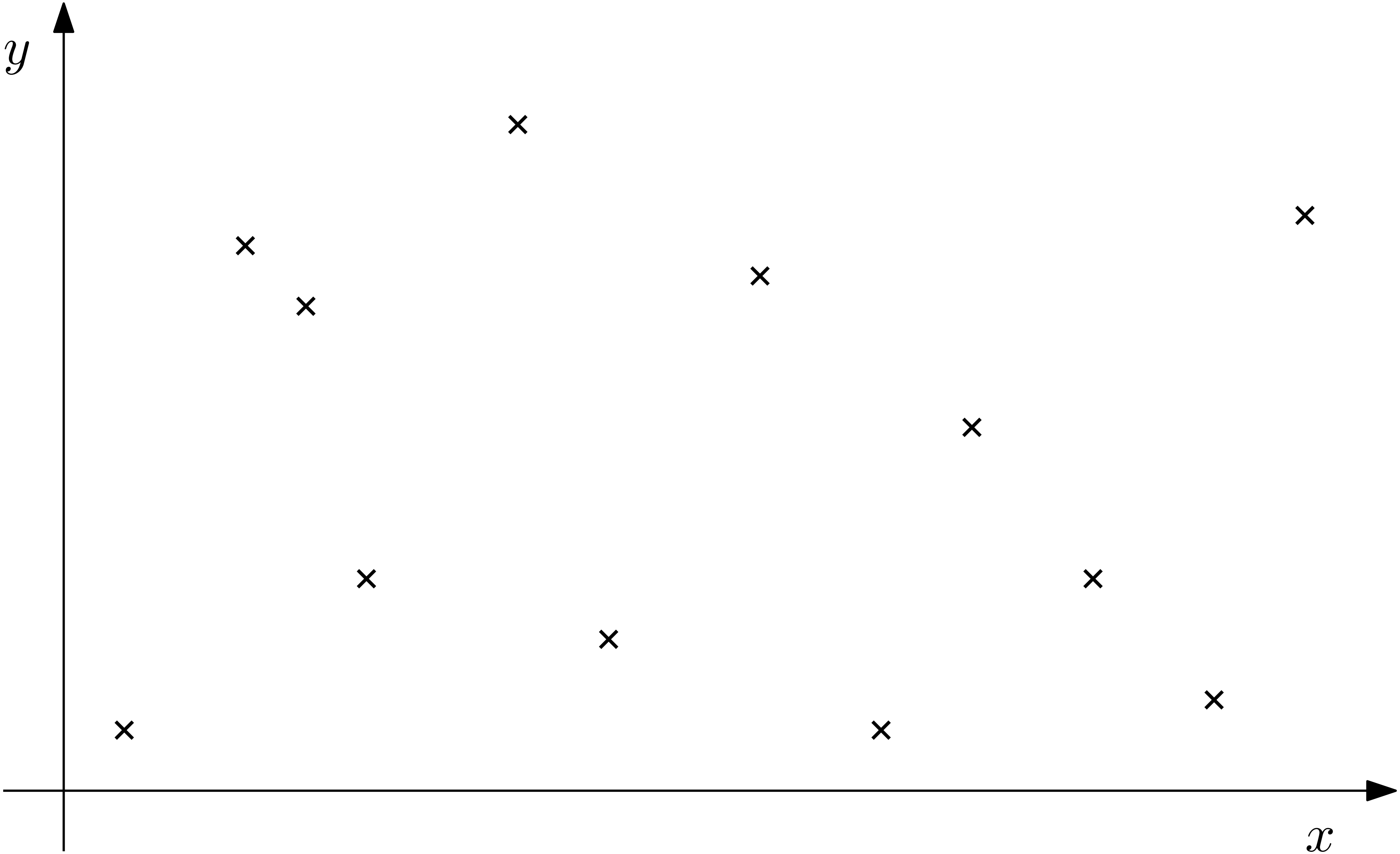
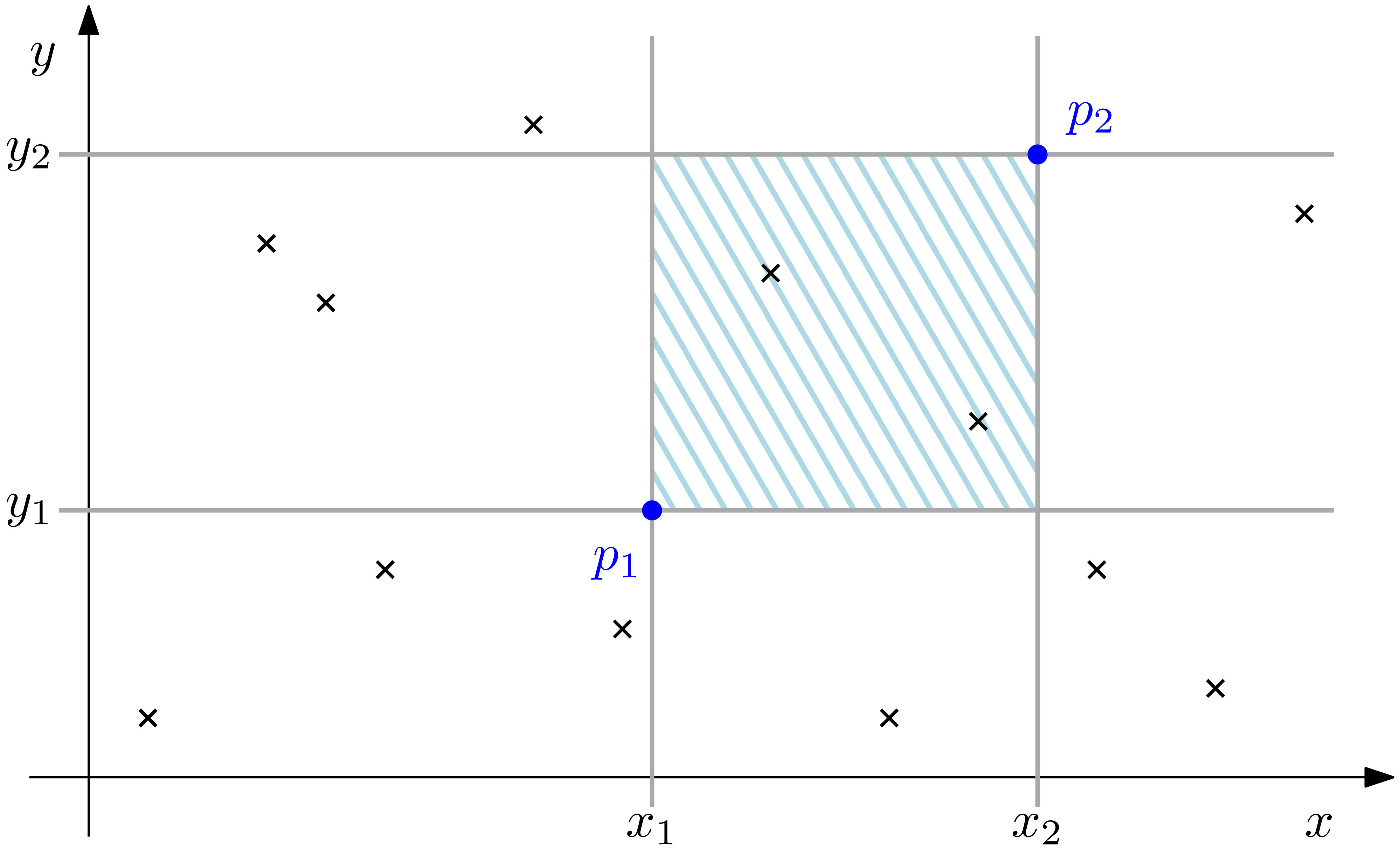
Range Trees

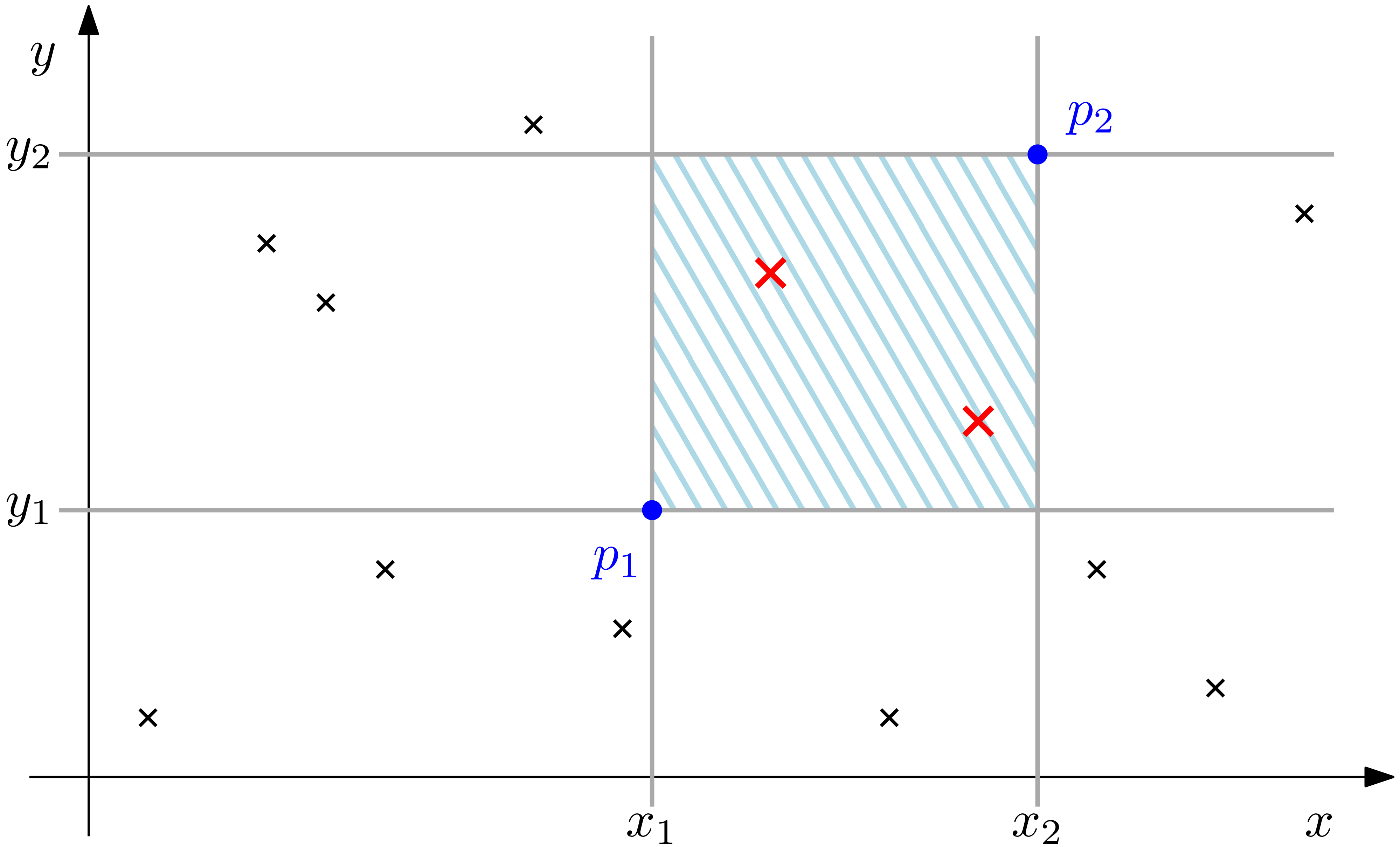
Range Trees



Range Trees



Range Trees



Range Trees

Input:

A set S of n D-dimensional points.

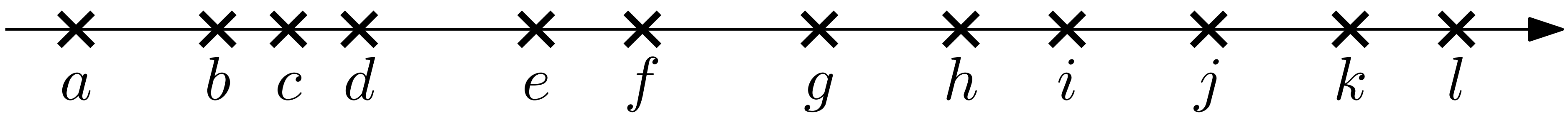
Goal:

Design a data stucture that, given $p_1 \in \mathbb{Z}^D, p_2 \in \mathbb{Z}^D$ can:

- Report *the number* of points $q \in S$ such that $p_1 \leq q \leq p_2$.
- Report *the set* of points $q \in S$ such that $p_1 \leq q \leq p_2$.
- Report the point $q \in S$, $p_1 \leq q \leq p_2$, with *smallest* D-th coordinate.
- ...

An easy case: $D = 1$

- Points are integers
- Store points in a sorted array (in time $O(n \log n)$).
- Perform queries by binary searching for p_1 and p_2

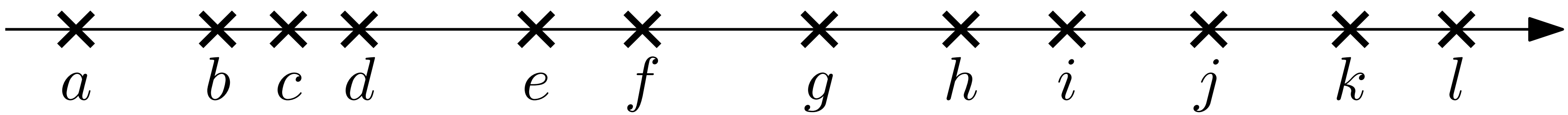


An easy case: $D = 1$

- Points are integers
- Store points in a sorted array (in time $O(n \log n)$).
- Perform queries by binary searching for p_1 and p_2

Query time: $O(\log n + k)$ $k =$ "size" of the output.

- $k = \#$ reported points.
- $k = \Theta(1)$ if we only care about the *number* of points.



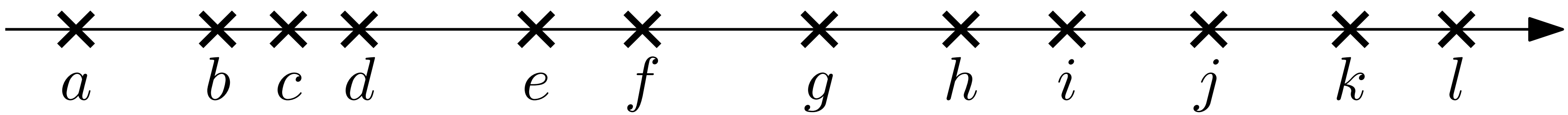
An easy case: $D = 1$

- Points are integers
- Store points in a sorted array (in time $O(n \log n)$).
- Perform queries by binary searching for p_1 and p_2

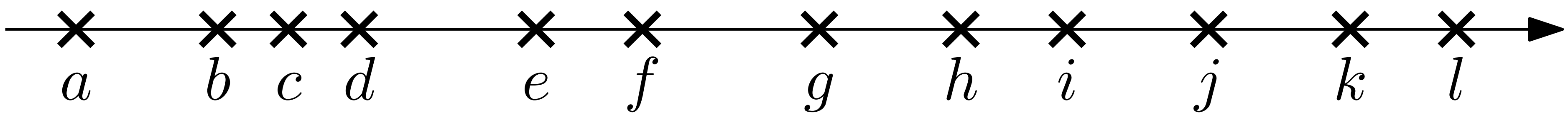
Query time: $O(\log n + k)$ $\qquad$ $k =$ "size" of the output.

- $k = \#$ reported points.
- $k = \Theta(1)$ if we only care about the *number* of points.

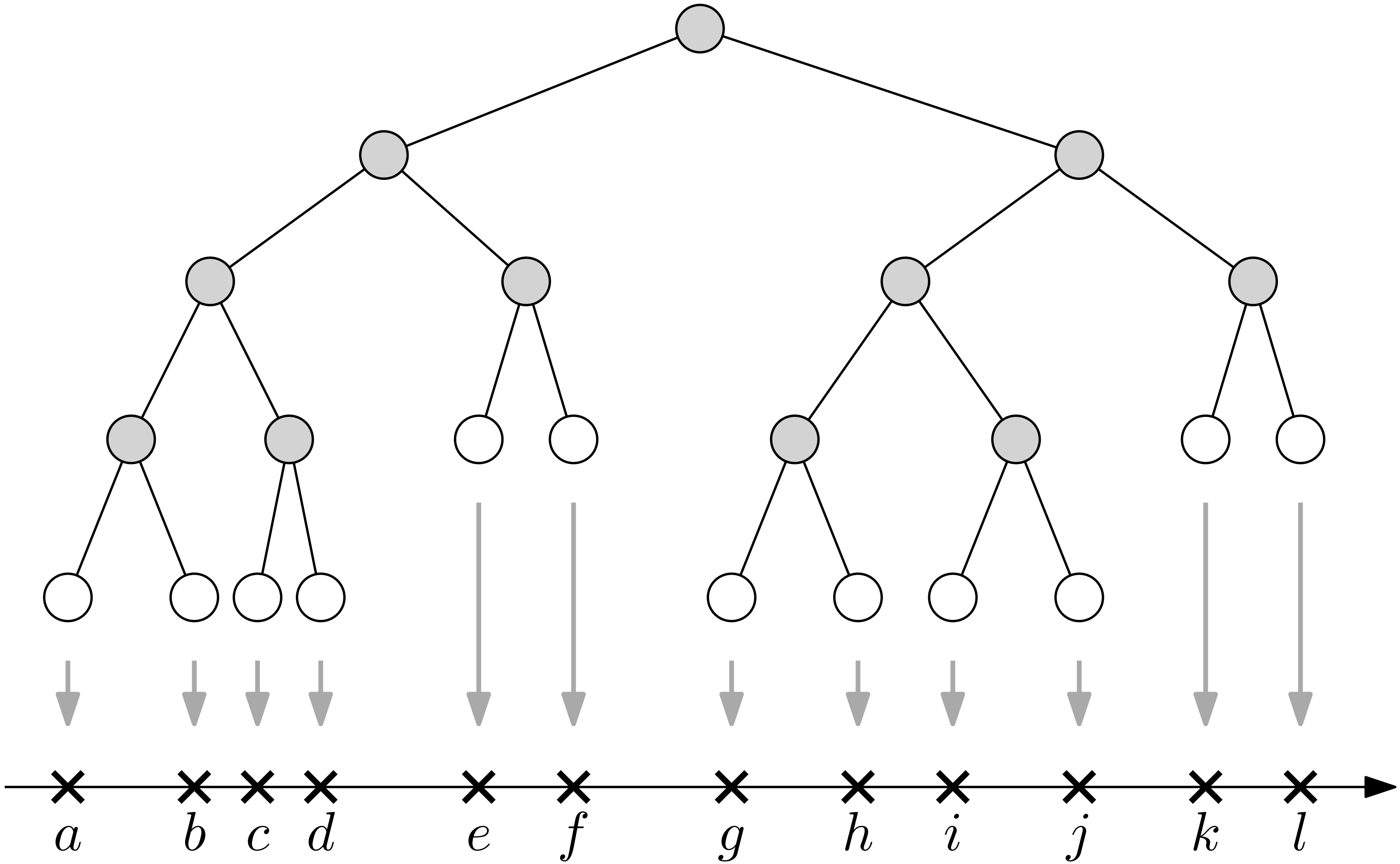
Space complexity: $O(n)$



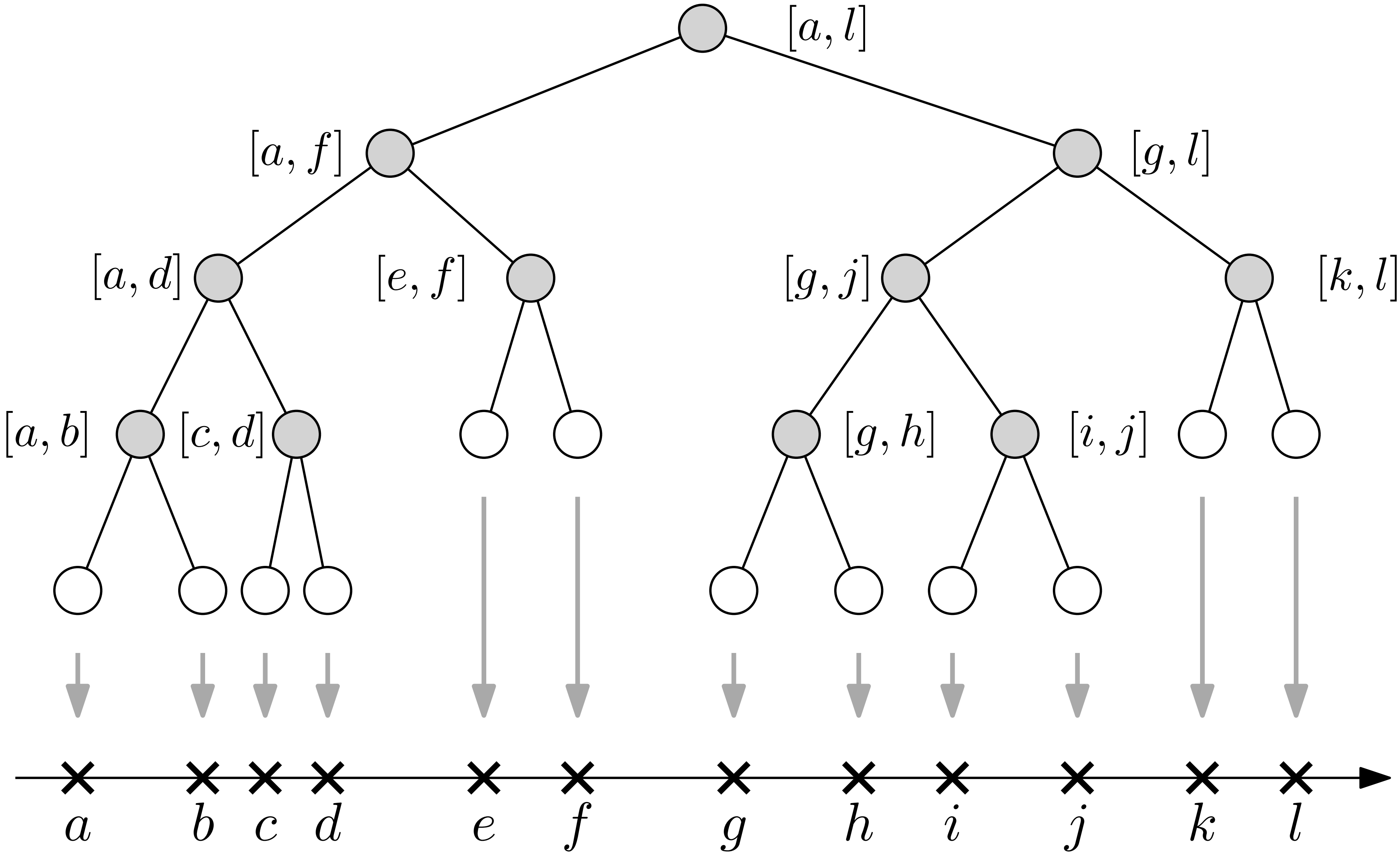
Range Trees: $D = 1$



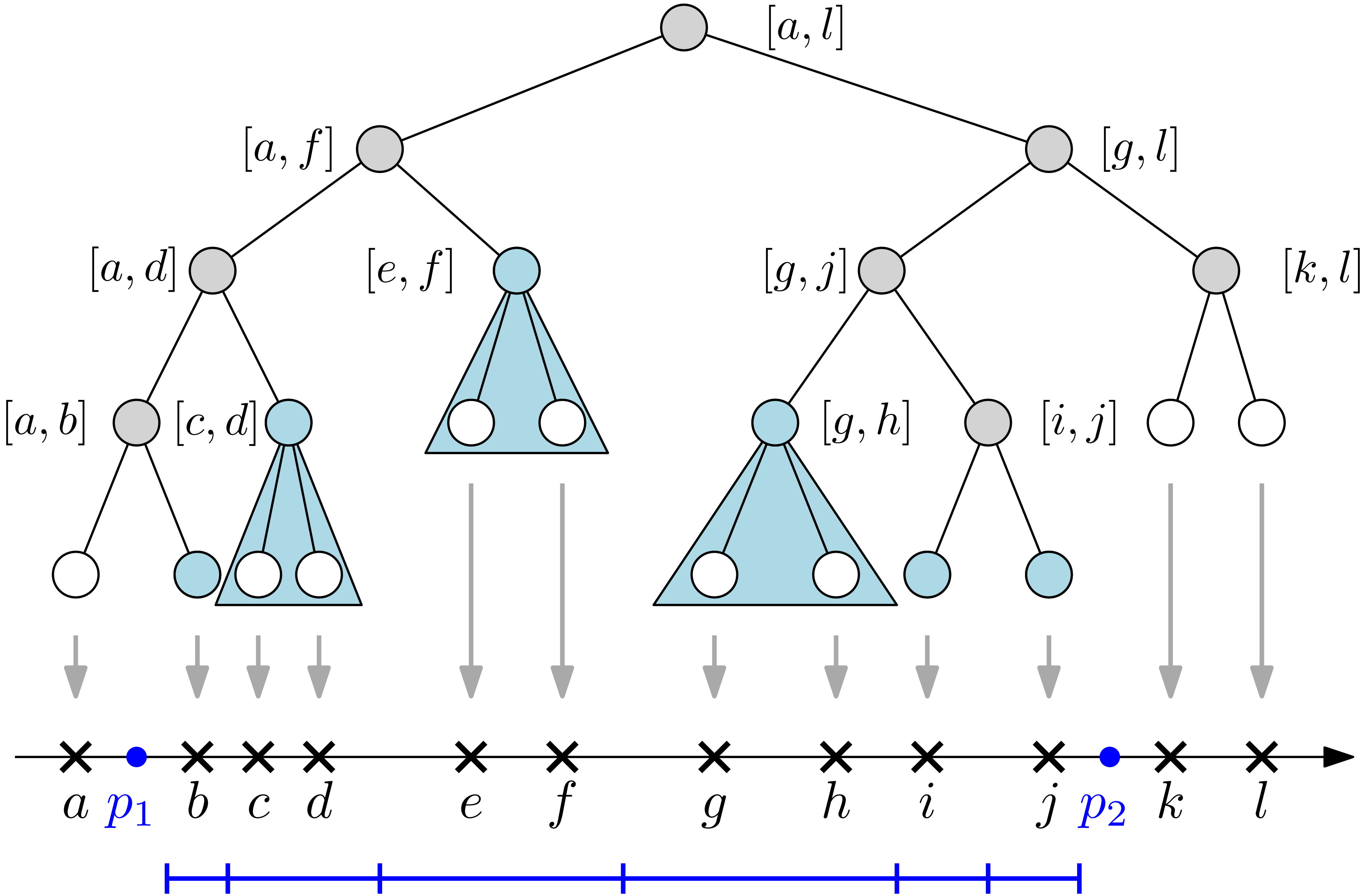
Range Trees: $D = 1$



Range Trees: $D = 1$



Range Trees: $D = 1$



Range Trees: $D = 1$

Construction:

- **Preliminarily** sort S (only once!)
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each. $O(1)$
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root of T has T_1 and T_2 as its left and right subtrees.
- Return T

Range Trees: $D = 1$

Construction:

- **Preliminarily** sort S (only once!)
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each. $O(1)$
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root of T has T_1 and T_2 as its left and right subtrees.
- Return T

Time: $O(n \log n) + T(n)$, where $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$

$$O(n \log n)$$

Range Trees: $D = 1$

Construction:

- **Preliminarily** sort S (only once!)
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each. $O(1)$
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root of T has T_1 and T_2 as its left and right subtrees.
- Return T

Time: $O(n \log n) + T(n)$, where $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$

$$O(n \log n)$$

What if S is already sorted?

Range Trees: $D = 1$

Construction:

- **Preliminarily** sort S (only once!)
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each. $O(1)$
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root of T has T_1 and T_2 as its left and right subtrees.
- Return T

Time: $O(n \log n) + T(n)$, where $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$

$$O(n \log n)$$

What if S is already sorted? $O(n)$ (we will need this later)

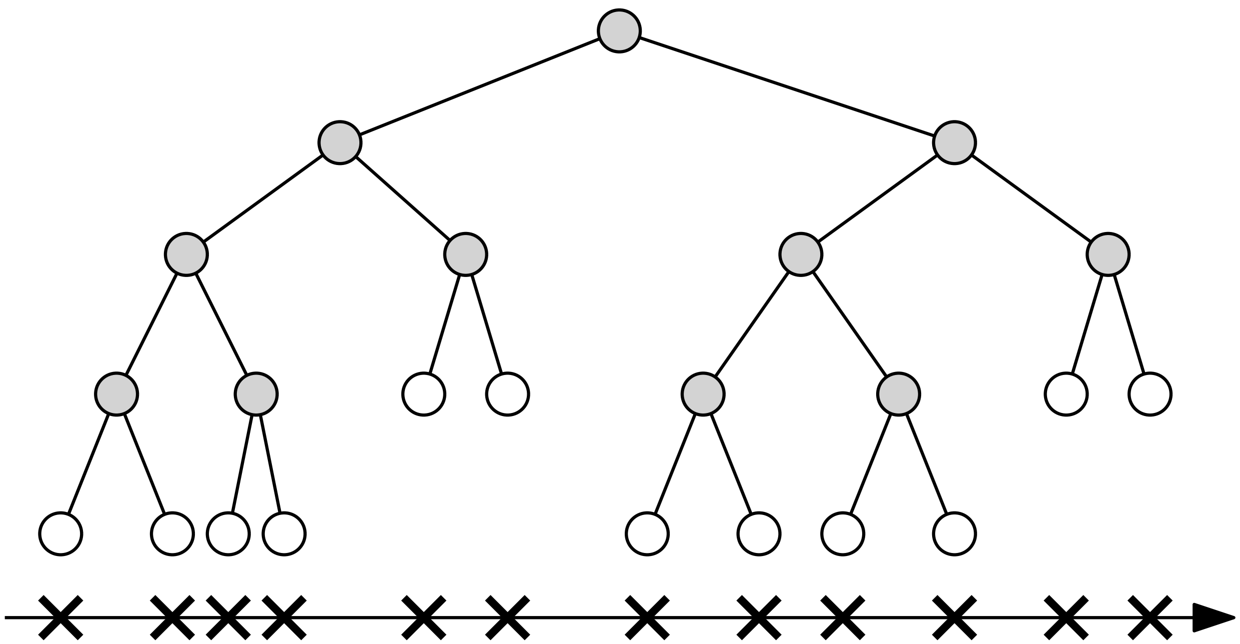
Range Trees: $D = 1$

Preprocessing time: $O(n \log n)$

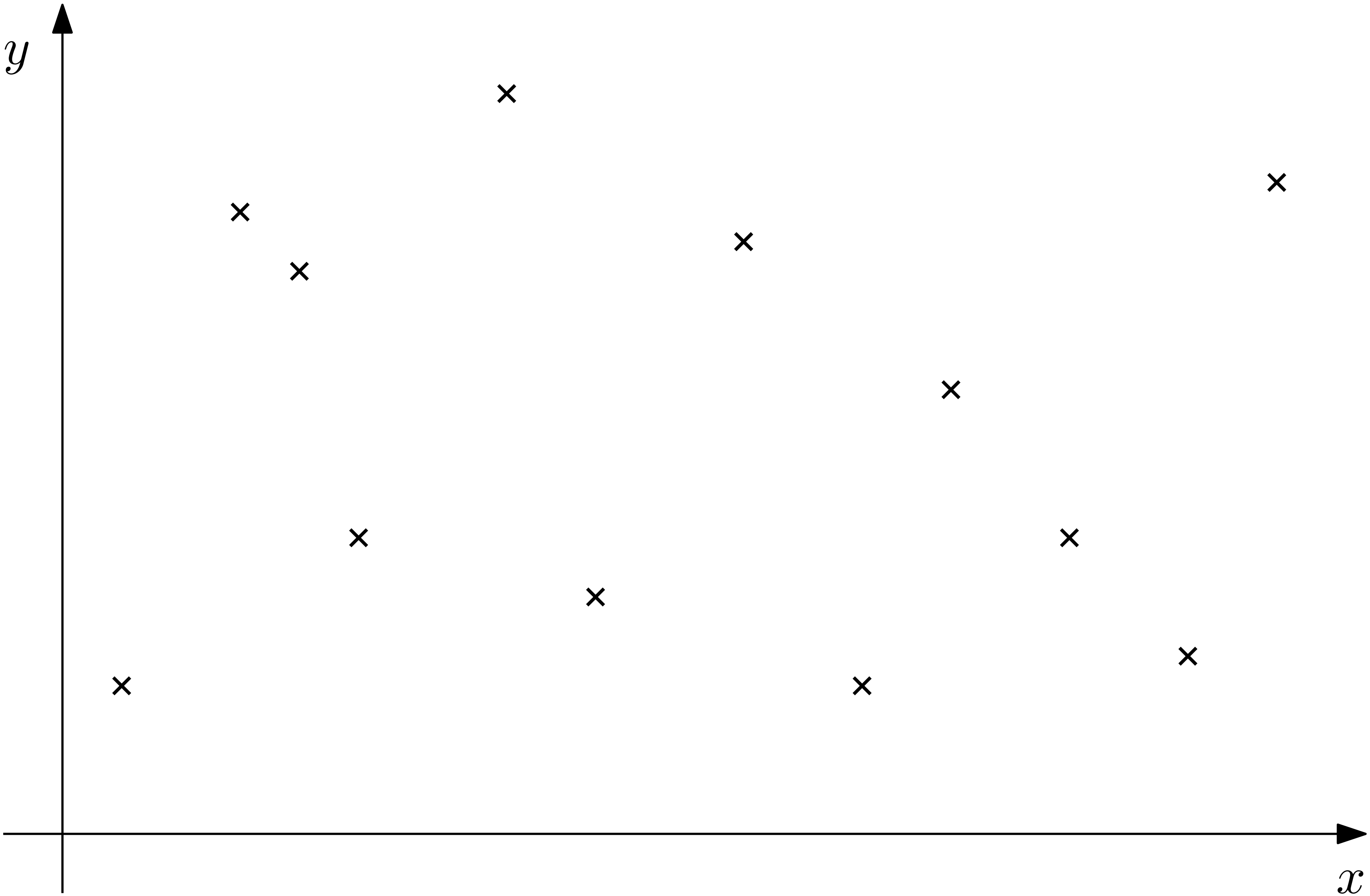
Query time: $O(\log n + k)$

- $k = \#$ reported points.
- $k = \Theta(1)$ if we only care about the *number* of points.

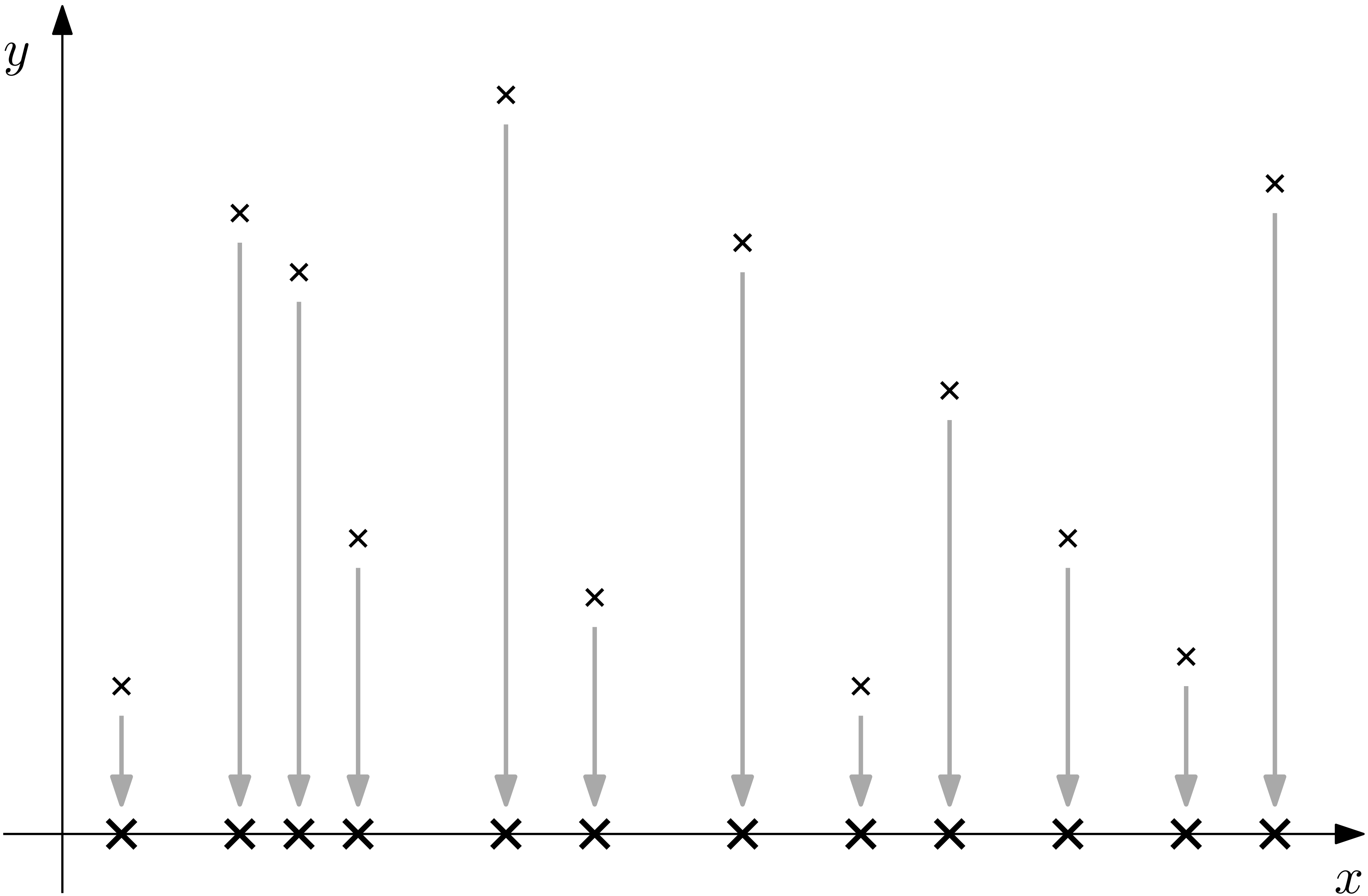
Space complexity: $O(n)$



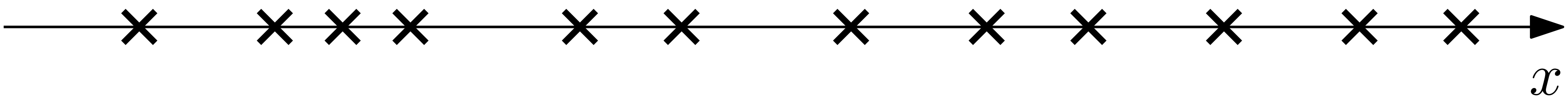
Range Trees: $D = 2$



Range Trees: $D = 2$

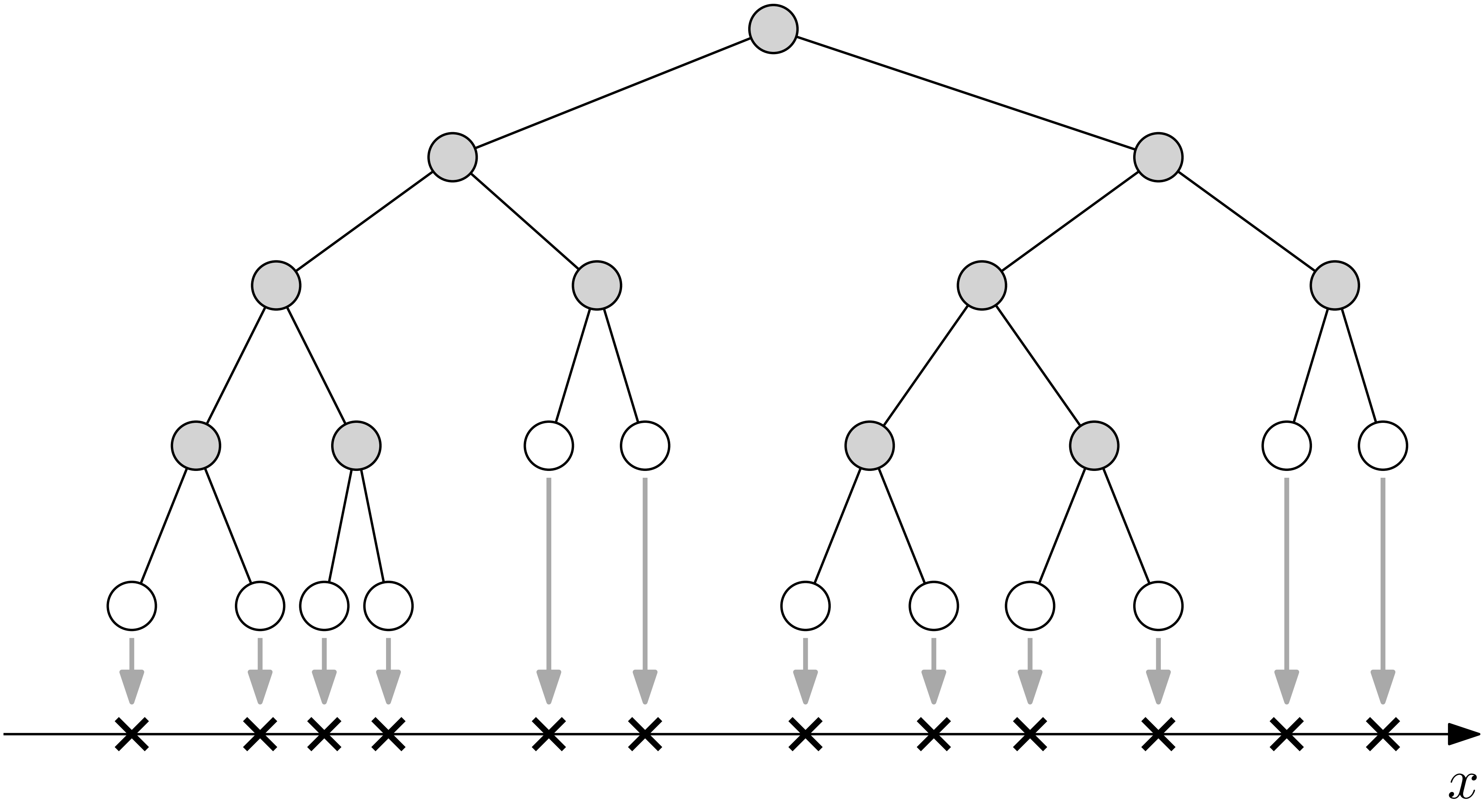


Range Trees: $D = 2$



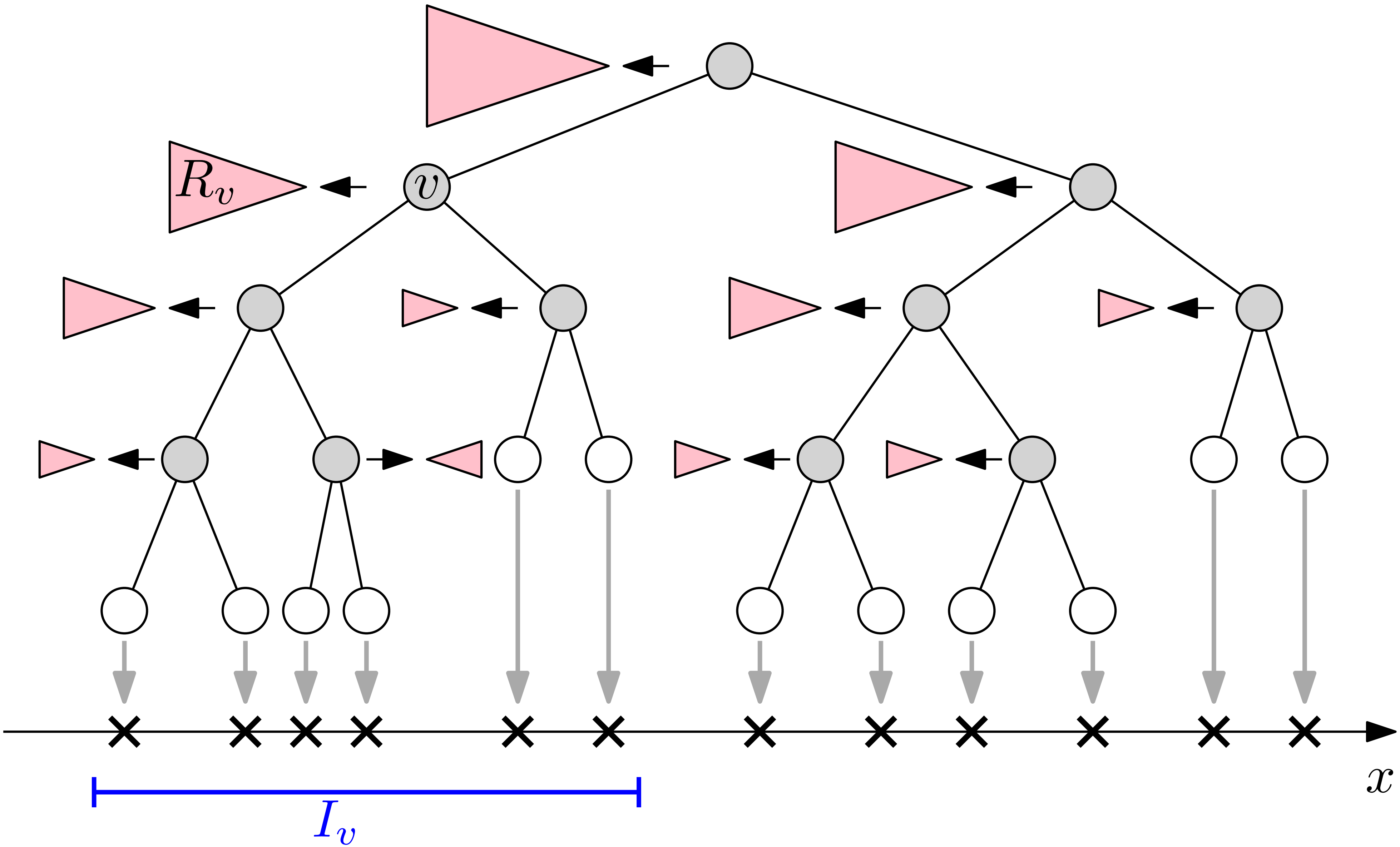
Range Trees: $D = 2$

Build a range tree on the set of x-coordinates of the points in S

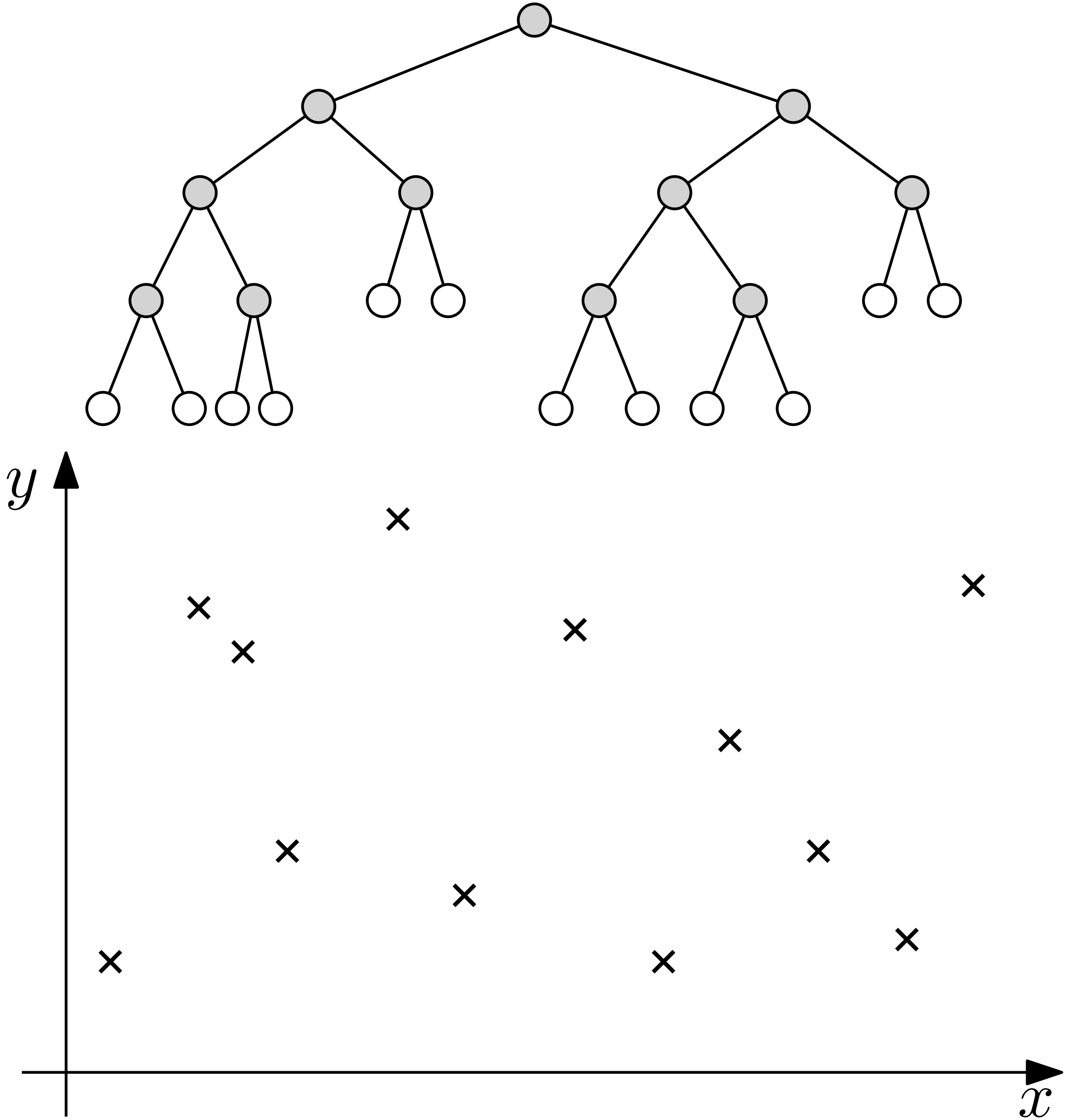


Range Trees: $D = 2$

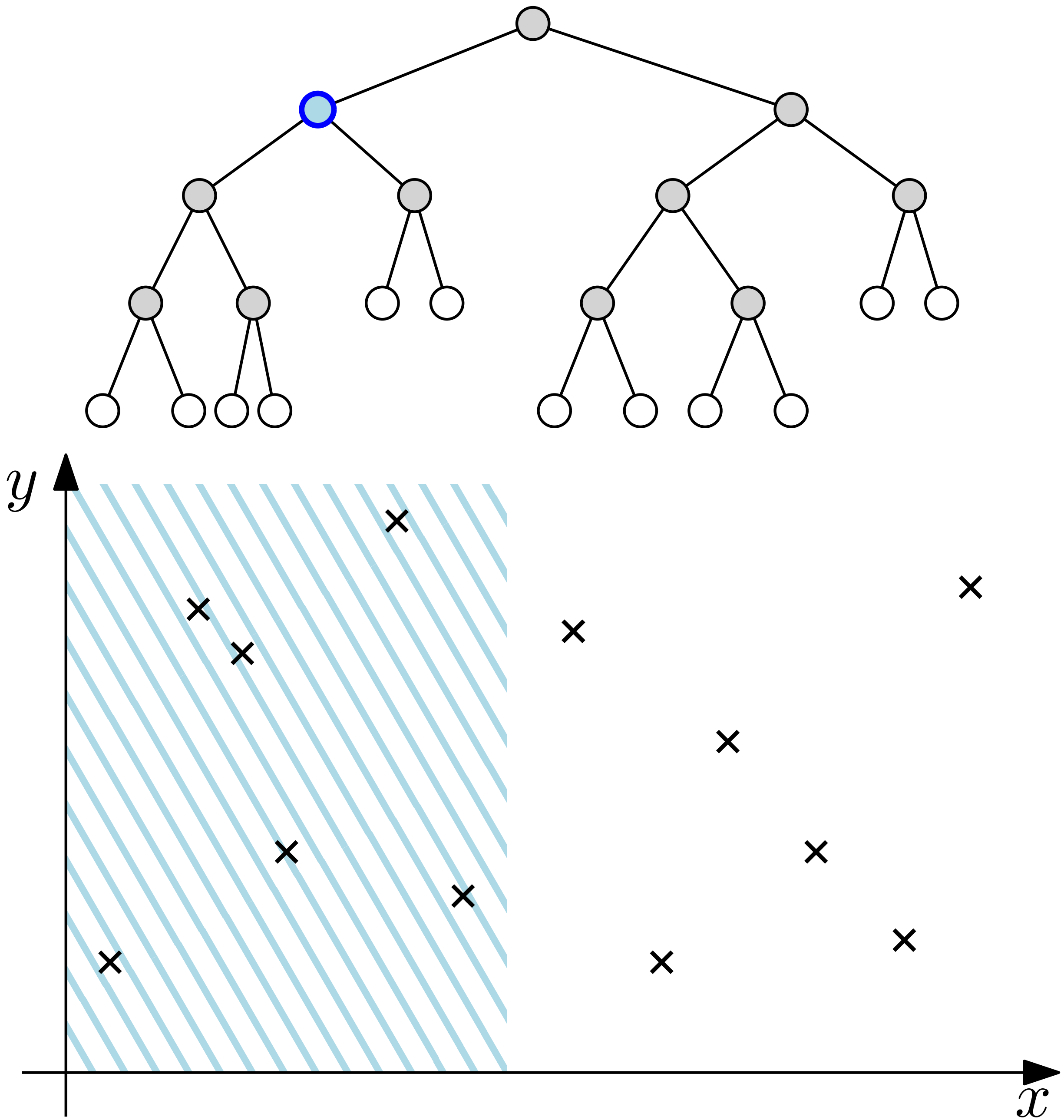
For each node v representing an interval $I_v = [x_1, x_2]$, build a range tree R_v on the y coodinates of the points in S with x-coordinate in I_v



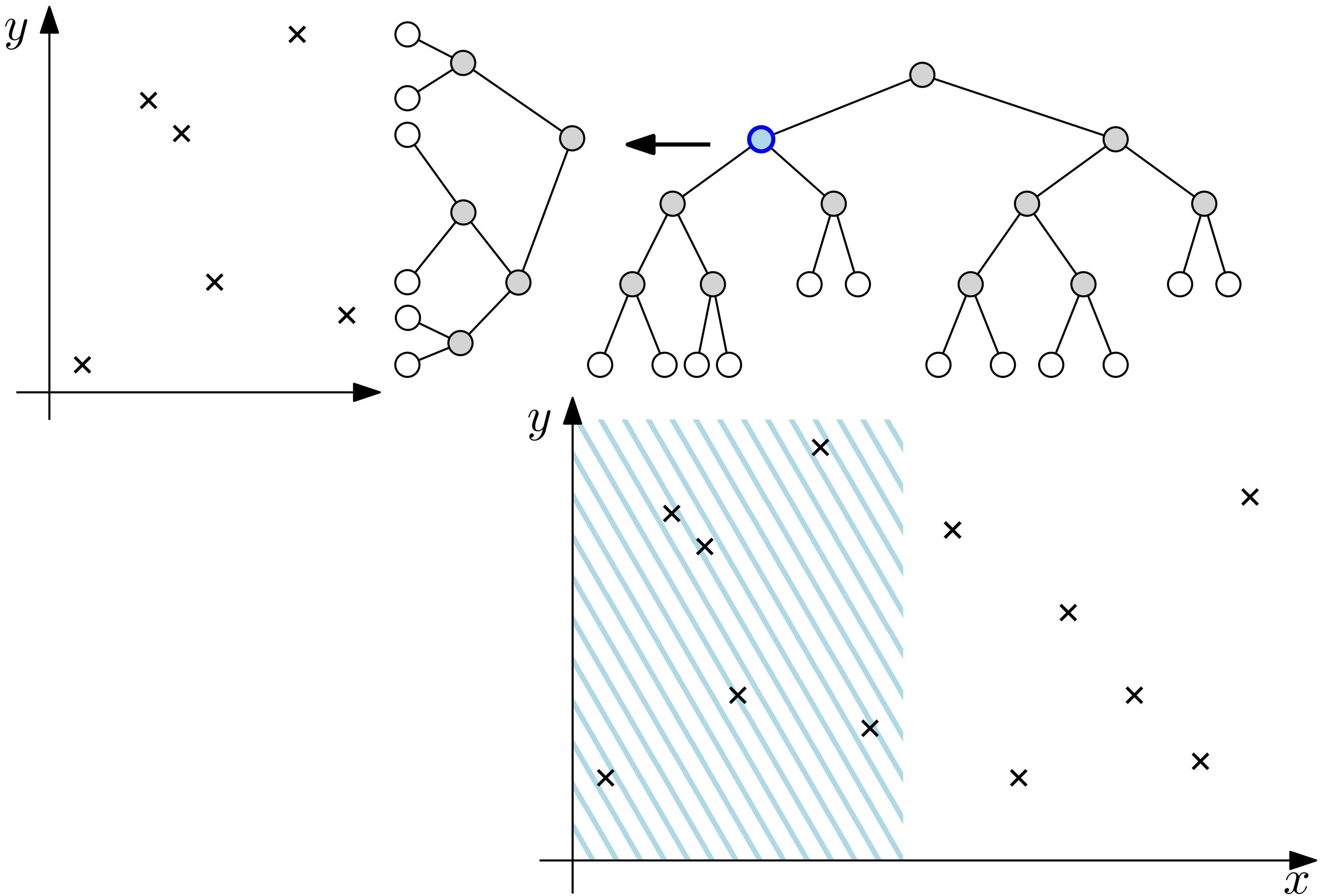
Range Trees: $D = 2$



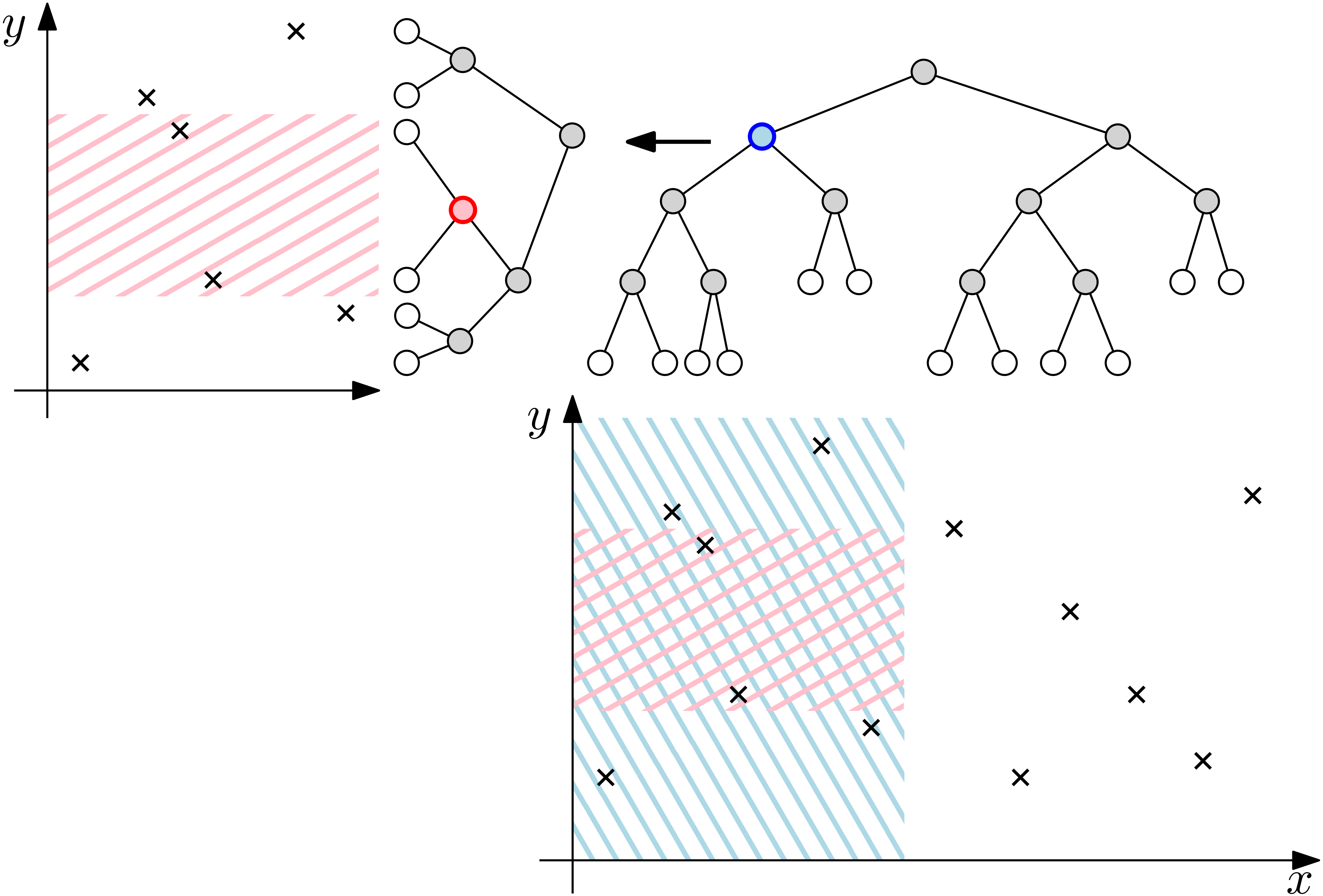
Range Trees: $D = 2$



Range Trees: $D = 2$



Range Trees: $D = 2$



Range Trees: $D = 2$

Construction:

- **Preliminarily** sort S on the x-coordinate.
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each.
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root v of T has T_1 and T_2 as its left and right subtrees.
- Store, in v, a pointer to a new 1D Range Tree on S
- Return T

Range Trees: $D = 2$

Construction:

- **Preliminarily** sort S on the x-coordinate.
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each.
- Recursively build T_1 and T_2 from S_1 and S_2, respectively.
- The root v of T has T_1 and T_2 as its left and right subtrees.
- Store, in v, a pointer to a new 1D Range Tree on S
- Return T

Time: $O(n \log n) + T(n)$, where $T(n) = 2 \cdot T(\frac{n}{2}) + O(n \log n)$

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

Range Trees: $D = 2$

S^y is the set S sorted on the y-coordinate

Construction:

- **Preliminarily** sort S on the x-coordinate.
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each.
- Recursively build (T_1, S_1^y) and (T_2, S_2^y) from S_1 and S_2, respectively.
- The root v of T has T_1 and T_2 as its left and right subtrees.
- Merge S_1^y and S_2^y into S^y.
- Store, in v, a pointer to a new 1D Range Tree on S^y
- Return (T, S^y)

Range Trees: $D = 2$

S^y is the set S sorted on the y-coordinate

Construction:

- **Preliminarily** sort S on the x-coordinate.
- Split S into S_1 and S_2 of $\approx \frac{n}{2}$ elements each.
- Recursively build (T_1, S_1^y) and (T_2, S_2^y) from S_1 and S_2, respectively.
- The root v of T has T_1 and T_2 as its left and right subtrees.
- Merge S_1^y and S_2^y into S^y.
- Store, in v, a pointer to a new 1D Range Tree on S^y
- Return (T, S^y)

Time: $O(n \log n) + T(n)$, where $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$

$$O(n \log n)$$

Range Trees: $D = 2$

To report the points $p_1 = (x_1, y_1) \leq q \leq p_2 = (x_2, y_2)$:

- Use T to find the $h = O(\log n)$ subtrees $R_1, \ldots, R_h$ that store the points $q = (x, y)$ with $x_1 \leq x \leq x_2$.

- For each tree $R_j \in \{R_1, \ldots, R_h\}$ representing the x-interval I_j:
  - Query R_j to report the number of/set of points $q = (x, y)$ with $x \in I_j$ and $y_1 \leq y \leq y_2$.

Range Trees: $D = 2$

To report the points $p_1 = (x_1, y_1) \leq q \leq p_2 = (x_2, y_2)$:

- Use T to find the $h = O(\log n)$ subtrees $R_1, \ldots, R_h$ that store the points $q = (x, y)$ with $x_1 \leq x \leq x_2$.

- For each tree $R_j \in \{R_1, \ldots, R_h\}$ representing the x-interval I_j:
  - Query R_j to report the number of/set of points $q = (x, y)$ with $x \in I_j$ and $y_1 \leq y \leq y_2$.

Time complexity:

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

Number of R_is   Time to query R_i "size" of the output

Range Trees: $D = 2$

Preprocessing time: $O(n \log n)$

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

- $k = \#$ reported points.
- $k = \Theta(1)$ if we only care about the *number* of points.

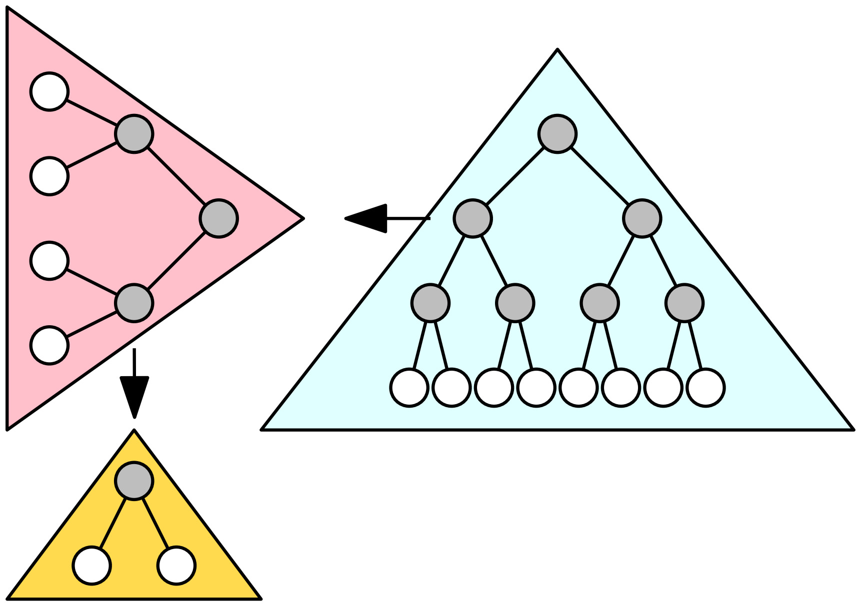
Space complexity:

- Bounded by the overall size of 1D Range Trees
- Each point belongs to $O(\log n)$ 1D Range Tees
- Total space: $O(n \log n)$

Higher dimensions: construction

To store points $p = (x, y, z, w, ...)$ in $D > 2$ dimensions:
Recursive construction:

- Build a Range Tree T on the first coordinate x of the points:
- For each subtree T_v of T associated with the interval $I_v = [x_1, x_2]$:
  - Construct a range tree R_v on the last $D - 1$ coordinates $(y, z \ldots)$ of the set of points $p = (x, y, \ldots)$ with $x \in I_v$.
  - Store, in v, a pointer to R_v.



Time: $O(n \log^{D-1} n)$.

Space: $O(n \log^{D-1} n)$.

Higher dimensions: query

Let $p_1 = (x_1, y_1, z_1, \dots)$, $p_2 = (x_2, y_2, z_2, \dots)$.

To report the points $p_1 \leq q \leq p_2$:

- Use T to find the $h = O(\log n)$ subtrees $R_1, \dots, R_h$ that store the points $q = (x, y, z, \dots)$ with $x_1 \leq x \leq x_2$.
- For each tree $R_j \in \{R_1, \dots, R_h\}$ representing the x-interval I_j:
  - Recursively query R_i to report the number/set of points q s.t. $x \in I_j$ and $(y_1, z_1, \dots) \leq q \leq (y_2, z_2, \dots)$.

Query time: $O(\log^D n + k)$.

Recap

D	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	

Fractional Cascading

Fractional Cascading: The problem

Input:

k sorted arrays $A_1, \ldots, A_k$ of n elements each:

A_1	4	9	15	22	23	38	41	50	53	58
A_2	3	7	10	11	15	17	20	36	62	64
A_3	21	23	29	35	37	40	52	57	61	66
A_4	2	5	6	15	24	27	39	50	54	76

Query:

Given x report, for $i = 1, \ldots, k$, x if $x \in A_i$ or its *predecessor* if $x \notin A_i$.

Fractional Cascading: The problem

Input:

k sorted arrays $A_1, \ldots, A_k$ of n elements each:

A_1	4	9	15	22	**23**	38	41	50	53	58
A_2	3	7	10	11	15	17	**20**	36	62	64
A_3	21	23	**29**	35	37	40	52	57	61	66
A_4	2	5	6	15	24	**27**	39	50	54	76

$x = 31$

Query:

Given x report, for $i = 1, \ldots, k$, x if $x \in A_i$ or its *predecessor* if $x \notin A_i$.

Fractional Cascading: The problem

Input:

k sorted arrays $A_1, \ldots, A_k$ of n elements each:

A_1	4	9	15	22	23	38	41	50	53	$\boxed{58}$
A_2	3	7	10	11	15	17	20	$\boxed{36}$	62	64
A_3	21	23	29	35	37	40	52	$\boxed{57}$	61	66
A_4	2	5	6	15	24	27	39	50	$\boxed{54}$	76

$x = 58$

Query:

Given x report, for $i = 1, \ldots, k$, x if $x \in A_i$ or its *predecessor* if $x \notin A_i$.

Fractional Cascading: A Trivial solution

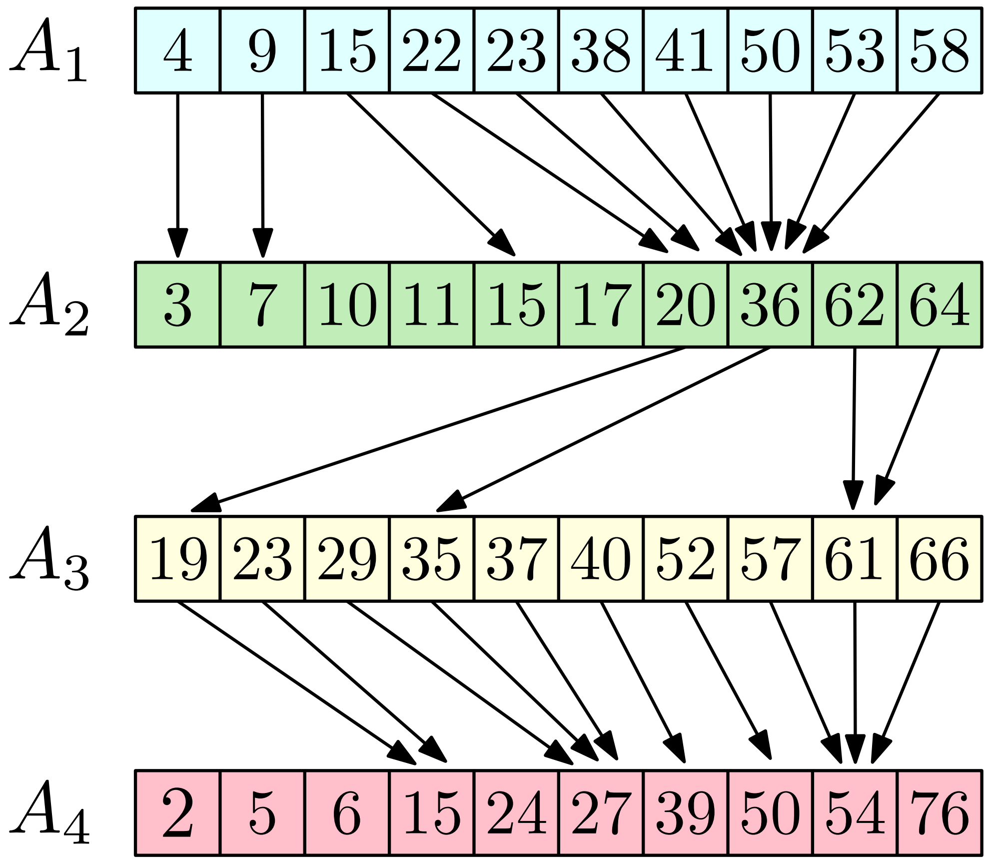
- For $i = 1, \ldots, k$:
  - Binary search for x in A_i

Time: $O(k \log n)$

Fractional Cascading

First idea: cross linking

Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.

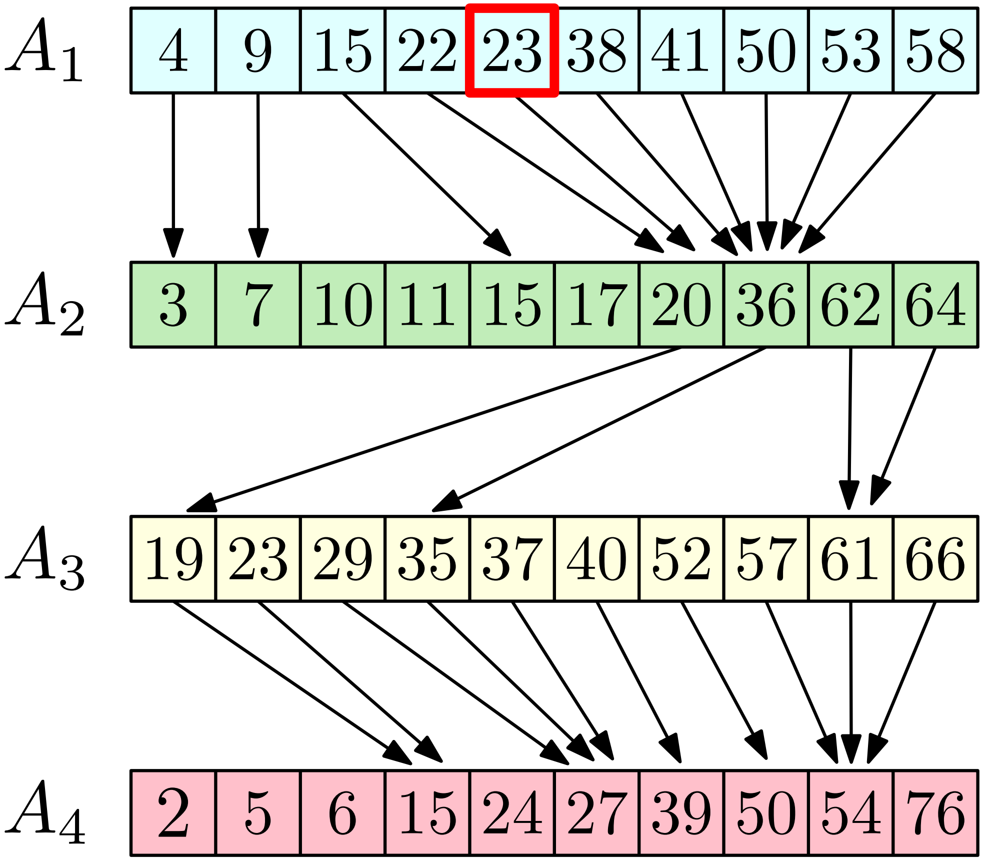


$x = 27$

Fractional Cascading

First idea: cross linking

Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.

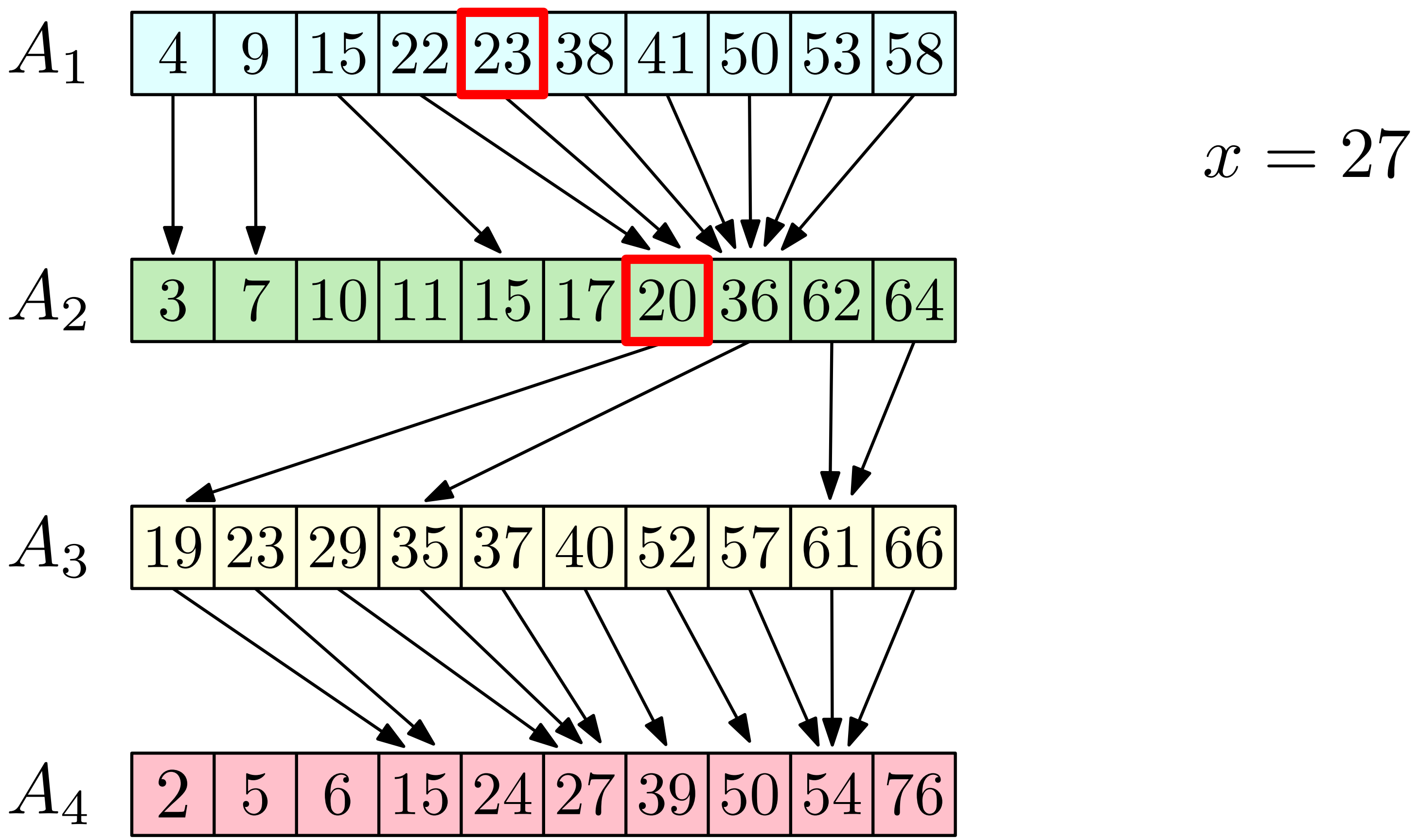


$x = 27$

Fractional Cascading

First idea: cross linking

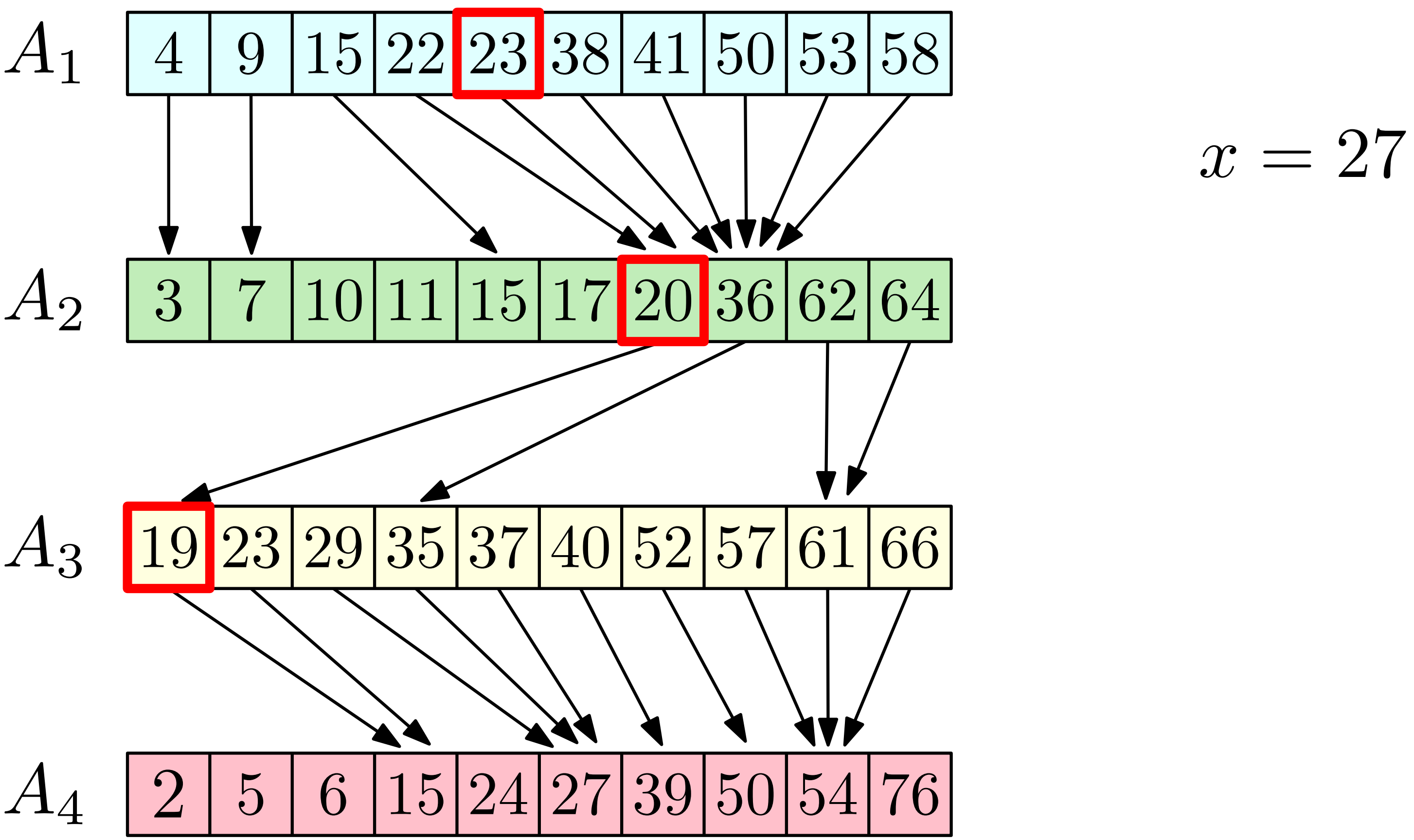
Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.



Fractional Cascading

First idea: cross linking

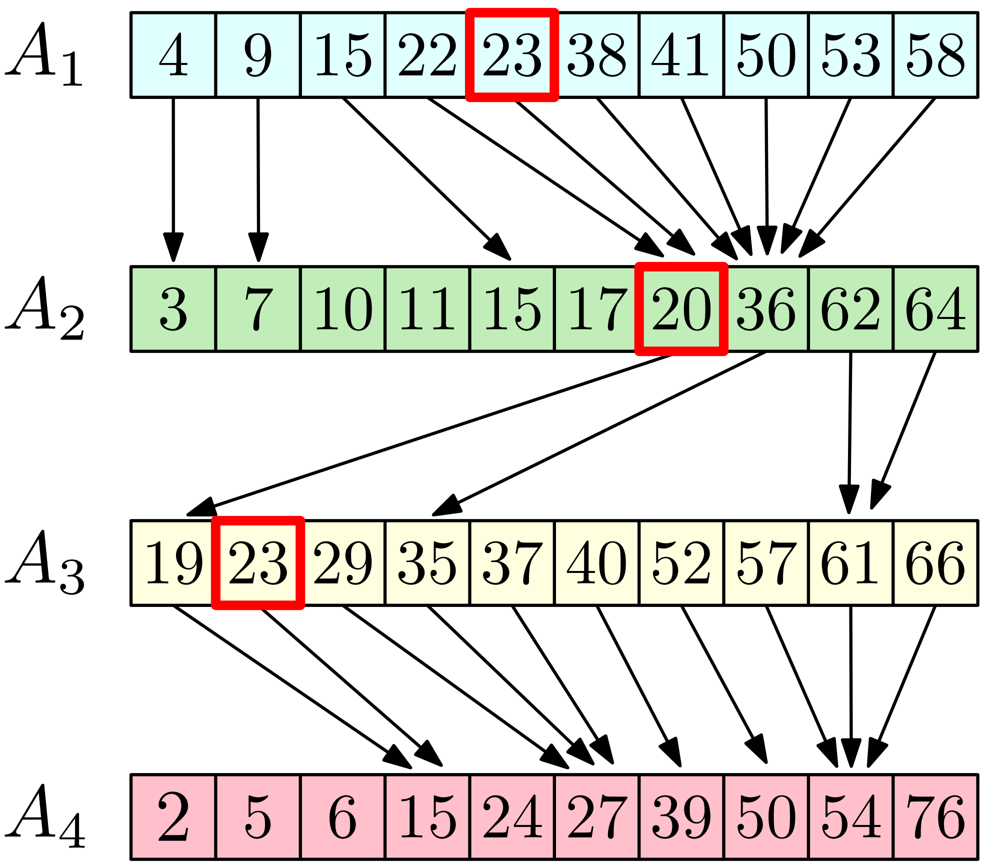
Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.



Fractional Cascading

First idea: cross linking

Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.

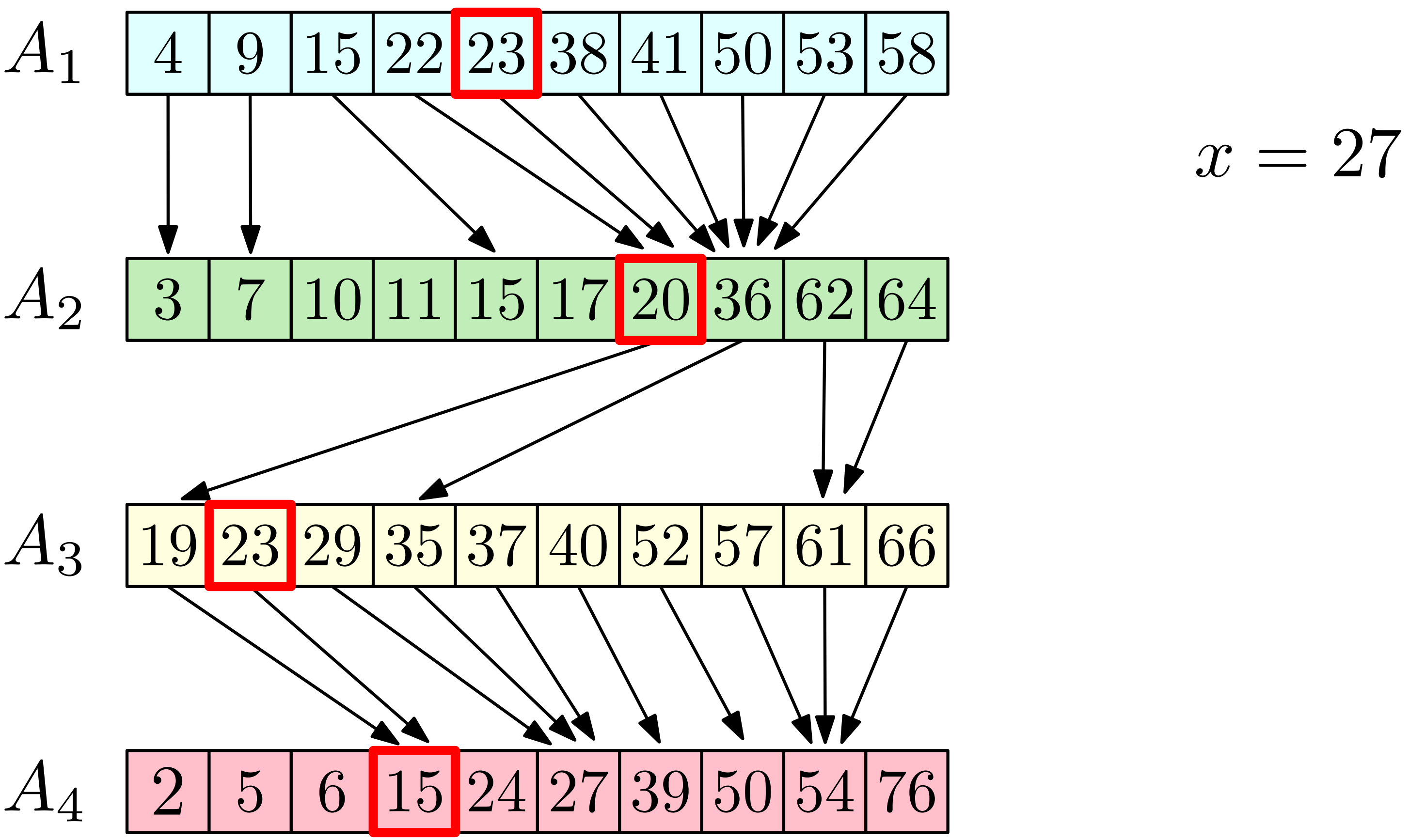


$x = 27$

Fractional Cascading

First idea: cross linking

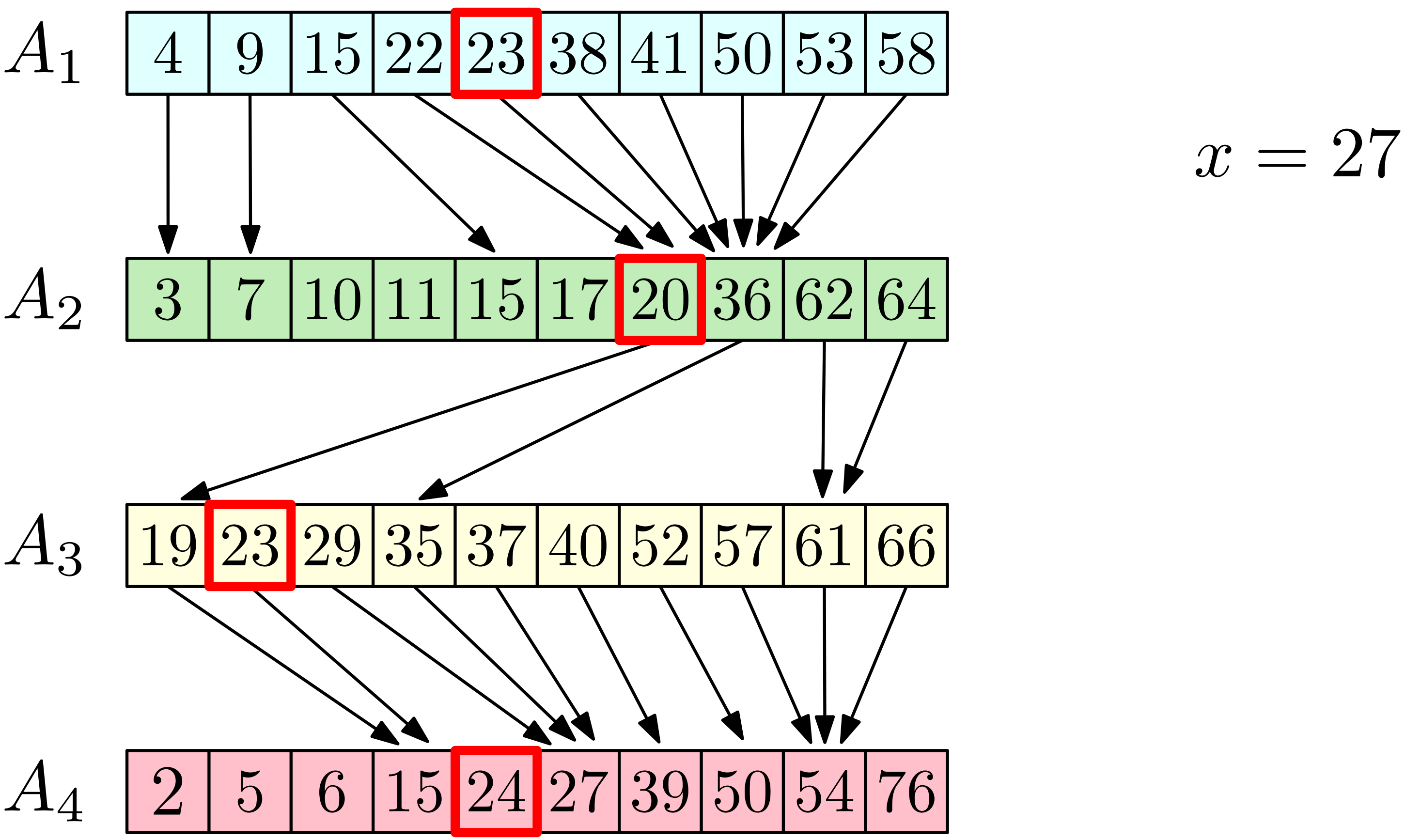
Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.



Fractional Cascading

First idea: cross linking

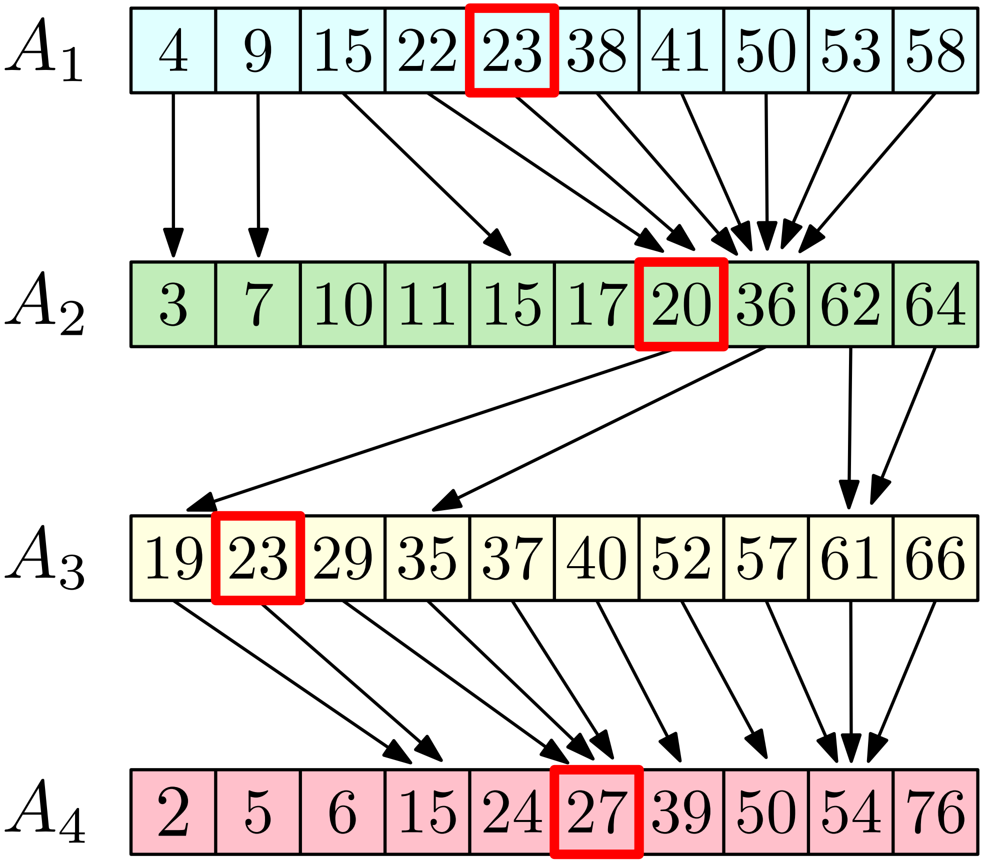
Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.



Fractional Cascading

First idea: cross linking

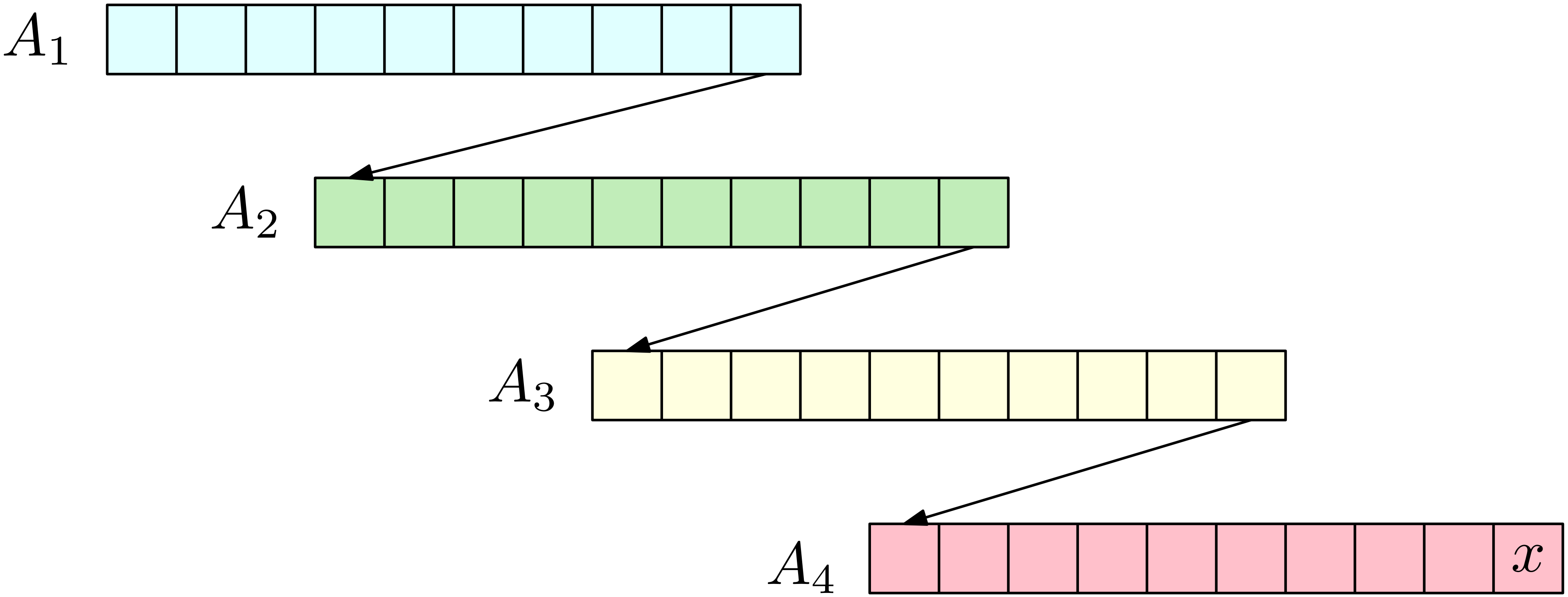
Keep pointers from $A_i[j]$ to the predecessor of $A_i[j]$ in A_{i+1}.



$x = 27$

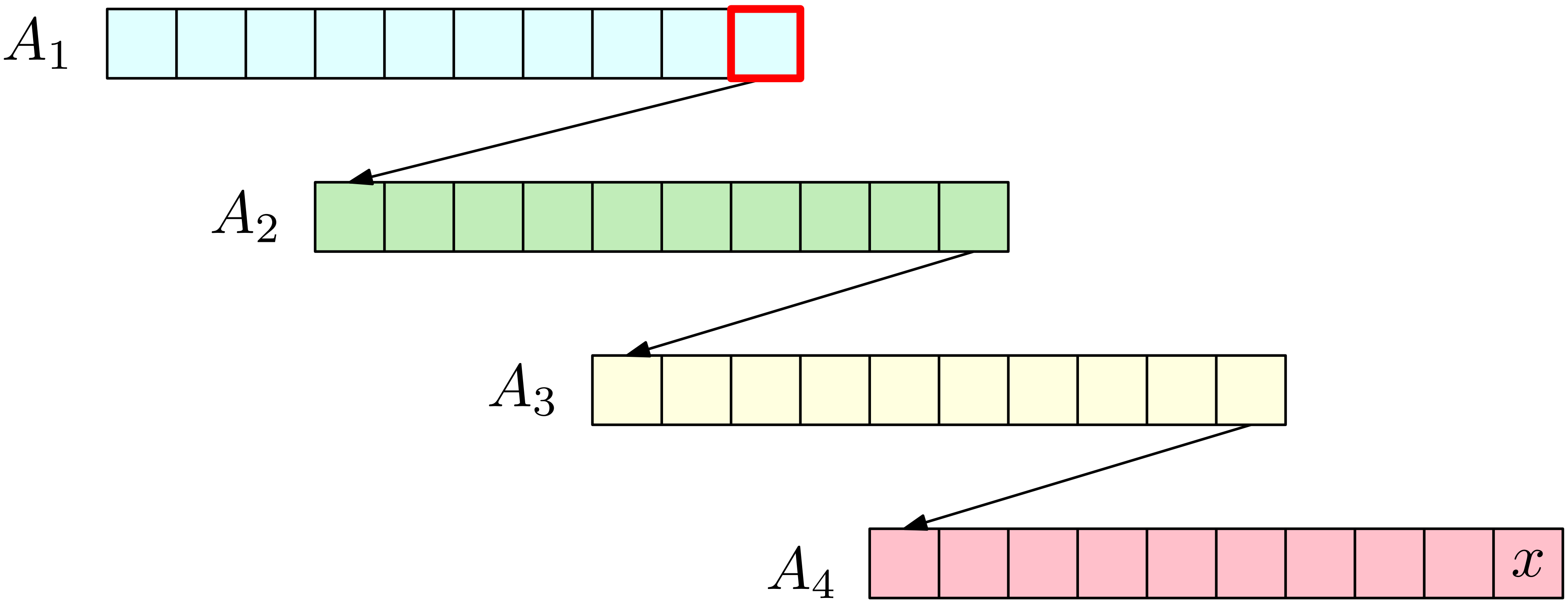
Fractional Cascading

How much time does it take?



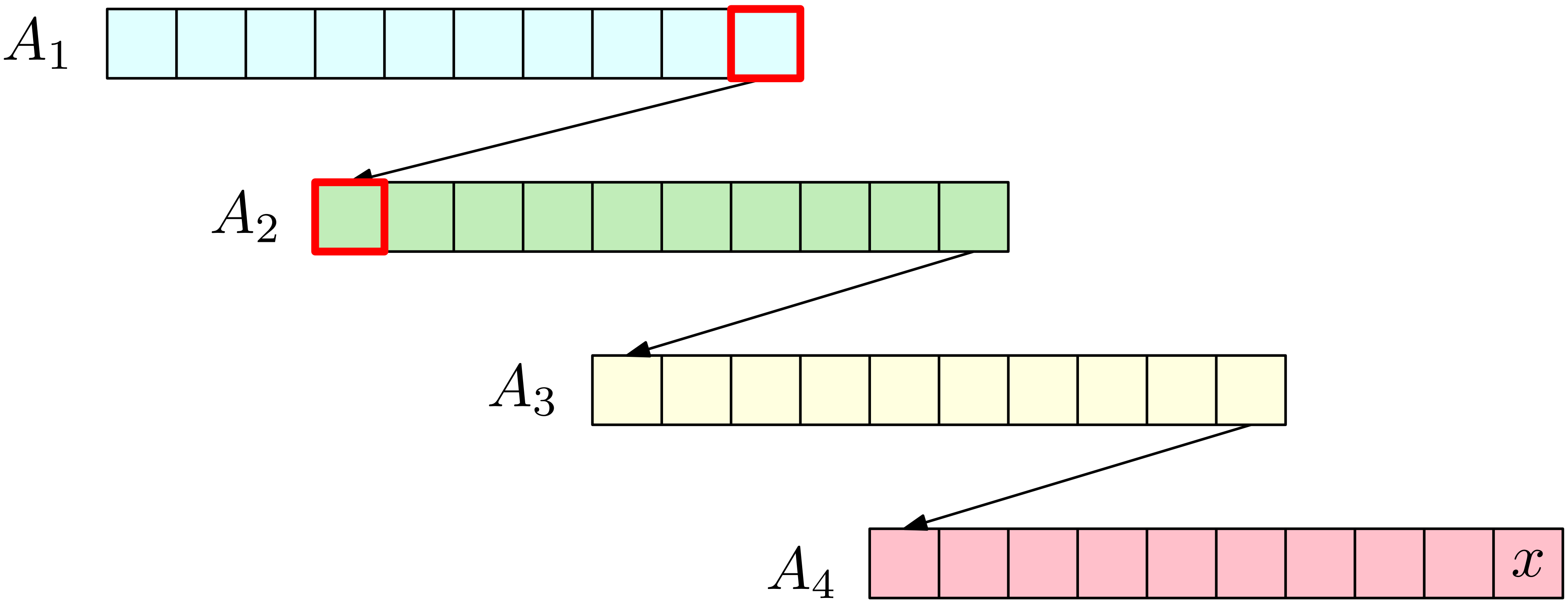
Fractional Cascading

How much time does it take?



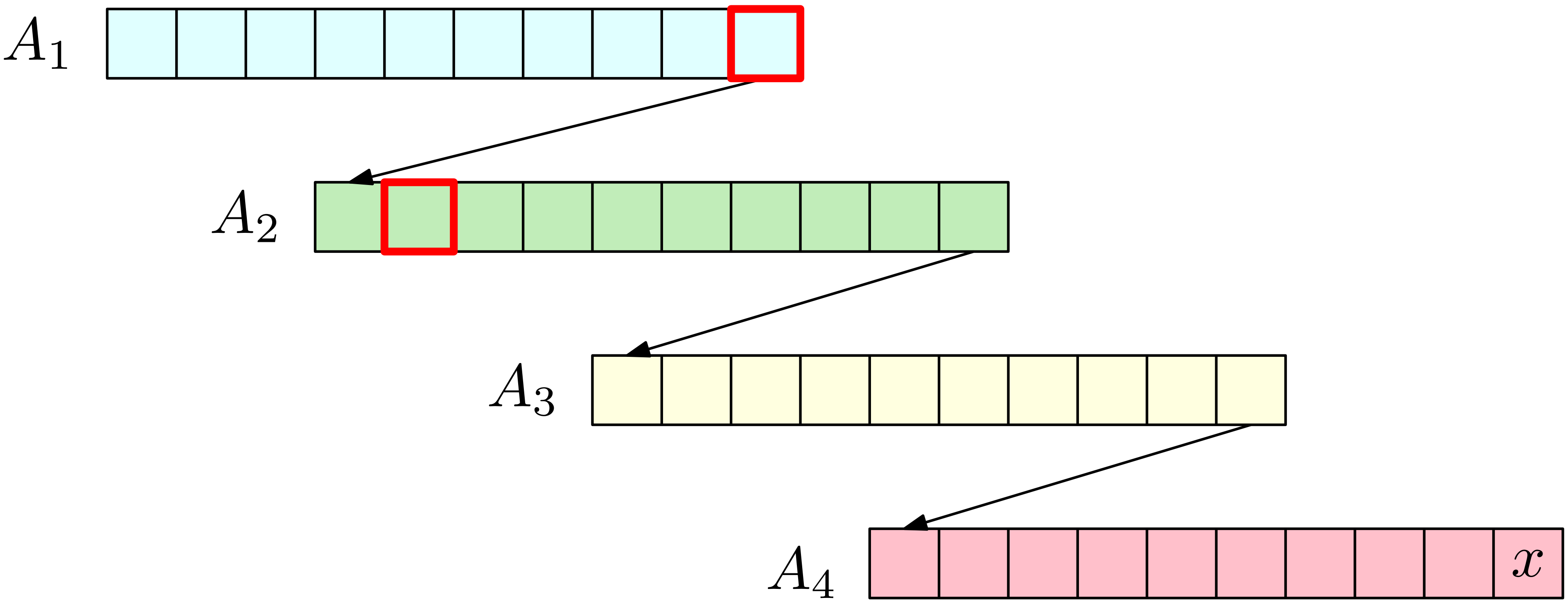
Fractional Cascading

How much time does it take?



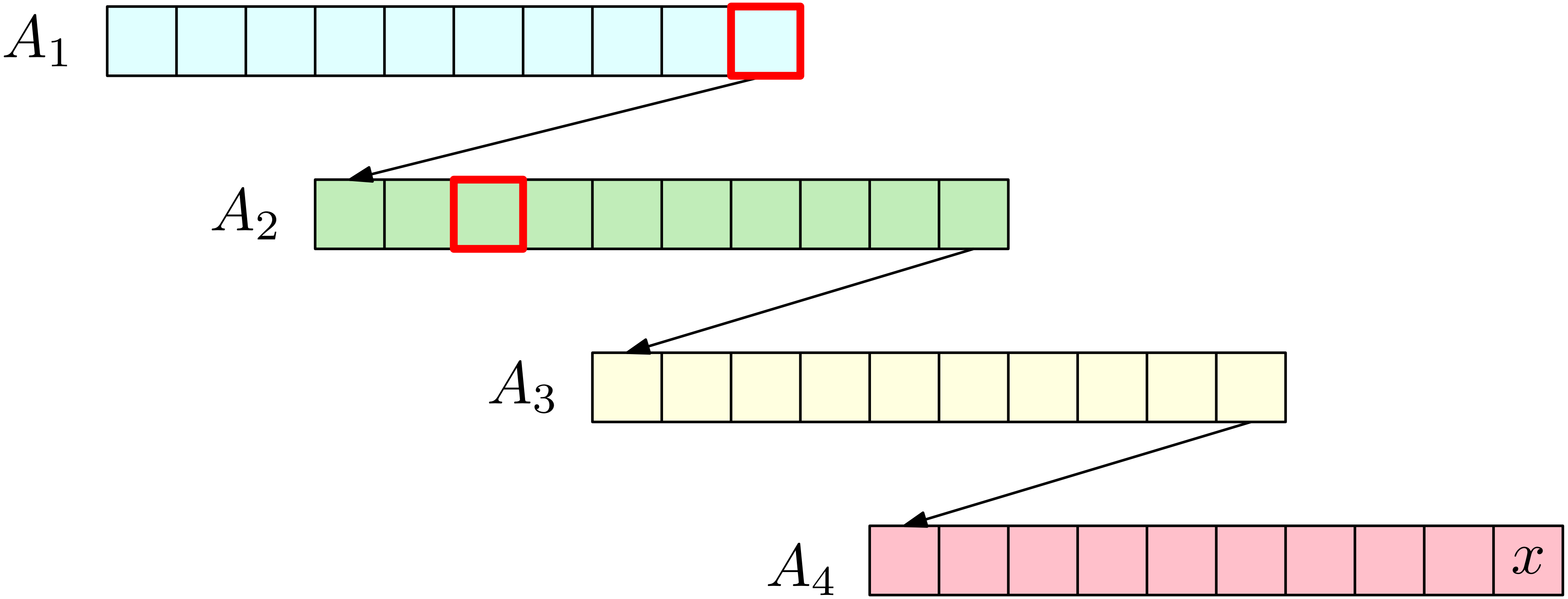
Fractional Cascading

How much time does it take?



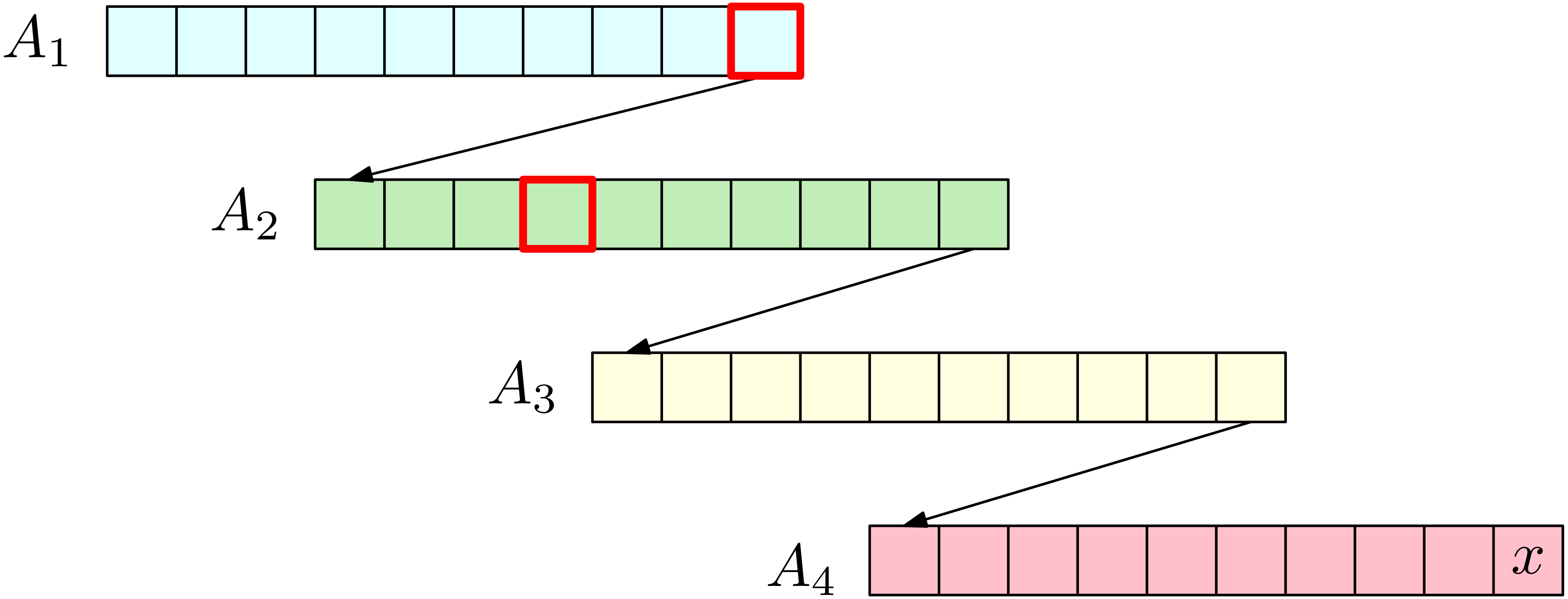
Fractional Cascading

How much time does it take?



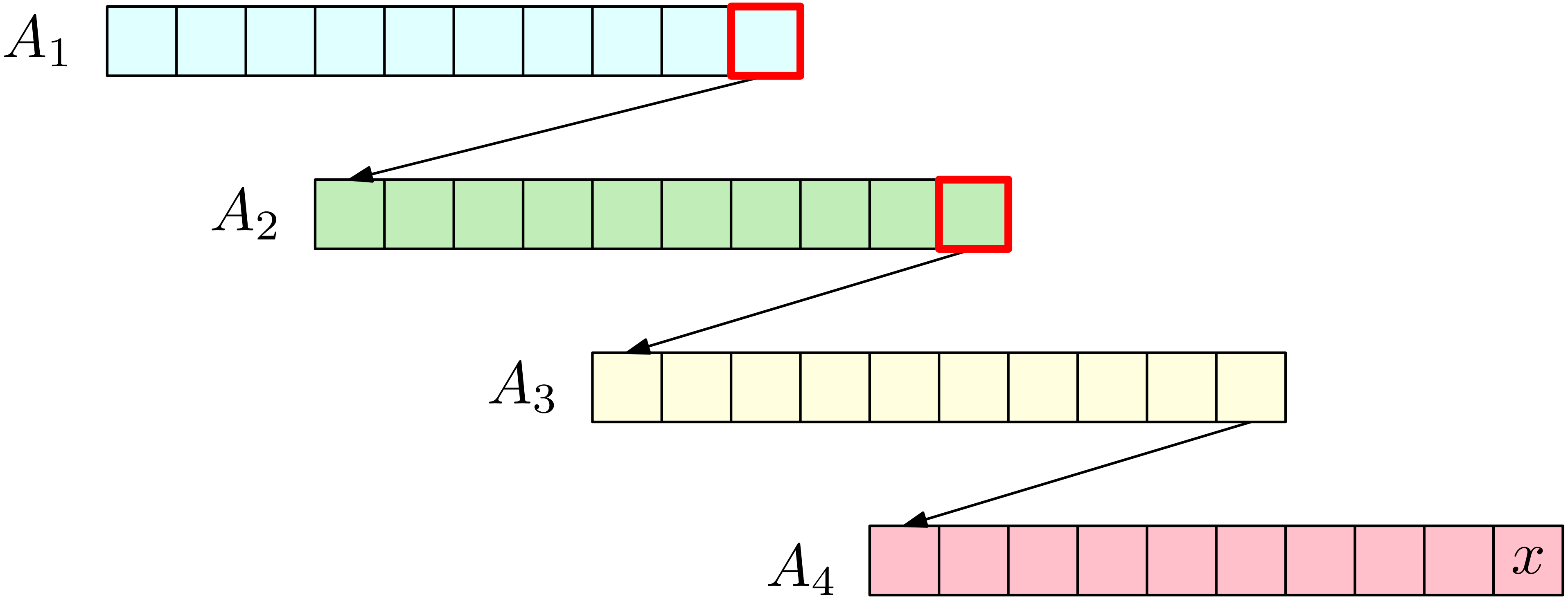
Fractional Cascading

How much time does it take?



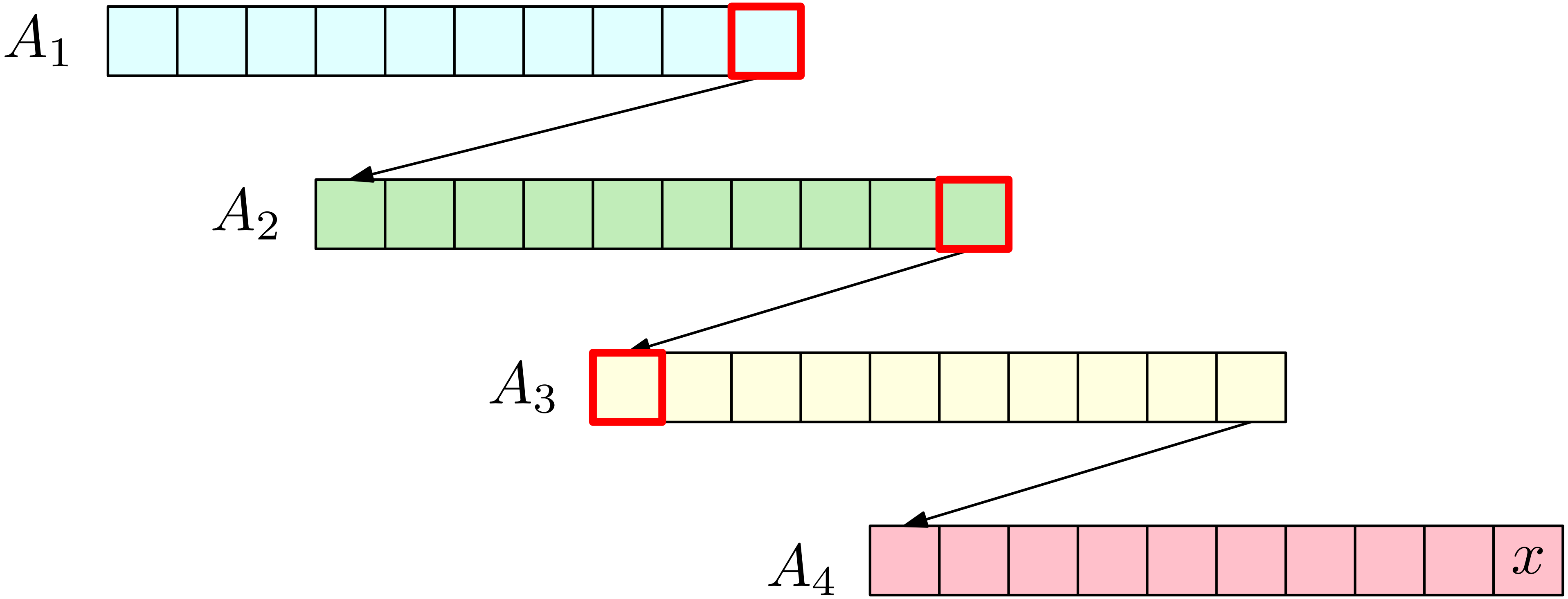
Fractional Cascading

How much time does it take?



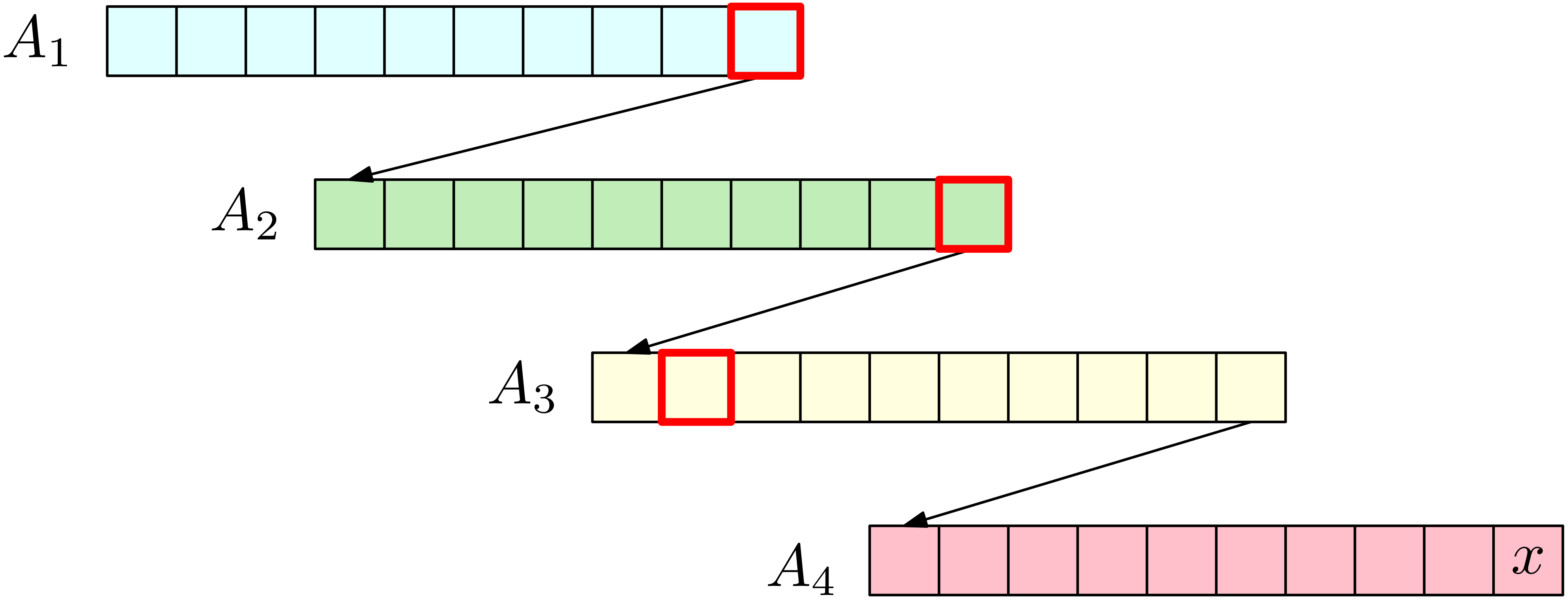
Fractional Cascading

How much time does it take?



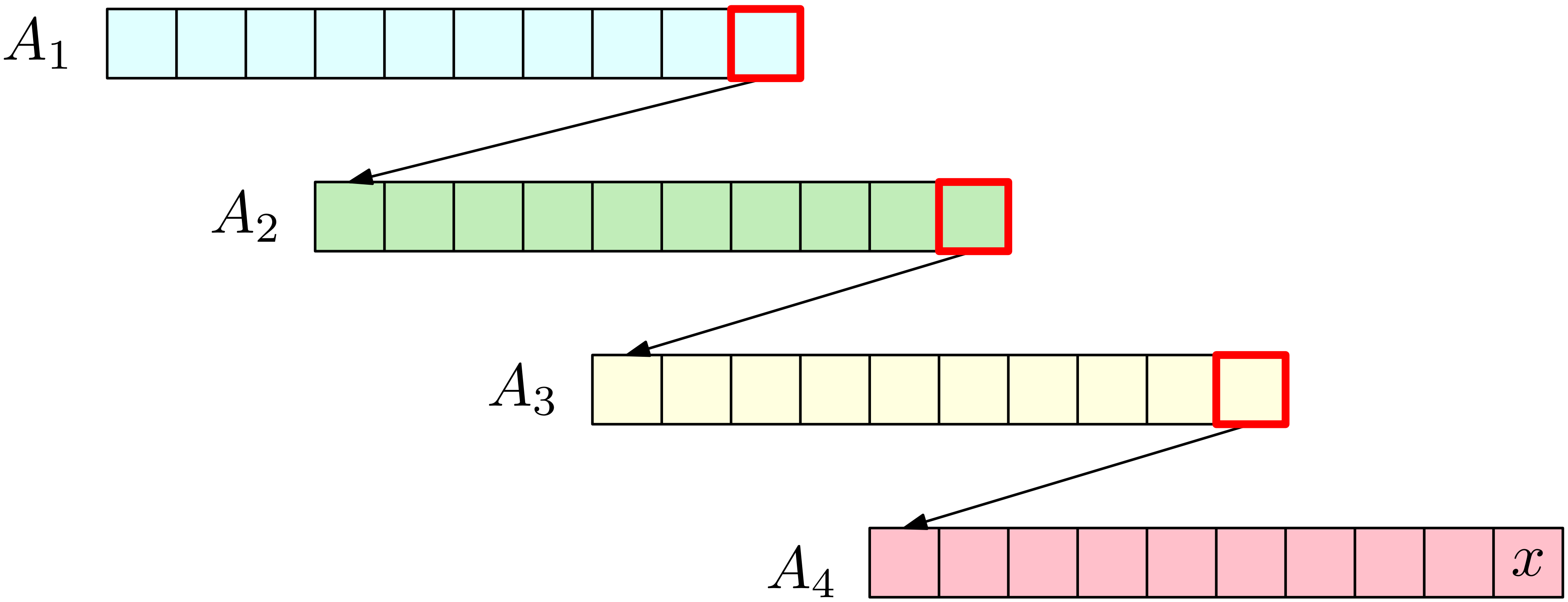
Fractional Cascading

How much time does it take?



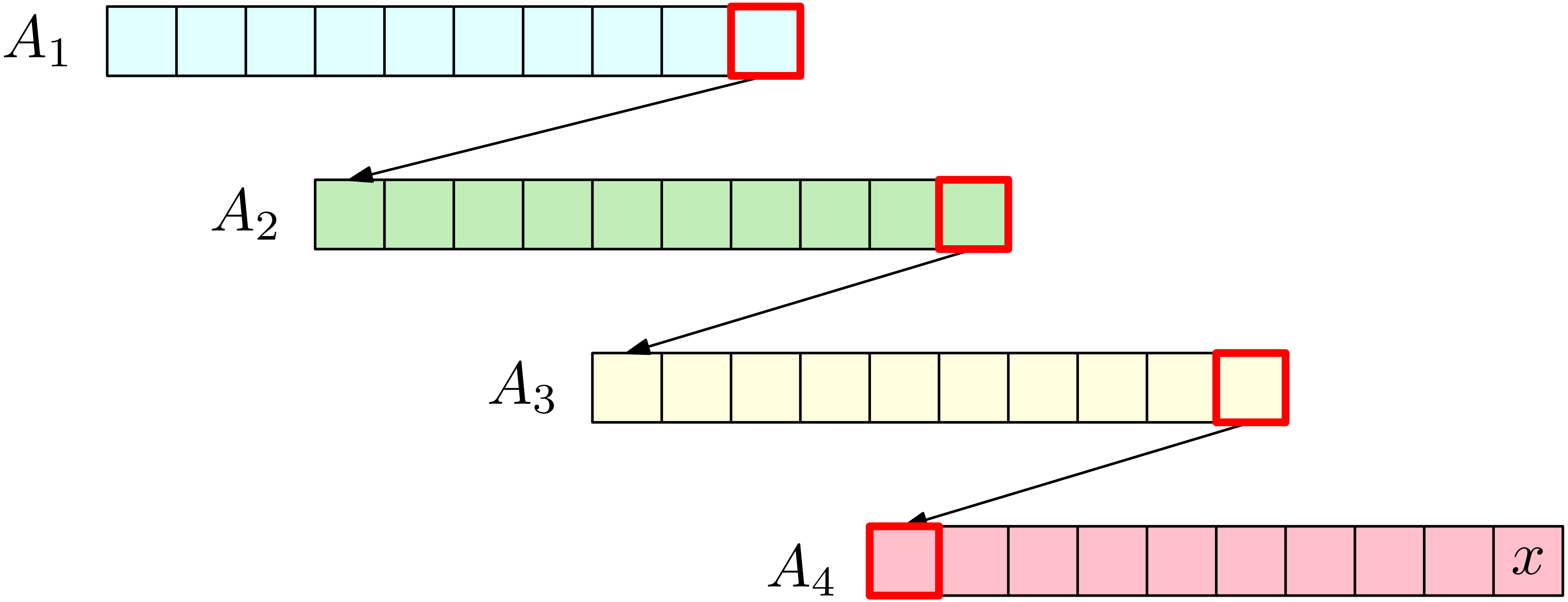
Fractional Cascading

How much time does it take?



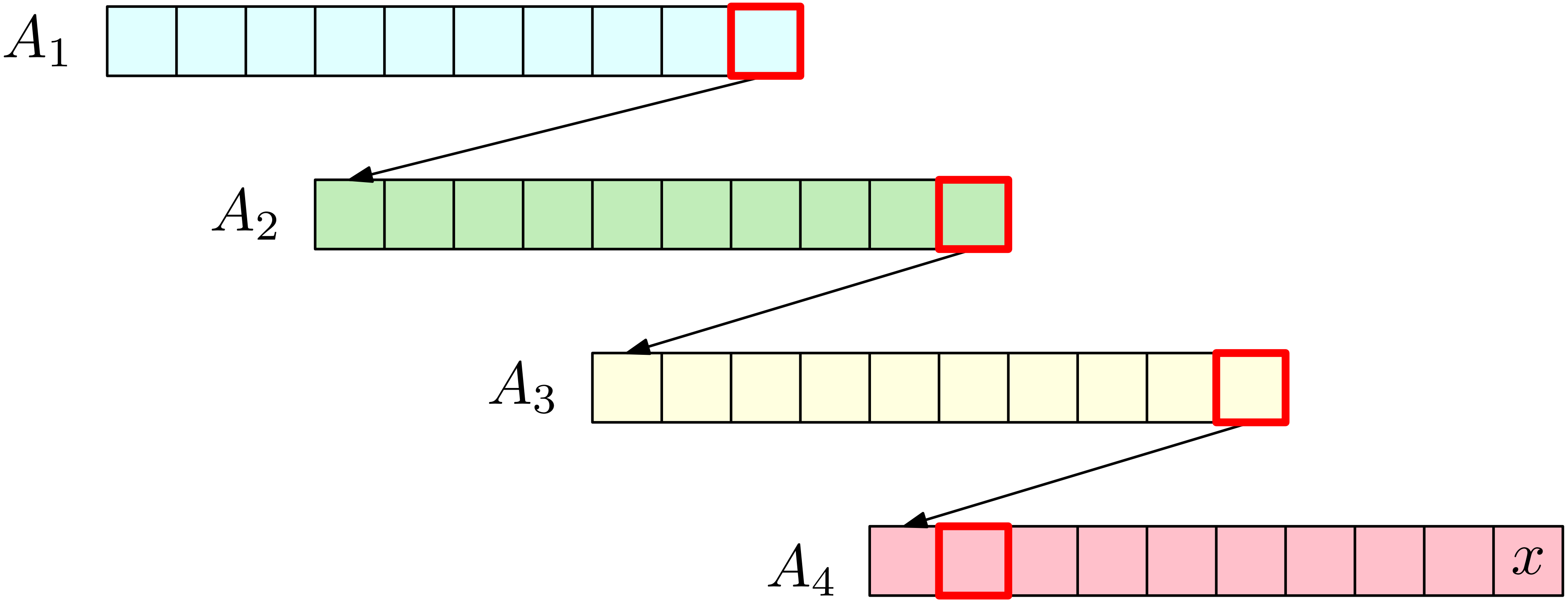
Fractional Cascading

How much time does it take?



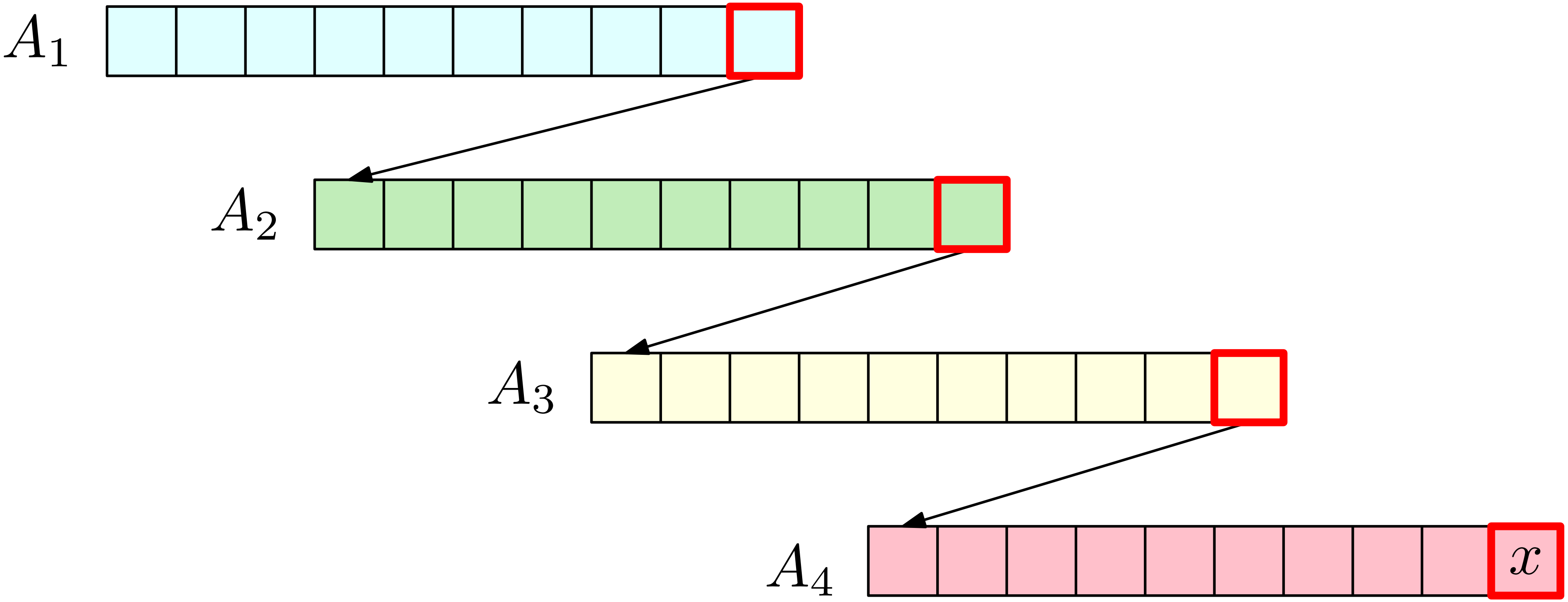
Fractional Cascading

How much time does it take?



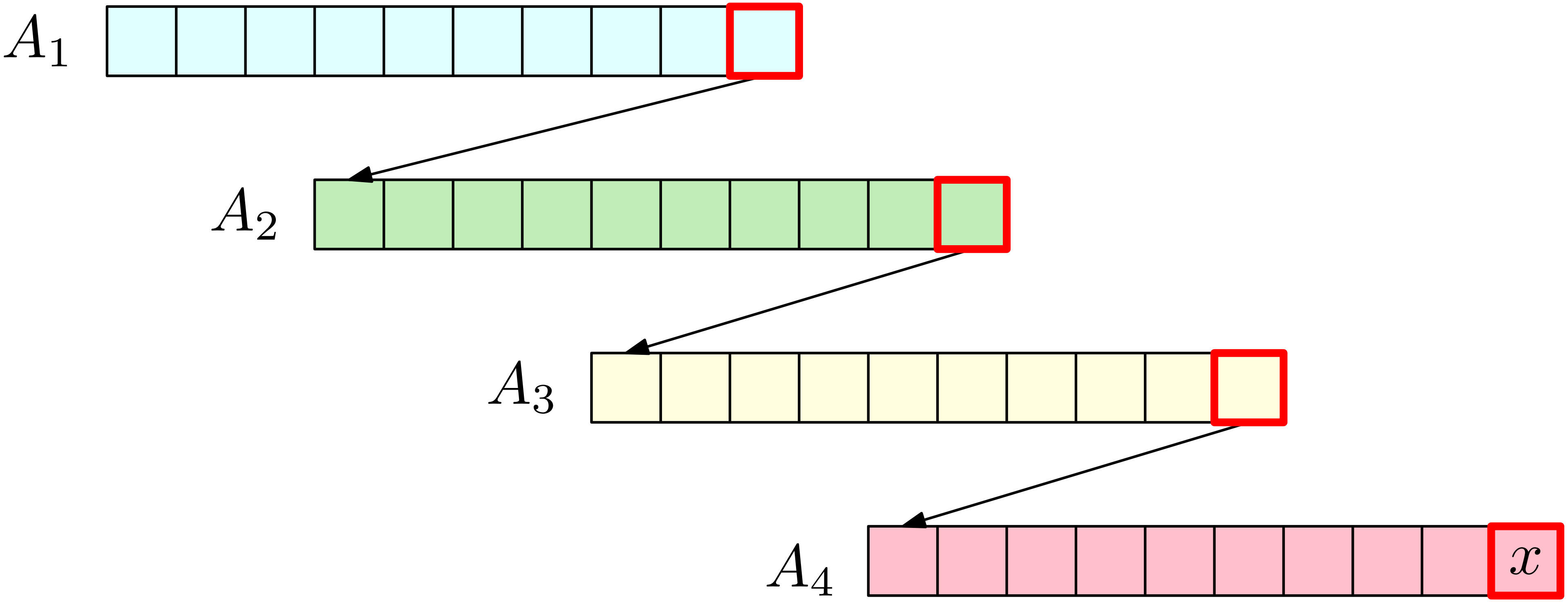
Fractional Cascading

How much time does it take?



Fractional Cascading

How much time does it take?

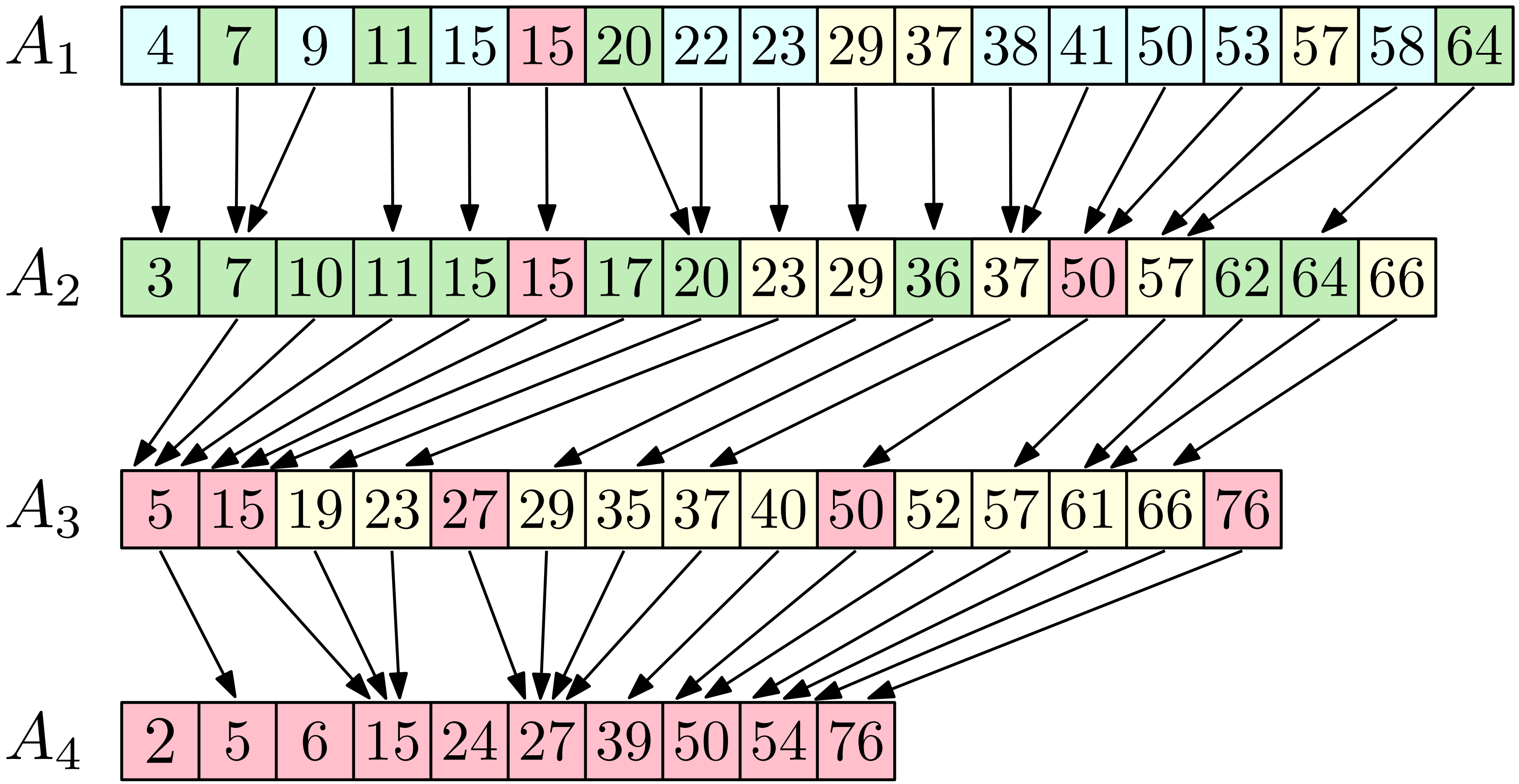


Worst-case time: $O(kn)$

Fractional Cascading

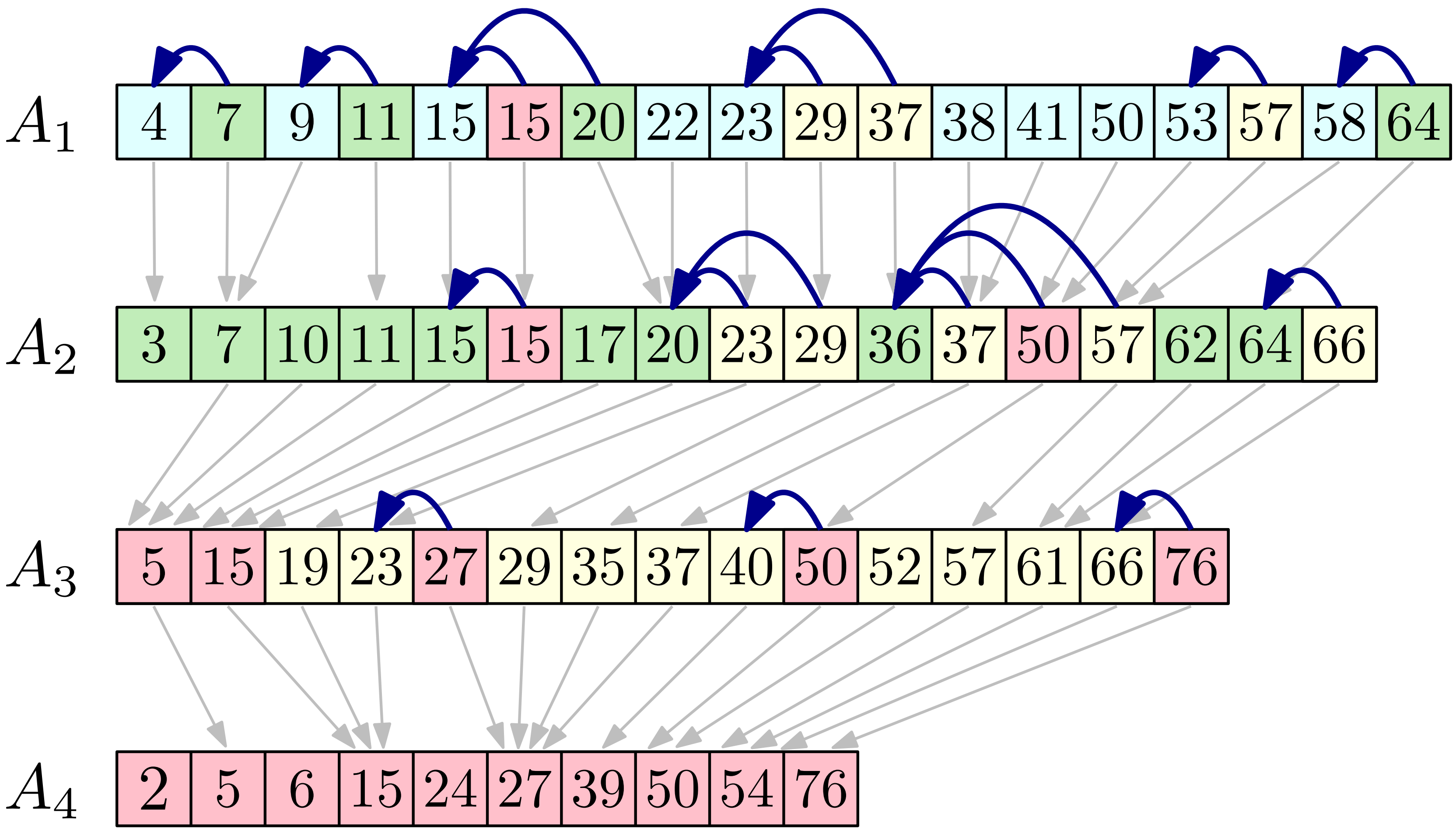
Second idea: fractional cascading

For $i = k, k-1, \ldots, 2$: Add every other element of A_i to A_{i-1}.



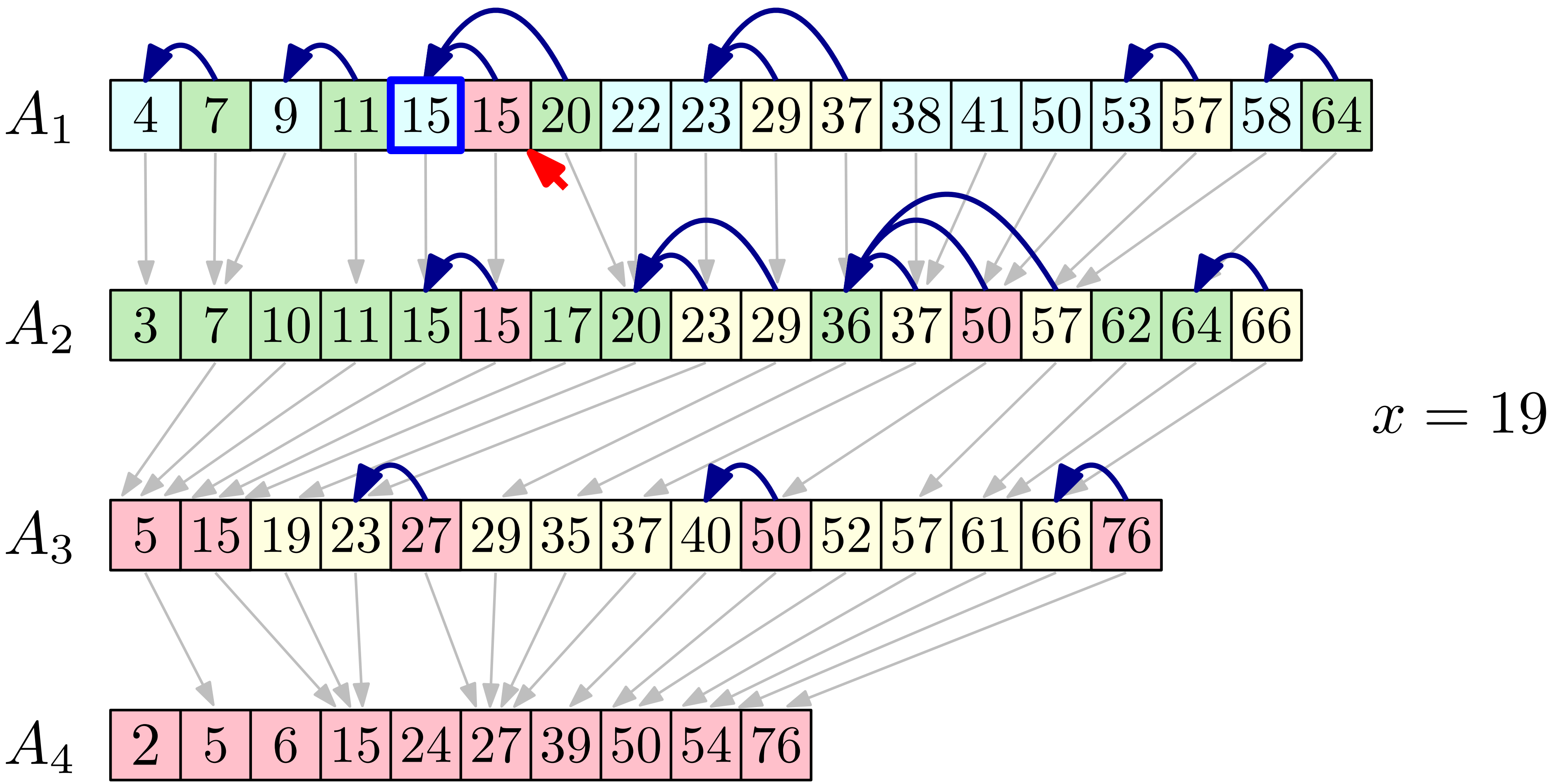
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



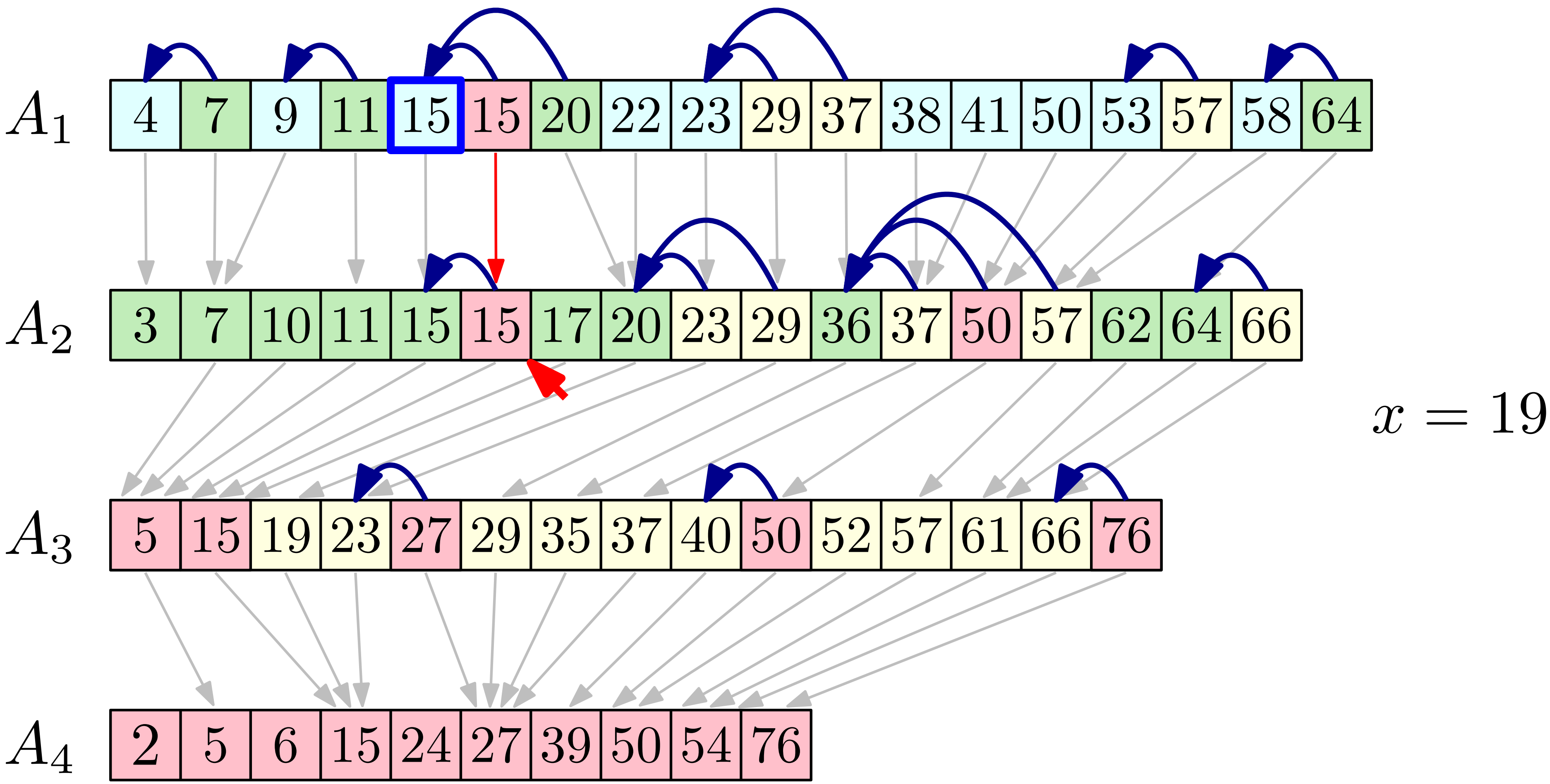
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



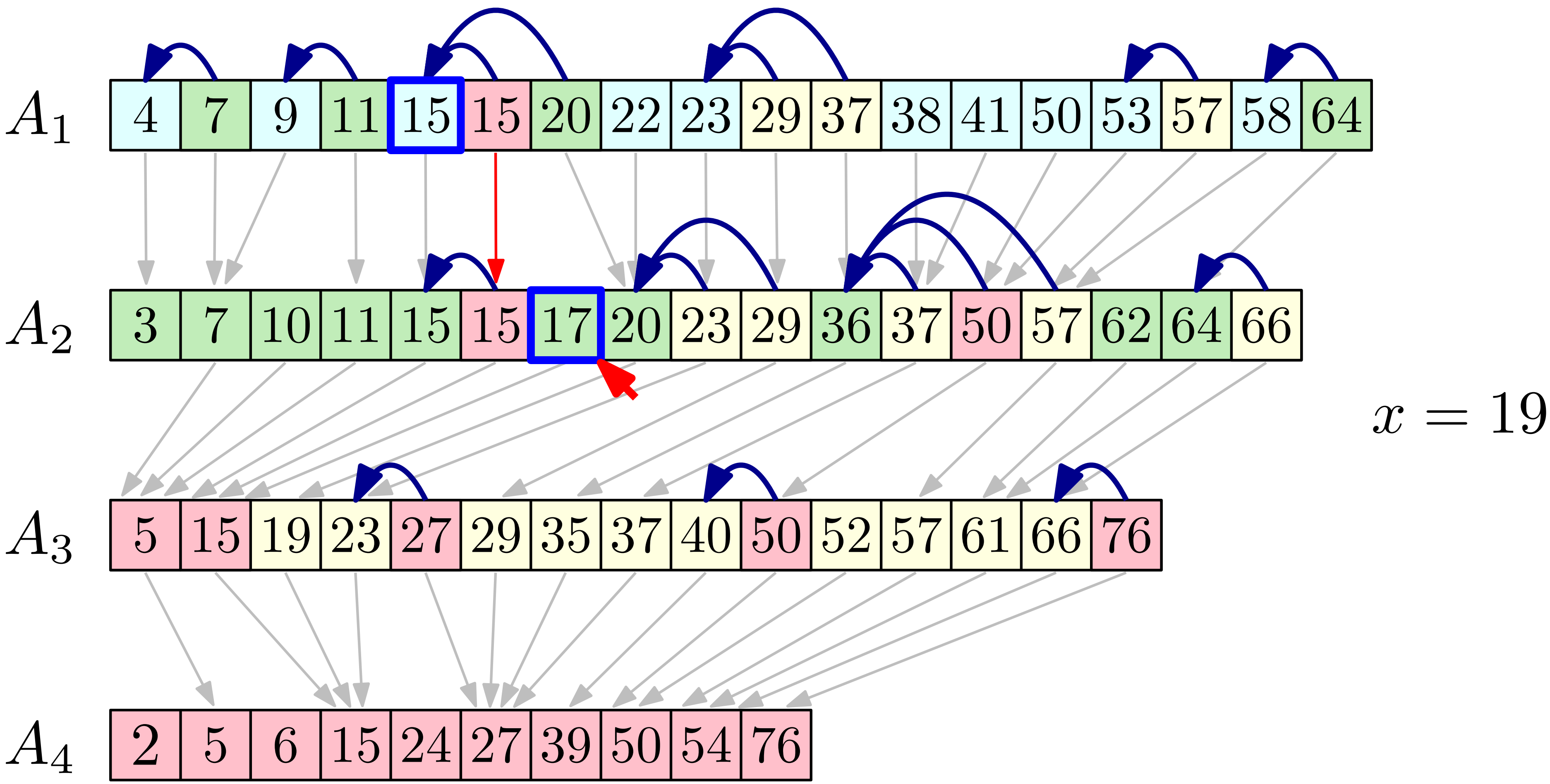
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



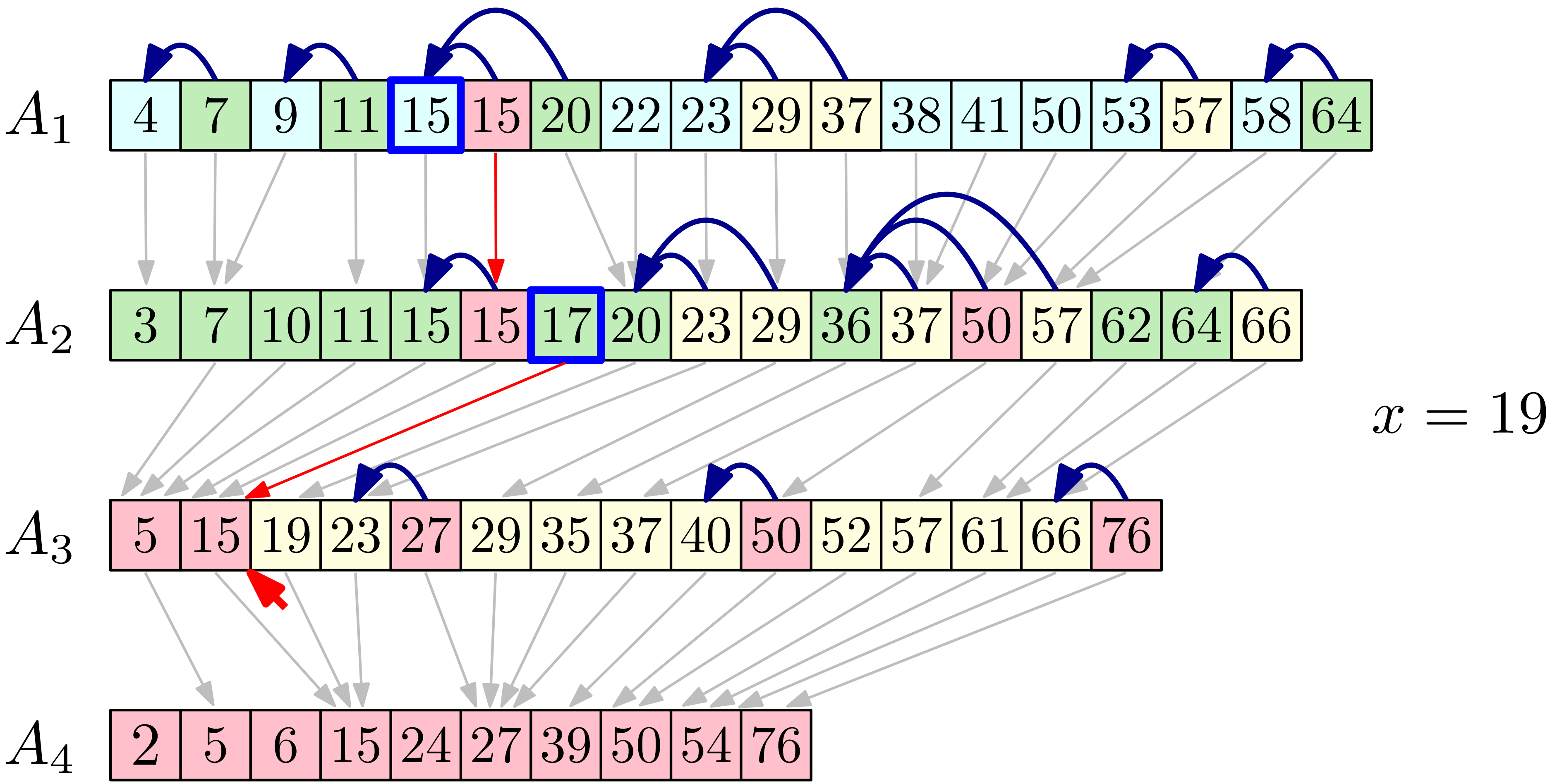
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



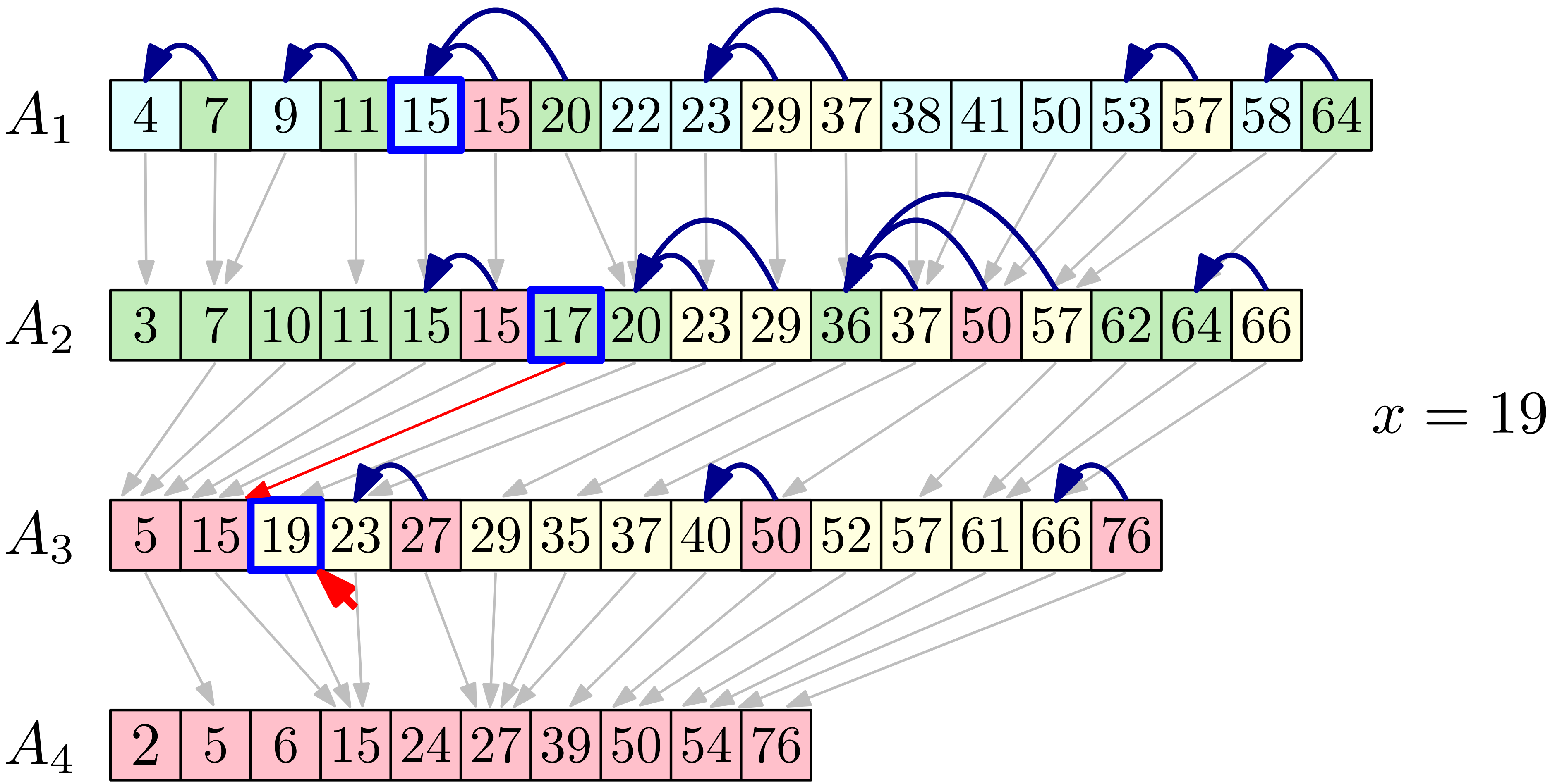
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



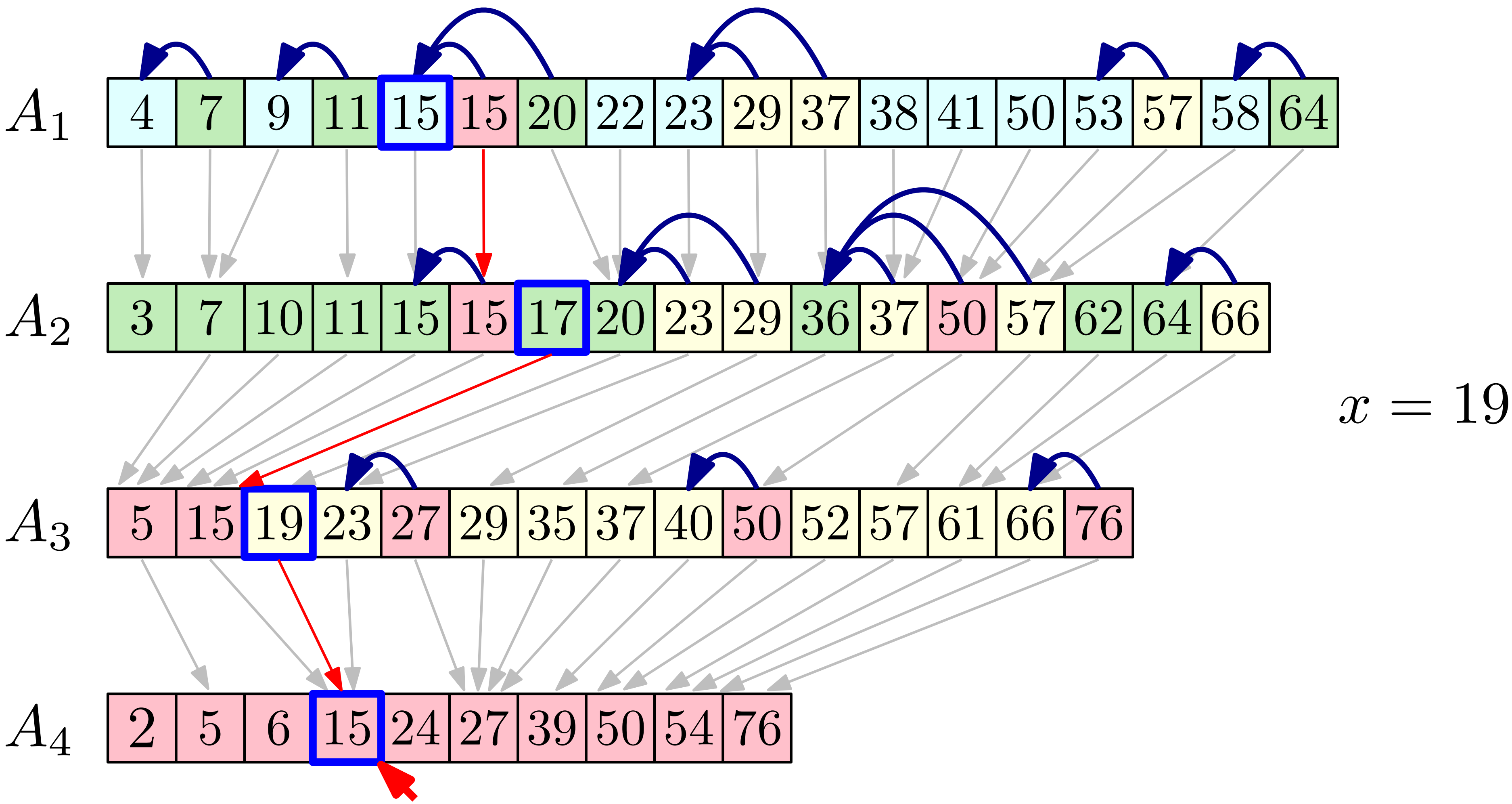
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



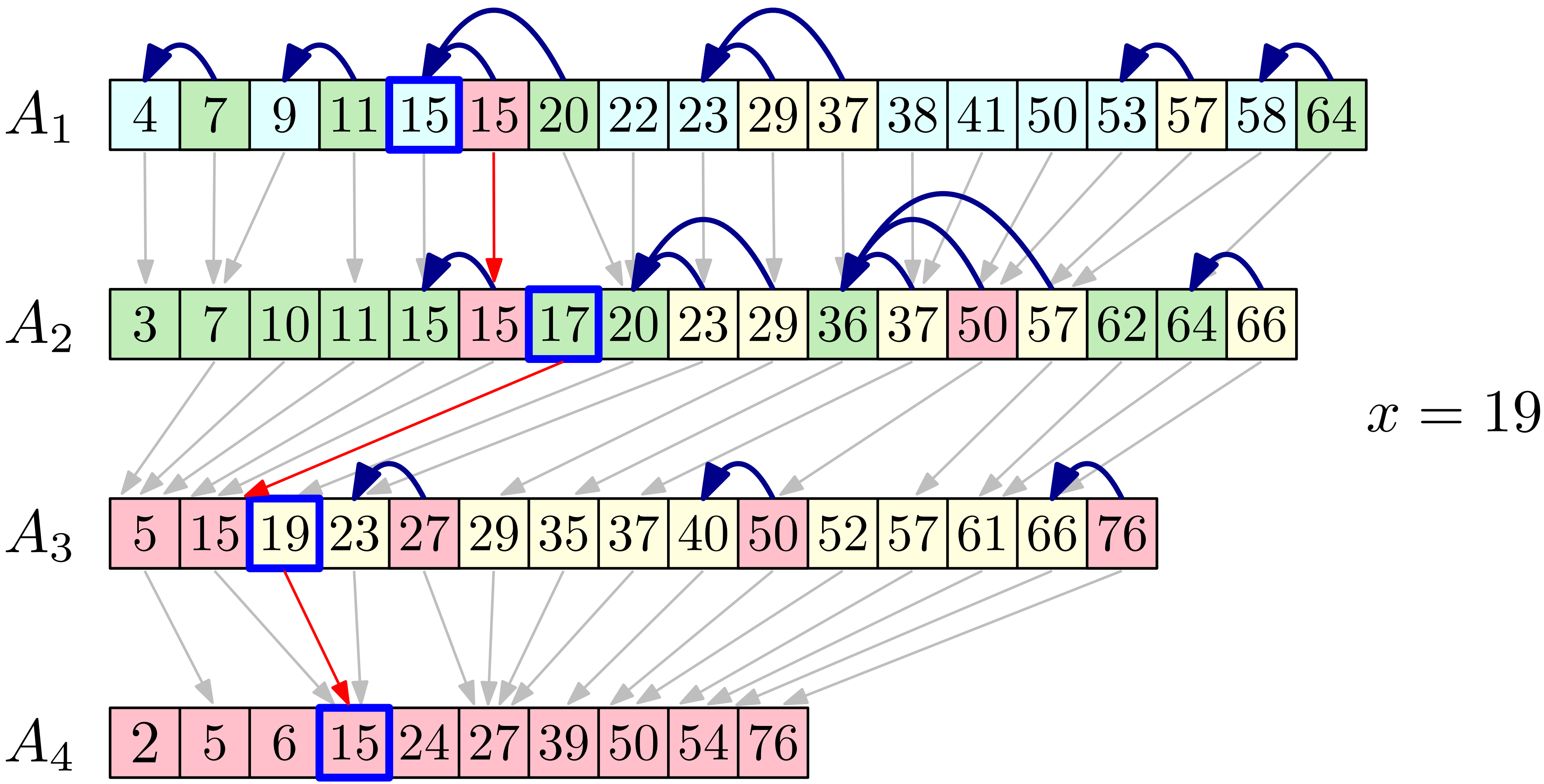
Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i



Fractional Cascading

Keep pointers from newly added elements to A_i to their predecessor among the original elements of A_i

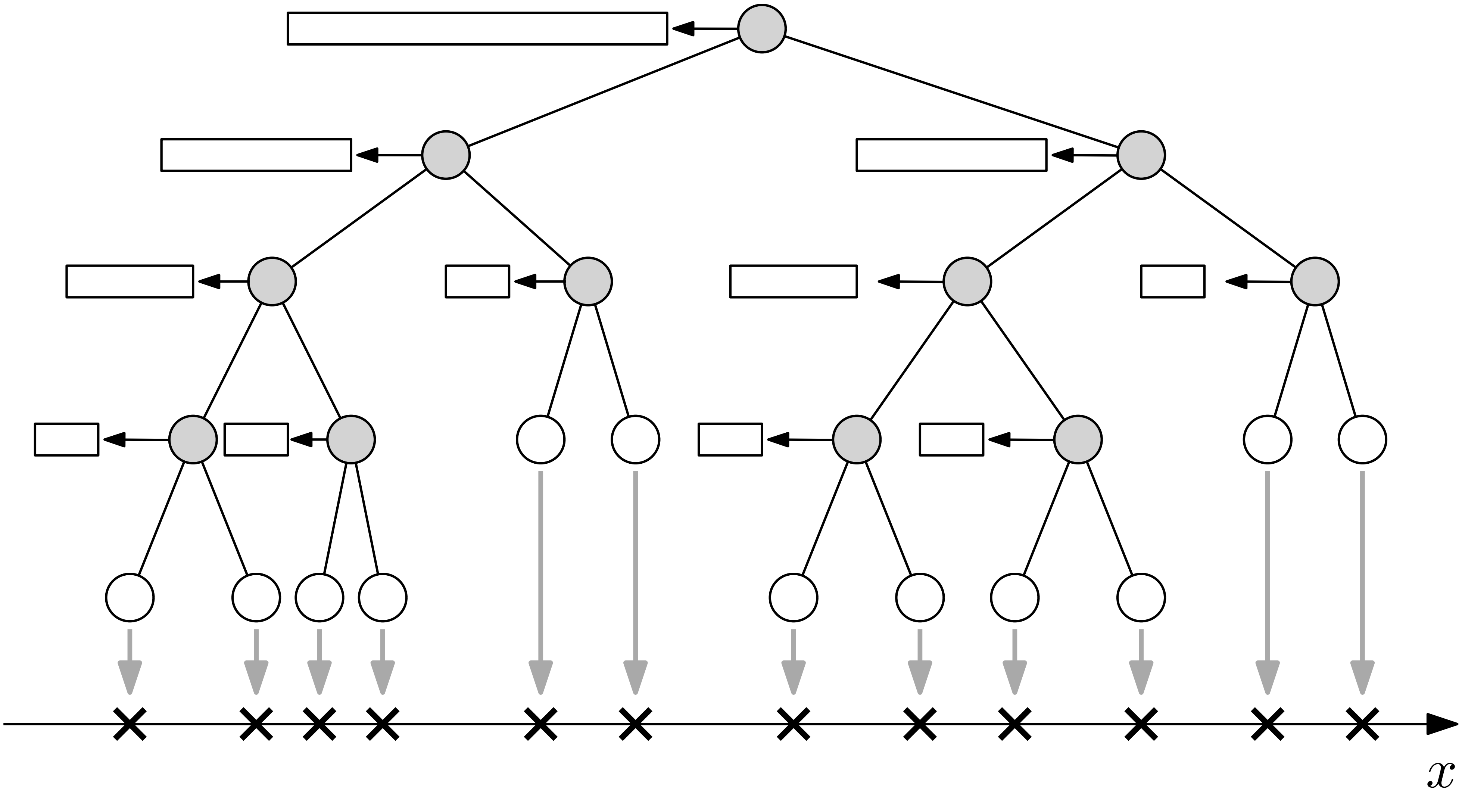


Size $O(kn)$ Preprocessing $O(kn)$ Query: $O(k + \log n)$

Layered Range Trees

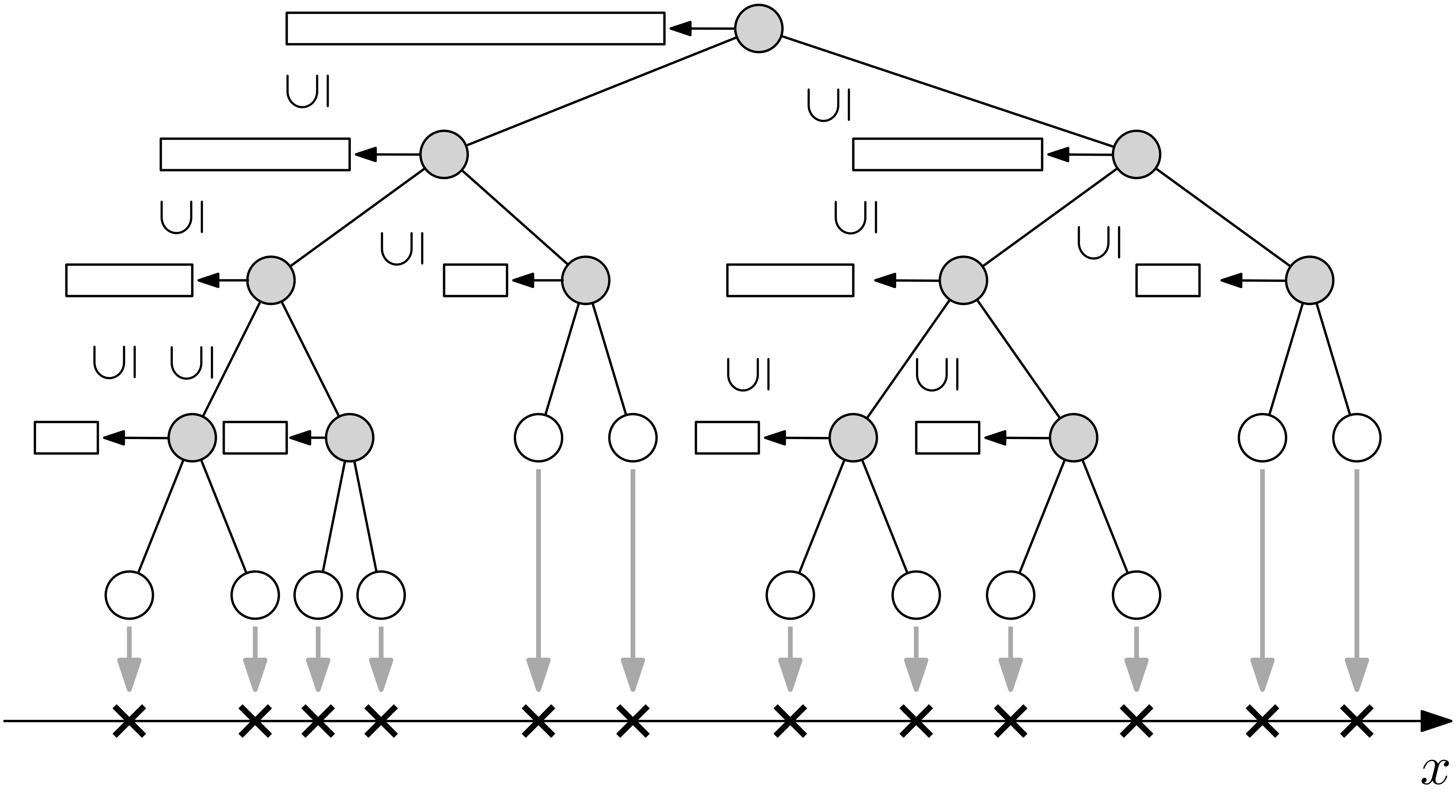
Layered Range Trees, $D = 2$

Reuse the cross-linking idea from fractional cascading



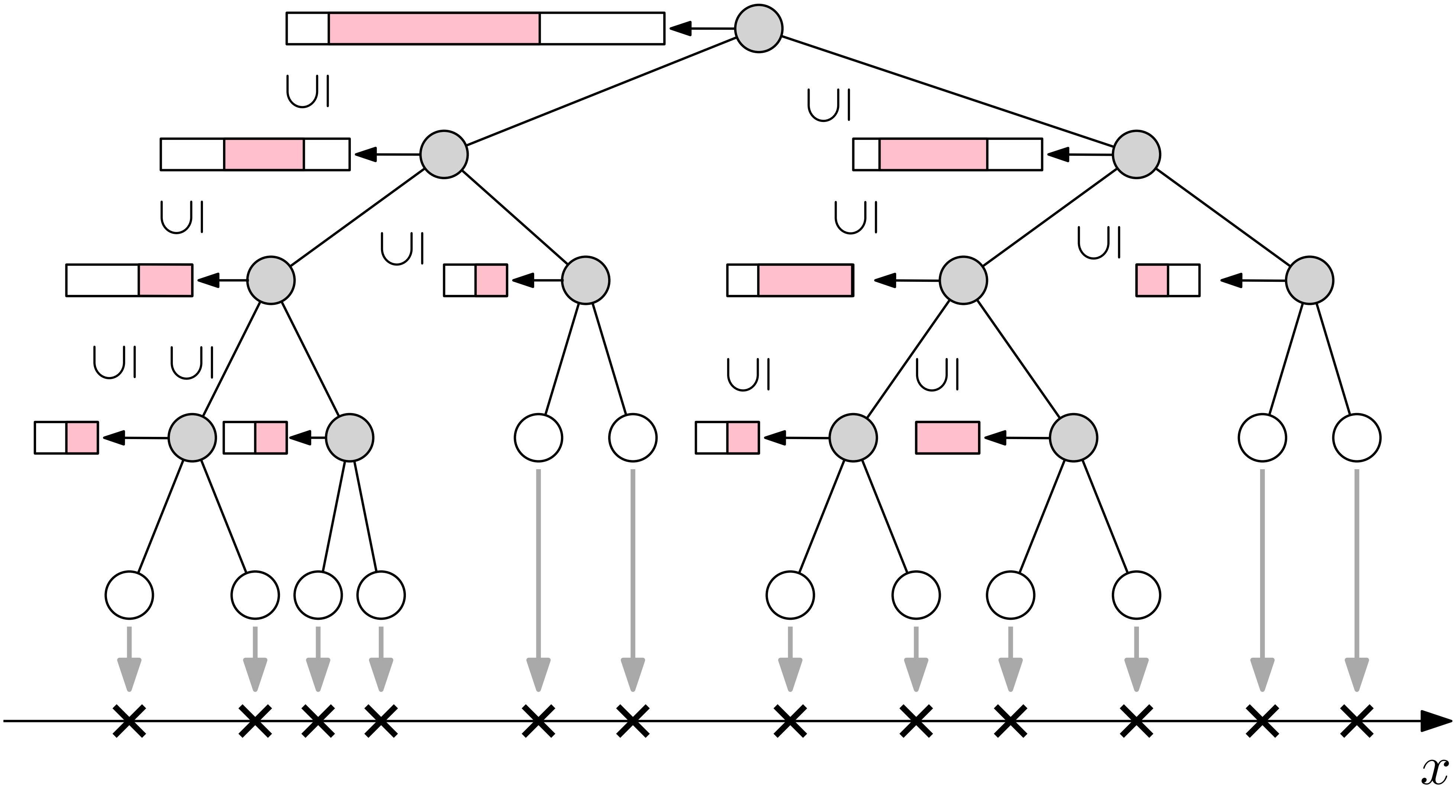
Layered Range Trees, $D = 2$

Reuse the cross-linking idea from fractional cascading



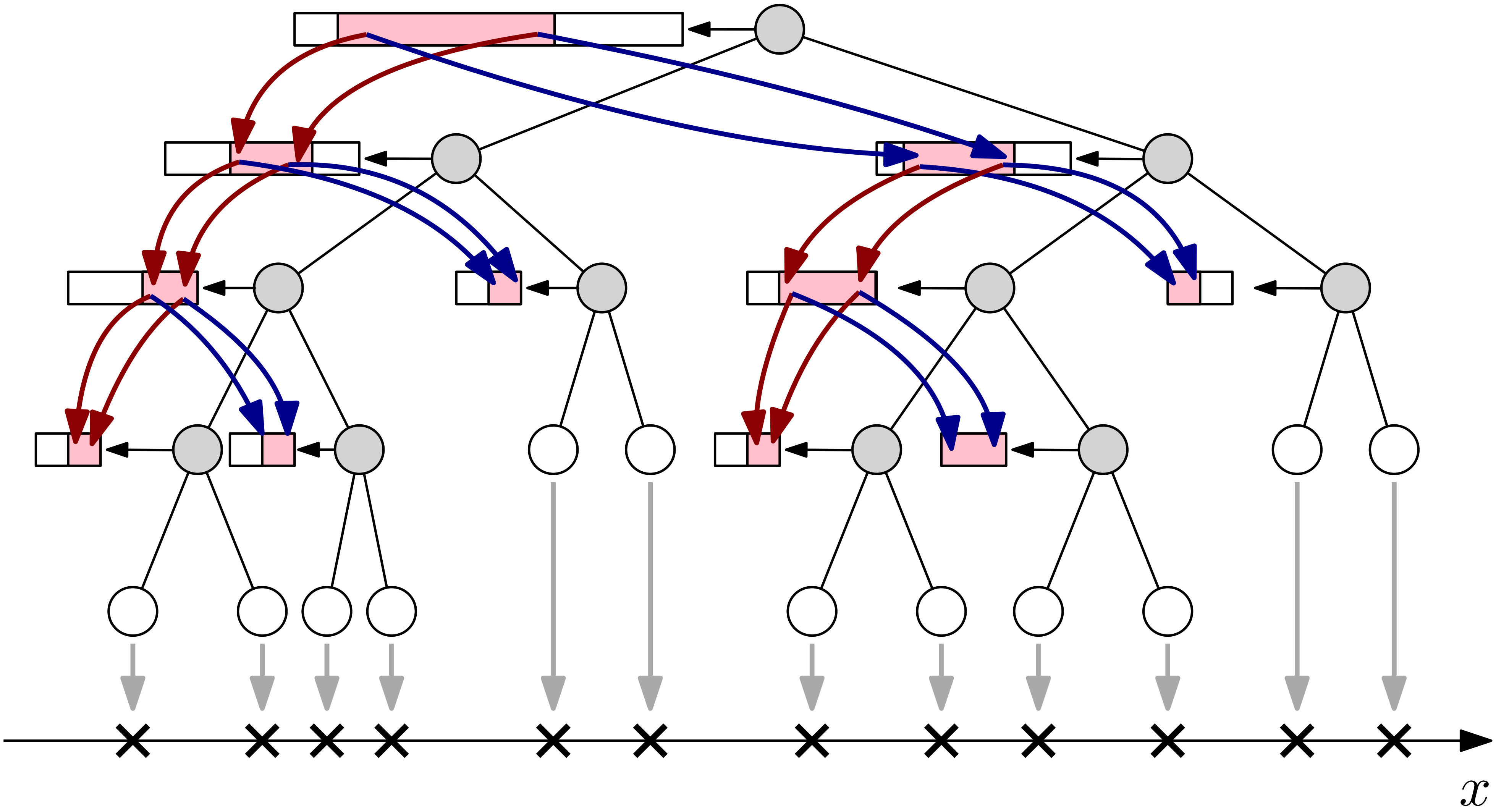
Layered Range Trees, $D = 2$

Reuse the cross-linking idea from fractional cascading



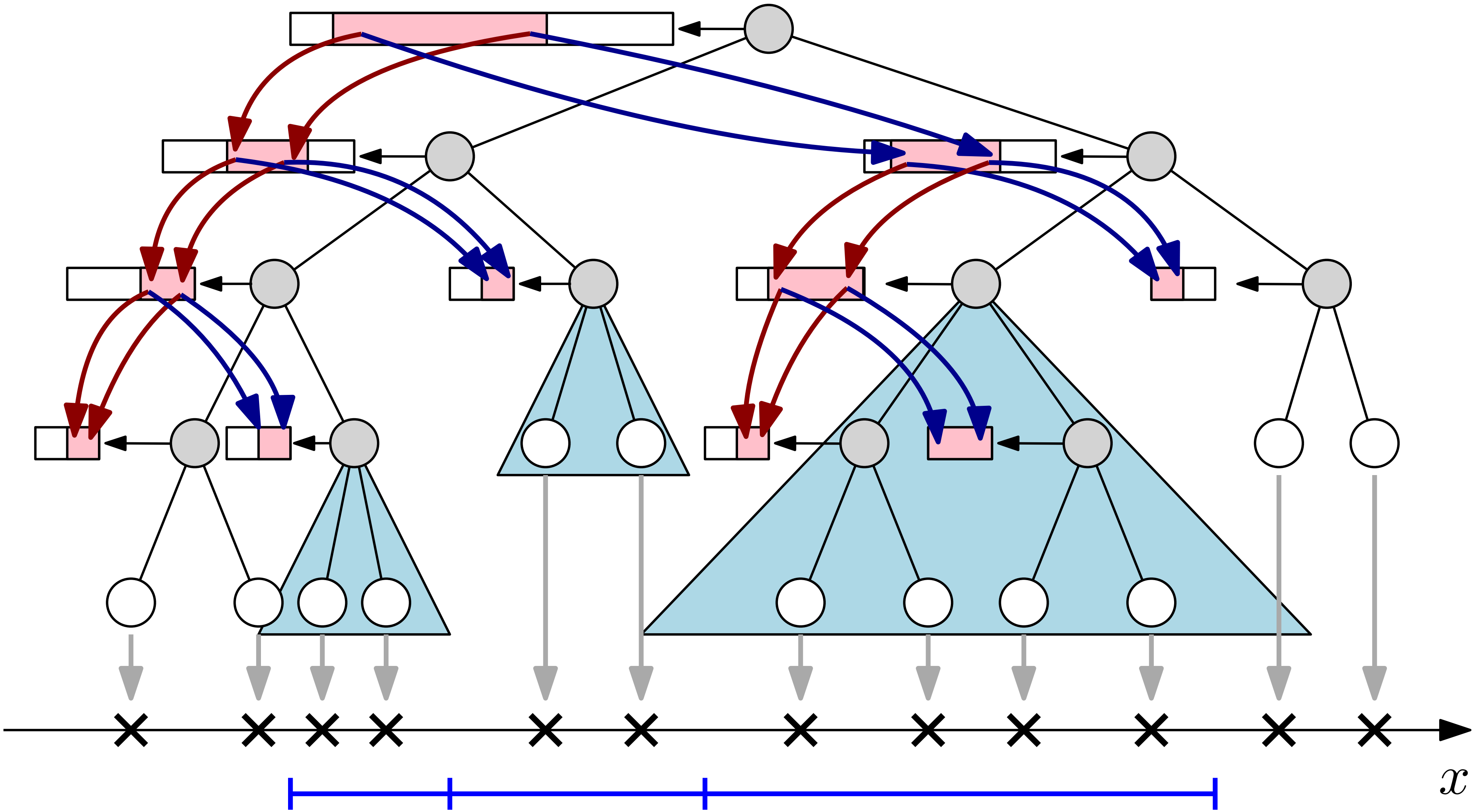
Layered Range Trees, $D = 2$

$\forall$ element y in the 1D range tree of v, store a pointer to the predecessor of y in the 1D range tree of the left/right child of v.



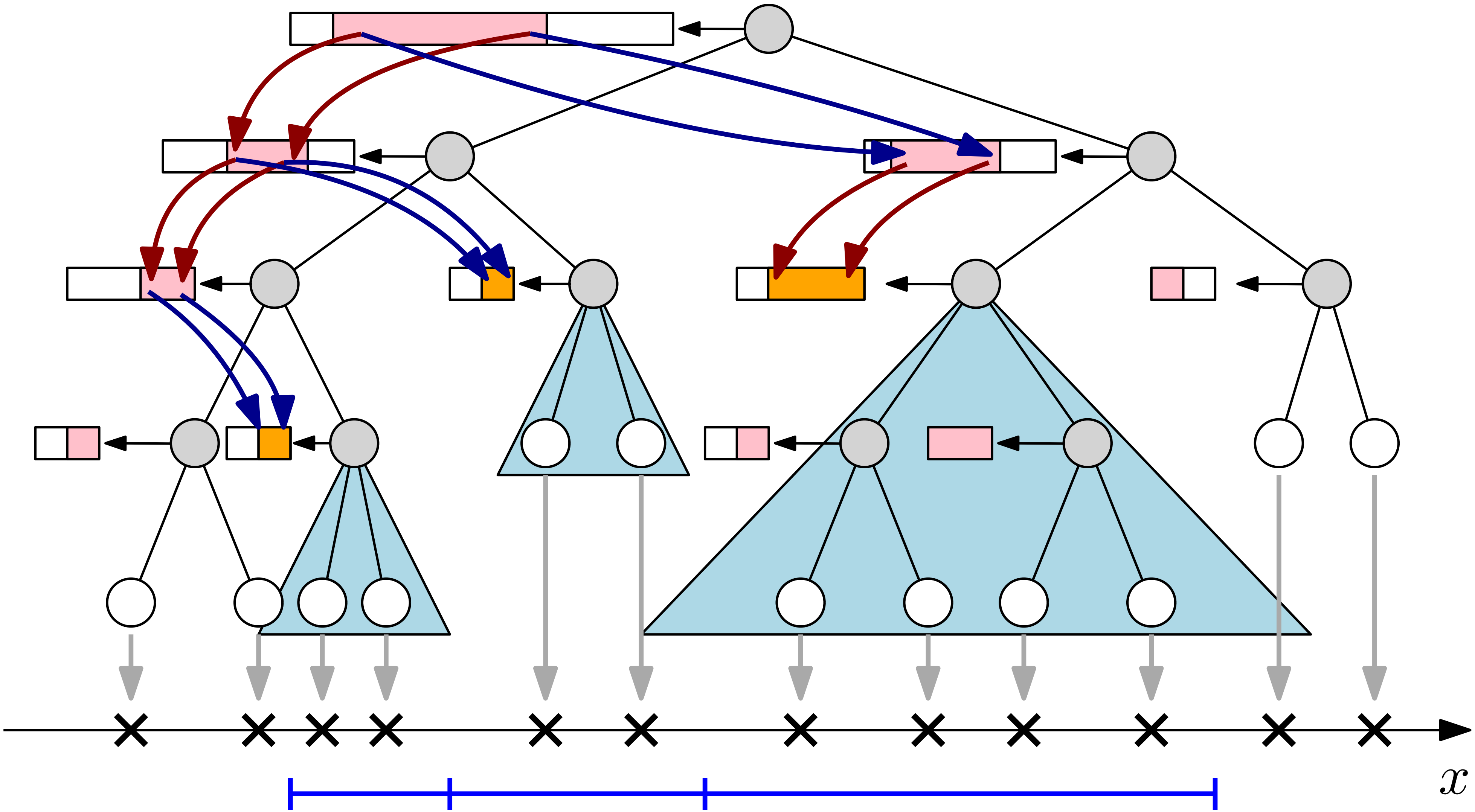
Layered Range Trees, $D = 2$

$\forall$ element y in the 1D range tree of v, store a pointer to the predecessor of y in the 1D range tree of the left/right child of v.



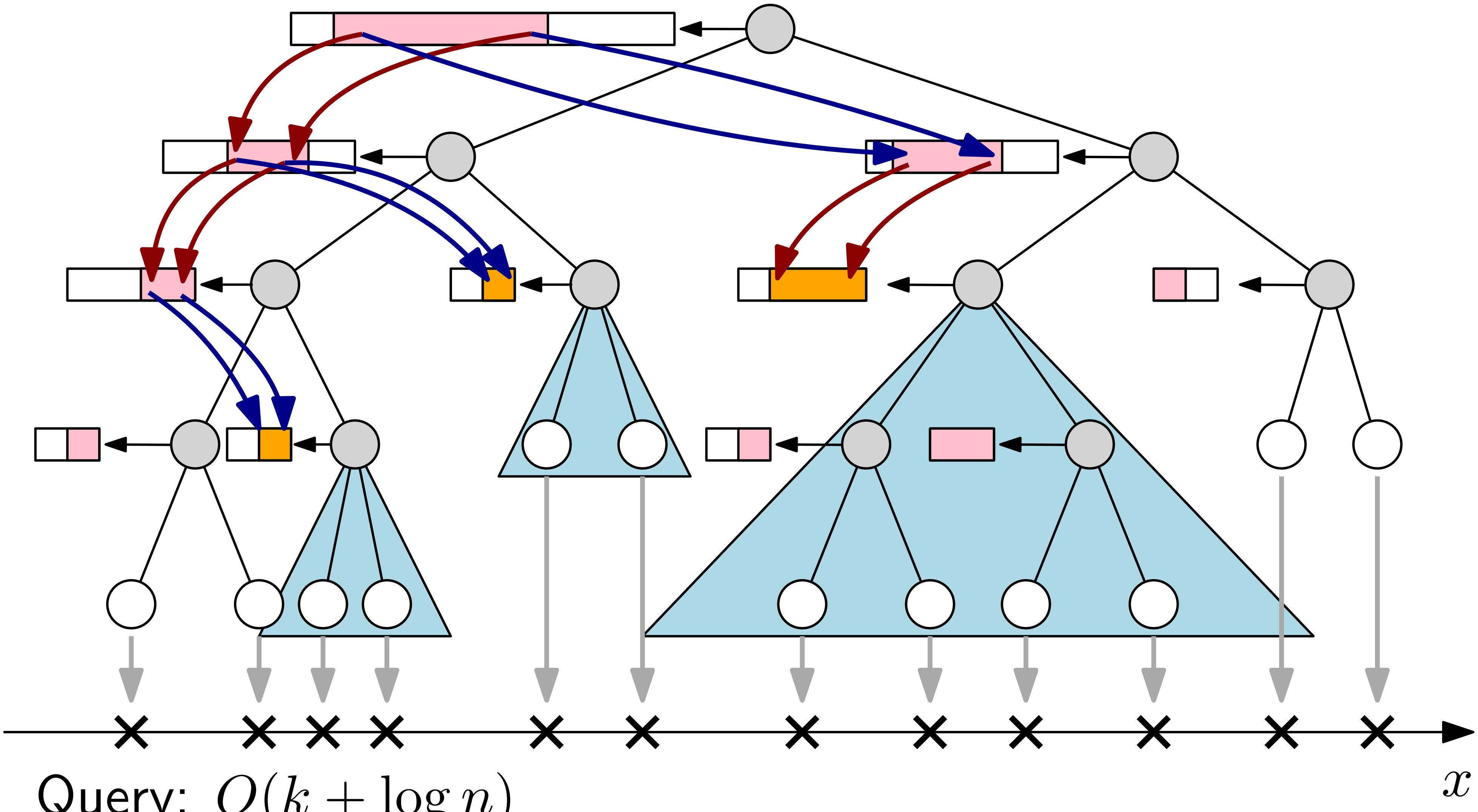
Layered Range Trees, $D = 2$

$\forall$ element y in the 1D range tree of v, store a pointer to the predecessor of y in the 1D range tree of the left/right child of v.



Layered Range Trees, $D = 2$

$\forall$ element y in the 1D range tree of v, store a pointer to the predecessor of y in the 1D range tree of the left/right child of v.



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

Recap

D	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	

Recap

D	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log n + k)$	with cross-linking

Recap

D	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log n + k)$	with cross-linking
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^{D-1} n + k)$	with cross-linking

Recap

D	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log n + k)$	with cross-linking
> 2	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^{D-1} n + k)$	with cross-linking

Can be made dynamic (supports point insertion / deletion) in $O(\log^D n)$ amortized time per update.