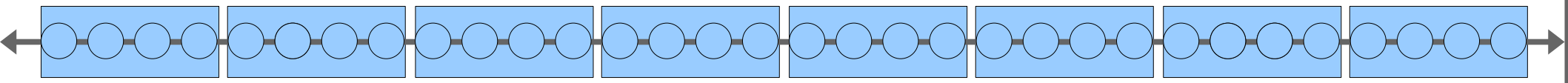
$$O(b + (n / b) \log n + k)$$


If b is too small, we do too many binary searches.
If b is too big, searches in these boundary blocks will take too long.

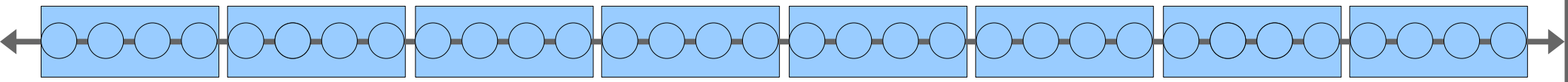
When you see these blocks, think "2D points, sorted by *y* coordinates."



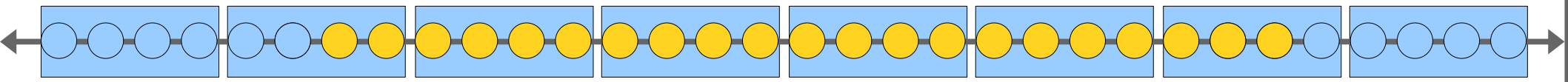
When you see these blocks, think “2D points, sorted by *y* coordinates.”



When you see these blocks, think "2D points, sorted by *y* coordinates."

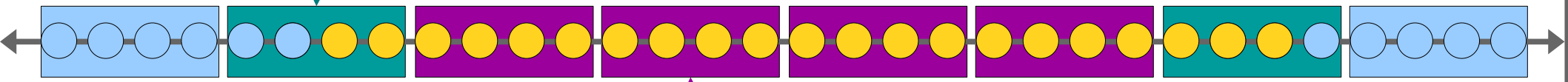


When you see these blocks, think “2D points, sorted by *y* coordinates.”



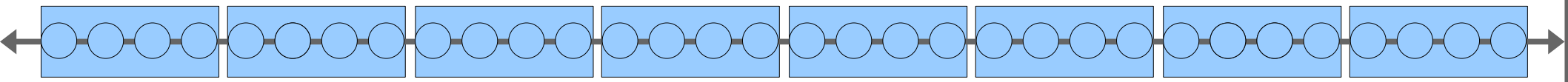
When you see these blocks, think "2D points, sorted by y coordinates."

Scan all points here; they may be out of both x and y range.



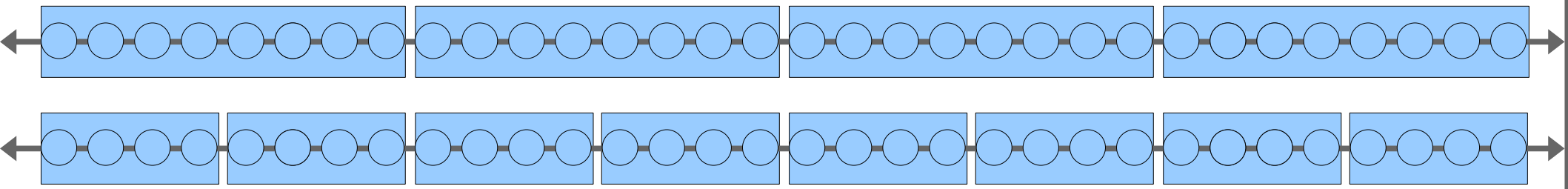
Binary search over y coordinates here; all points have valid x coordinates.

When you see these blocks, think "2D points, sorted by *y* coordinates."



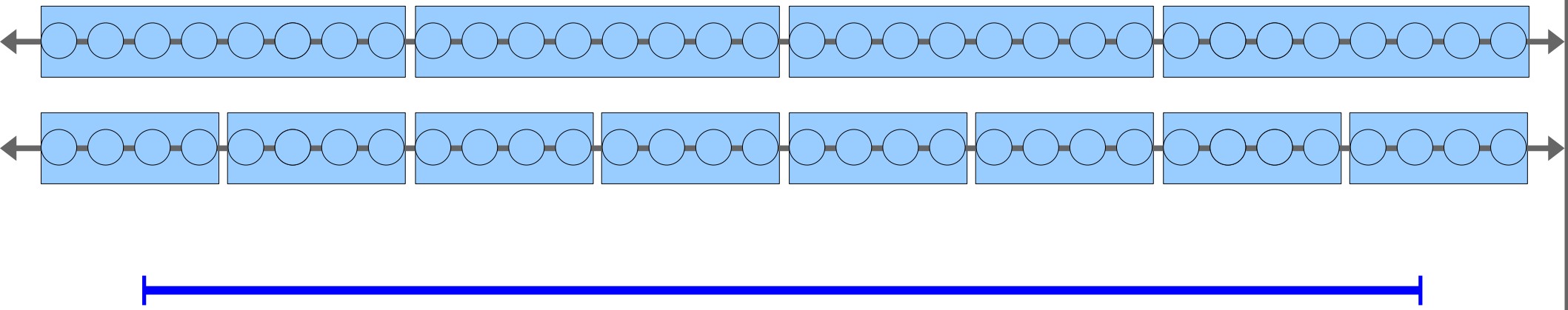
When you see these blocks, think "2D points, sorted by y coordinates."

Store blocks of two different sizes: b and $2b$.



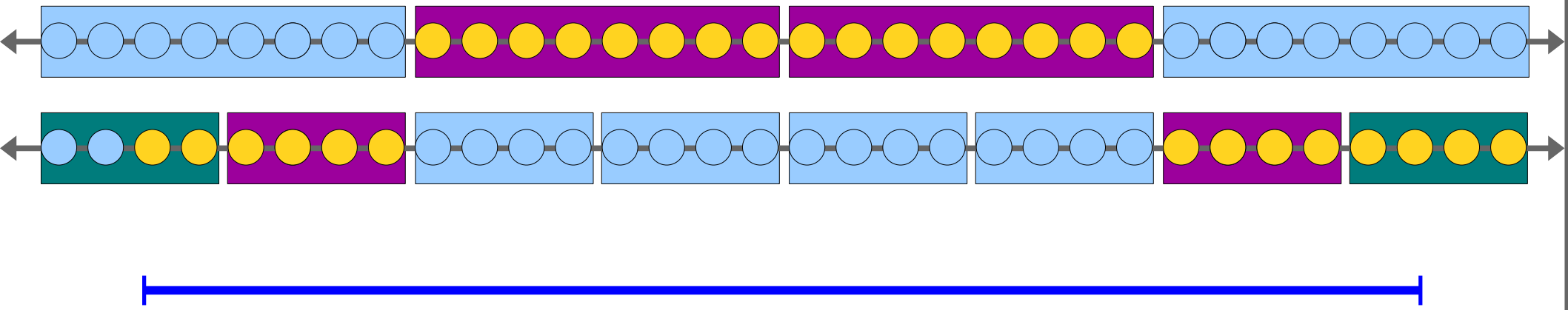
When you see these blocks, think "2D points, sorted by y coordinates."

Store blocks of two different sizes: b and $2b$.



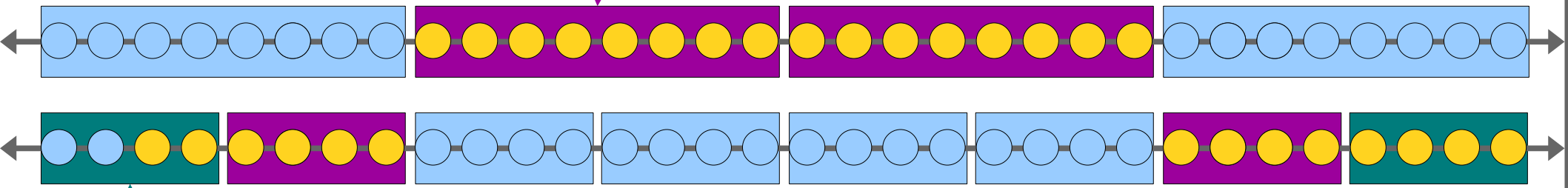
When you see these blocks, think "2D points, sorted by y coordinates."

Store blocks of two different sizes: b and $2b$.



Two binary searches on ranges of size $2b$.
Two binary searches on ranges of size b.
Two linear scans on ranges of size b.

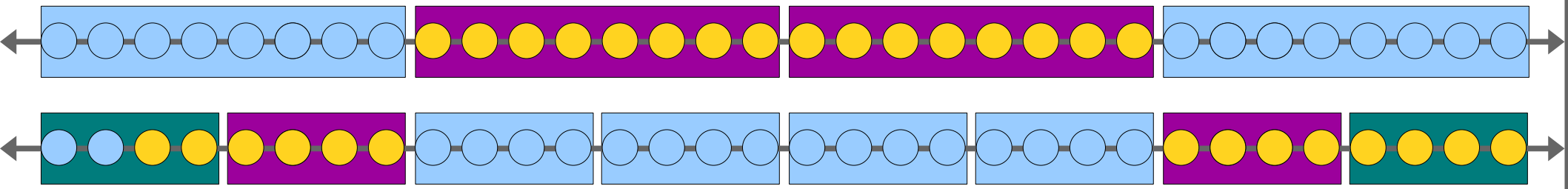
Binary search over y coordinates here; all points have valid x coordinates.

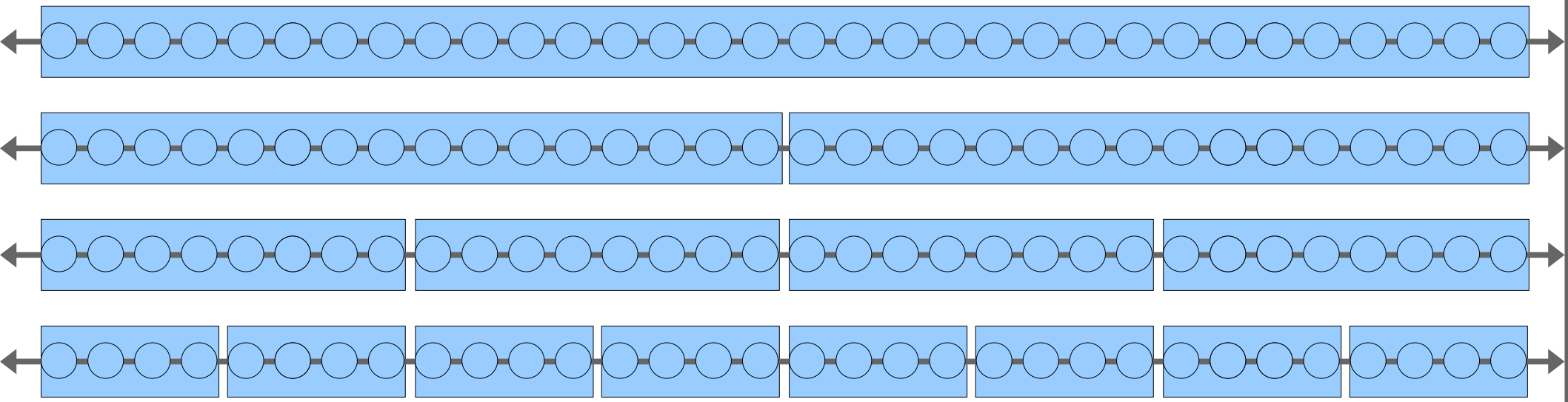


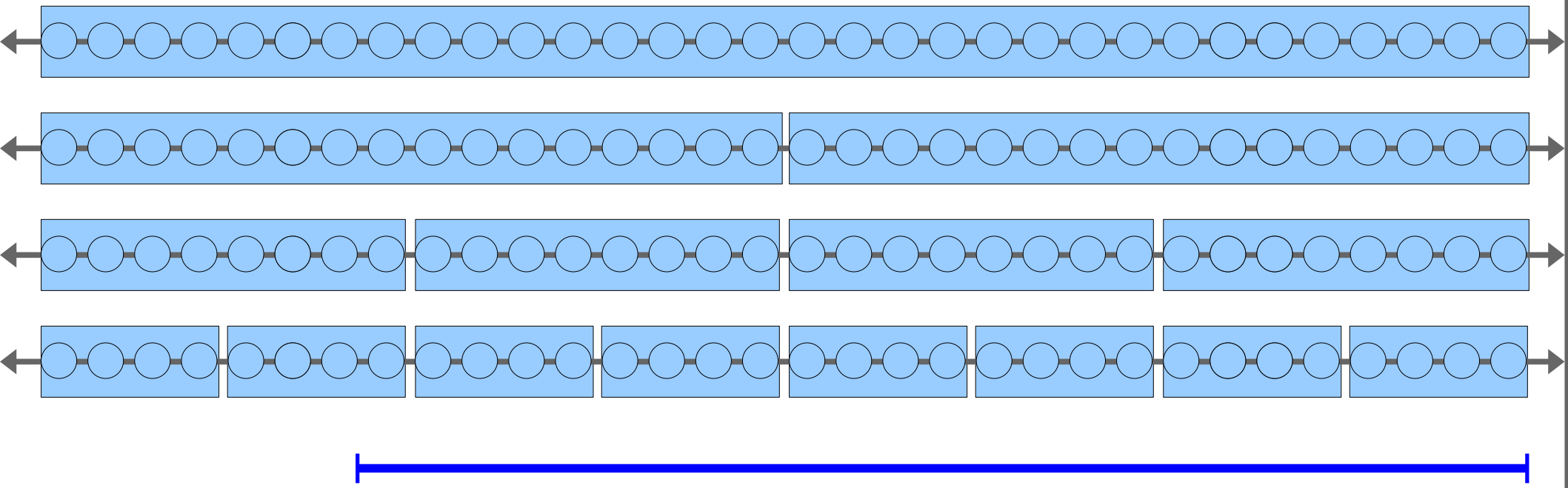
Scan all points here; they may be out of both x and y range.

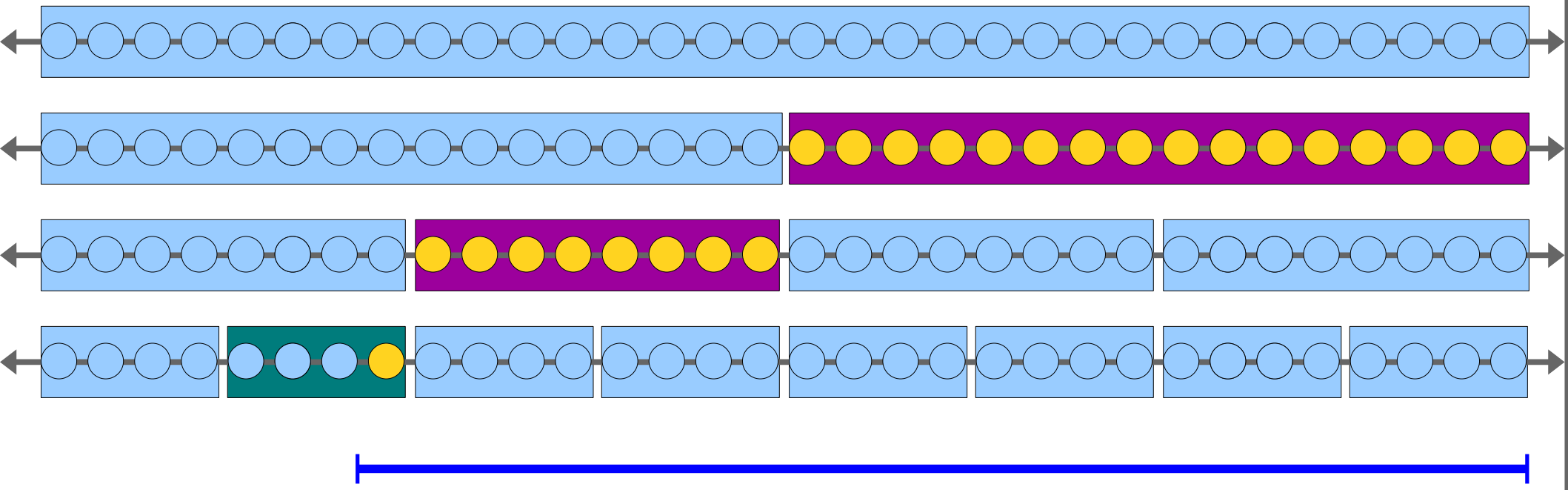
Two binary searches on ranges of size $2b$.
Two binary searches on ranges of size b.
Two linear scans on ranges of size b.

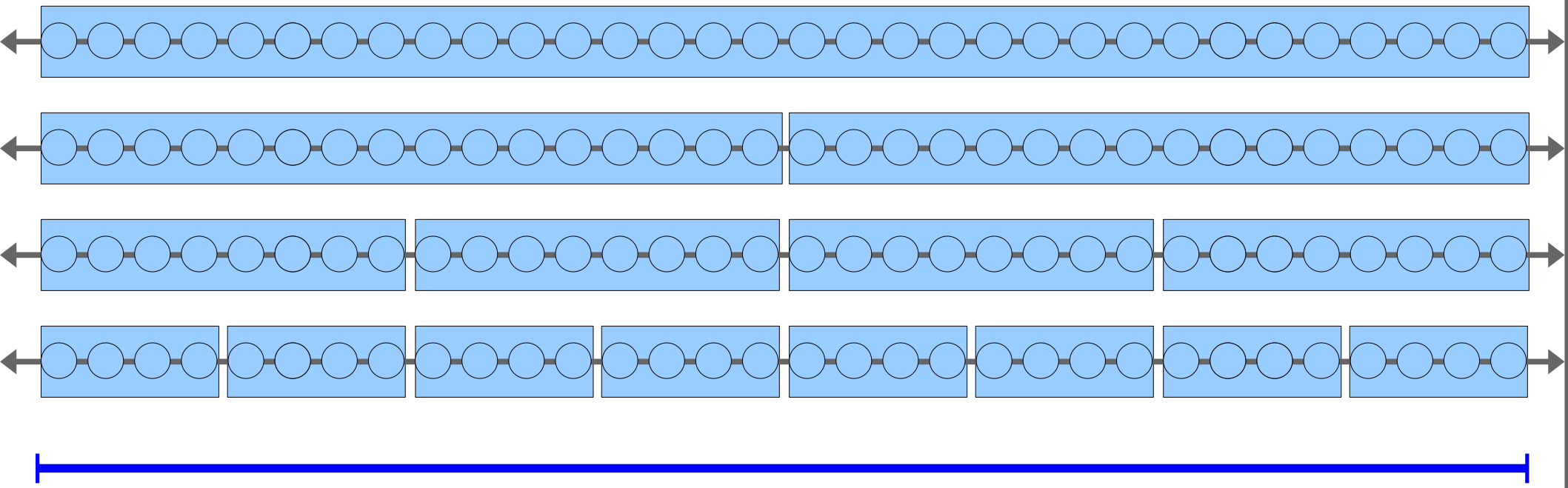
Key Idea: Adding layers with larger block sizes reduces the number of binary searches we need to do, speeding up the search.

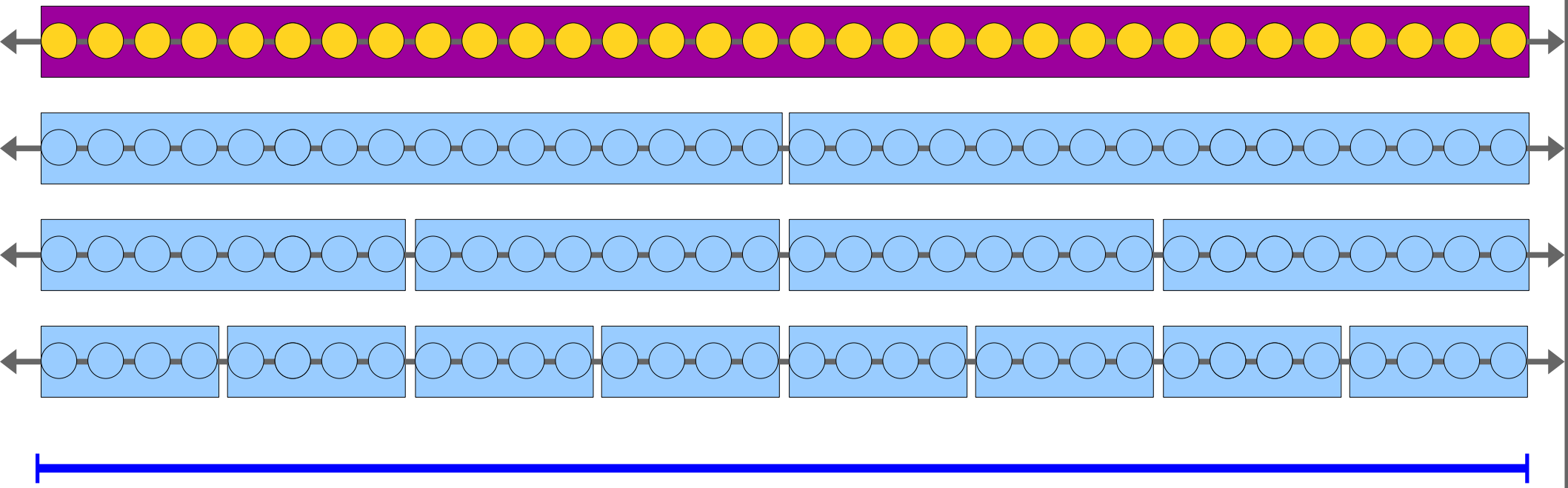


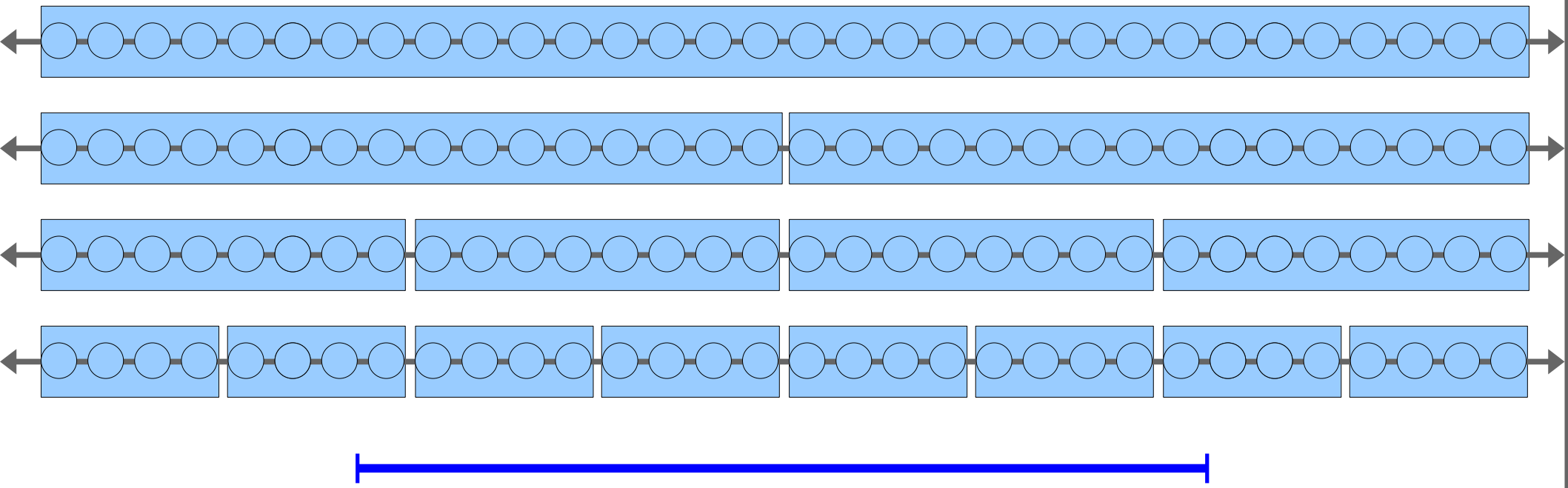


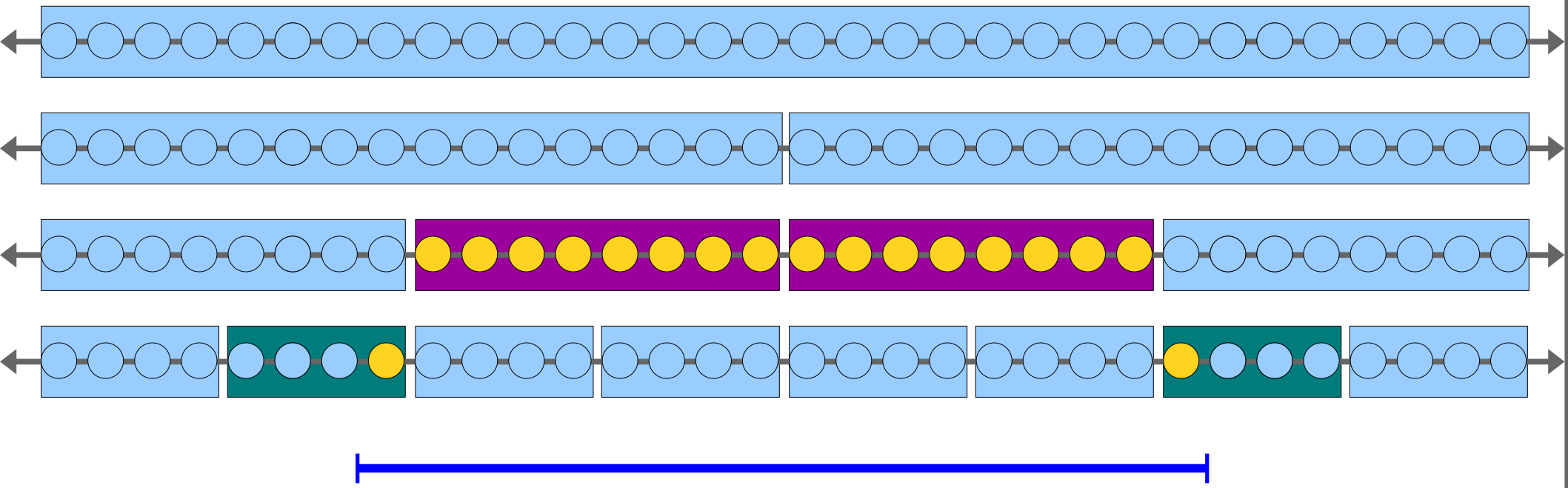


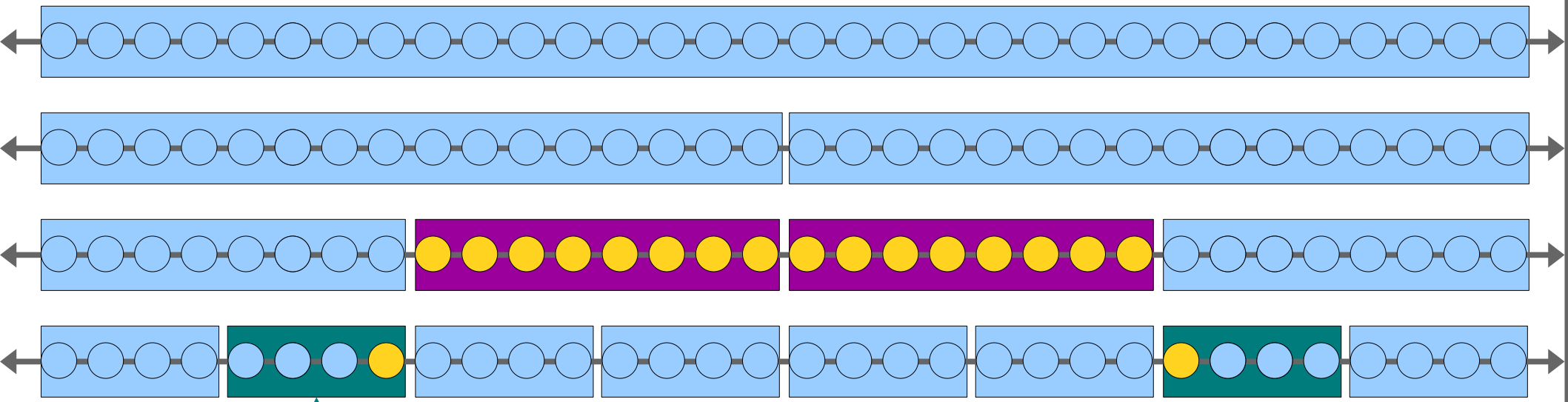




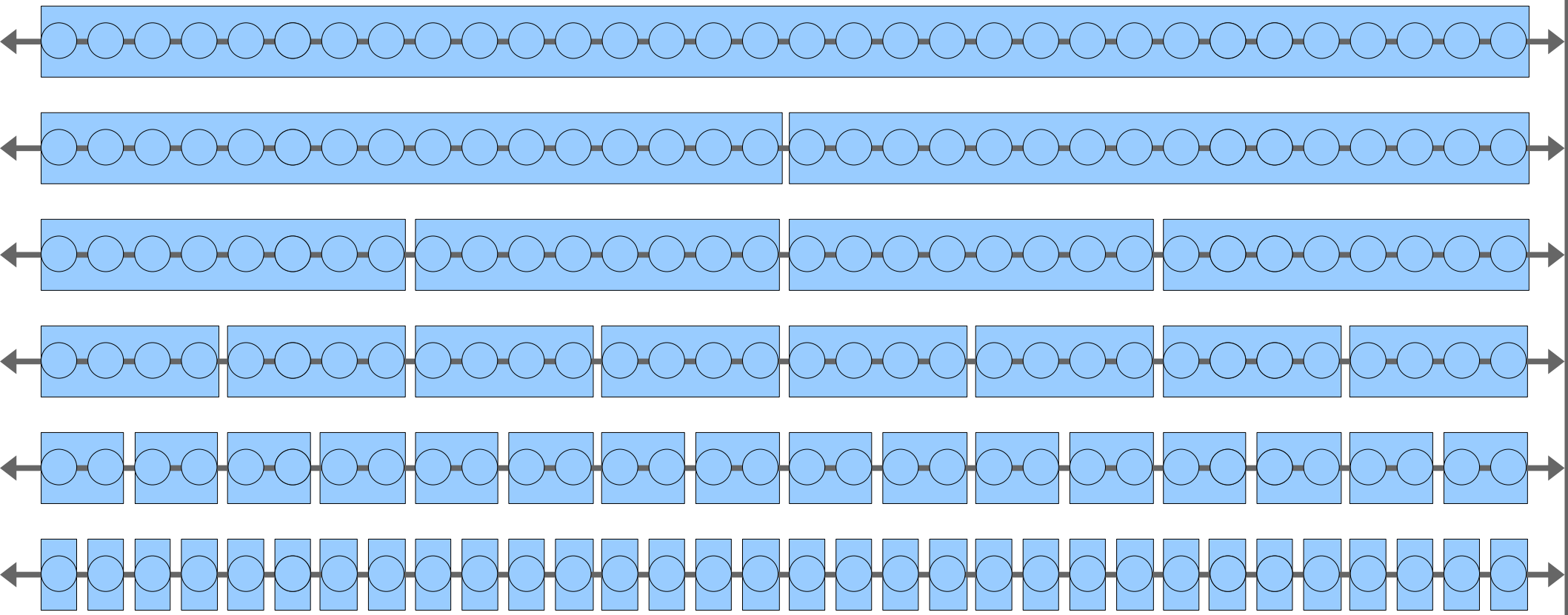


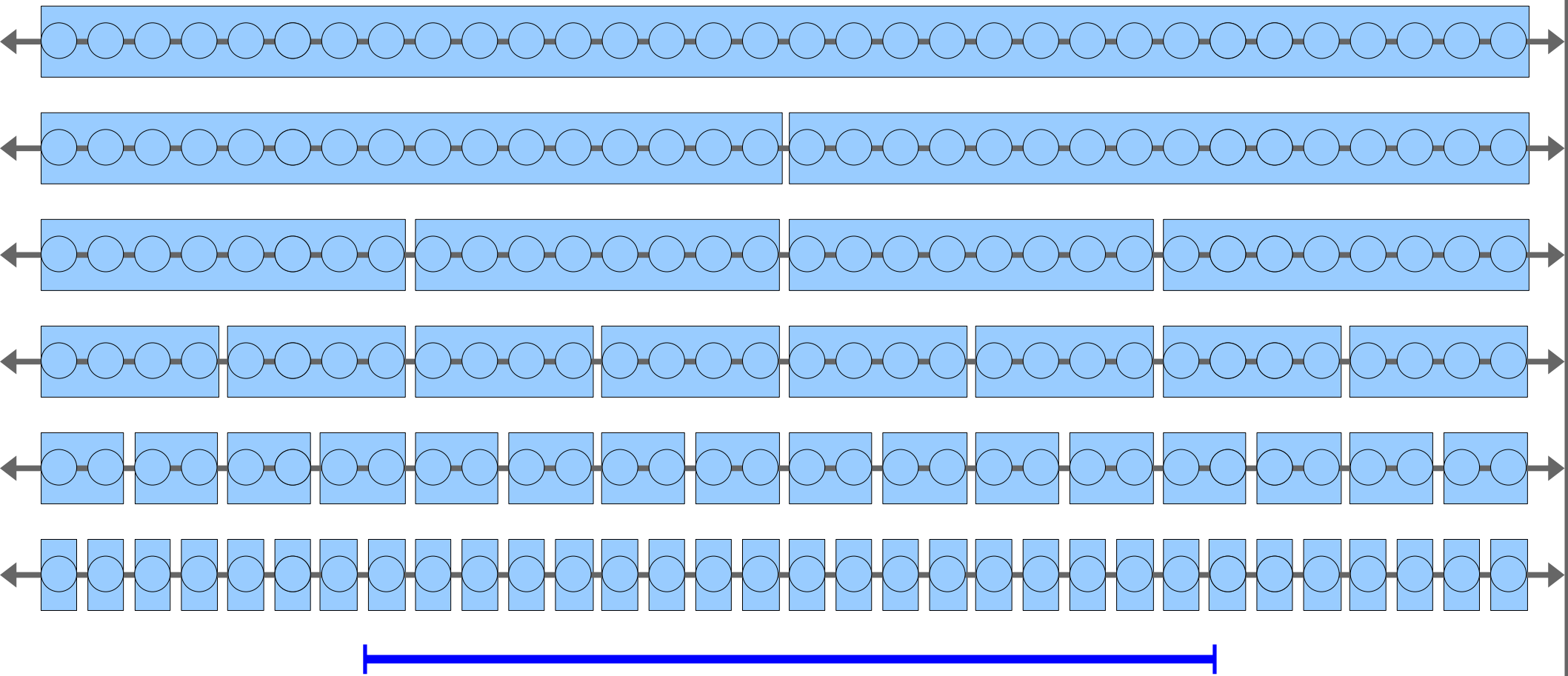


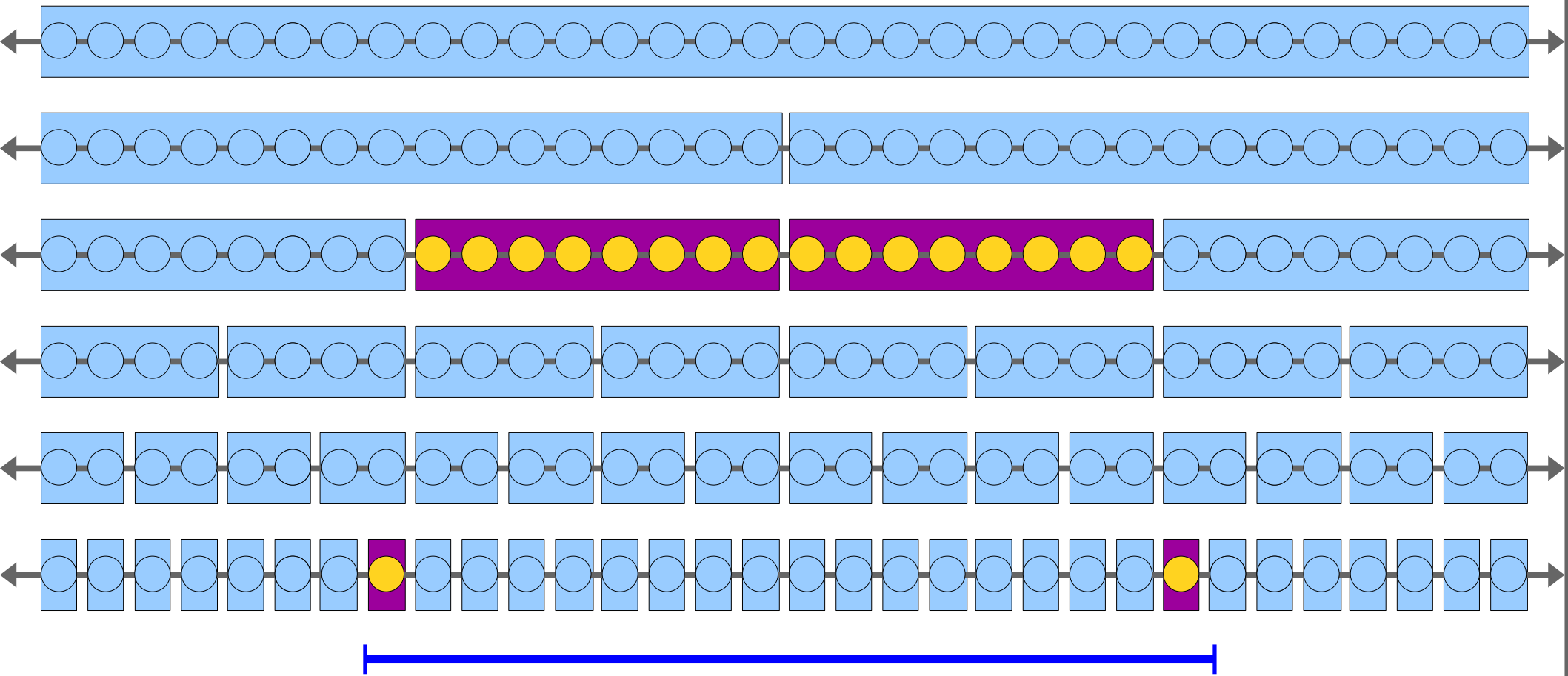


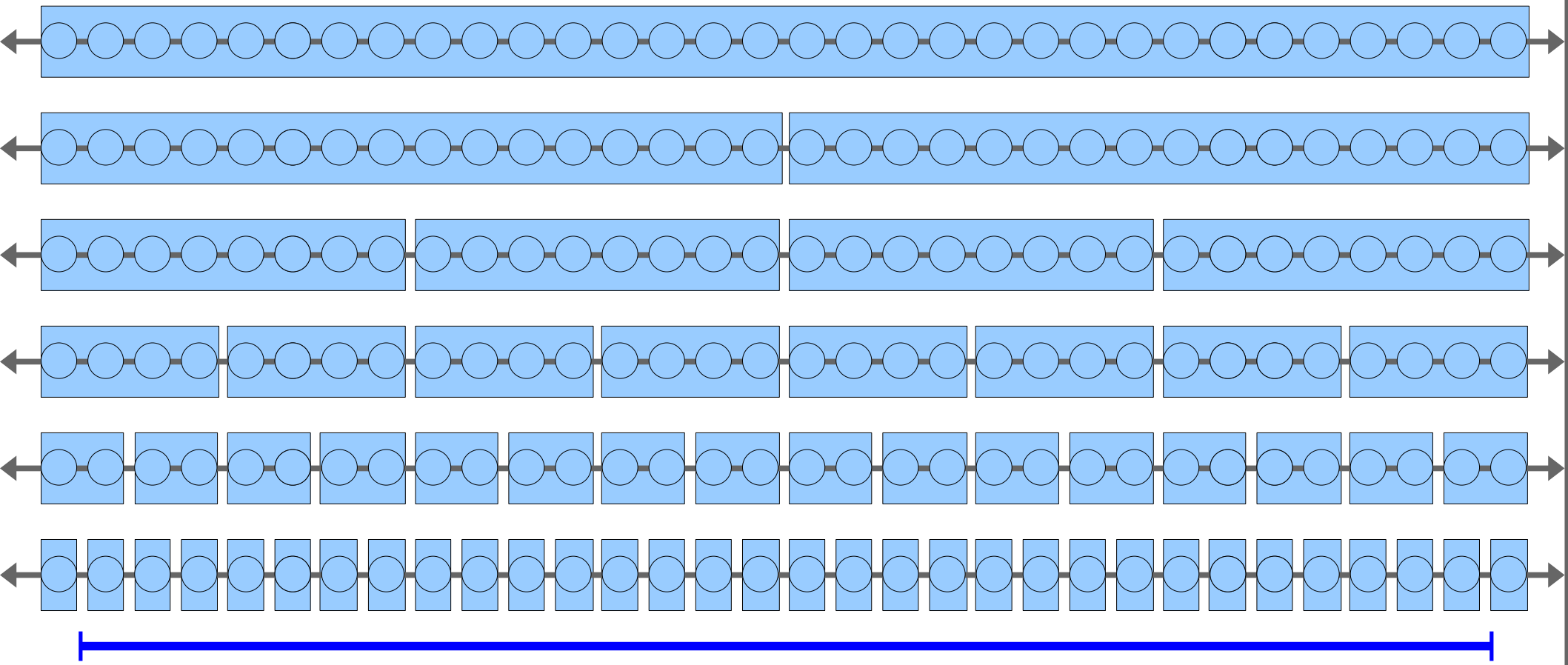


We still have to do a linear scan on all points in this block.

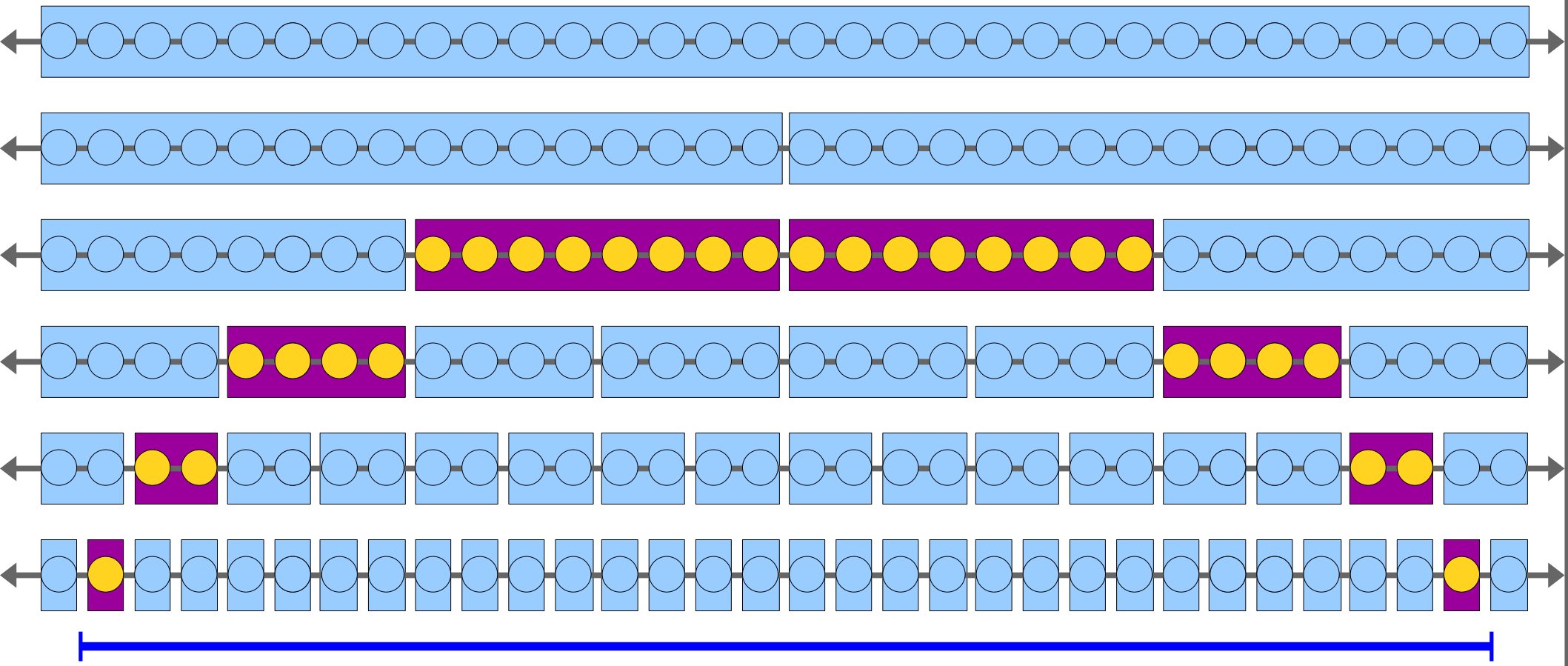








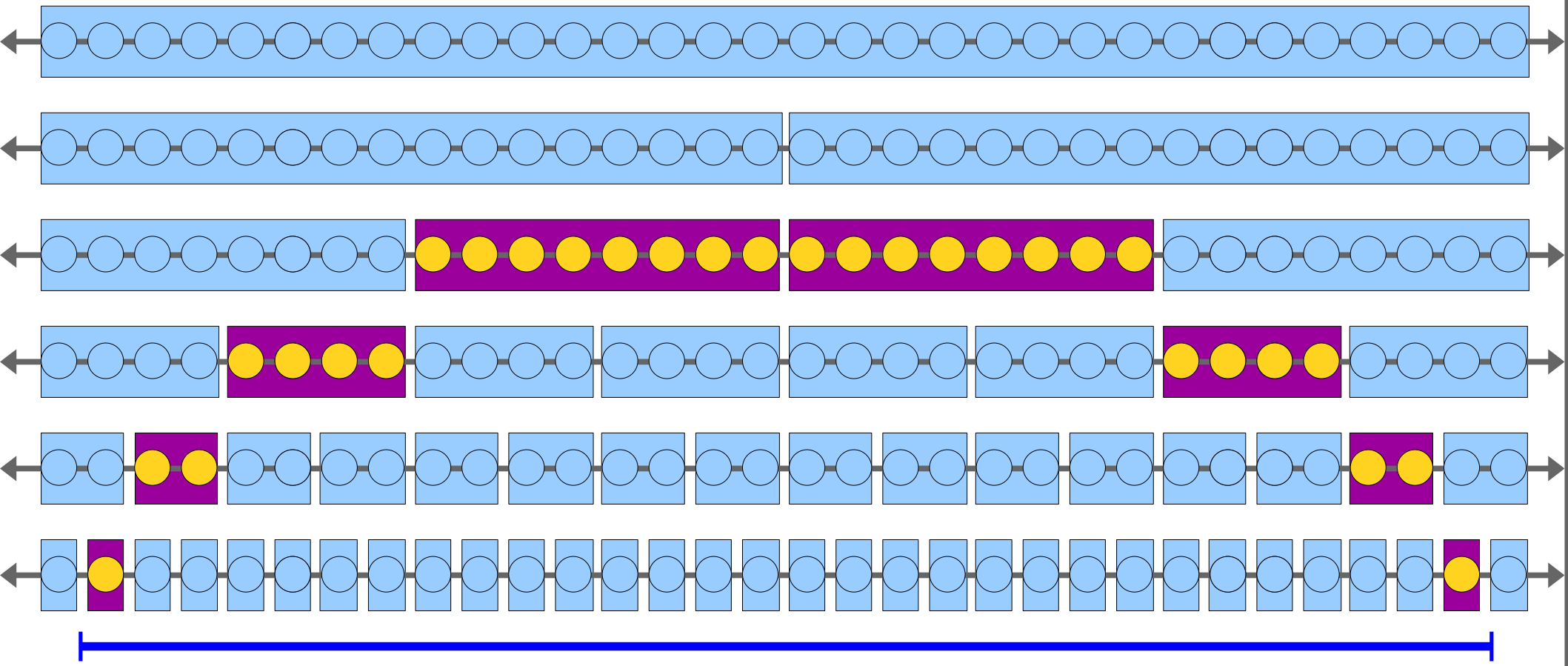
Every block used is fully contained in the range. We can use binary search in each of these blocks to find all points with matching y coordinates.



Claim: Every query range can be covered using only O(log n) total blocks.

Why?

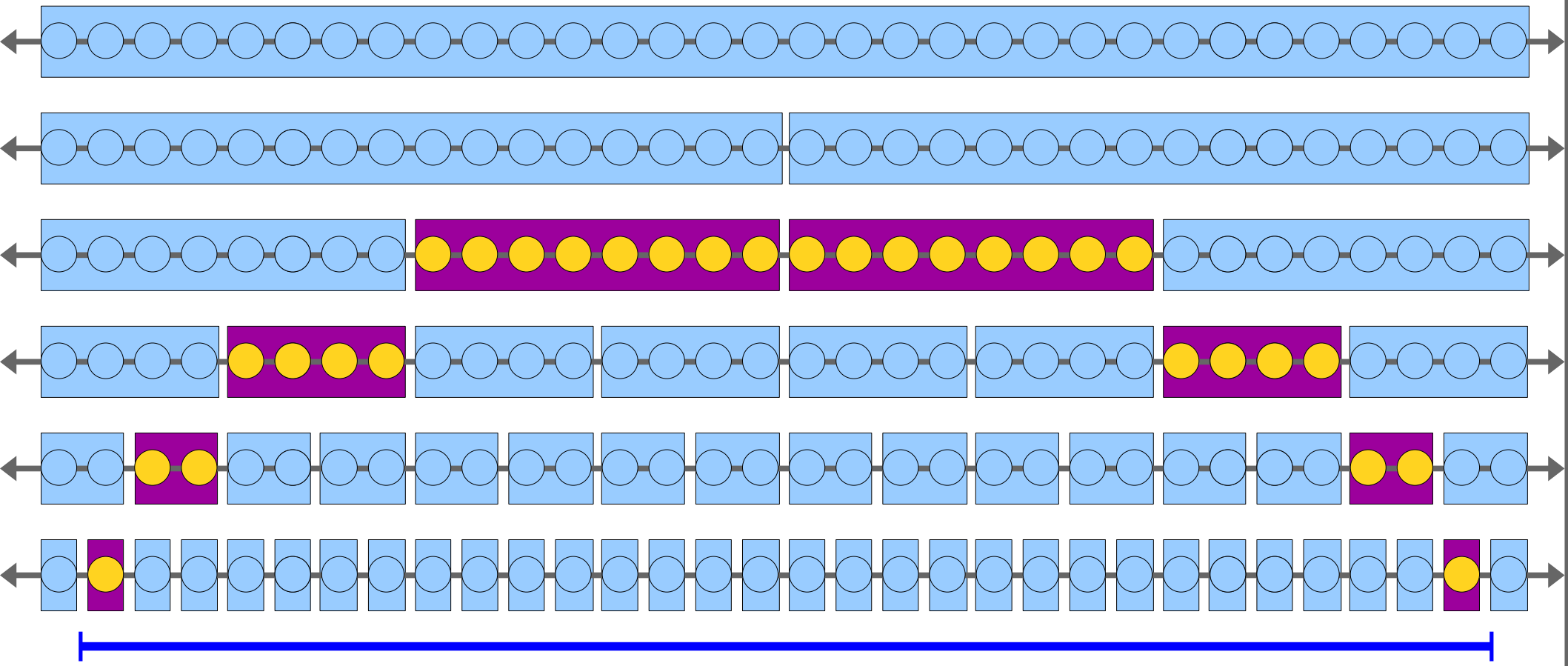
Formulate a hypothesis, but ***don't post anything in chat just yet***.



Claim: Every query range can be covered using only O(log n) total blocks.

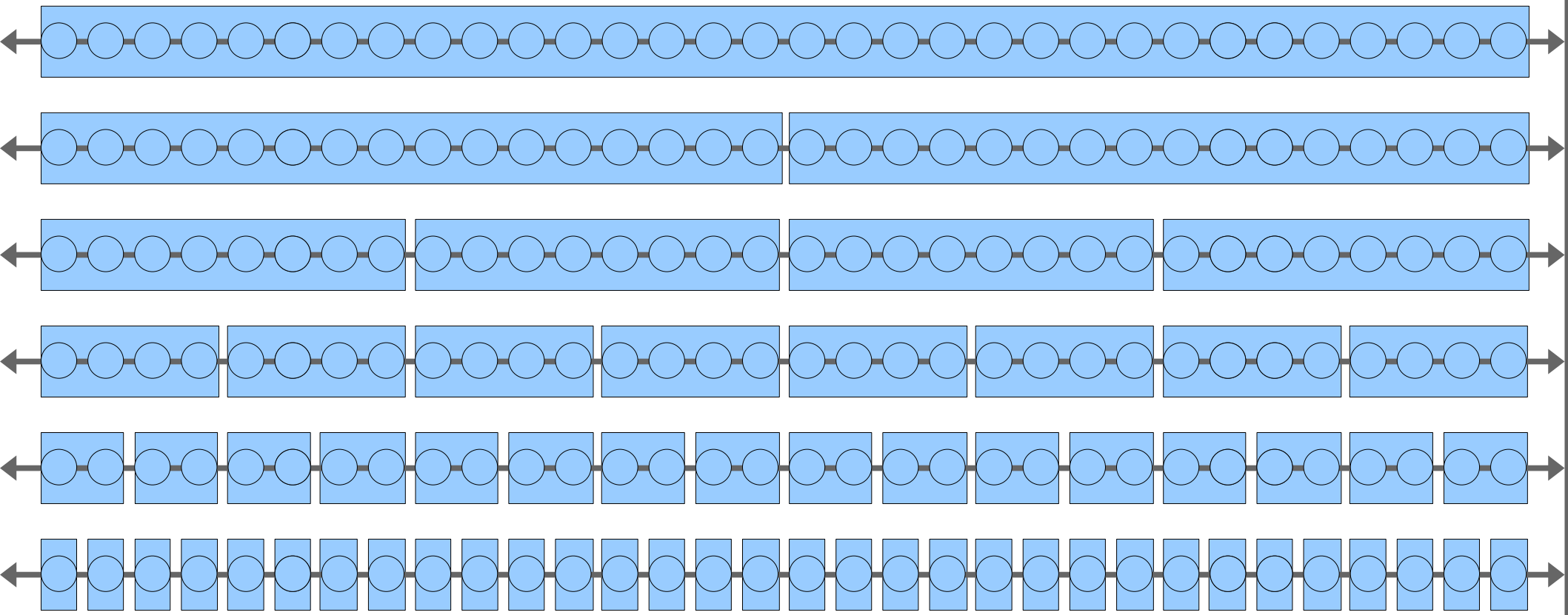
Why?

Now, ***private chat me your best guess***. Not sure? Answer with “??”.



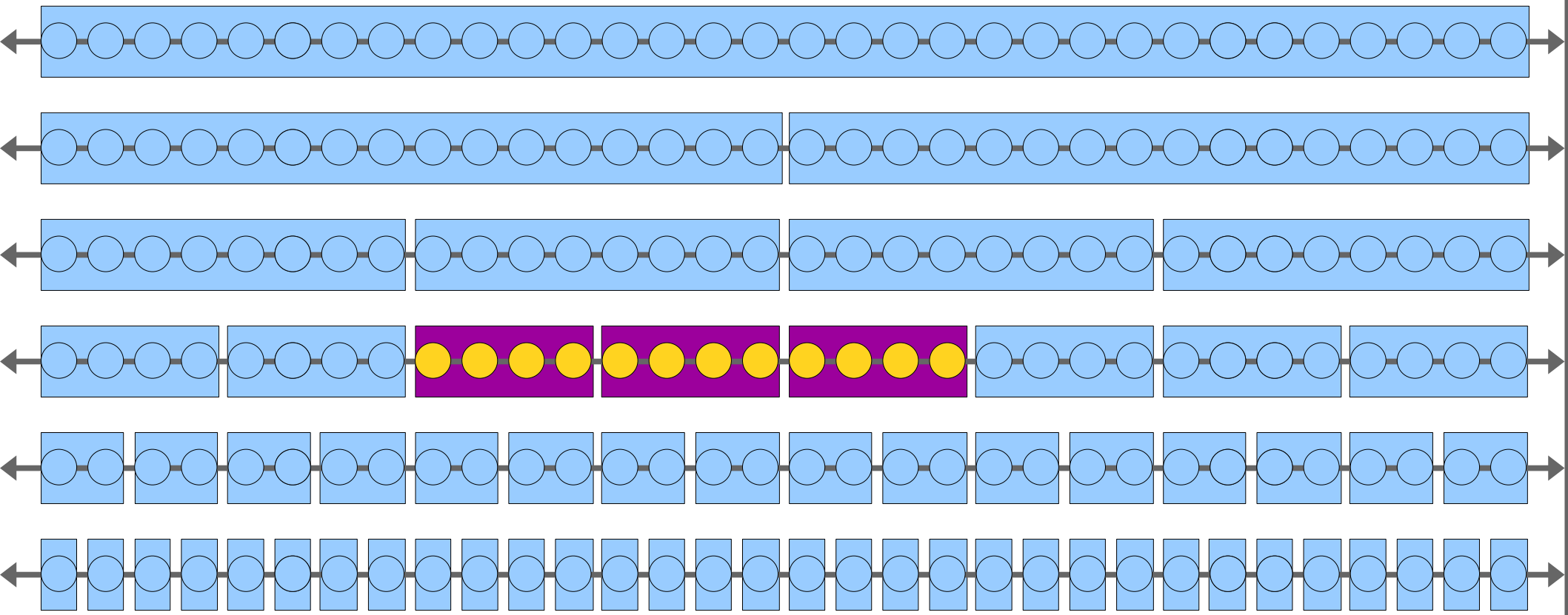
Claim: Every query range can be covered using only O(log n) total blocks.

Proof Idea: We use at most two blocks per row; there are O(log n) rows.



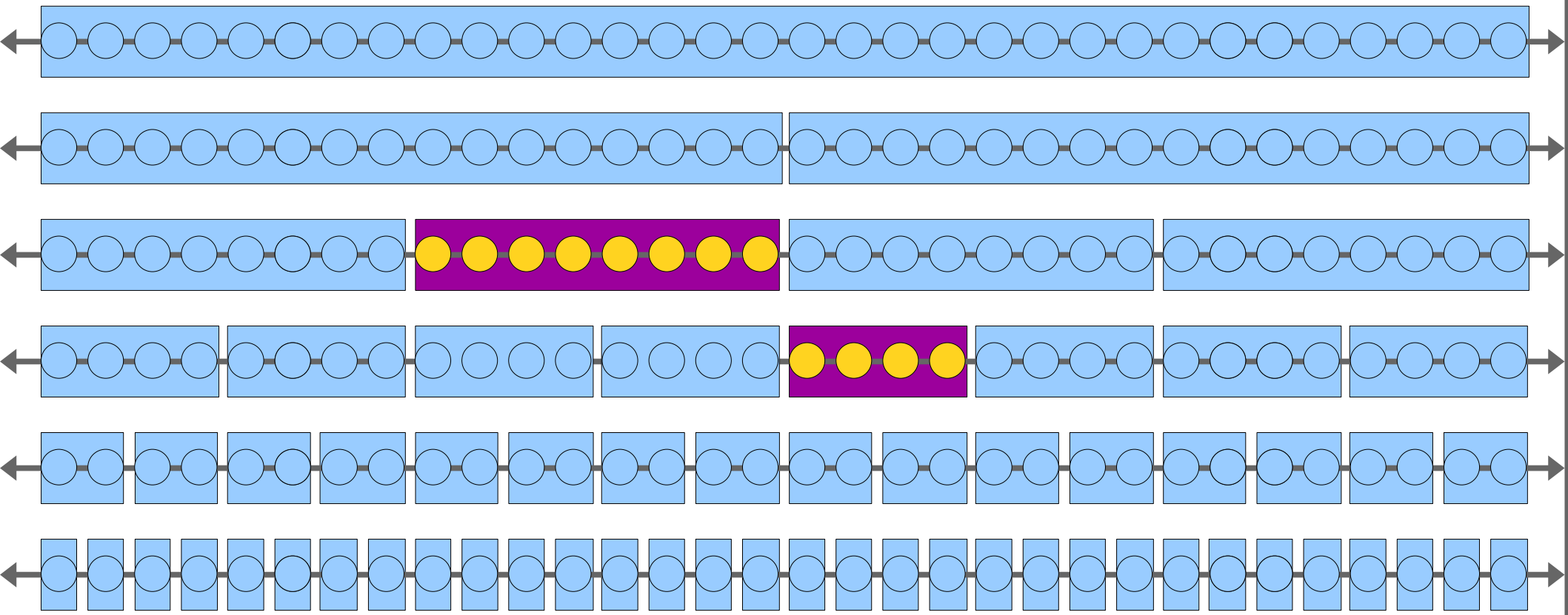
Claim: Every query range can be covered using only O(log n) total blocks.

Proof Idea: We use at most two blocks per row; there are O(log n) rows.



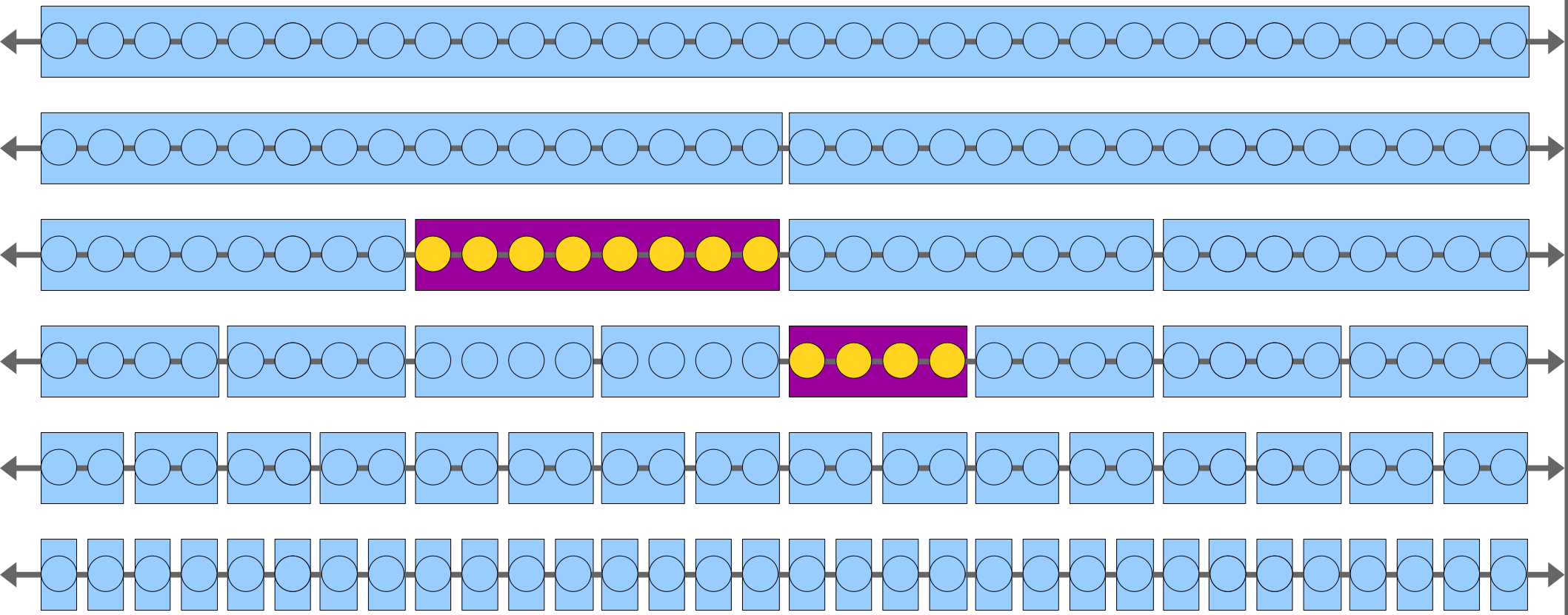
Claim: Every query range can be covered using only O(log n) total blocks.

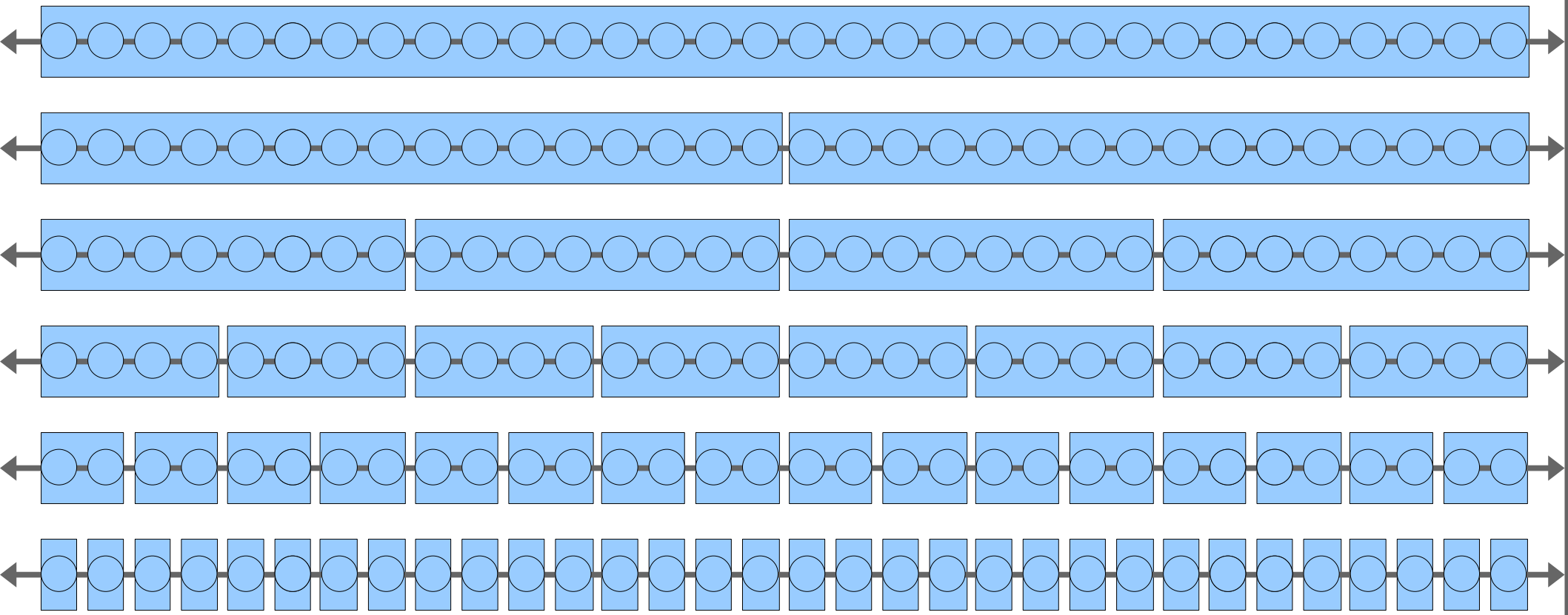
Proof Idea: We use at most two blocks per row; there are O(log n) rows.



Claim: Every query range can be covered using only O(log n) total blocks.

Question: How do we figure out which blocks to use?

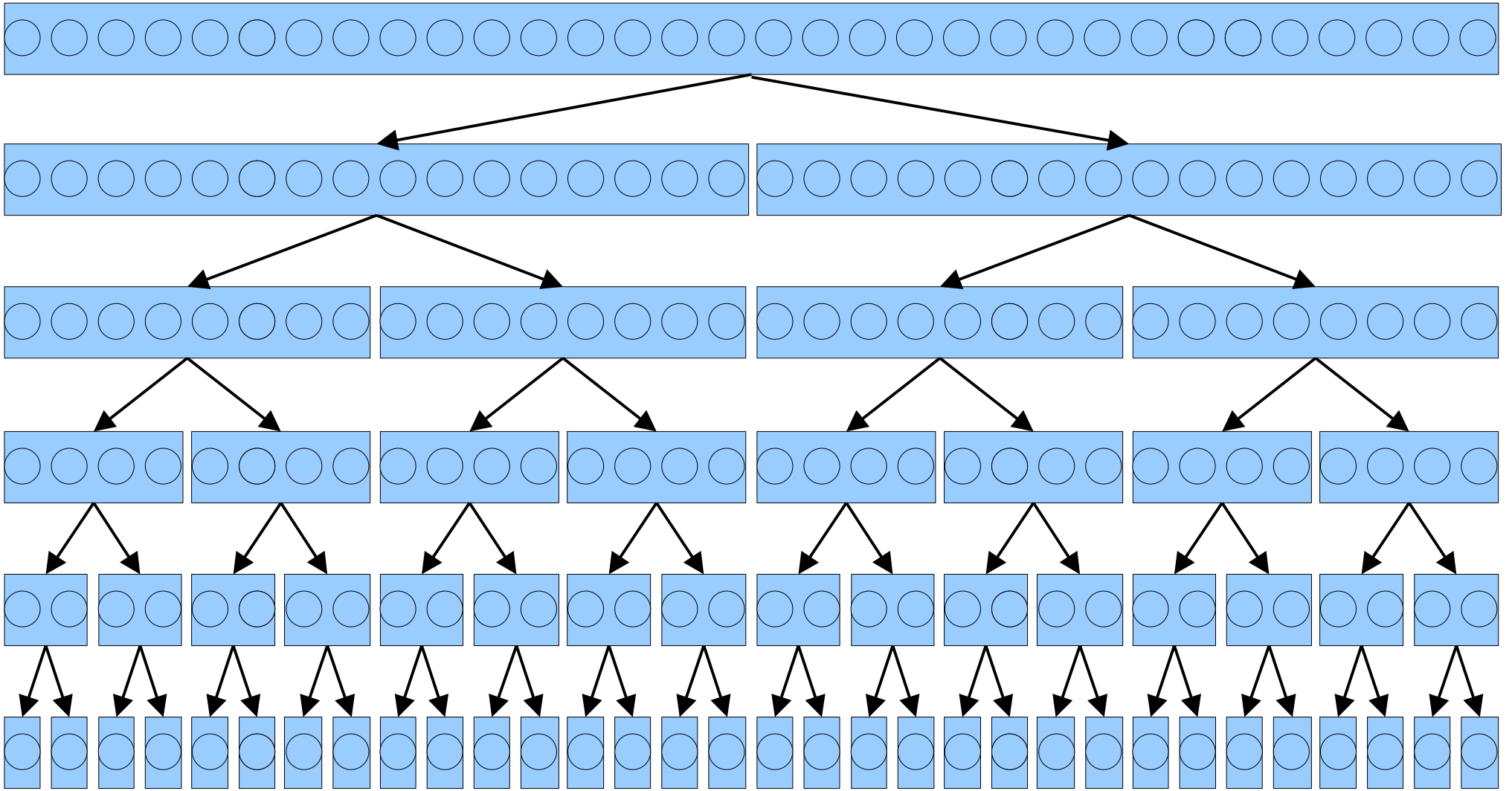




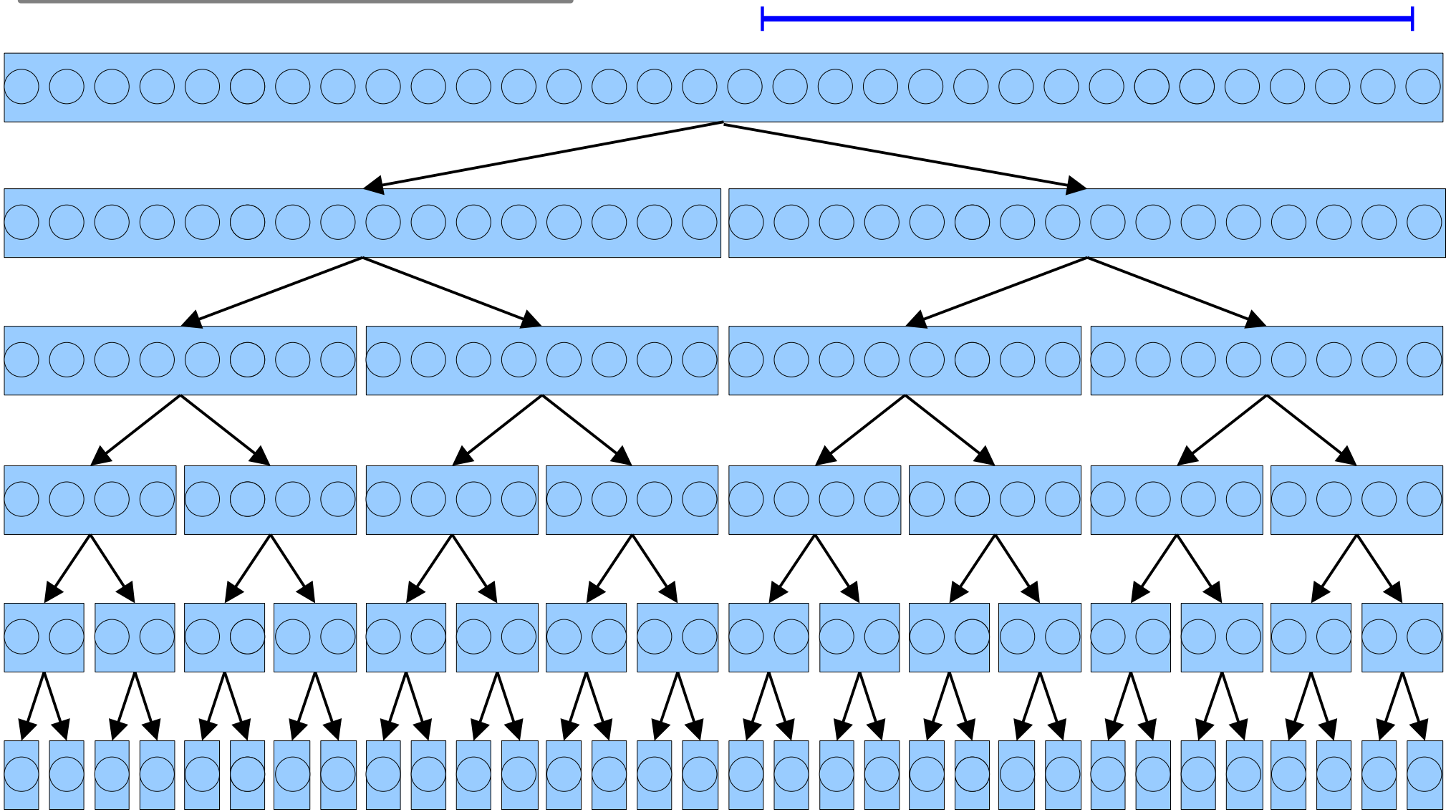
Each node represents a range of points, stored sorted by their y coordinates.

Each internal node stores pointers to two child nodes representing the first and second half of the points in its range.

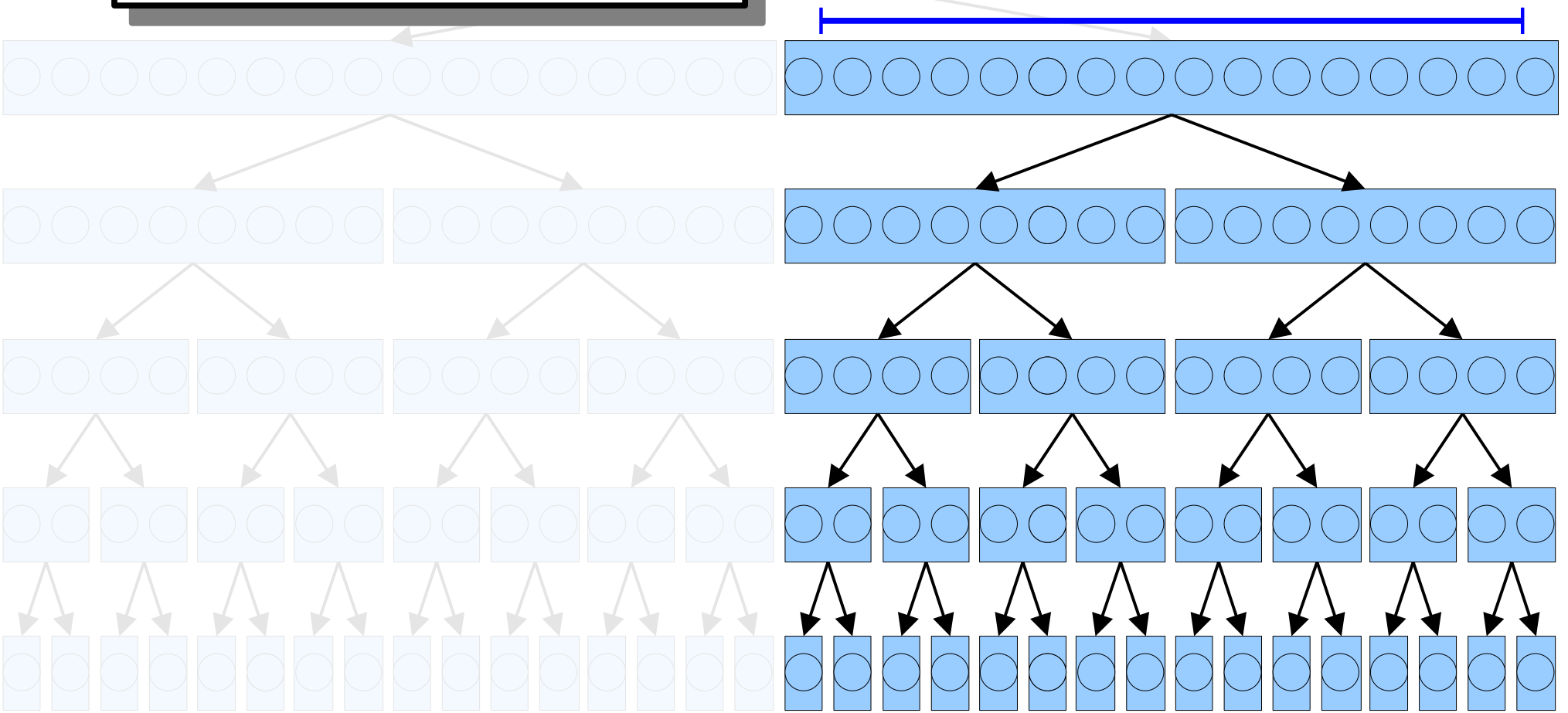
This data structure is called a ***range tree***.



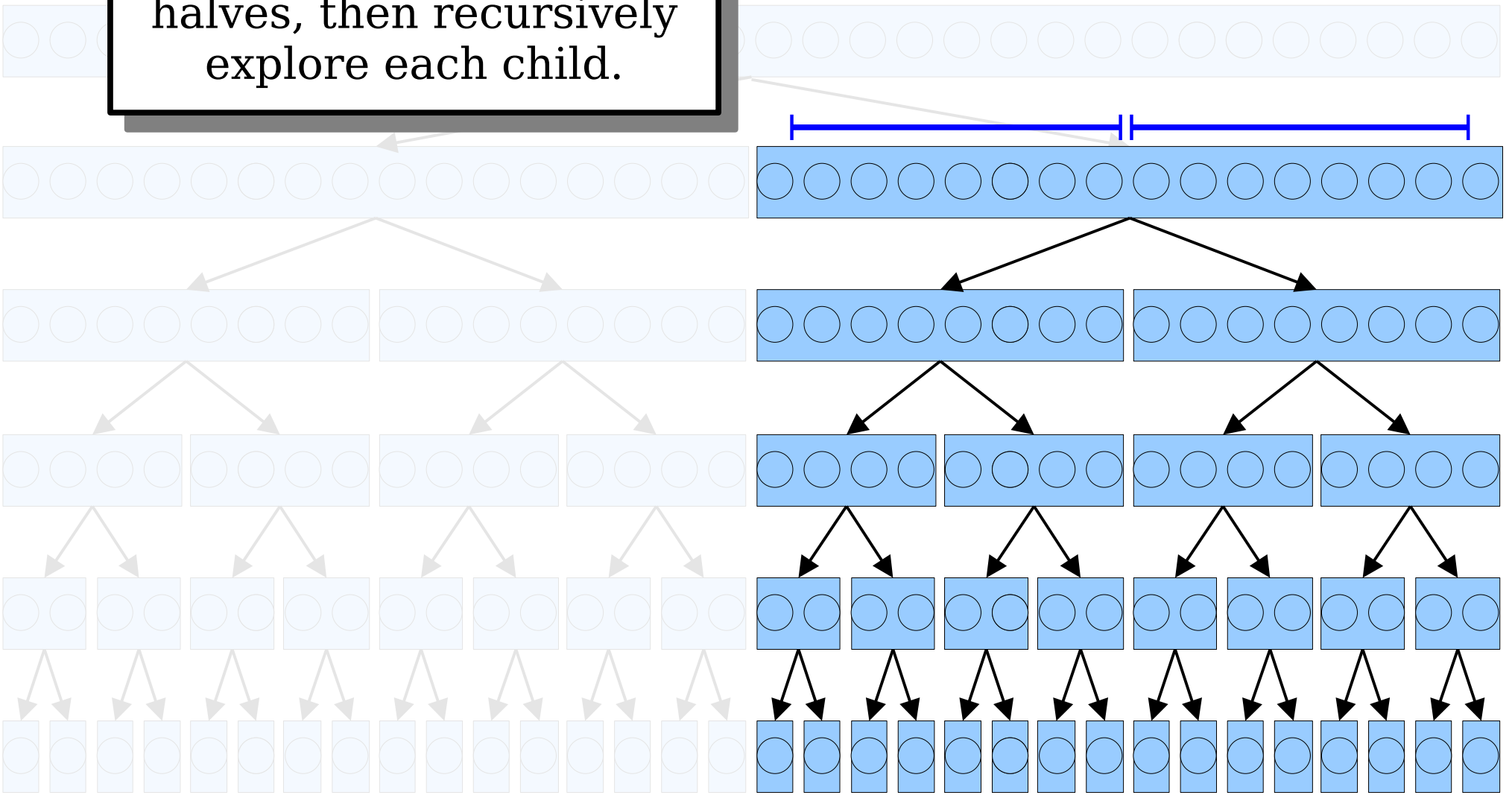
Range is purely in the right half; explore right child.

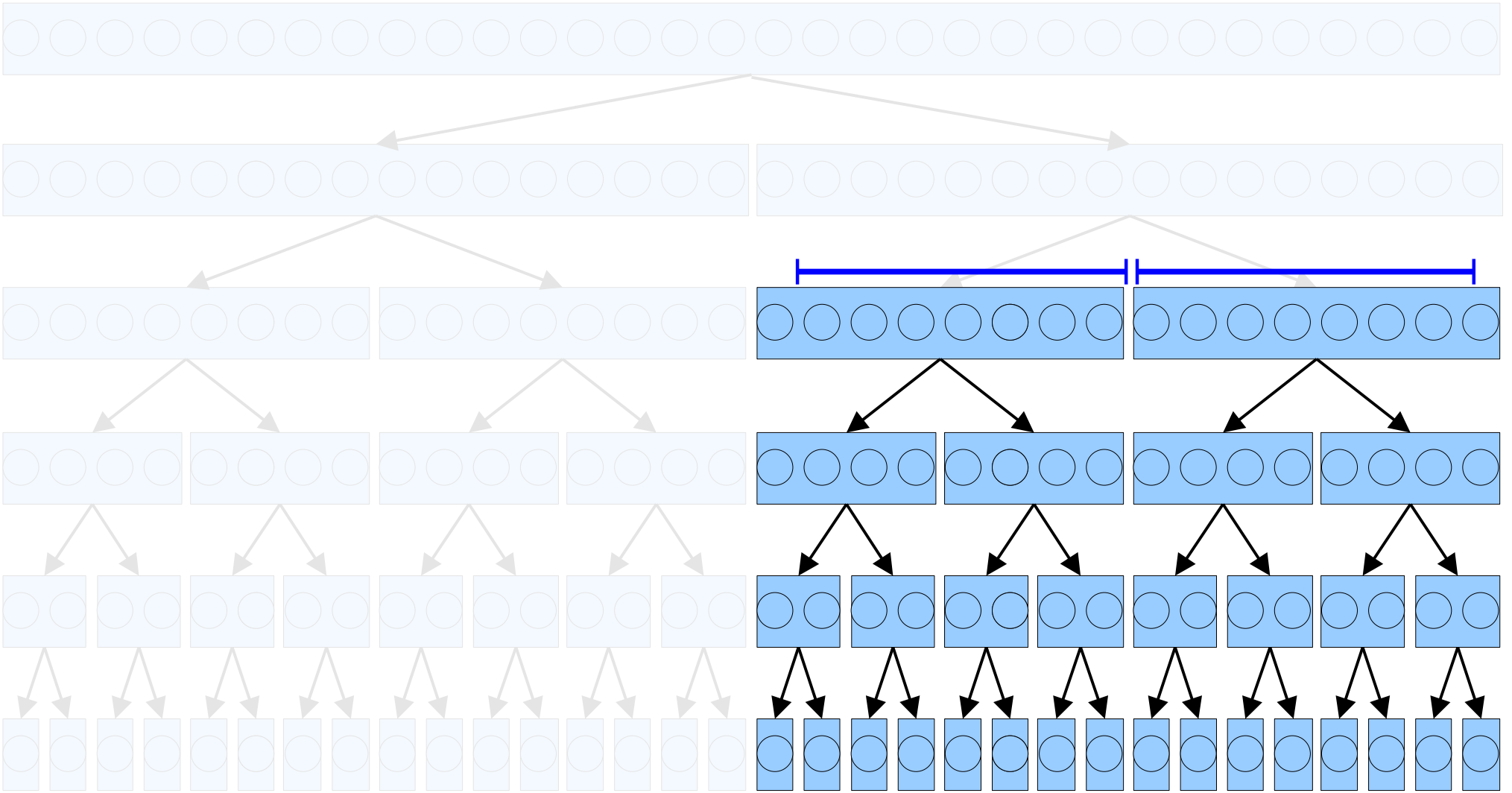


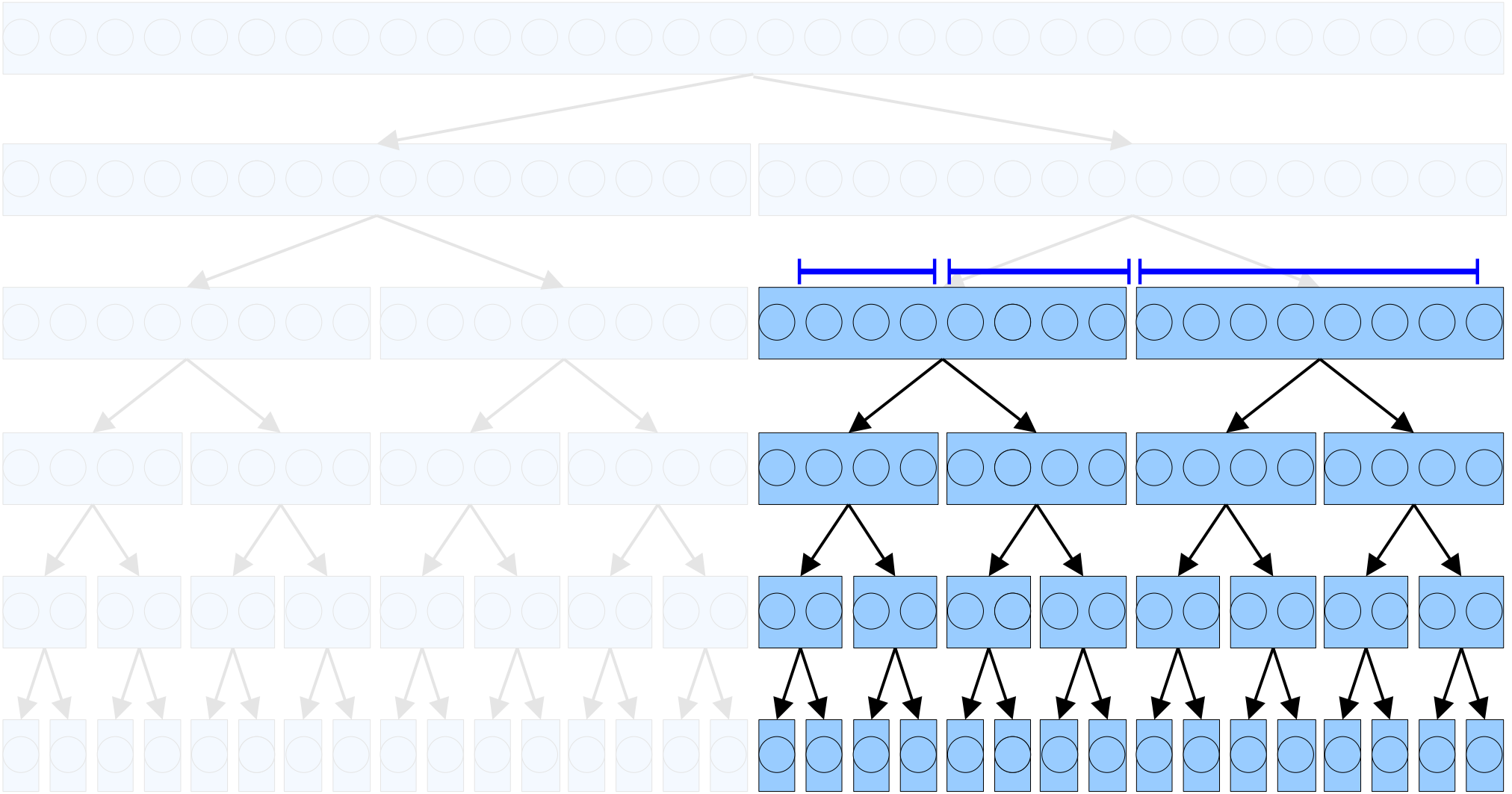
Range spans both halves. Split the range into two halves, then recursively explore each child.

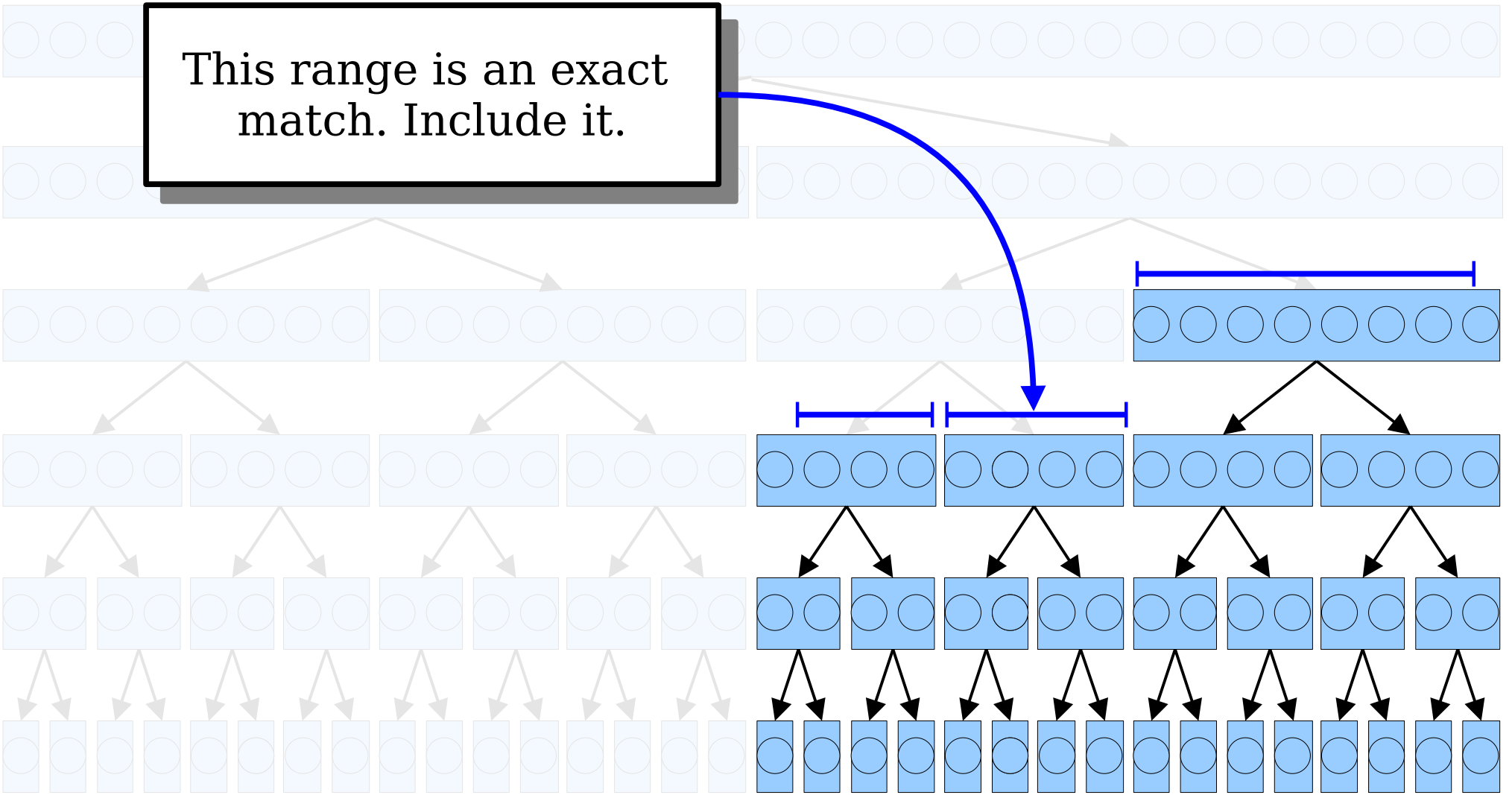


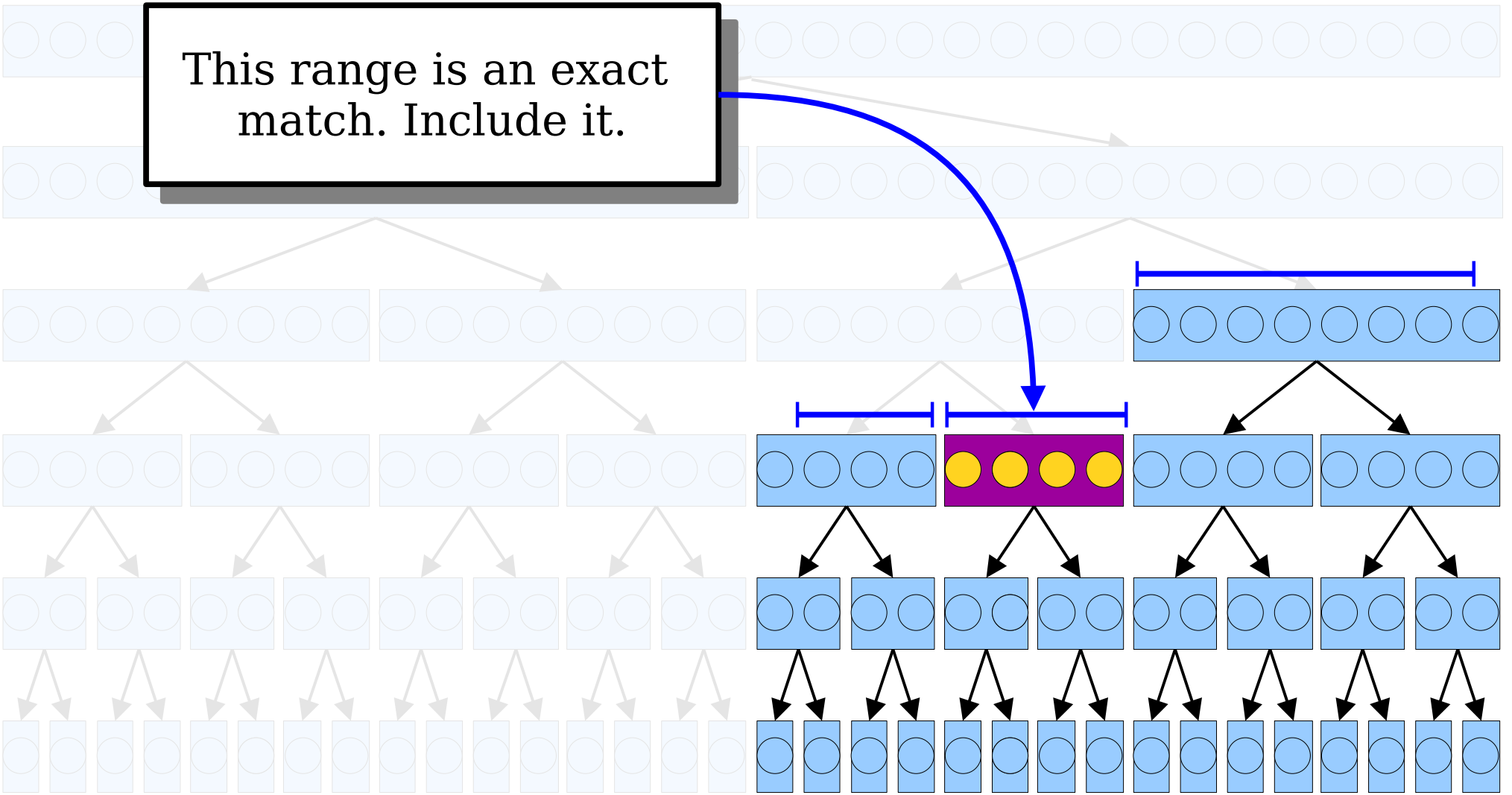
Range spans both halves. Split the range into two halves, then recursively explore each child.

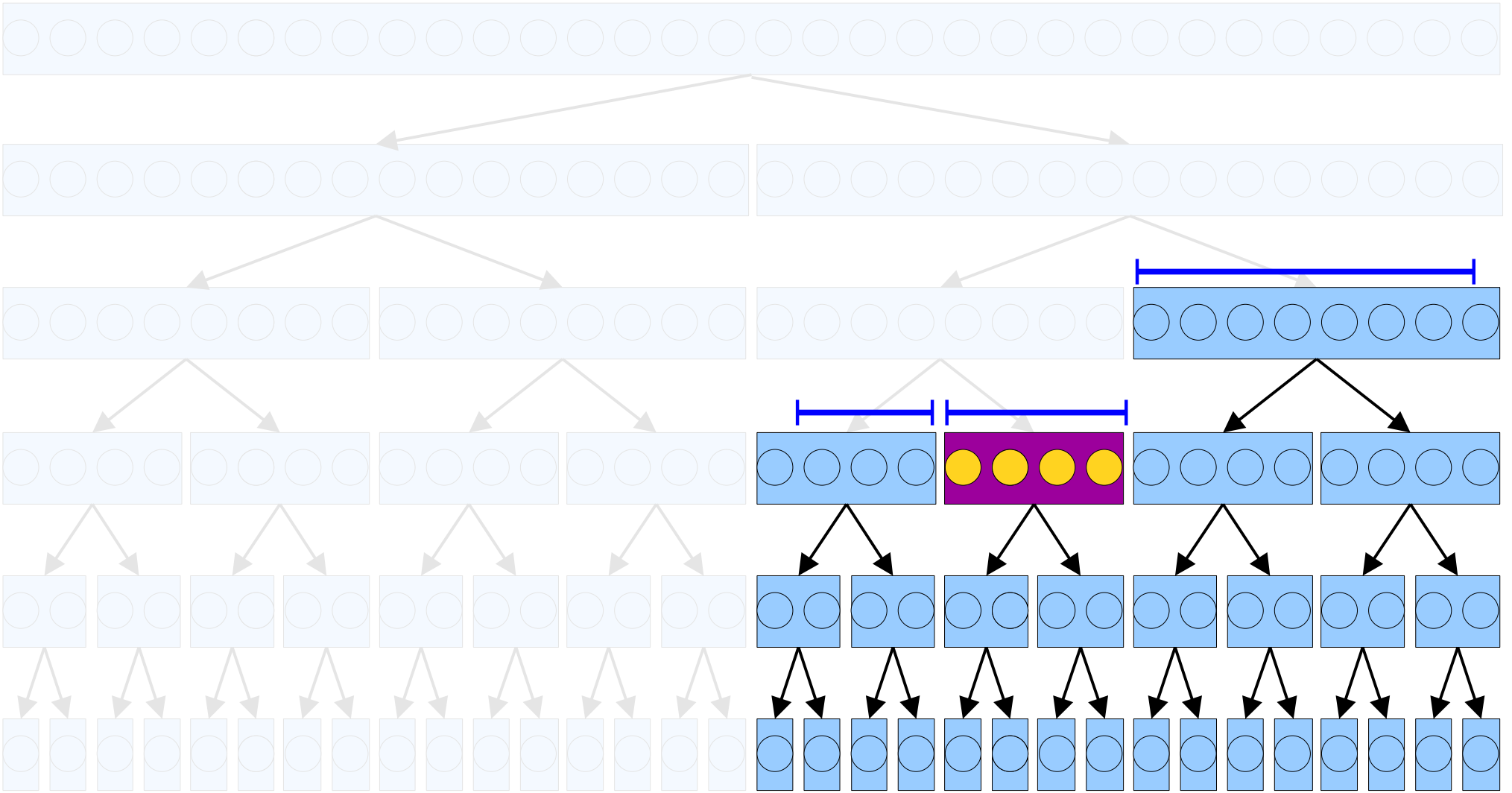


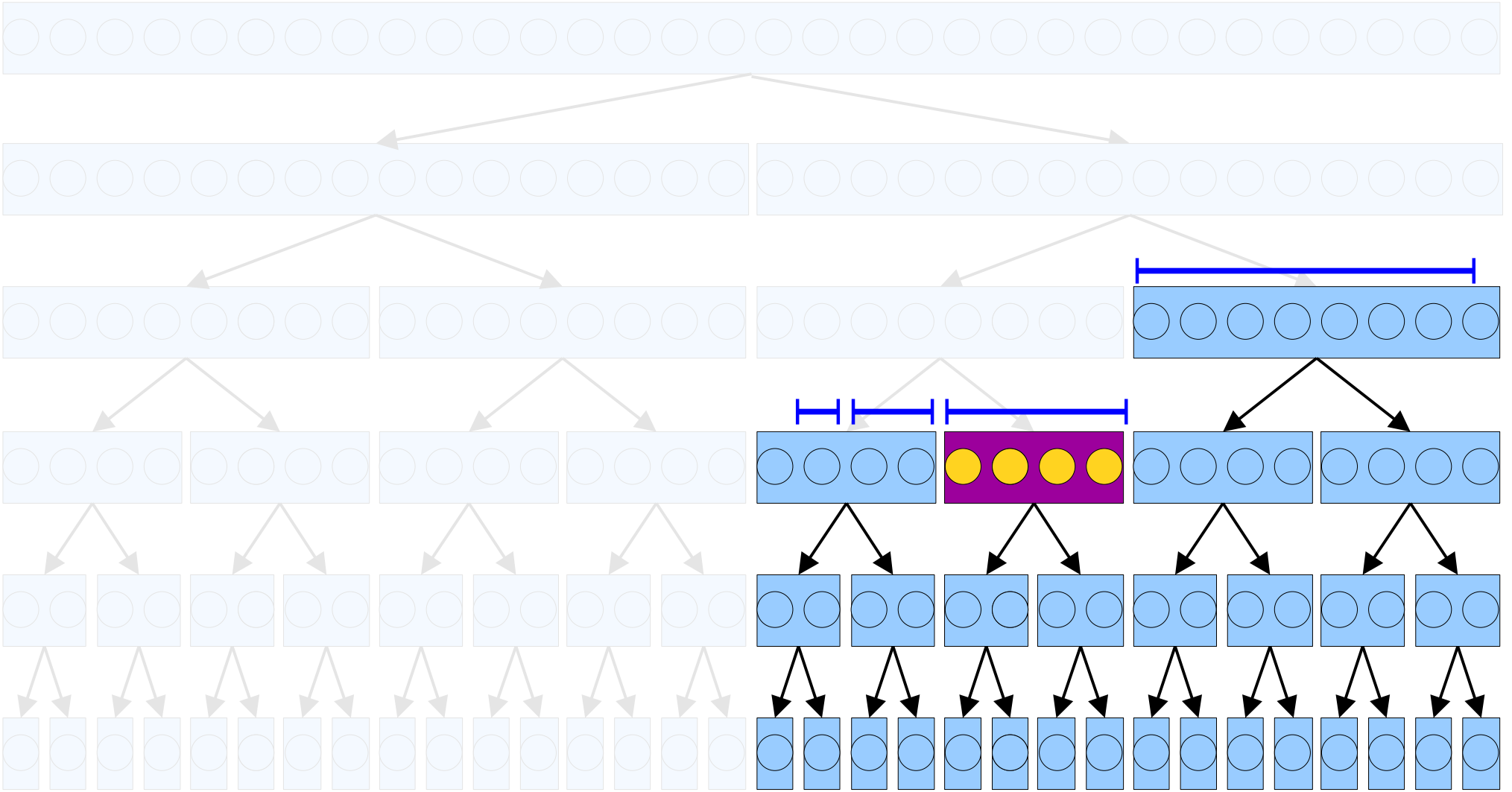


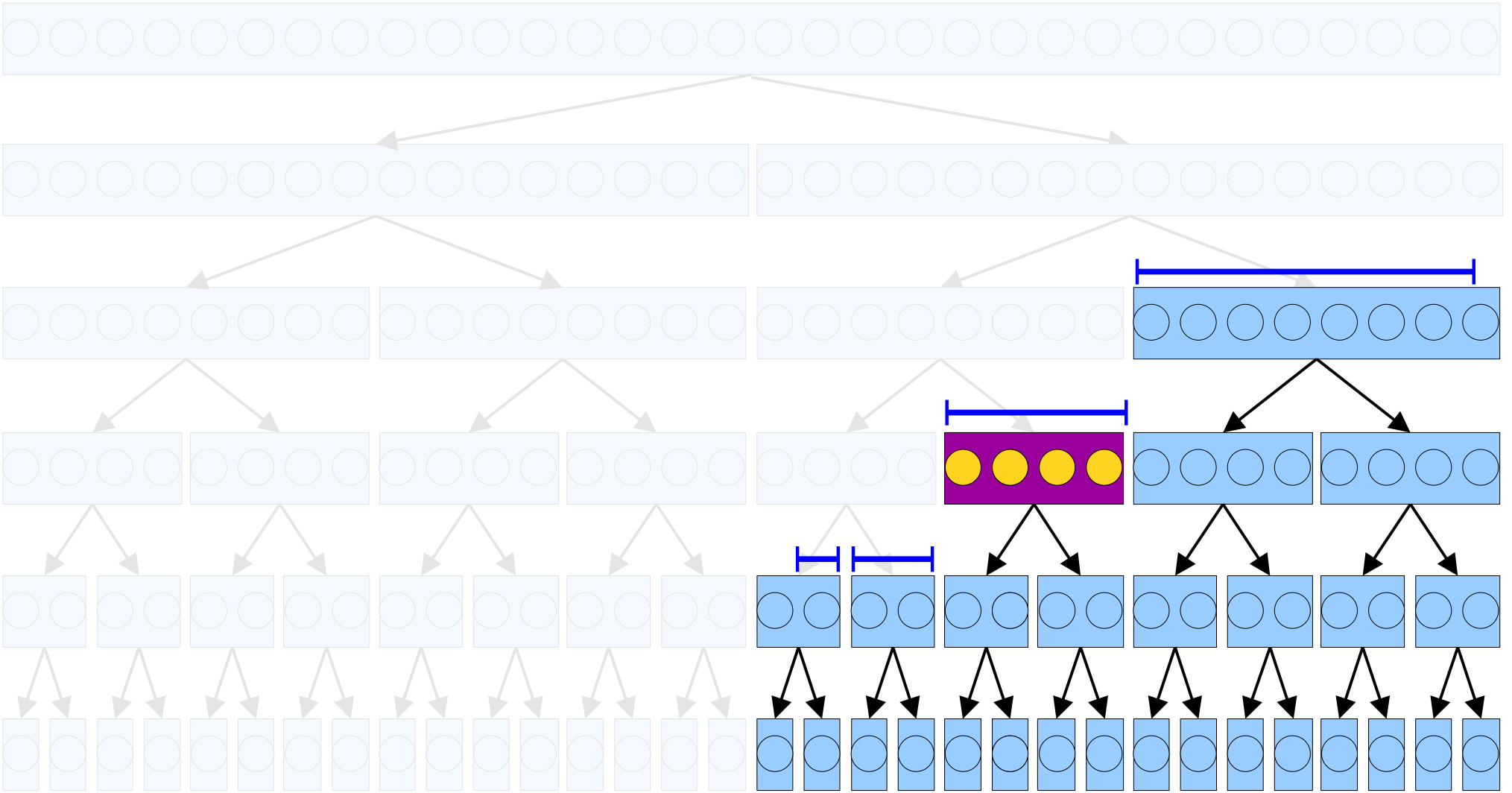


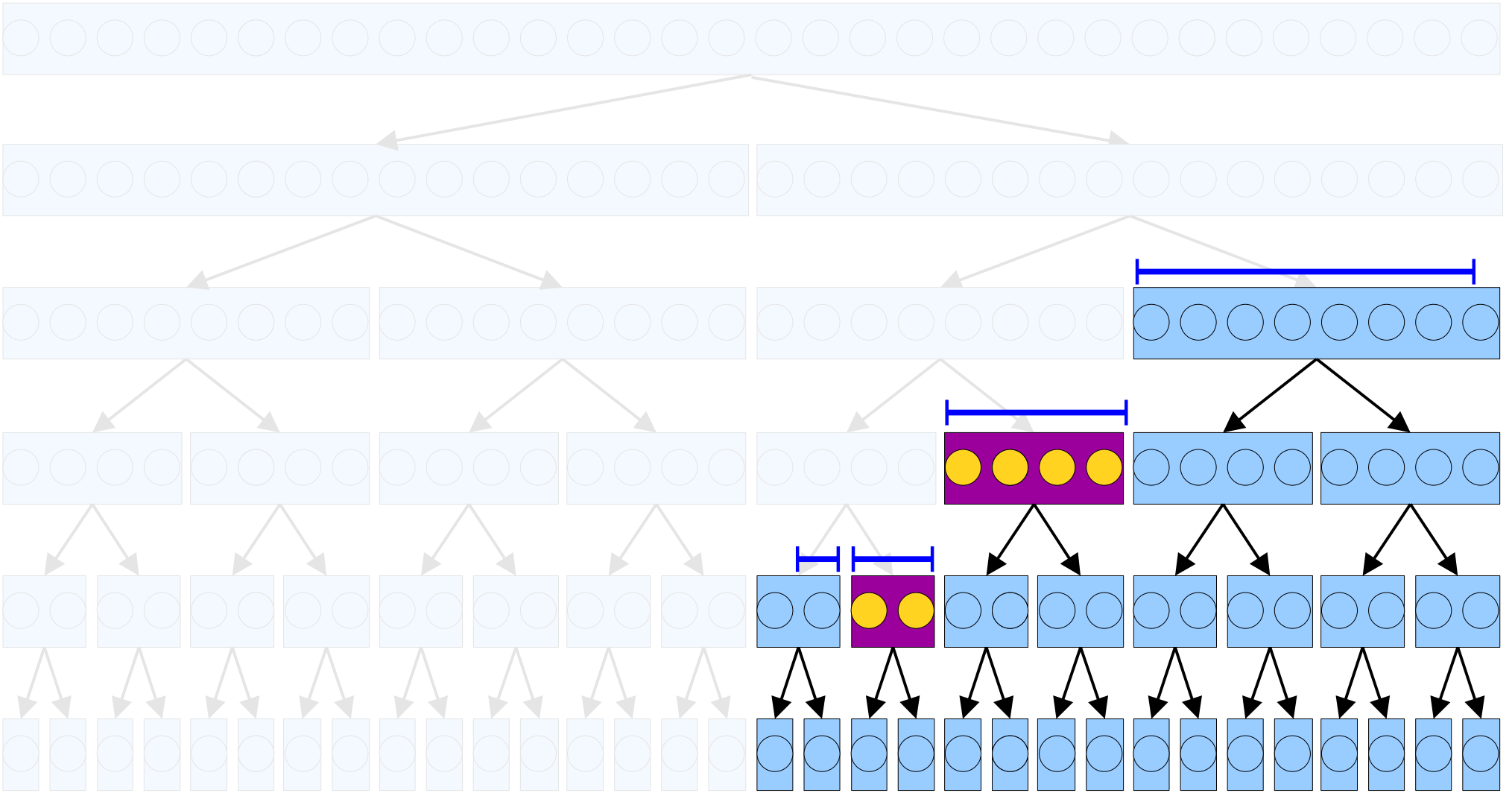

This range is an exact match. Include it.

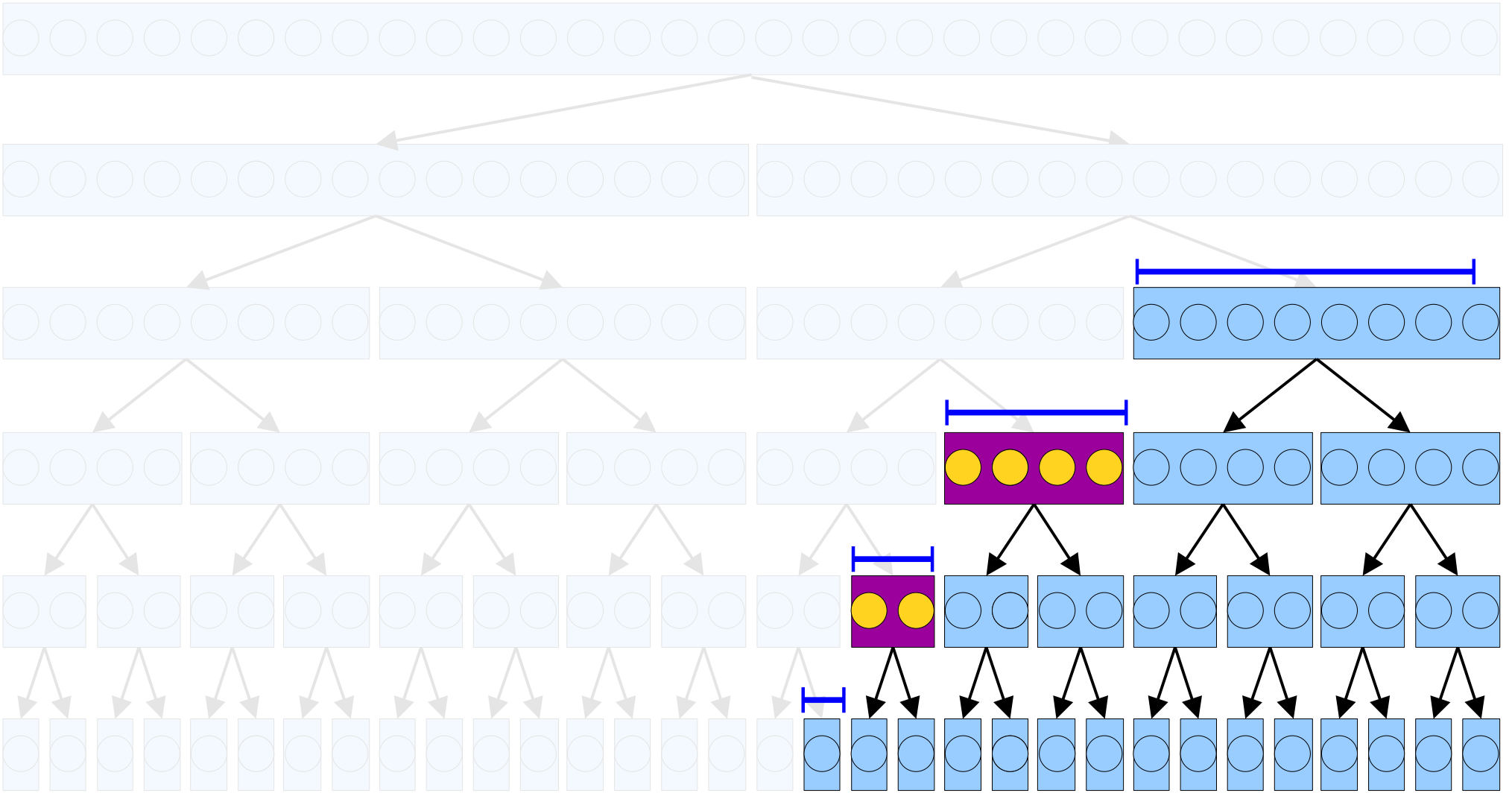

This range is an exact match. Include it.

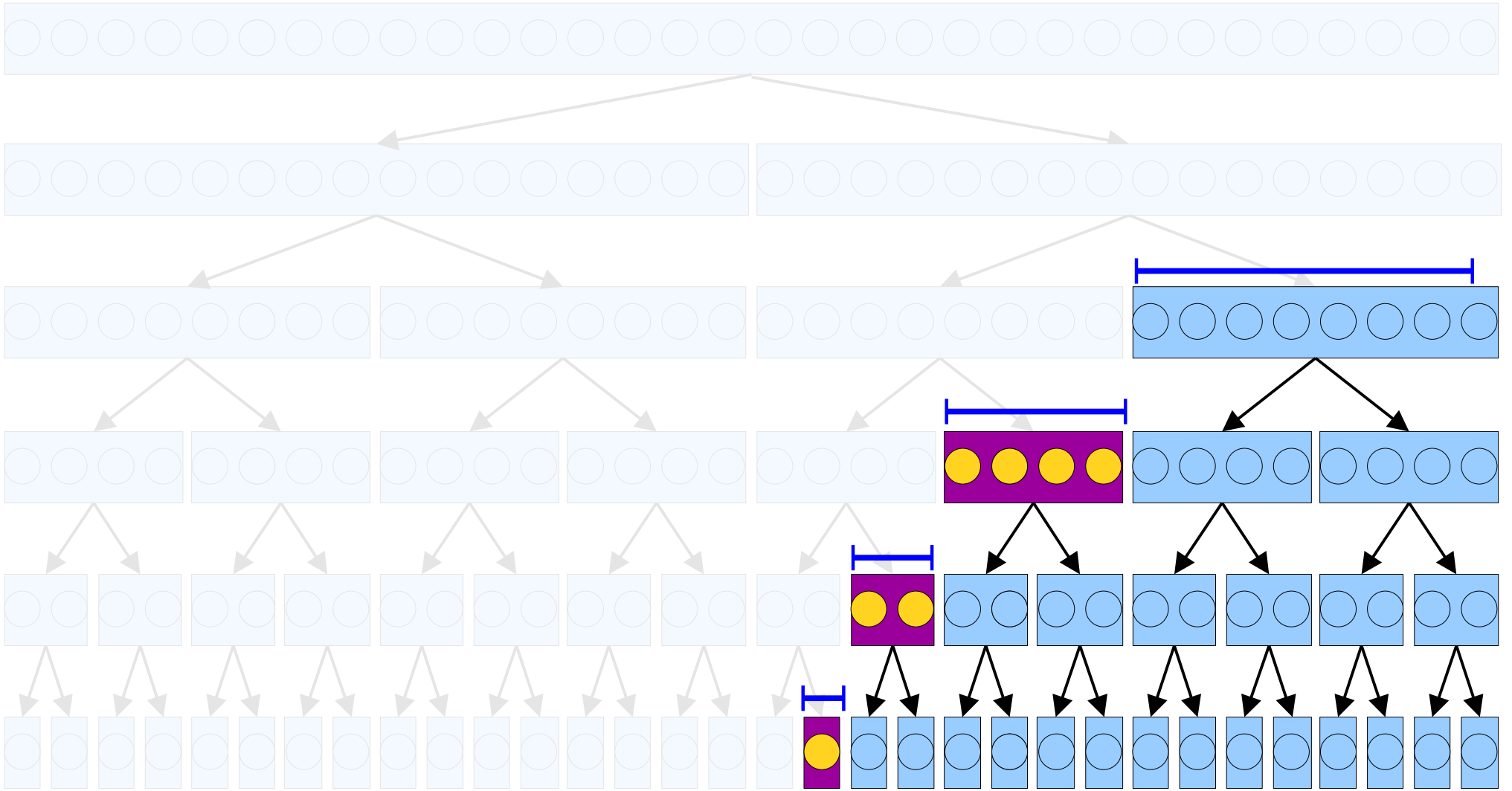


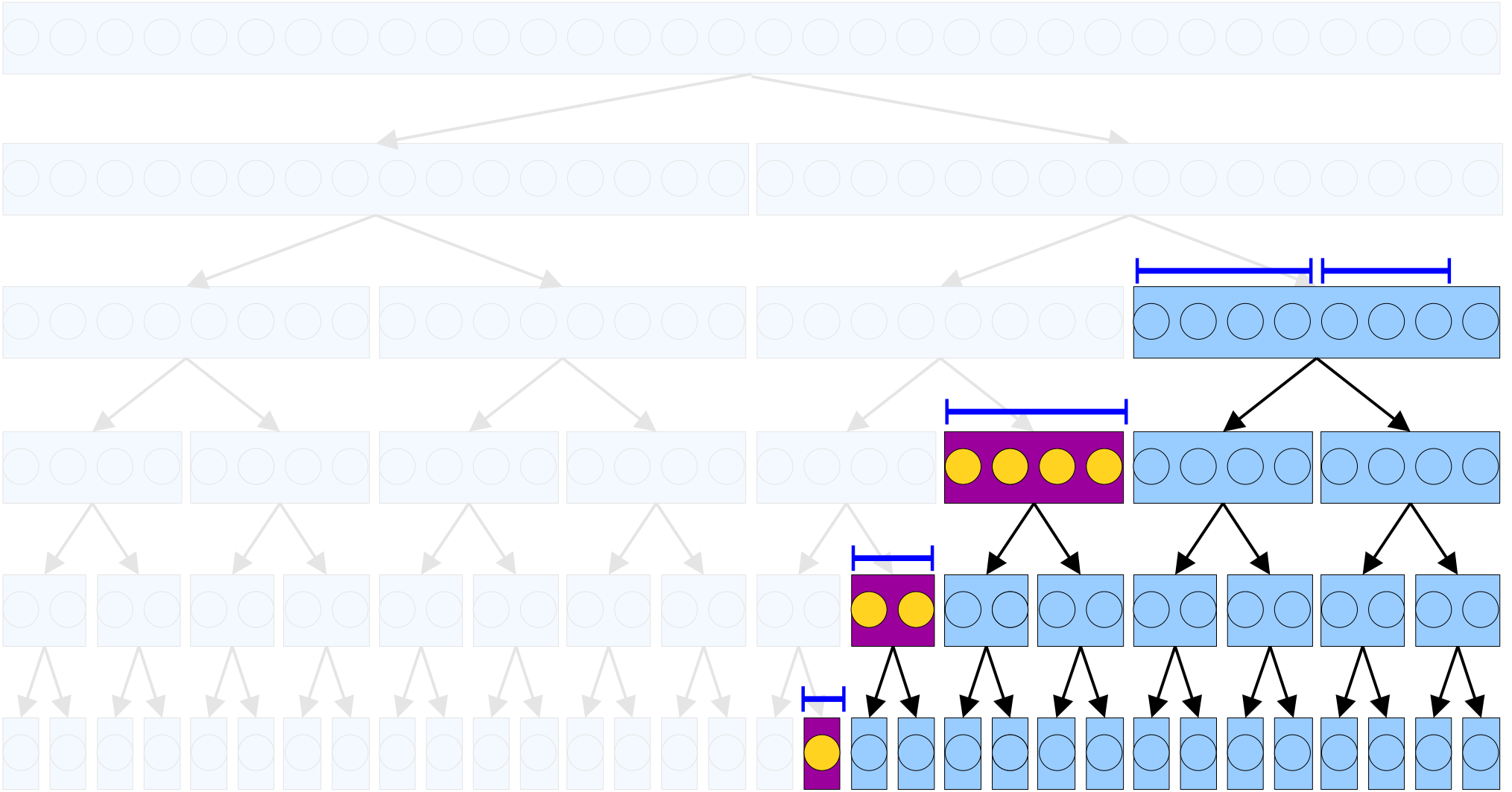


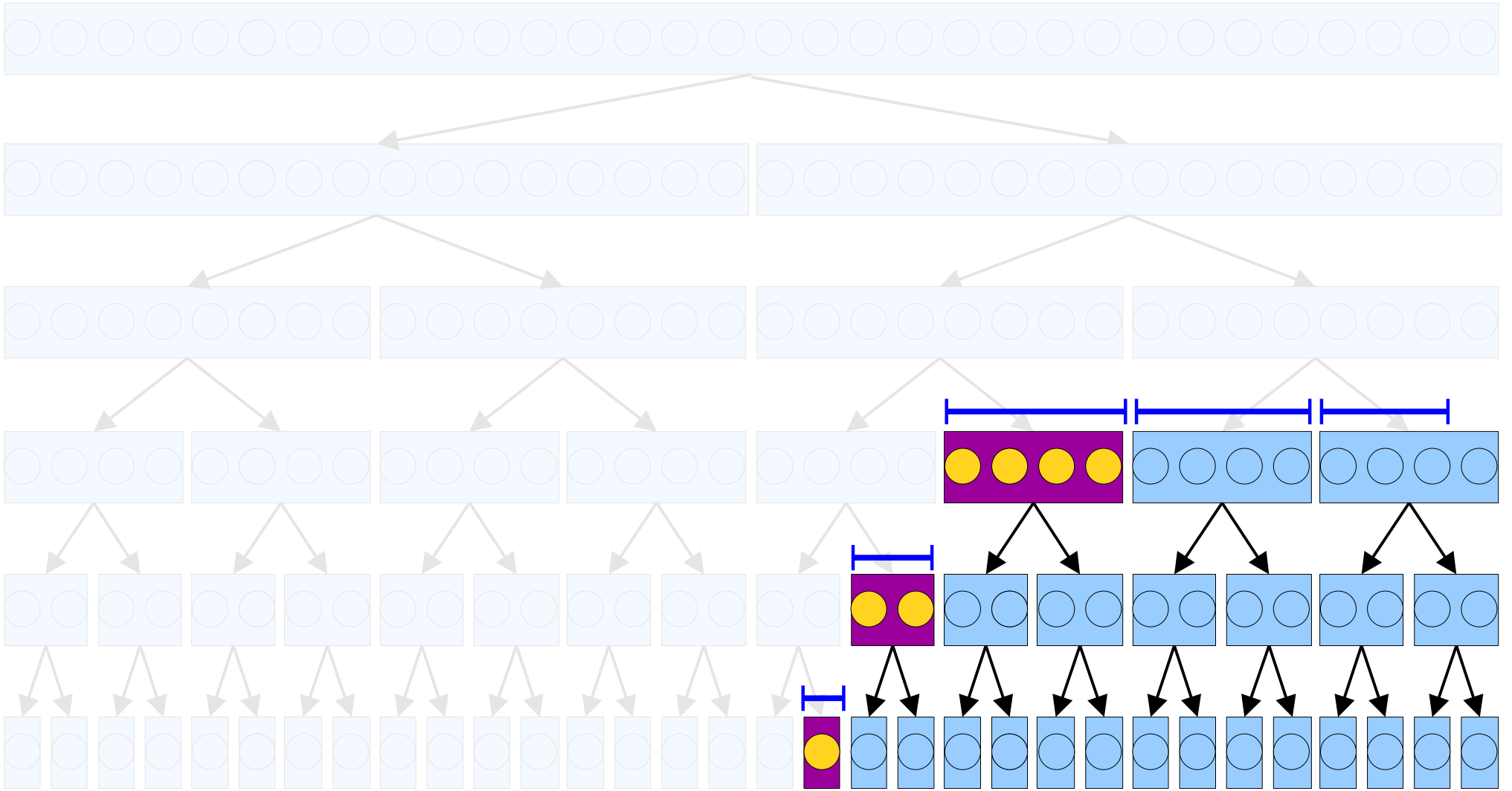


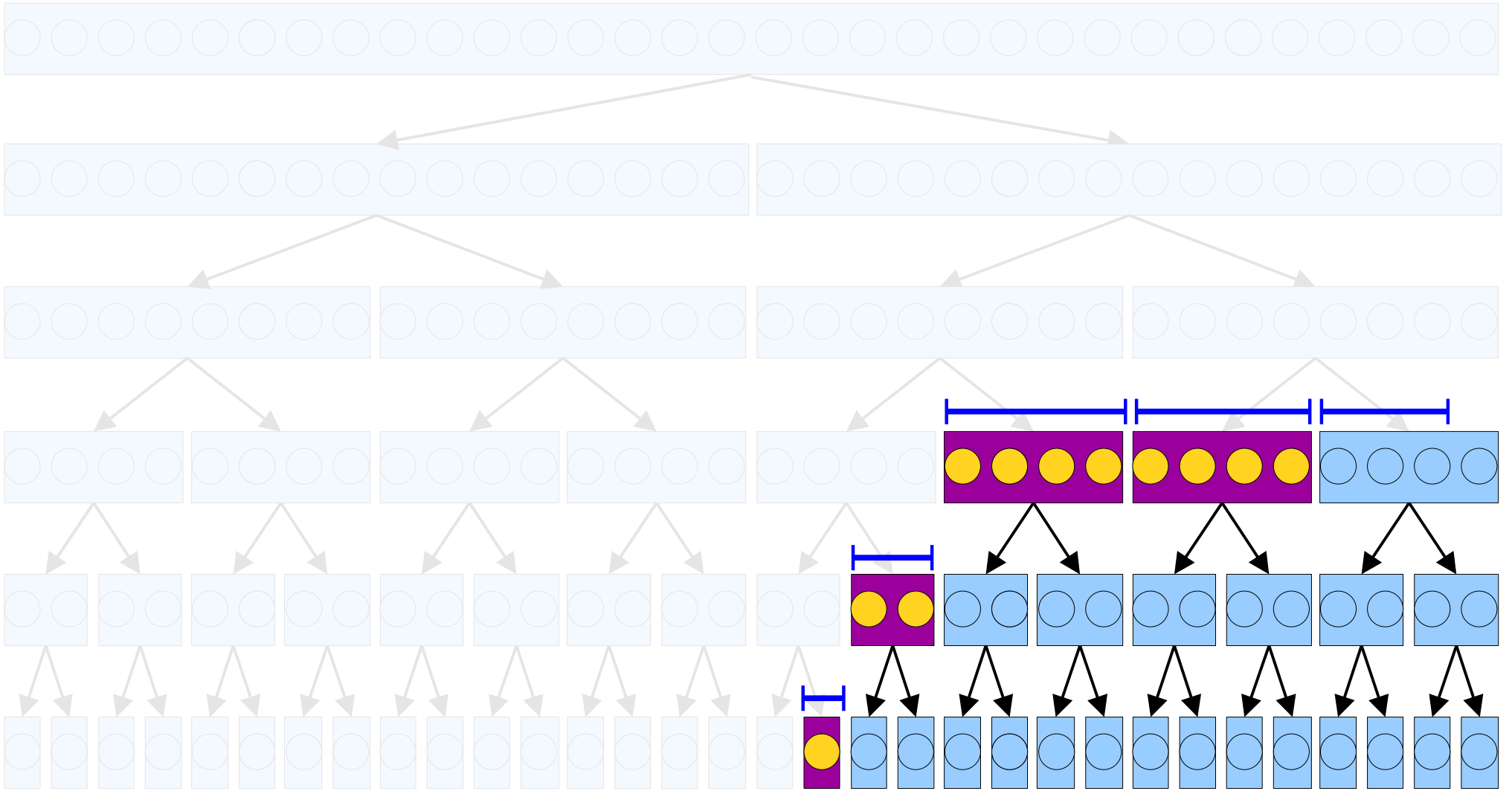


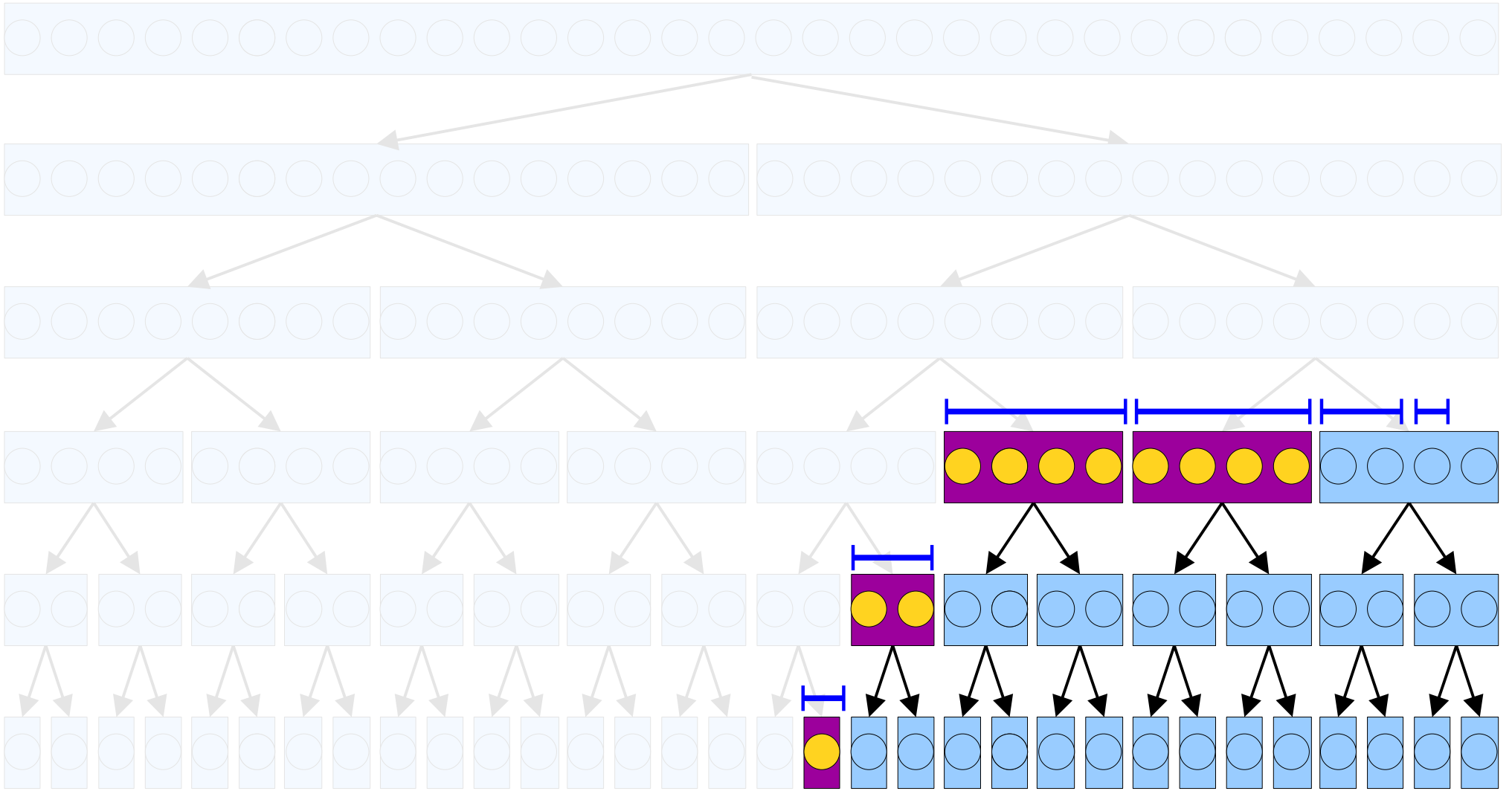


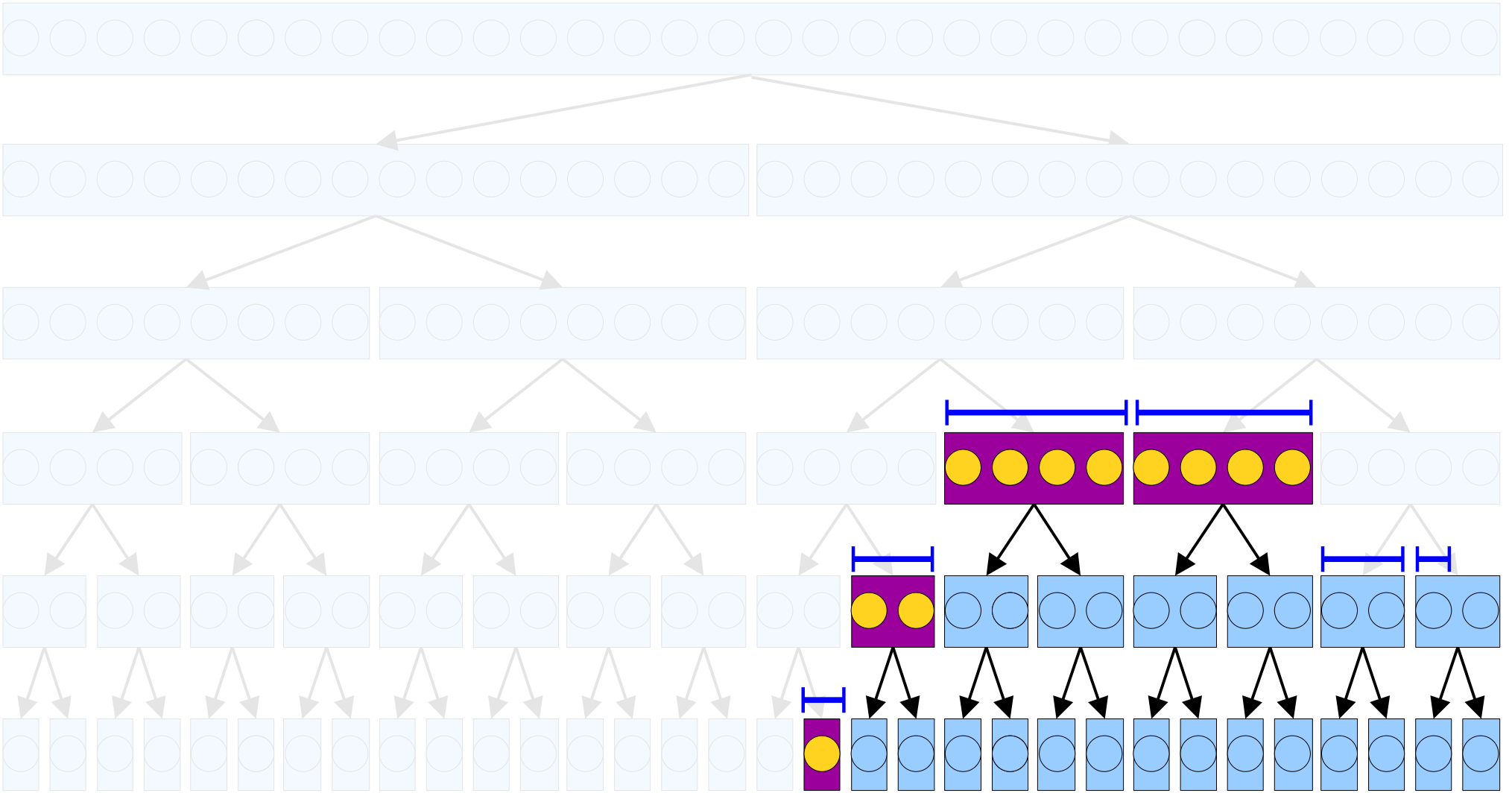


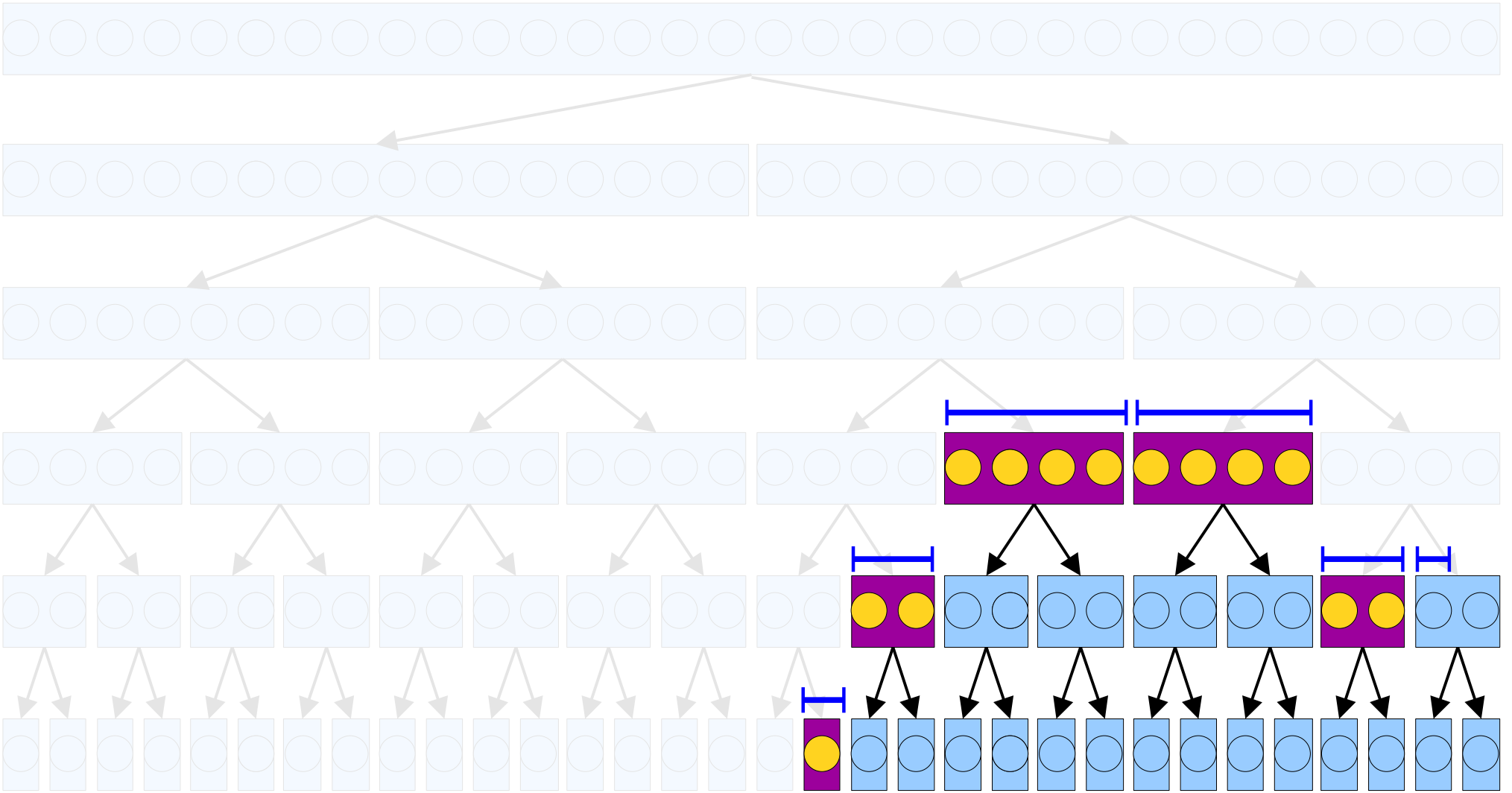


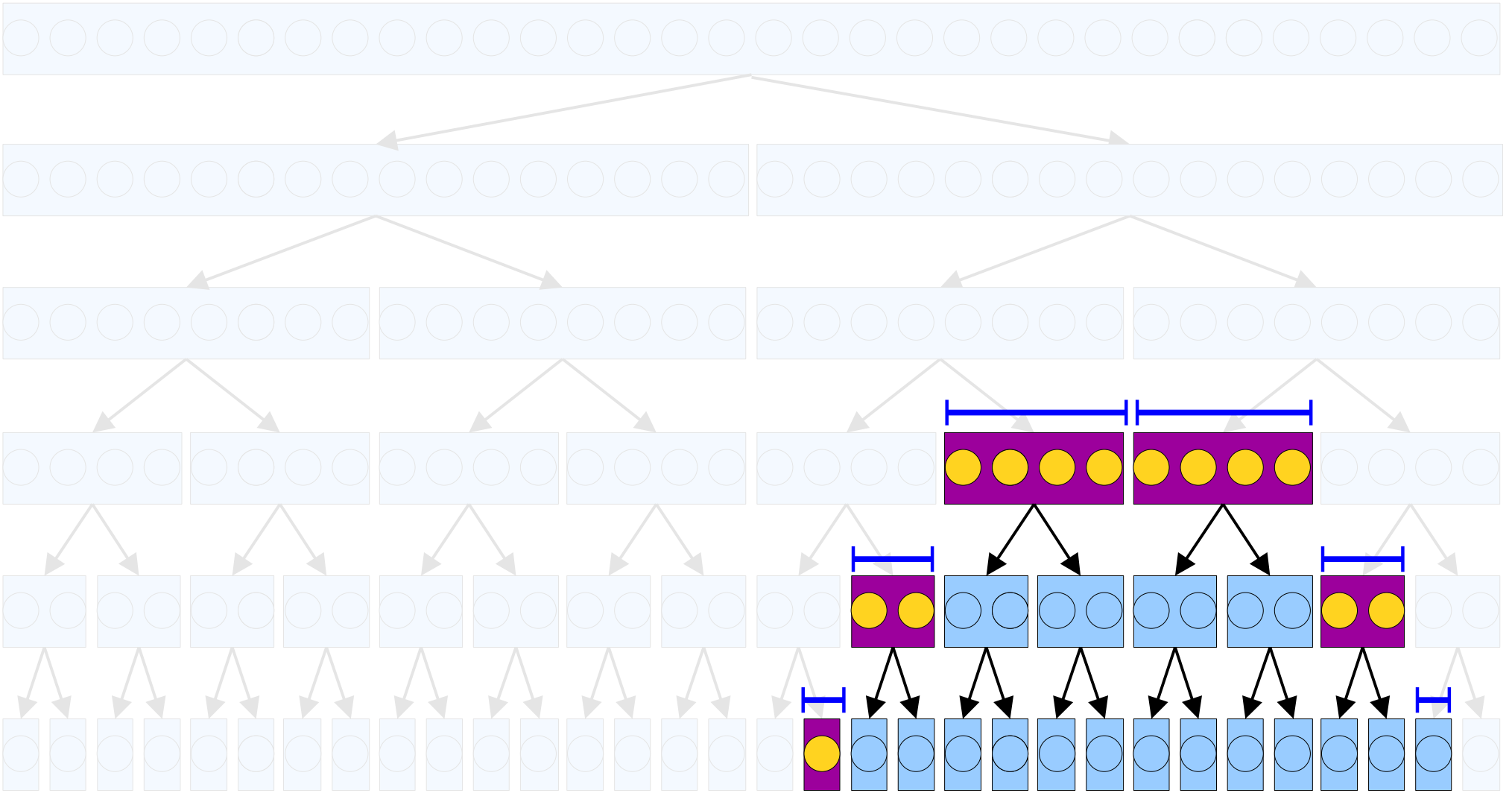


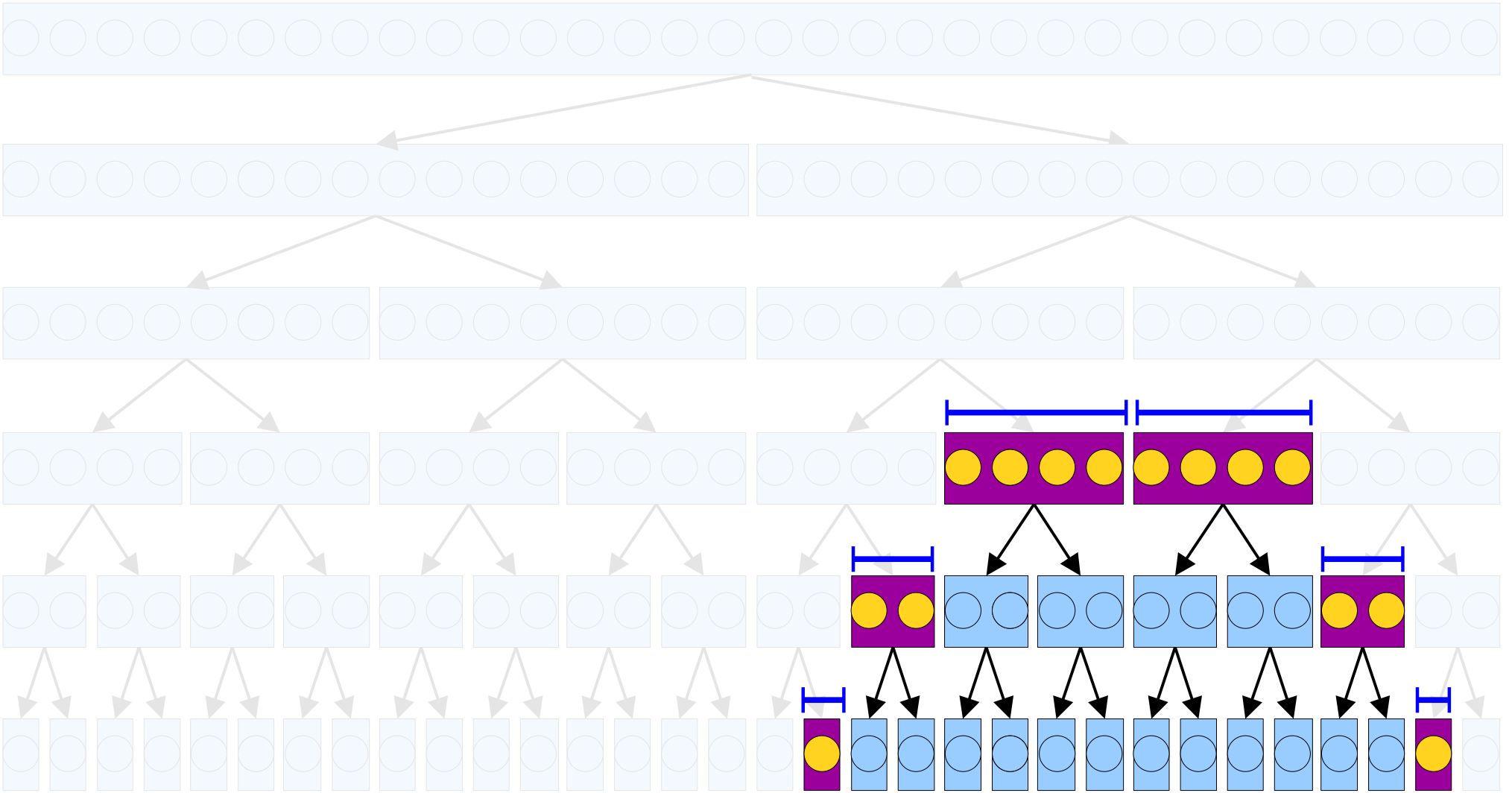




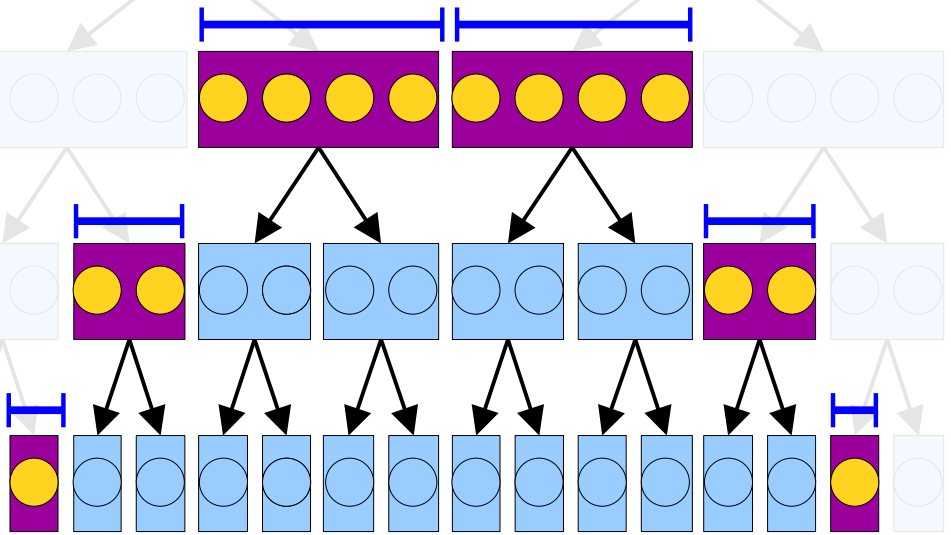








Now, do binary searches in the y direction in each block to find all matching points.



Range Tree Searches

- Here's the procedure for searching a range tree:
  - Walk the tree to find a collection of O(log n) blocks spanning the range of x coordinates to check.
  - For each of those blocks, use binary search to find the points in the block whose y coordinates are in range.
- Total work per query:

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

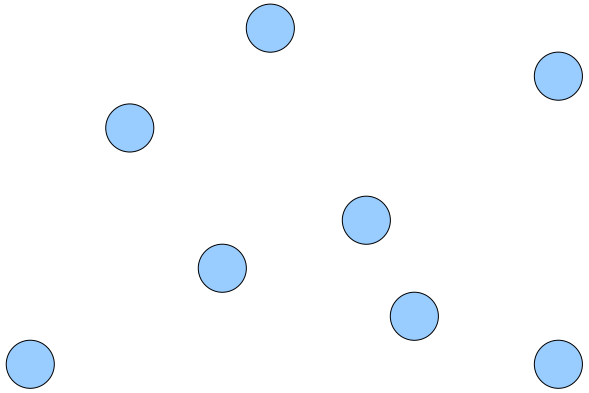
$$= \mathbf{O(\log^2 n + k)}.$$

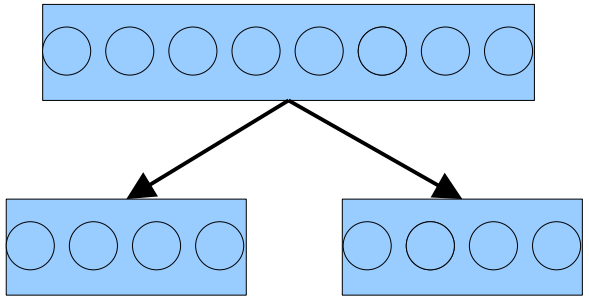
Great exercise: Show the O($\log^2 n$) bound is tight in the worst case.

Range Tree Construction

- What is the preprocessing time on a range tree?
  - We start with n points, which we need to sort by y coordinate.
  - We then recursively build two child nodes.
- Recurrence:

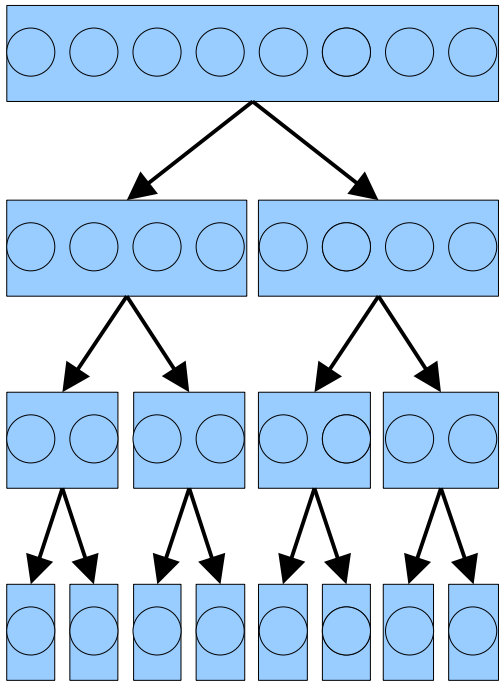
  $T(n) = 2T(n / 2) + O(n \log n)$.
- This recurrence solves to **$O(n \log^2 n)$**.





Range Tree Space Usage

- How much space does this data structure use?
  - Each layer stores one copy of each point.
  - There are O(log n) layers.
- Total space usage: **O(n log n)**.



The Story So Far

- The range tree uses more space than our blocking structure, but has better query times.
- As always: ***Can we do better?***

	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$
Range Tree*	$O(n \log^2 n)$	$O(\log^2 n + k)$	$O(n \log n)$

** Can easily be improved – stay tuned!*

The Story So Far

- The range tree uses more space than our blocking structure, but has better query times.
- As always: ***Can we do better?***

	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$
Range Tree*	$O(n \log^2 n)$	$O(\log^2 n + k)$	$O(n \log n)$

** Can easily be improved – stay tuned!*

Improving our Preprocessing

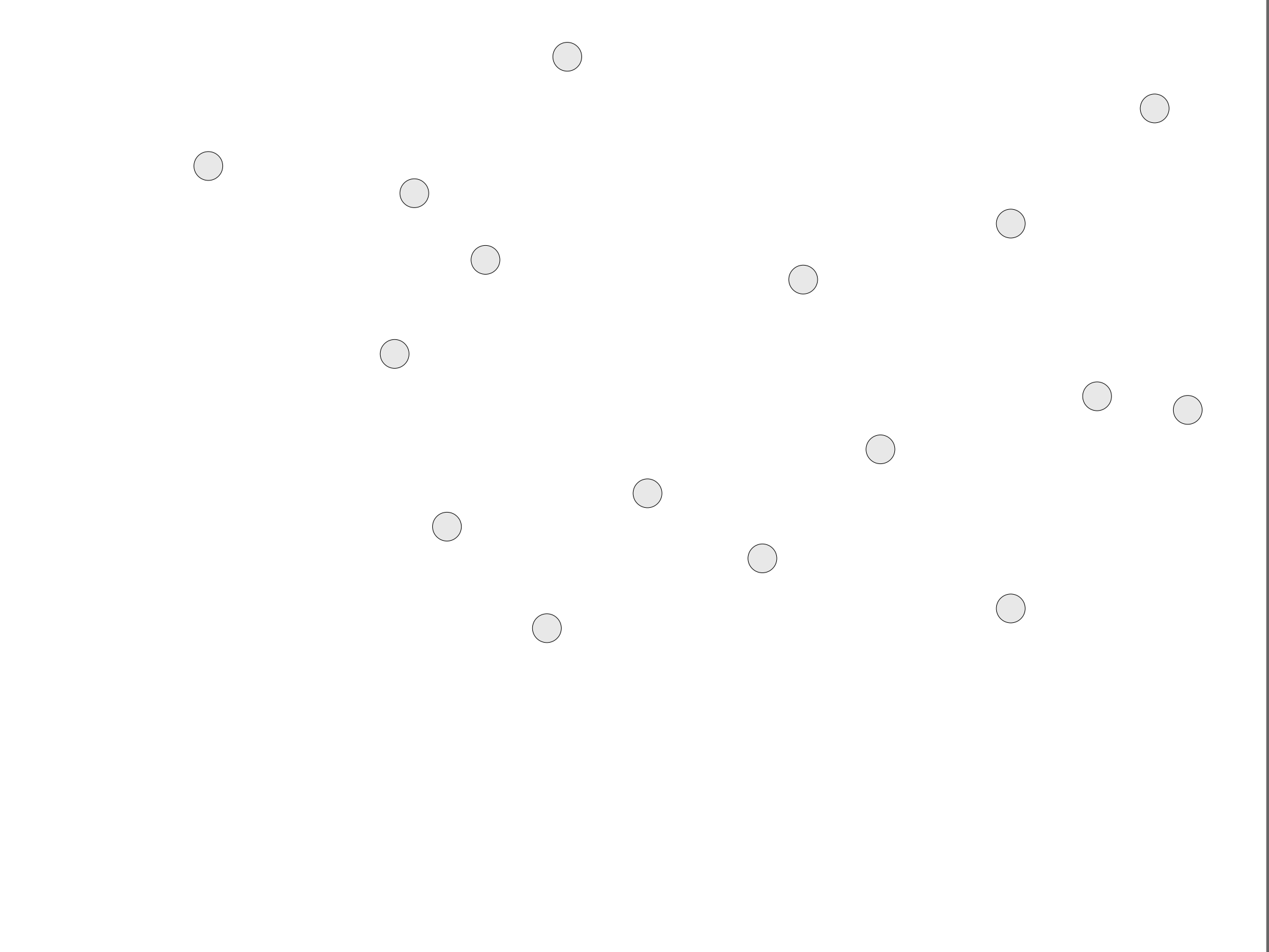
- The recurrence on our preprocessing time is

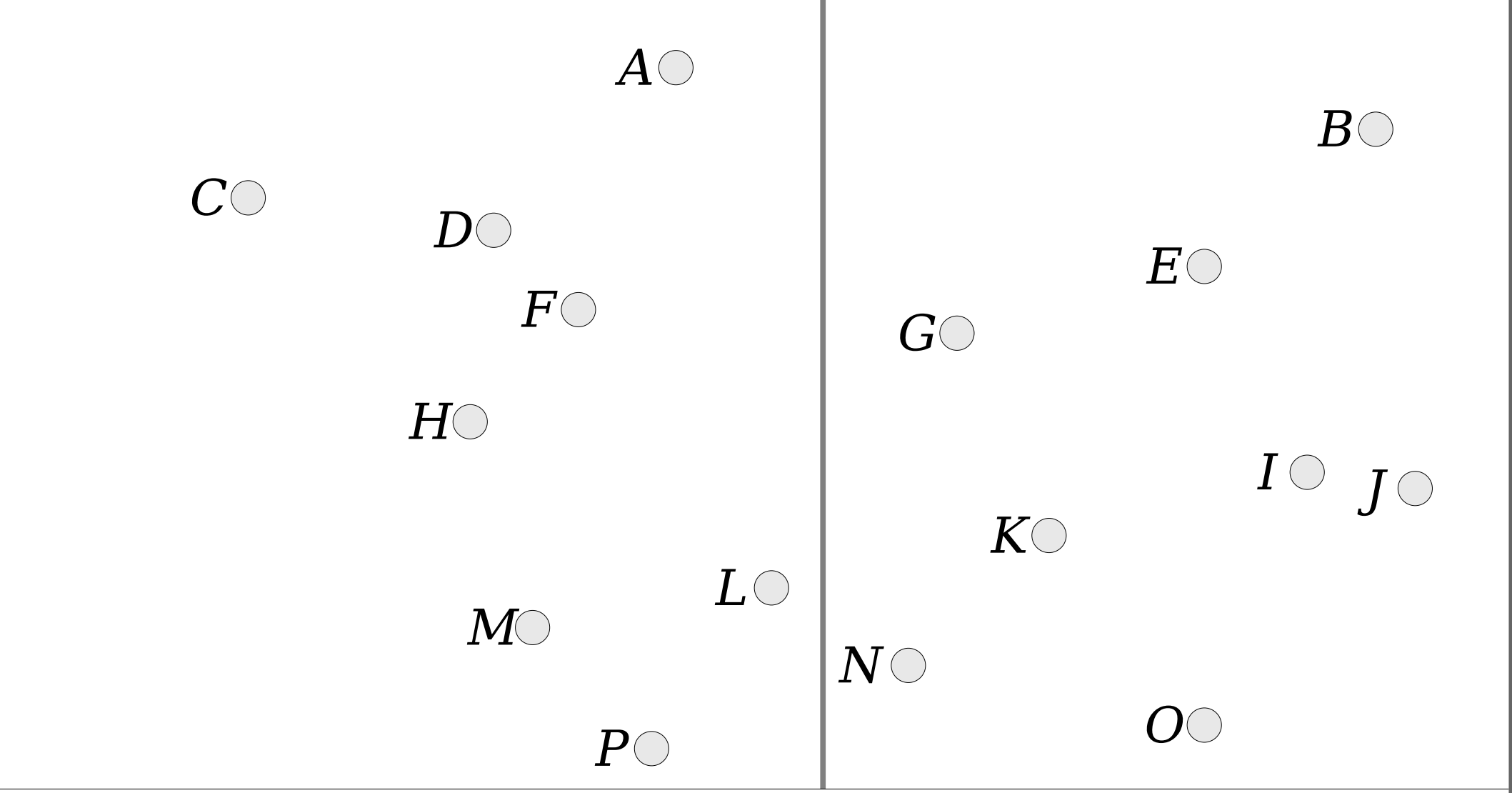
$$T(n) = 2T(n / 2) + O(n \log n)$$

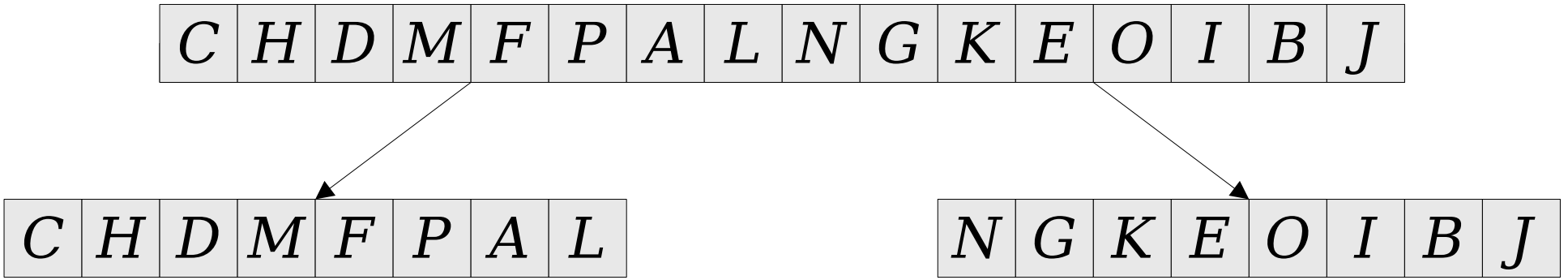
  because each tree node must

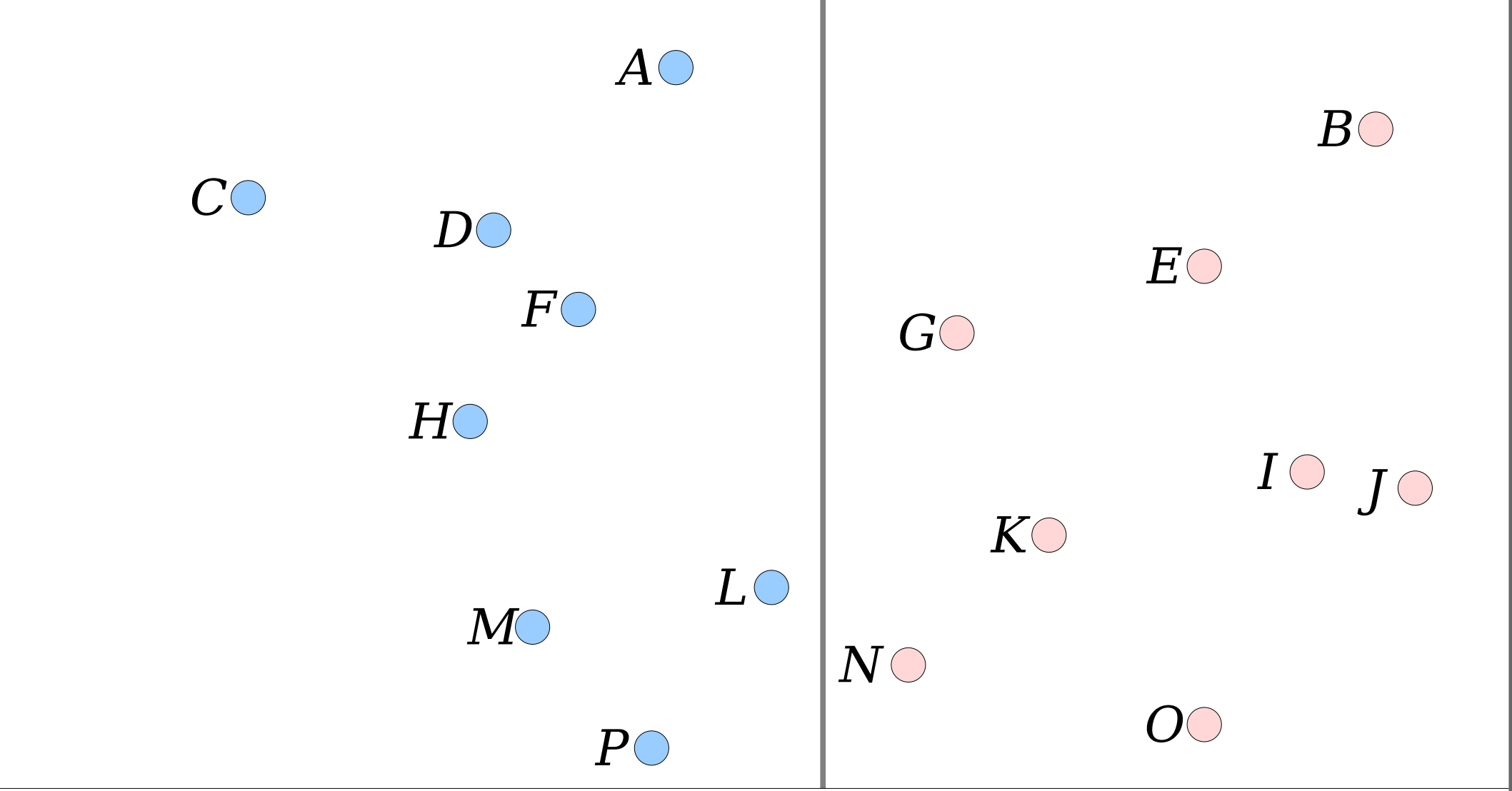
  - sort all its points by their y coordinate, and
  - recursively construct its children.

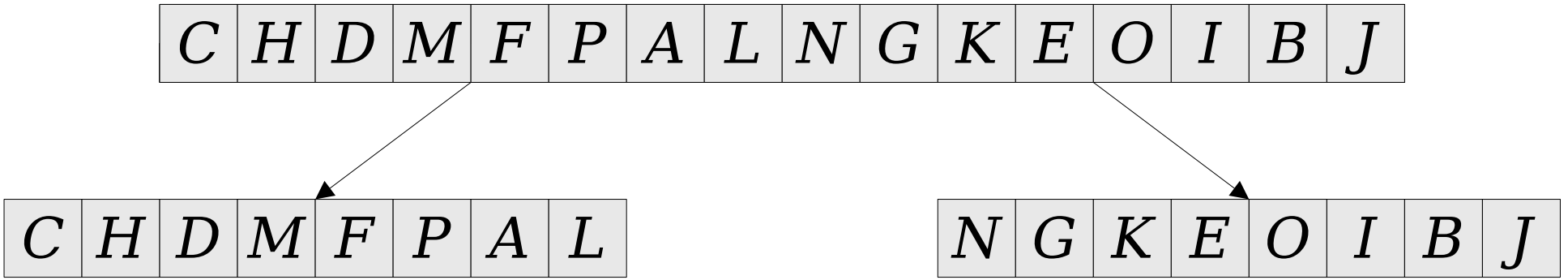
- We have to construct the child nodes, since that's essential to the tree shape.
- ***Question:*** Can we get rid of the sorting step?

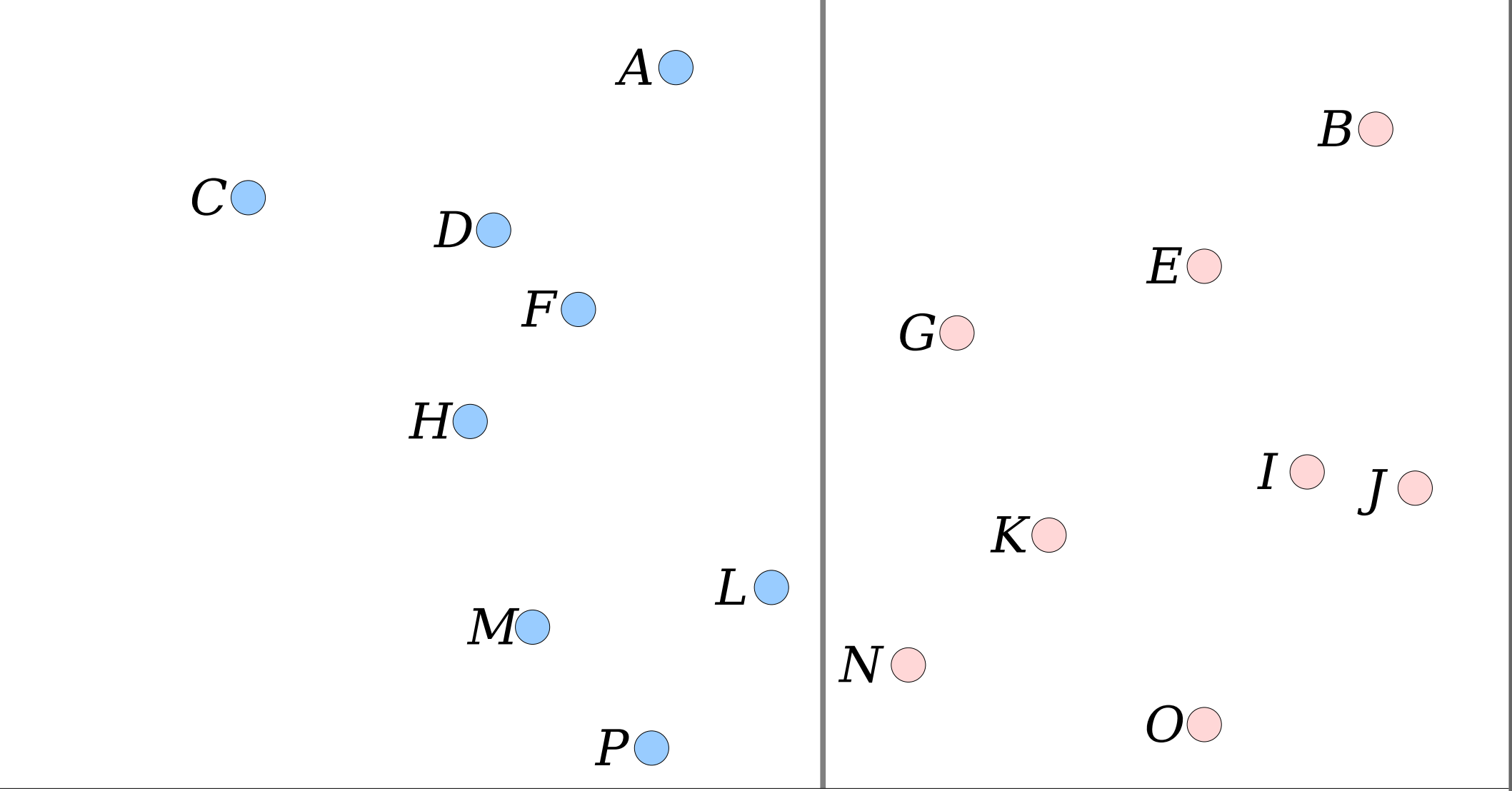


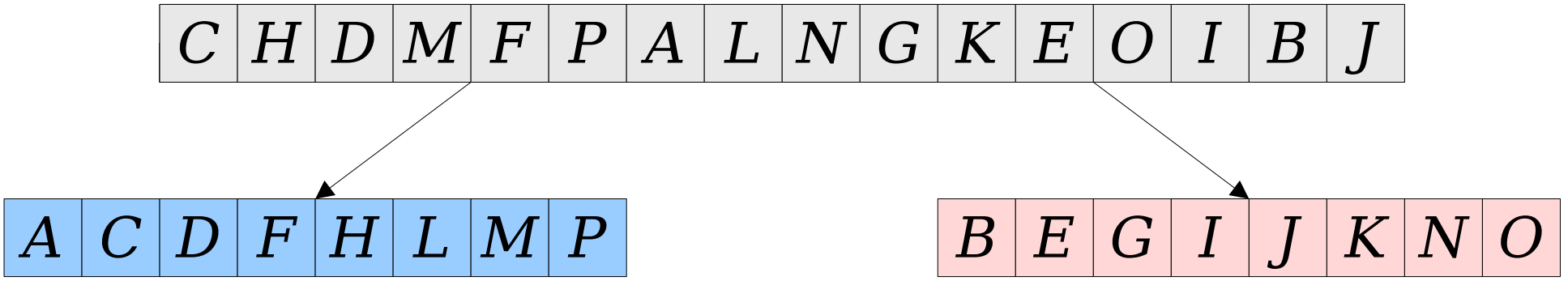

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

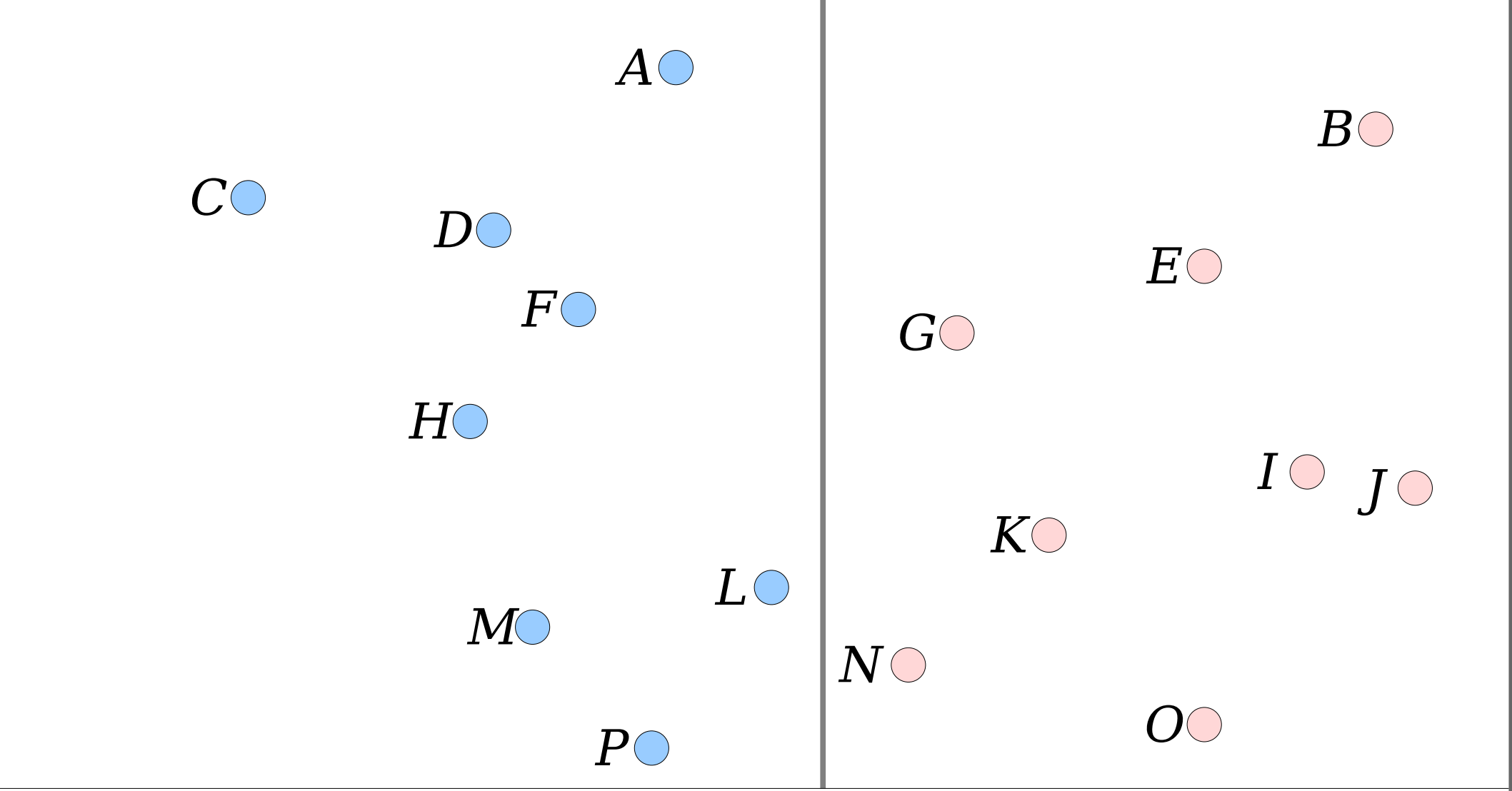

C H D M F P A L N G K E O I B J
C H D M F P A L
N G K E O I B J

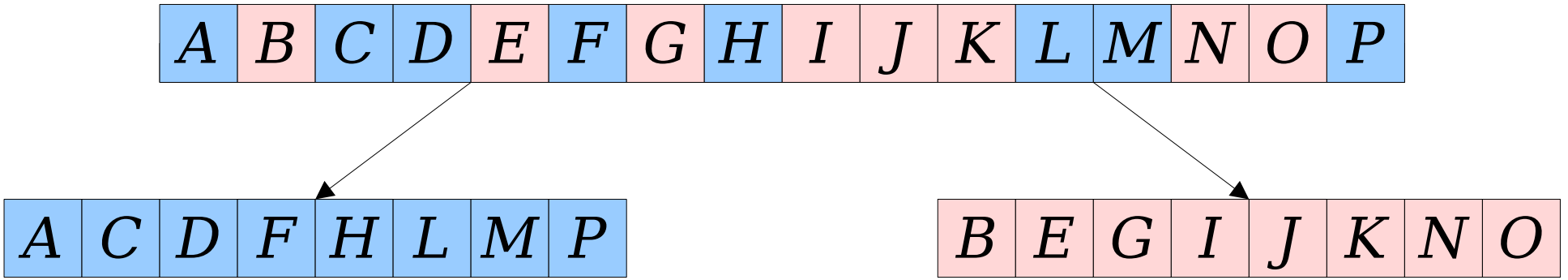

A
C
D
F
H
L
M
P
B
E
G
I
J
K
N
O


C H D M F P A L N G K E O I B J
C H D M F P A L
N G K E O I B J


A
C
D
F
H
L
M
P
B
E
G
I
J
K
N
O


C H D M F P A L N G K E O I B J
A C D F H L M P
B E G I J K N O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

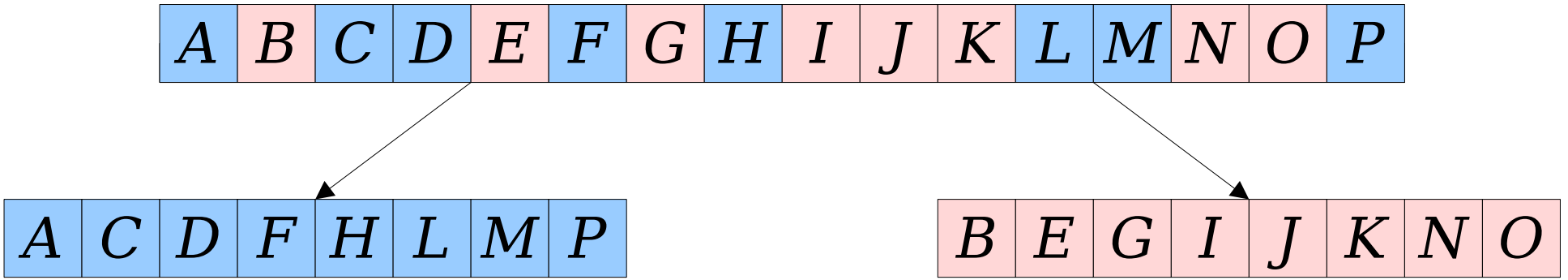

A B C D E F G H I J K L M N O P
A C D F H L M P
B E G I J K N O

Speeding Up Construction

- ***Idea:*** Instead of sorting points by y coordinates at each node, recursively merge the sorted arrays from the two children.
- This reduces the per-node cost of construction to

$$T(n) = 2T(n / 2) + O(n),$$

which solves to **$O(n \log n)$**.



The Story So Far

- By harnessing correlations across nodes in the range tree, we were able to reduce the preprocessing time.
- ***Question:*** Can we take advantage of similar insights to knock the query time down a bit more, too?

	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$
Range Tree	$O(n \log n)$	$O(\log^2 n + k)$	$O(n \log n)$

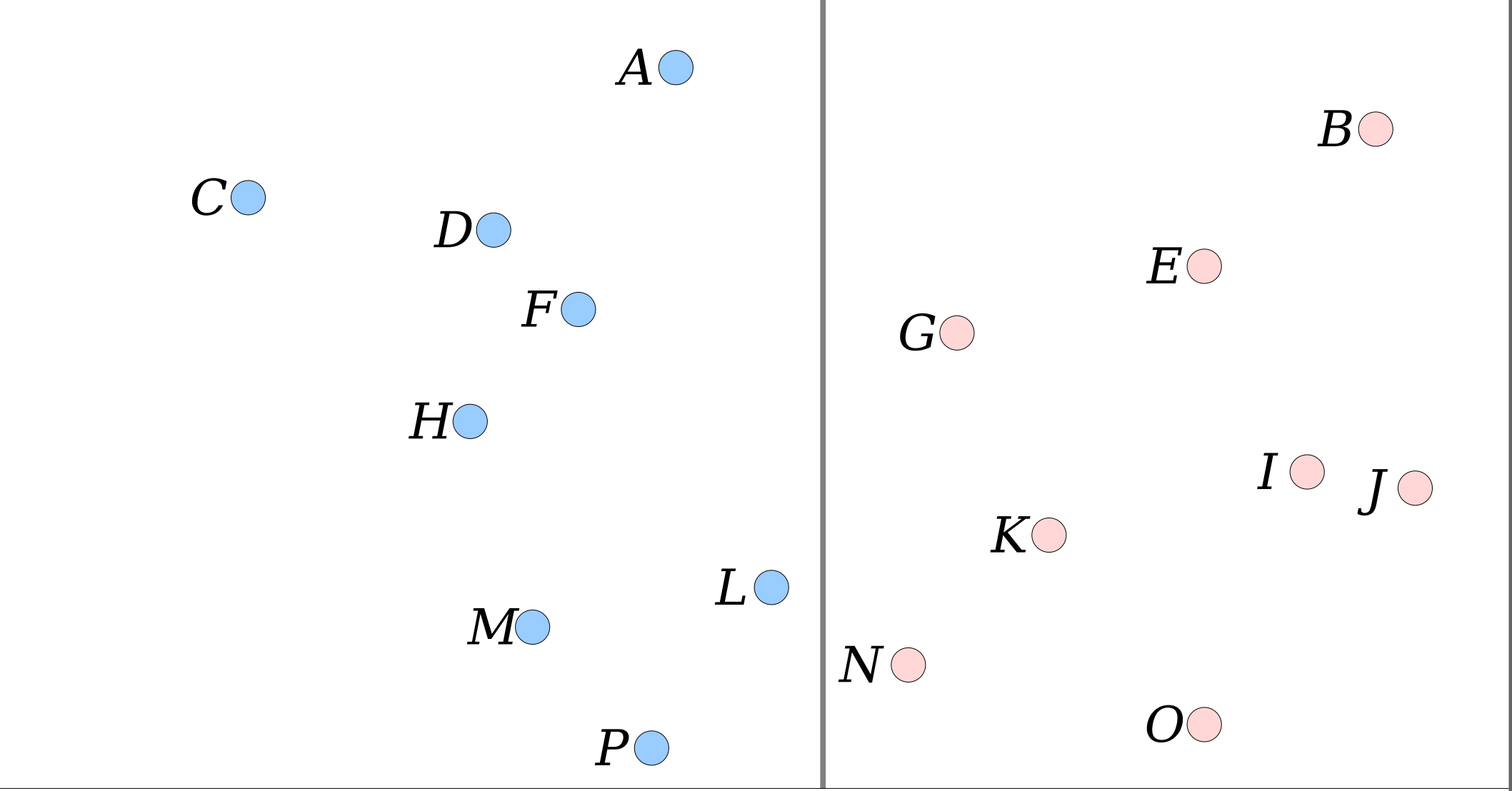
The Story So Far

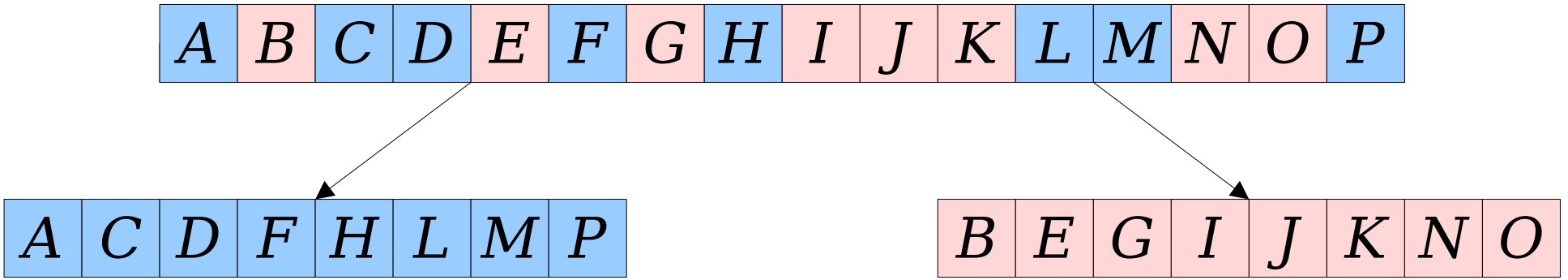
- By harnessing correlations across nodes in the range tree, we were able to reduce the preprocessing time.
- ***Question:*** Can we take advantage of similar insights to knock the query time down a bit more, too?

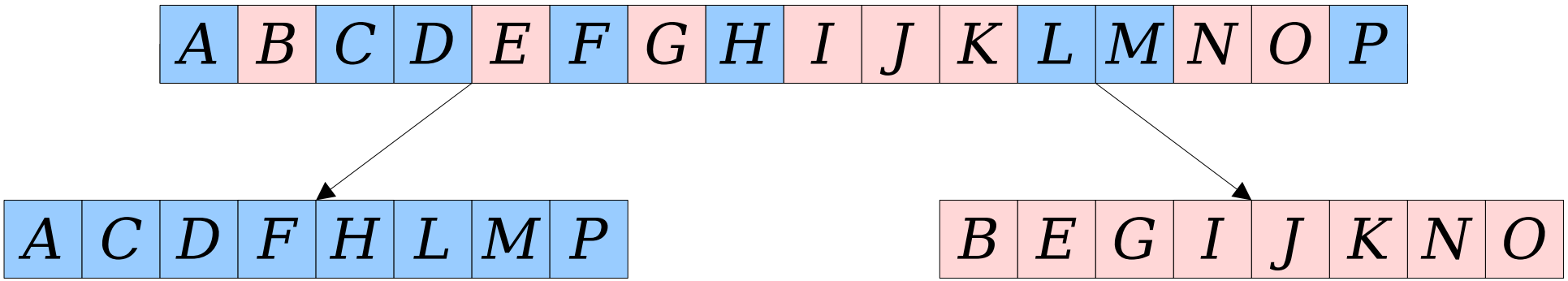
	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$
Range Tree	$O(n \log n)$	$O(\log^2 n + k)$	$O(n \log n)$

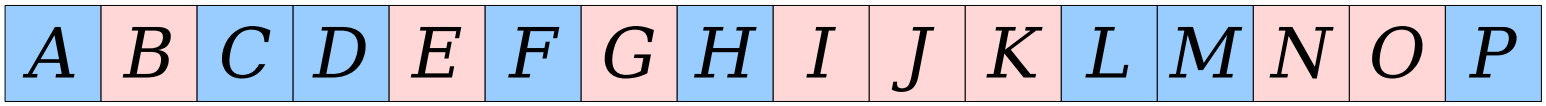
Speeding up Queries

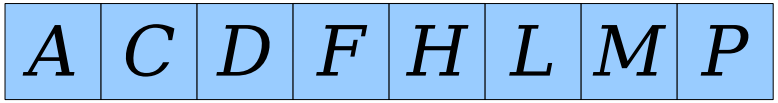
- Our current query time is $O(\log^2 n + k)$. We can't improve the k term. Can we reduce that $\log^2 n$ part?
- Refresher: The $\log^2 n$ term arises because our query
  - looks at $O(\log n)$ blocks, and
  - does a binary search at each block to find all the matching points.
- ***Question:*** Can we make those binary searches run faster?

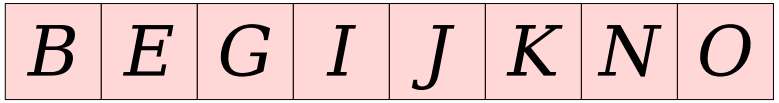

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P


A B C D E F G H I J K L M N O P
A C D F H L M P
B E G I J K N O


A B C D E F G H I J K L M N O P
A C D F H L M P
B E G I J K N O

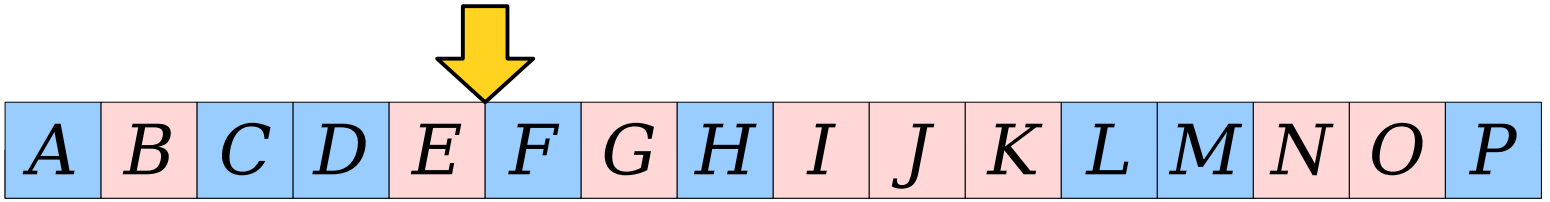

A B C D E F G H I J K L M N O P

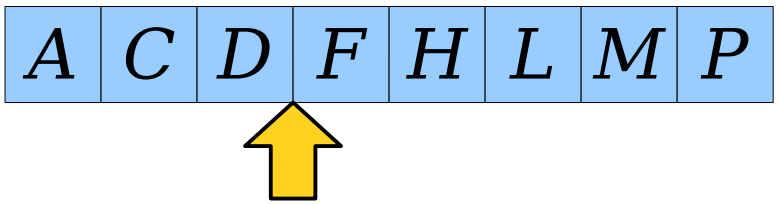

A C D F H L M P

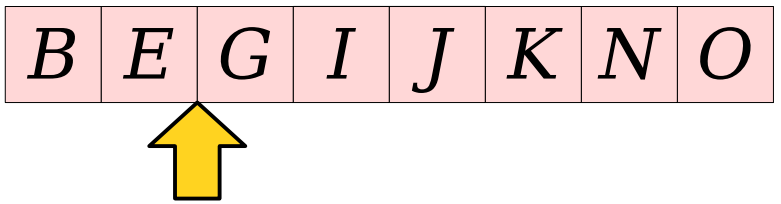

B E G I J K N O

Suppose we do a binary search in the top node for the first point whose *y* coordinate is in the range.

If we did this same binary search in the two children, where would those searches end?

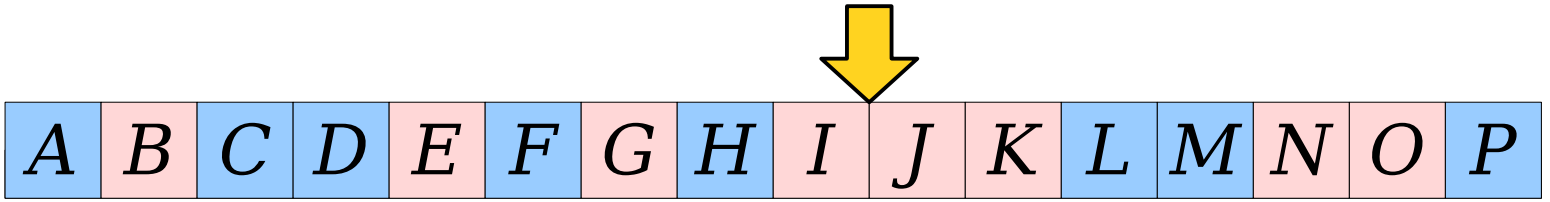


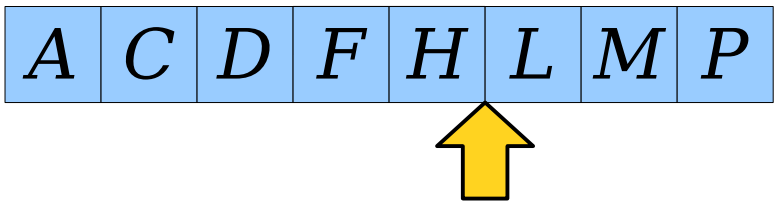


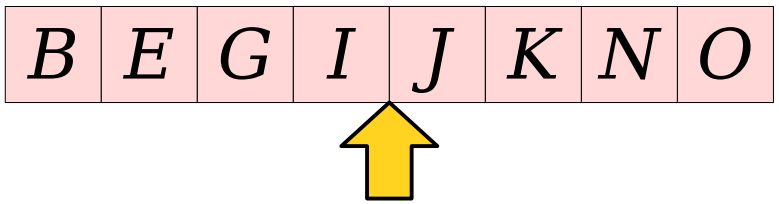


Suppose we do a binary search in the top node for the first point whose *y* coordinate is in the range.

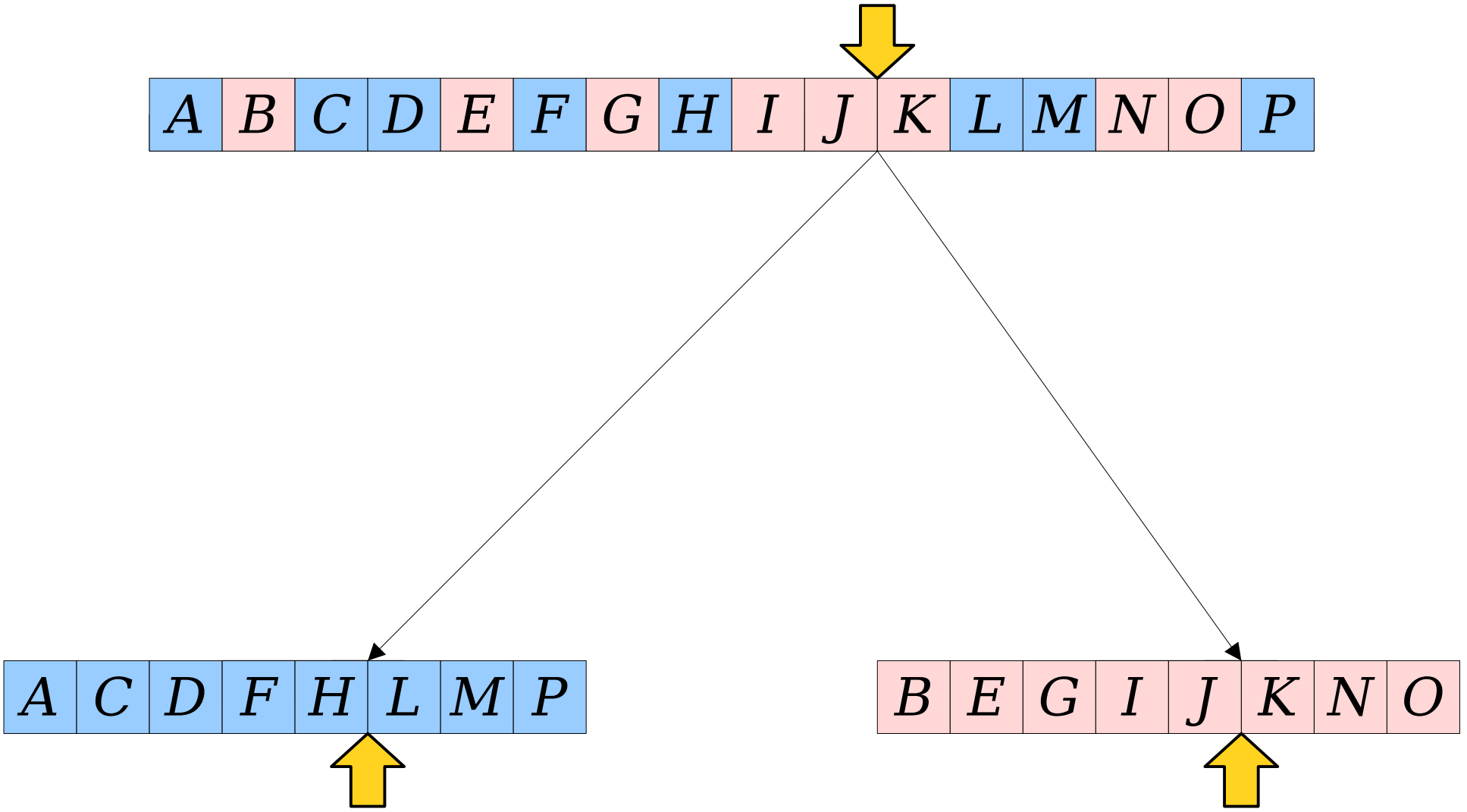
If we did this same binary search in the two children, where would those searches end?





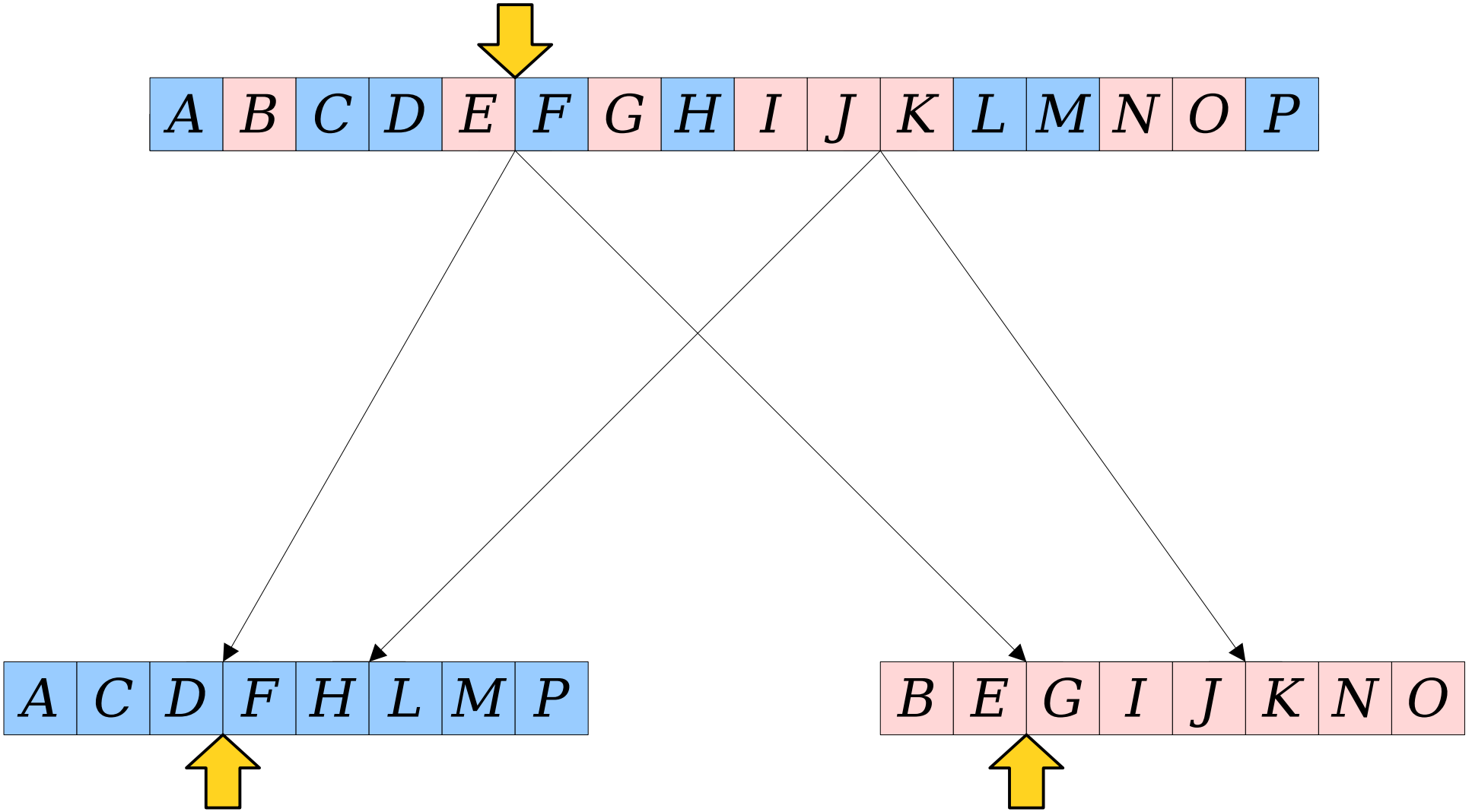


Idea: Precompute, in the top node, where each binary search would end in its children.



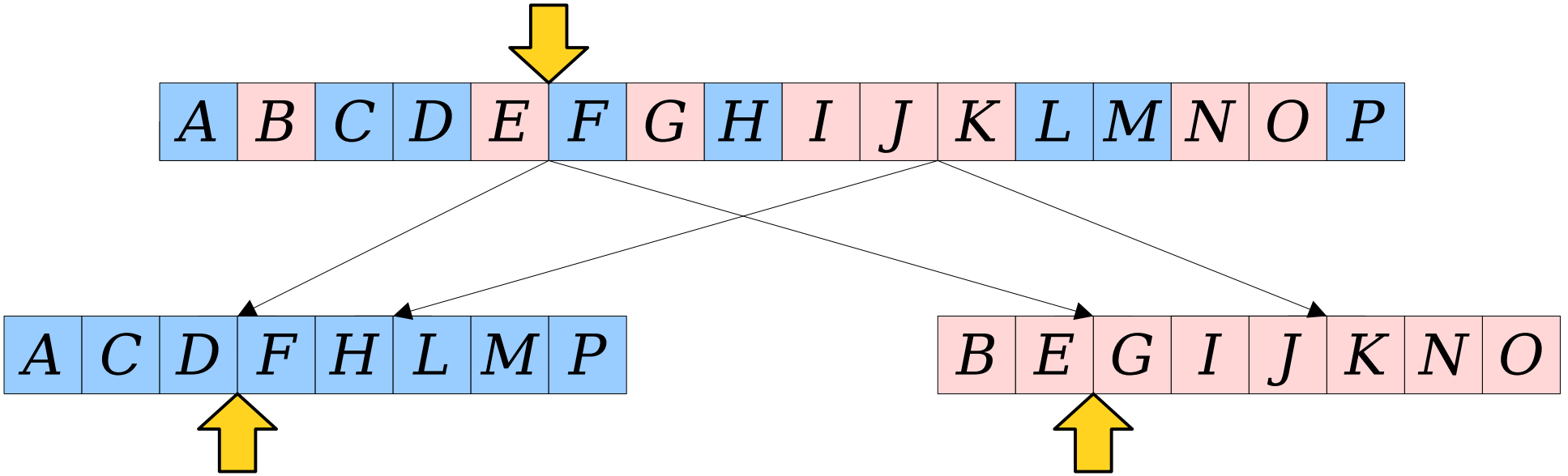
Idea: Precompute, in the top node, where each binary search would end in its children.

Claim: This can be done when merging the two child arrays together, requiring $O(n)$ total work.

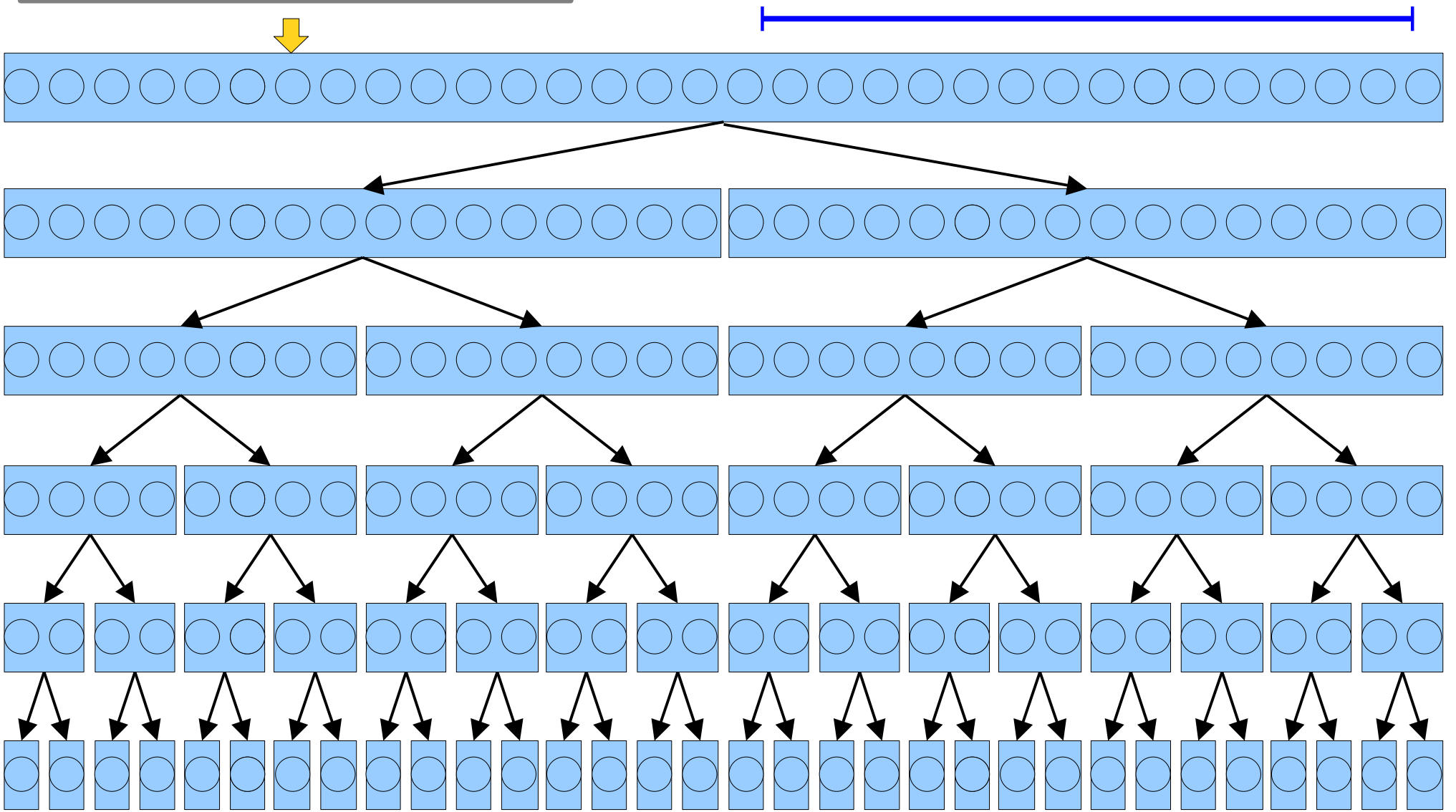


Geometric Cascading

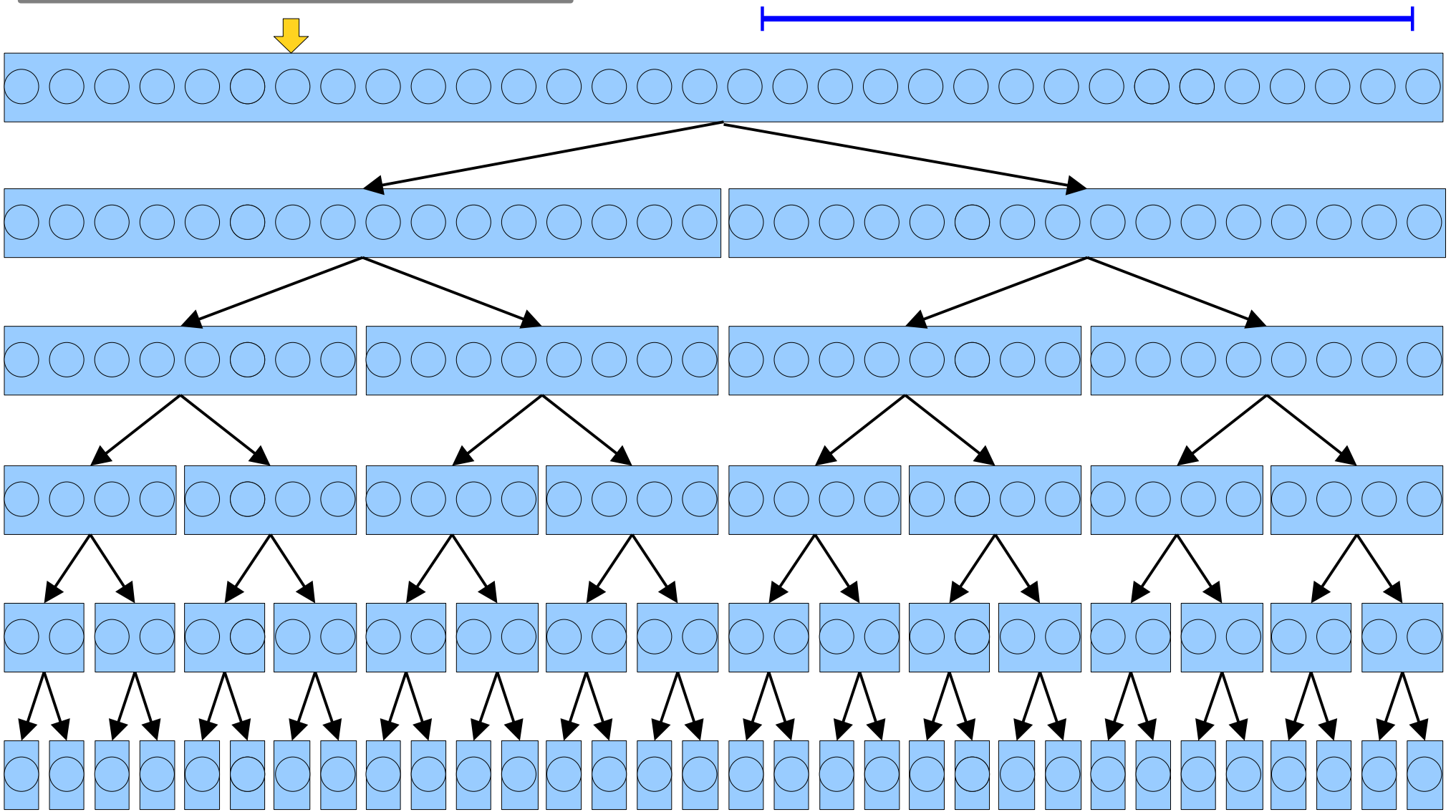
- These pointers from an array to its children are called ***downpointers***. The technique is called ***geometric cascading***.
- Intuitively, downpointers speed up our binary searches by using the fact that the arrays have similar structures.



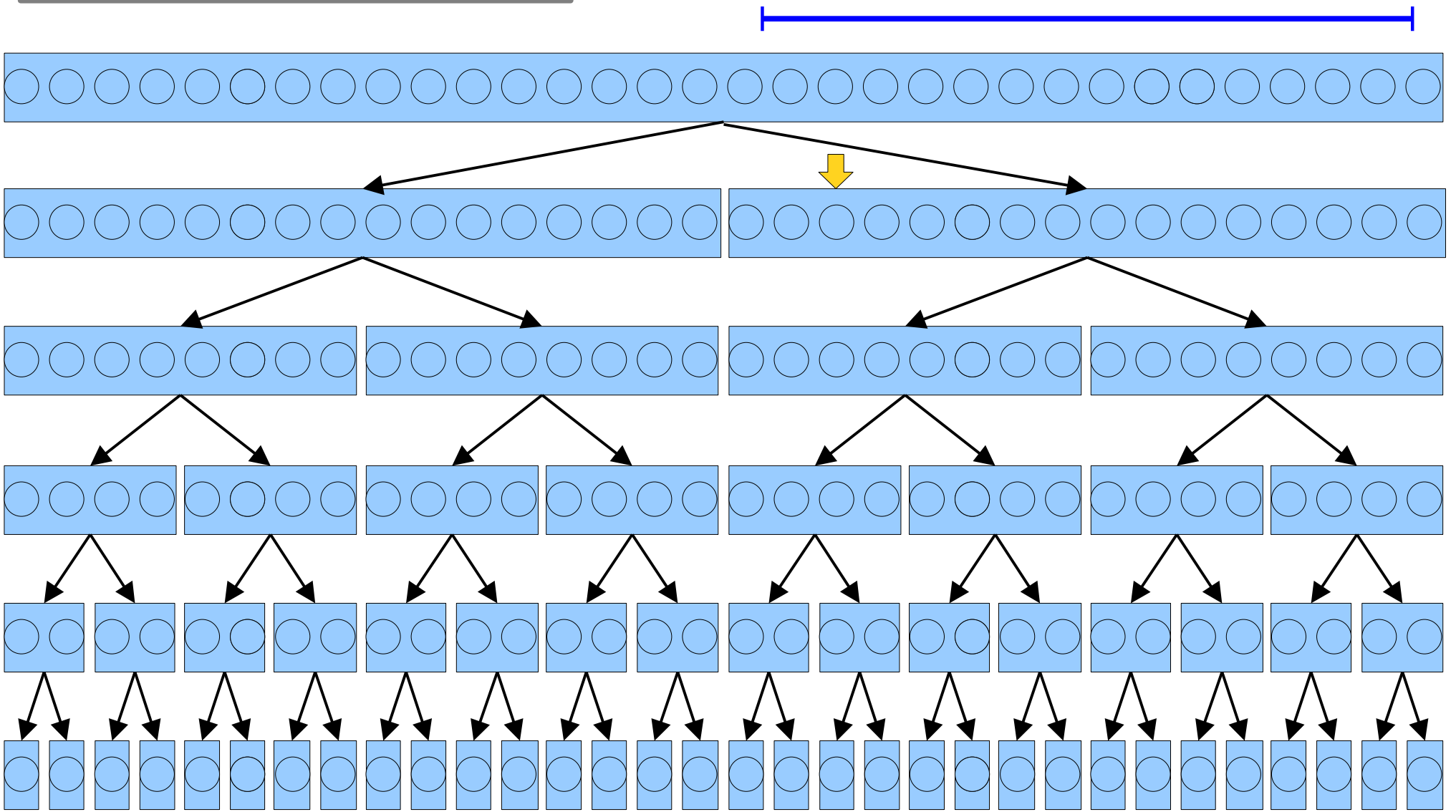
Begin with a binary search to find the first point in the y range.

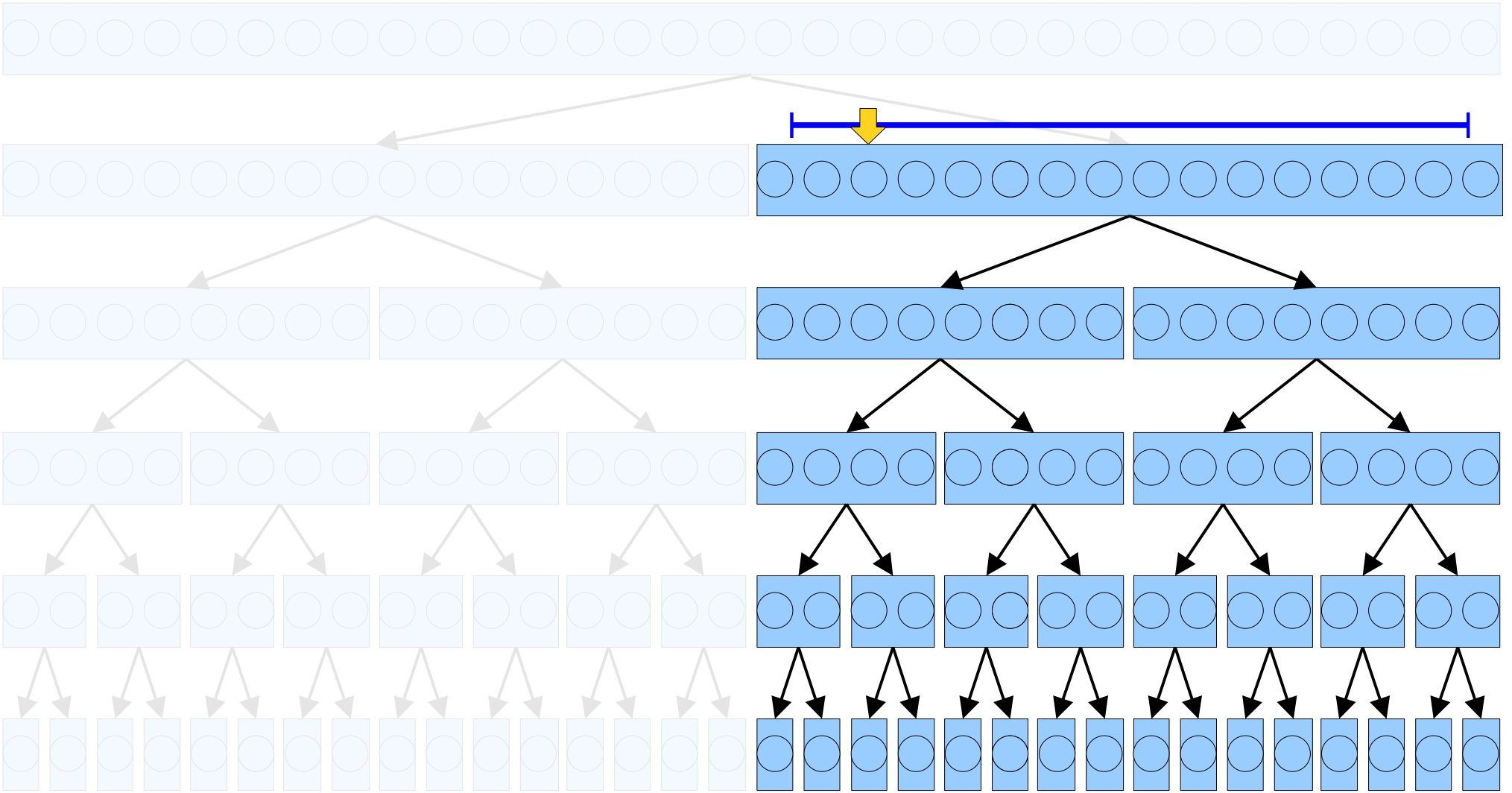


We're moving to the right, so follow the ***downpointer*** we precomputed as we do.

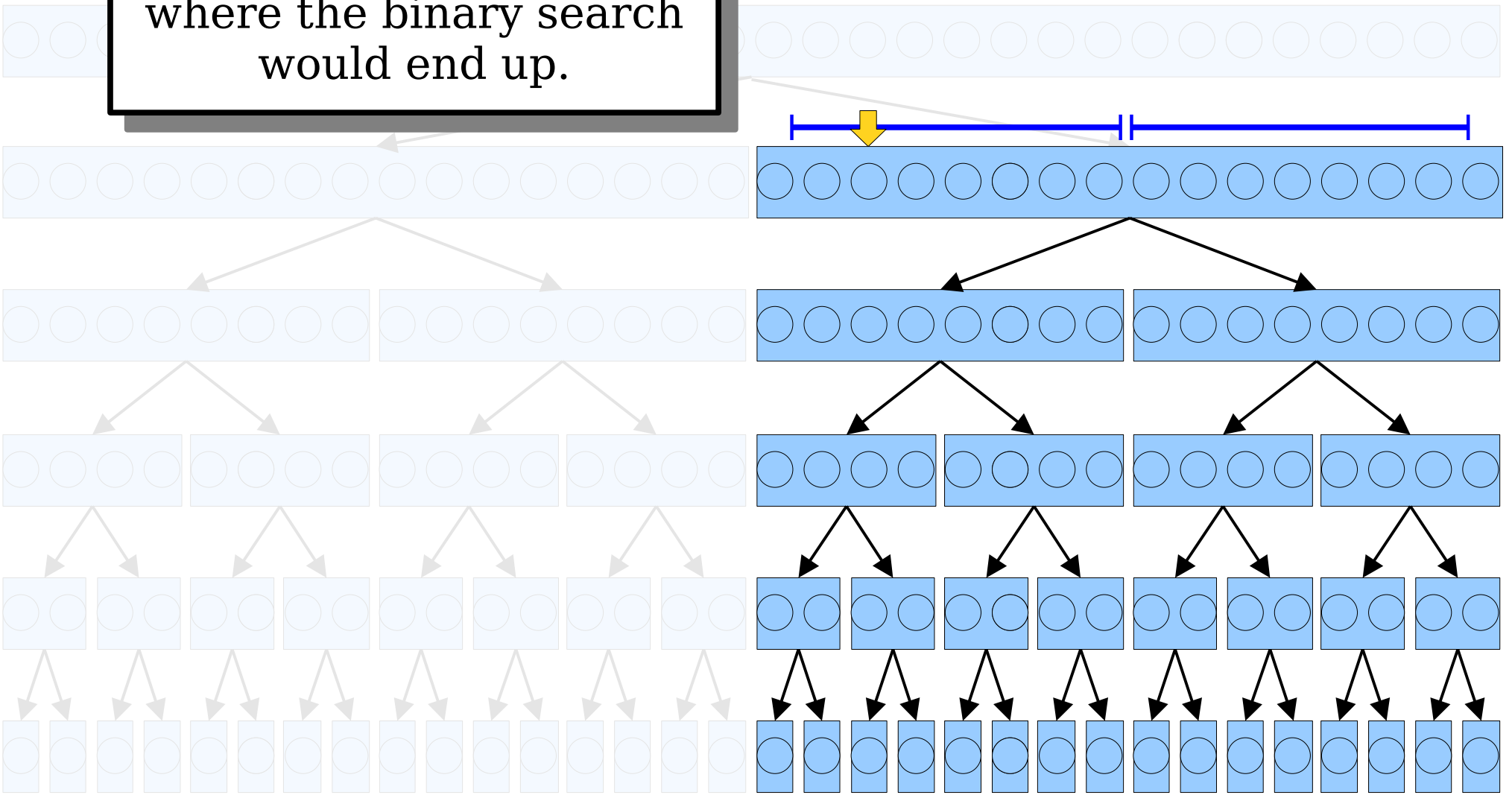


We're moving to the right, so follow the ***downpointer*** we precomputed as we do.

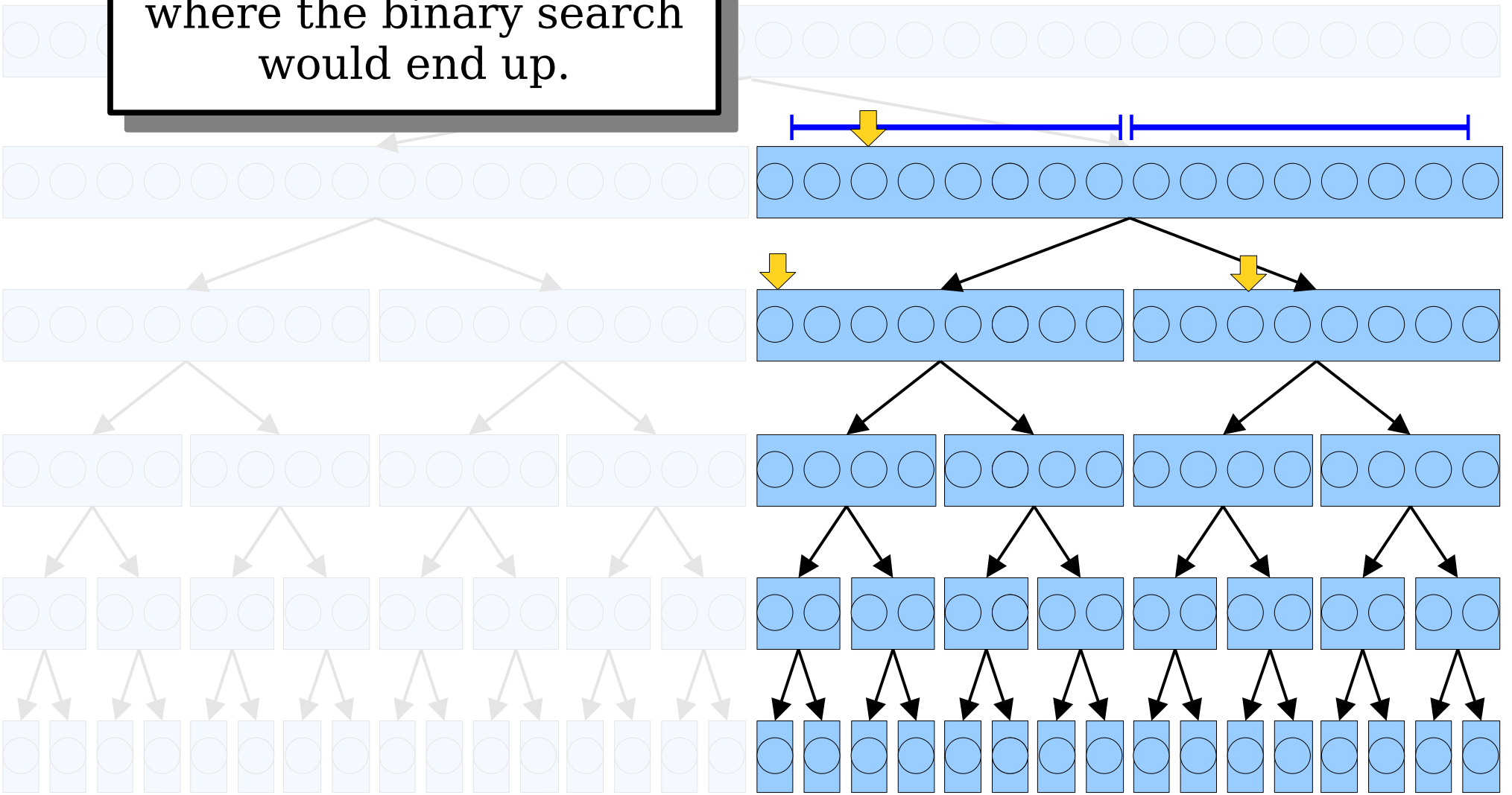


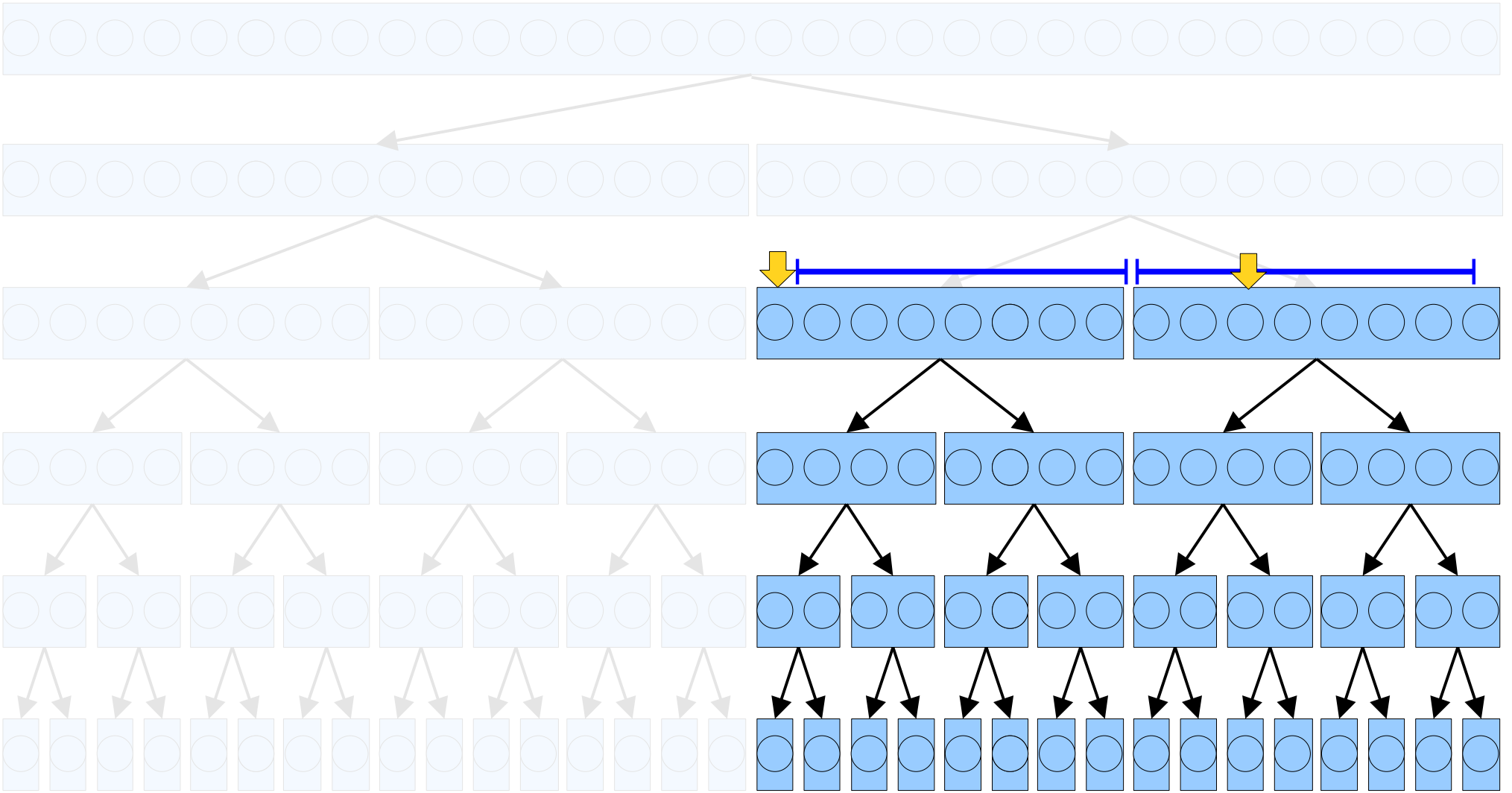


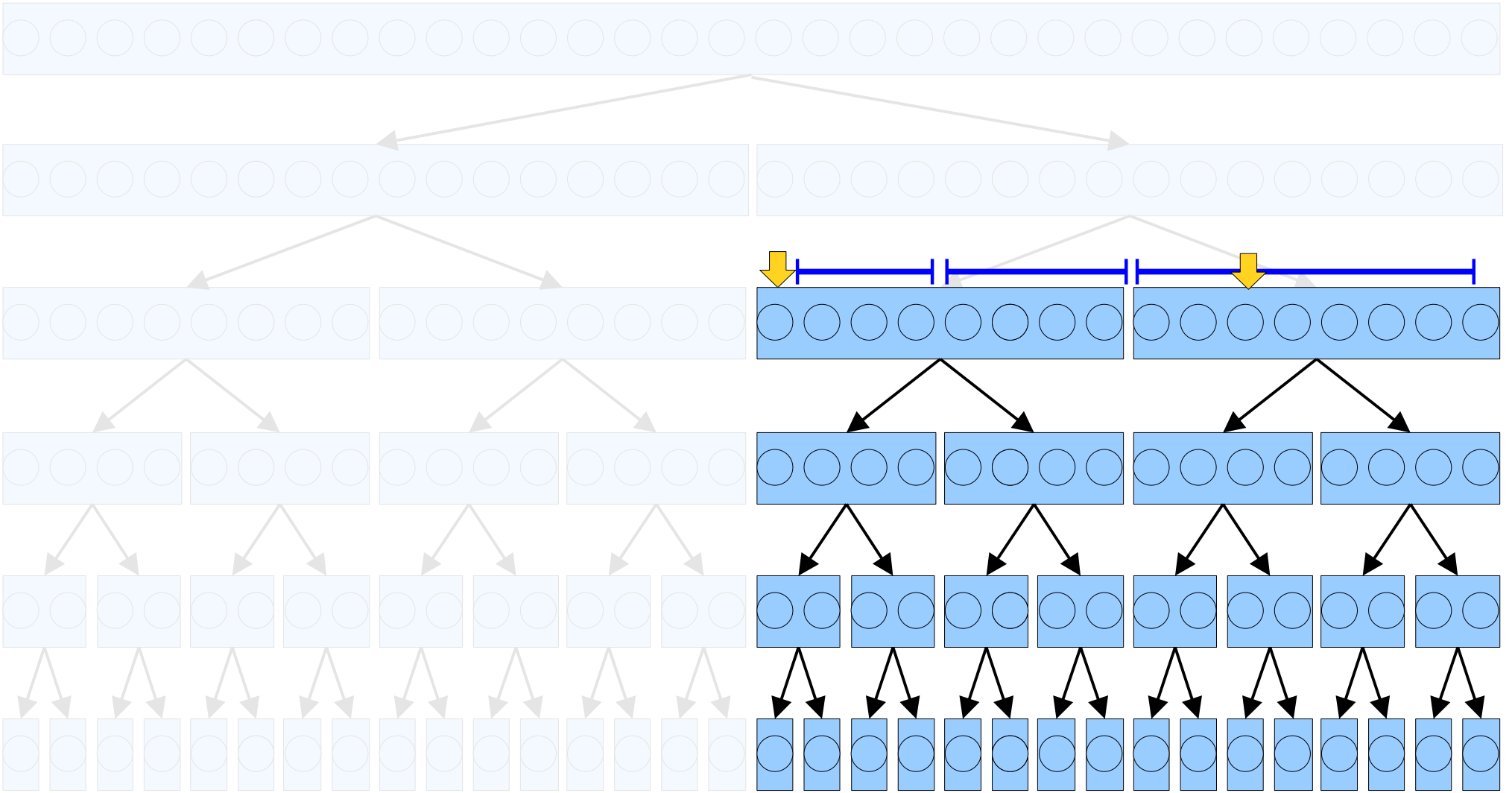
Keep following downpointers to track where the binary search would end up.

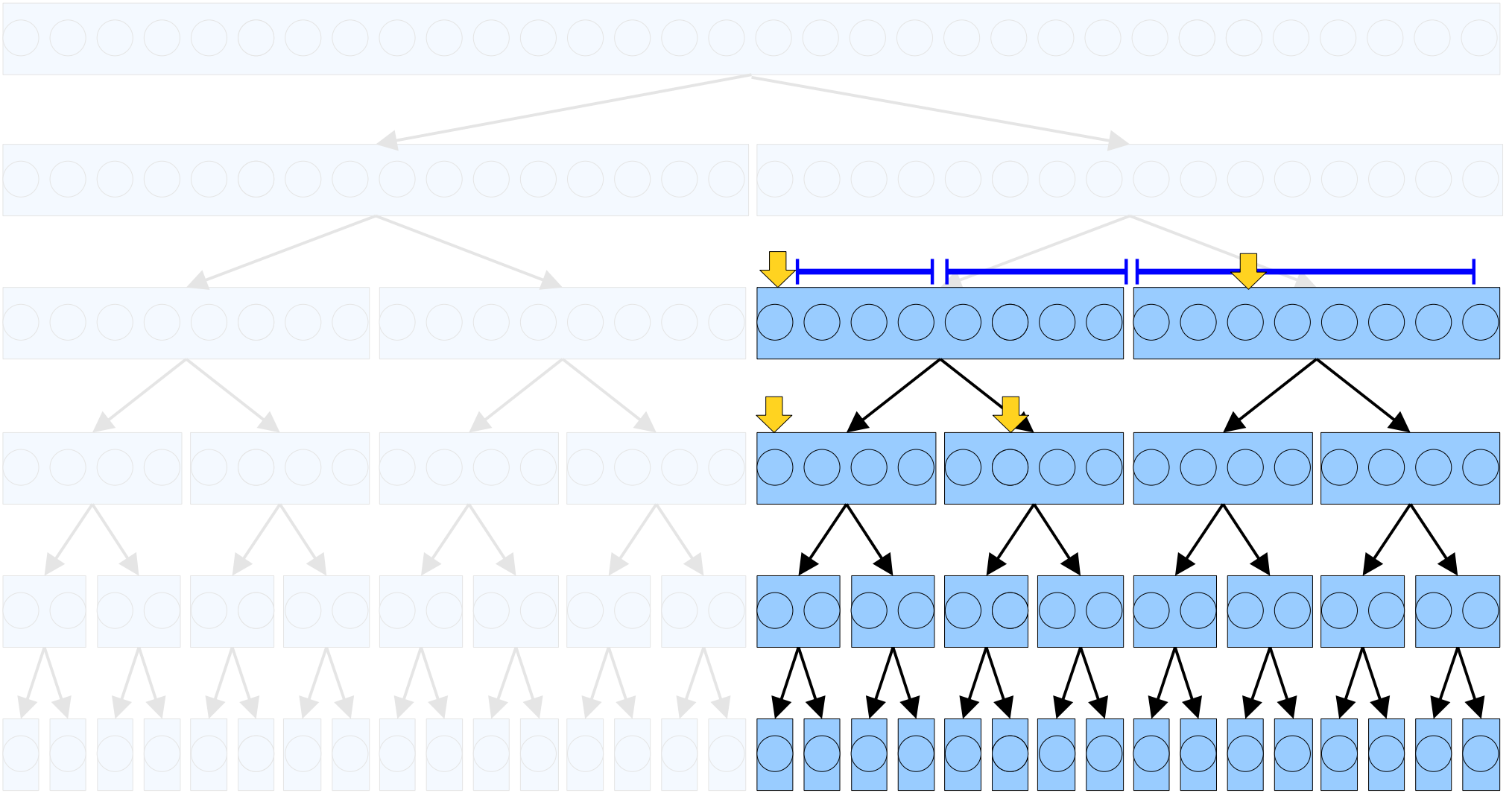


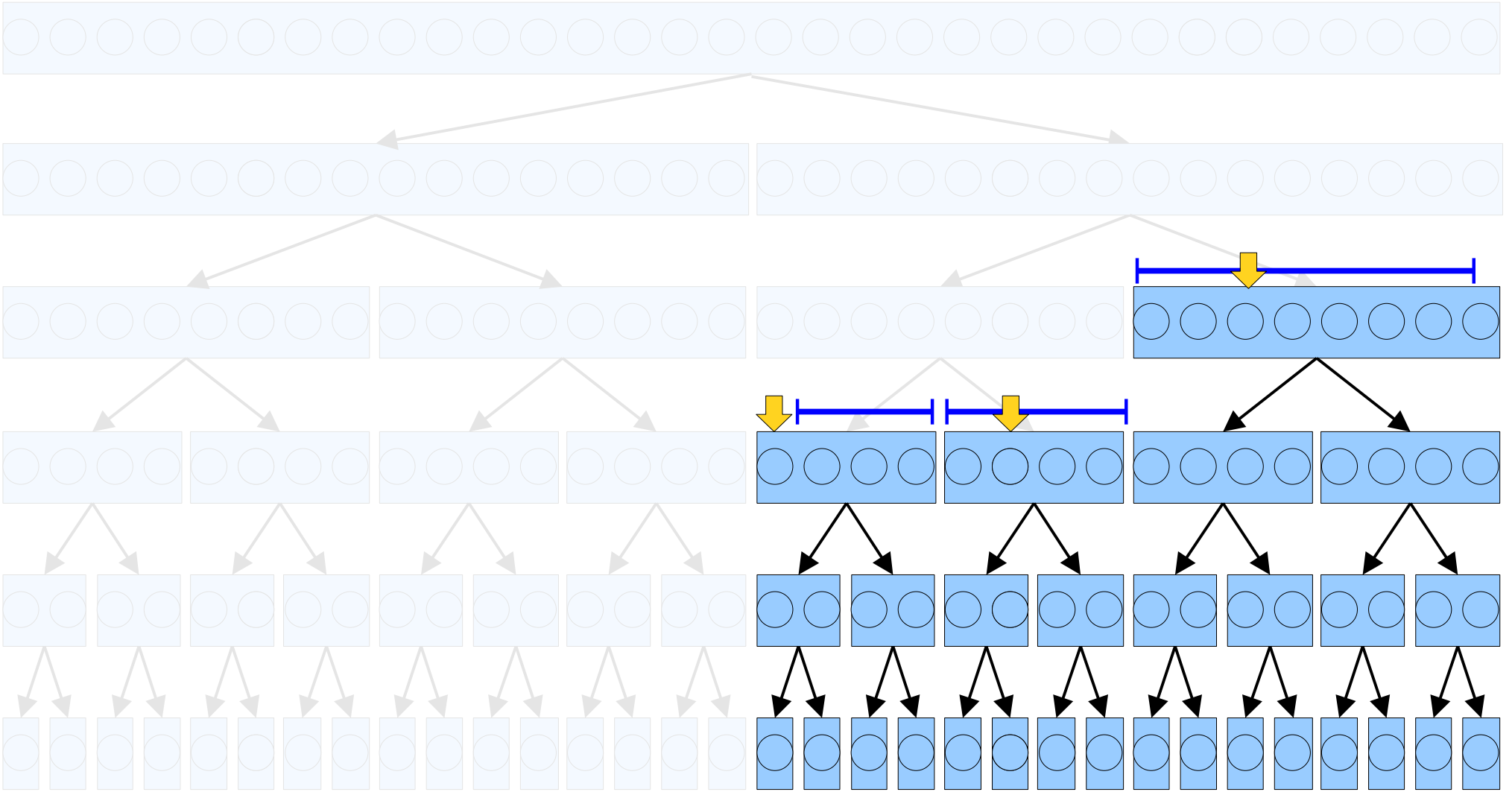
Keep following downpointers to track where the binary search would end up.

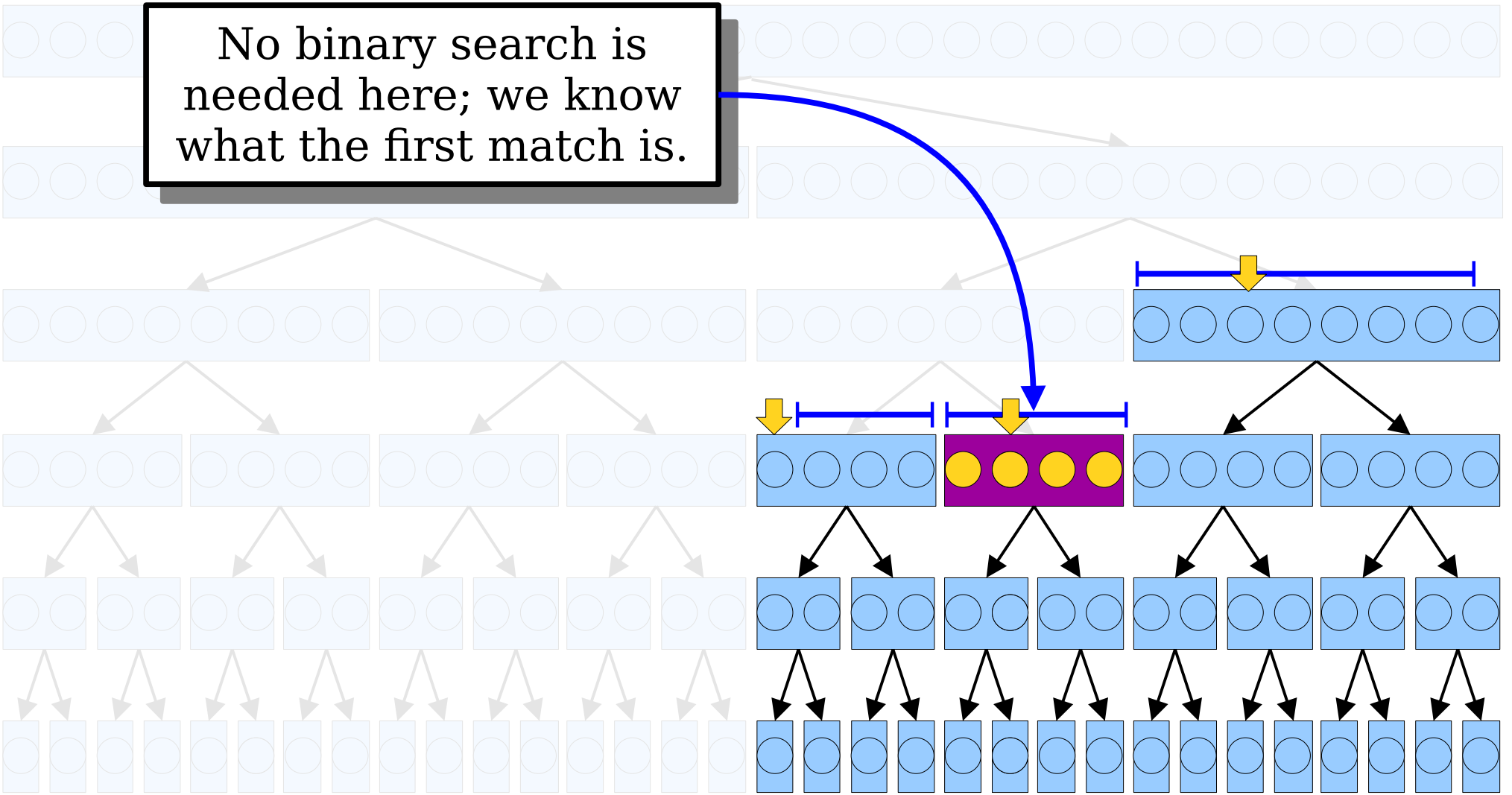


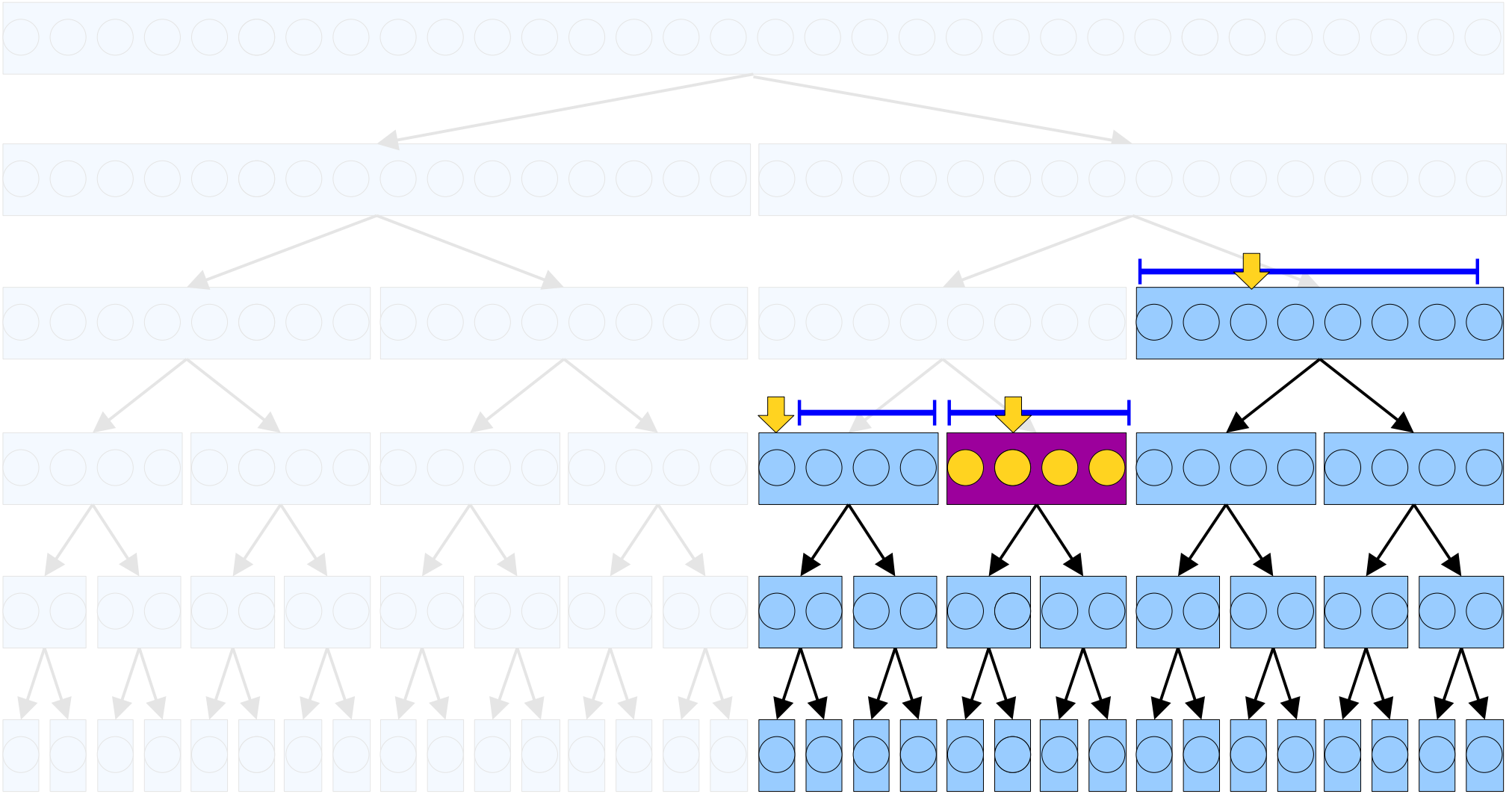


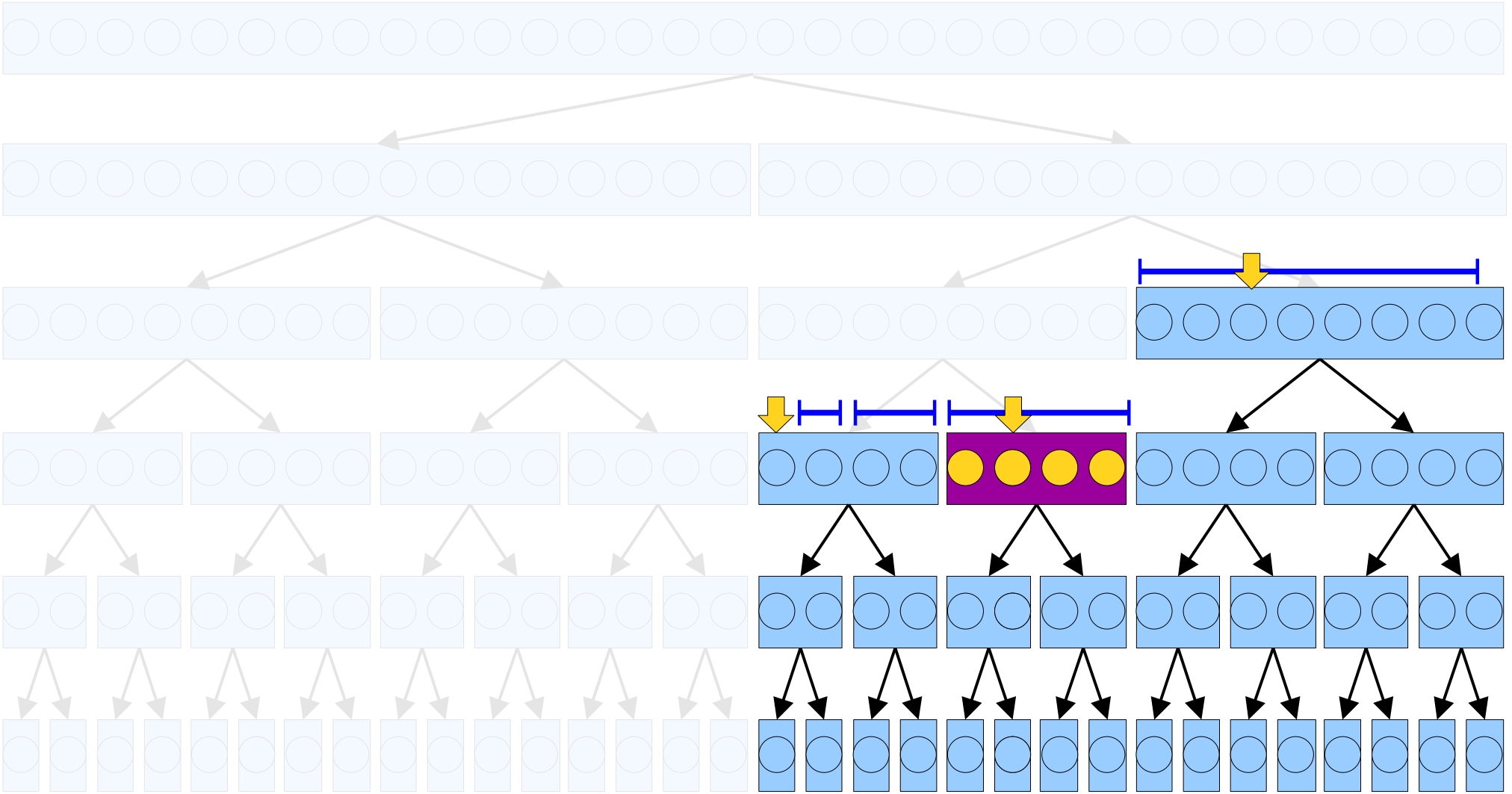


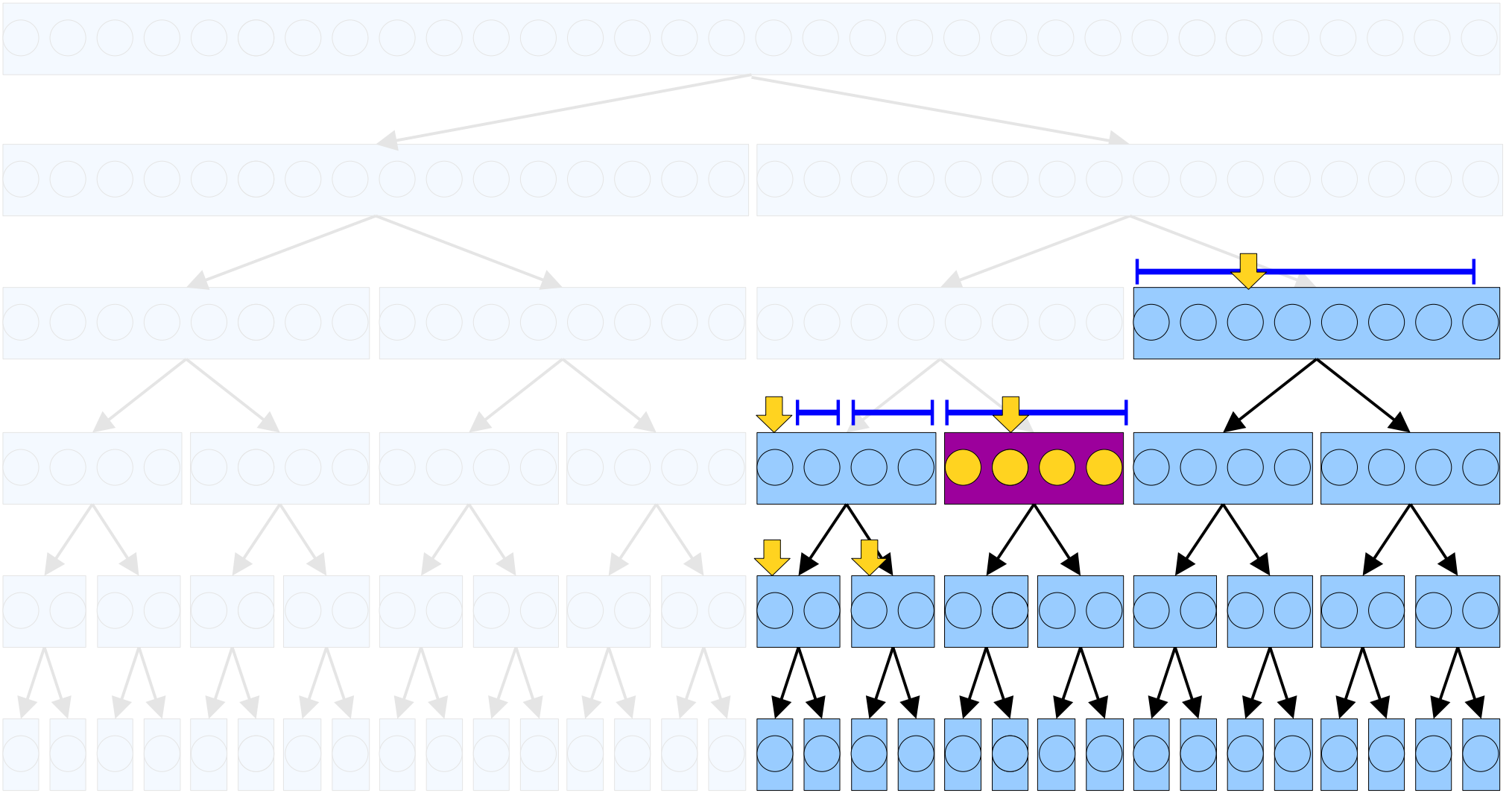


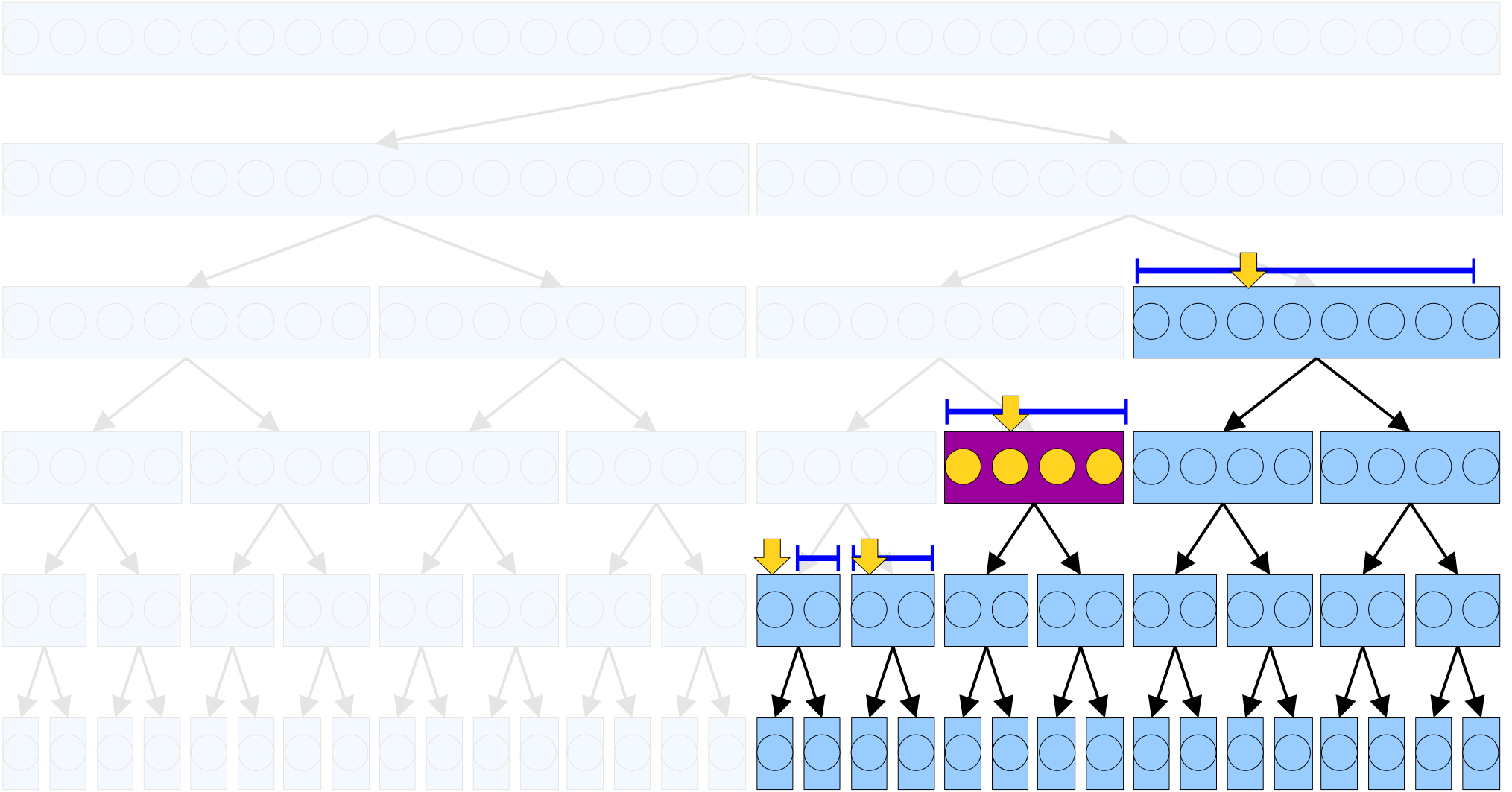


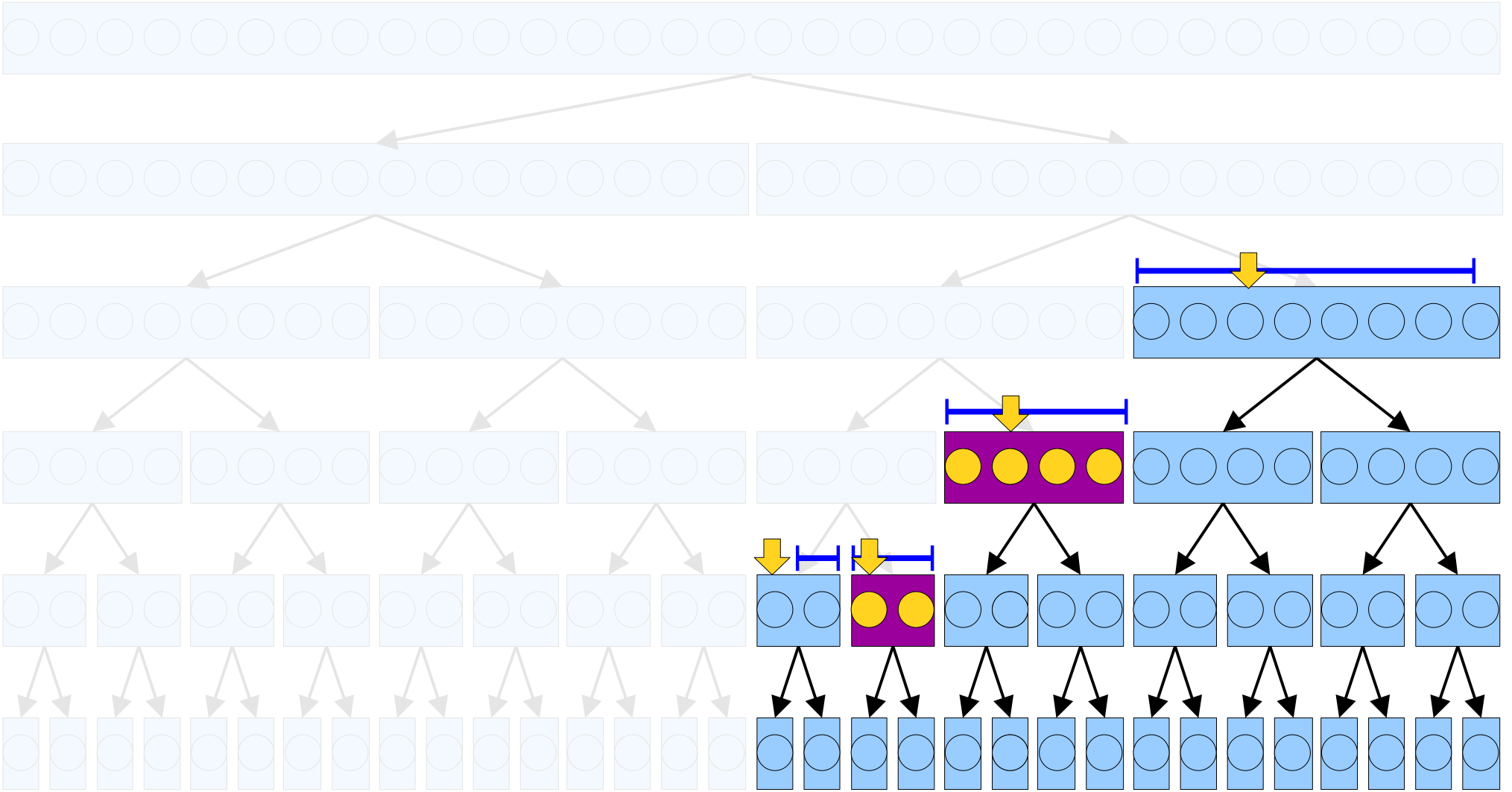

No binary search is needed here; we know what the first match is.

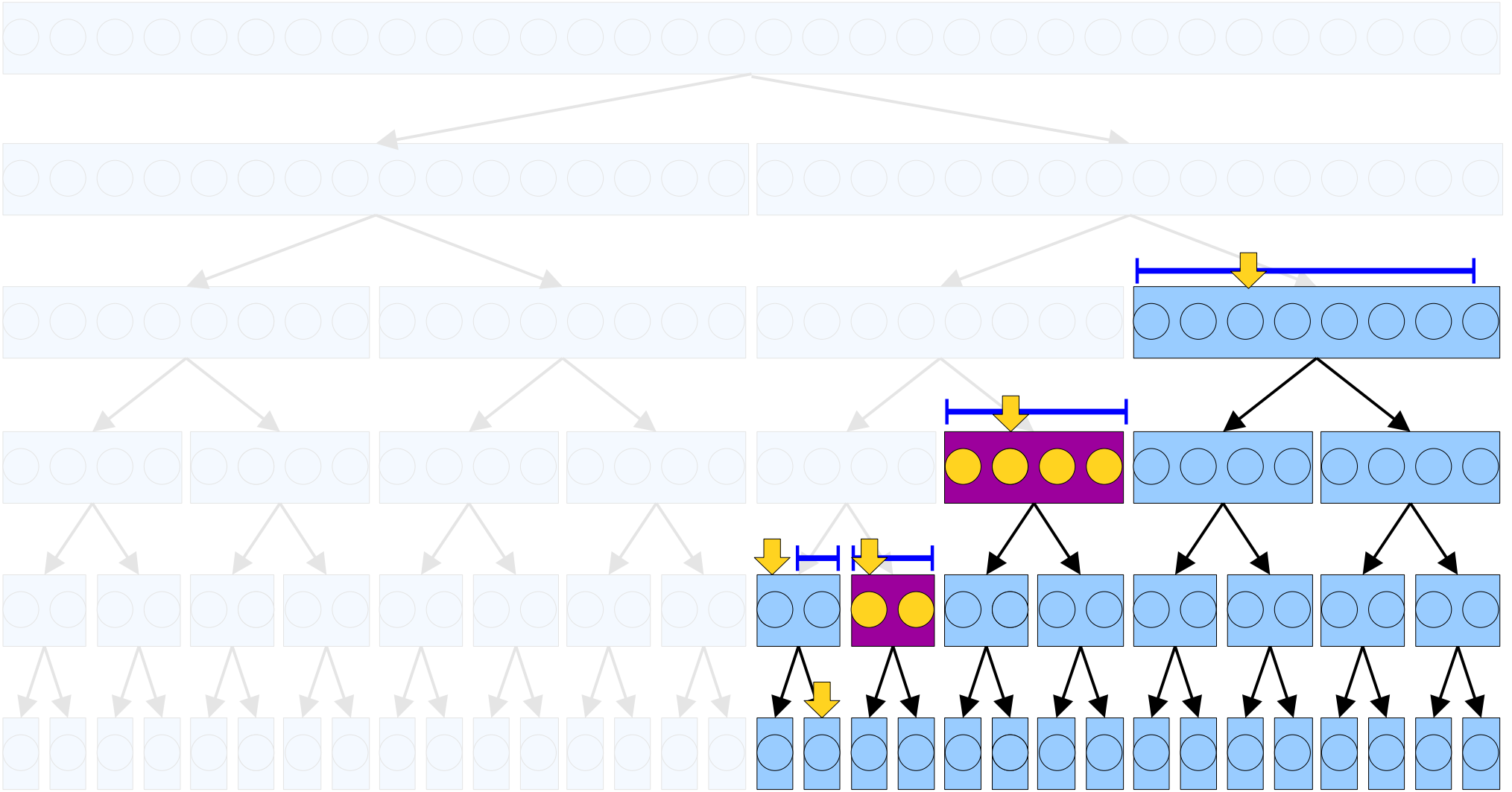


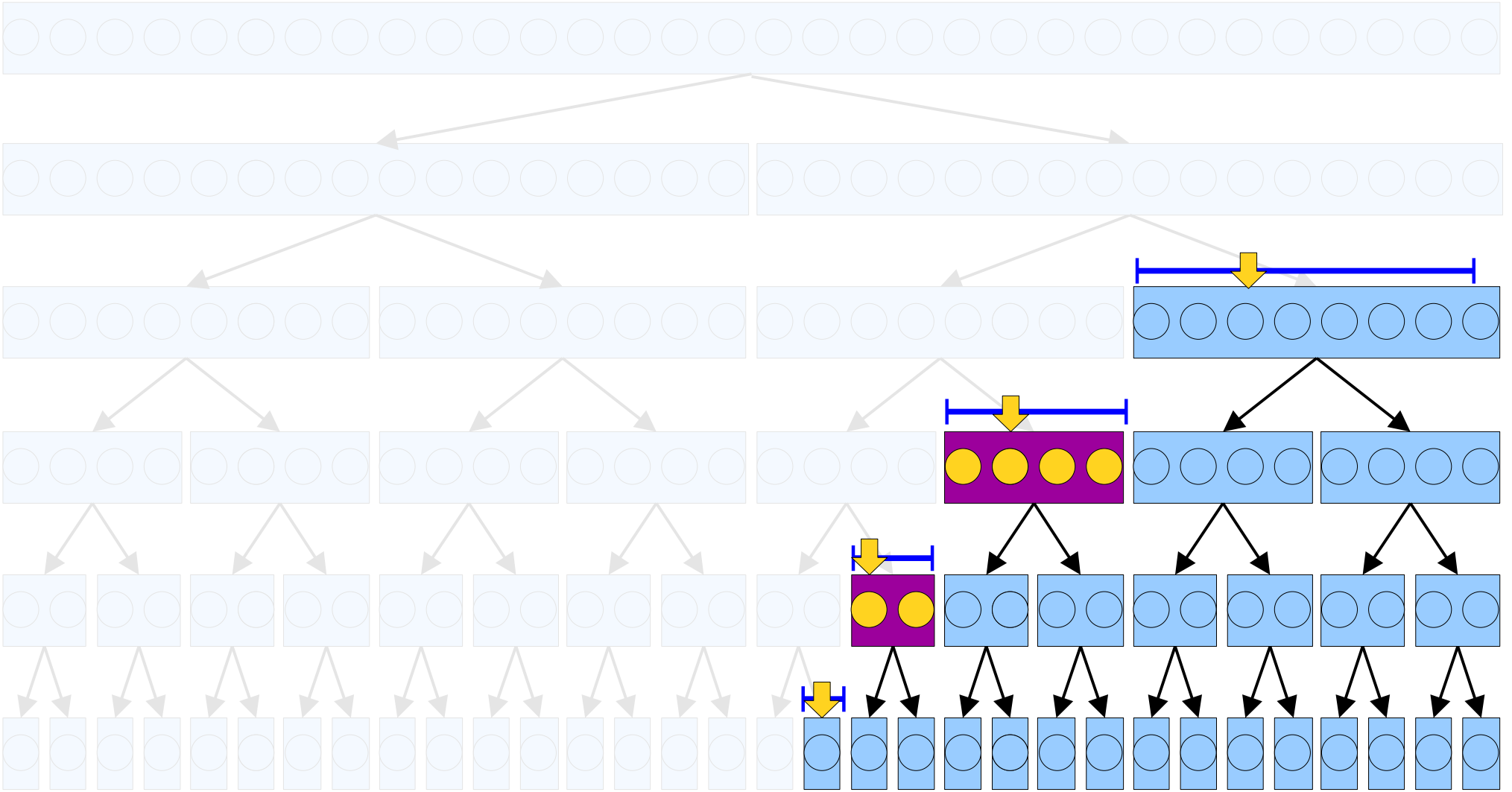


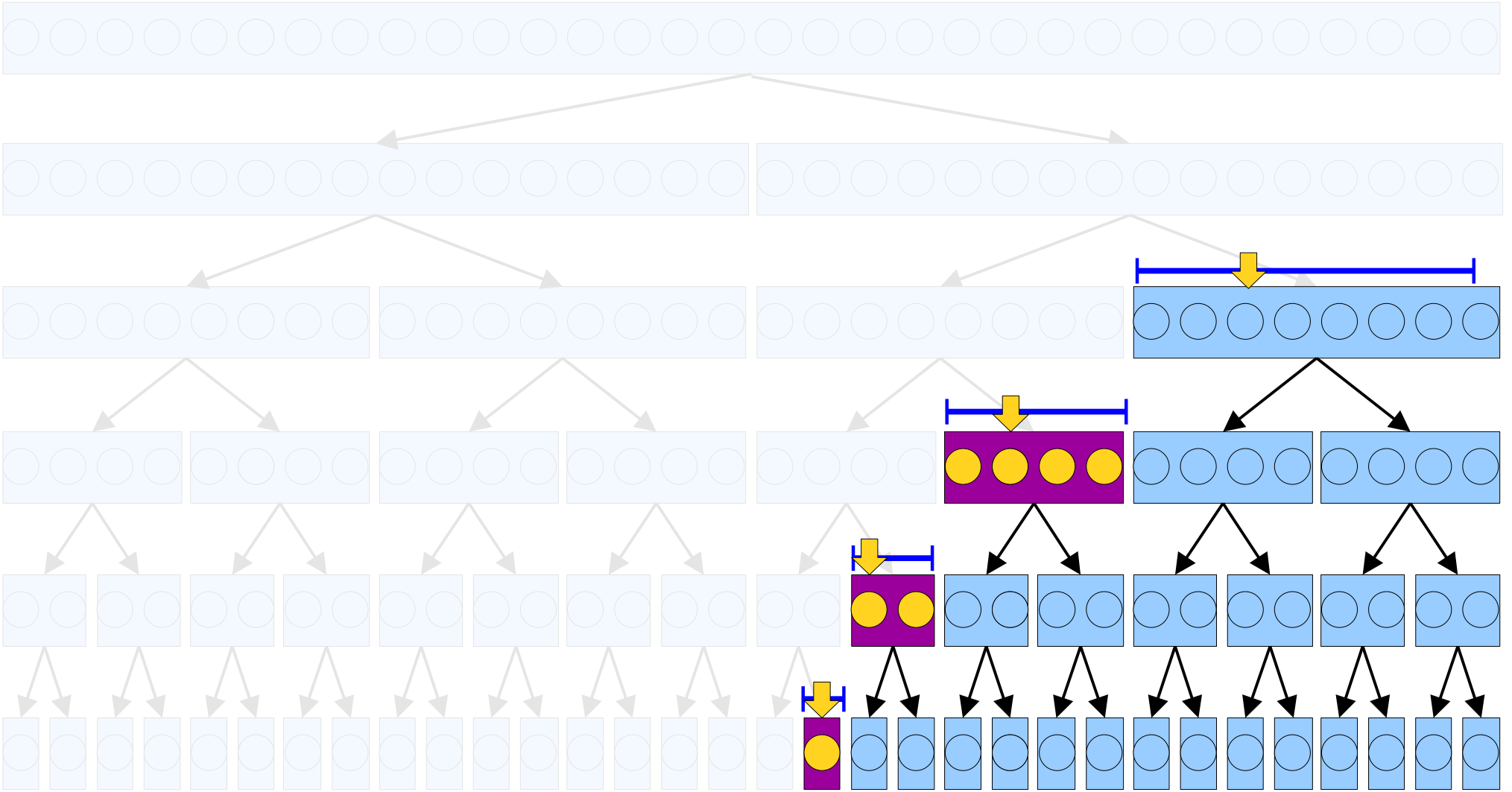


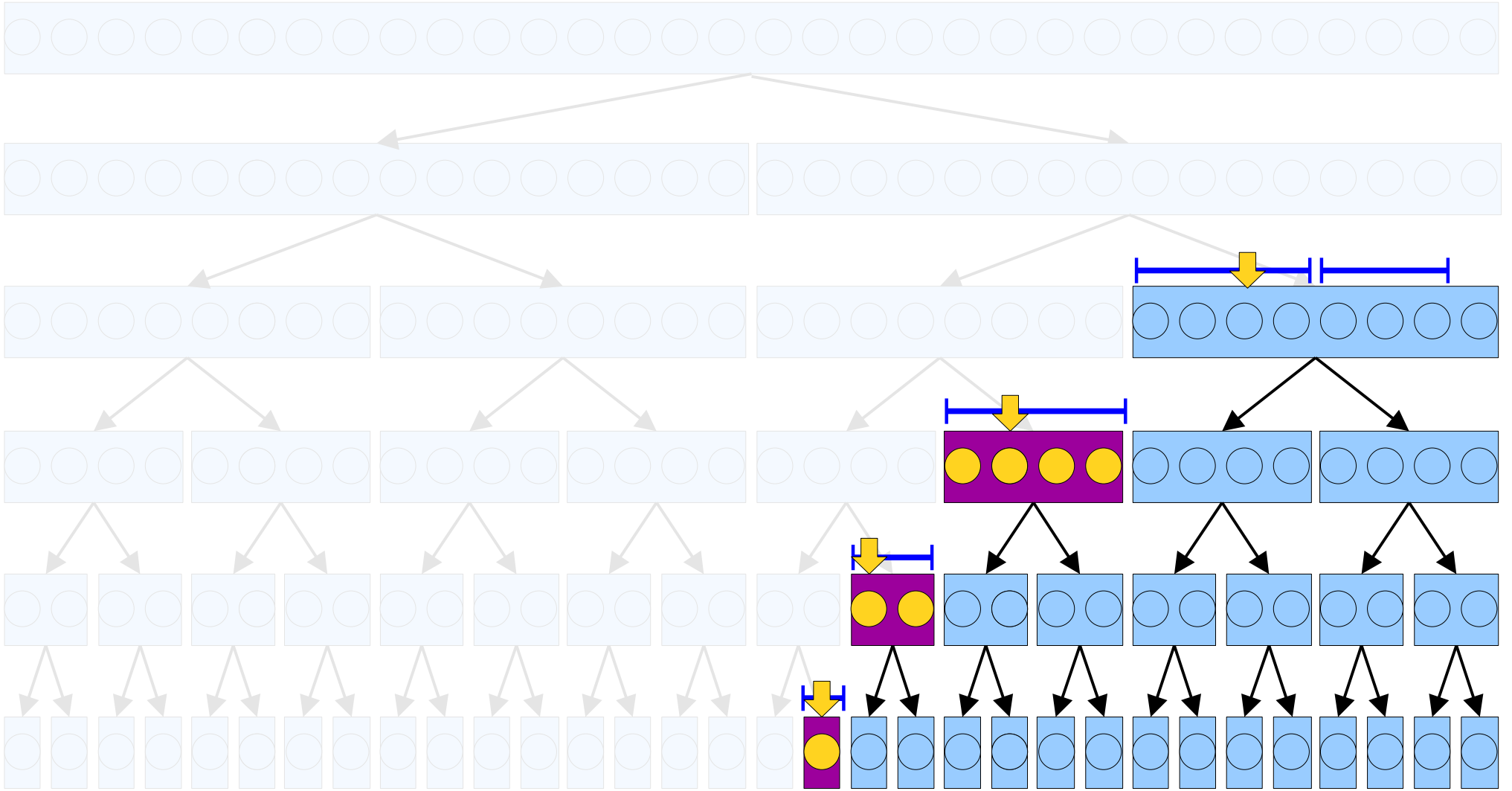


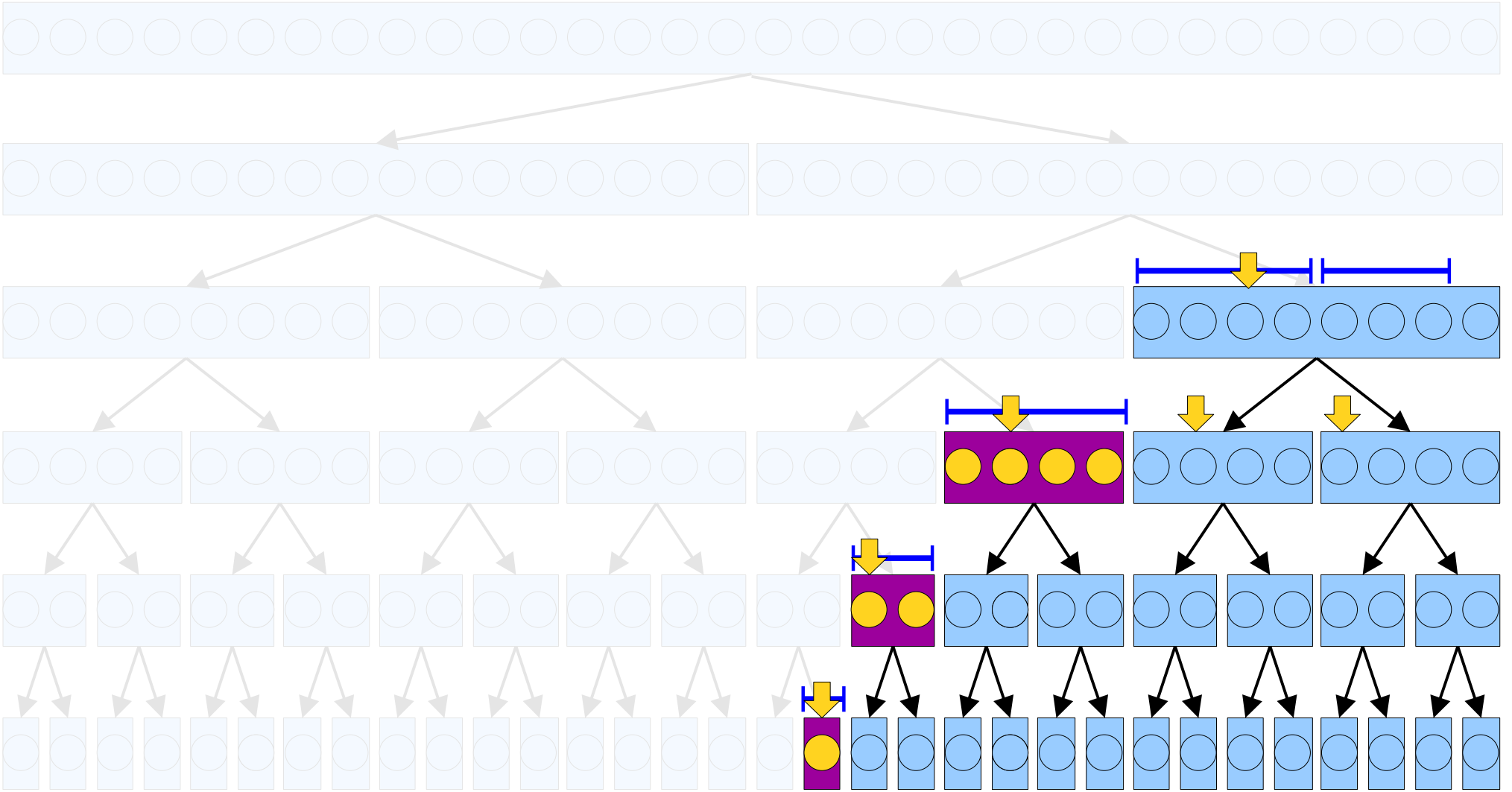


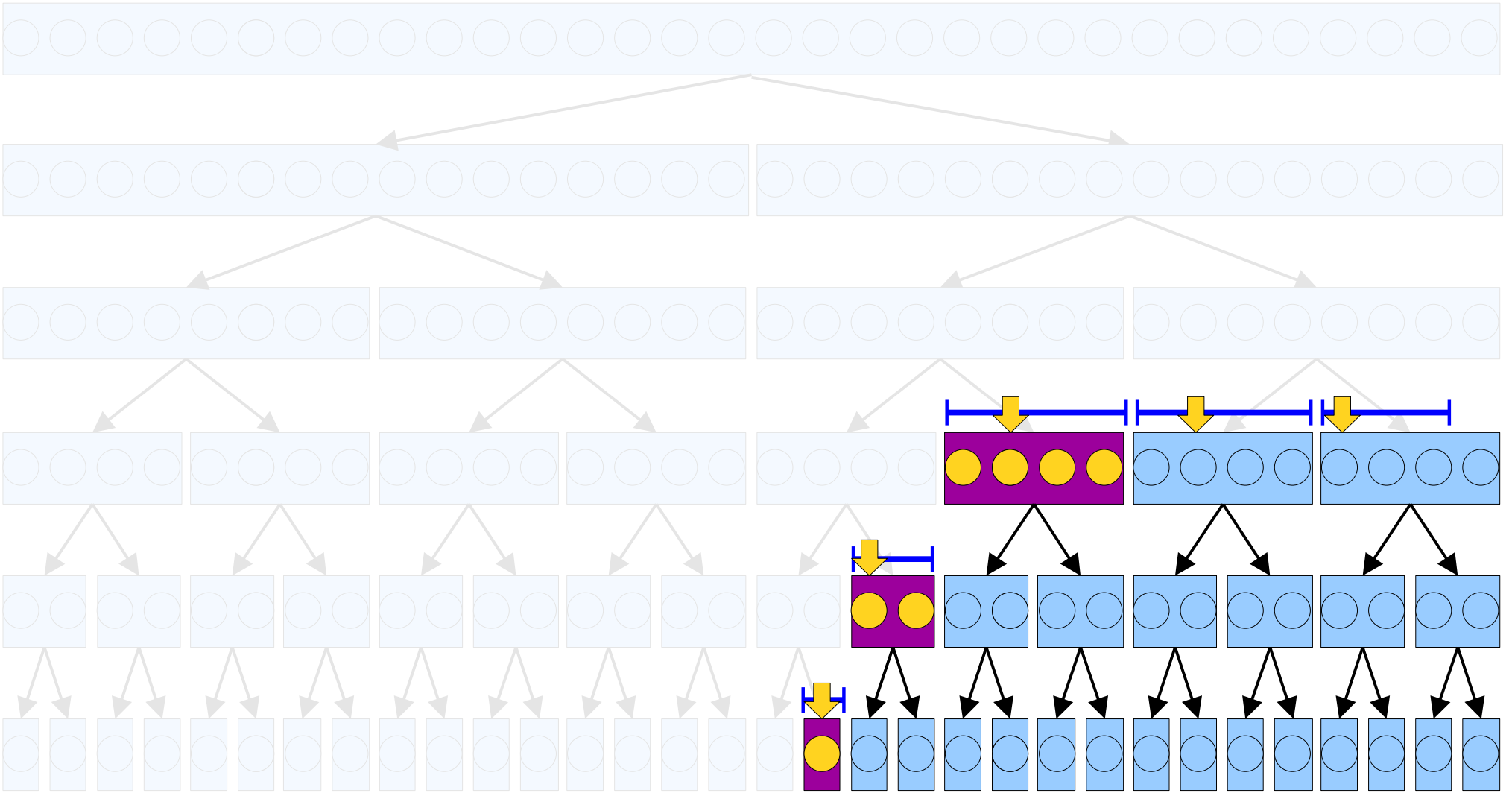


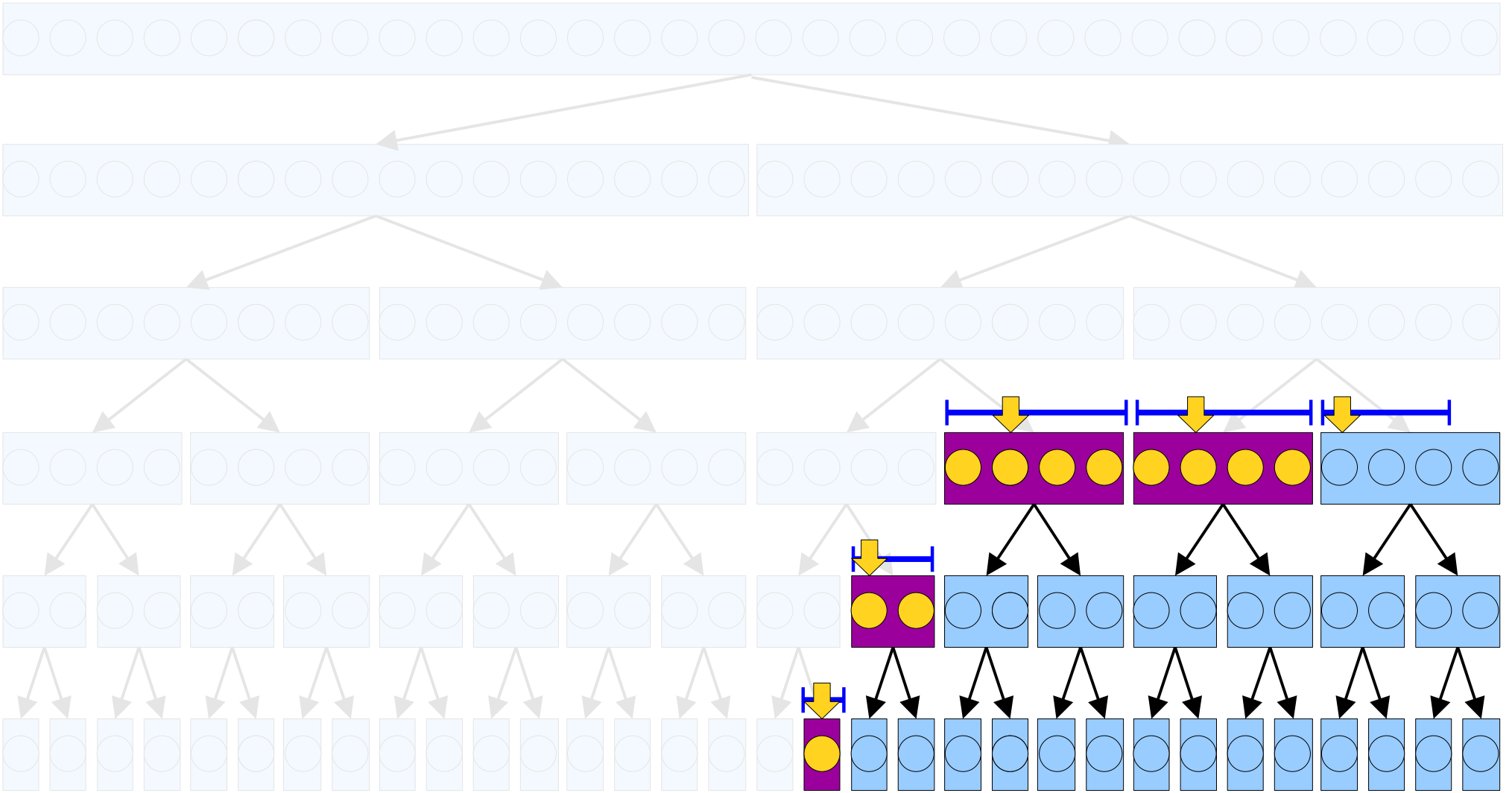


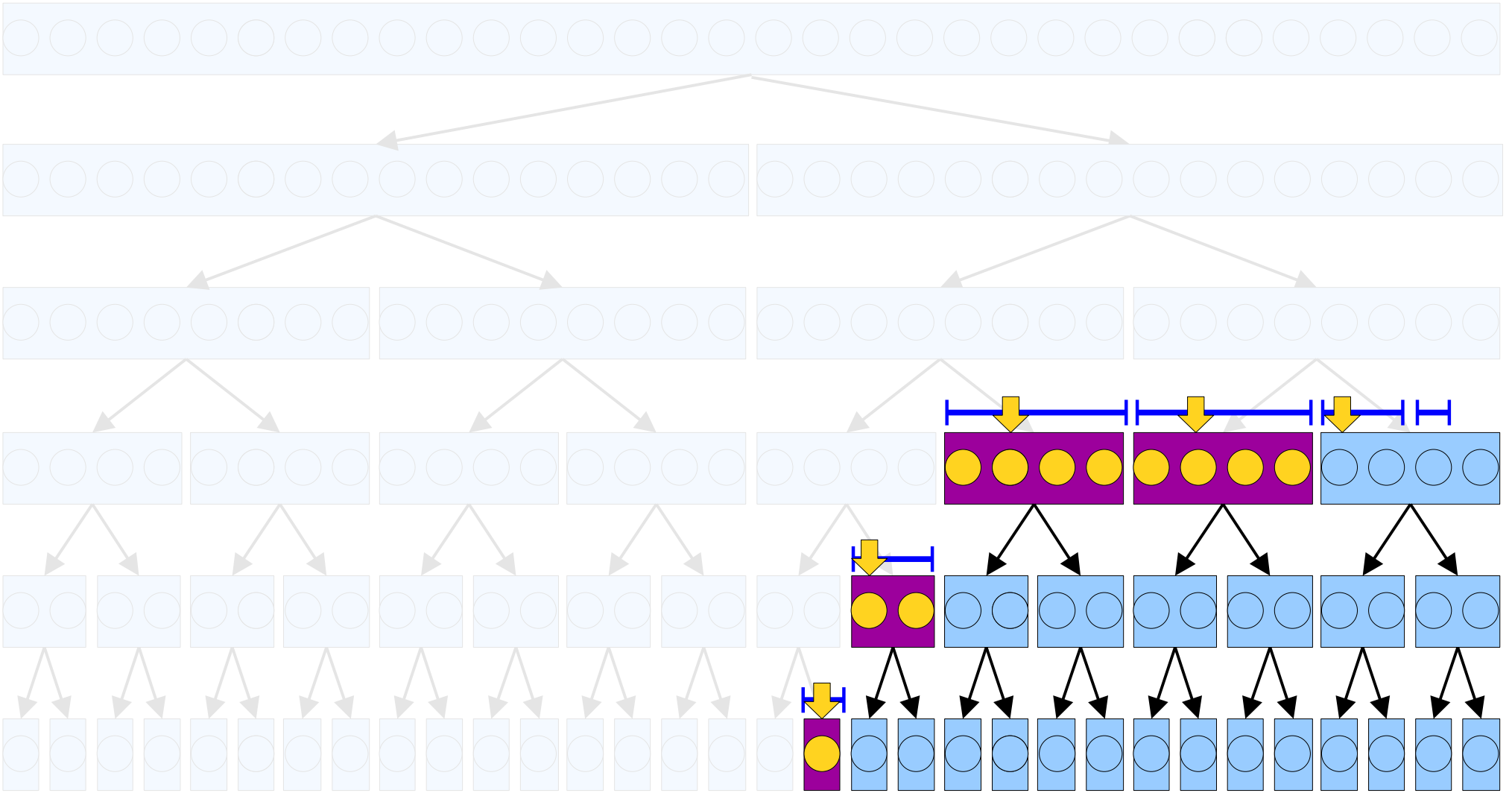


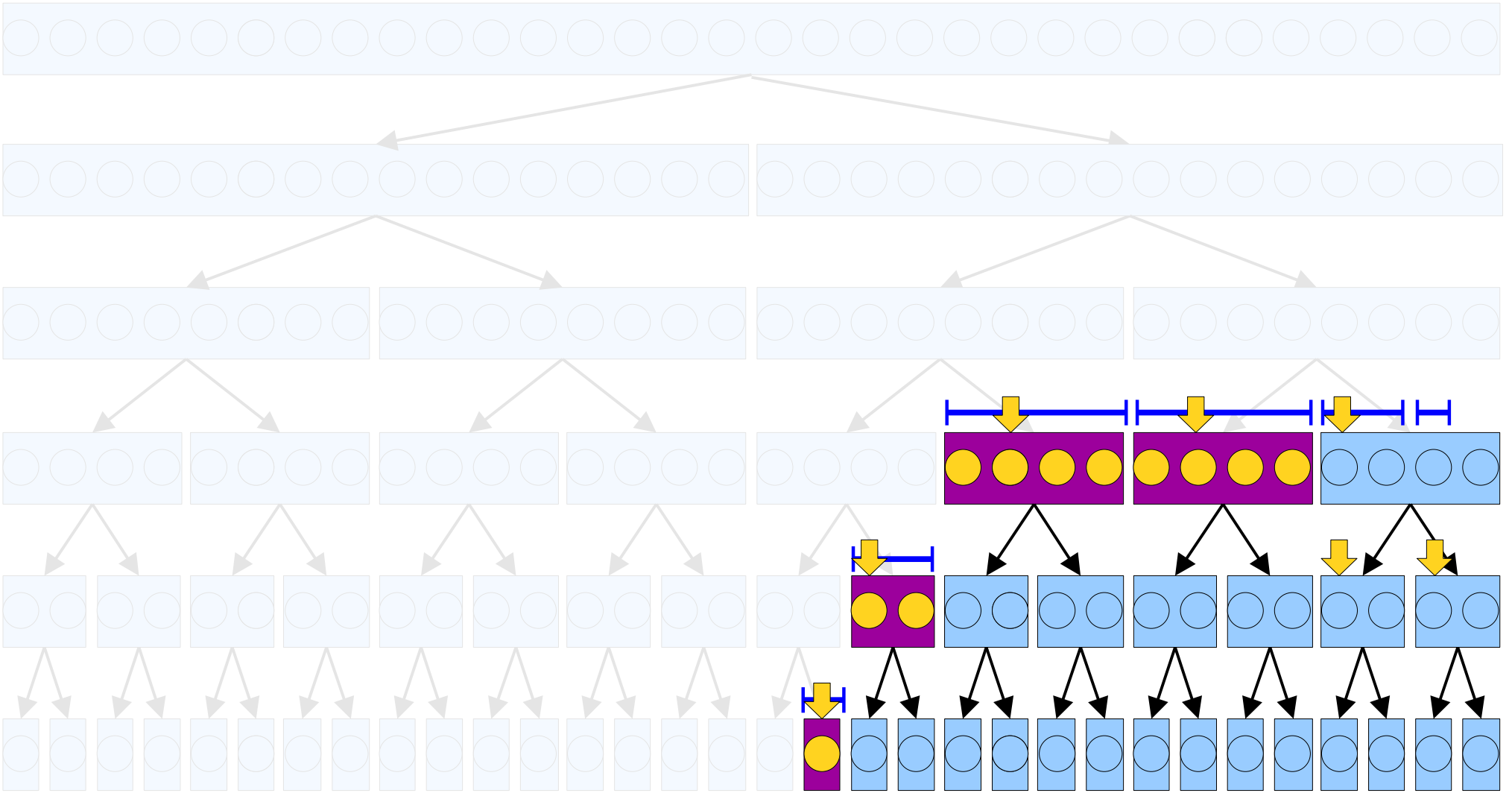


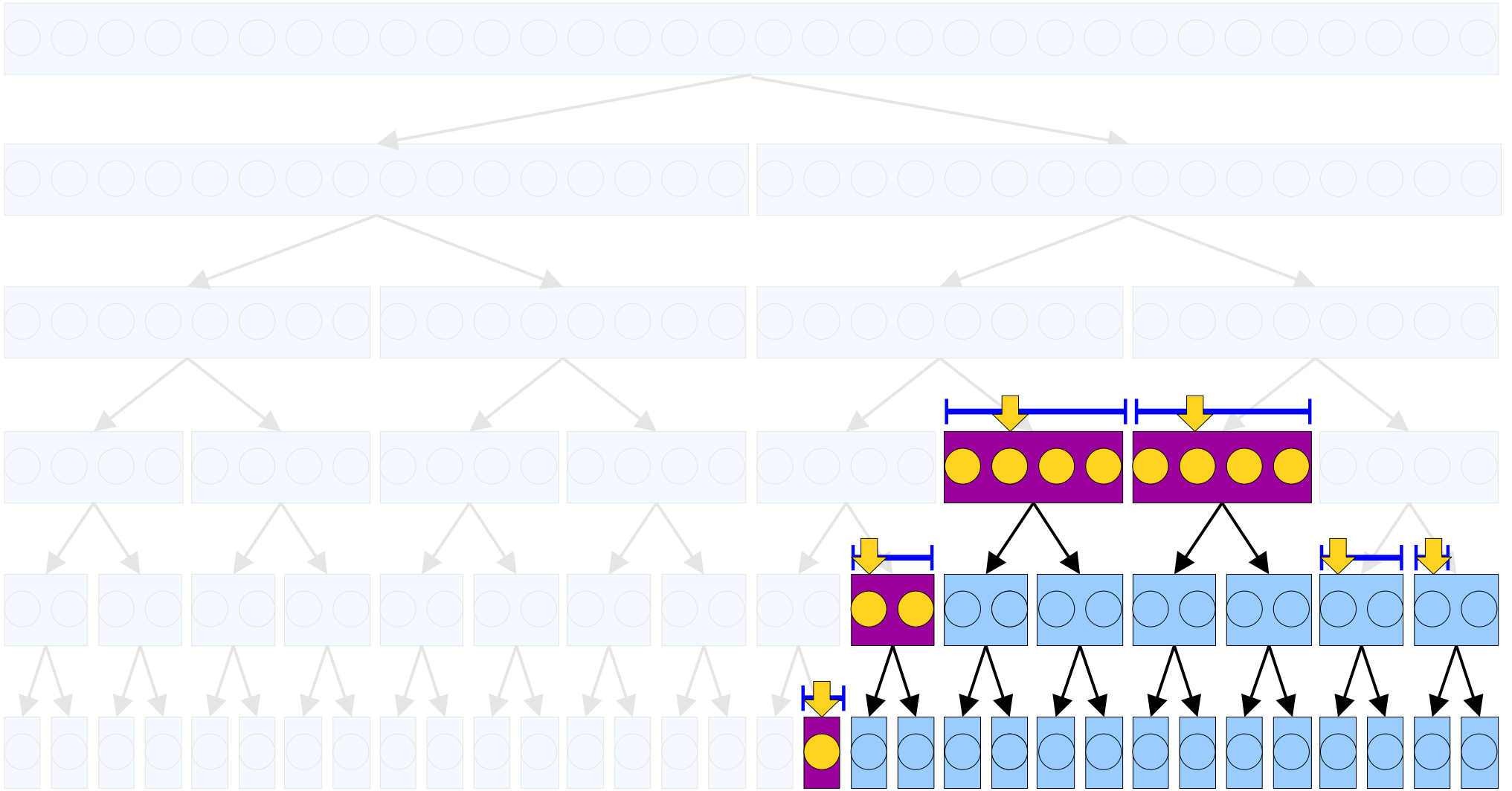


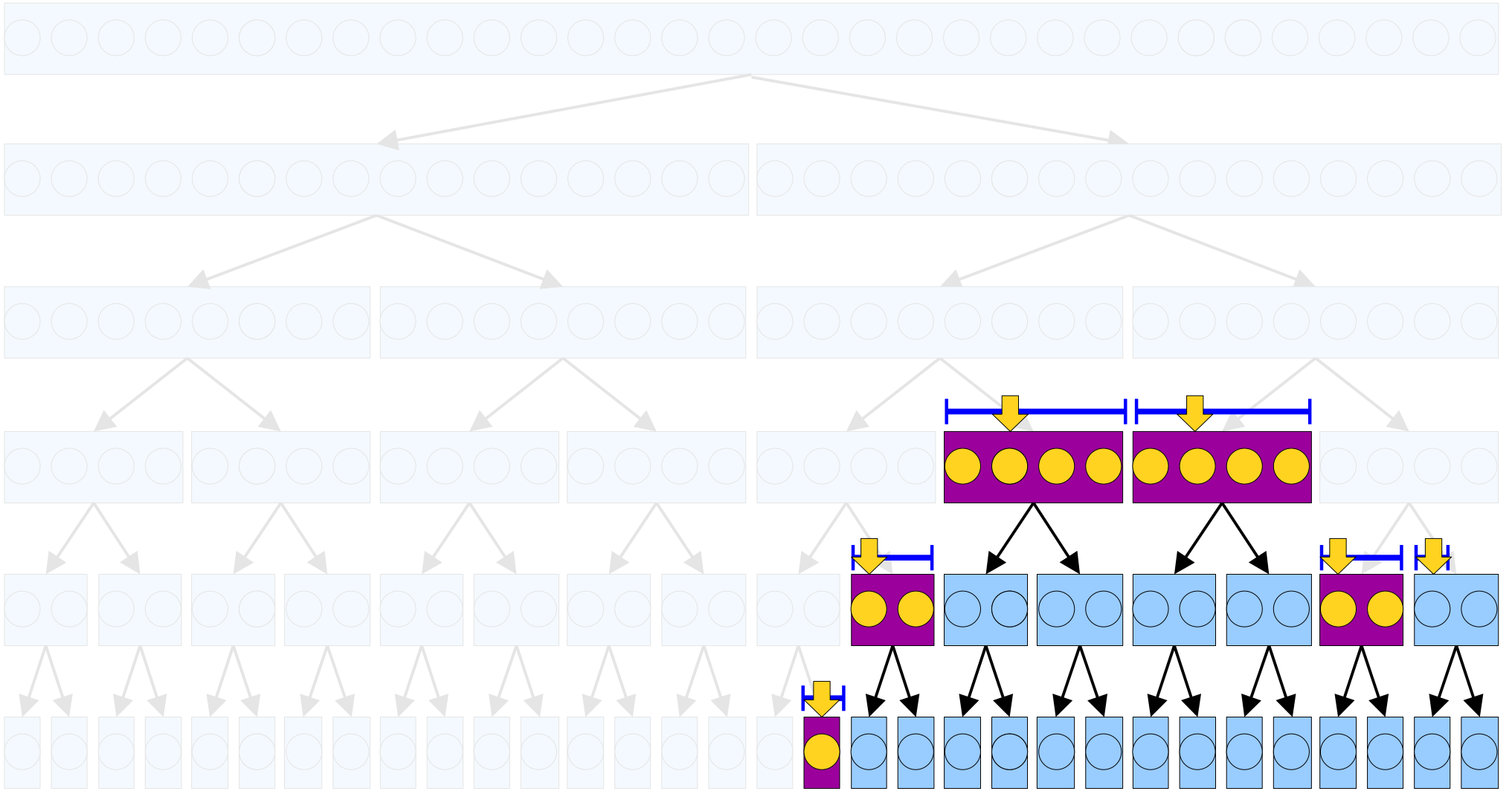


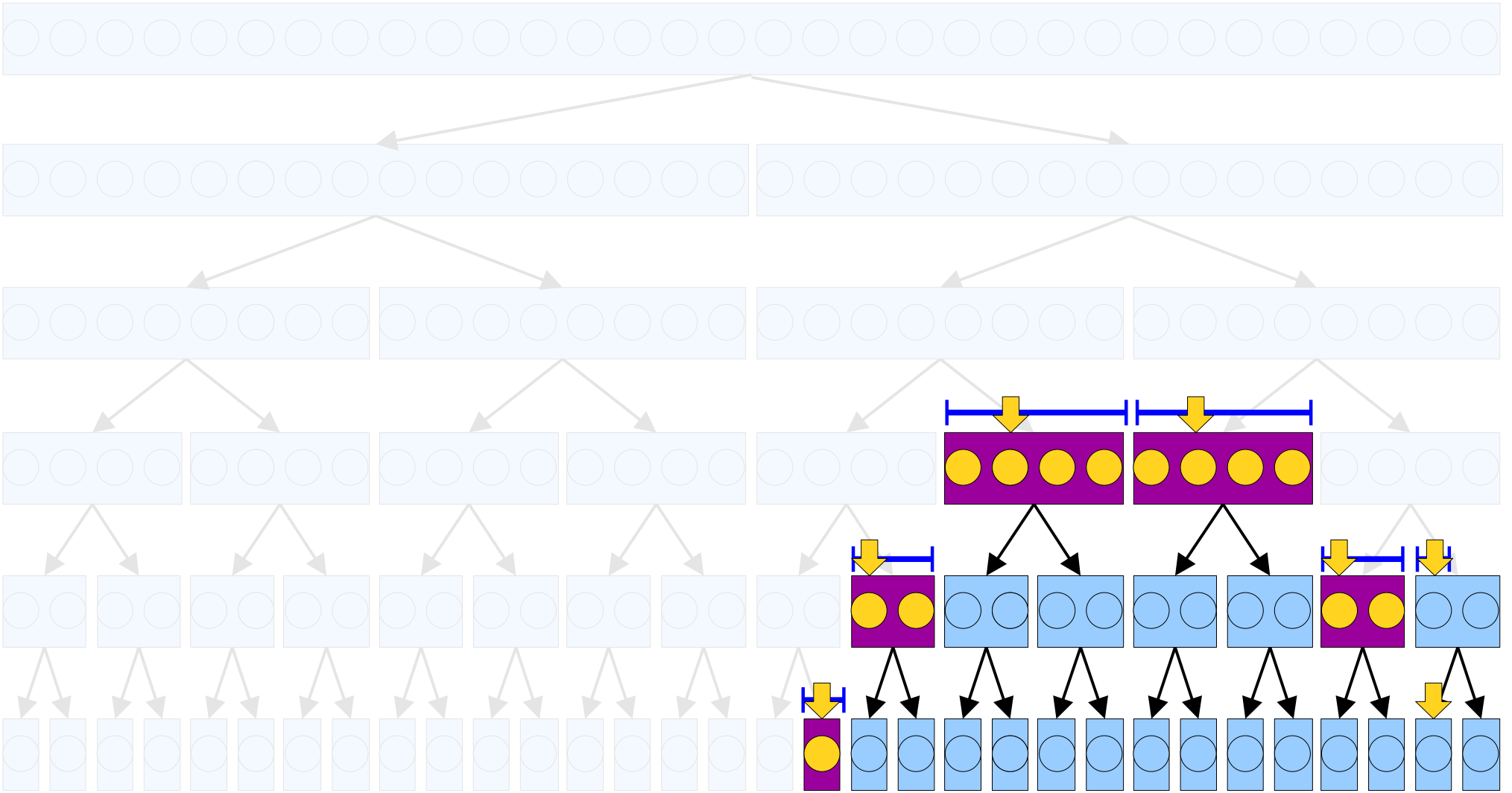


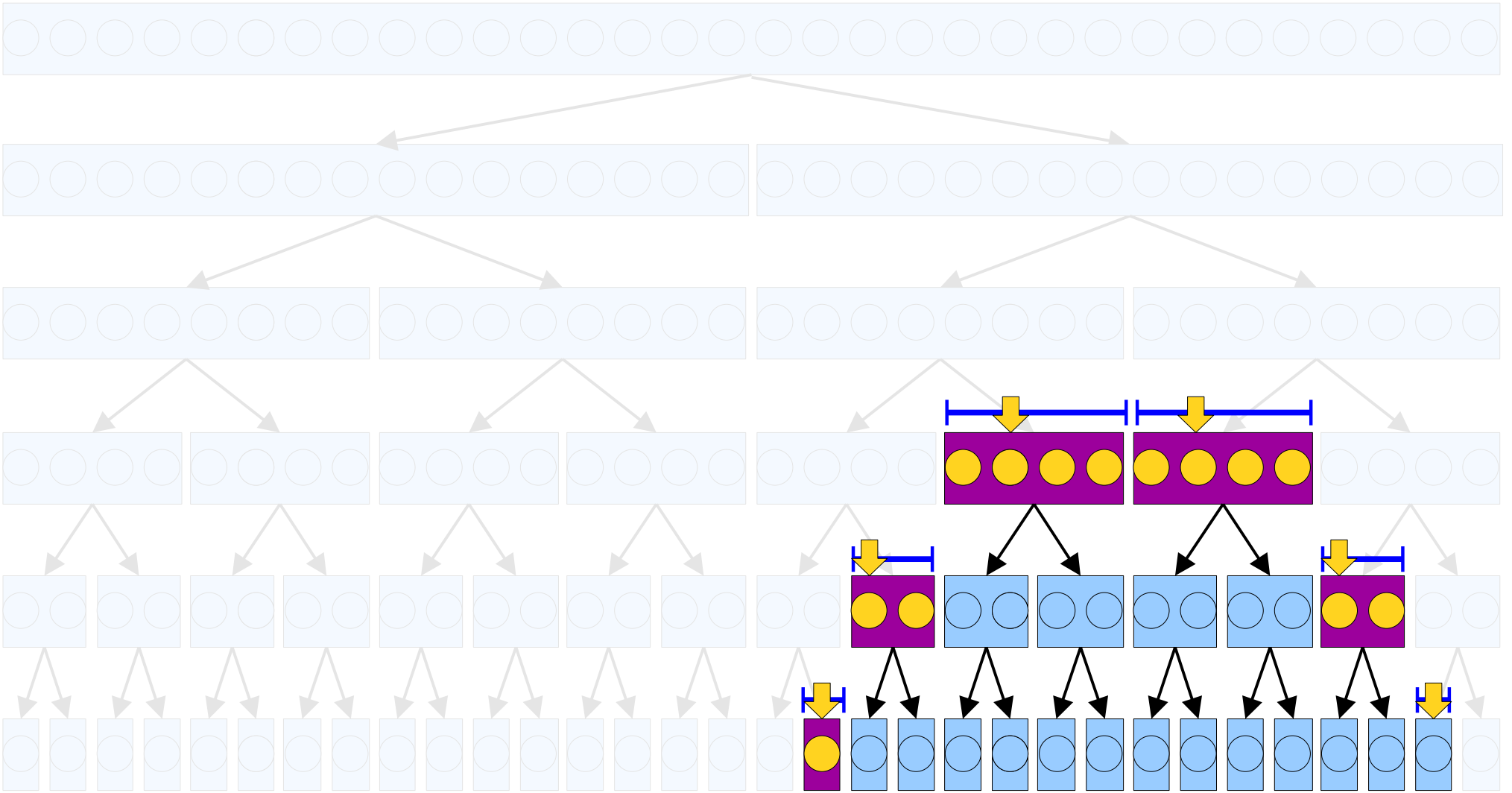


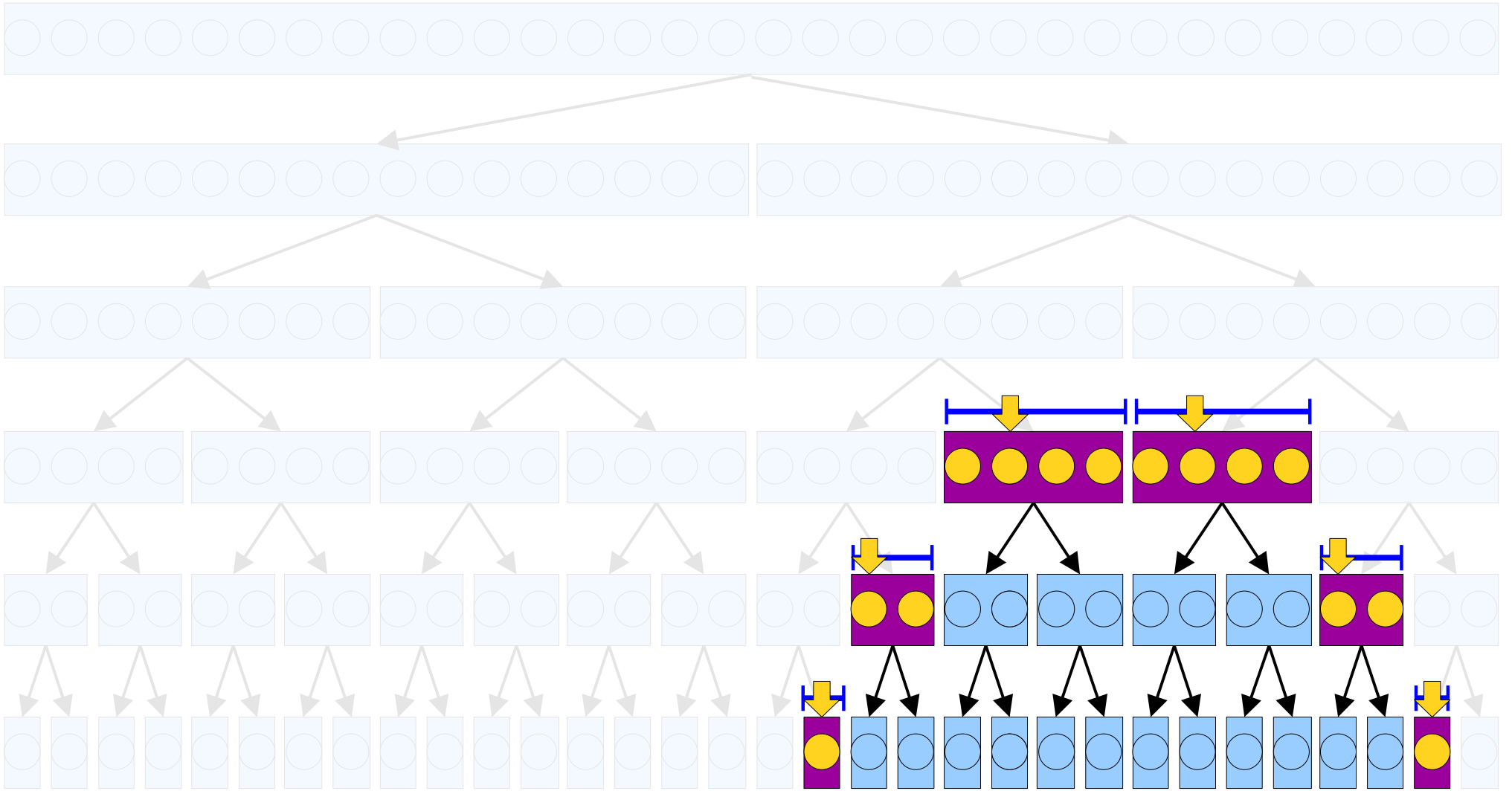




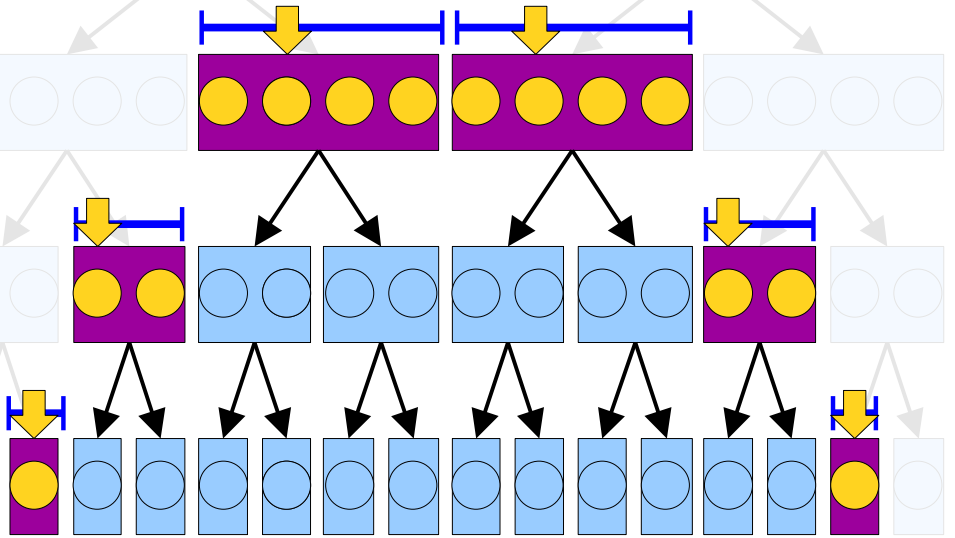








We no longer need to do binary searches in these ranges. Simply scan forward until we stop finding points in the y range.



Layered Range Trees

- A ***layered range tree*** is a range tree that uses downpointers.
- Here's how to do a query in a layered range tree.
  - Do a single binary search in the sorted array at the top of the tree to find the first point whose y coordinate is in range.
  - As before, walk the tree to find the O(log n) ranges to search. As you do, follow downpointers into each array so we know where a binary search in that block would start.
  - In each of the O(log n) blocks, do linear scans from the binary search pickup point to find all the matches. Every point scanned will be in the range (until we find a mismatch per block).
- Total work done:

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

$$= \mathbf{O(\log n + k)}.$$

Layered Range Trees

- The space usage in a layered range tree is (asymptotically) the same as for a range tree. We just store two extra downpointers per array slot.
- Downpointers can be filled in during preprocessing at no additional (asymptotic) cost.
  - Good exercise: Work through the details of how two merge the two child arrays while filling in downpointers.
- Time and space cost for preprocessing: **O(*n* log *n*)**.

The Final Scorecard

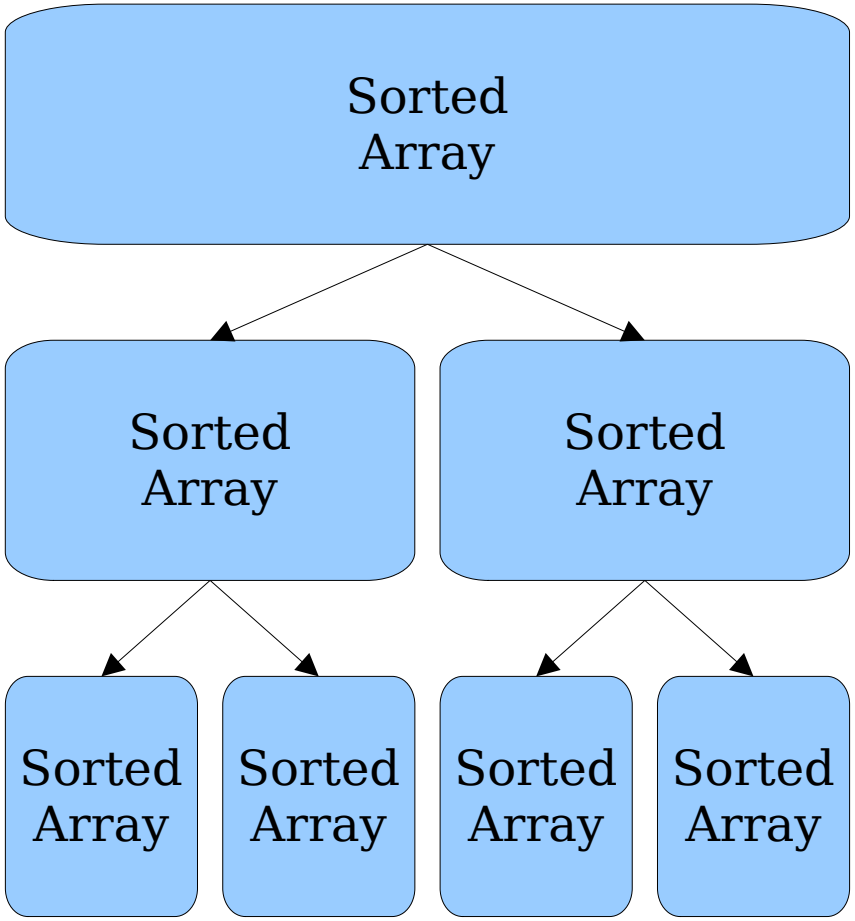
- Layered range trees are (asymptotically) as efficient as the precompute-all structure, yet use *significantly* less space!
- Further improvements are possible using some advanced techniques; come talk to me after class if you're curious how!

	Preprocessing Time	Query Time	Space Usage
Linear Scan	$O(1)$	$O(n)$	$O(n)$
Precompute-All	$O(n^5)$	$O(\log n + k)$	$O(n^5)$
Blocking	$O(n \log n)$	$O\left(\sqrt{n \log n} + k\right)$	$O(n)$
Range Tree	$O(n \log n)$	$O(\log^2 n + k)$	$O(n \log n)$
Layered Range Tree	$O(n \log n)$	$O(\log n + k)$	$O(n \log n)$

Higher-Dimensional Searches

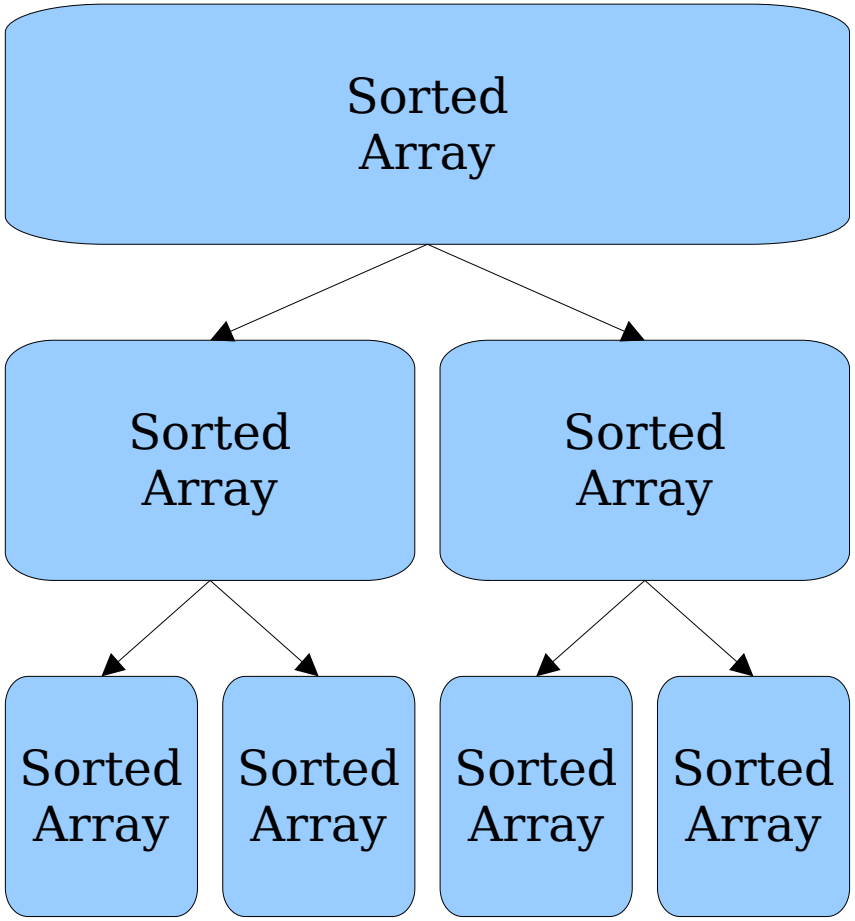
3D Searches

- How might we do these searches in three dimensions?
  - i.e. Find all 3D points in an axis-aligned rectangular prism.
- Our 2D approach looks like this:
  - Use binary search over x coordinates to identify O(log n) lower-dimensional structures (sorted arrays) to search.
  - As efficiently as possible, search those structures for all points in the appropriate y range.
- Can we scale this up?



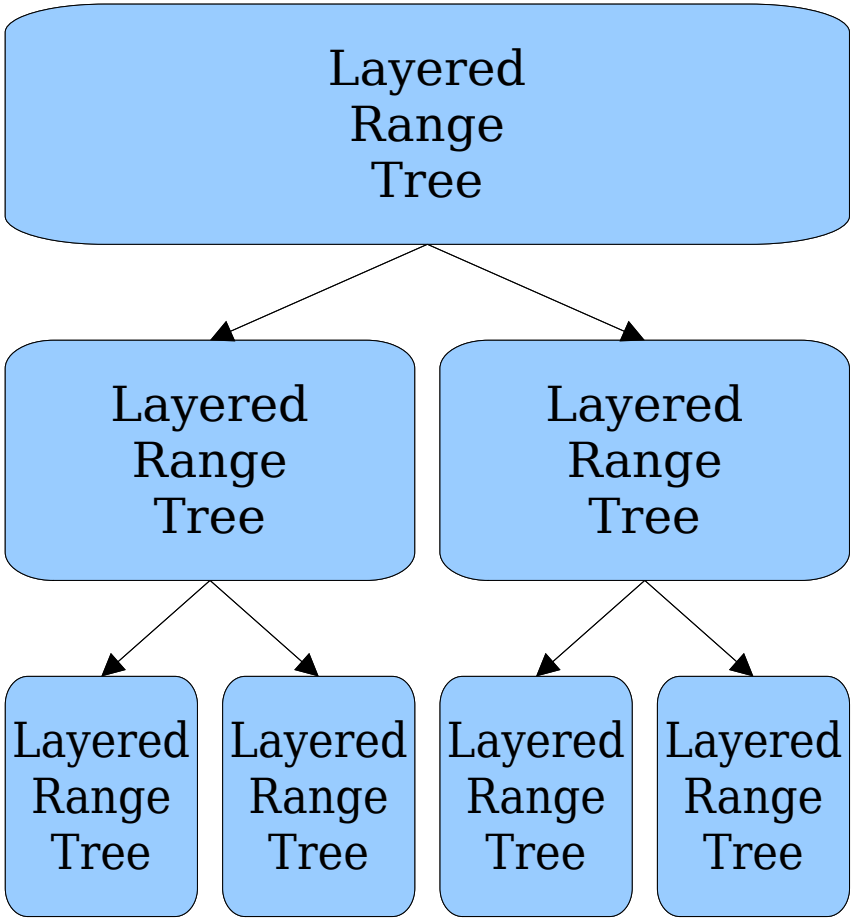
3D Searches

- ***Main insight:*** Our 2D approach works by
  - using a binary search over one dimension to find O(log n) data structures sorted by the other dimension, then
  - searching those structures as efficiently as possible and aggregating the results.
- ***Basic idea:*** Build a tree structure as in the range tree to remove one dimension, then recursively process the remaining dimensions.



3D Searches

- Here's how to do this in 3D:
  - Break the points apart into blocks based on their z coordinates using the same strategy we saw earlier.
  - Build a 2D range search structure for each block based on their x and y coordinates.
  - Query by searching in the z direction to find which range search structures to query, then query them for all points whose x and y coordinates match.
- ***Question:*** How time- and space-efficient is this approach?

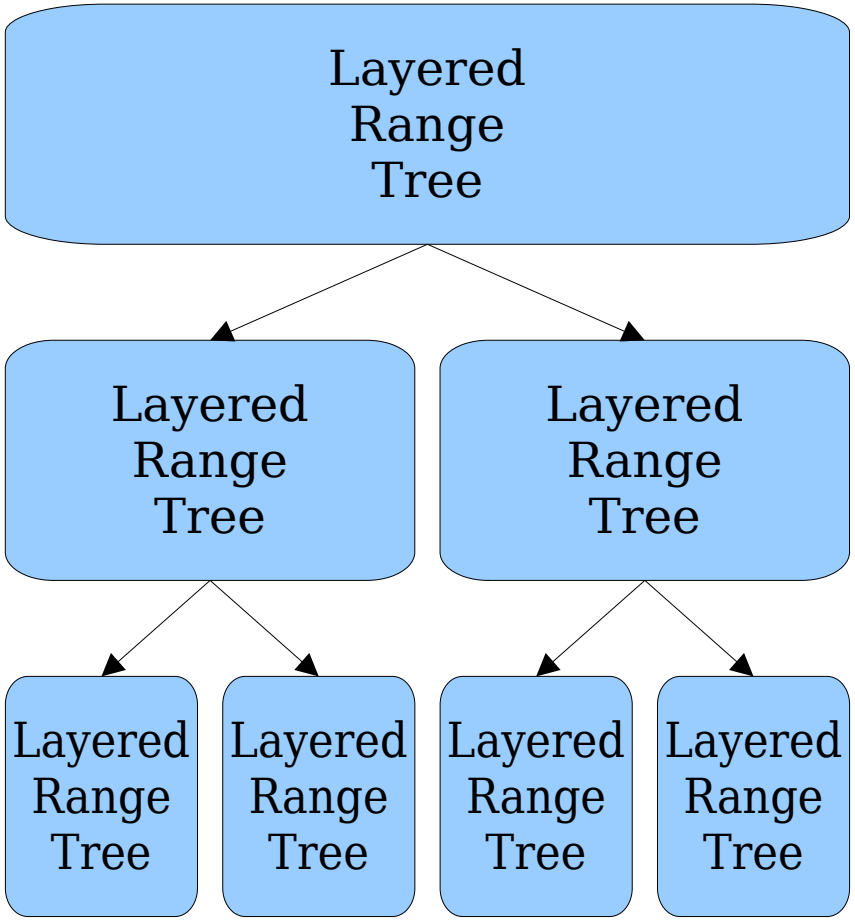


3D Searches

- Preprocessing time and space usage are given by the recurrence

  $$T(n) = 2T(n / 2) + O(n \log n),$$

  which solves to **$O(n \log^2 n)$**.
- Query time:
  - Do $O(\log n)$ work to identify $O(\log n)$ layered range trees, each of which stores points where one coordinate definitely matches.
  - Search each structure. Base time is $O(\log n)$ per query, plus extra time per match.
- Total query time: **$O(\log^2 n + k)$**.



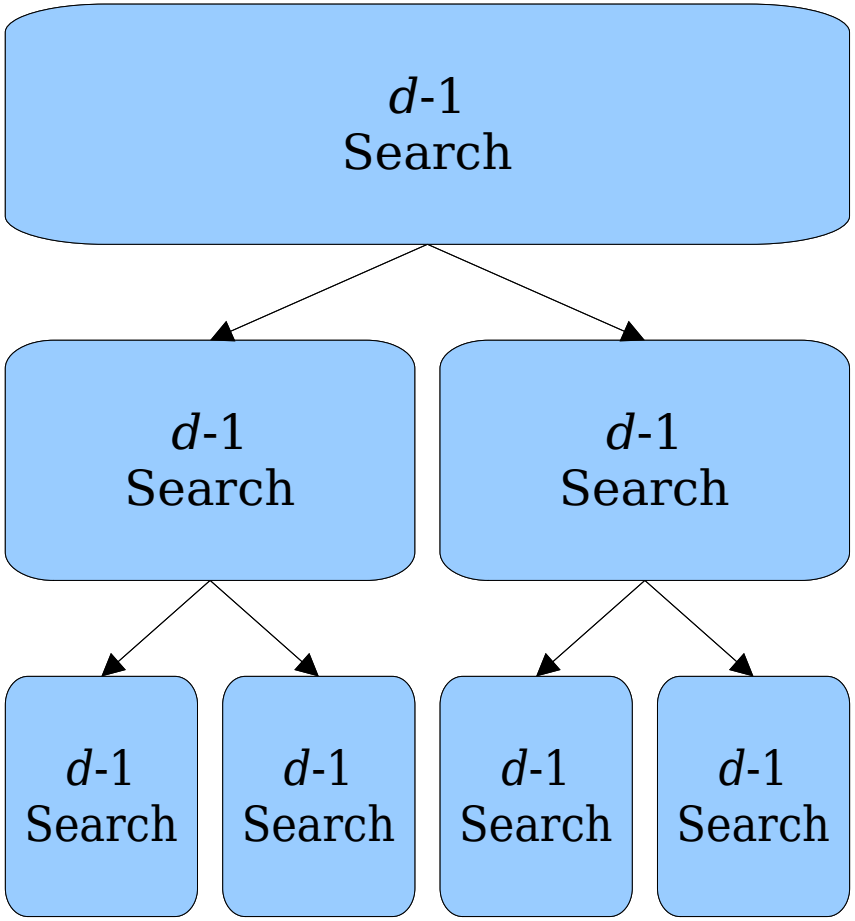
3D Searches

- Our original 2D range tree performed 2D searches in time $O(\log^2 n + k)$.
- By using some clever insights about the structure of 2D queries (geometric cascading), we dropped that down to $O(\log n + k)$.
- By using some clever observations about 3D searches, we can reduce the cost of 3D searches as well.

	Query Time	Space Usage
3D Layered Range Tree	$O(\log^2 n + k)$	$O(n \log^2 n)$
Chazelle/Guibas (1986)	$O(\log n + k)$	$O(n \log^3 n)$
Alstrup et al (2000)	$O(\log n + k)$	$O(n \log^{1+\varepsilon} n)$ *(for any fixed $\varepsilon > 0$)*

Higher Dimensional Searches

- Here's a recursive construction for any dimension d:
  - Build a tree of blocks based on just one coordinate.
  - Recursively construct (d – 1)-dimensional data structures for each block.
  - Query by determining which blocks have the initial coordinate in the right place, then recursively querying the blocks to find the matches.
- Each added dimension multiplies the time and space usage of the previous dimension by O(log n).



	Query Time	Space Usage
1D Base Case: Sorted Arrays (1978)	$O(\log^d n + k)$	$O(n \log^{d-1} n)$
2D Base Case: Layered Range Trees (1978)	$O(\log^{d-1} n + k)$	$O(n \log^{d-1} n)$
3D Base Case: Chazelle/Guibas (1986)	$O(\log^{d-2} n + k)$	$O(n \log^{d+3} n)$
3D Base Case: Alstrup et al (2000)	$O(\log^{d-2} n + k)$	$O(n \log^{d-2+\varepsilon} n)$ *(for any fixed $\varepsilon > 0$)*
4D Base Case: Nekrich (2021)	$O\left(\frac{\log^{d-3} n}{\log \log n} + k\right)$	$O(n \log^{d-4+\varepsilon} n)$ ***(for any fixed $\varepsilon > 0$)***

More to Explore

- Geometric cascading and downpointers generalize to ***fractional cascading***, a powerful building block for multidimensional data structures.
- For general halfspace queries, where we want to find all points above or below a given line, there are ***ham sandwich trees*** that find matches in time $O(n^{\log_2 \varphi} + k)$.
- The ***k-d tree*** uses linear space and solves orthogonal range searching in $O(n)$ space and query time $O(n^{1/2} + k)$, generalizing to $O(n^{1 - 1/d} + k)$ for higher dimensions.
- Range searches where all points are on an $n \times n$ grid can be optimized to run in time $O(\log \log n + k)$, and this is provably optimal under some reasonable assumptions.
- ***R-trees***, while less theoretically optimal, are a powerful tool in practice for doing 2D range searches. They're based on B-trees and some recursive cleverness.

Next Time

- ***Plane Graphs***
  - Partitioning space into regions.
- ***Point-in-Polygon Searches***
  - Which region of space is a point in?
- ***Persistent Data Structures***
  - Version control meets data structures.