Range Minimum Queries

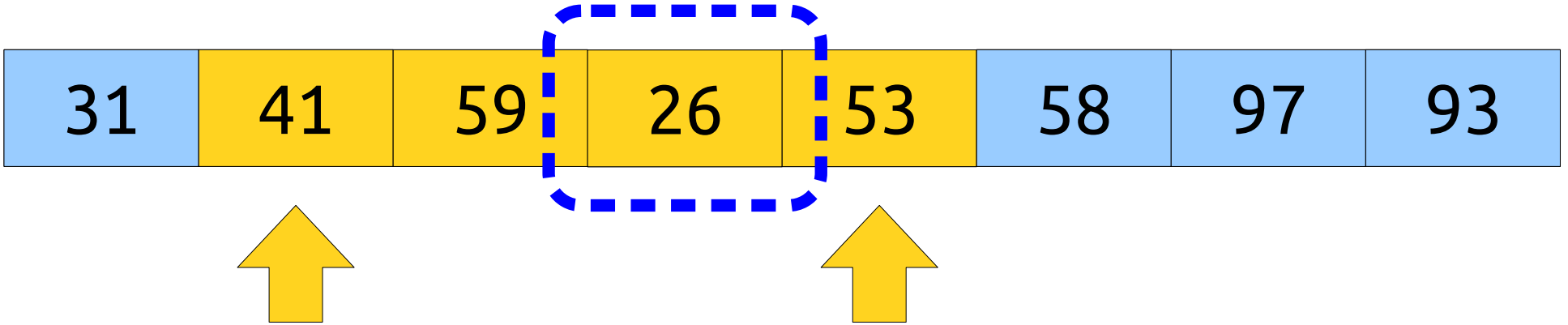
Part Two

Recap from Last Time

The RMQ Problem

- The ***Range Minimum Query*** (***RMQ***) problem is the following:

  Given a fixed array A and two indices $i \leq j$, what is the smallest element out of A[i], A[$i + 1$], …, A[$j - 1$], A[j]?

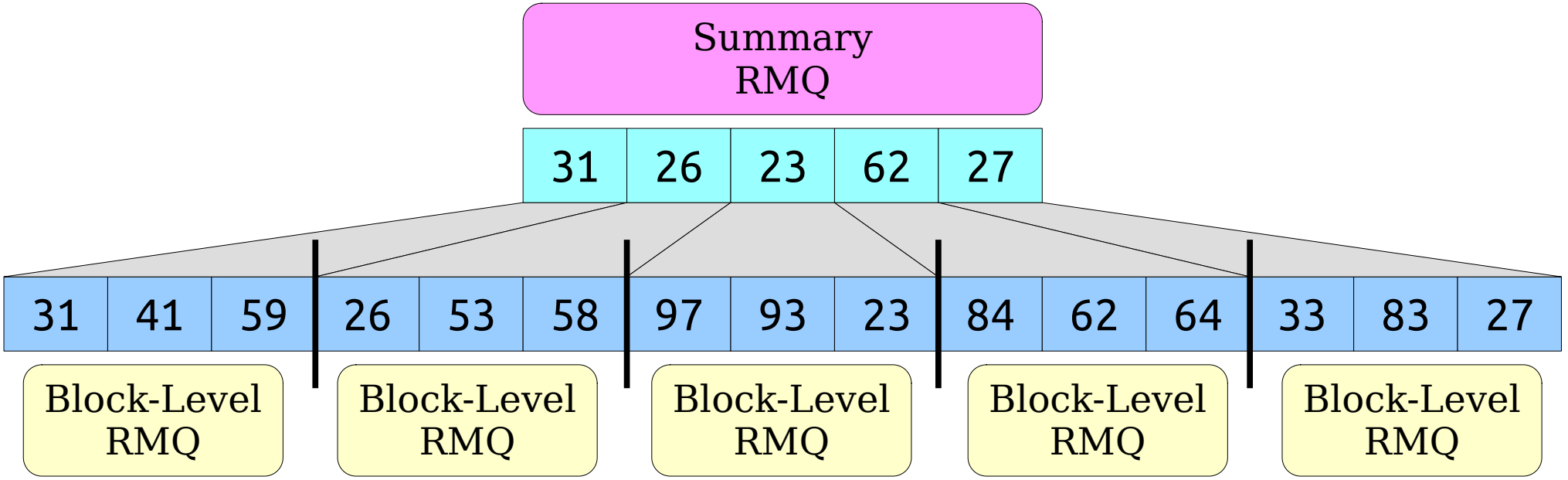


Some Notation

- We'll say that an RMQ data structure has time complexity **⟨*p*(*n*), *q*(*n*)⟩** if
  - preprocessing takes time at most $p(n)$ and
  - queries take time at most $q(n)$.
- Last time, we saw structures with the following runtimes:
  - $\langle O(n^2), O(1) \rangle$ (full preprocessing)
  - $\langle O(n \log n), O(1) \rangle$ (sparse table)
  - $\langle O(n \log \log n), O(1) \rangle$ (hybrid approach)
  - $\langle O(n), O(n^{1/2}) \rangle$ (blocking)
  - $\langle O(n), O(\log n) \rangle$ (hybrid approach)
  - $\langle O(n), O(\log \log n) \rangle$ (hybrid approach)

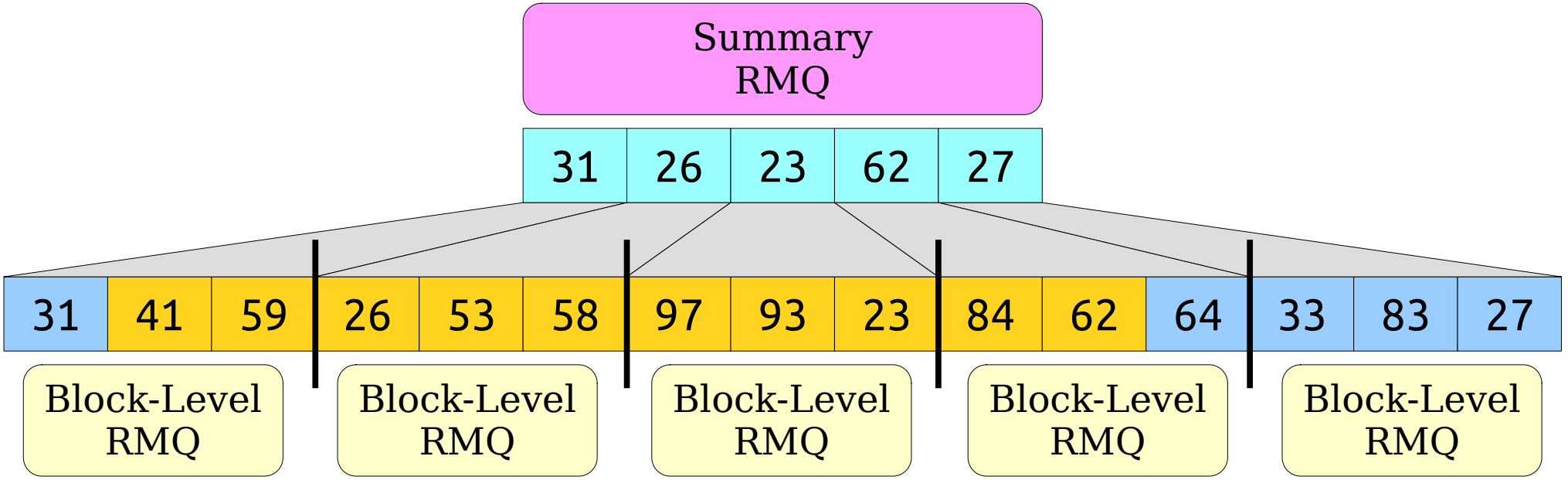
The Framework

- Split the input into blocks of size b.
- Form an array of the block minima.
- Construct a “summary” RMQ structure over the block minima.
- Construct “block” RMQ structures for each block.
- Aggregate the results together.



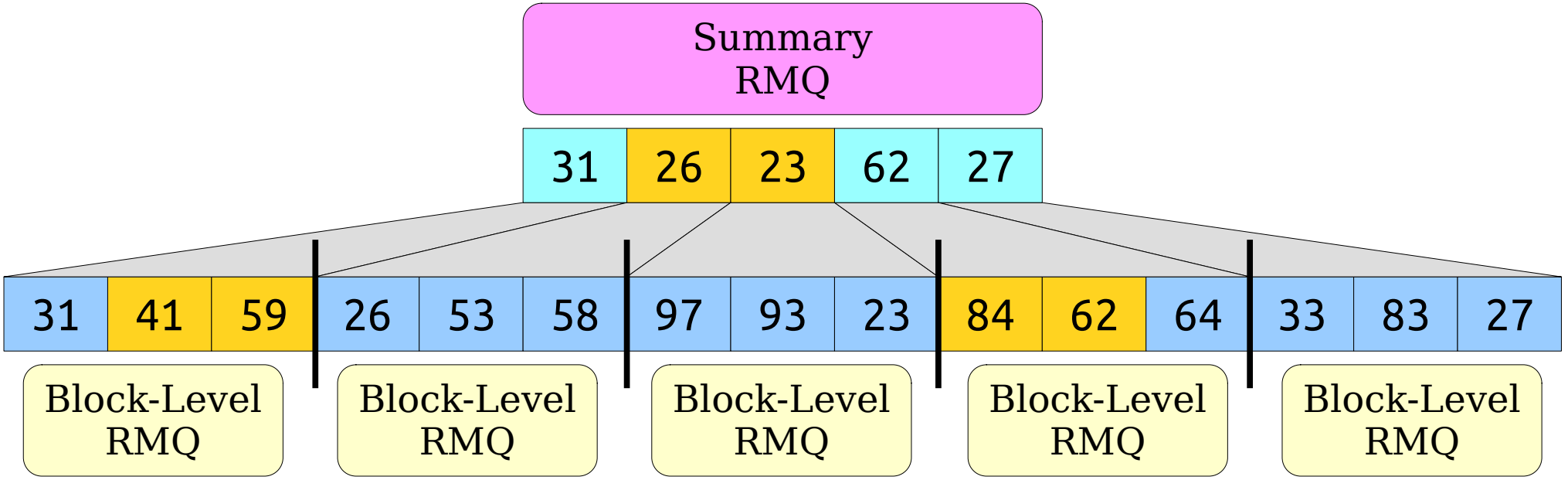
The Framework

- Split the input into blocks of size b.
- Form an array of the block minima.
- Construct a “summary” RMQ structure over the block minima.
- Construct “block” RMQ structures for each block.
- Aggregate the results together.



The Framework

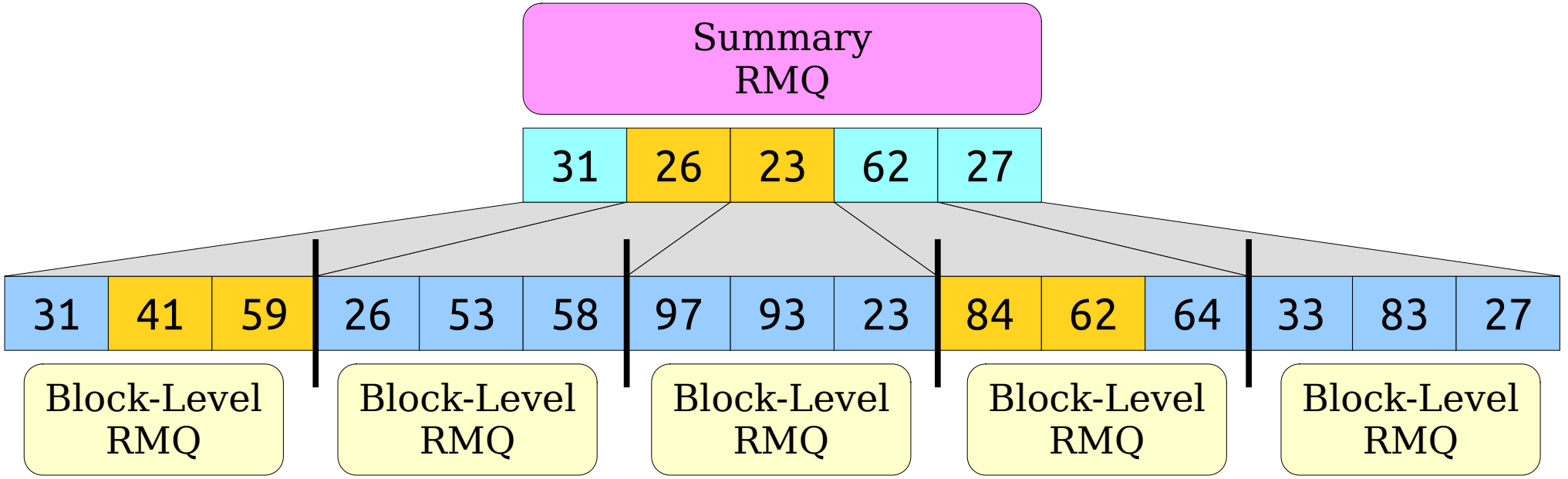
- Split the input into blocks of size b.
- Form an array of the block minima.
- Construct a “summary” RMQ structure over the block minima.
- Construct “block” RMQ structures for each block.
- Aggregate the results together.



The Framework

- Suppose we use a $\langle p_1(n), q_1(n) \rangle$-time RMQ solution for the summary and a $\langle p_2(n), q_2(n) \rangle$-time RMQ solution within each block. Let the block size be b.
- In the hybrid structure, the preprocessing time is

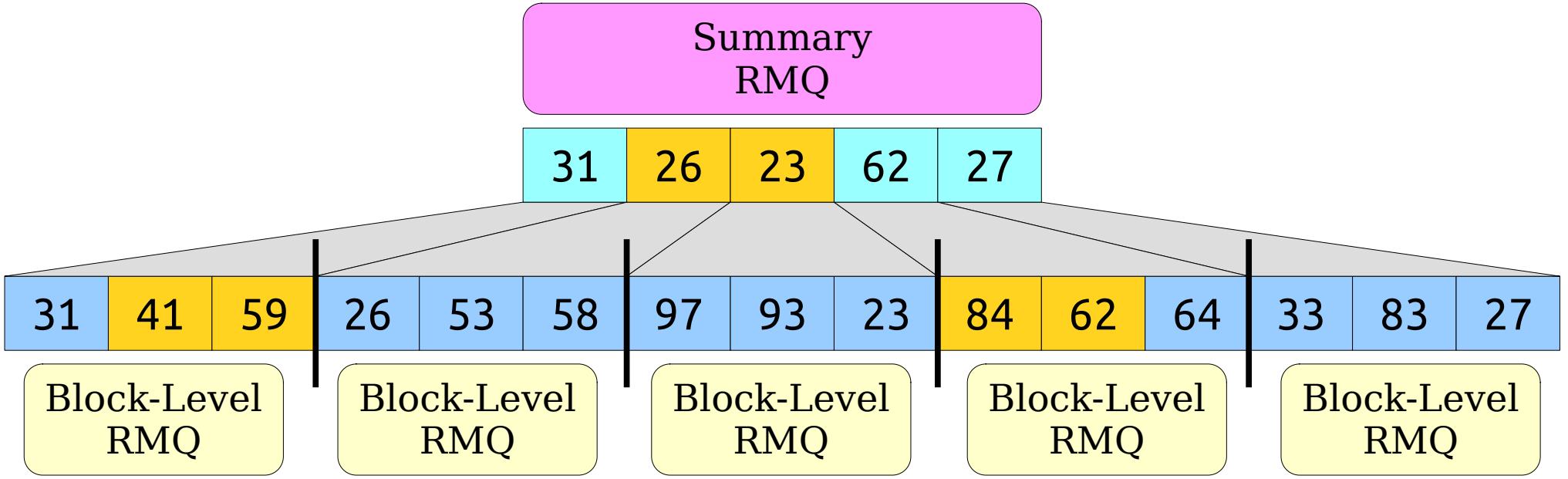
$$\mathbf{O(n + p_1(n / b) + (n / b)\, p_2(b))}$$



The Framework

- Suppose we use a $\langle p_1(n), q_1(n) \rangle$-time RMQ solution for the summary and a $\langle p_2(n), q_2(n) \rangle$-time RMQ solution within each block. Let the block size be b.
- In the hybrid structure, the query time is

$$\mathbf{O(q_1(n / b) + q_2(b))}$$



Is there an $\langle O(n), O(1) \rangle$ solution to RMQ?

Yes!

New Stuff!

An Observation

The Limits of Hybrids

- The preprocessing time on a hybrid structure is

$$O(n + p_1(n / b) + (n / b)\, p_2(b)).$$

- The query time is

$$O(q_1(n / b) + q_2(b)).$$

The Limits of Hybrids

- The preprocessing time on a hybrid structure is

$$O(n + p_1(n / b) + (n / b)\, p_2(b)).$$

- The query time is

$$O(q_1(n / b) + q_2(b)).$$

- What do $p_2(b)$ and $q_2(b)$ need to be if we want to build a $\langle O(n), O(1) \rangle$ RMQ structure?

Formulate a hypothesis!
Discuss with your neighbors!

The Limits of Hybrids

- The preprocessing time on a hybrid structure is

$$O(n + p_1(n / b) + \mathbf{(n / b)\ p_2(b)}).$$

- The query time is

$$O(q_1(n / b) + \mathbf{q_2(b)}).$$

- What do $p_2(b)$ and $q_2(b)$ need to be if we want to build a $\langle O(n), O(1) \rangle$ RMQ structure?

$$\mathbf{p_2(b) = O(b)} \qquad \mathbf{q_2(b) = O(1)}$$

- ***Problem:*** We can't build an optimal RMQ structure unless we already have one!
- ***Or can we?***

The Limits of Hybrids

The preprocessing time on a hybrid structure is

$$O(n + p_1(n / b) + \mathbf{(n / b)\ p_2(b)}).$$

The c

p(b)).

What

o be if we want

to bu

ucture?

b) = O(1)

Problem: We can't build an optimal RMQ structure unless we already have one!

Or can we?

This is the work required to construct an RMQ structure for each block.

Each block has size b.

Number of blocks: $O(n / b)$.

A Key Difference

- Our original problem is

  Solve RMQ on a single array in time ⟨O(*n*), O(1)⟩.

- The new problem is

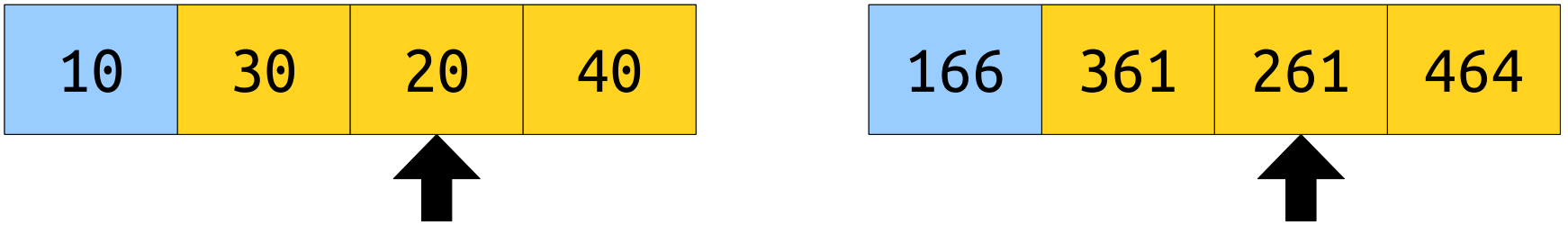
  Solve RMQ on a large number of small arrays with O(1) query time and *total* preprocessing time O(*n*).

- These are not the same problem.
- ***Question:*** Why is this second problem any easier than the first?

An Observation

An Observation

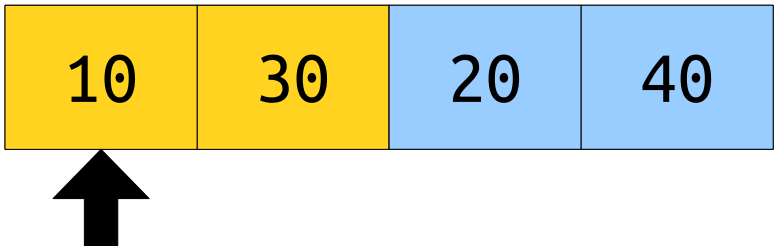
An Observation

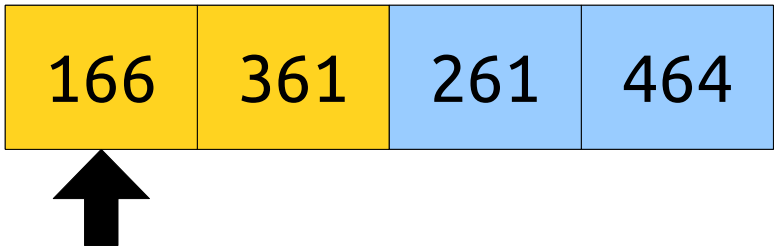


An Observation

An Observation

An Observation

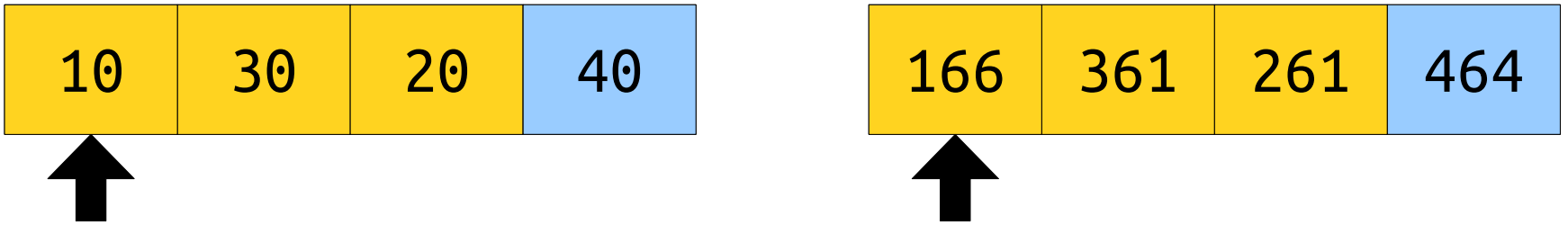




An Observation

An Observation

An Observation



An Observation

10	30	20	40

166	361	261	464

Claim: The indices of the answers to any range minimum queries on these two arrays are the same.

Modifying RMQ

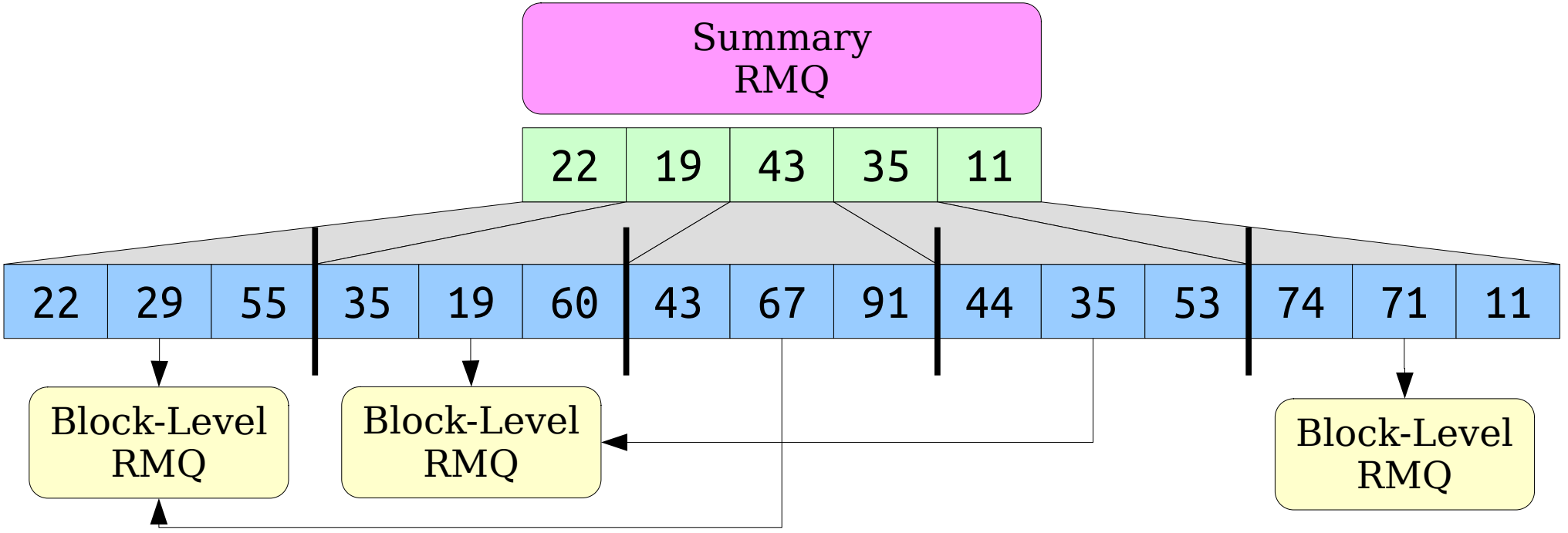
- From this point forward, let's have $\text{RMQ}_A(i, j)$ denote the ***index*** of the minimum value in the range rather than the value itself.
- ***Observation:*** If RMQ structures return indices rather than values, we can use a single RMQ structure for both of these arrays:

10	30	20	40

166	361	261	464

Where We're Going

- Suppose we use an ⟨O(n log n), O(1)⟩ sparse table for the top and the ⟨O(n^2), O(1)⟩ precompute-all structures for the blocks.
- However, whenever possible, we share block-level RMQ structures across multiple blocks.
- Assuming there aren't "too many" different types of blocks, and assuming we can find and group blocks efficiently, this overall strategy might let us reach a ⟨O(n), O(1)⟩ solution!



Two Big Questions

How can we tell when two blocks can share RMQ structures?

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

(Without an answer, this whole approach doesn't work!)

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

(Without an answer, this whole approach doesn't work!)

How many block types are there, as a function of b?

(We need to tune b to ensure that many blocks are shared. What value of b should we pick?)

The Adventure Begins!

Some Notation

- Let B_1 and B_2 be blocks of length b.
- We'll say that B_1 and B_2 ***have the same block type*** (denoted $\boldsymbol{B_1 \sim B_2}$) if the following holds:

$$\textbf{For all } 0 \leq i \leq j < b\textbf{:}$$
$$\textbf{RMQ}_{B_1}(i, j) = \textbf{RMQ}_{B_2}(i, j)$$

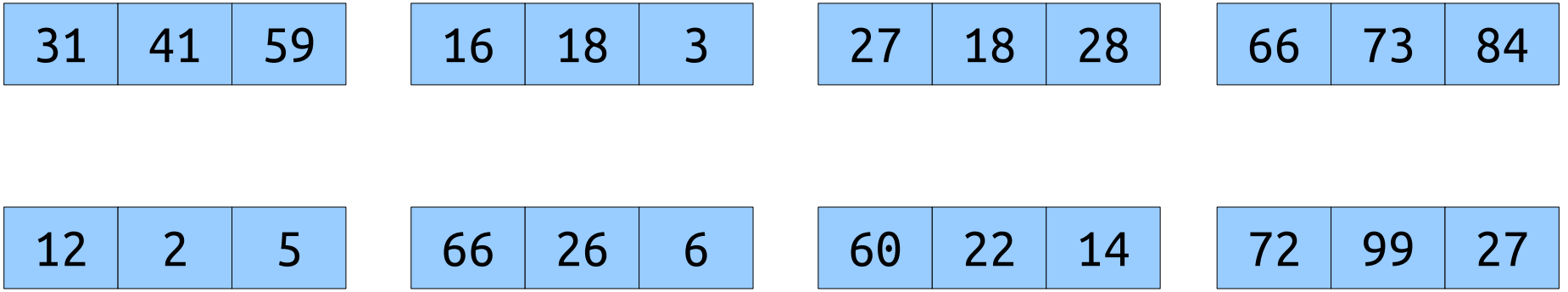
- Intuitively, the RMQ answers for B_1 are always the same as the RMQ answers for B_2.
- If we build an RMQ to answer queries on some block B_1, we can reuse that RMQ structure on some other block B_2 iff $B_1 \sim B_2$.

Detecting Block Types

- For this approach to work, we need to be able to check whether two blocks have the same block type.
- ***Problem:*** Our formal definition of $B_1 \sim B_2$ is defined in terms of RMQ.
  - Not particularly useful *a priori;* we don't want to have to compute RMQ structures on B_1 and B_2 to decide whether they have the same block type!
- Is there a simpler way to determine whether two blocks have the same type?

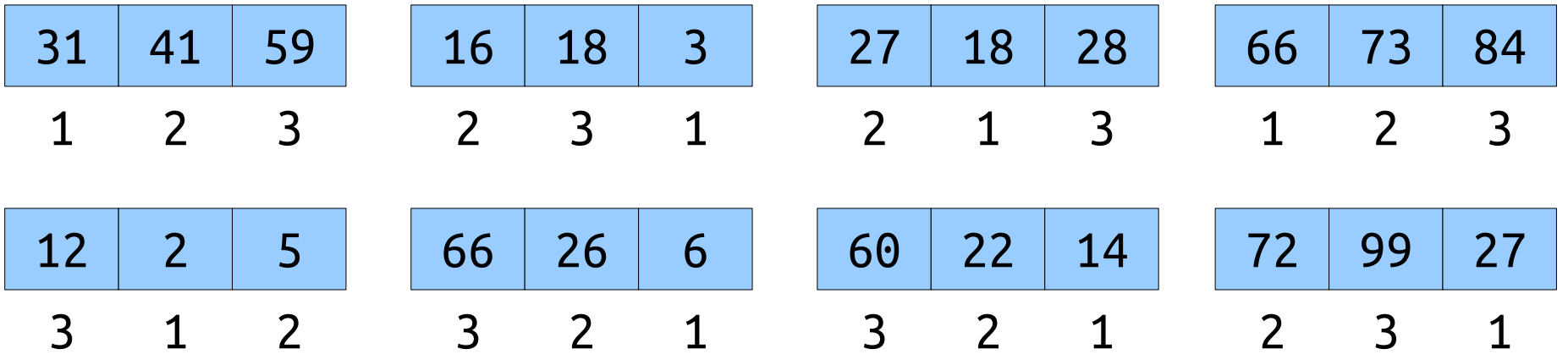
An Initial Idea

- Since the elements of the array are ordered and we're looking for the smallest value in certain ranges, we might look at the permutation types of the blocks.



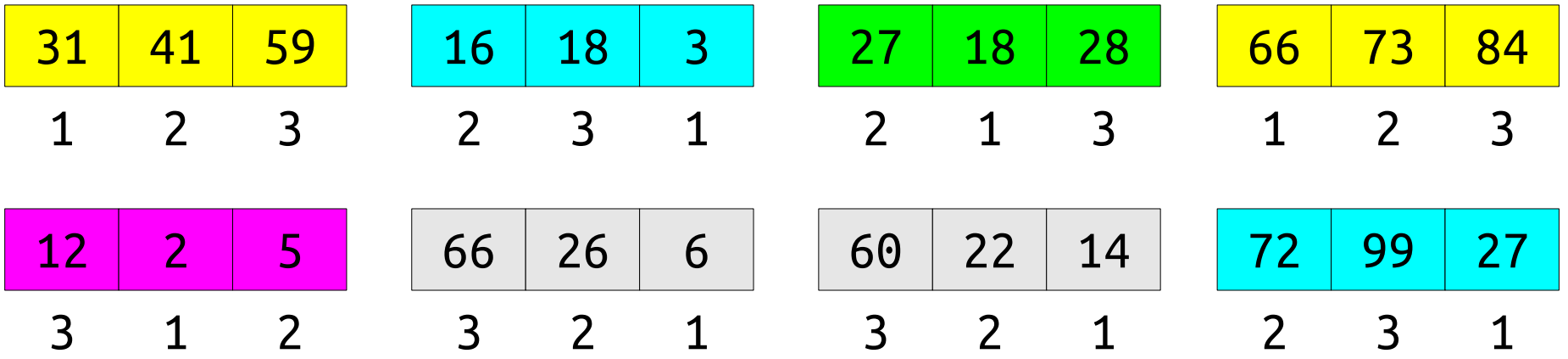
An Initial Idea

- Since the elements of the array are ordered and we're looking for the smallest value in certain ranges, we might look at the permutation types of the blocks.



An Initial Idea

- Since the elements of the array are ordered and we're looking for the smallest value in certain ranges, we might look at the permutation types of the blocks.



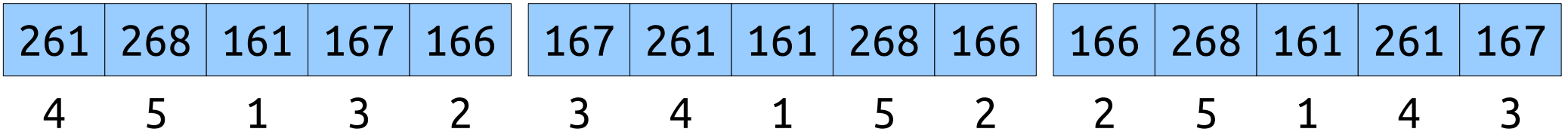
- ***Claim:*** If B_1 and B_2 have the same permutation on their elements, then $B_1 \sim B_2$.

Some Problems

- There are two main problems with this approach.
- ***Problem One:*** It's possible for two blocks to have different permutations but the same block type.

Some Problems

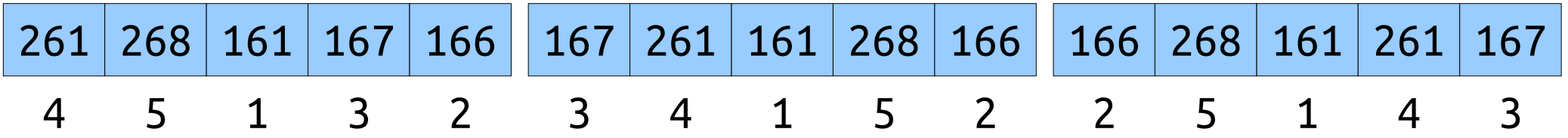
- There are two main problems with this approach.
- ***Problem One:*** It's possible for two blocks to have different permutations but the same block type.
- All three of these blocks have the same block type but different permutation types:



Discuss why with your neighbor!

Some Problems

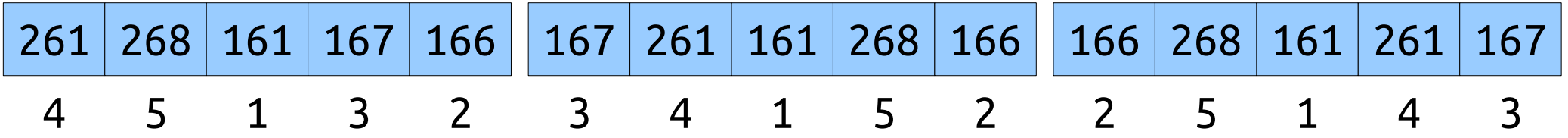
- There are two main problems with this approach.
- ***Problem One:*** It's possible for two blocks to have different permutations but the same block type.
- All three of these blocks have the same block type but different permutation types:



- ***Problem Two:*** The number of possible permutations of a block is $b!$.
  - b has to be absolutely minuscule for $b!$ to be small.

Some Problems

- There are two main problems with this approach.
- ***Problem One:*** It's possible for two blocks to have different permutations but the same block type.
- All three of these blocks have the same block type but different permutation types:



- ***Problem Two:*** The number of possible permutations of a block is $b!$.
  - b has to be absolutely minuscule for $b!$ to be small.
- Is there a better criterion we can use?

An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

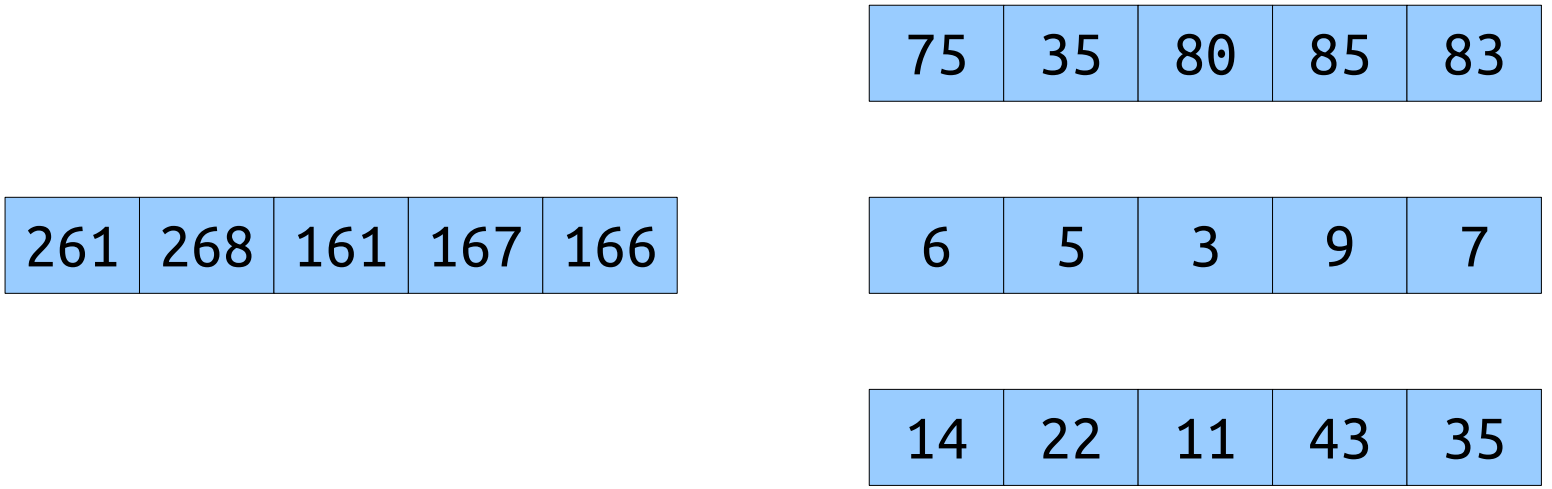
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

261	268	161	167	166

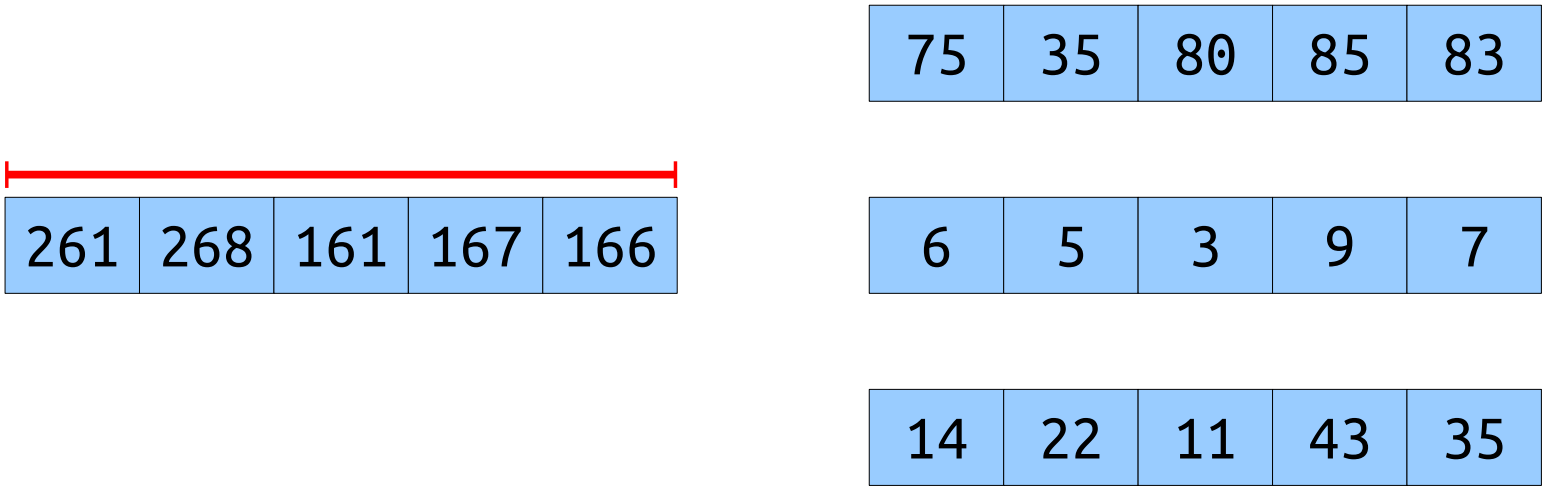
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



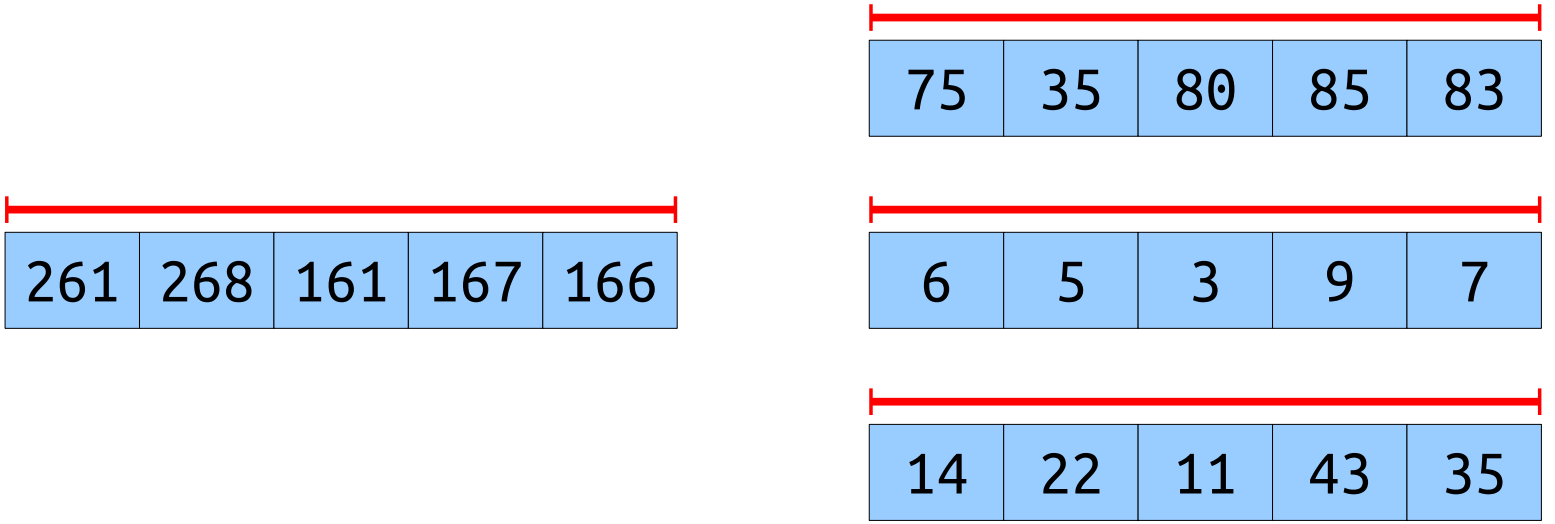
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



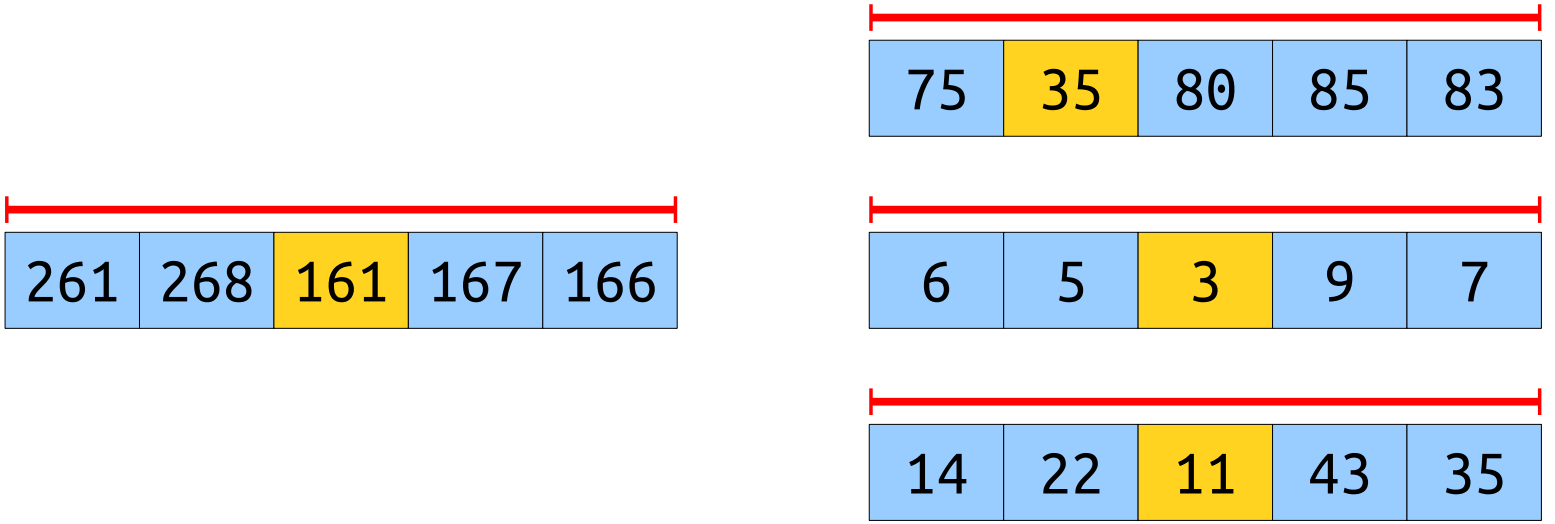
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



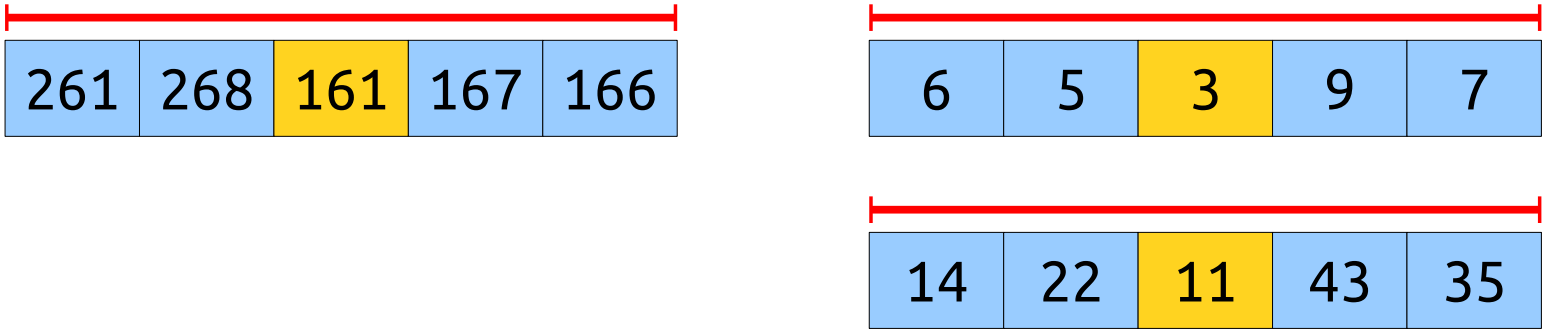
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

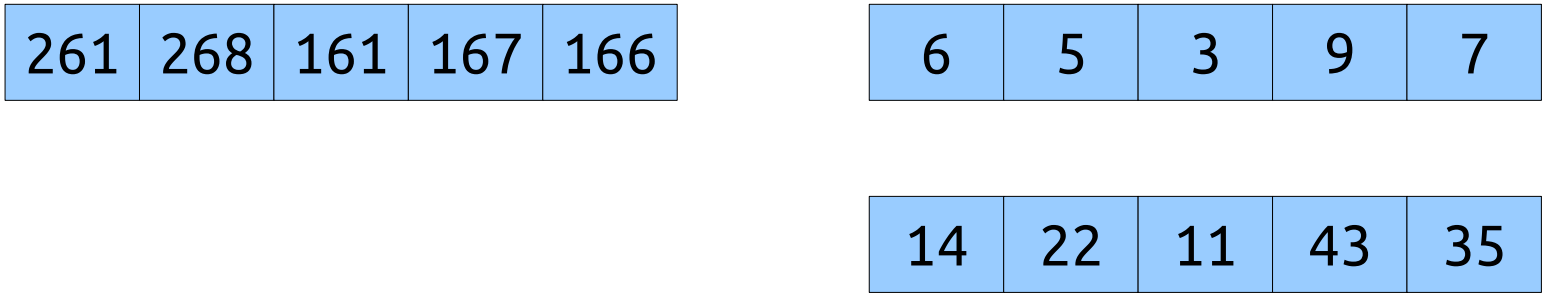
261	268	161	167	166

6	5	3	9	7

14	22	11	43	35

An Observation

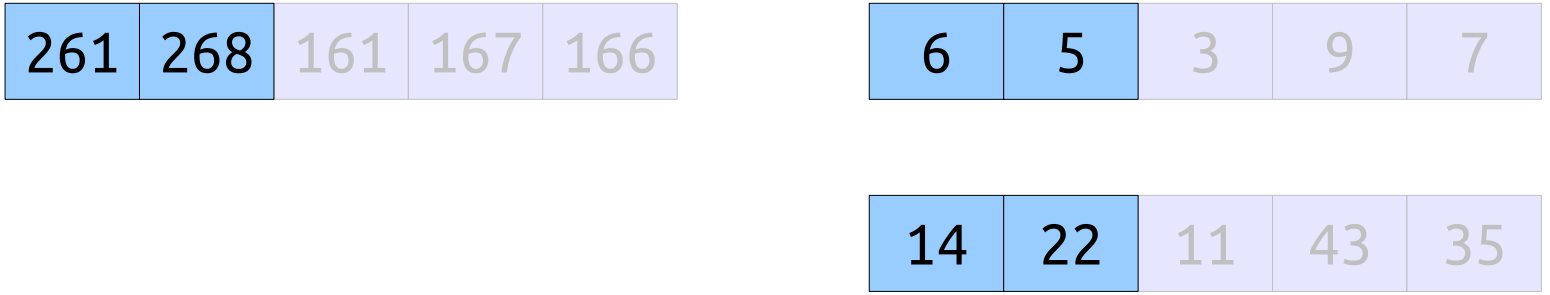
- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

An Observation

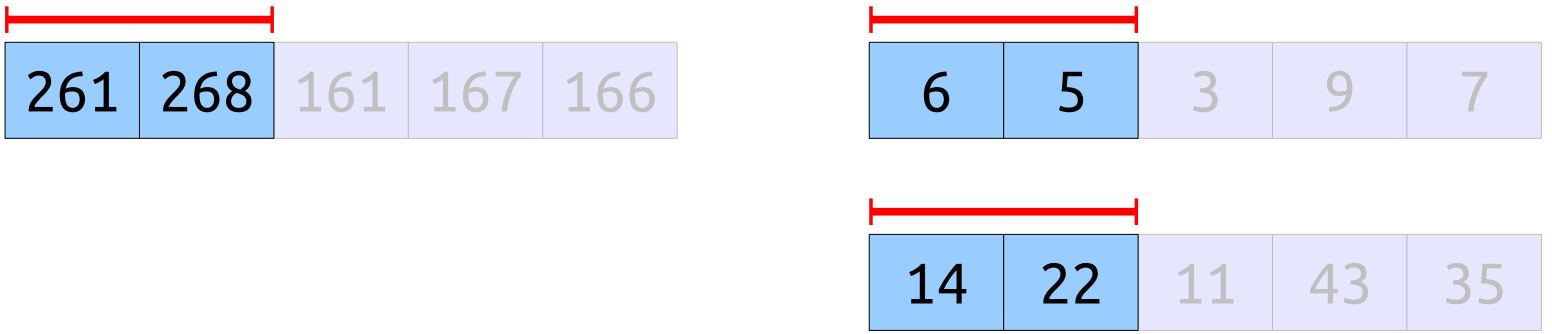
- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

An Observation

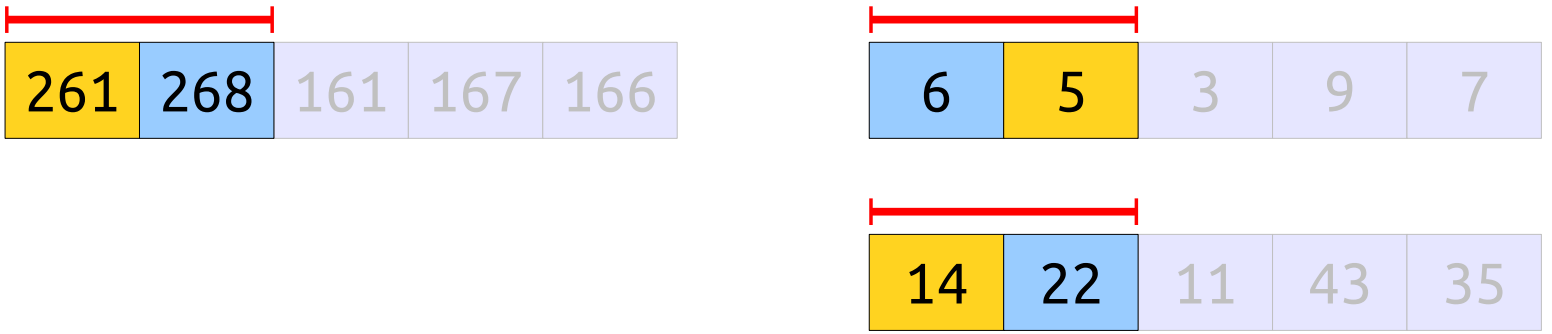
- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.



- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

An Observation

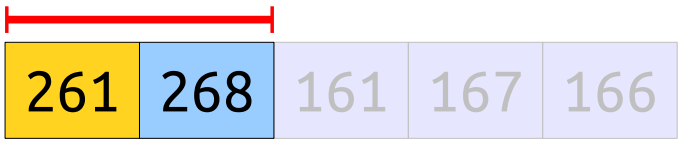
- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

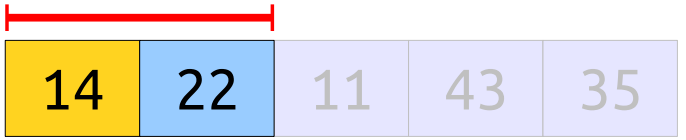


- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.





- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

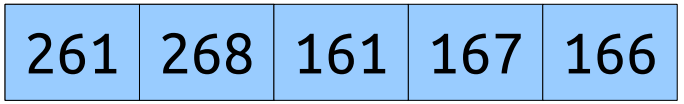
An Observation

- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

An Observation

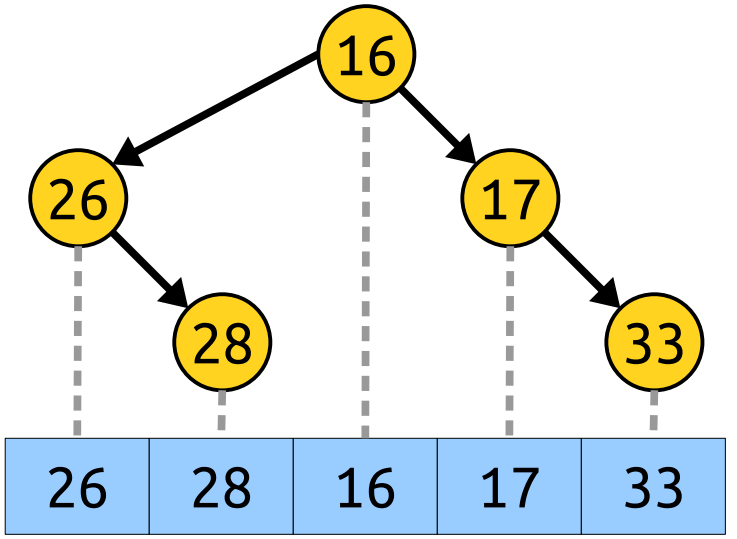
- ***Claim:*** If $B_1 \sim B_2$, the minimum elements of B_1 and B_2 must occur at the same position.

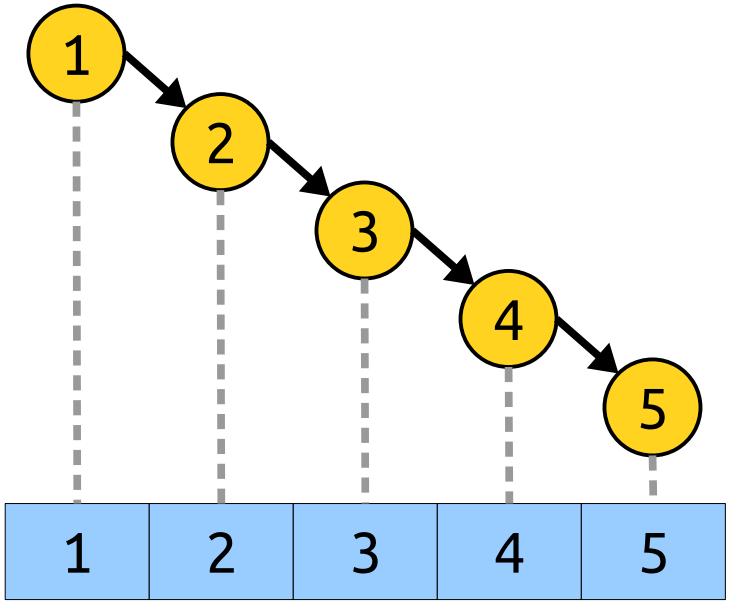


- ***Claim:*** This property must hold recursively on the subarrays to the left and right of the minimum.

Cartesian Trees

- A ***Cartesian tree*** for an array is a binary tree built as follows:
  - The root of the tree is the minimum element of the array.
  - Its left and right subtrees are formed by recursively building Cartesian trees for the subarrays to the left and right of the minimum.
  - (Base case: if the array is empty, the Cartesian tree is empty.)
- This is ***mechanical description*** of Cartesian trees; it defines Cartesian trees by showing how to make them.



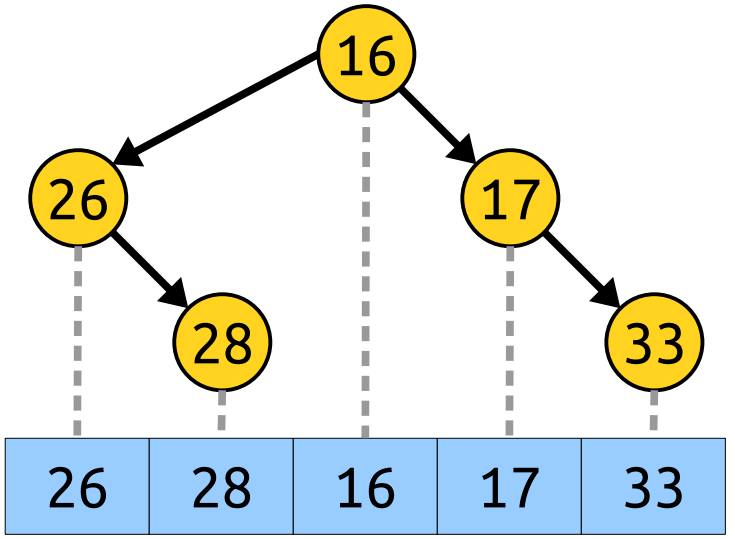


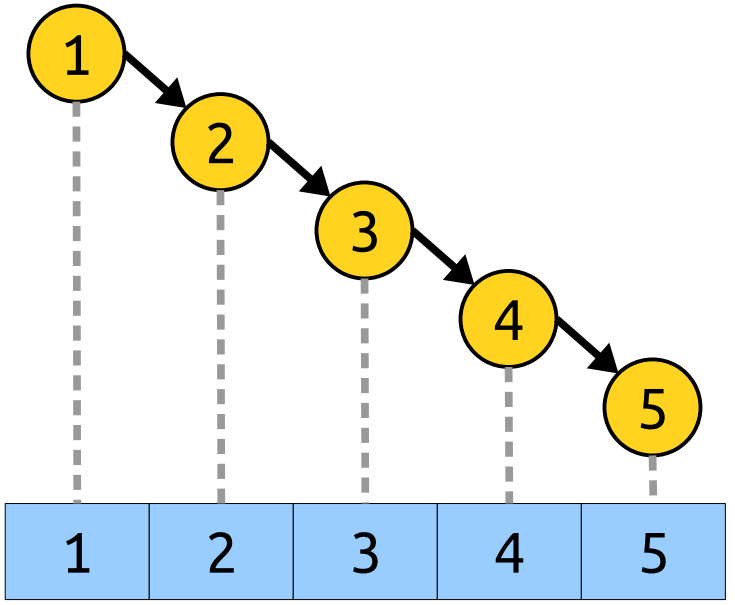
Cartesian Trees

- A ***Cartesian tree*** can also be defined as follows:

  The Cartesian tree for an array is a binary tree obeying the min-heap property whose inorder traversal gives back the original array.

- This is called an ***operational description***; it says what properties the tree has rather than how to find it.
- Having multiple descriptions of the same object is incredibly useful – this will be a recurring theme this quarter!

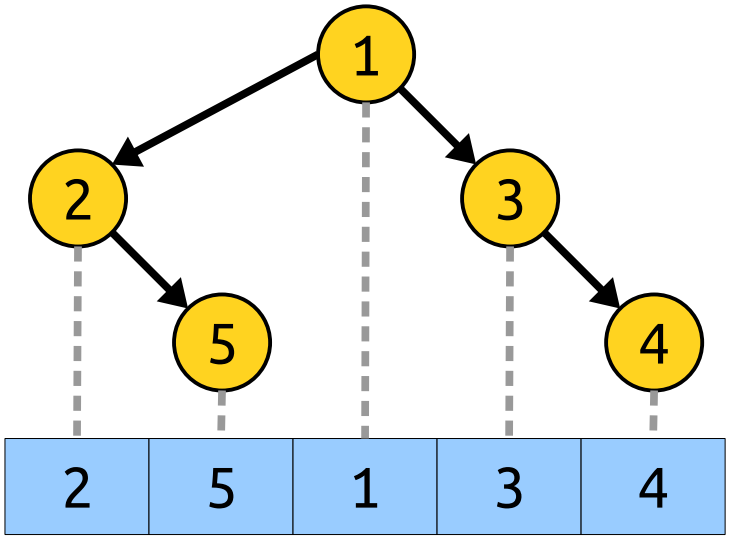


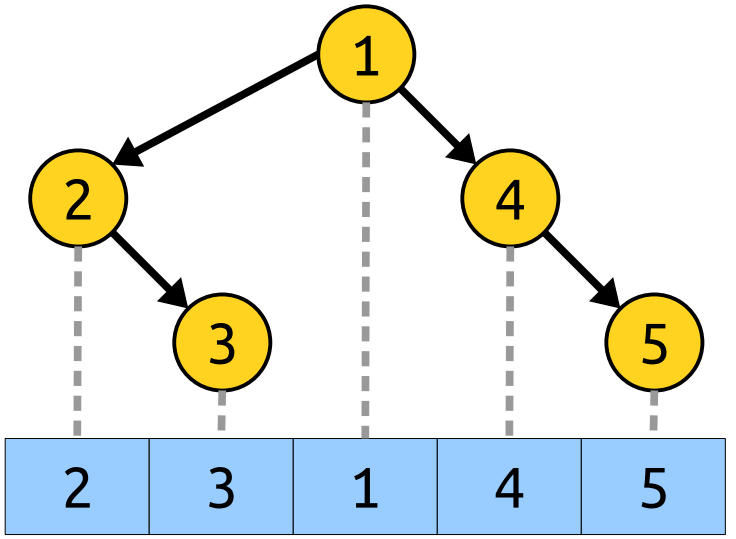


Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.

"same shape"





Cartesian Trees and RMQ

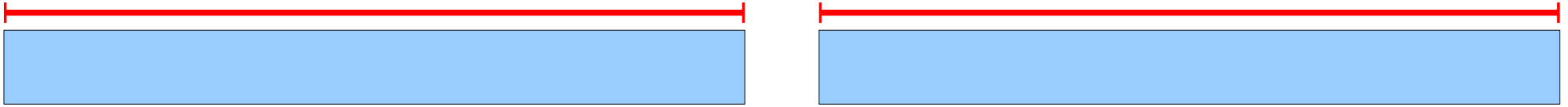
- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - ($\Rightarrow$) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.

Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - ($\Rightarrow$) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.

Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.

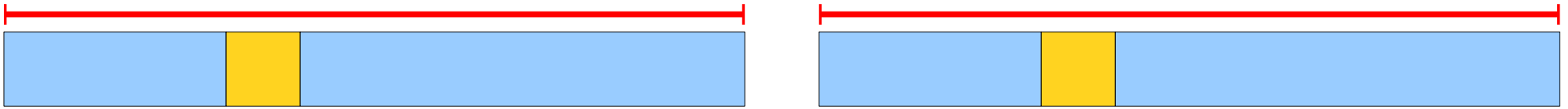


Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.

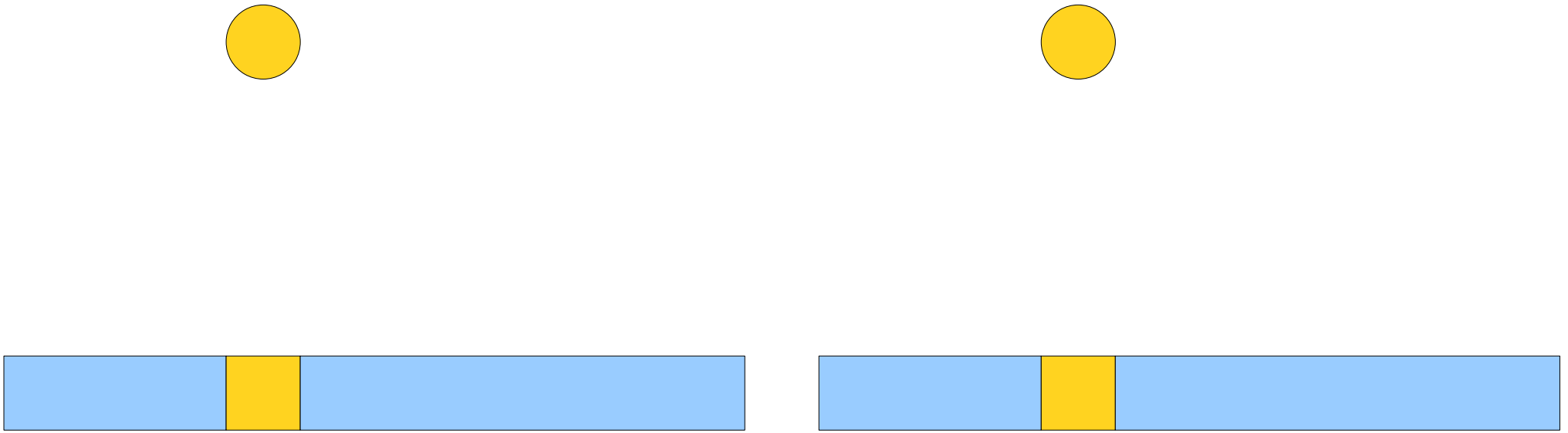
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.



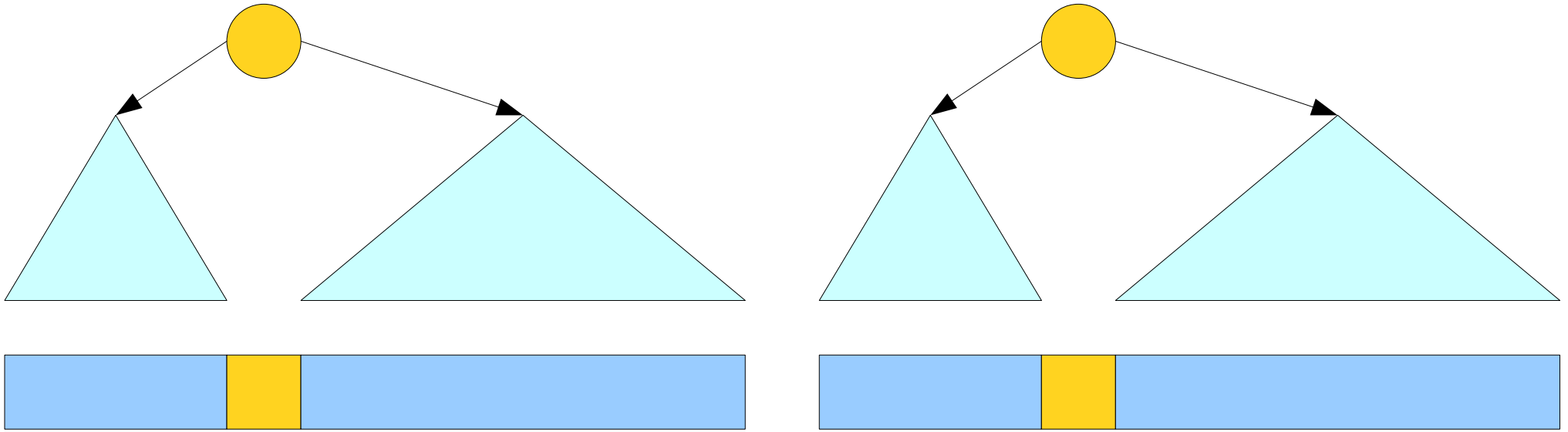
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.



Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇒) Induction. B_1 and B_2 have equal RMQs, so corresponding ranges have minima at the same positions.



Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - ($\Leftarrow$) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.

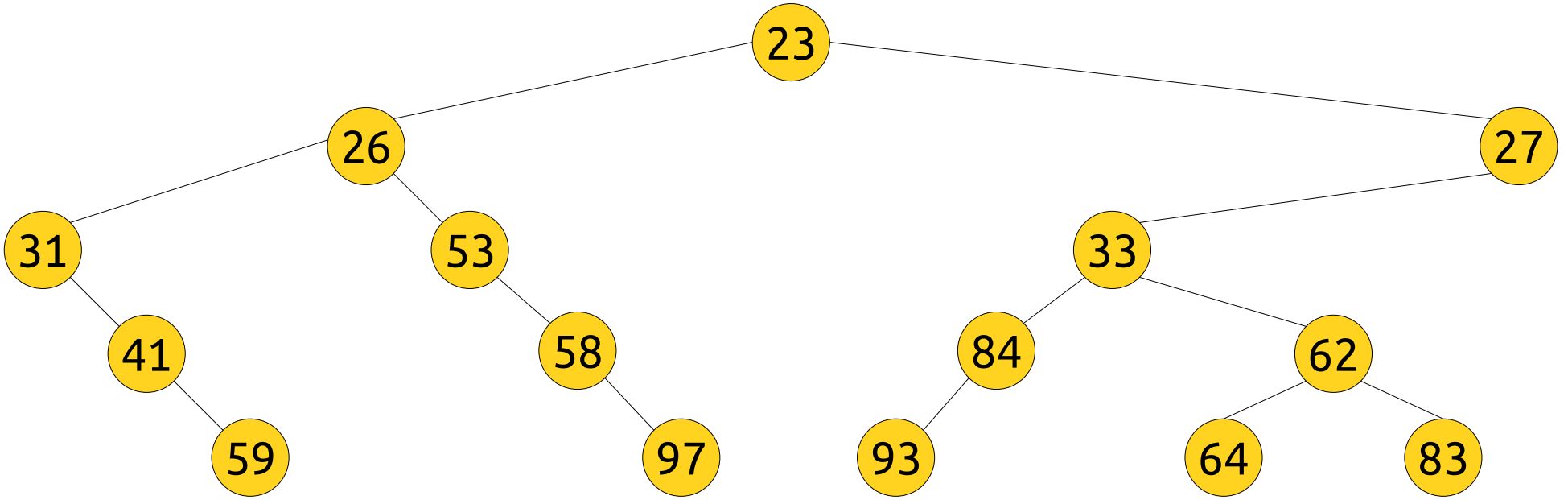
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.

31	41	59	26	53	58	97	23	93	84	33	64	62	83	27

Cartesian Trees and RMQ

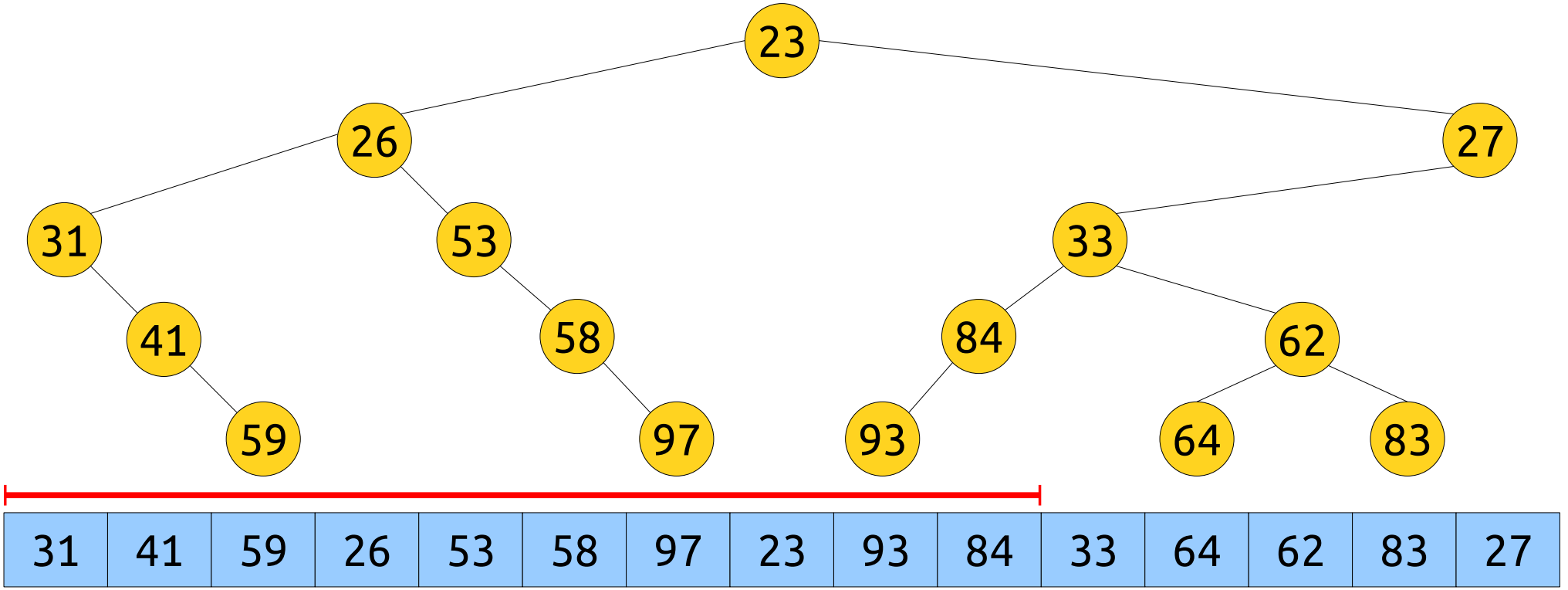
- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



31	41	59	26	53	58	97	23	93	84	33	64	62	83	27

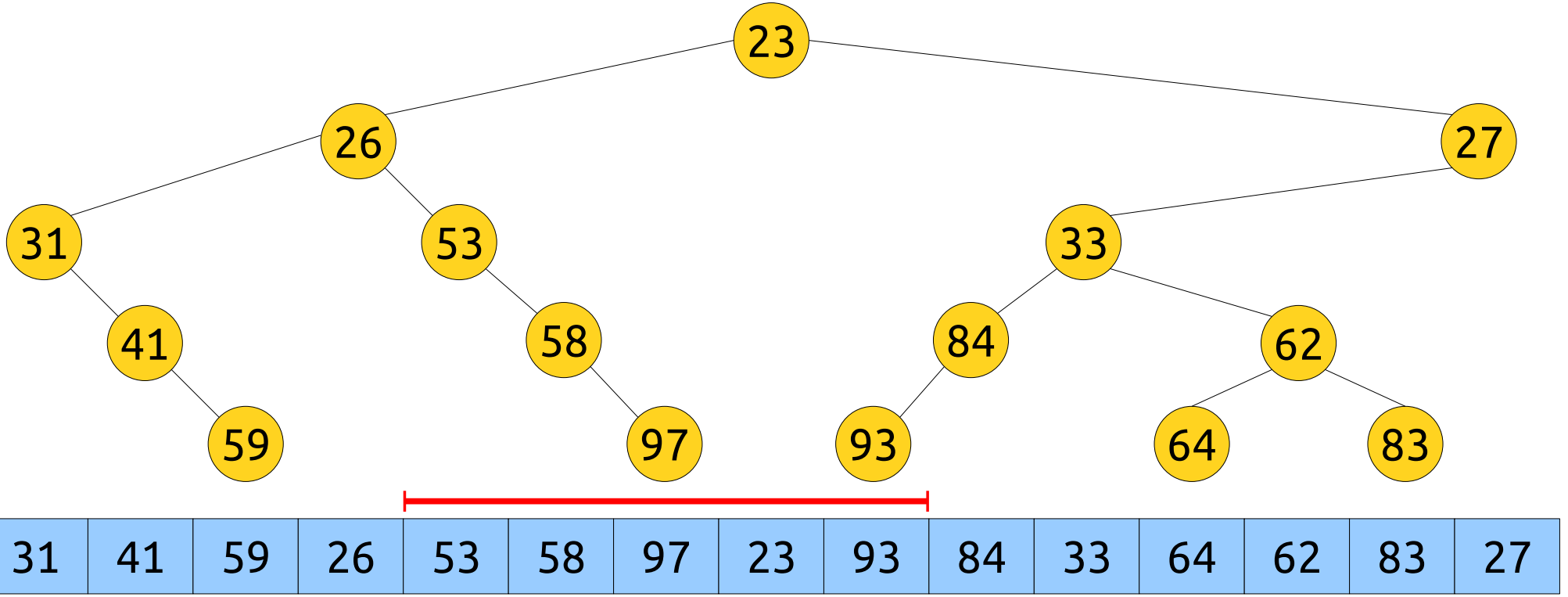
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



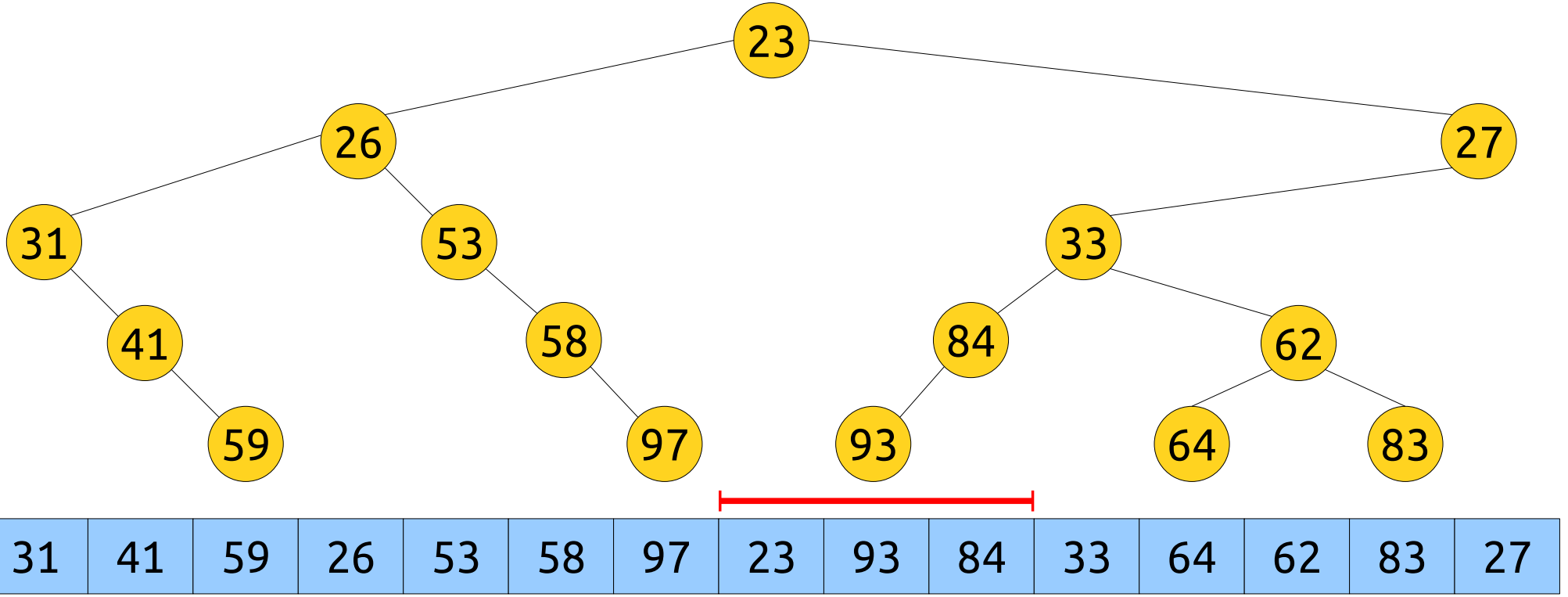
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



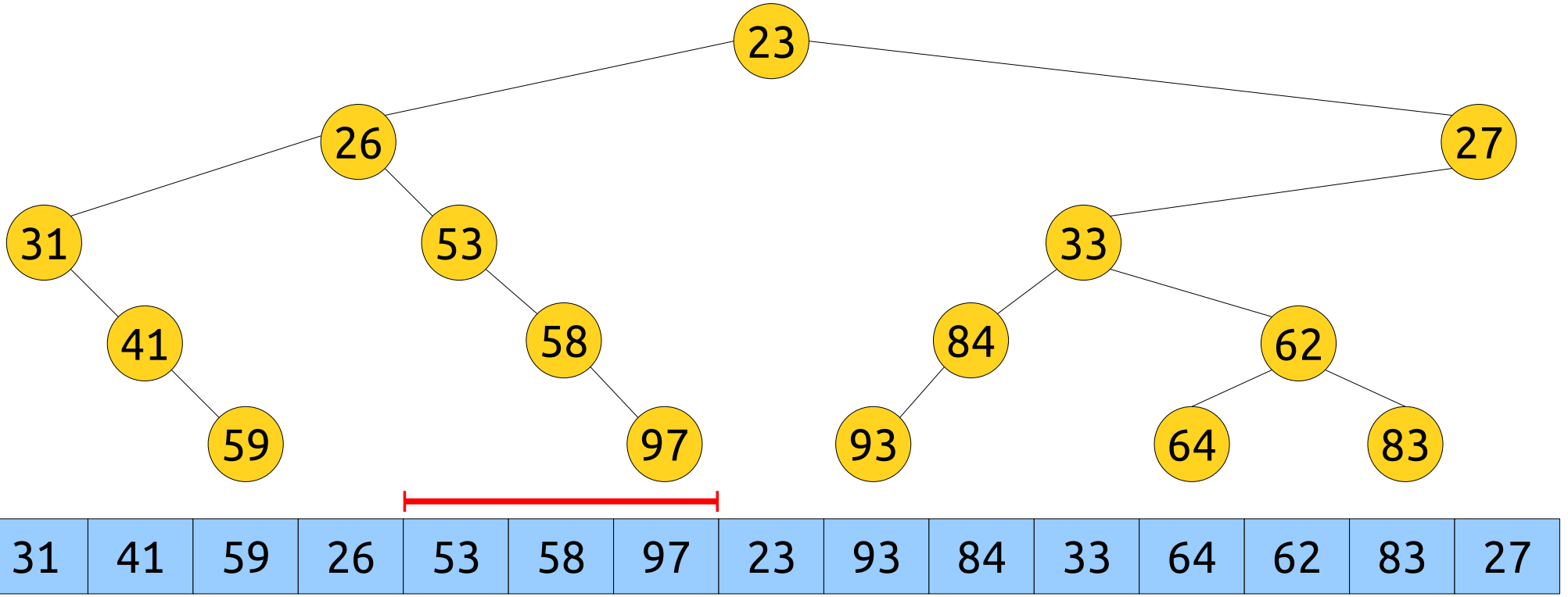
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



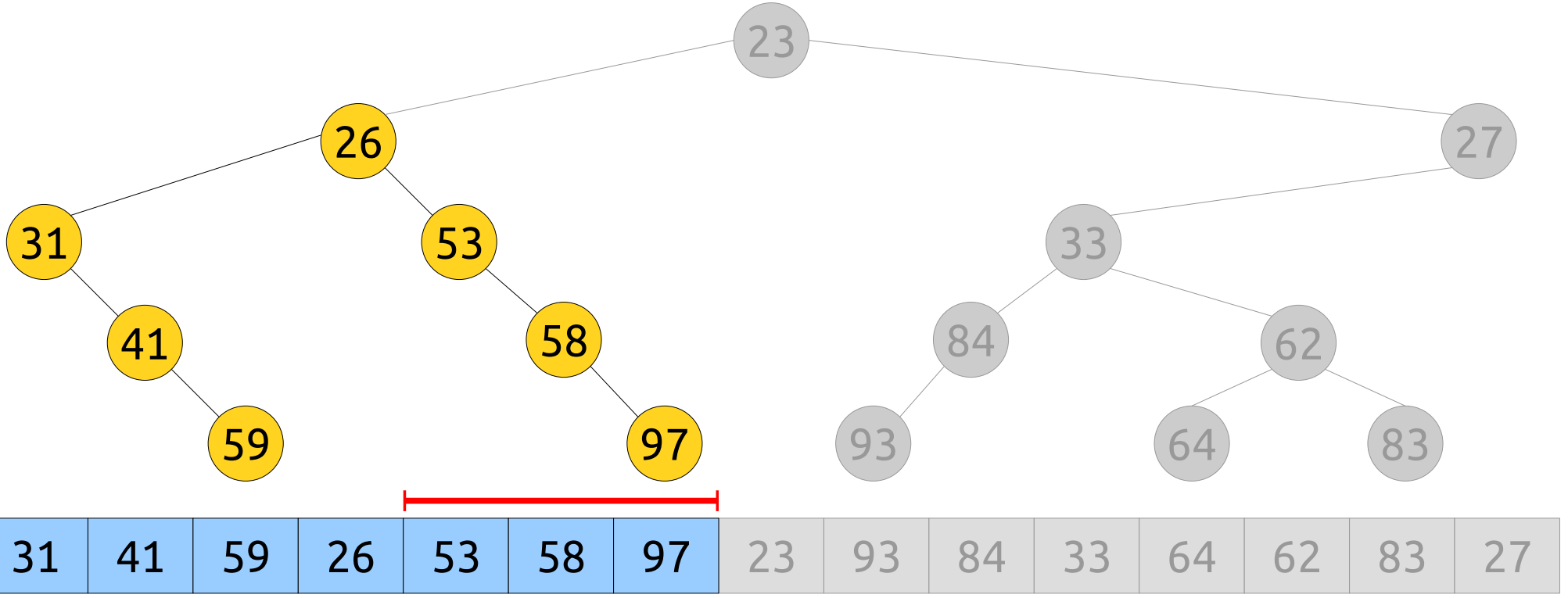
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



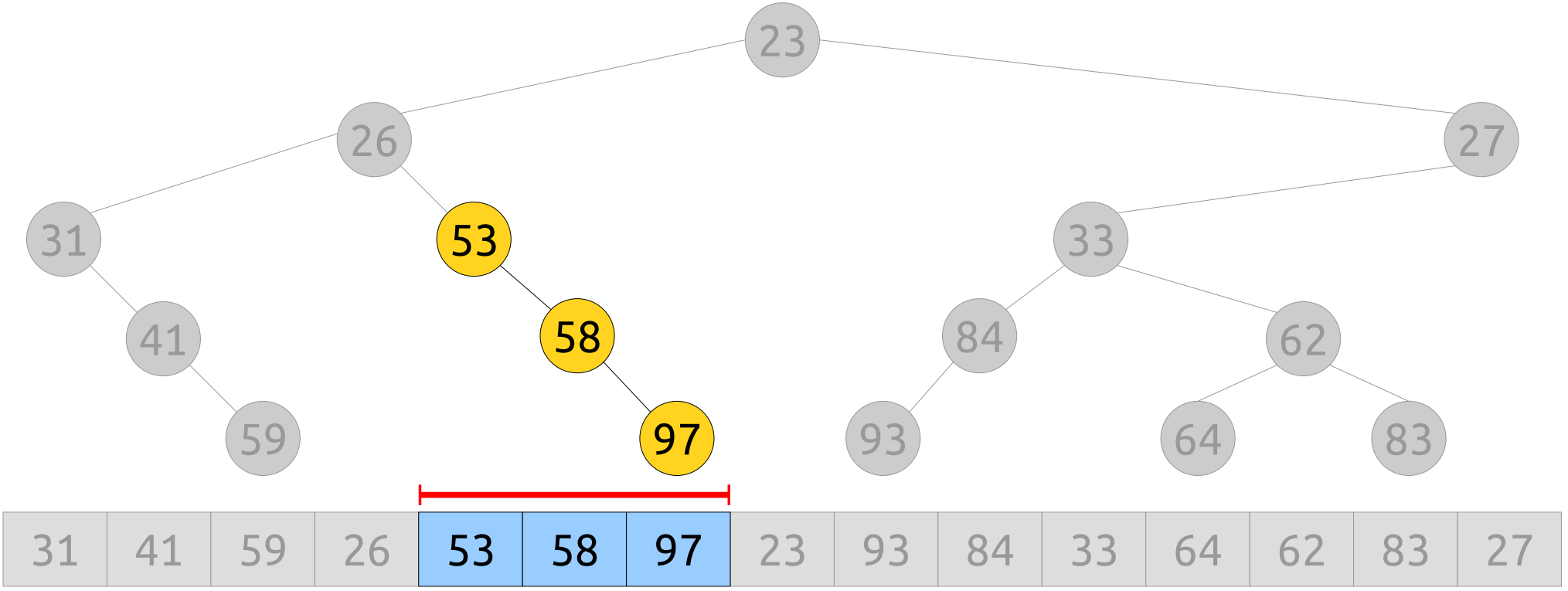
Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



Cartesian Trees and RMQ

- ***Theorem:*** Let B_1 and B_2 be blocks of length b. Then $B_1 \sim B_2$ iff B_1 and B_2 have isomorphic Cartesian trees.
- ***Proof sketch:***
  - (⇐) Induction. It's possible to answer RMQ using a recursive walk on the Cartesian tree.



How can we tell when two blocks can share RMQ structures?

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When those blocks have isomorphic Cartesian trees!

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When those blocks have isomorphic Cartesian trees!
But how do we check that?

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When those blocks have isomorphic Cartesian trees!
But how do we check that?

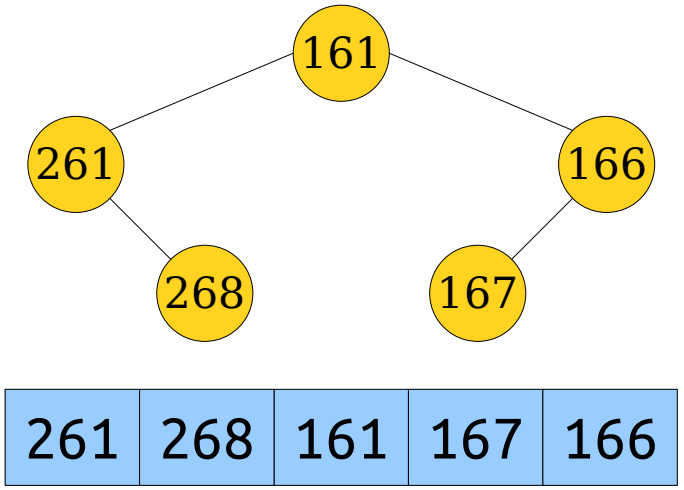
How many block types are there, as a function of b?

¯_(ツ)_/¯

How quickly can we build a Cartesian tree?

Building Cartesian Trees

- Here's a naïve algorithm for constructing Cartesian trees:
  - Find the minimum value.
  - Recursively build a Cartesian tree for the array to the left of the minimum.
  - Recursively build a Cartesian tree with the elements to the right of the minimum.
  - Return the overall tree.
- How efficient is this approach?



Building Cartesian Trees

- This algorithm works by
  - doing a linear scan over the array to find the minimum value, then
  - recursively processing the left and right halves on the array.
- This is a divide-and-conquer algorithm! Here's a runtime recurrence:

$$T(n) = T(n_{left}) + T(n_{right}) + O(n)$$

Building Cartesian Trees

- This algorithm works by
  - doing a linear scan over the array to find the minimum value, then
  - recursively processing the left and right halves on the array.
- This is a divide-and-conquer algorithm! Here's a runtime recurrence:

$$T(n) = T(n_{left}) + T(n_{right}) + O(n)$$

- ***Question:*** What does this recurrence solve to? (Hint: where have you seen this recurrence?)

Formulate a hypothesis!
Discuss with your neighbors!

Building Cartesian Trees

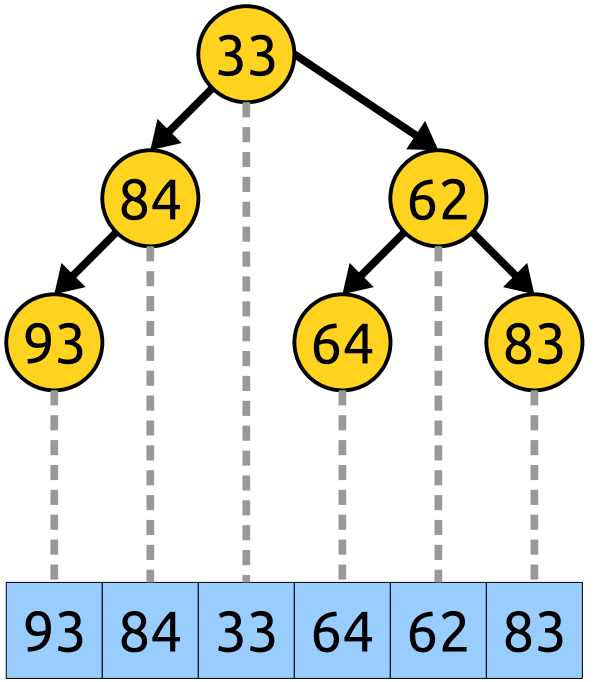
- This algorithm works by
  - doing a linear scan over the array to find the minimum value, then
  - recursively processing the left and right halves on the array.
- This is a divide-and-conquer algorithm! Here's a runtime recurrence:

$$T(n) = T(n_{left}) + T(n_{right}) + O(n)$$

- ***Question:*** What does this recurrence solve to? (Hint: where have you seen this recurrence?)
- This is the same recurrence relation that comes up in the analysis of quicksort!
  - If the min is always in the middle, runtime is $\Theta(n \log n)$.
  - If the min is always all the way to the side, runtime is $\Theta(n^2)$.
- ***Can we do better?***

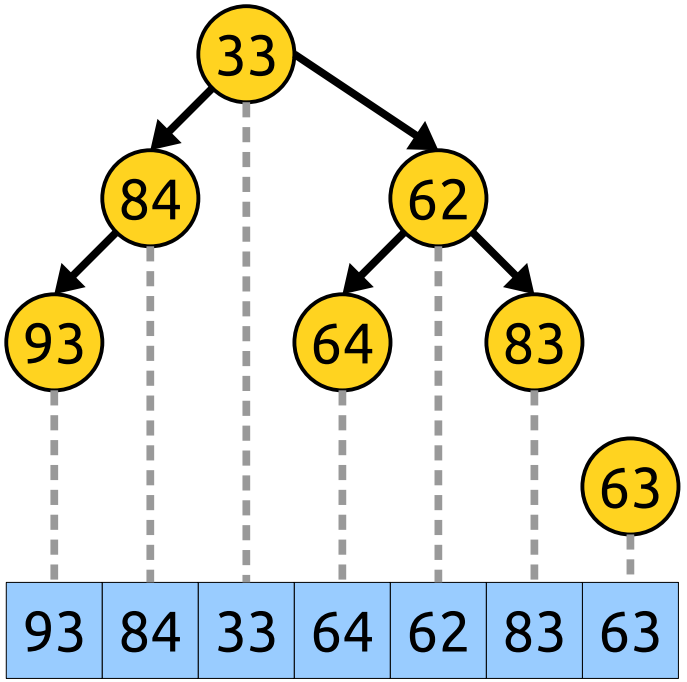
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



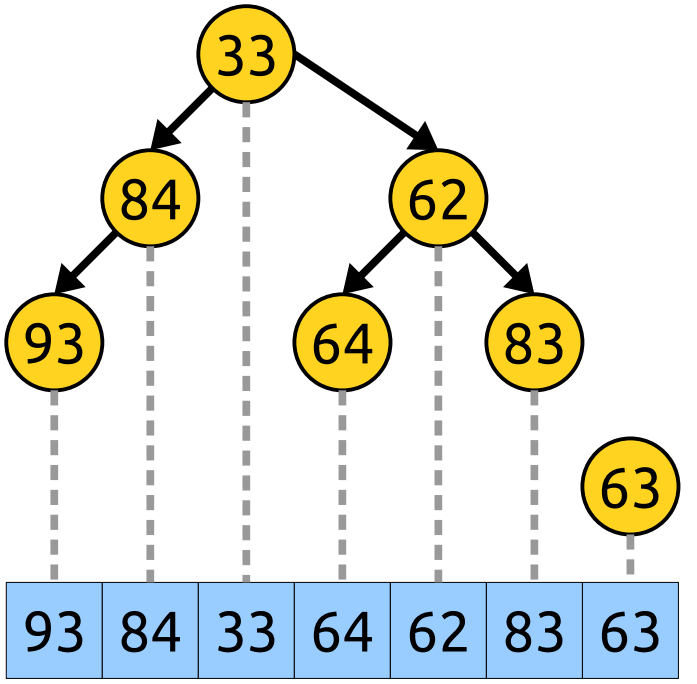
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



A Better Approach

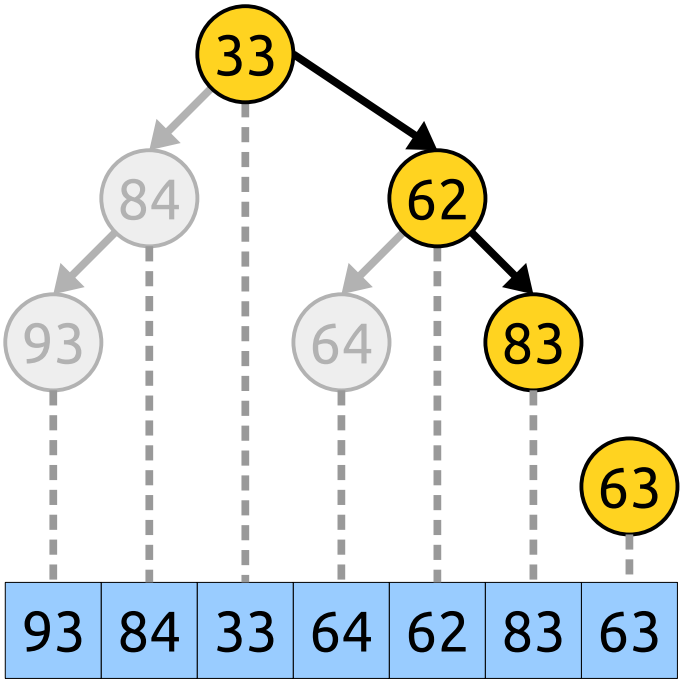
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 1: After adding this node, it must be the rightmost node in the tree. (An inorder traversal of a Cartesian tree gives back the original array.)

A Better Approach

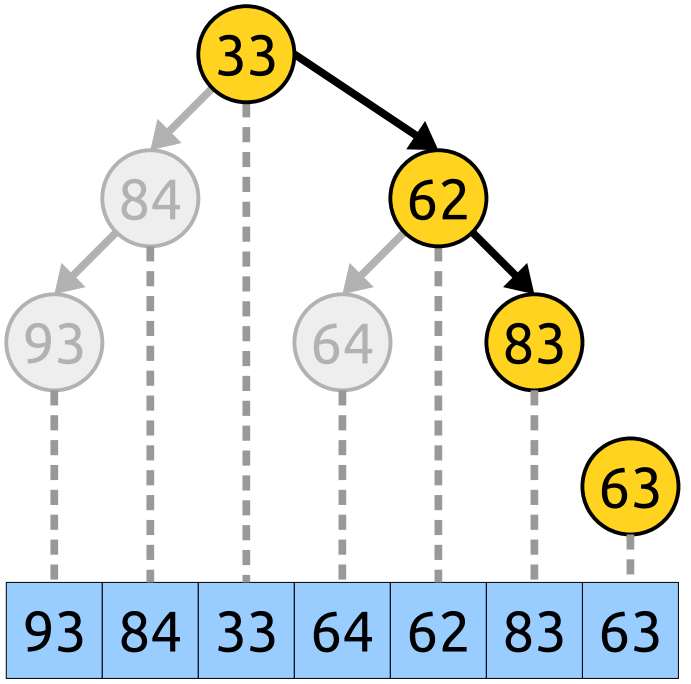
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 1: After adding this node, it must be the rightmost node in the tree. (An inorder traversal of a Cartesian tree gives back the original array.)

A Better Approach

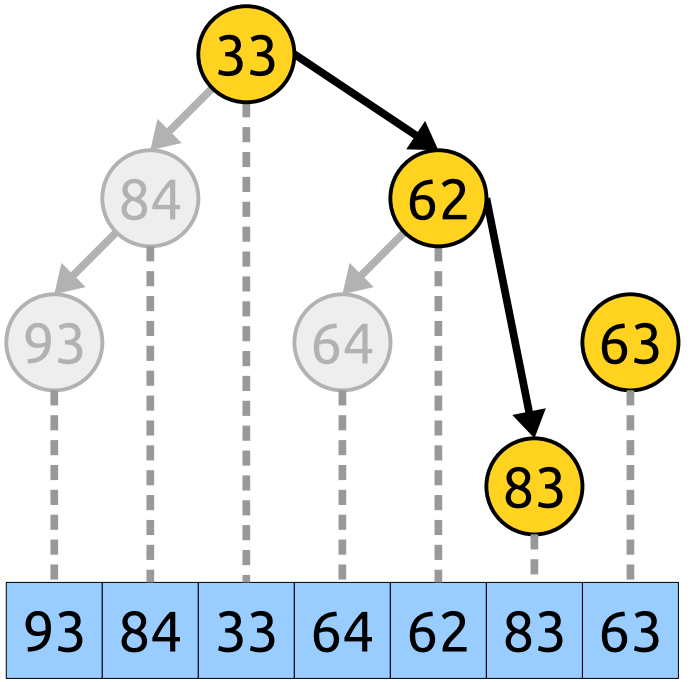
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 2: Cartesian trees are min-heaps (each node's value is at least as large as its parent's).

A Better Approach

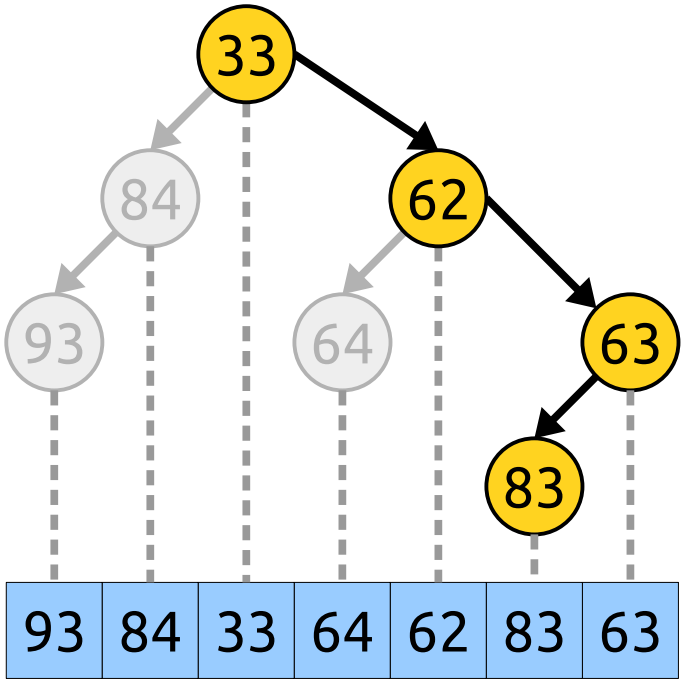
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 2: Cartesian trees are min-heaps (each node's value is at least as large as its parent's).

A Better Approach

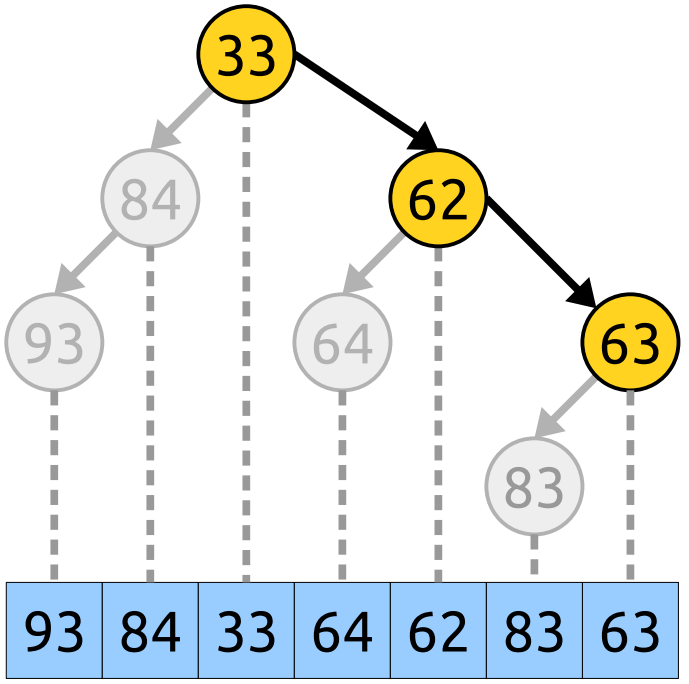
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 2: Cartesian trees are min-heaps (each node's value is at least as large as its parent's).

A Better Approach

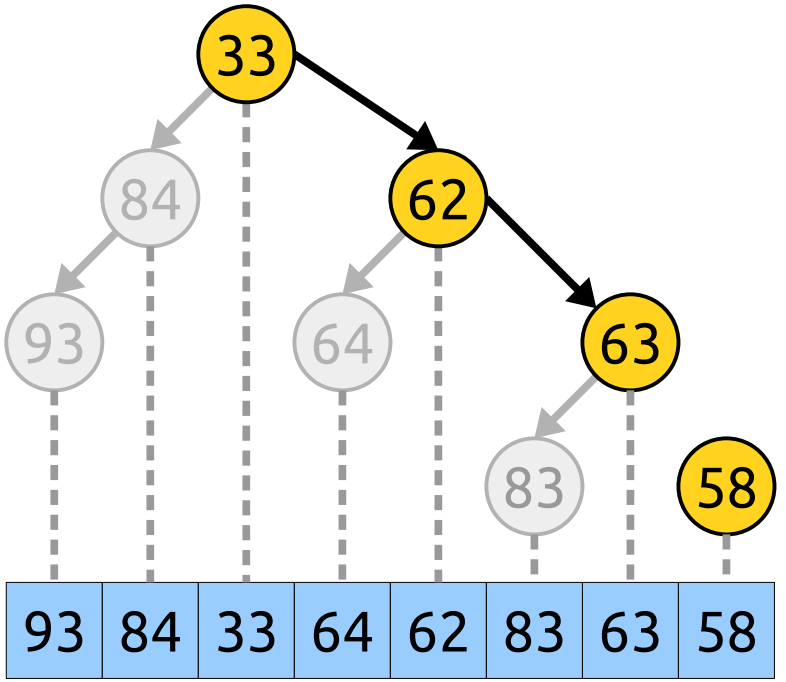
- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



Observation 2: Cartesian trees are min-heaps (each node's value is at least as large as its parent's).

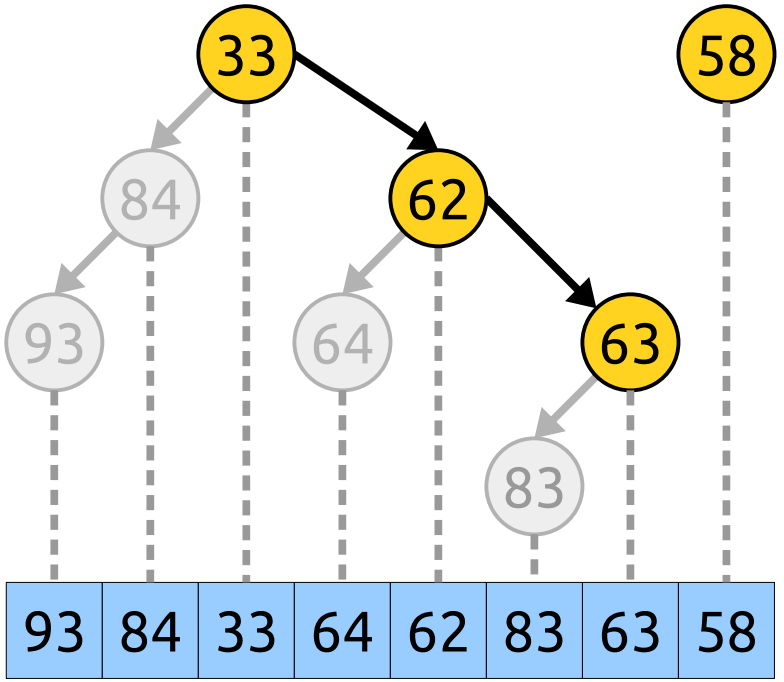
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



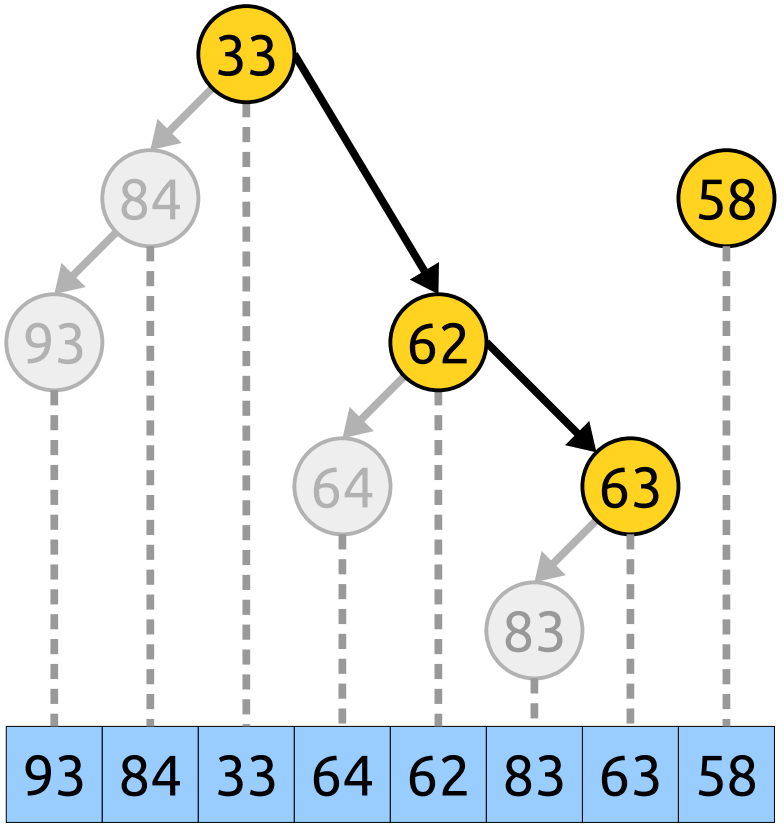
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



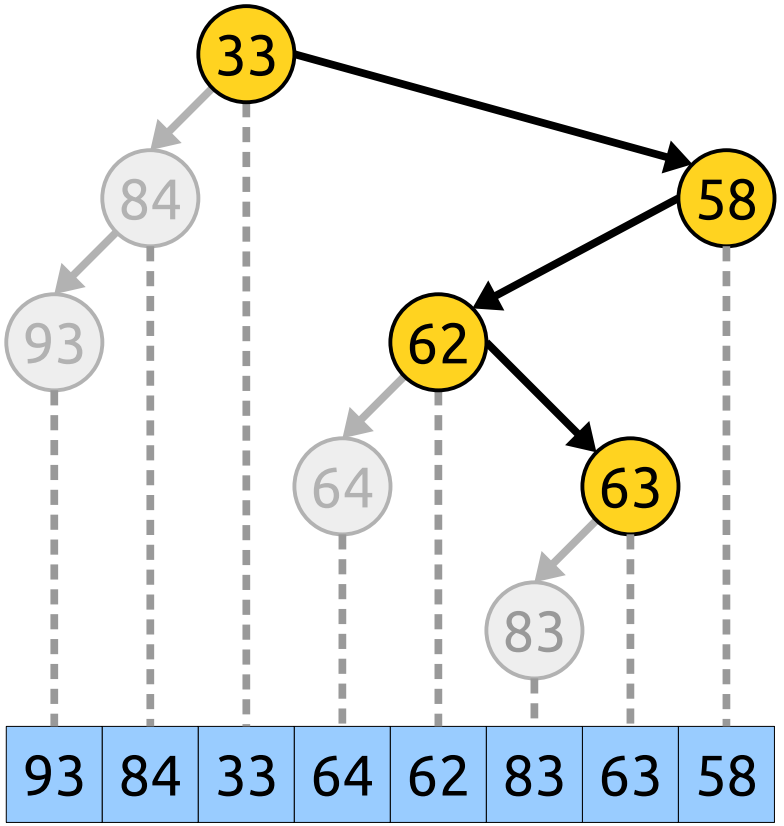
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



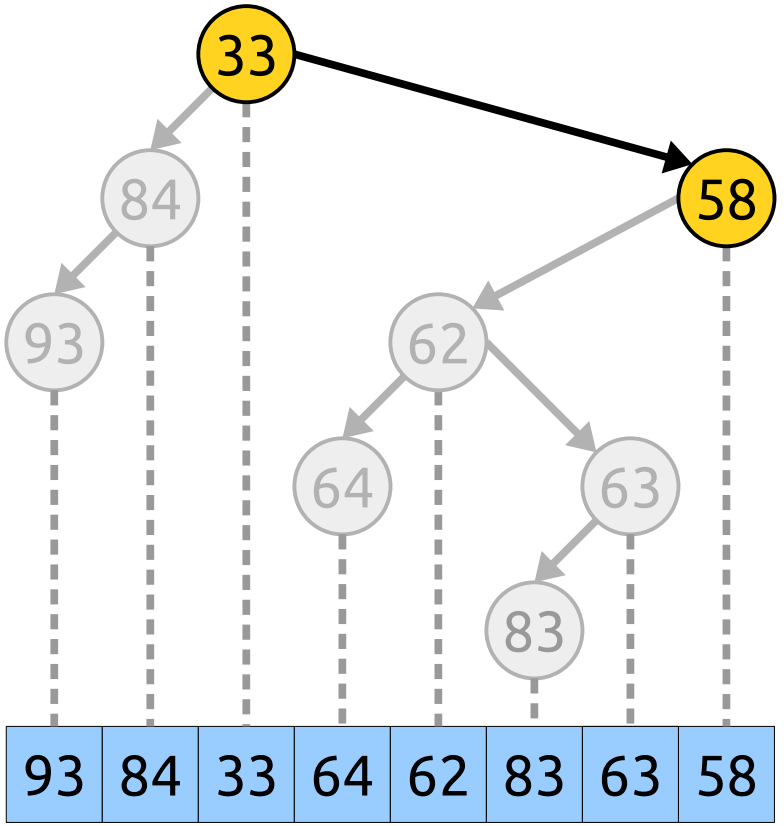
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



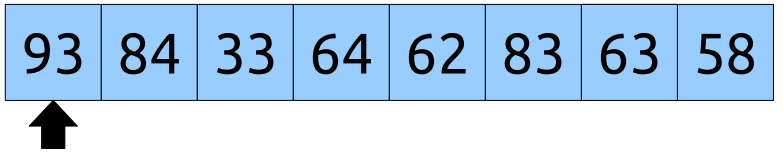
A Better Approach

- It's possible to build a Cartesian tree over an array of length k faster than the naive algorithm.
- ***High-level idea:*** Build a Cartesian tree for the first element, then the first two, then the first three, then the first four, etc.



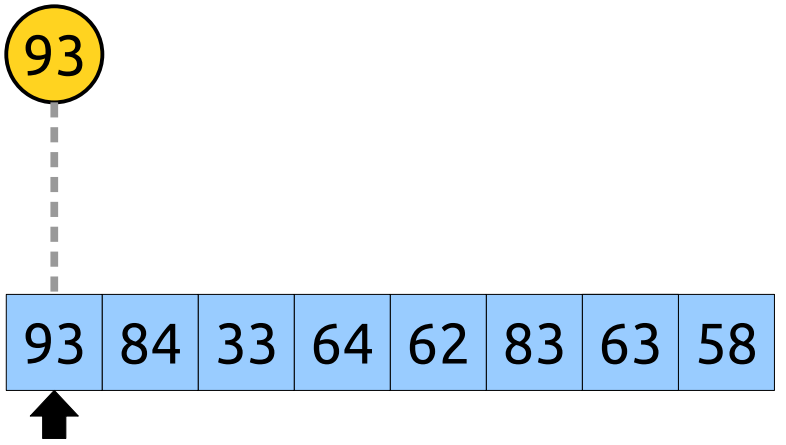
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



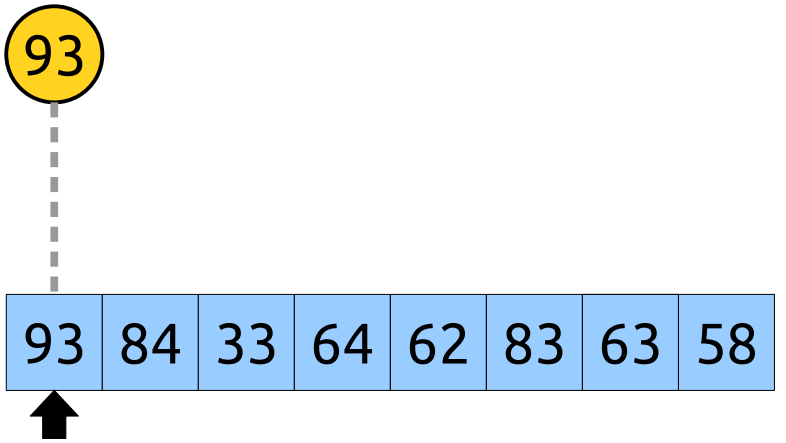
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



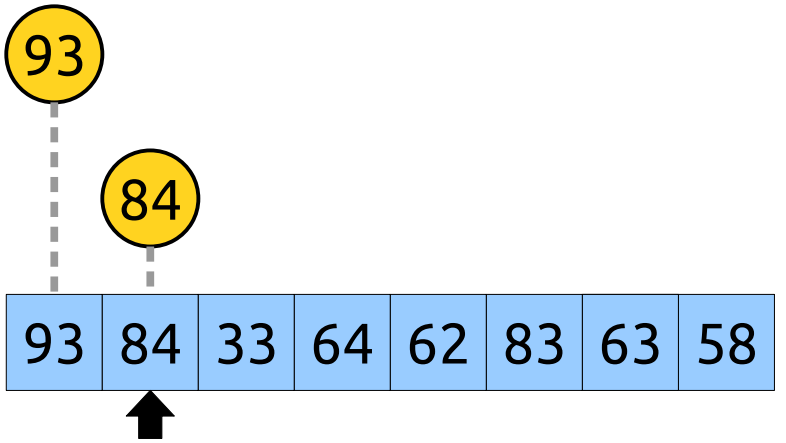
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



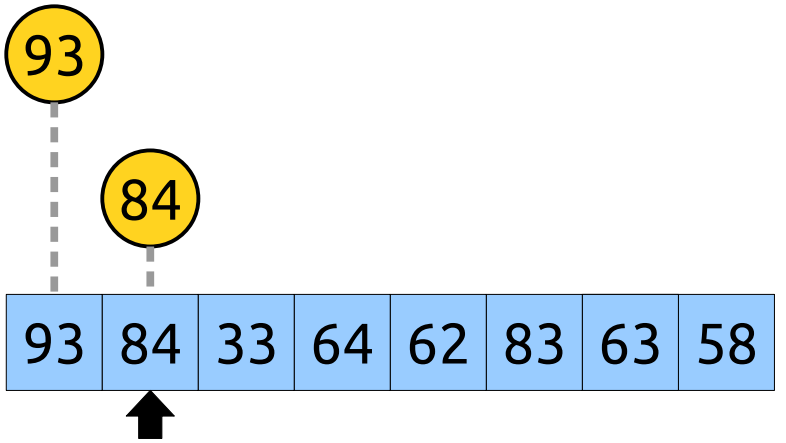
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



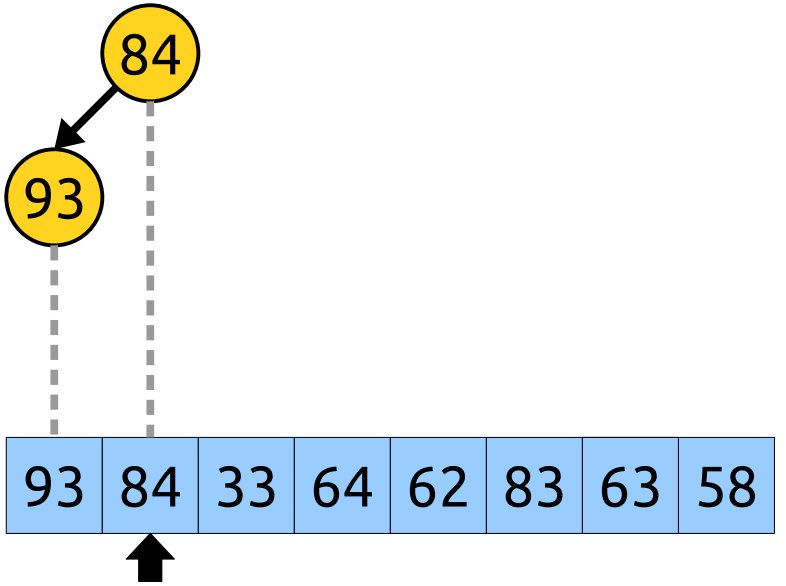
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



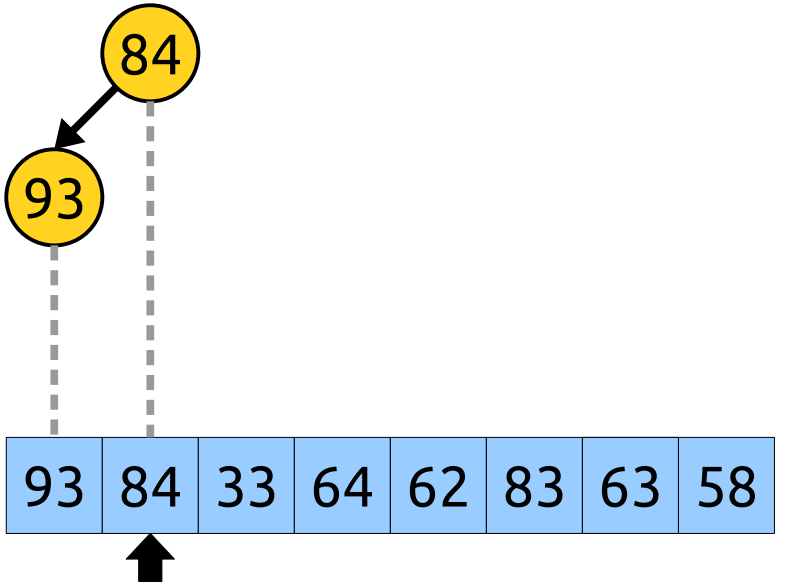
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



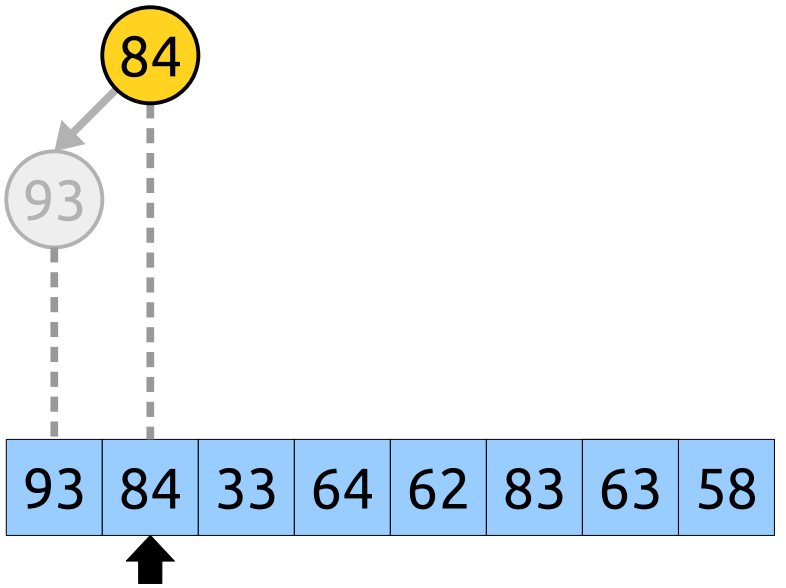
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



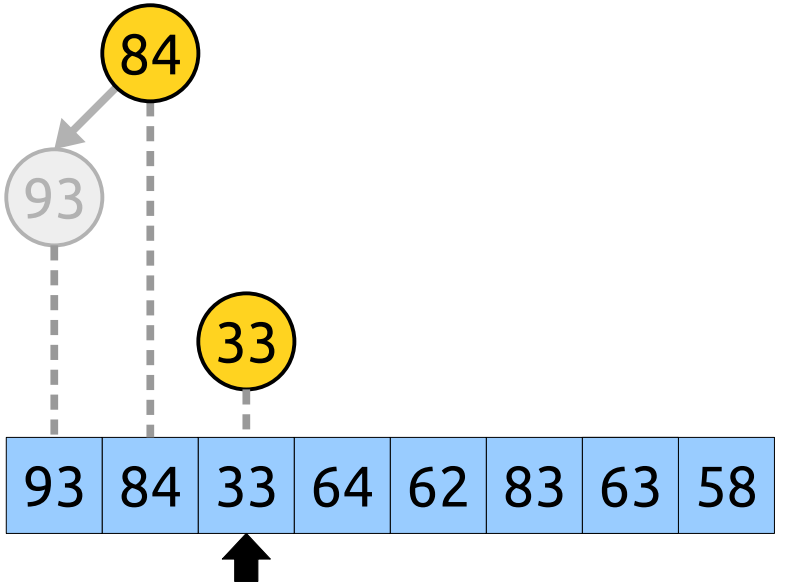
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



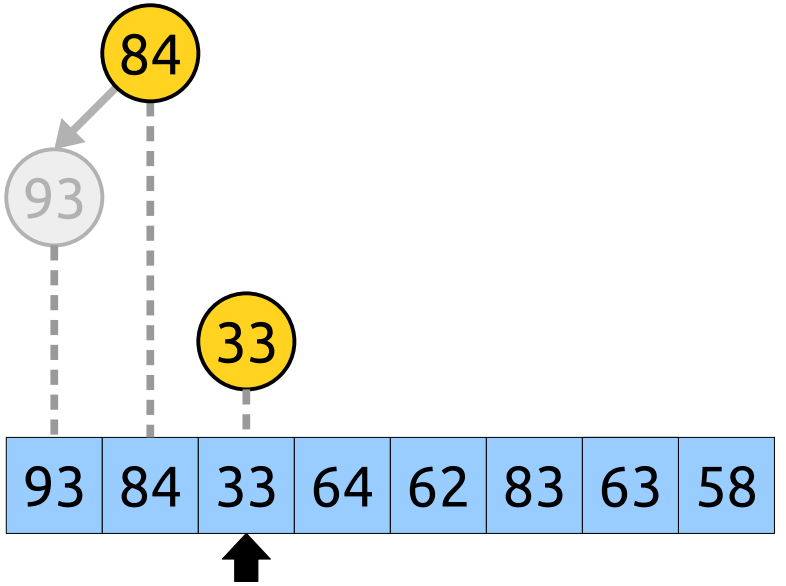
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



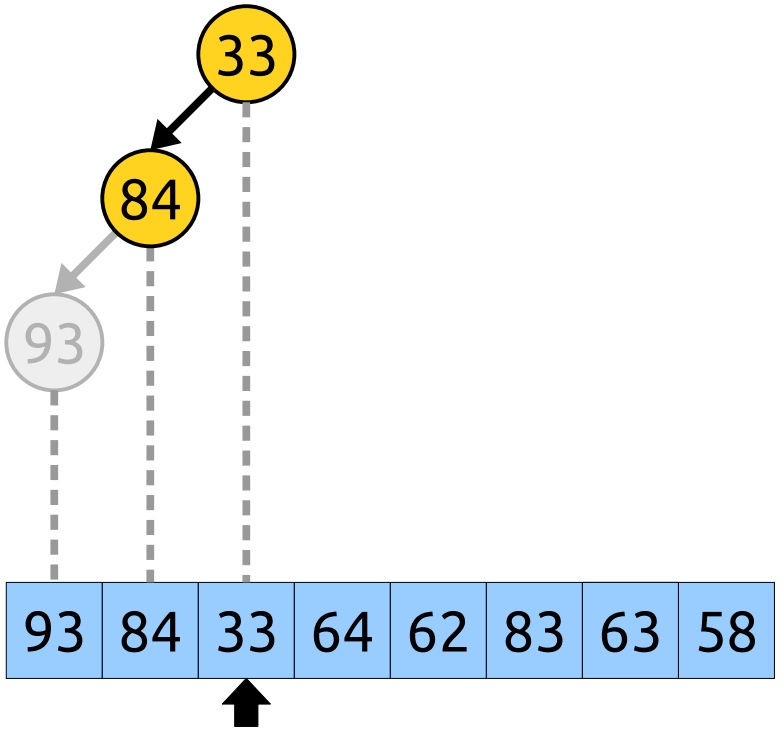
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



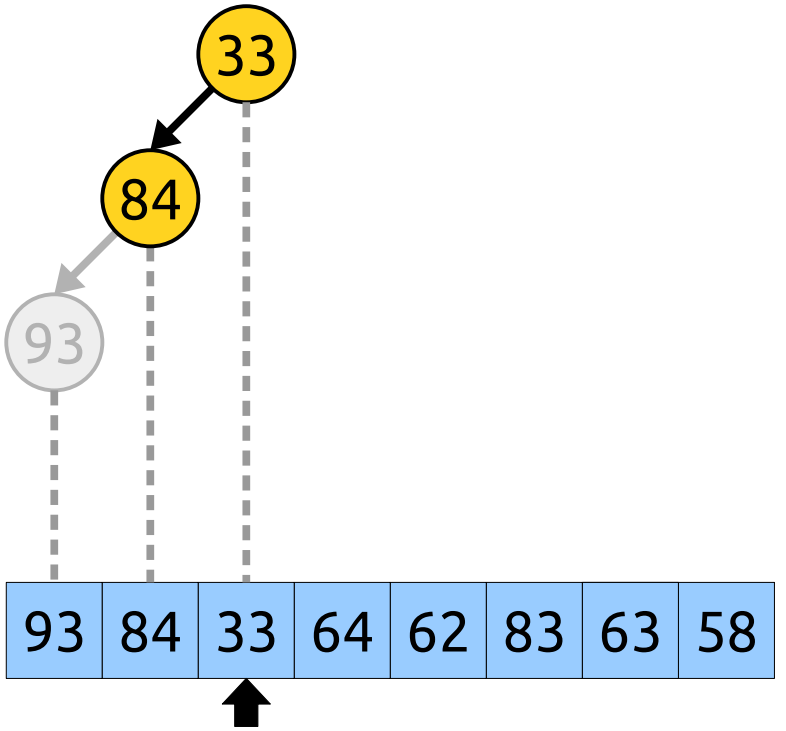
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



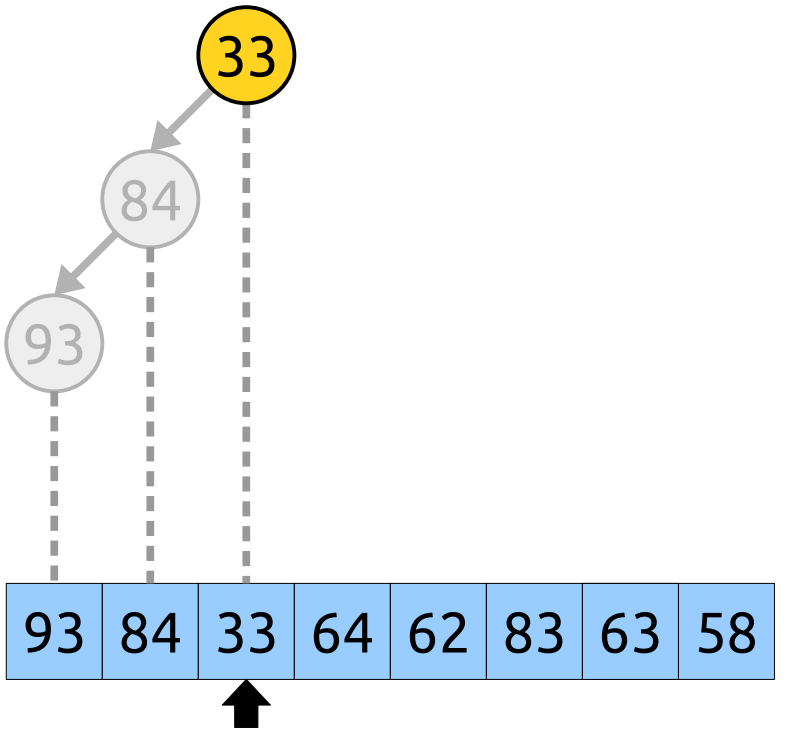
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



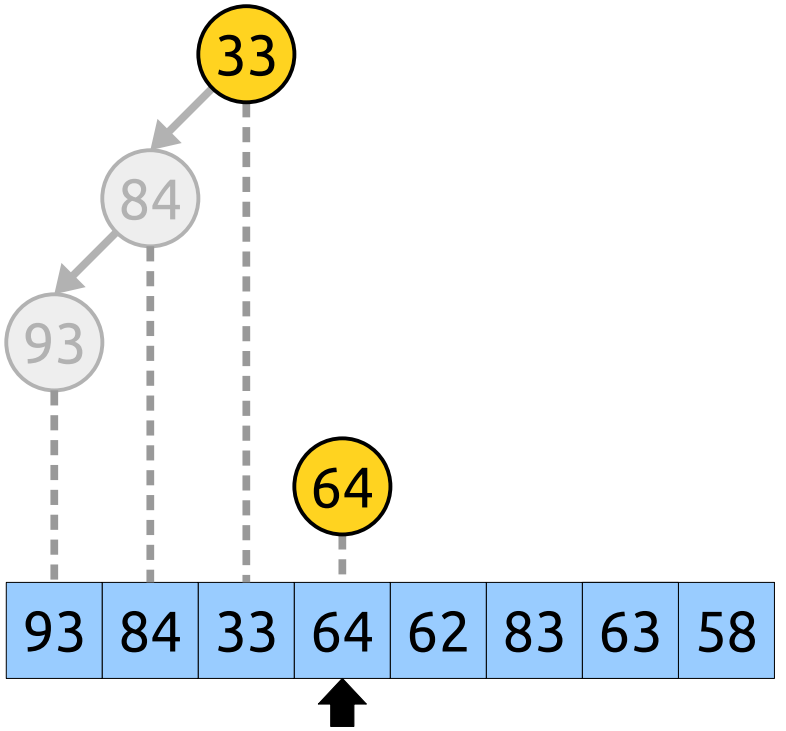
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



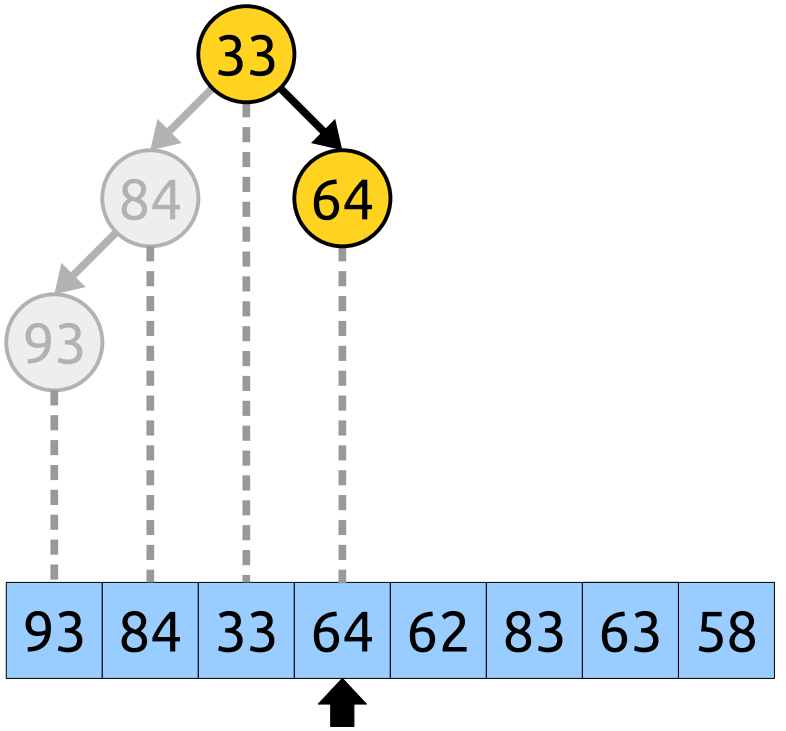
A Better Approach

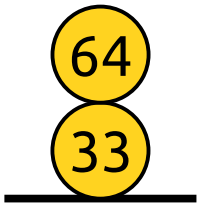
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



A Better Approach

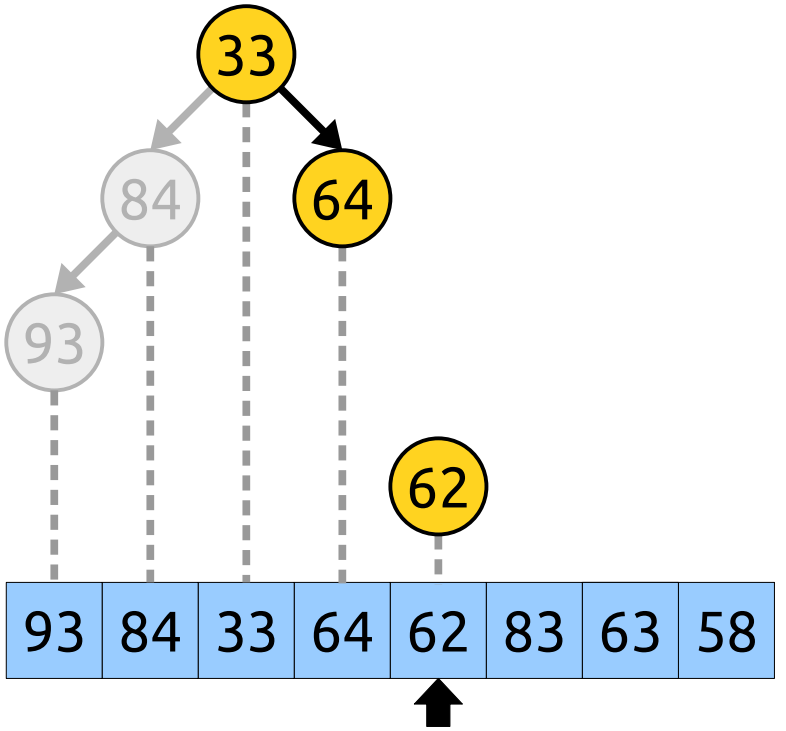
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

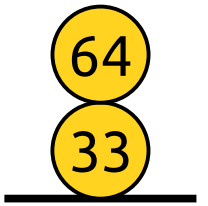




A Better Approach

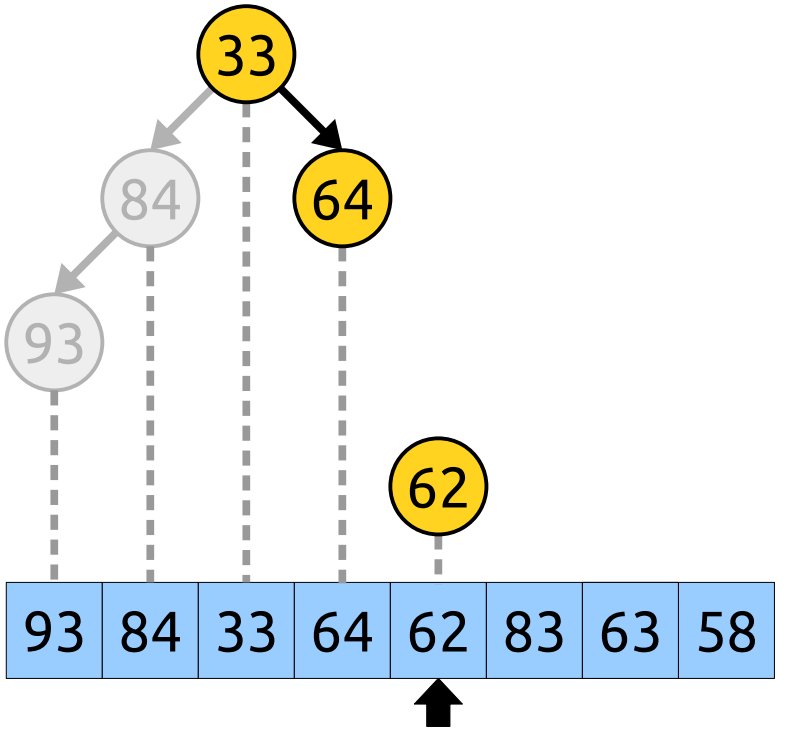
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

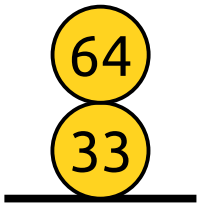




A Better Approach

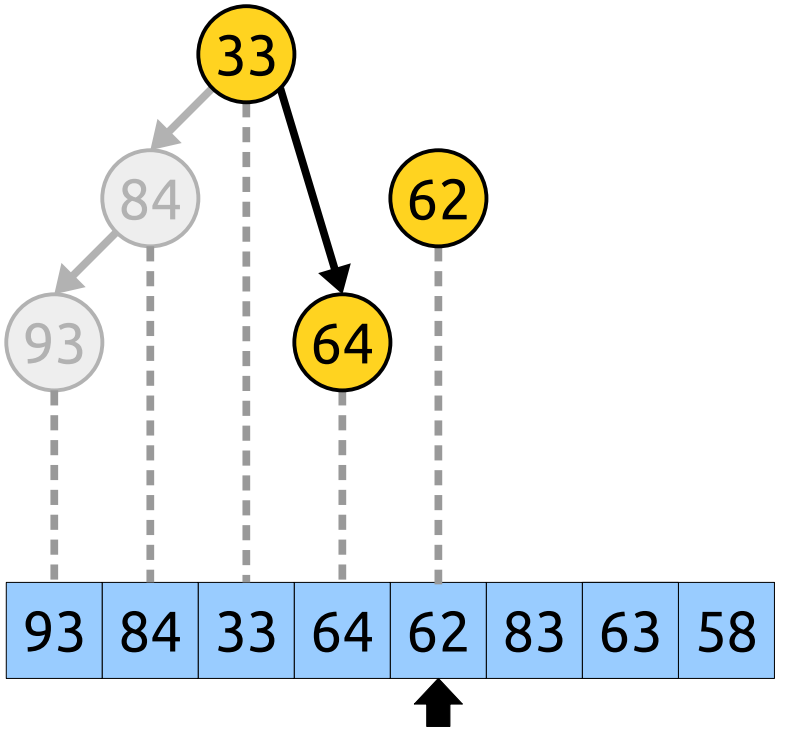
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.





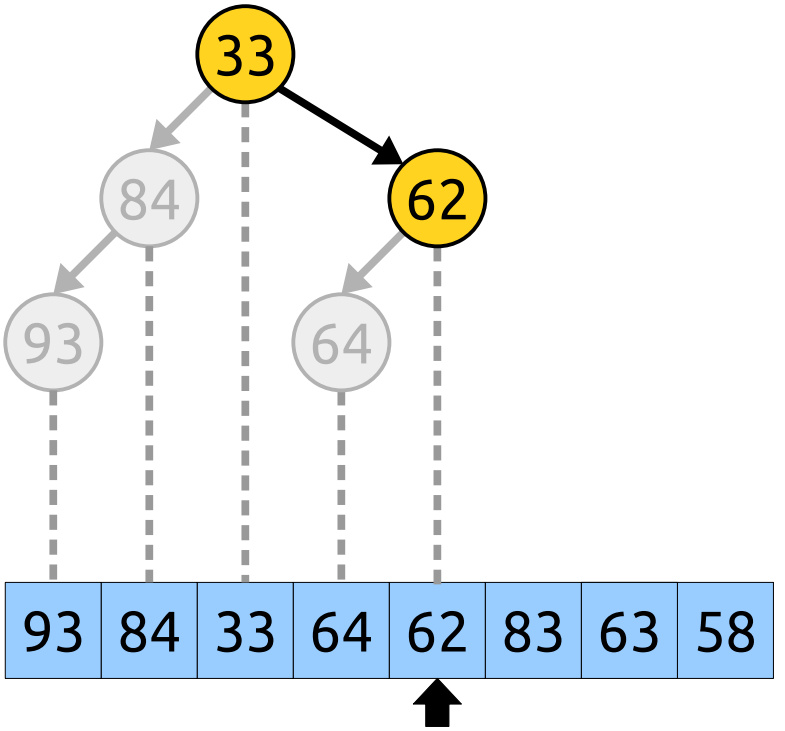
A Better Approach

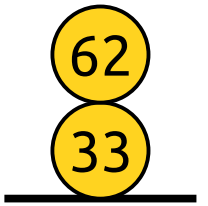
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



A Better Approach

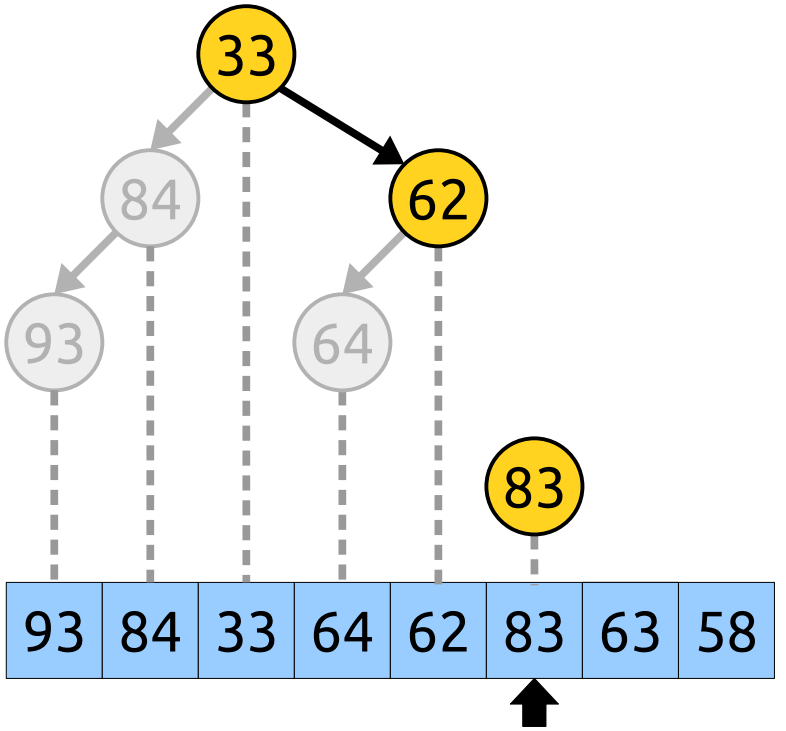
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

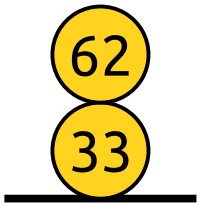




A Better Approach

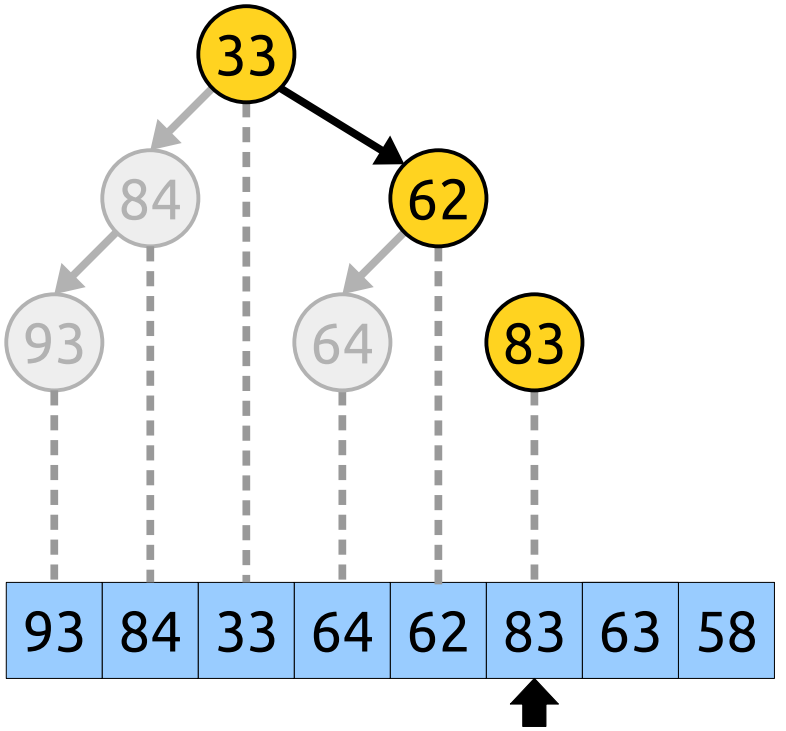
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

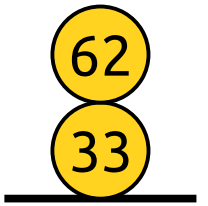




A Better Approach

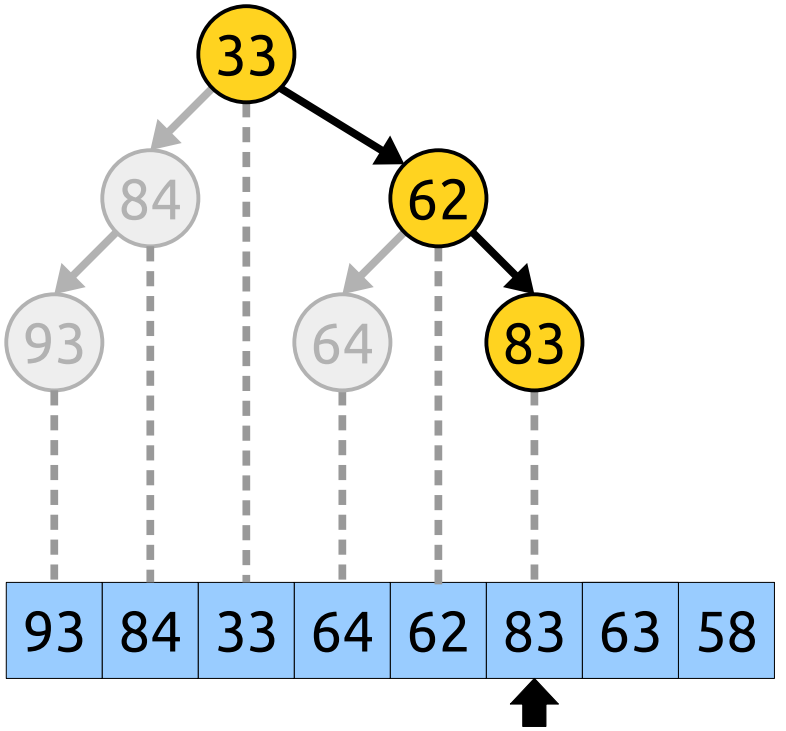
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

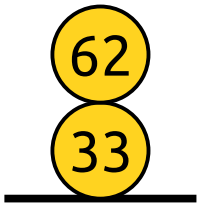




A Better Approach

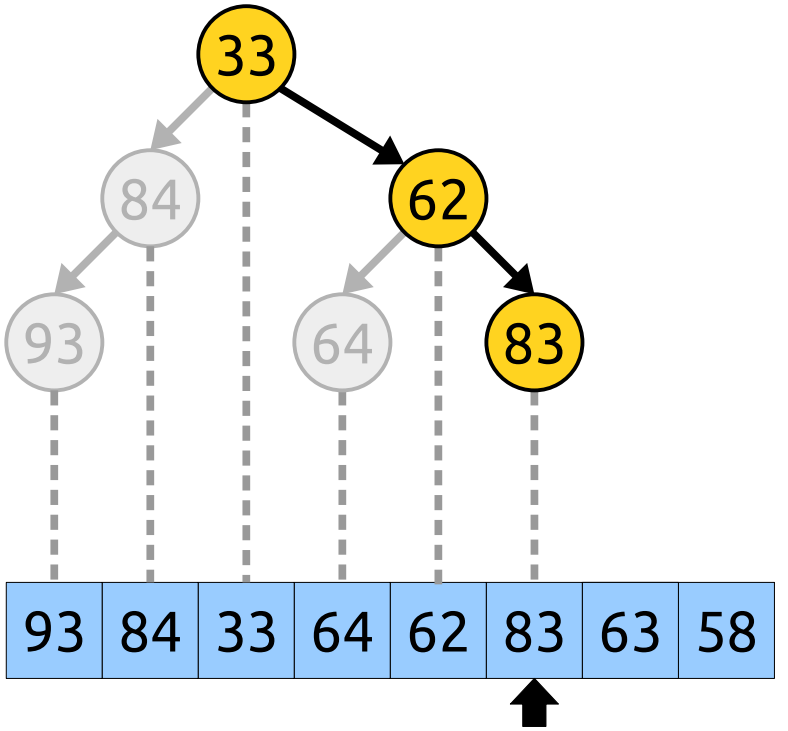
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.





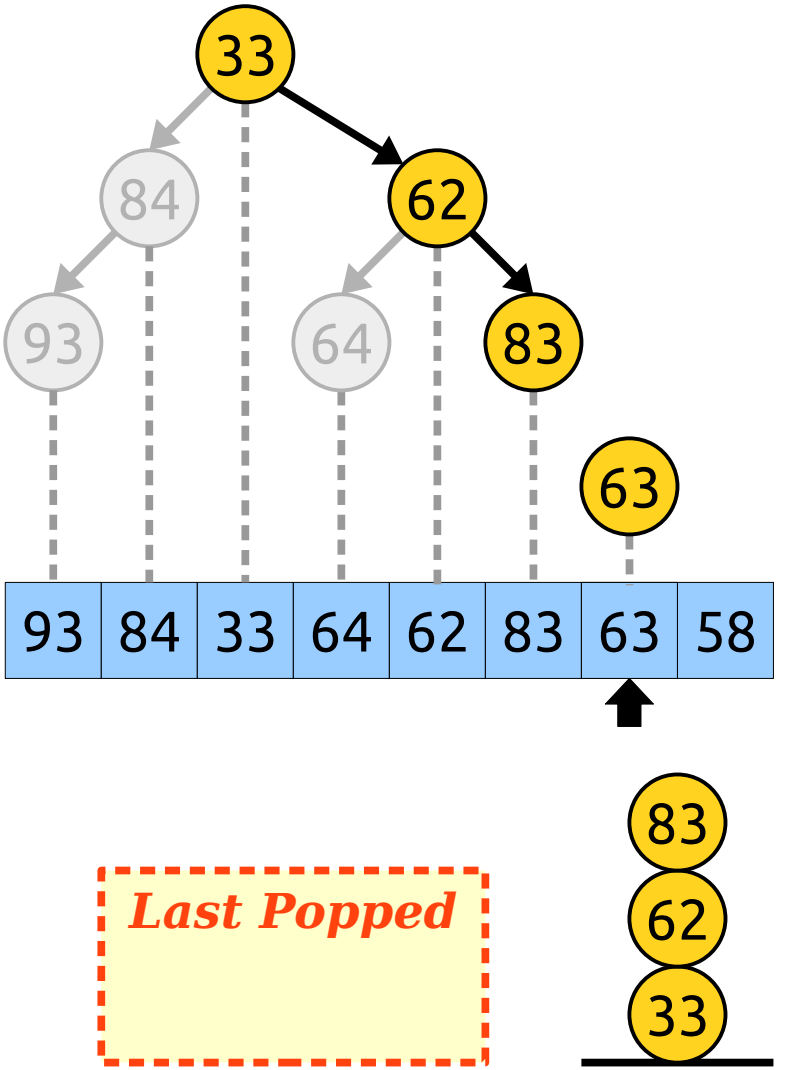
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



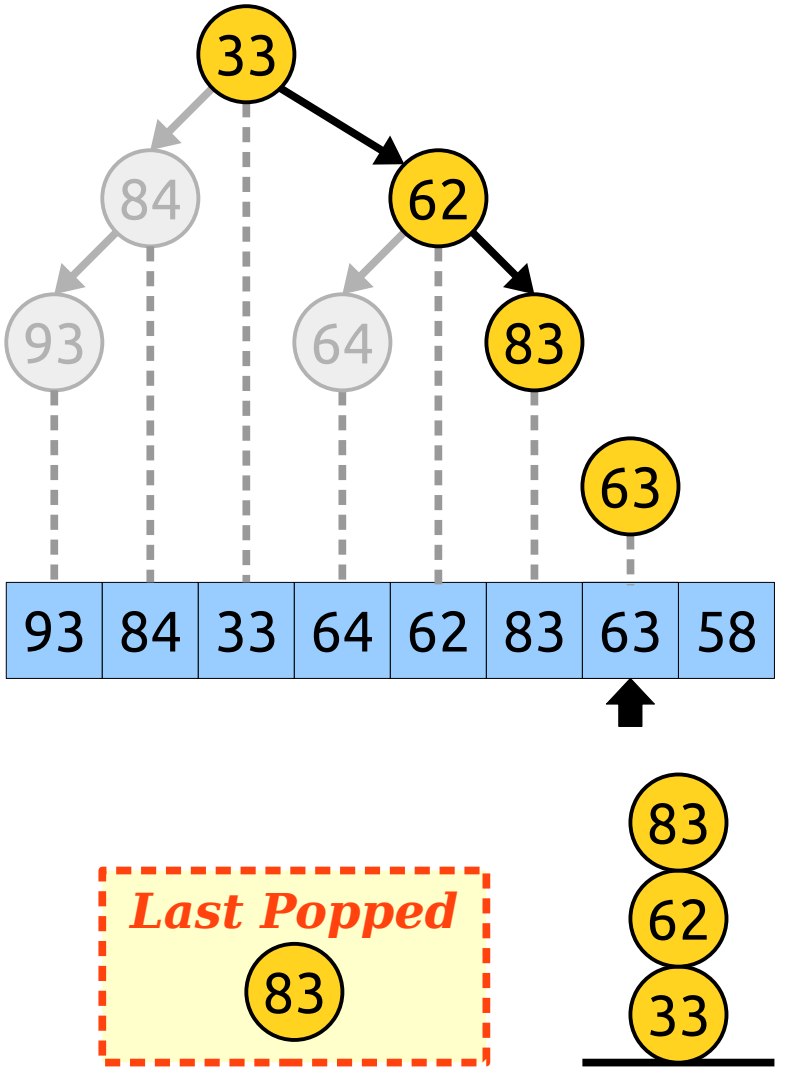
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



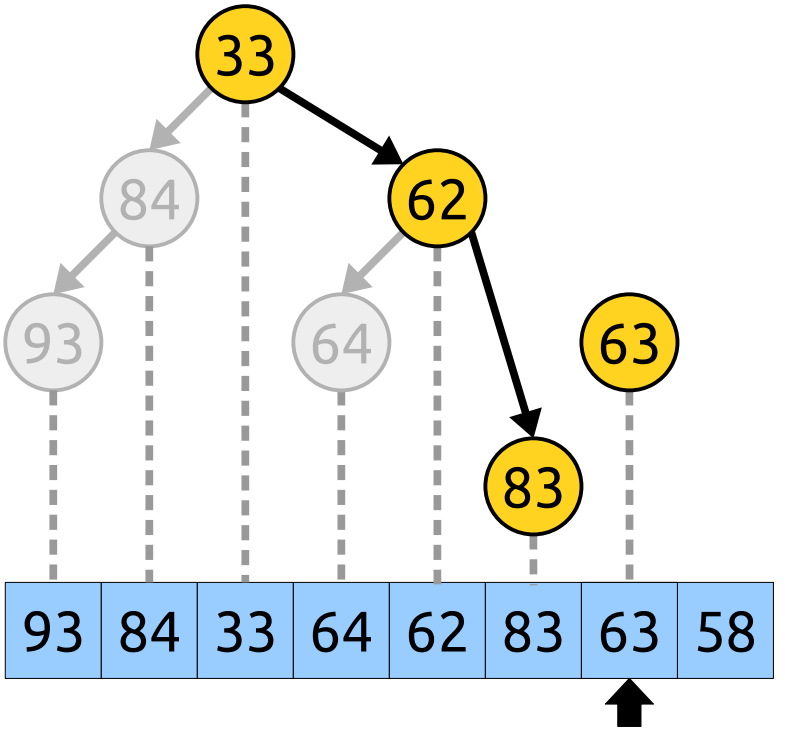
A Better Approach

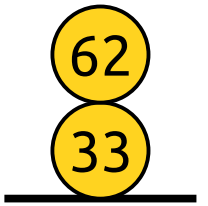
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



A Better Approach

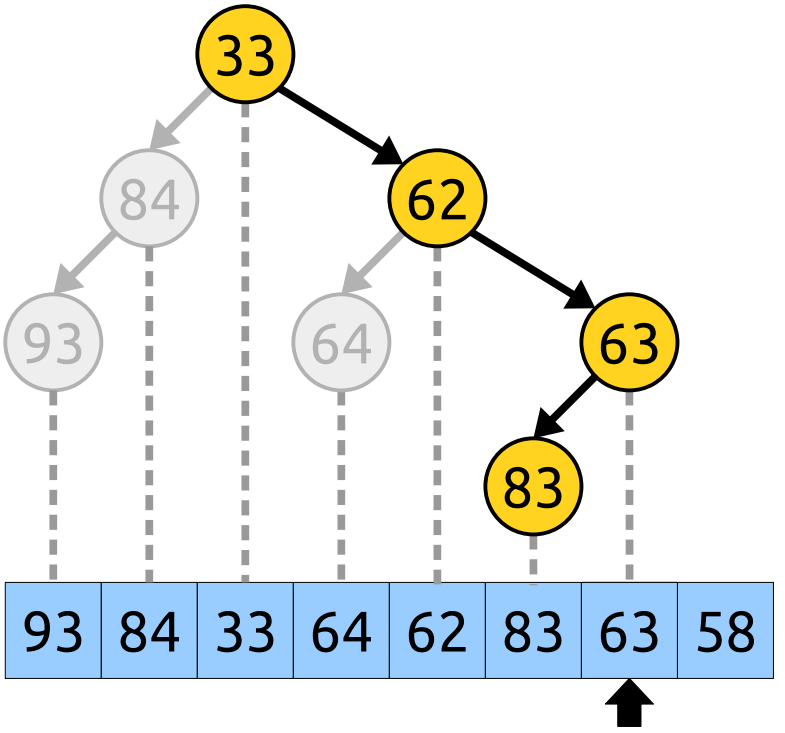
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

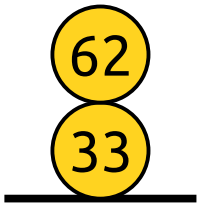




A Better Approach

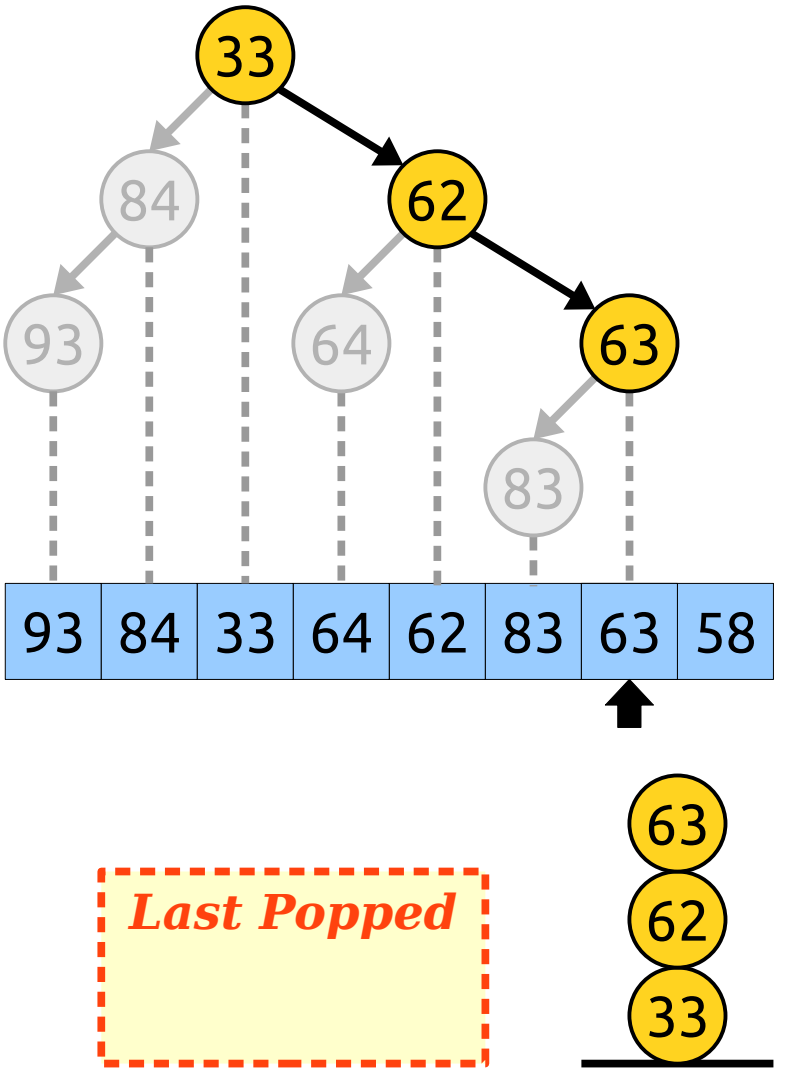
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.





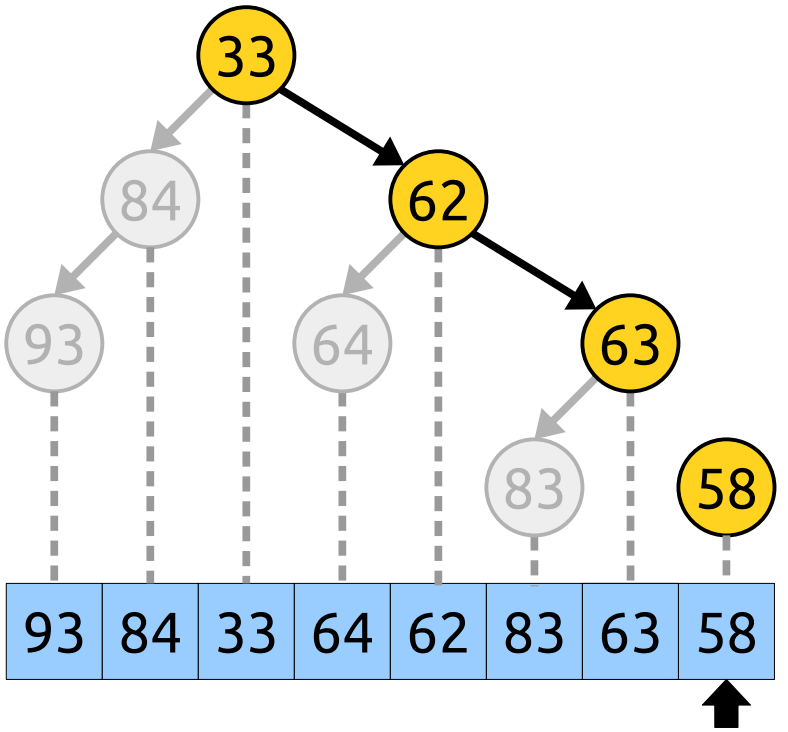
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



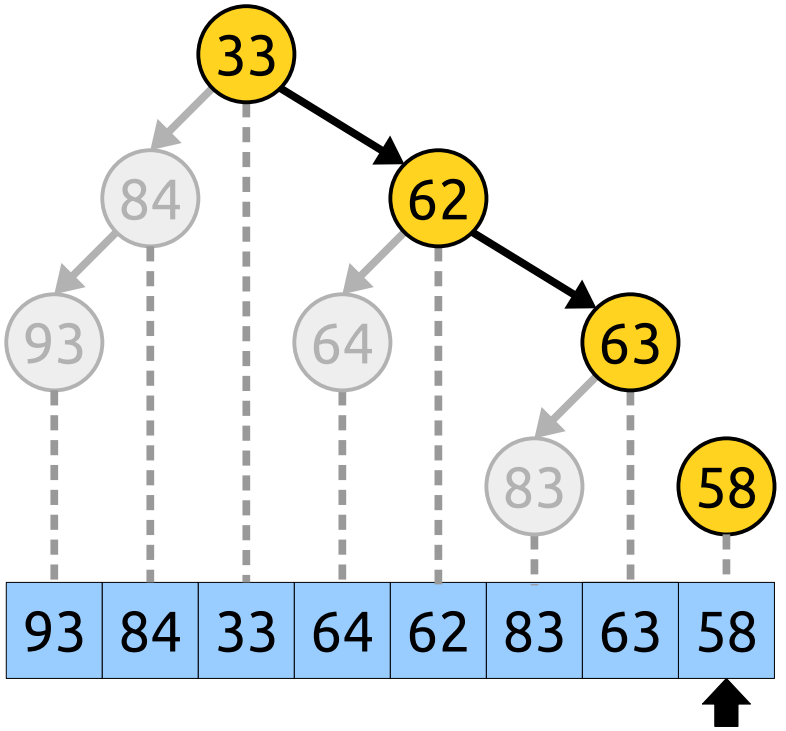
A Better Approach

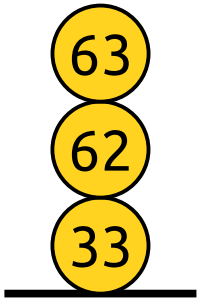
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



A Better Approach

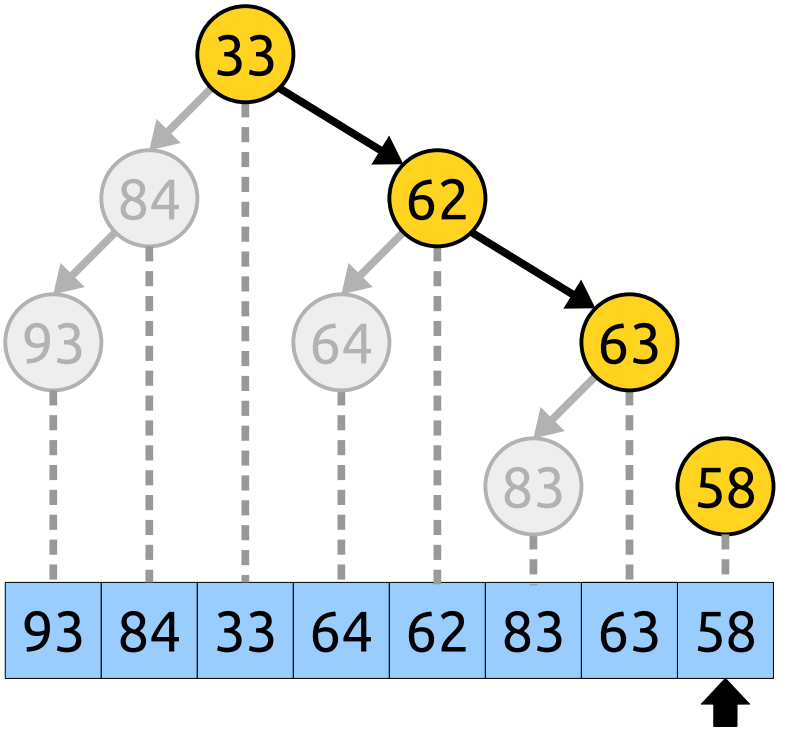
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

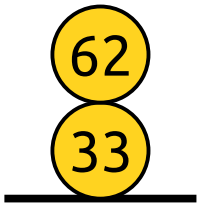




A Better Approach

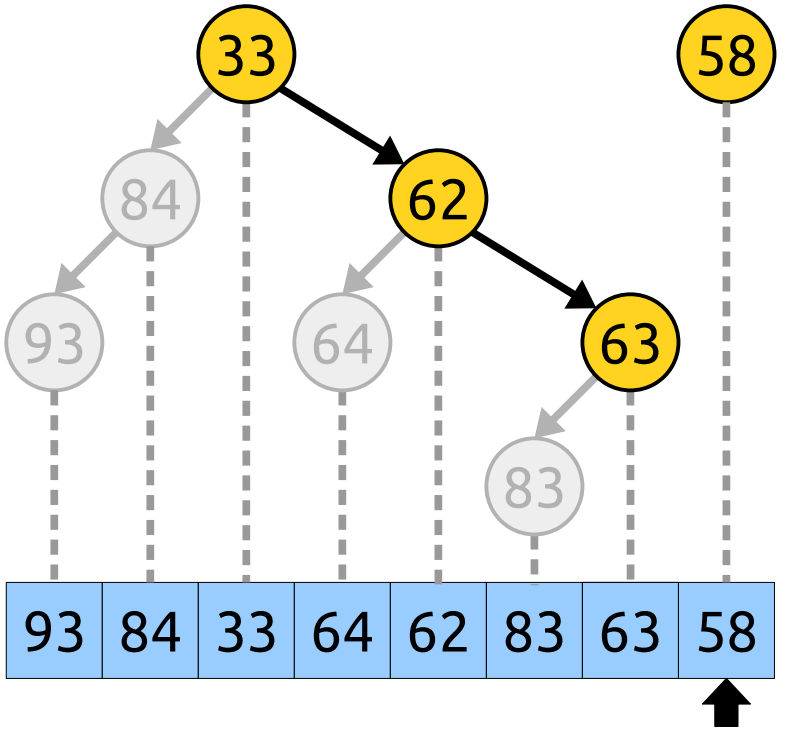
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.





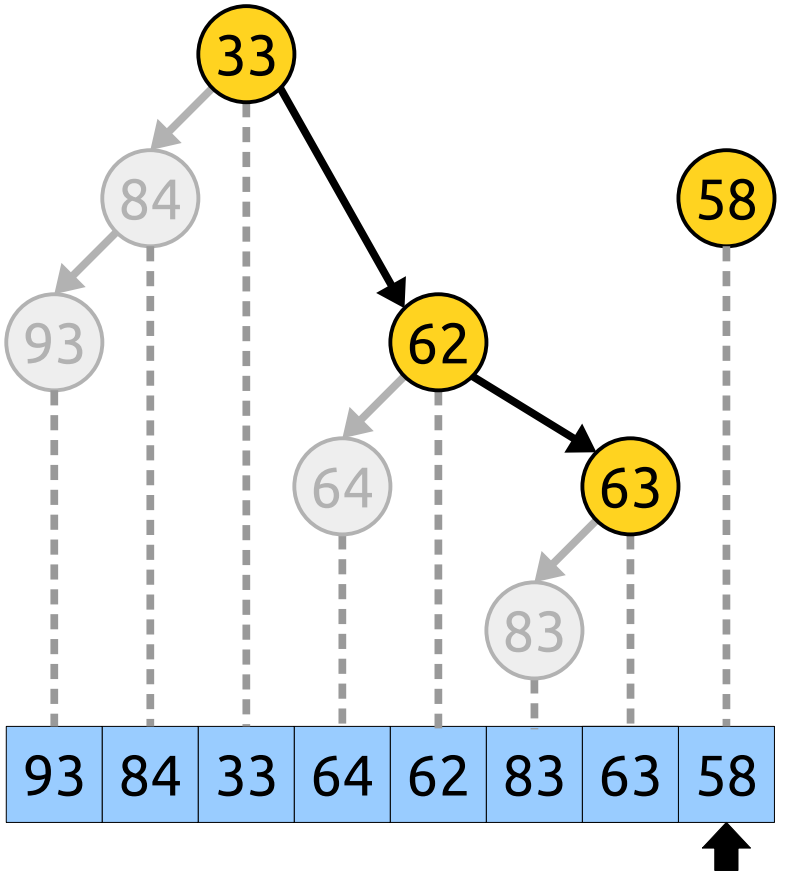
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



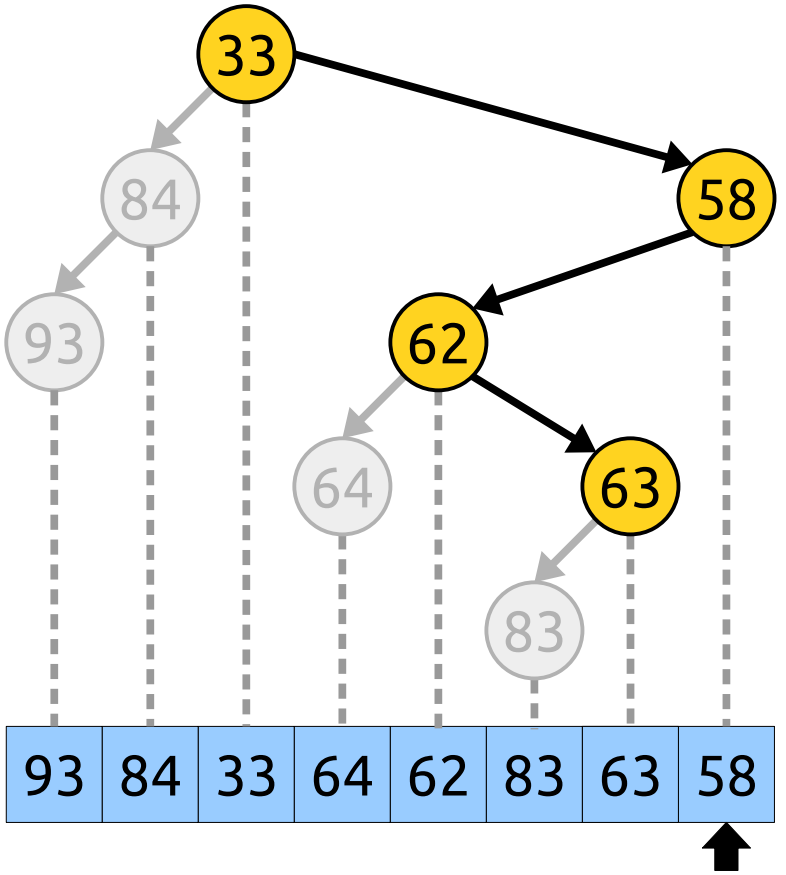
A Better Approach

- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



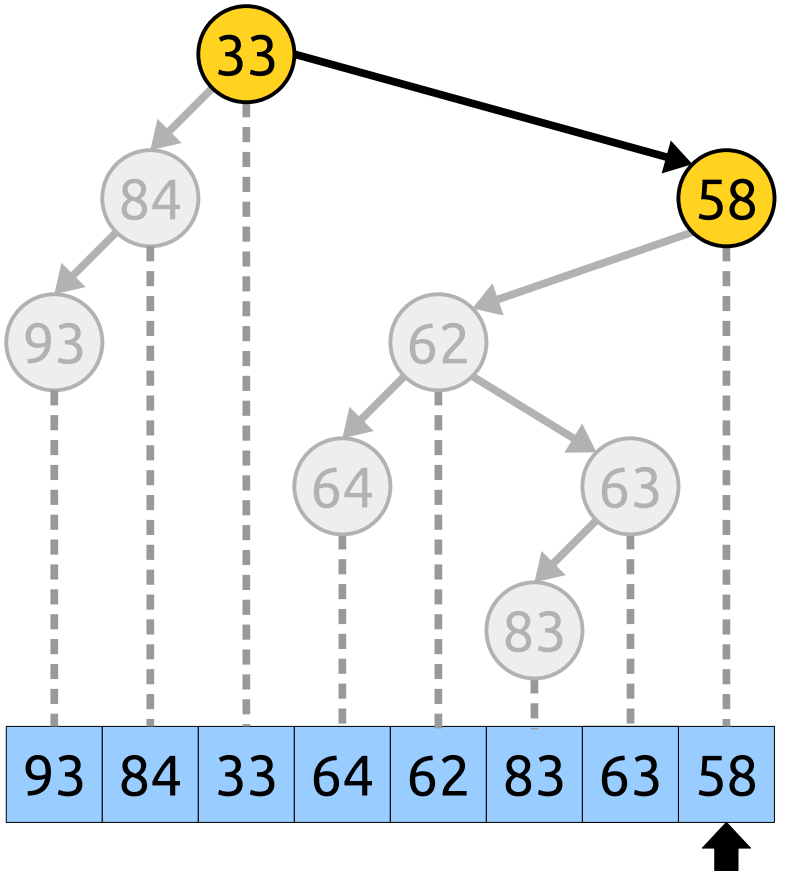
A Better Approach

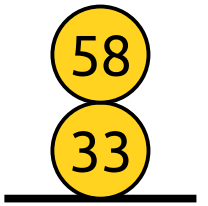
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.



A Better Approach

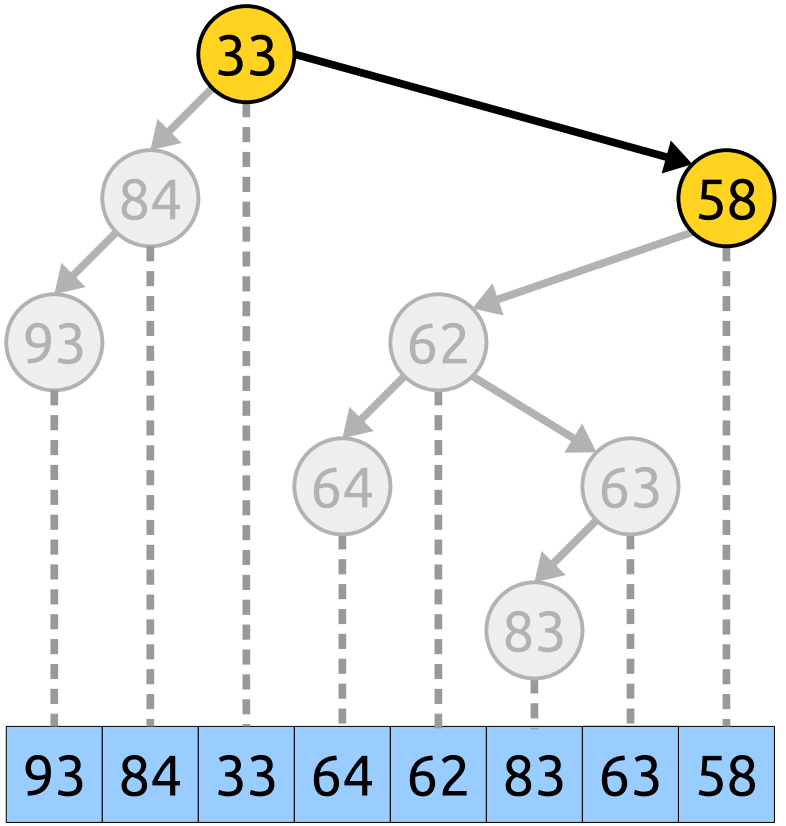
- We can implement this algorithm efficiently by maintaining a stack of the nodes in the right spine.
- Pop the stack until the new value is no bigger than the stack top (or the stack is empty). Remember the last node popped this way.
- Rewire the tree by
  - making the stack top point to the new node, and
  - making the most-recently-popped node the new node's left child.

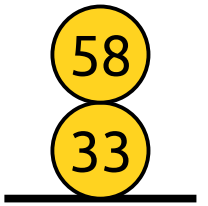




A Better Approach

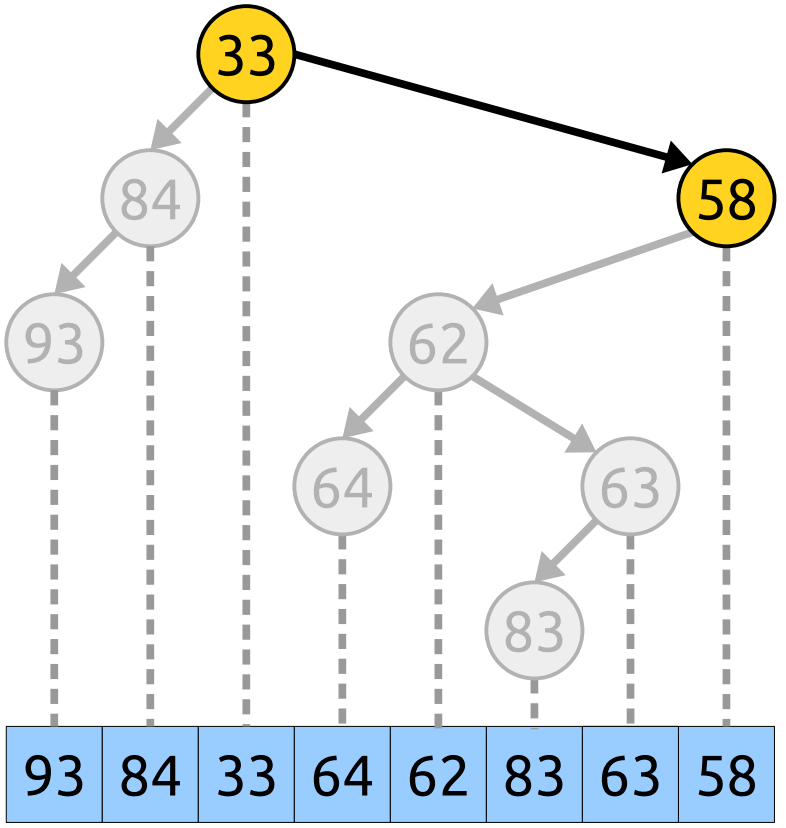
- How fast is this new approach on an array of k elements?
- Adding each element to the tree might take time $O(k)$, since we may have to pop $O(k)$ elements off the stack.
- Since there are k elements, that gives a time bound of $O(k^2)$.
- ***Question:*** Is this bound tight?

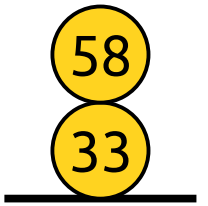




A Better Approach

- ***Claim:*** This algorithm takes time **O(*k*)** on an array of size k.
- ***Idea:*** Each element is pushed onto the stack at most once, when it's created. Each element can therefore be popped at most once.
- Therefore, there are at $O(k)$ pushes and $O(k)$ pops, so the runtime is $O(k)$.





How can we tell when two blocks can share RMQ structures?

When those blocks have isomorphic Cartesian trees!
And we can check this in time O(b)!
But there are lots of pairs of blocks to check!

How many block types are there, as a function of b?

¯_(ツ)_/¯

How can we tell when two blocks can share RMQ structures?

When those blocks have isomorphic Cartesian trees!
And we can check this in time O(b)!
But there are lots of pairs of blocks to check!

How many block types are there, as a function of b?

¯_(ツ)_/¯

Theorem: The number of distinct Cartesian tree shapes for arrays of length b is at most 4^b.

In case you're curious, the actual number is

$$\frac{1}{b+1}\binom{2b}{b},$$

which is roughly equal to

$$\frac{4^b}{b^{3/2}\sqrt{\pi}}.$$

Look up the ***Catalan numbers*** *for more information!*

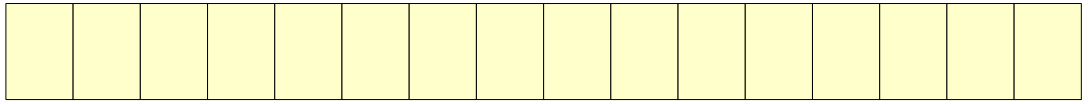
Proof Approach

- Our stack-based algorithm for generating Cartesian trees produces a Cartesian tree for every possible input array.
- Therefore, if we can count the number of possible executions of that algorithm, we can count the number of Cartesian trees.
- Using a simple counting scheme, we can show that there are at most 4^b possible executions.

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

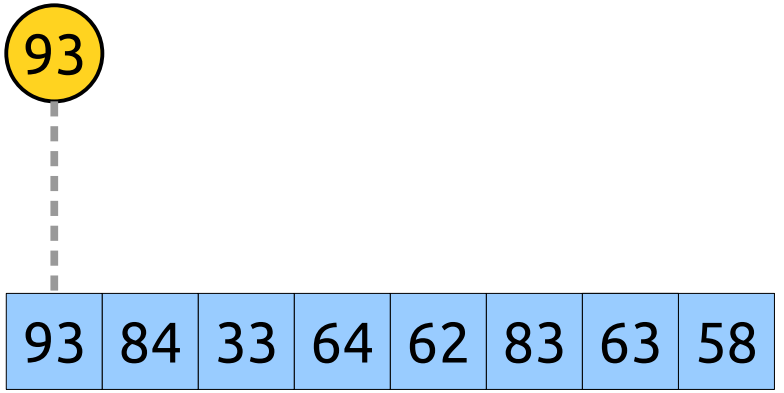
93	84	33	64	62	83	63	58

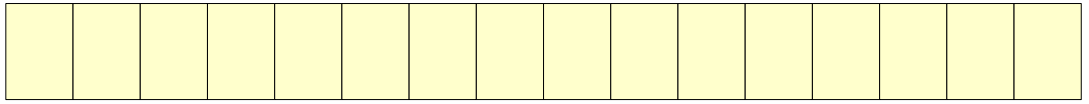


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

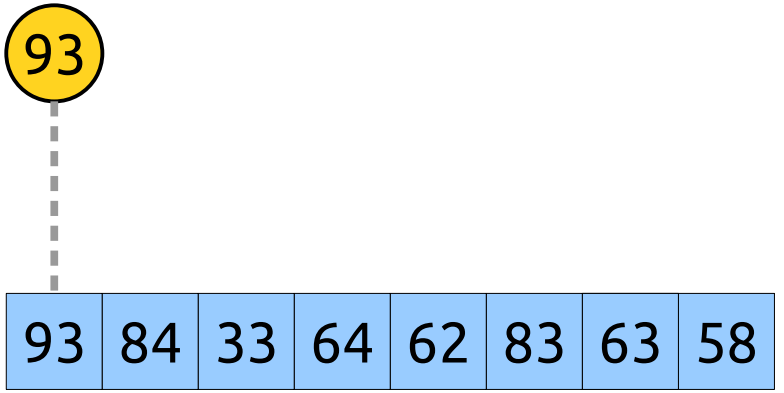


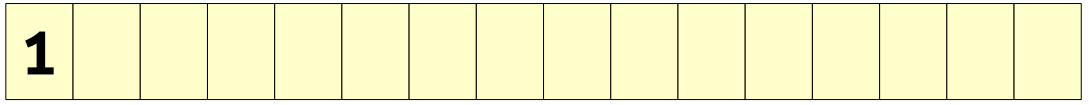


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

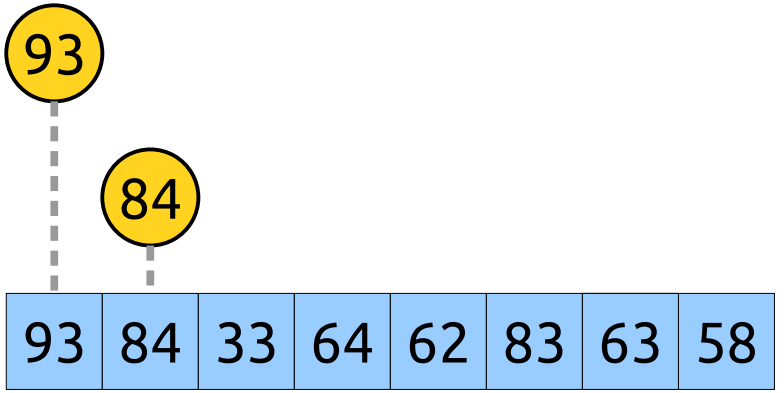


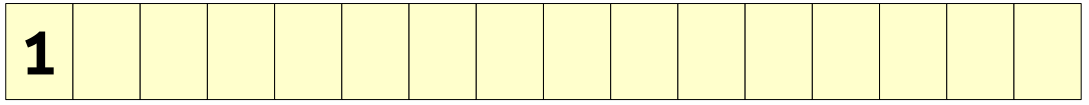


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

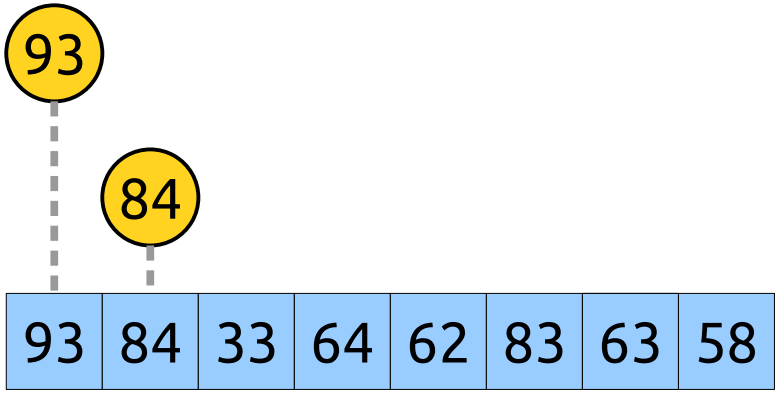


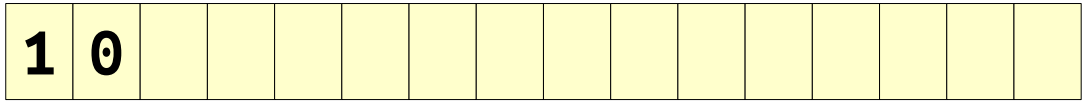


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

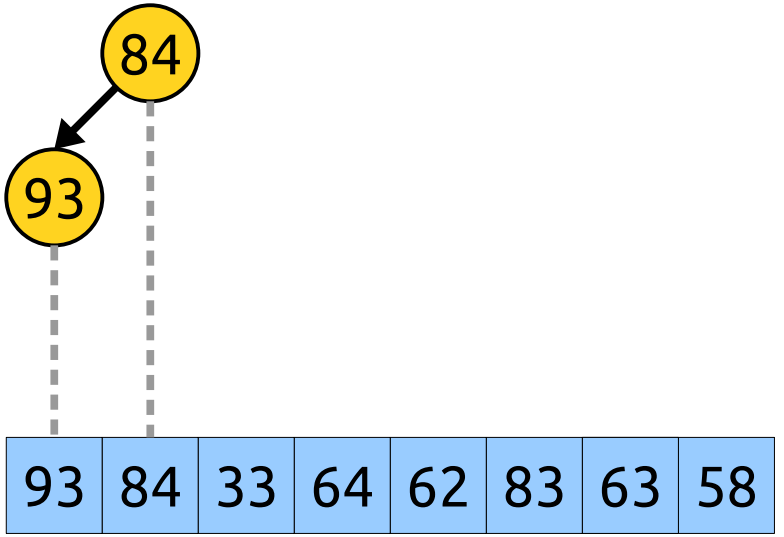


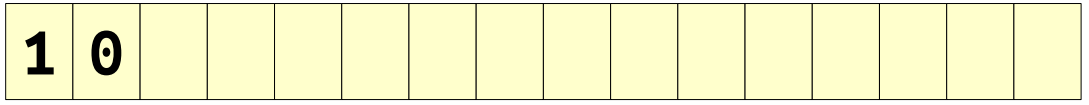


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

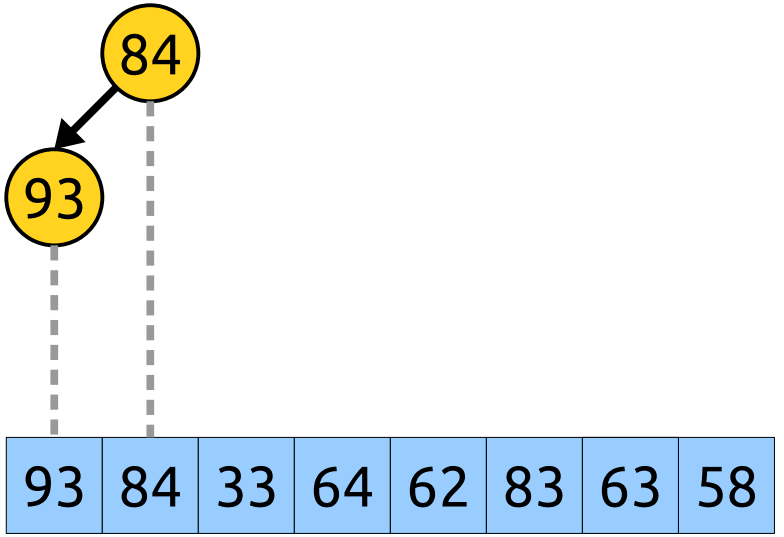


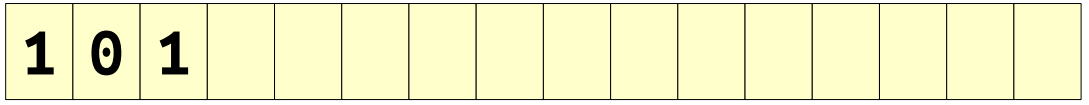


0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

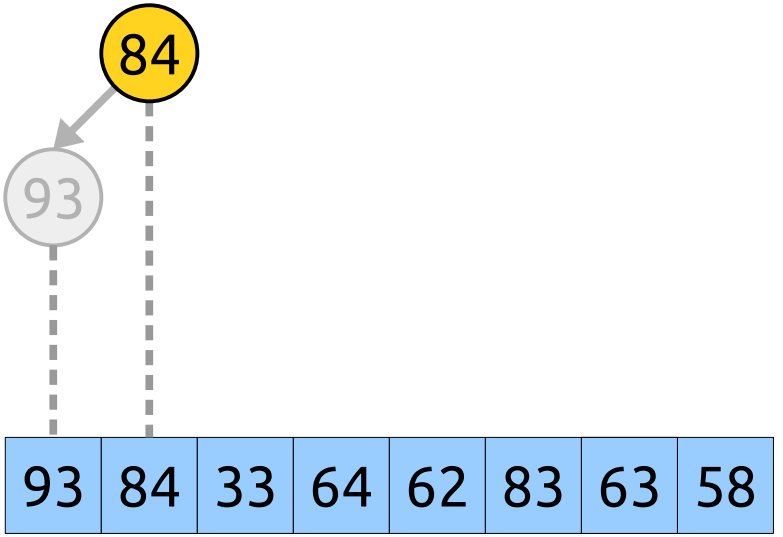




0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

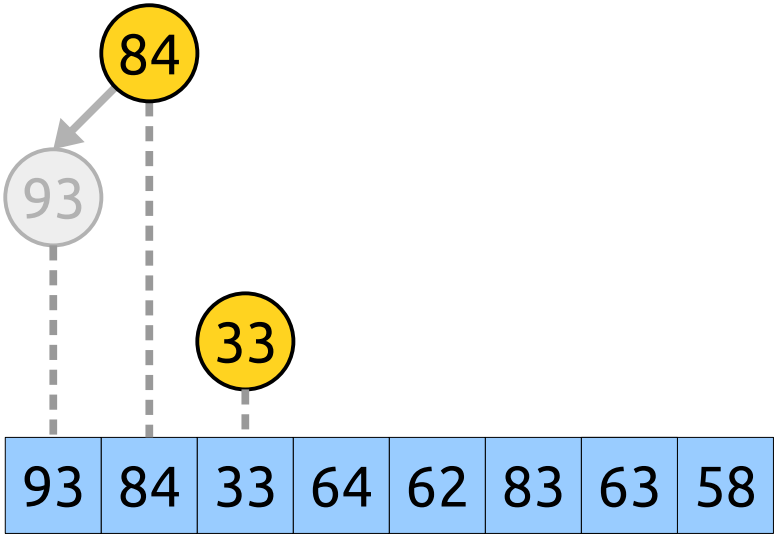


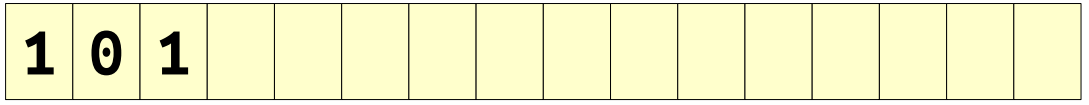
1	0	1												

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

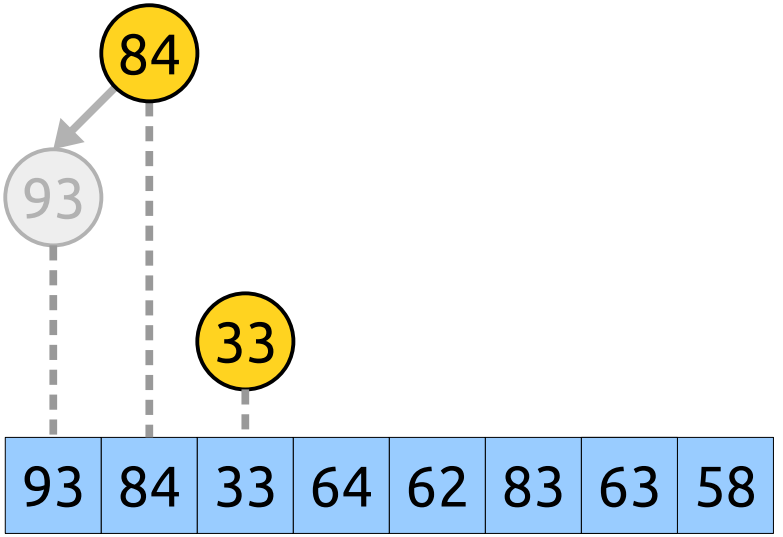




0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

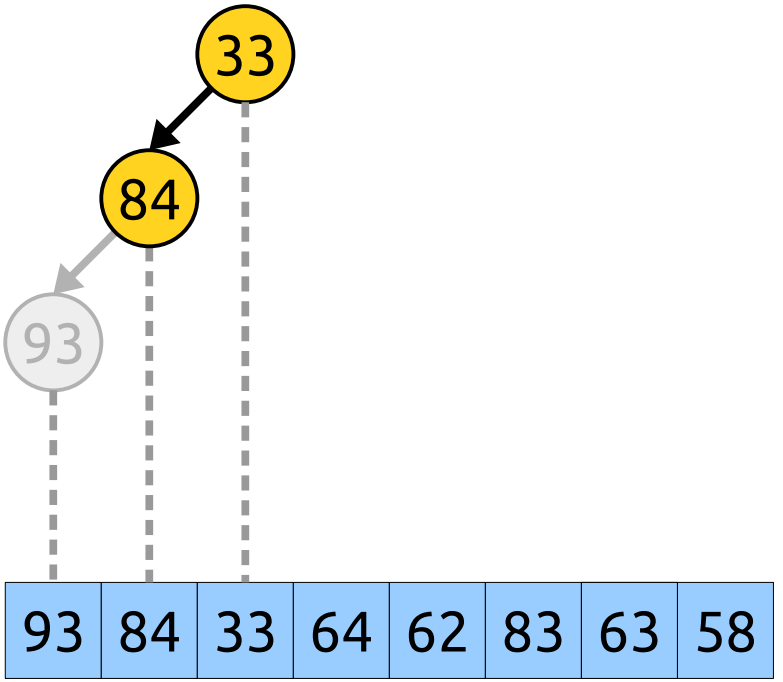


1	0	1	0												

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

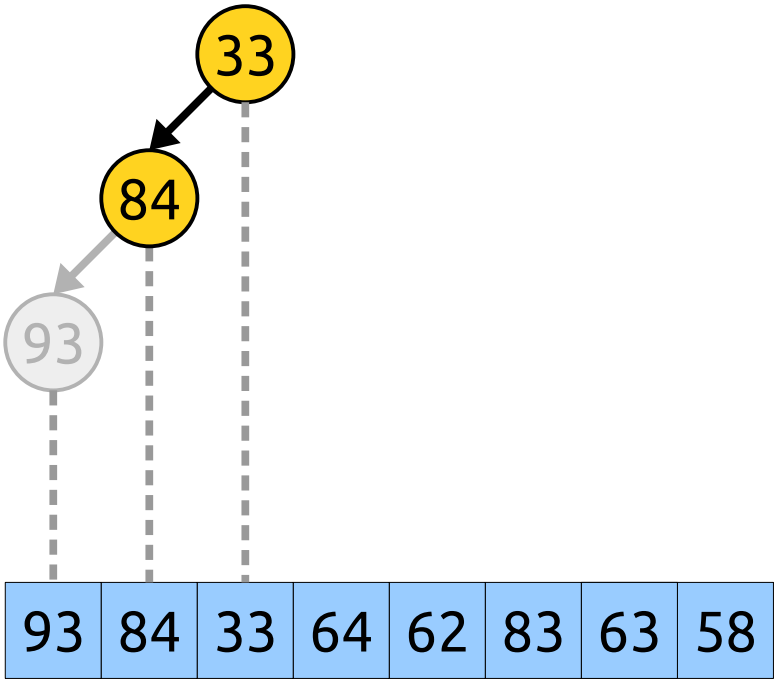


1	0	1	0												

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

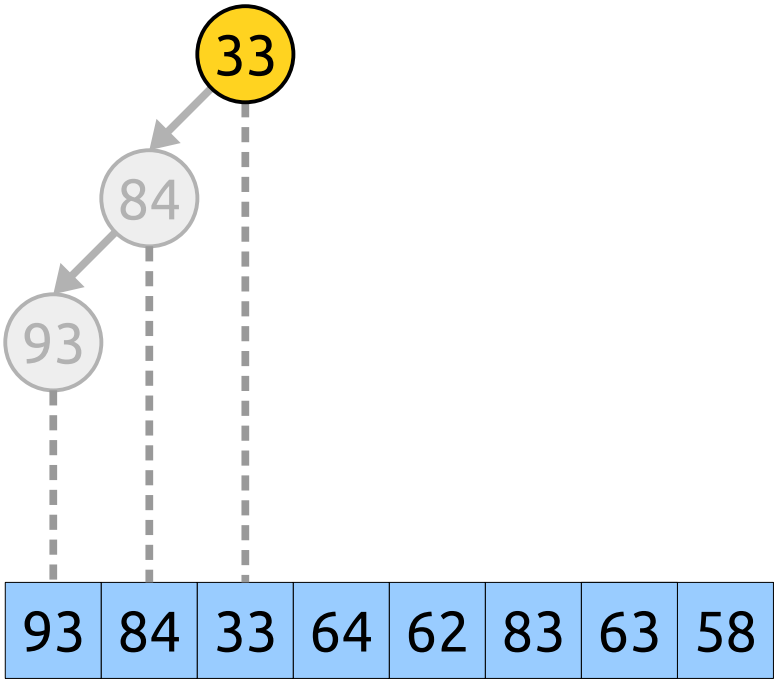


1	0	1	0	1											

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

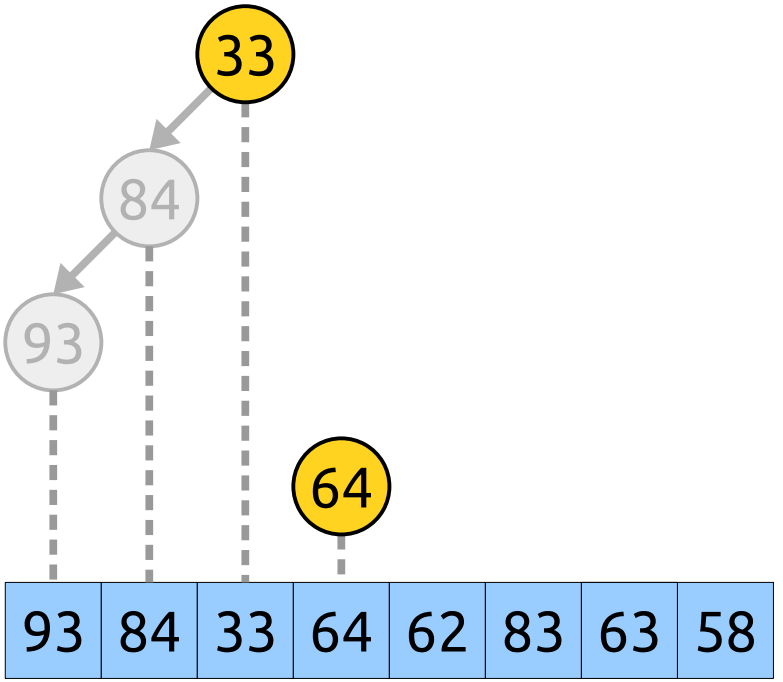


1	0	1	0	1											

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

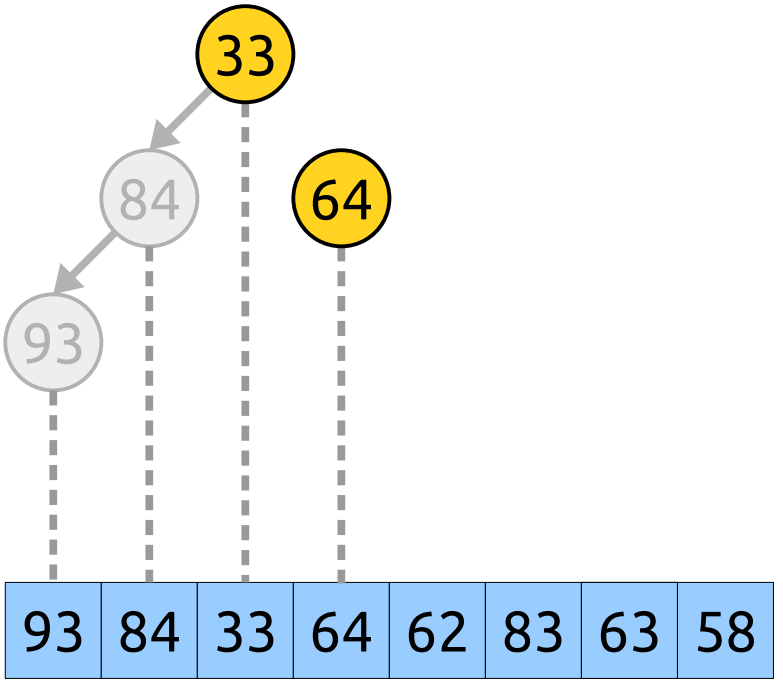


1	0	1	0	1											

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

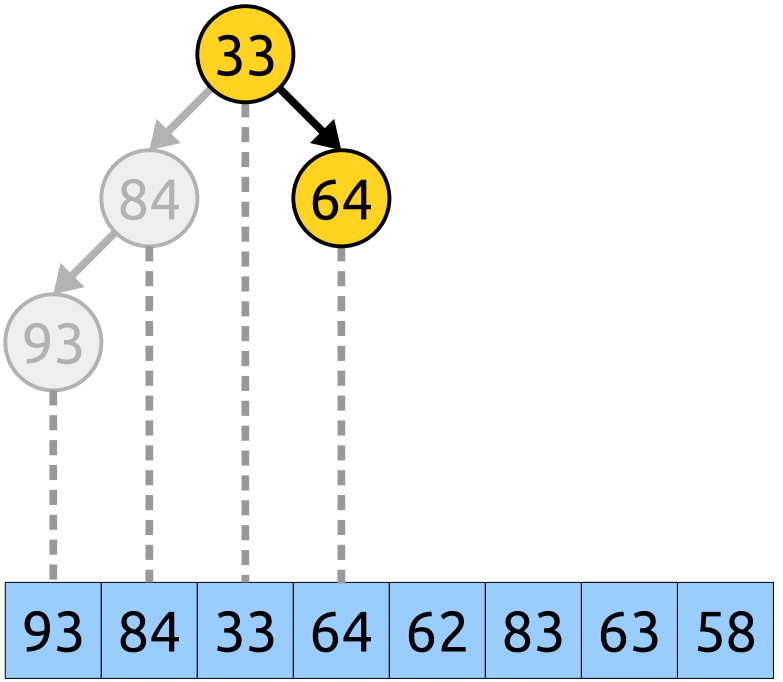


1	0	1	0	1											

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

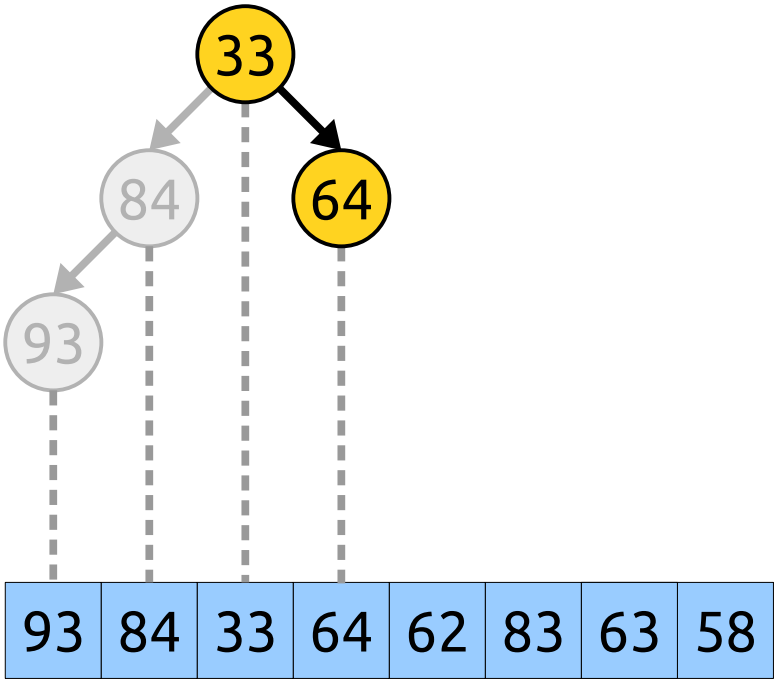


1	0	1	0	1											

0 means pop; 1 means push

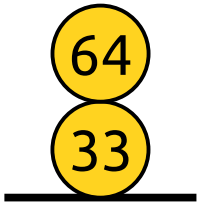
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



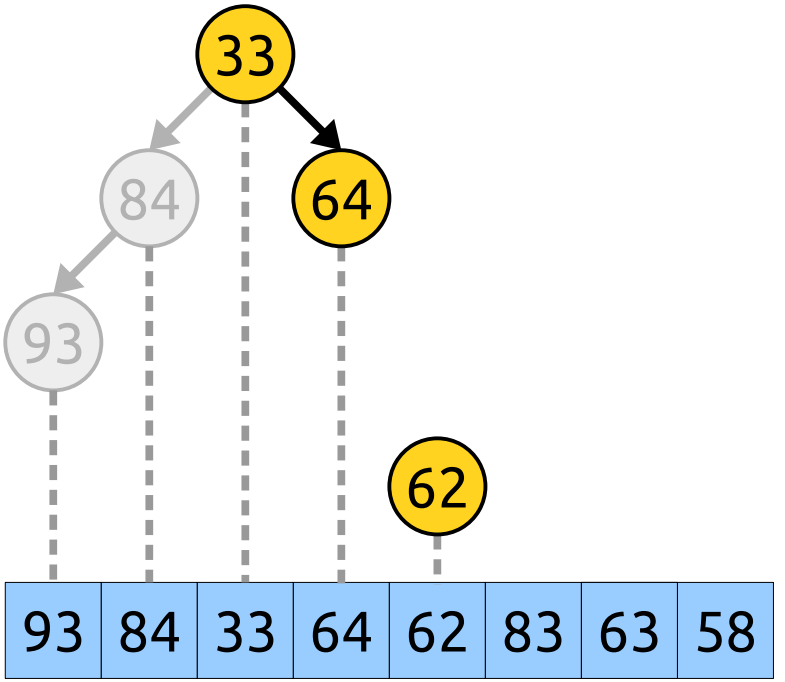
1	0	1	0	1	1										

0 means pop; 1 means push



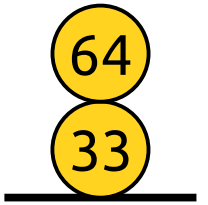
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



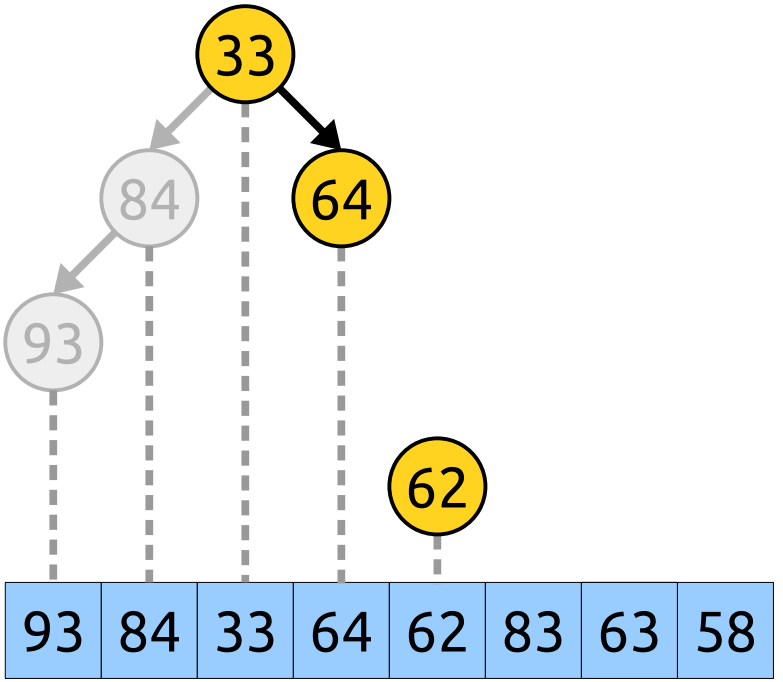
1	0	1	0	1	1										

0 means pop; 1 means push



Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

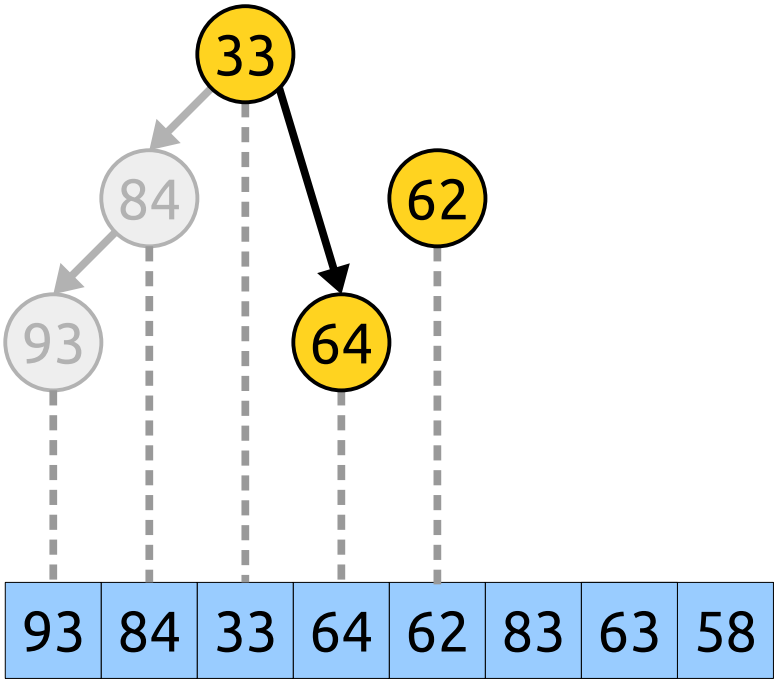


1	0	1	0	1	1	0									

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

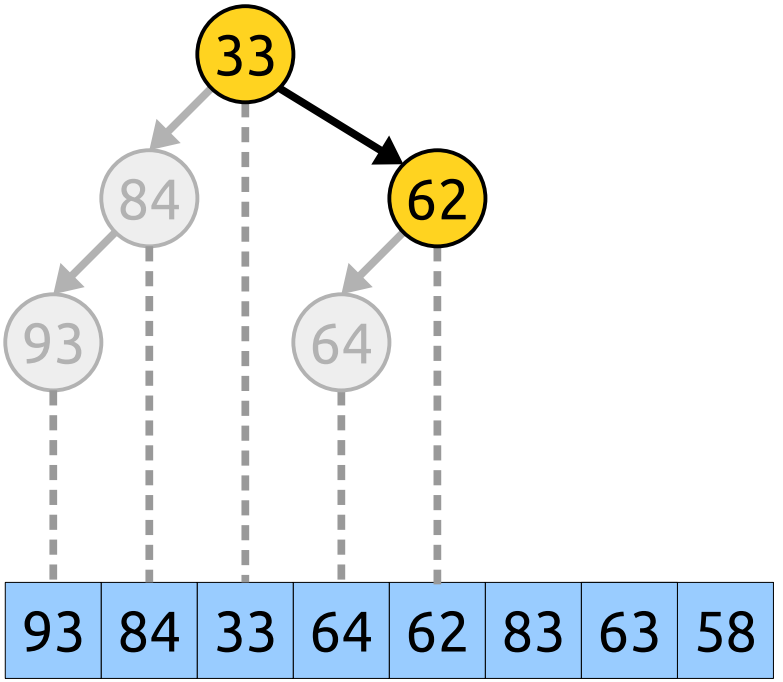


1	0	1	0	1	1	0									

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

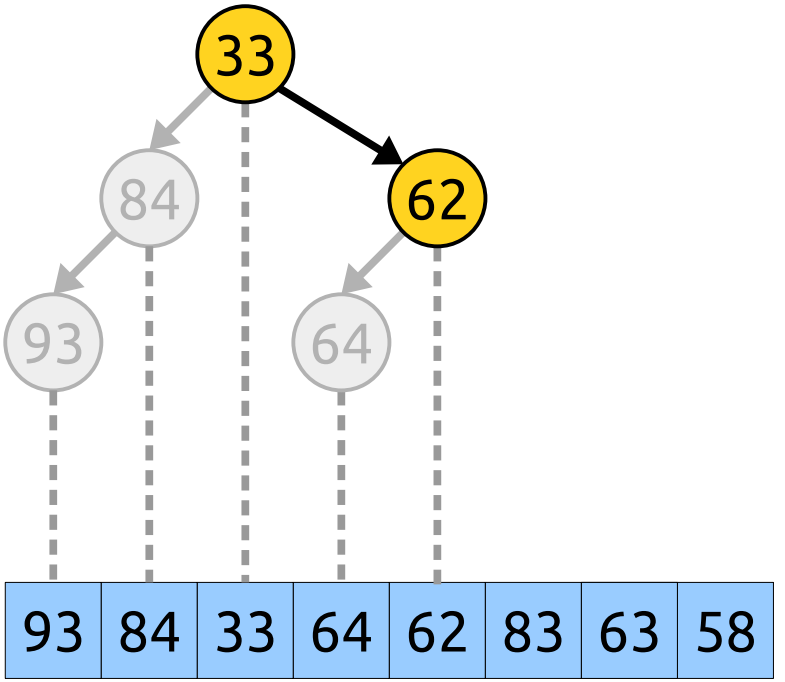


1	0	1	0	1	1	0									

0 means pop; 1 means push

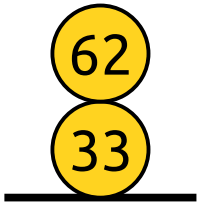
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



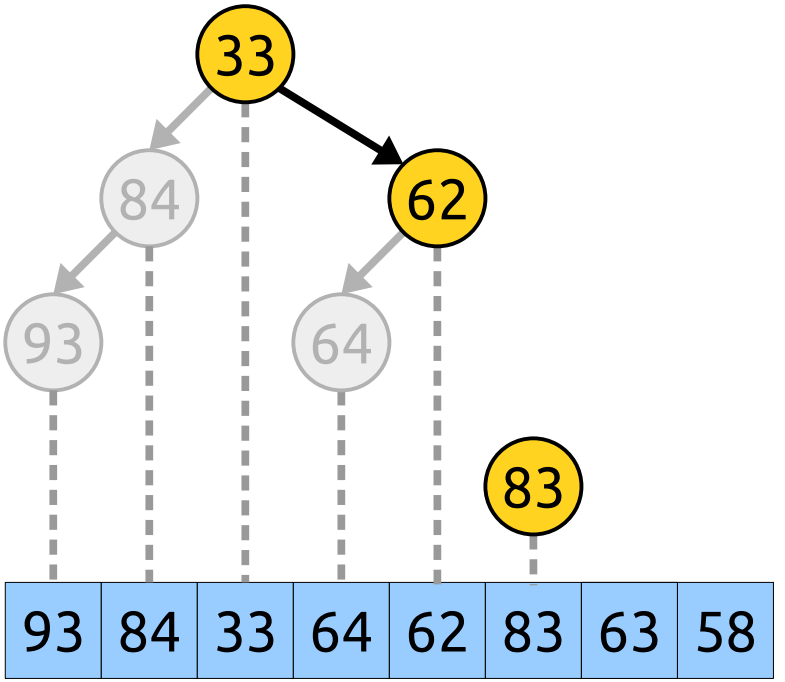
1	0	1	0	1	1	0	1								

0 means pop; 1 means push



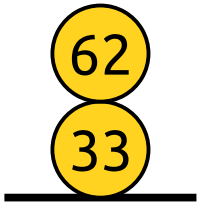
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



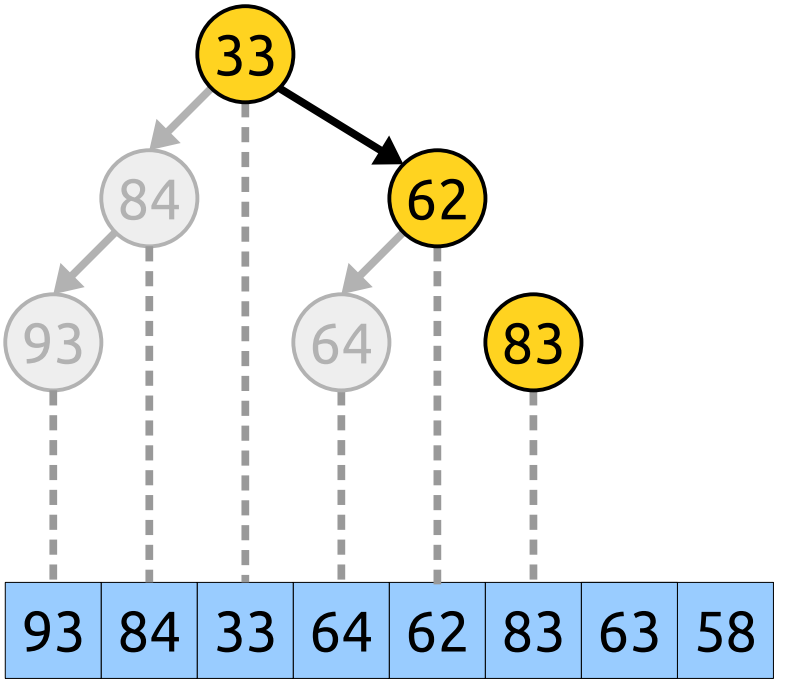
1	0	1	0	1	1	0	1								

0 means pop; 1 means push



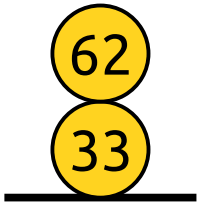
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



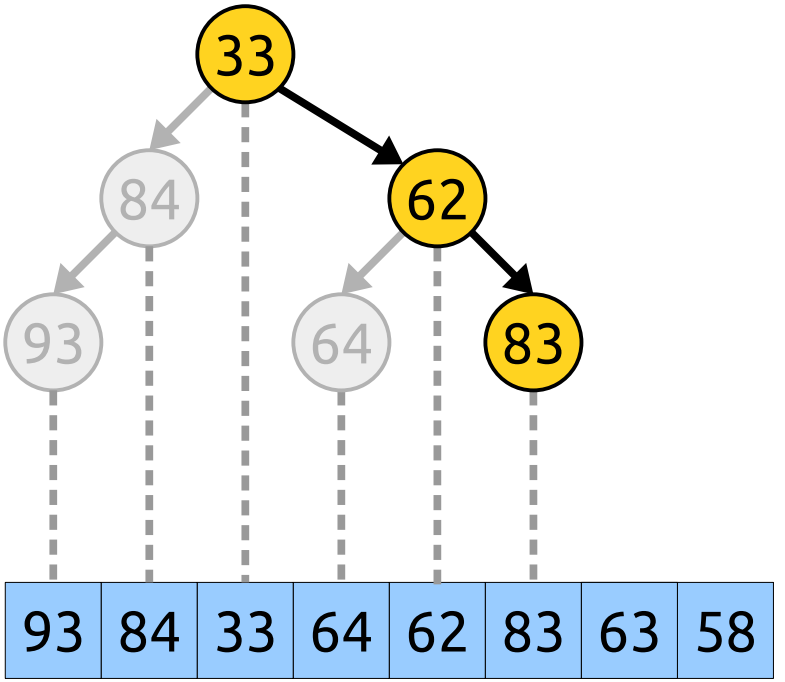
1	0	1	0	1	1	0	1								

0 means pop; 1 means push



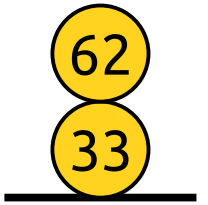
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



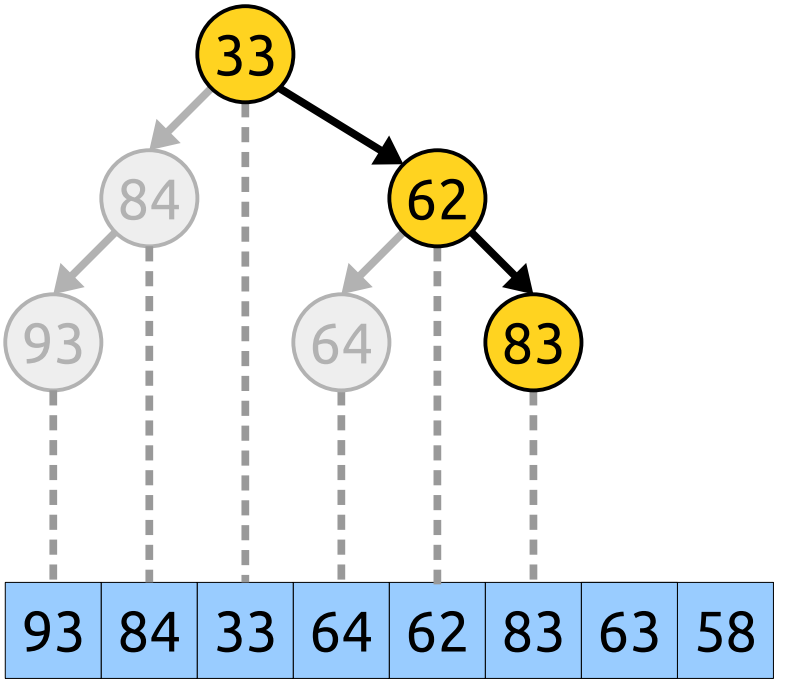
1	0	1	0	1	1	0	1								

0 means pop; 1 means push



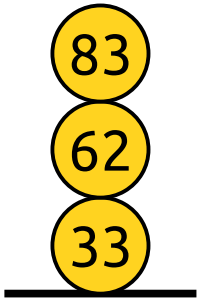
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



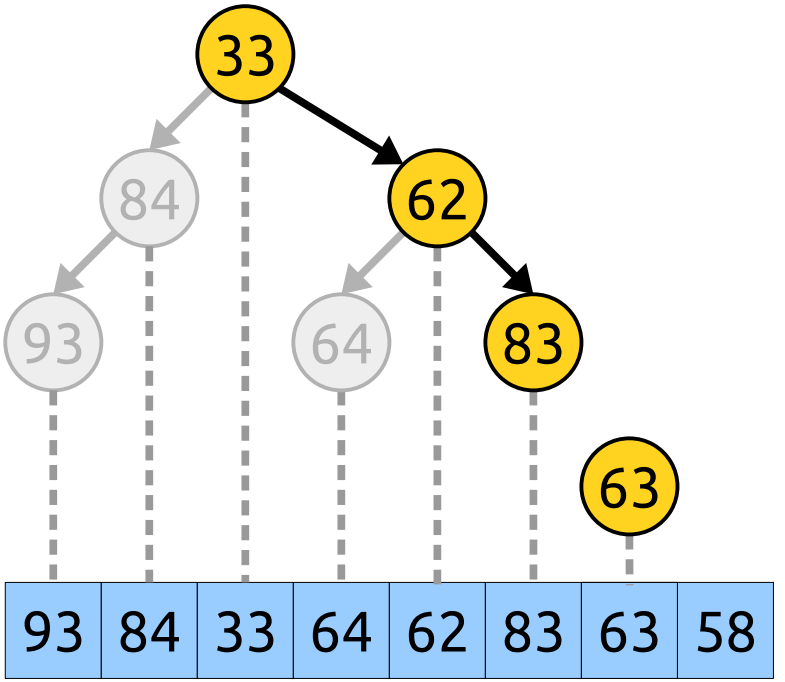
1	0	1	0	1	1	0	1	1							

0 means pop; 1 means push



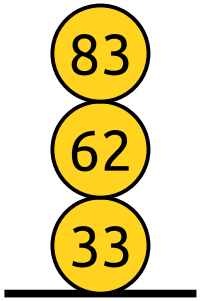
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



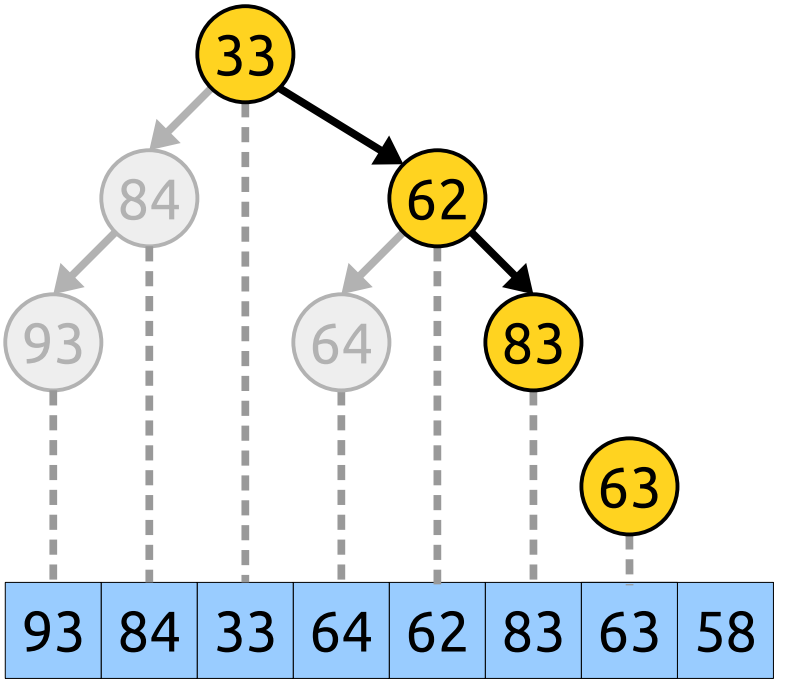
1	0	1	0	1	1	0	1	1							

0 means pop; 1 means push



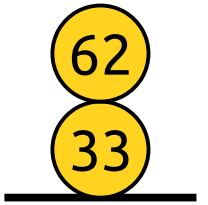
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



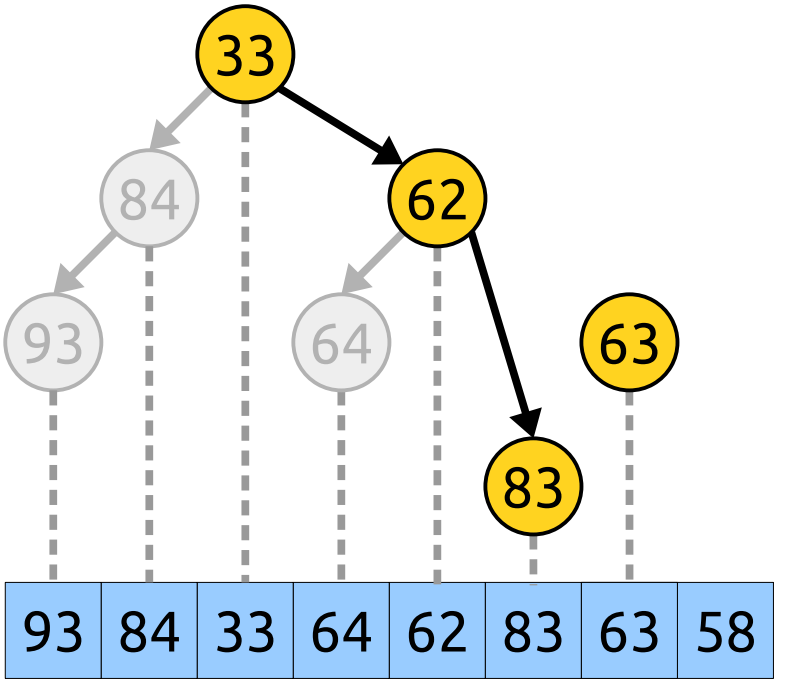
1	0	1	0	1	1	0	1	1	0						

0 means pop; 1 means push



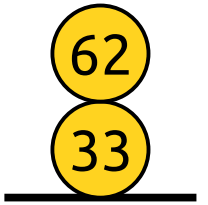
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



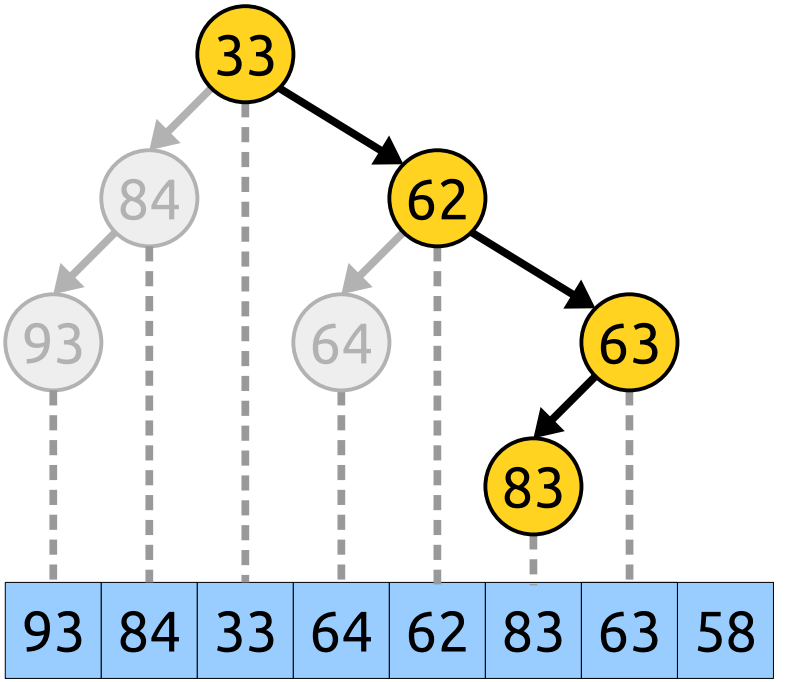
1	0	1	0	1	1	0	1	1	0						

0 means pop; 1 means push



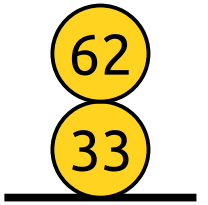
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



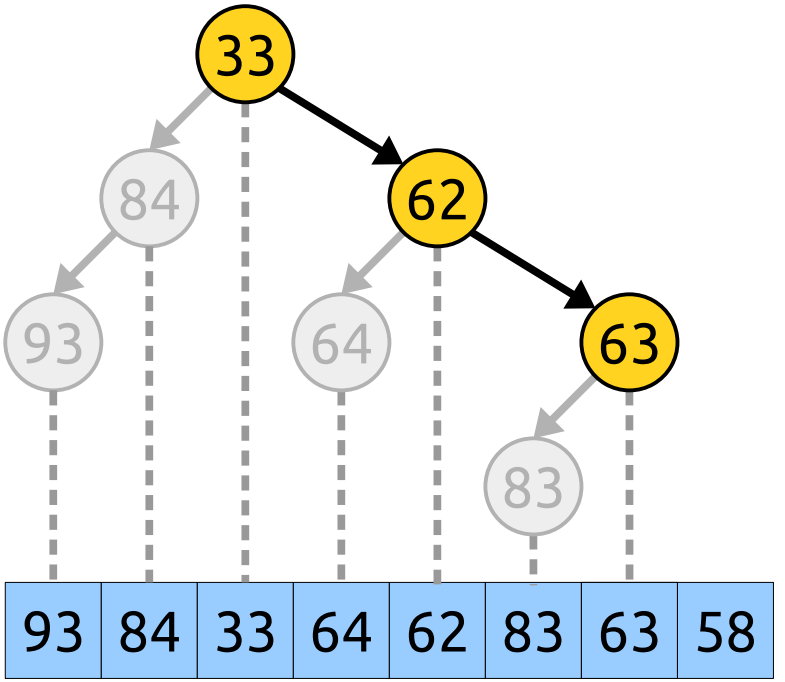
1	0	1	0	1	1	0	1	1	0						

0 means pop; 1 means push



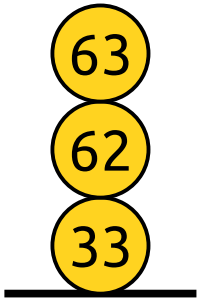
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



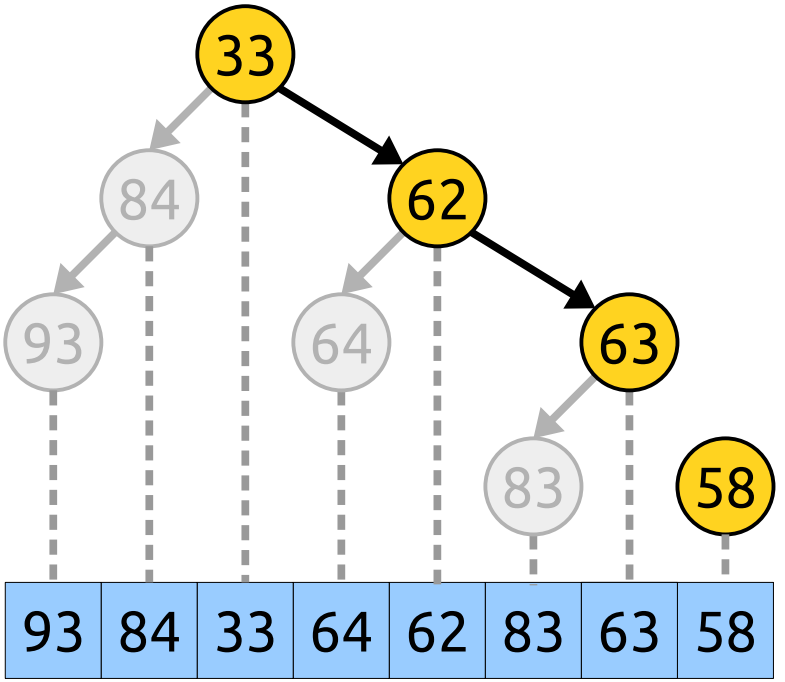
1	0	1	0	1	1	0	1	1	0	1					

0 means pop; 1 means push



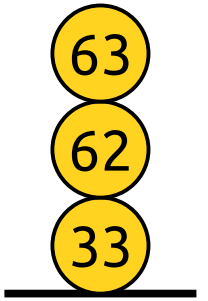
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



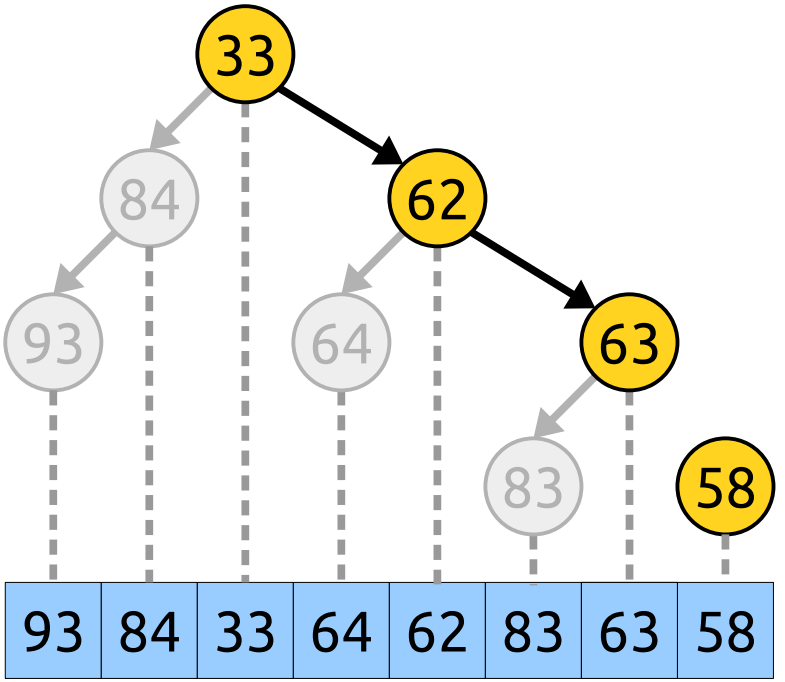
1	0	1	0	1	1	0	1	1	0	1					

0 means pop; 1 means push



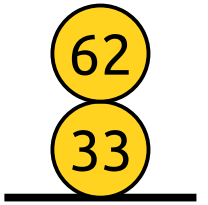
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



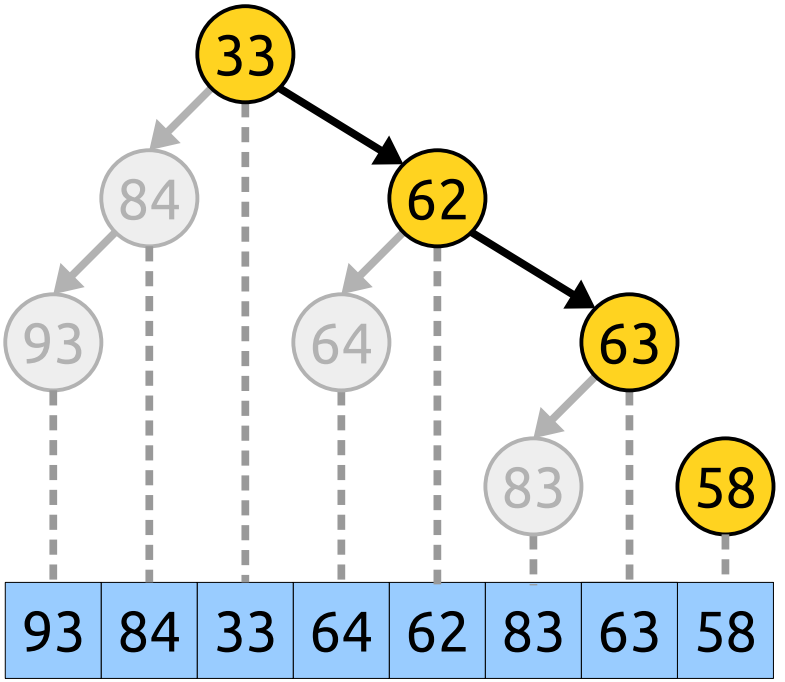
1	0	1	0	1	1	0	1	1	0	1	0				

0 means pop; 1 means push



Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

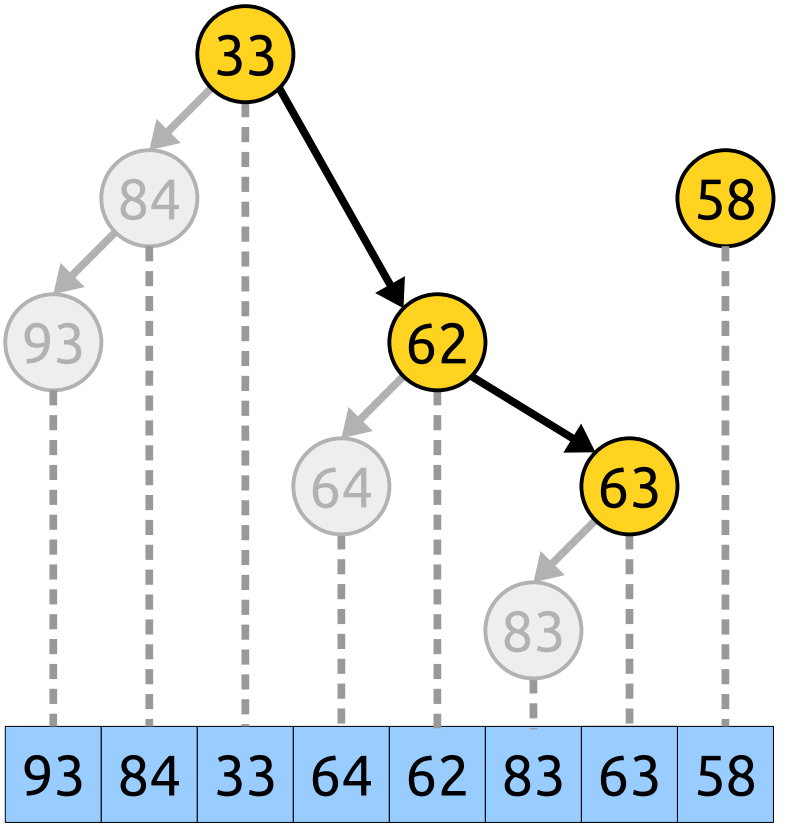


1	0	1	0	1	1	0	1	1	0	1	0	0			

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

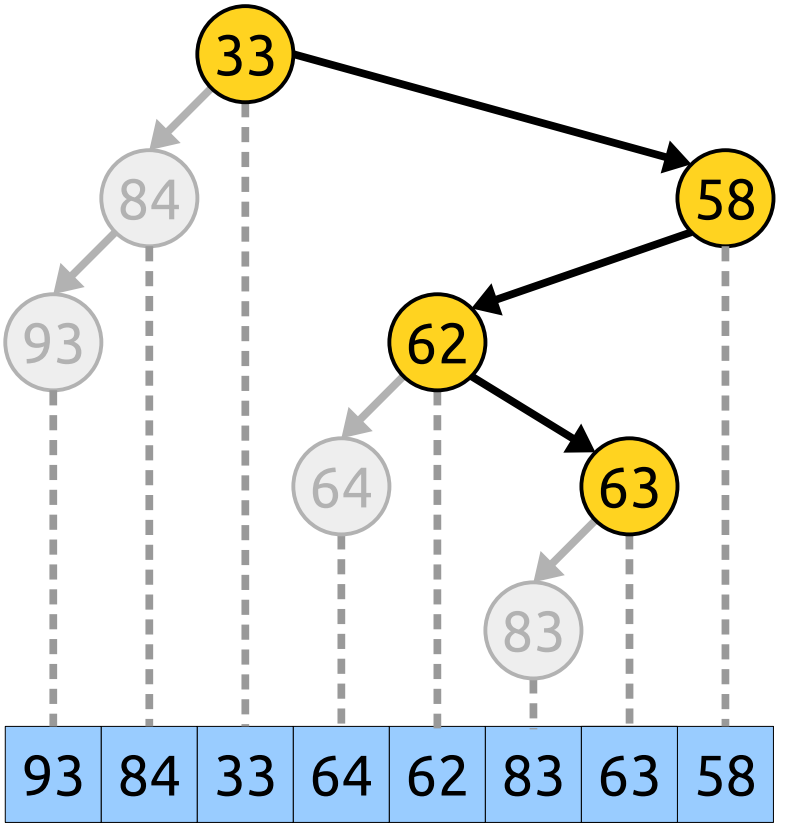


1	0	1	0	1	1	0	1	1	0	1	0	0			

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

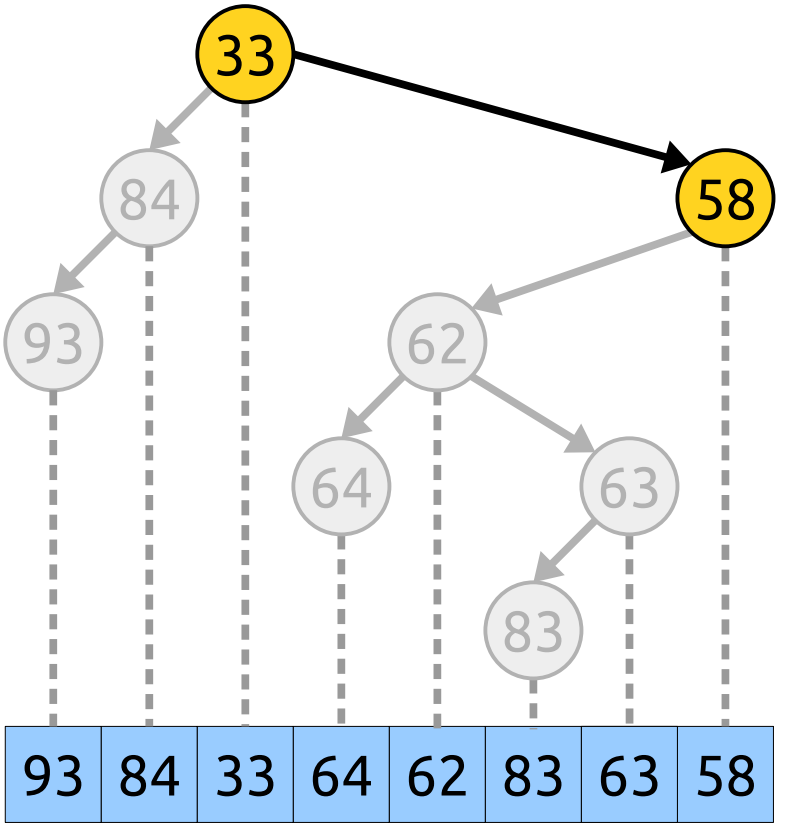


1	0	1	0	1	1	0	1	1	0	1	0	0			

0 means pop; 1 means push

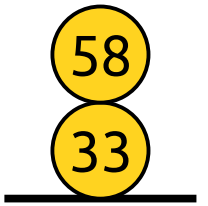
Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



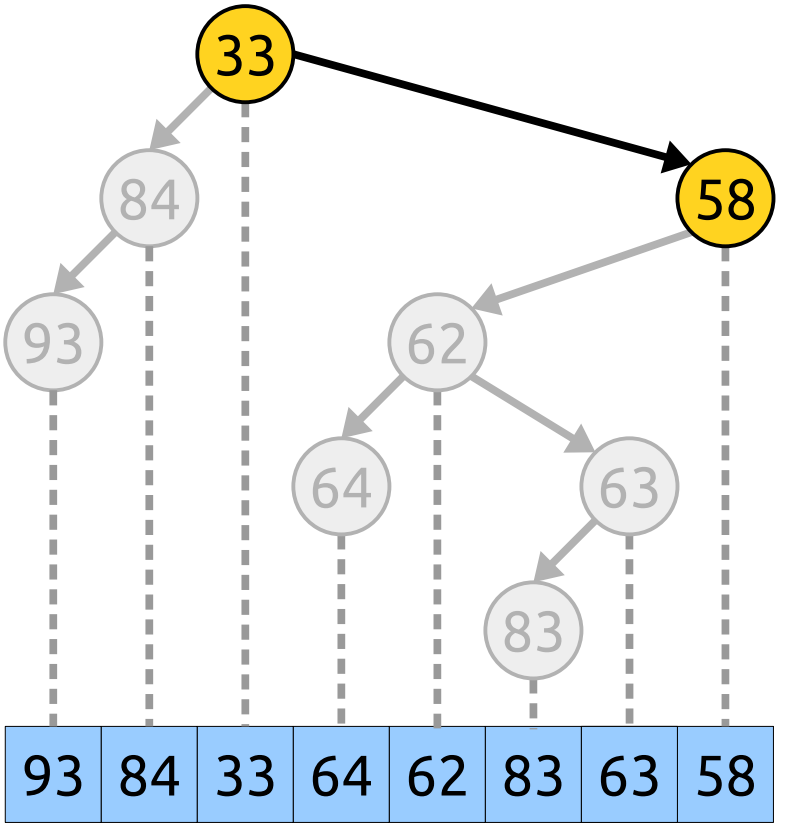
1	0	1	0	1	1	0	1	1	0	1	0	0	1		

0 means pop; 1 means push



Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

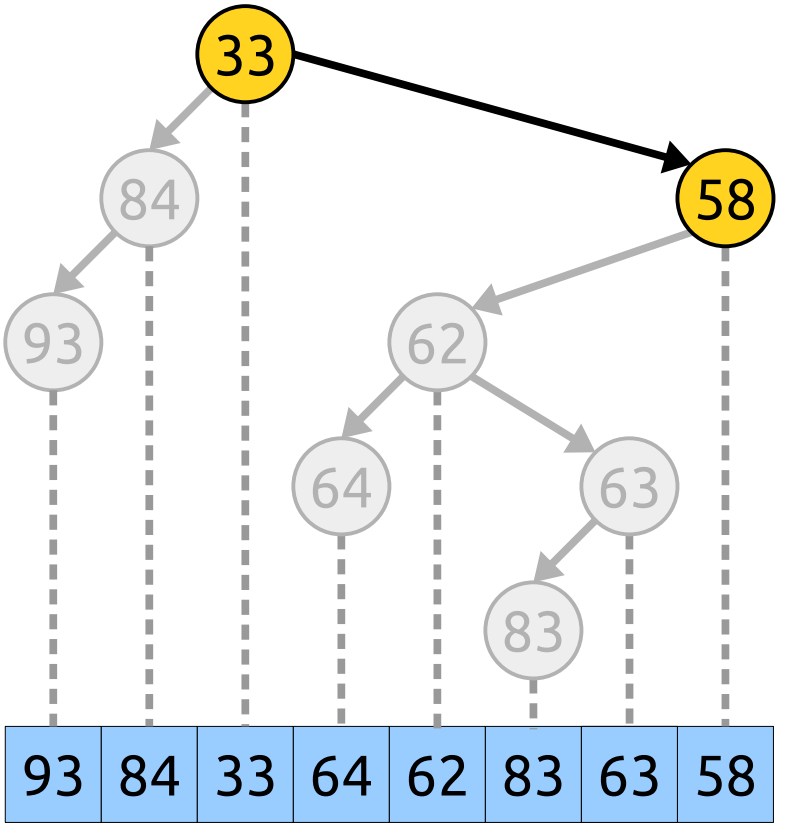


1	0	1	0	1	1	0	1	1	0	1	0	0	1	0	

0 means pop; 1 means push

Cartesian Tree Numbers

- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.

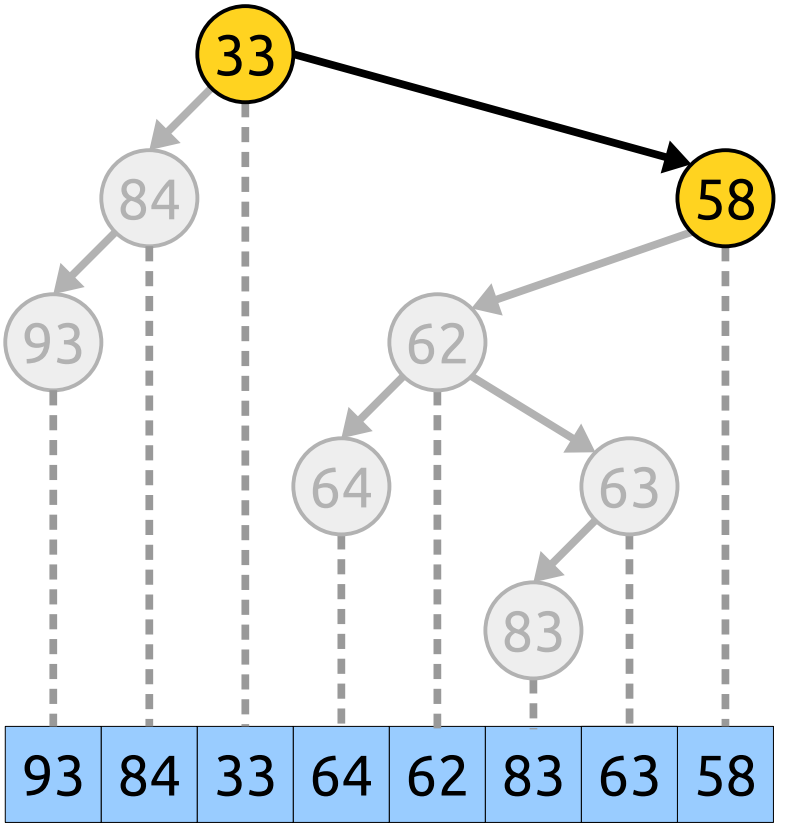


1	0	1	0	1	1	0	1	1	0	1	0	0	1	0	0

0 means pop; 1 means push

Cartesian Tree Numbers

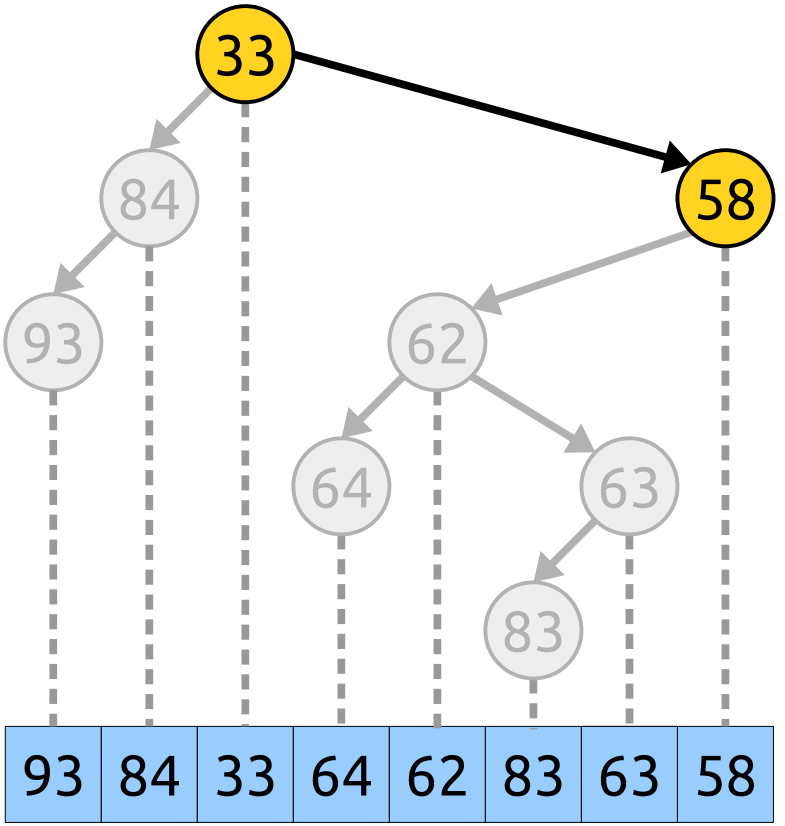
- There are at most $2b$ stack operations during the execution of the algorithm: b pushes and no more than b pops.
- Represent the execution of the algorithm as a $2b$-bit number, where 1 means “push” and 0 means “pop.” We'll pad the end with 0's (pretend we pop everything from the stack).
- This number is the ***Cartesian tree number*** of a block.



1 0 1 0 1 1 0 1 1 0 1 0 0 1 0 0 (44,452)

Cartesian Tree Numbers

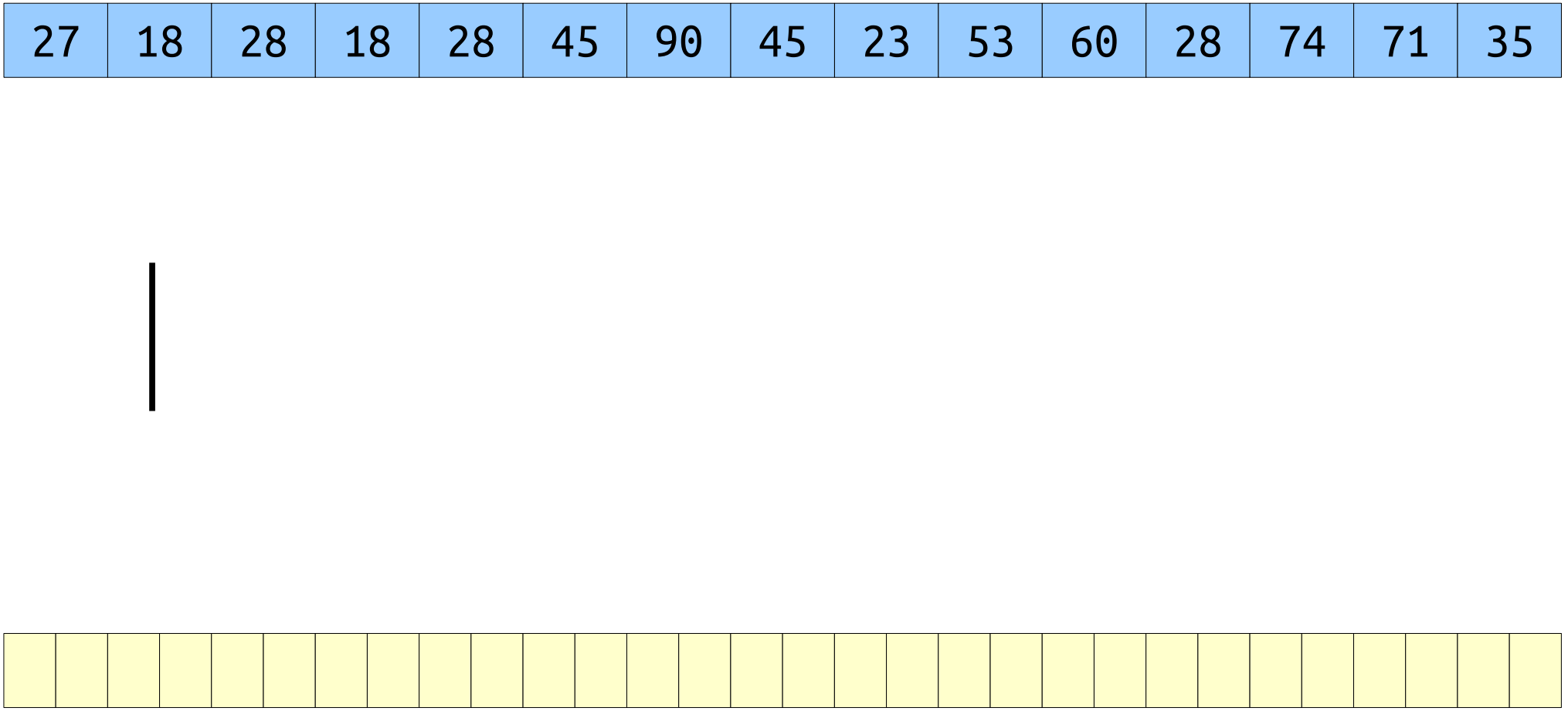
- Two blocks can share an RMQ structure iff they have the same Cartesian tree.
- ***Observation:*** If all we care about is finding blocks that can share RMQ structures, ***we never need to build Cartesian trees!*** Instead, we can just compute the Cartesian tree number for each block.



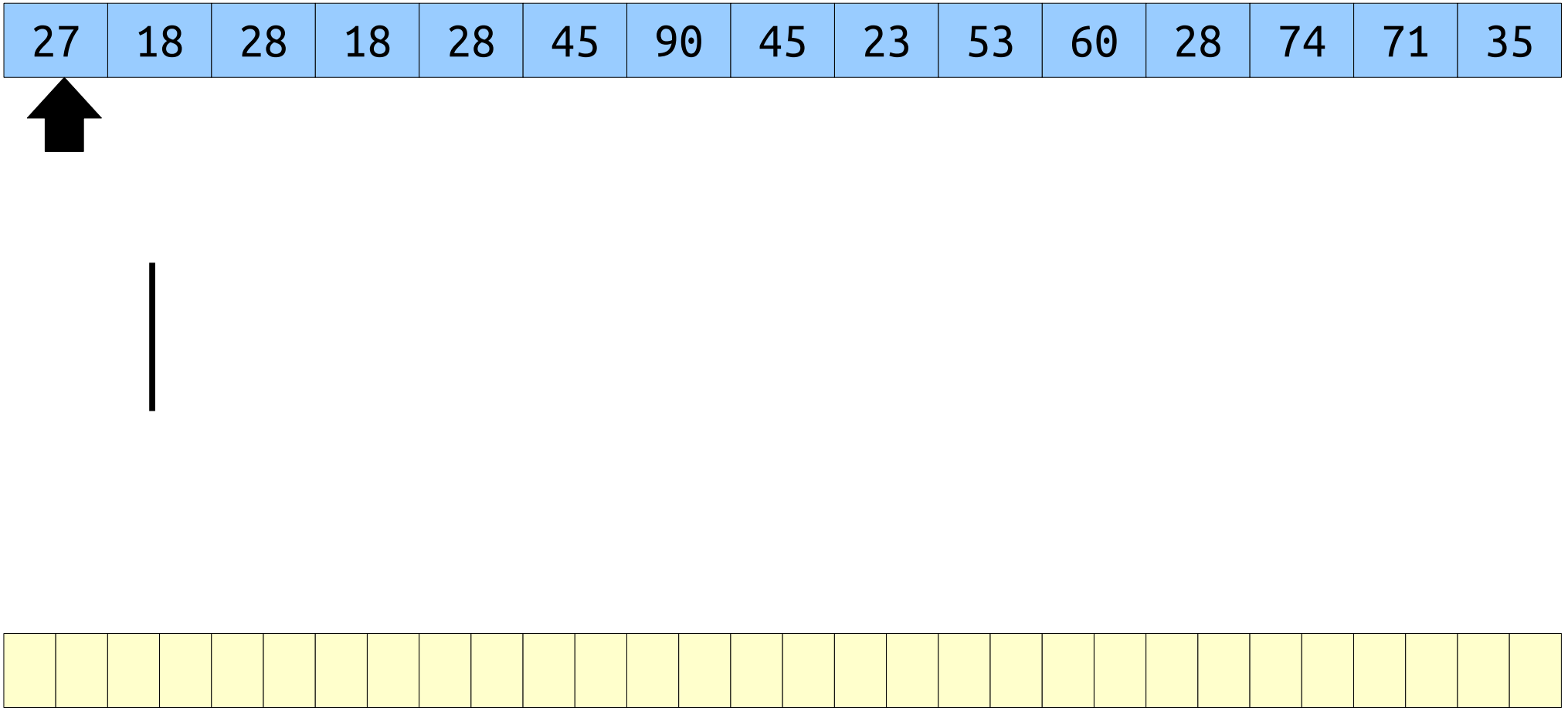
1 0 1 0 1 1 0 1 1 0 1 0 0 1 0 0

(44,452)

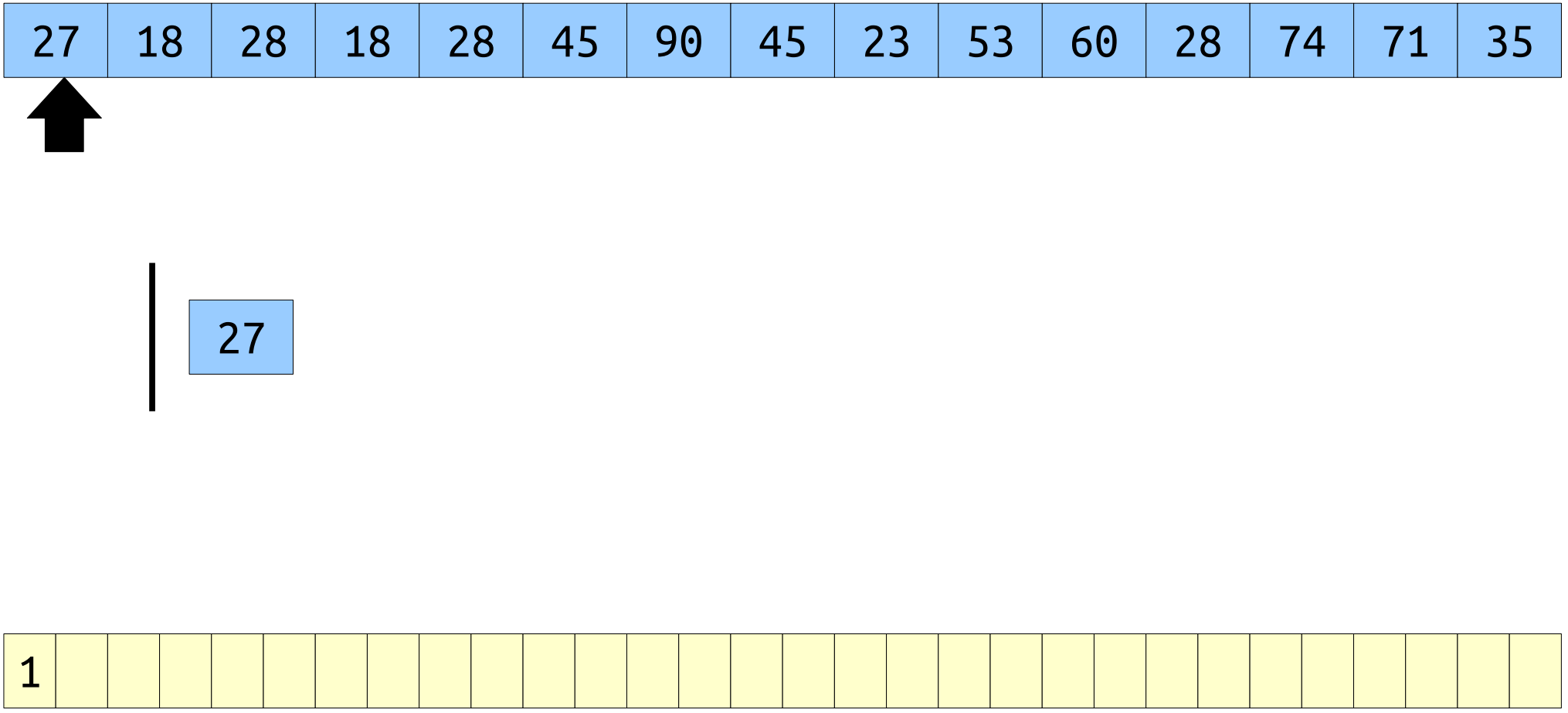
Treeless Tree Numbers



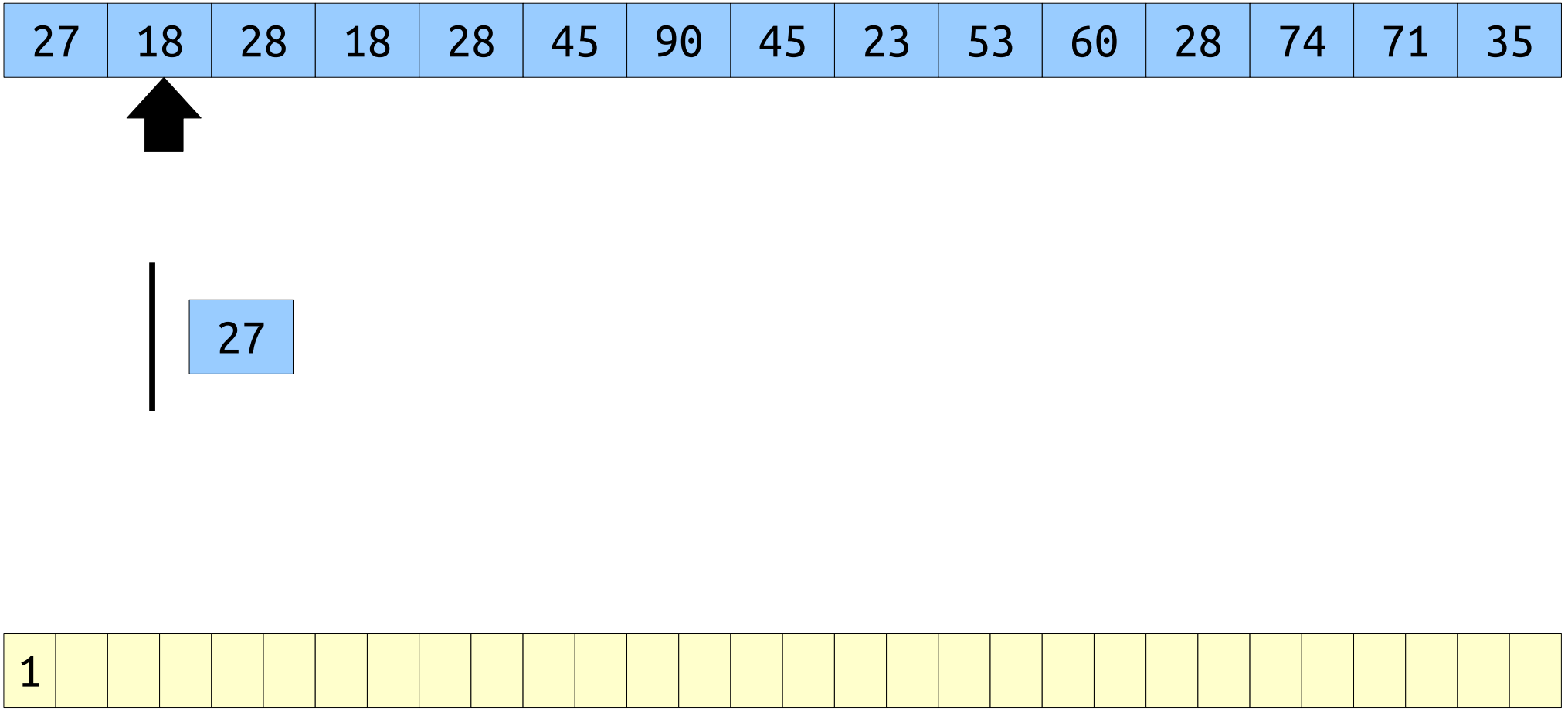
Treeless Tree Numbers



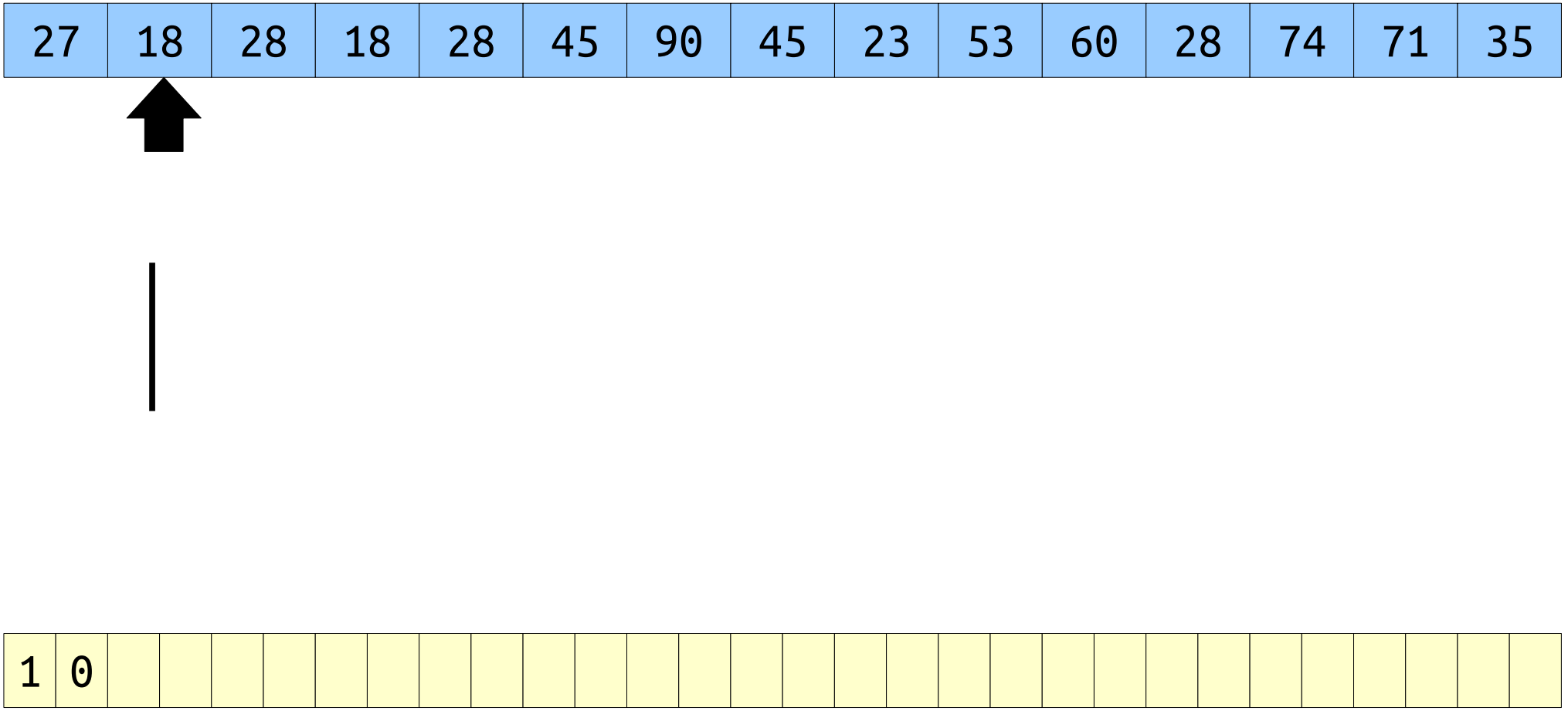
Treeless Tree Numbers



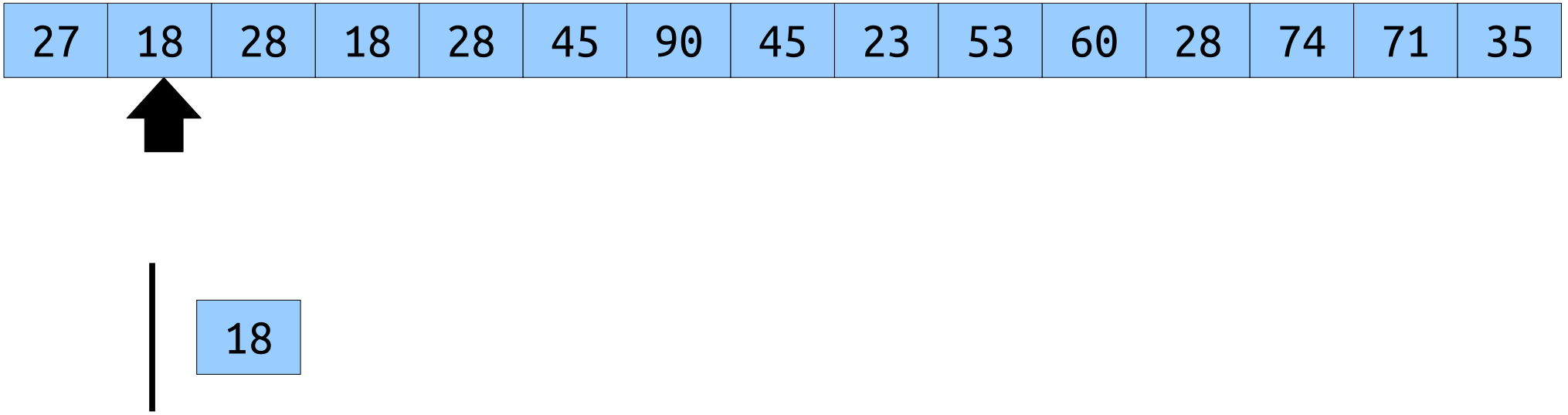
Treeless Tree Numbers



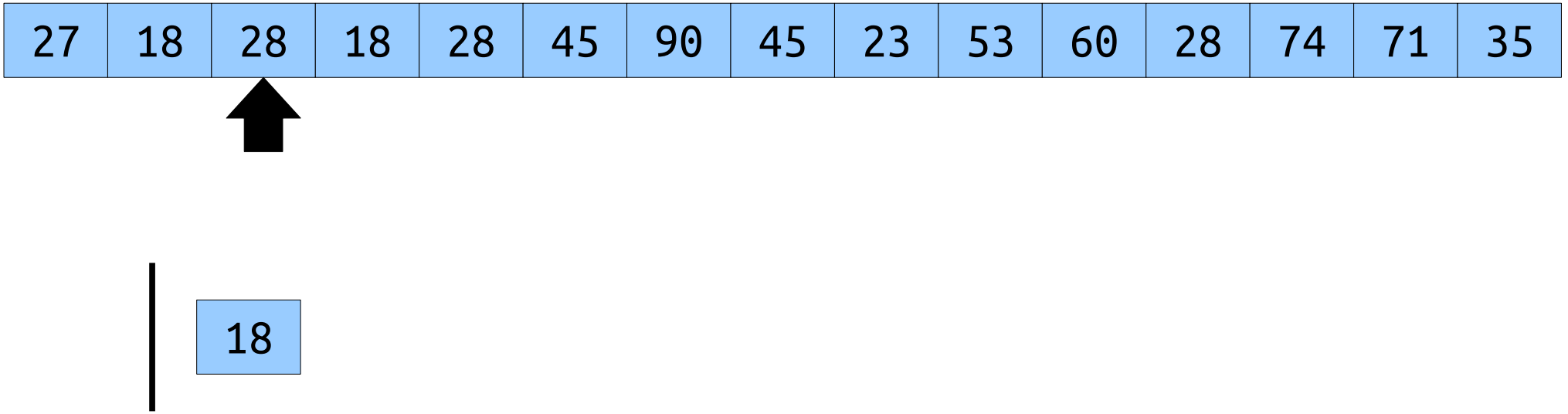
Treeless Tree Numbers



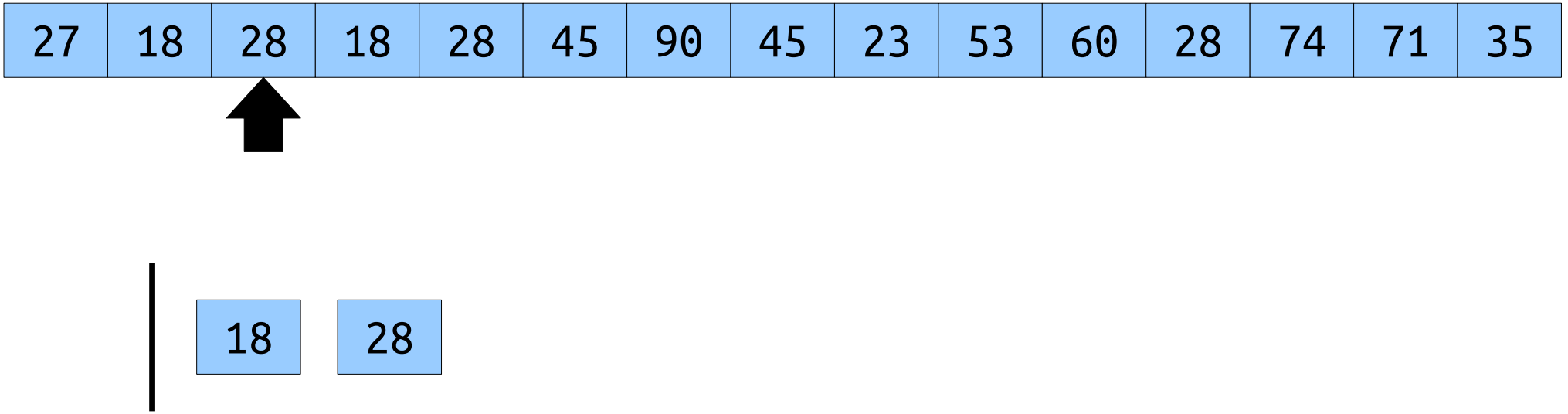
Treeless Tree Numbers

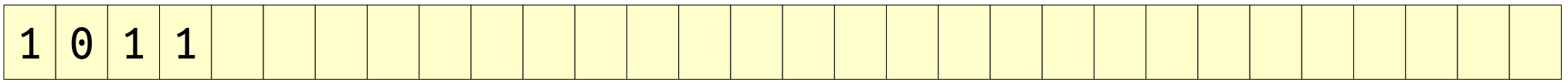


Treeless Tree Numbers

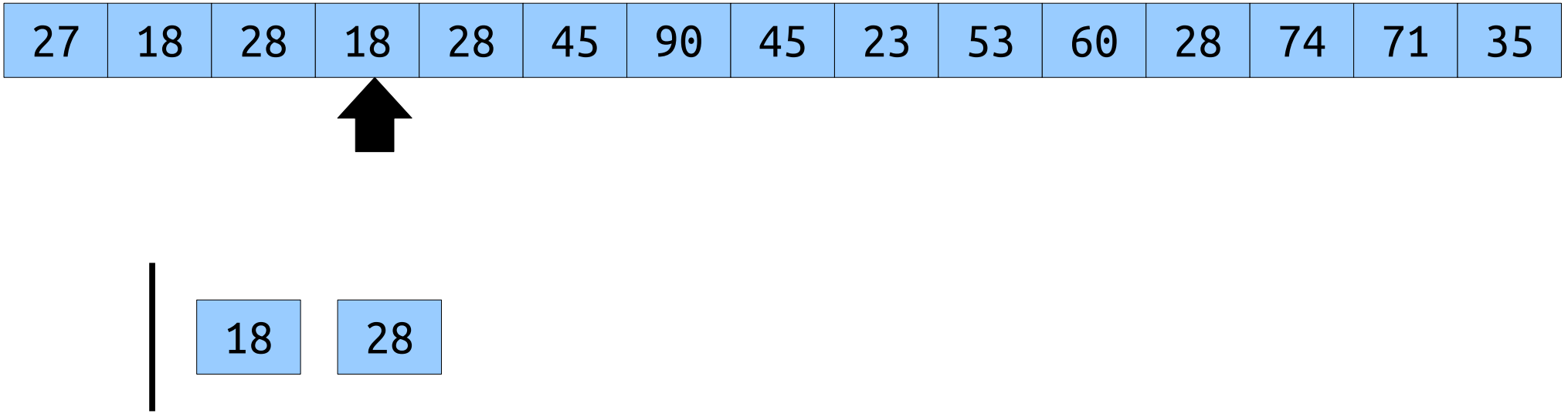


Treeless Tree Numbers

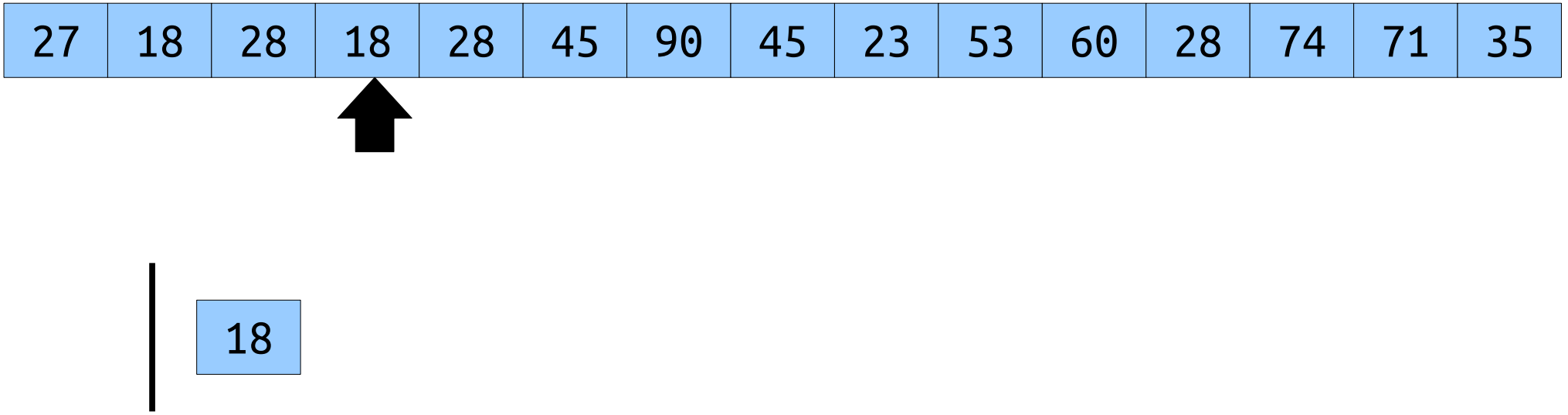




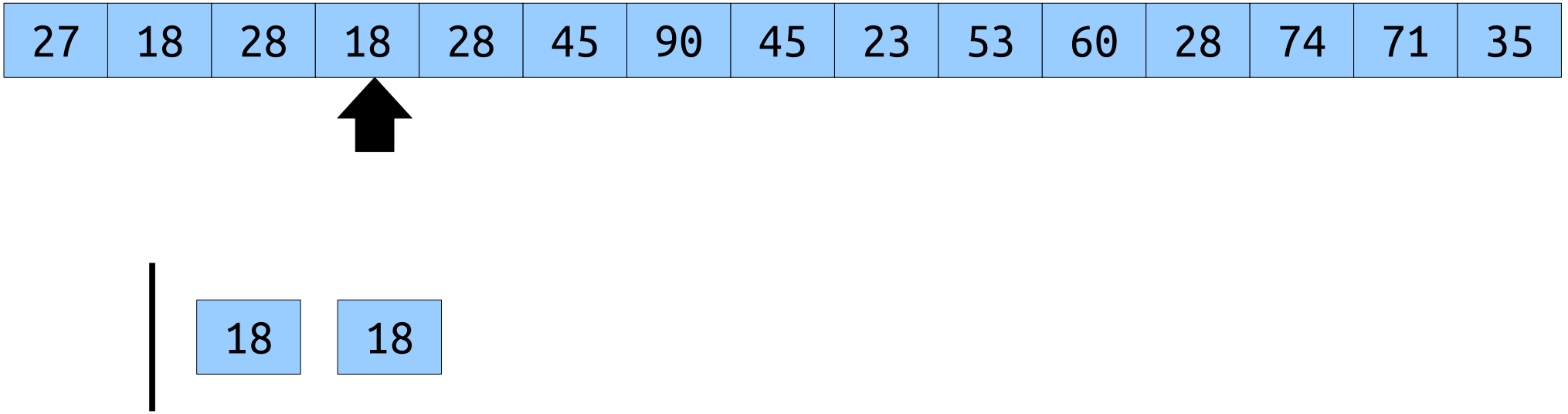
Treeless Tree Numbers

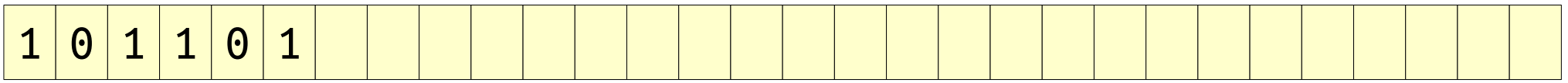


Treeless Tree Numbers

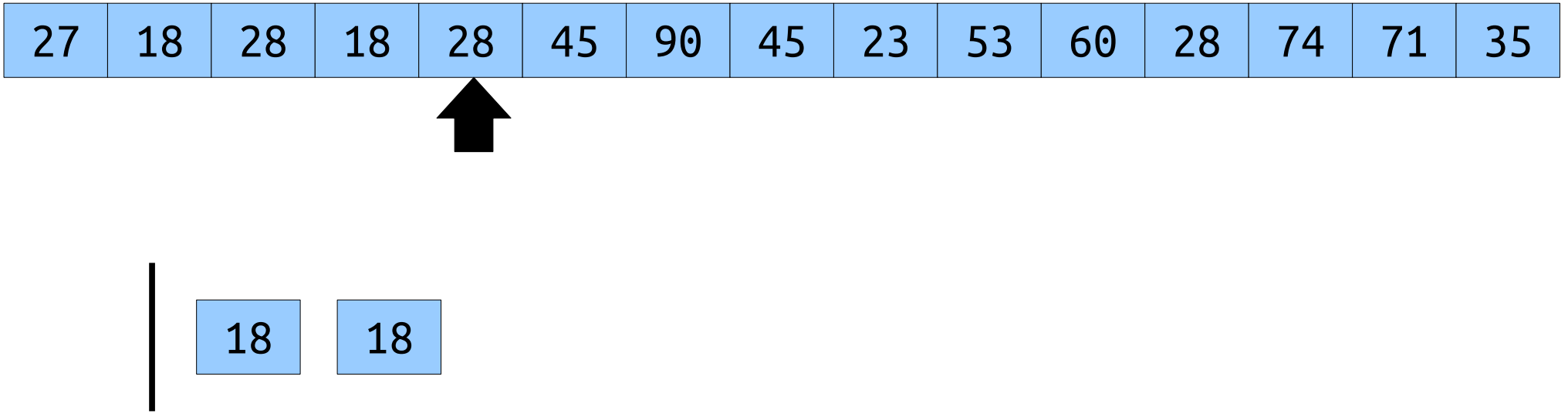


Treeless Tree Numbers

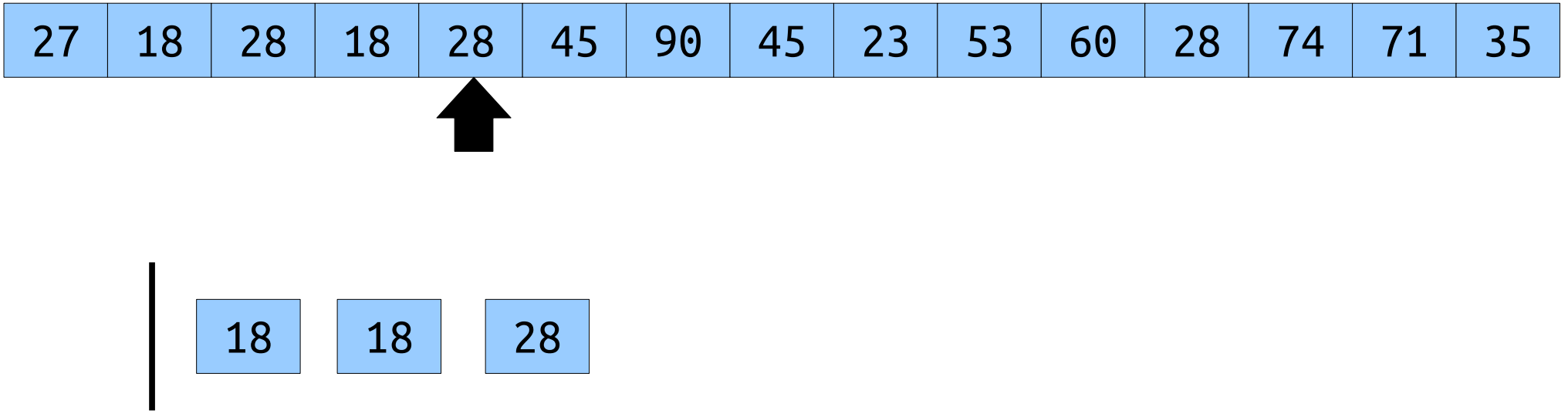


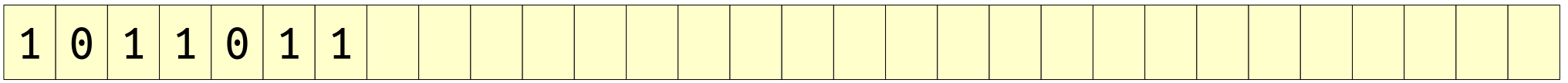


Treeless Tree Numbers

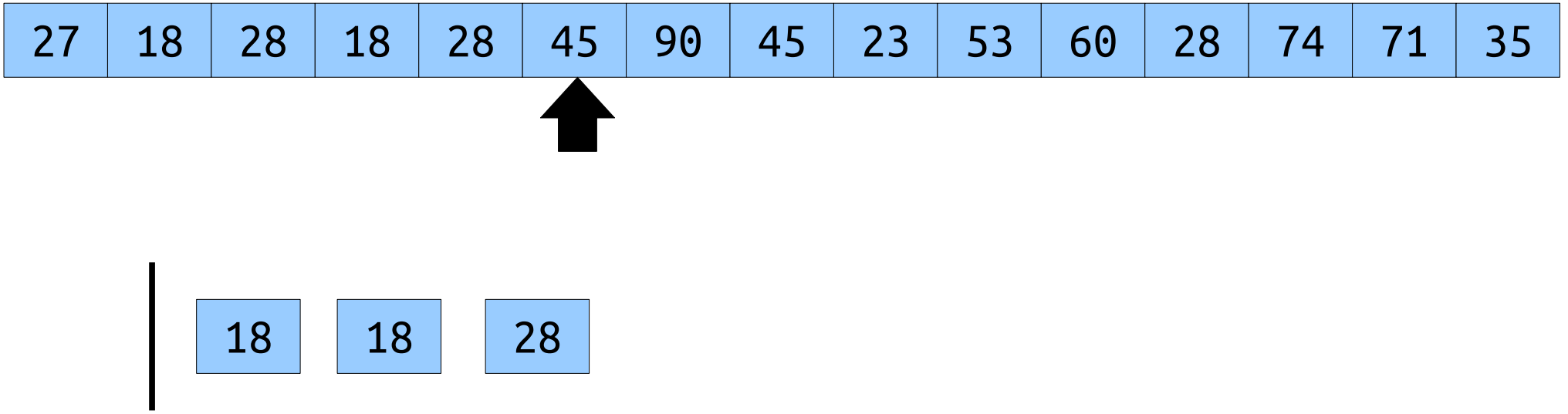


Treeless Tree Numbers

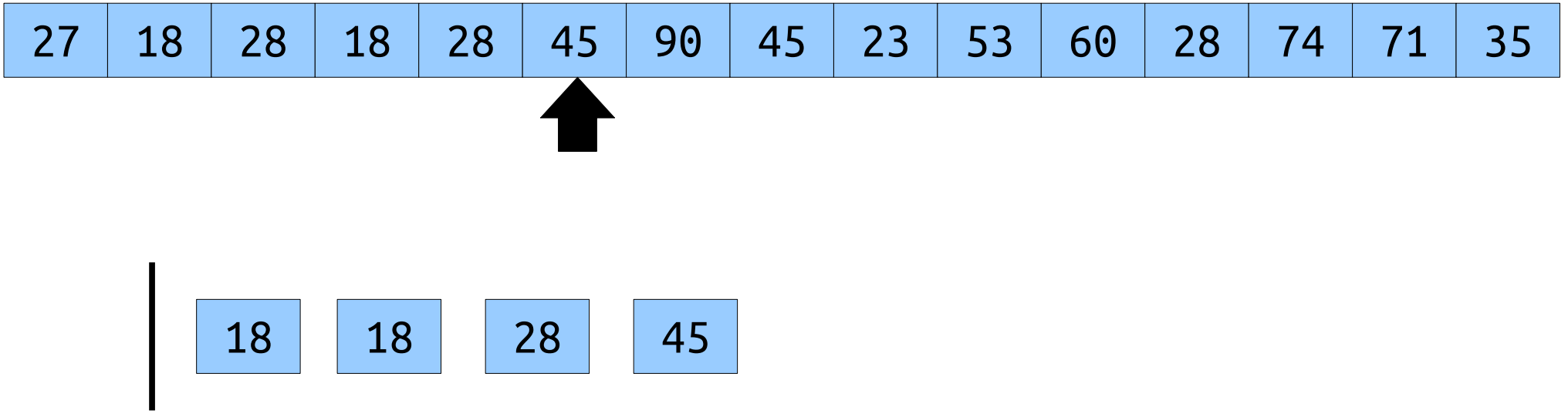




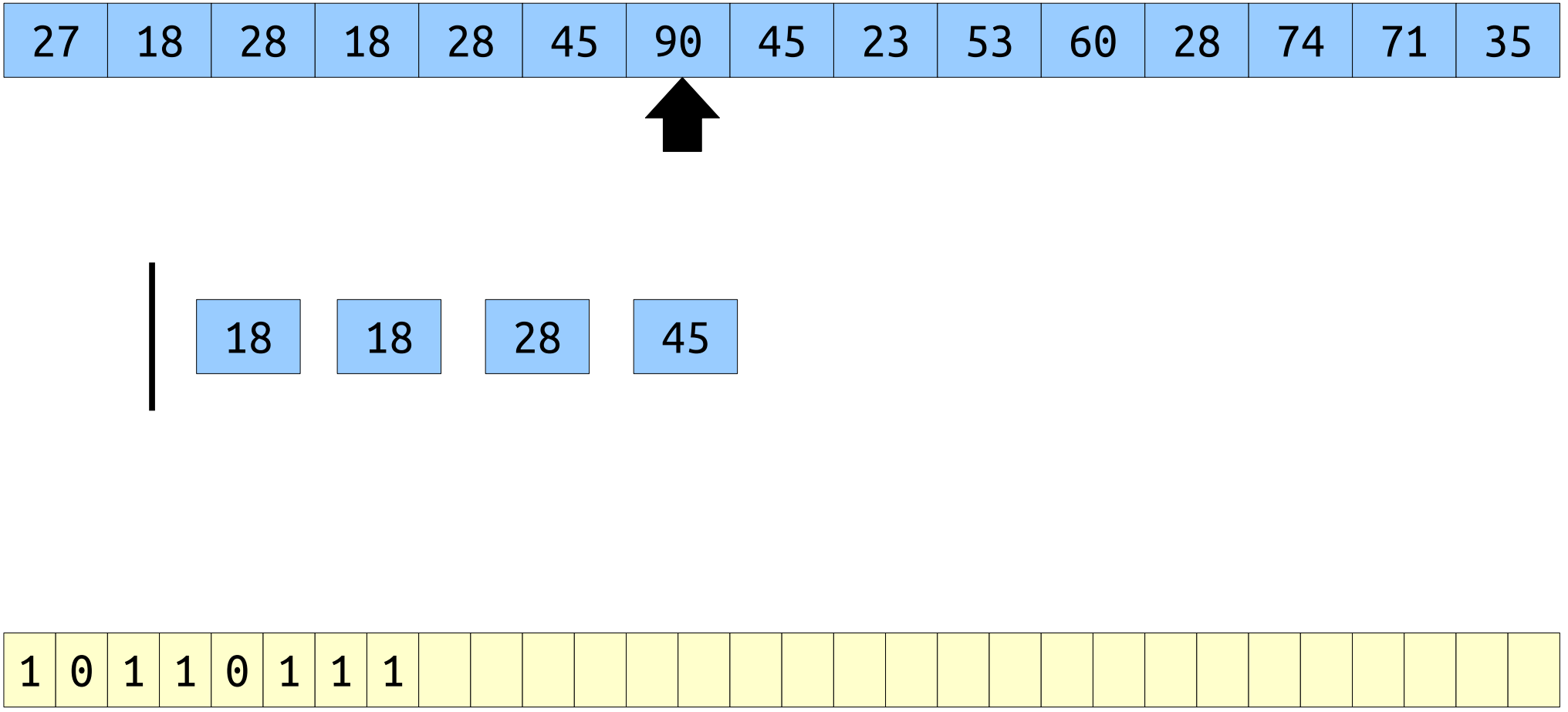
Treeless Tree Numbers



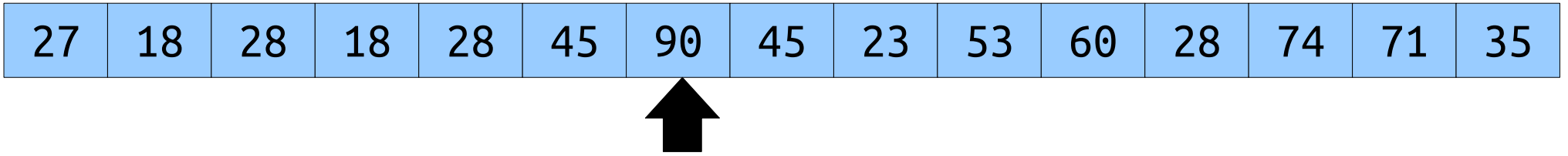
Treeless Tree Numbers

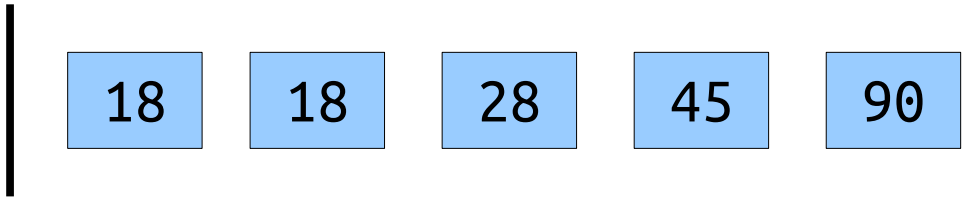


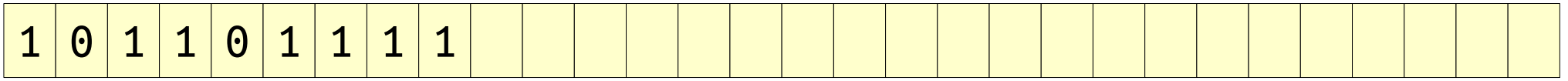
Treeless Tree Numbers



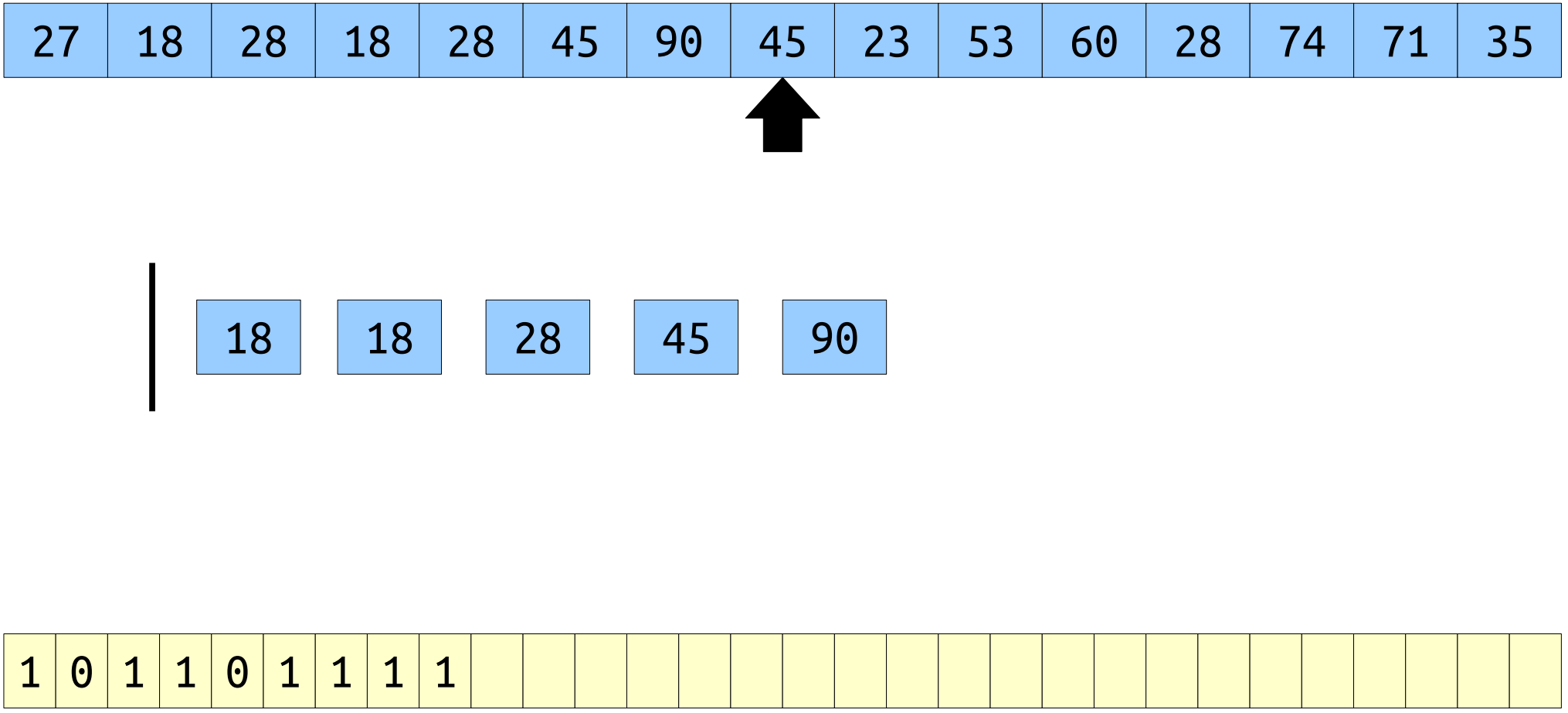
Treeless Tree Numbers



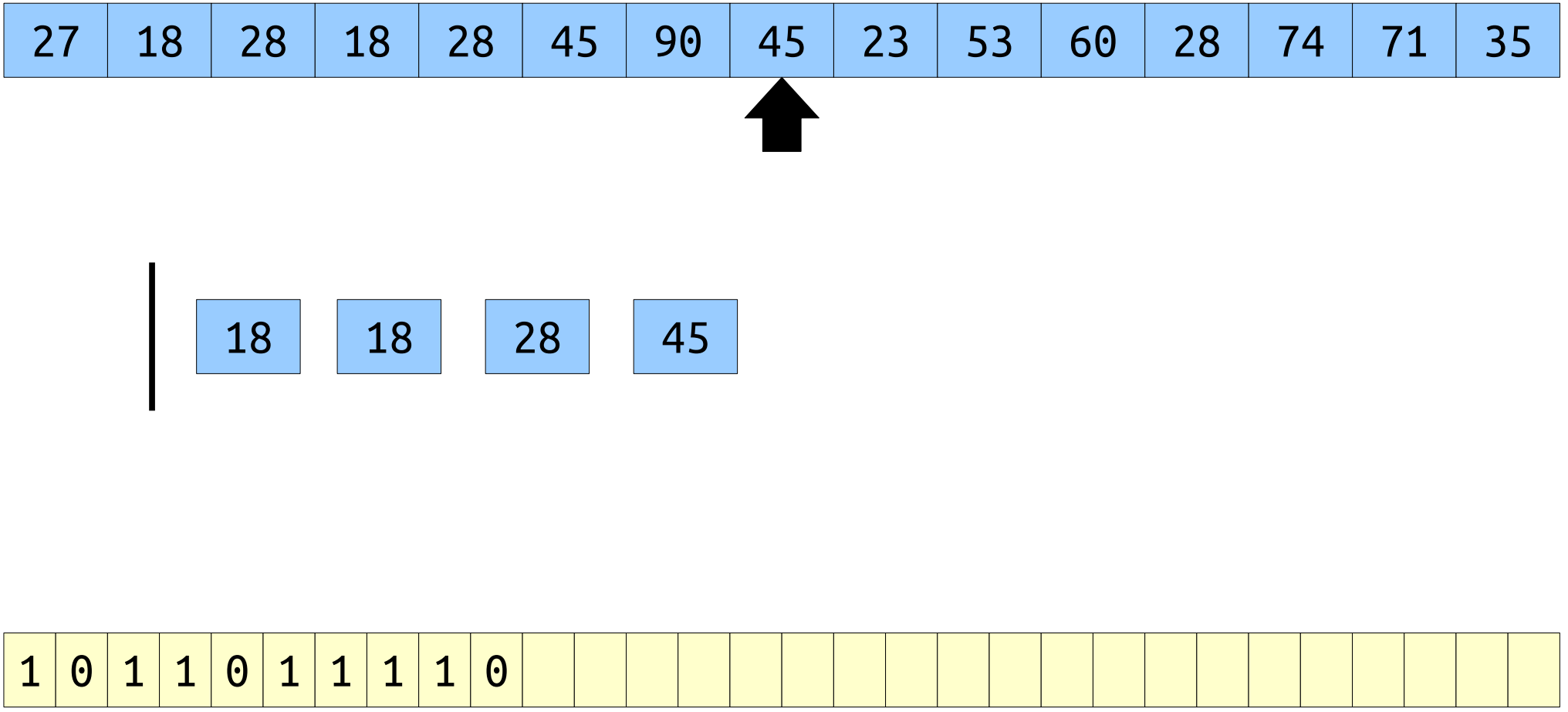




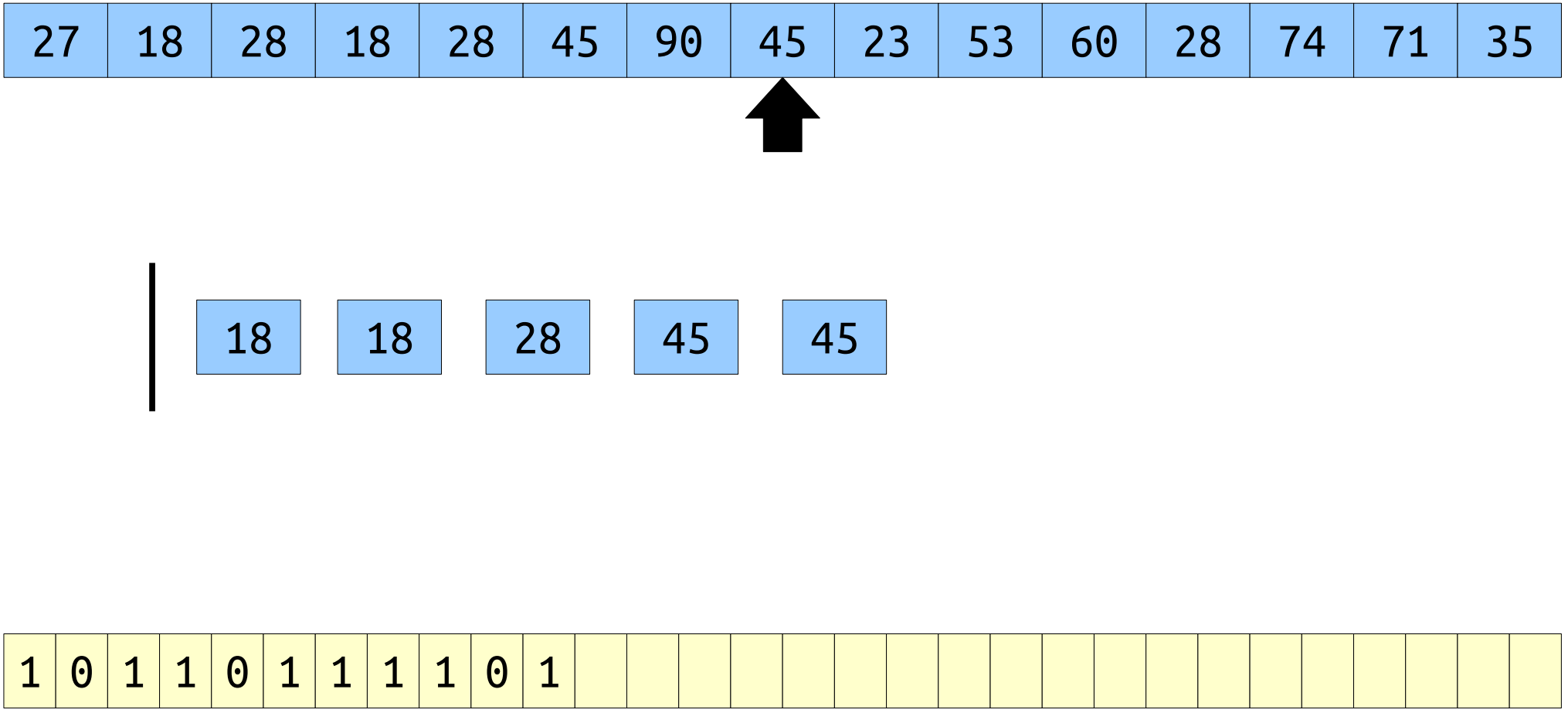
Treeless Tree Numbers



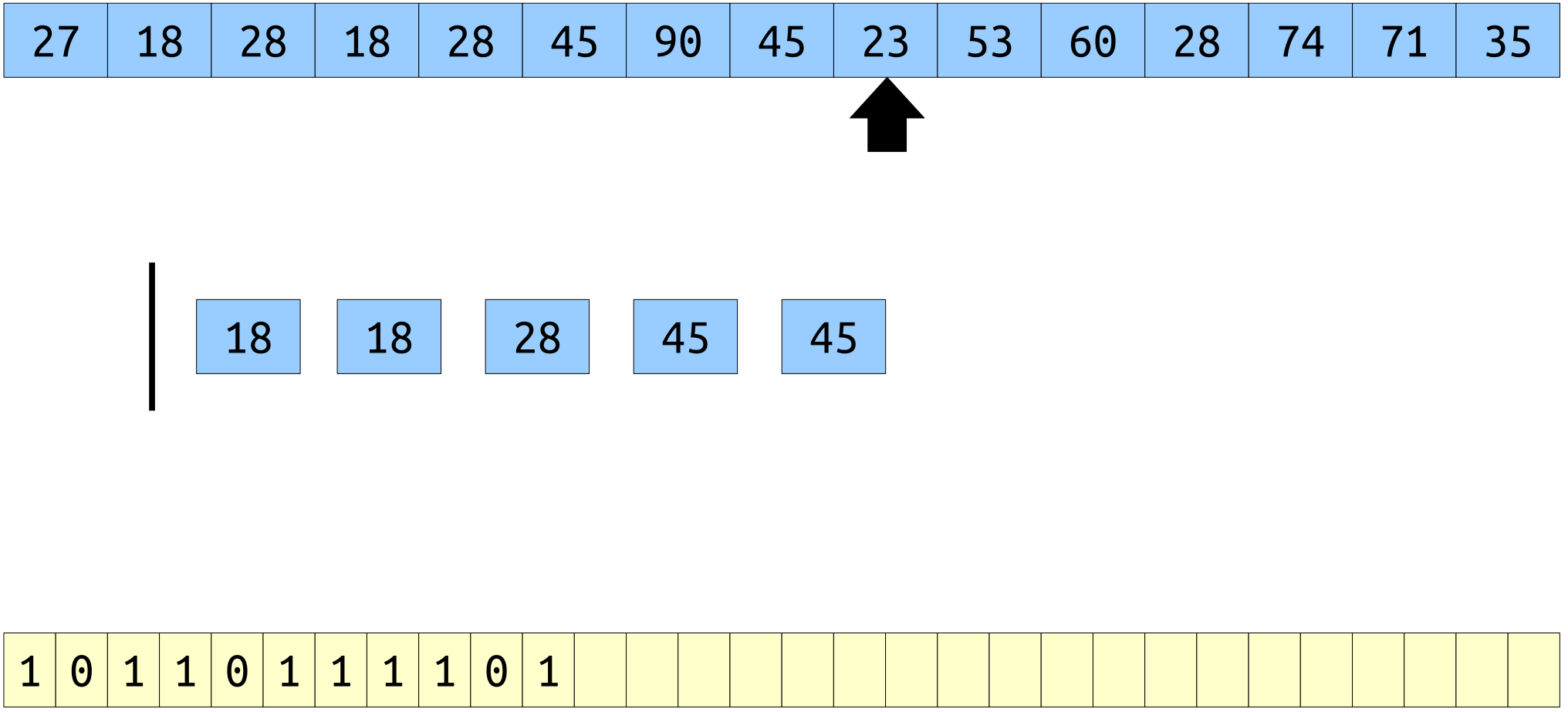
Treeless Tree Numbers



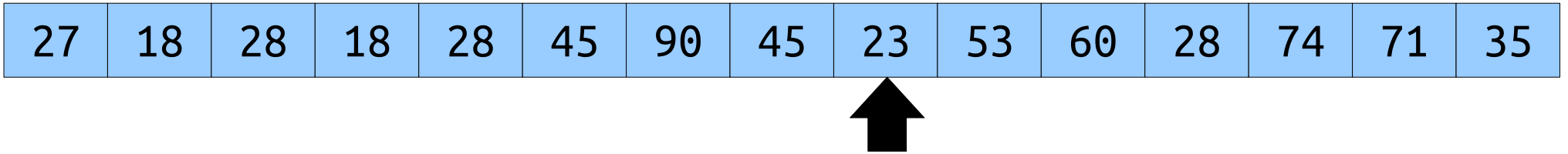
Treeless Tree Numbers

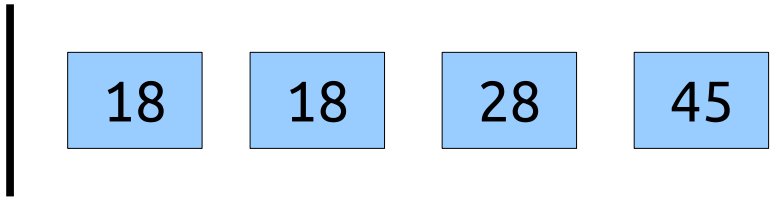


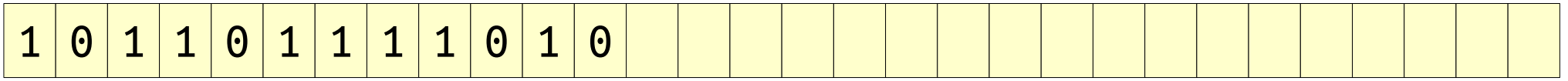
Treeless Tree Numbers



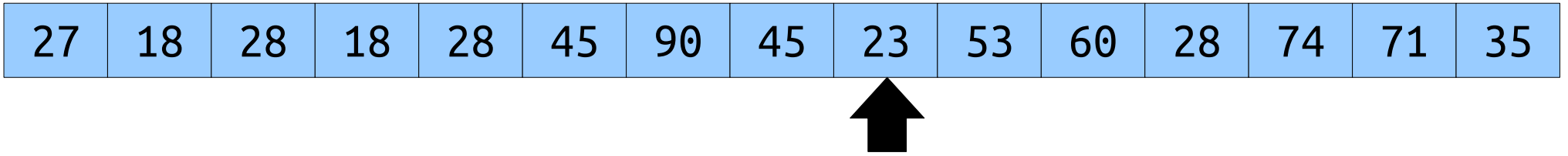
Treeless Tree Numbers

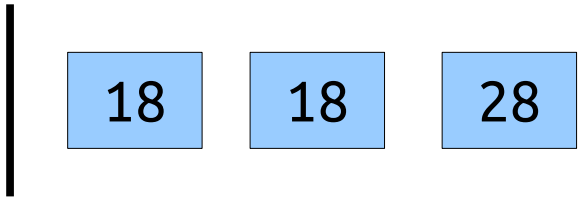


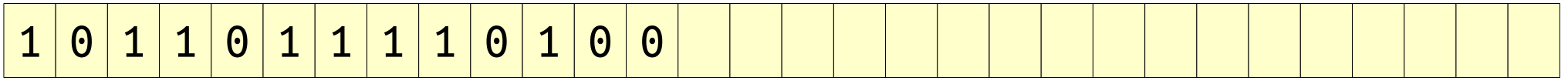




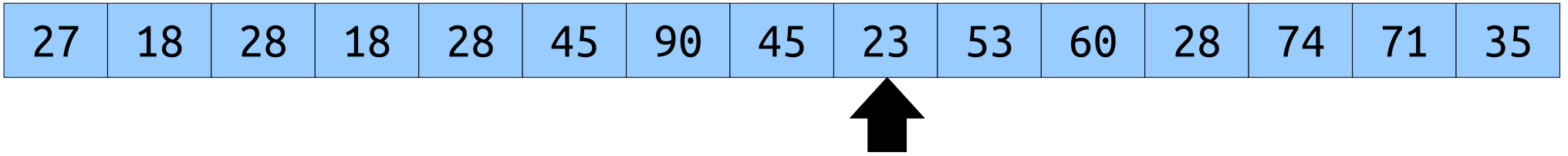
Treeless Tree Numbers

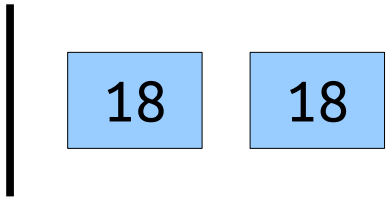


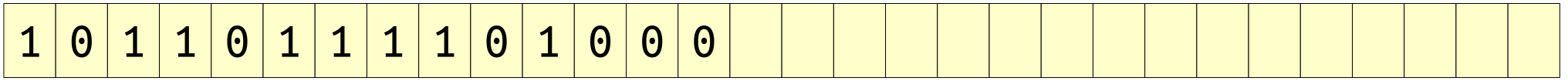




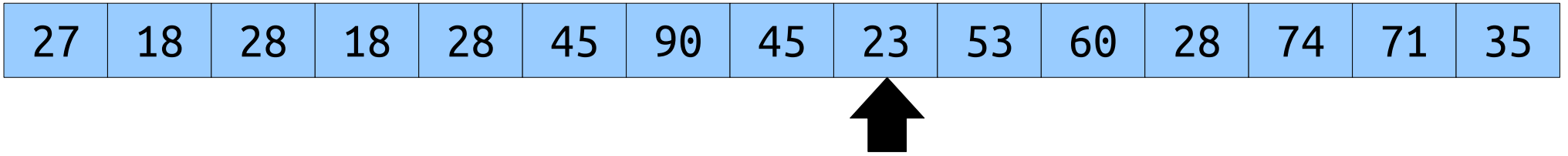
Treeless Tree Numbers

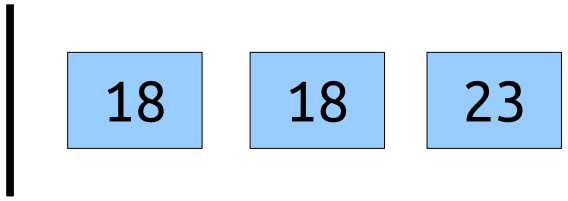


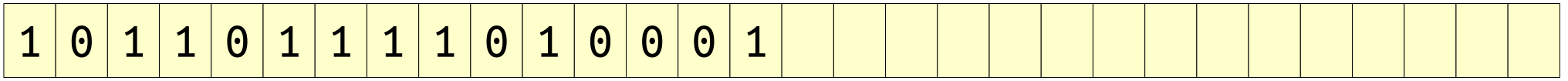




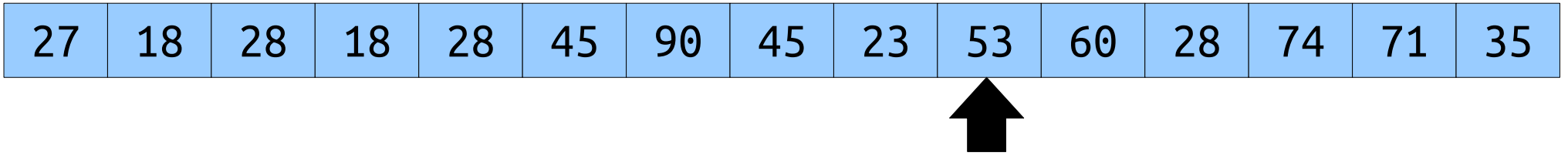
Treeless Tree Numbers

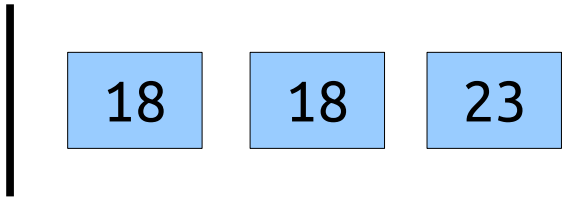


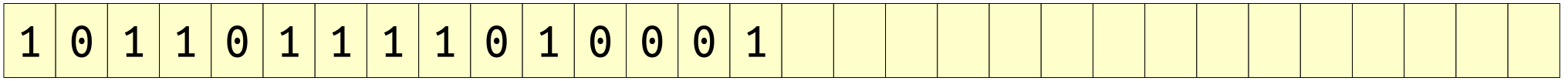




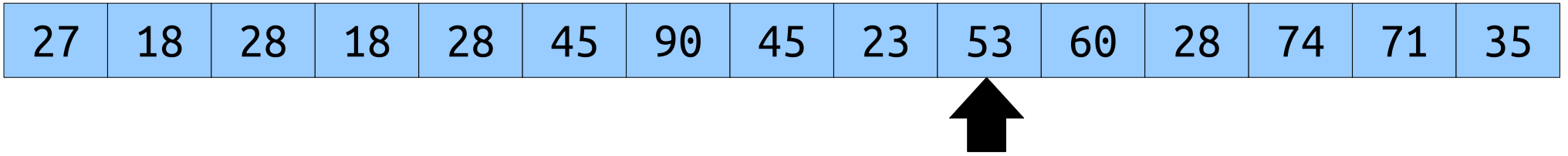
Treeless Tree Numbers

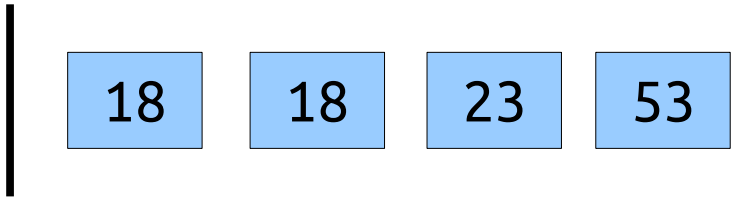


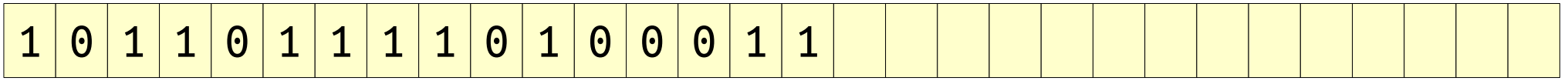




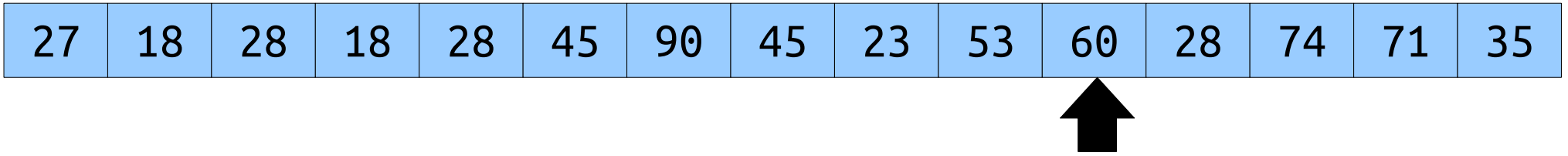
Treeless Tree Numbers

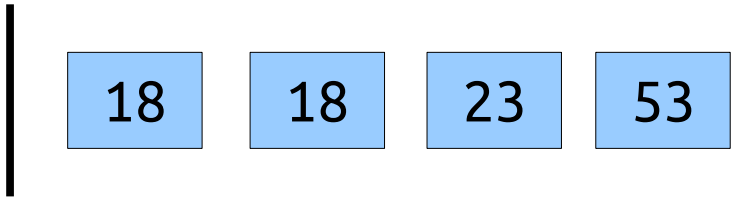


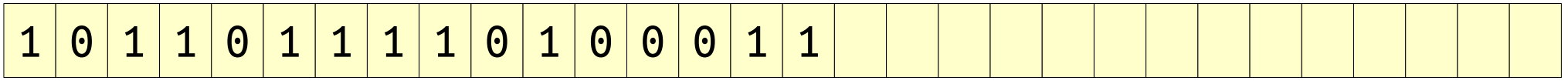




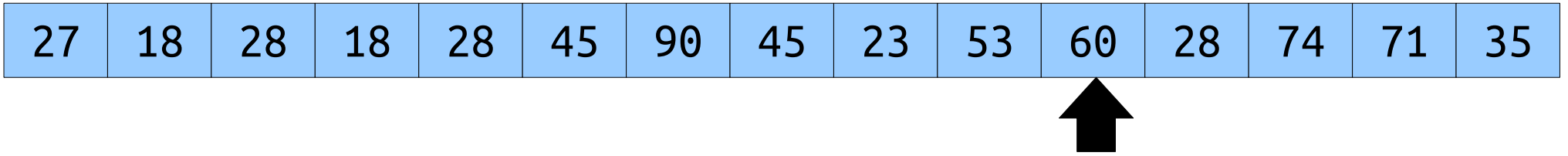
Treeless Tree Numbers

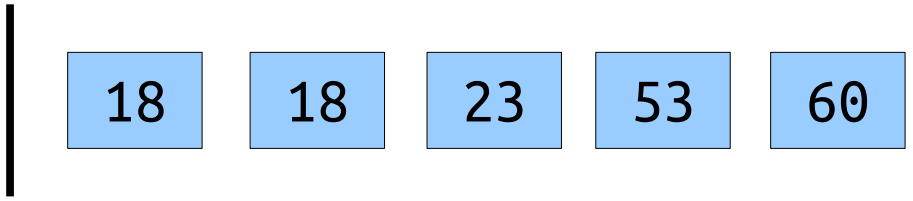


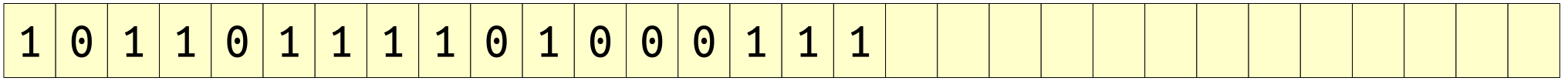




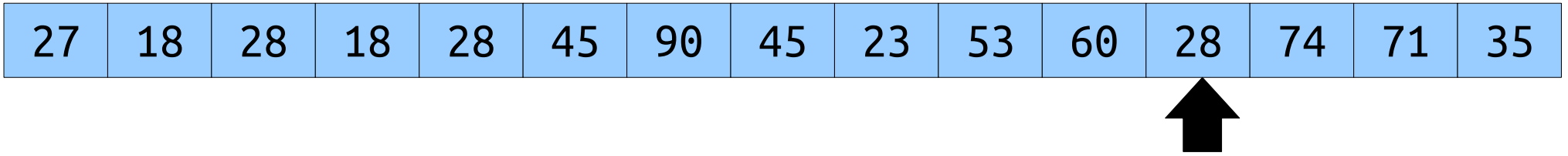
Treeless Tree Numbers

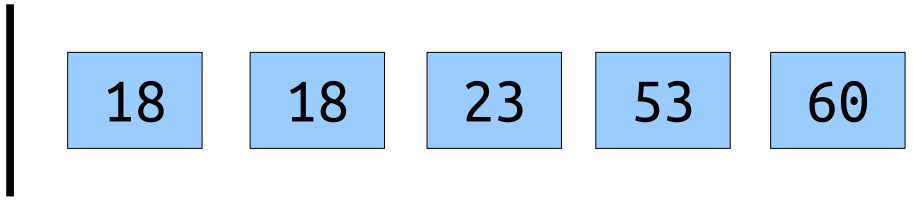


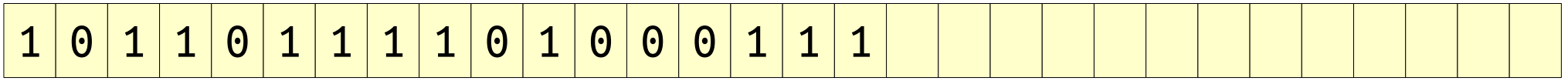




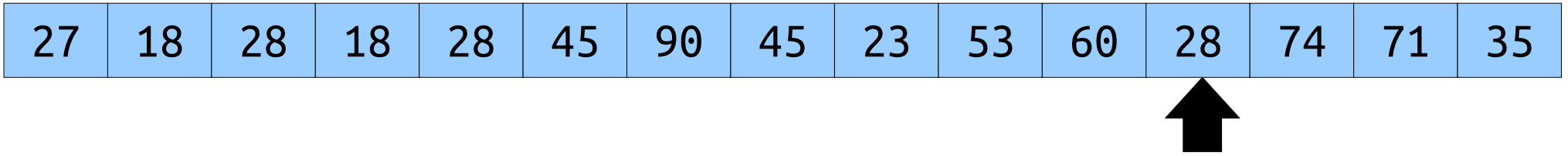
Treeless Tree Numbers

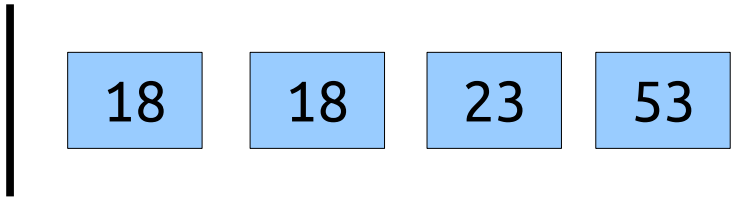


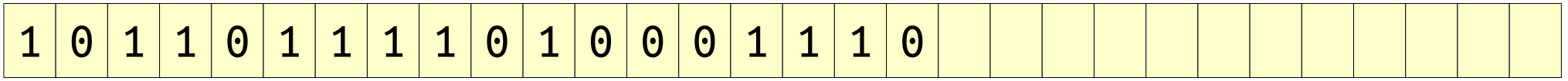




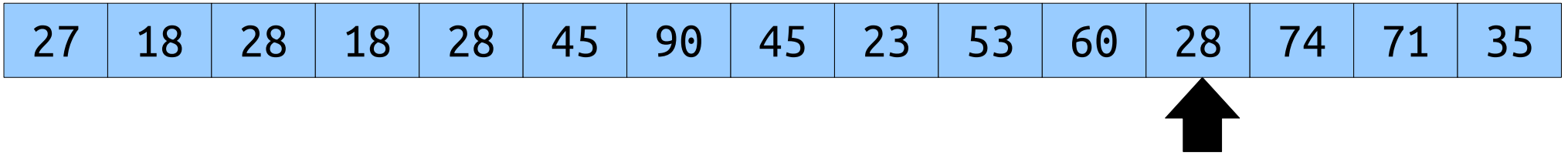
Treeless Tree Numbers

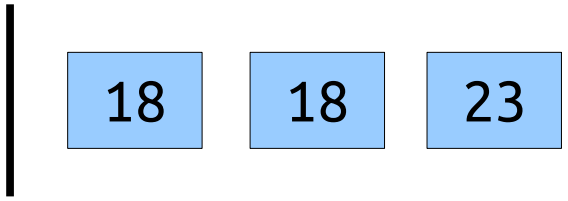


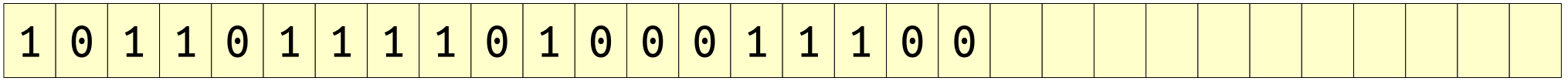




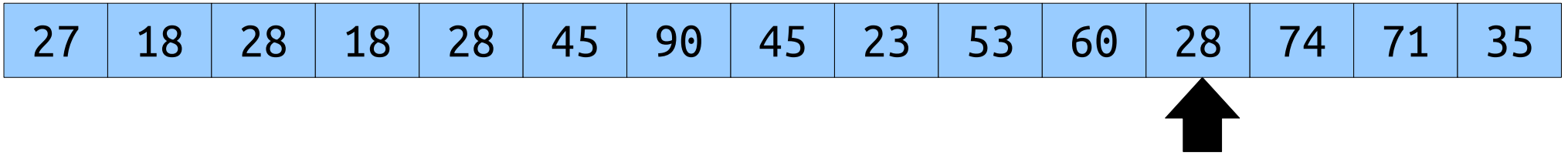
Treeless Tree Numbers

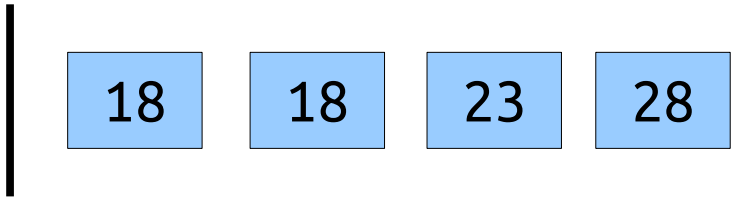


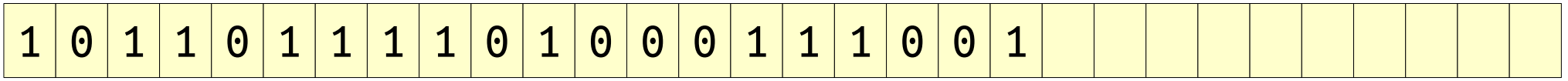




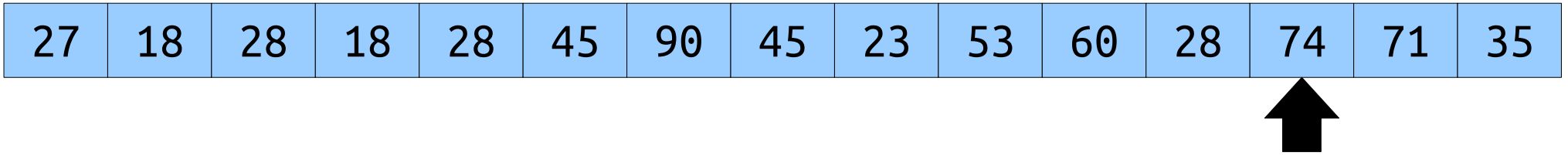
Treeless Tree Numbers

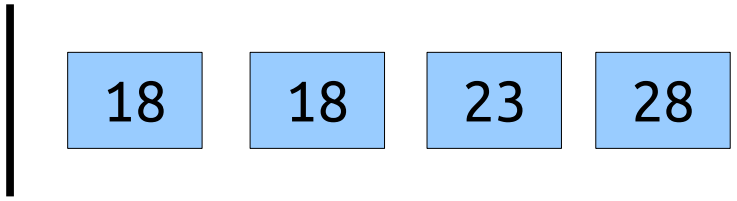


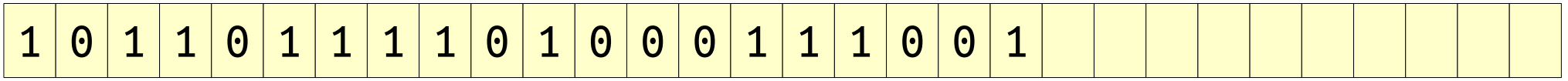




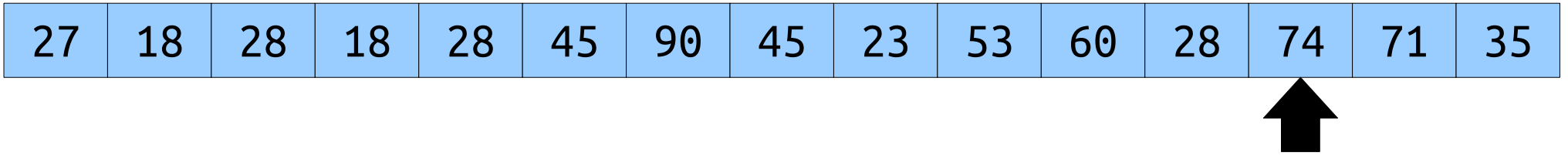
Treeless Tree Numbers

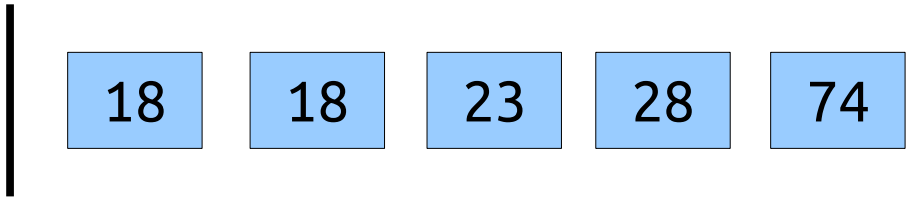


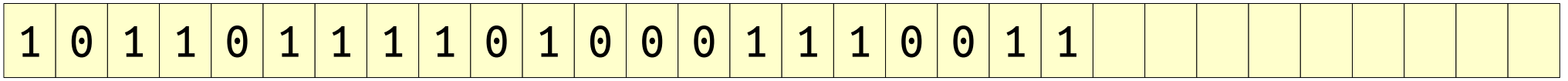




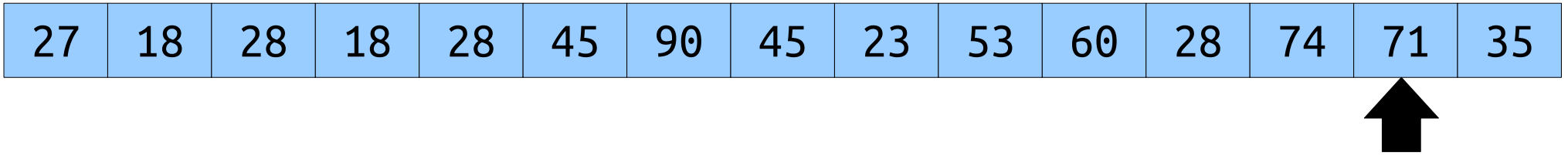
Treeless Tree Numbers

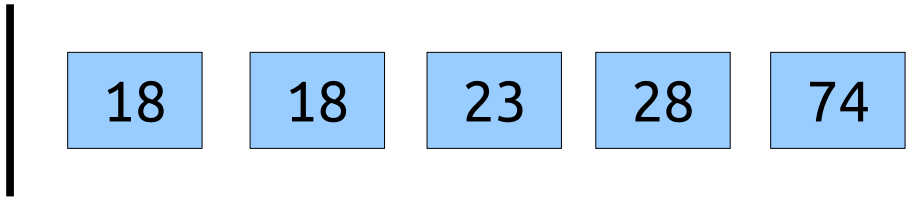


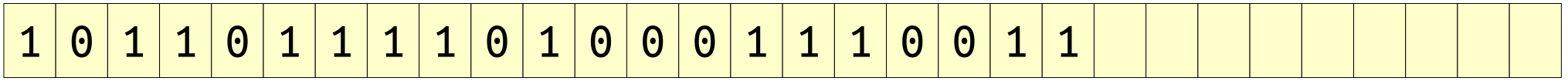




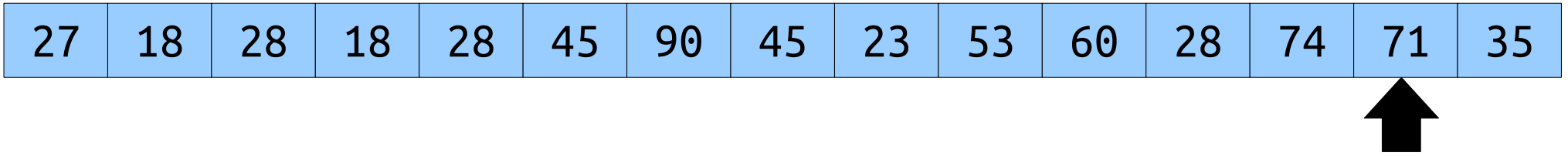
Treeless Tree Numbers

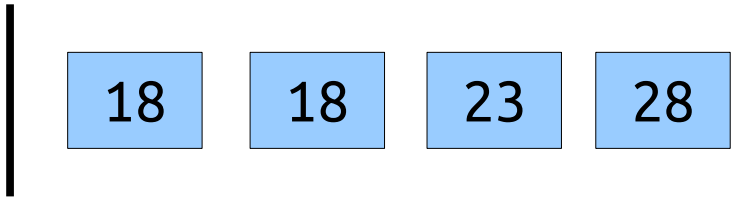


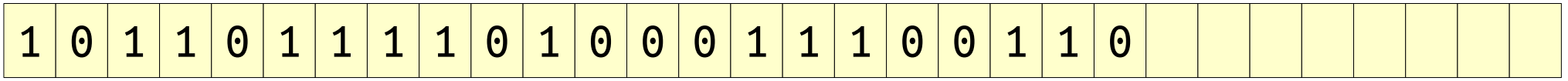




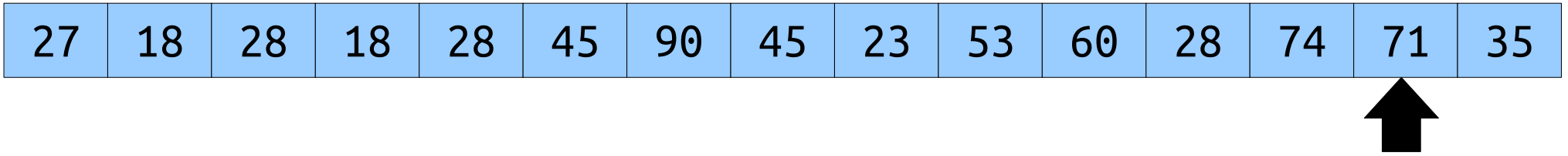
Treeless Tree Numbers

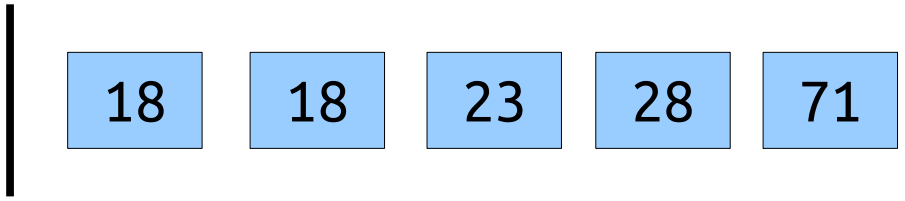


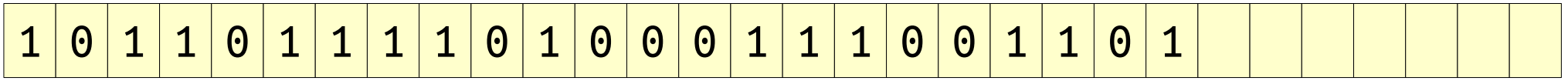




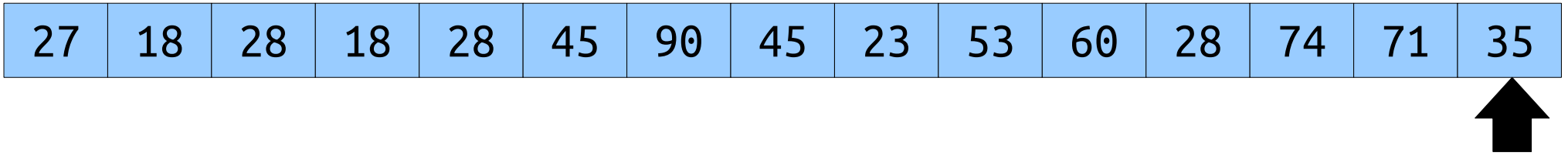
Treeless Tree Numbers

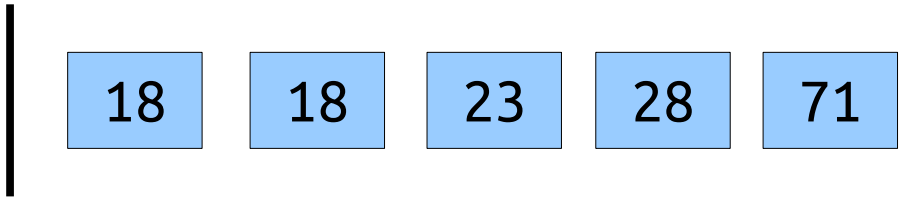


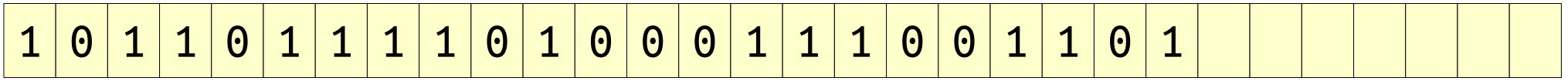




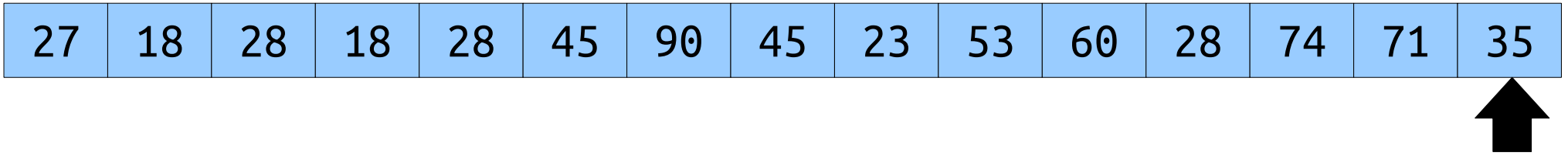
Treeless Tree Numbers

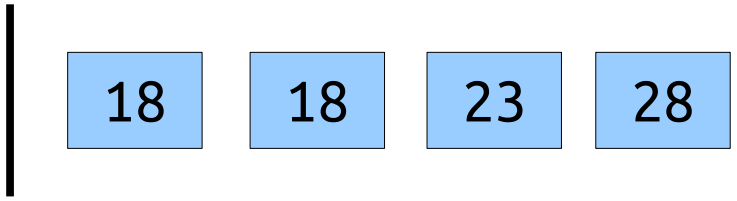


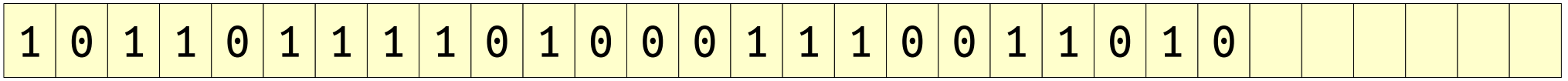




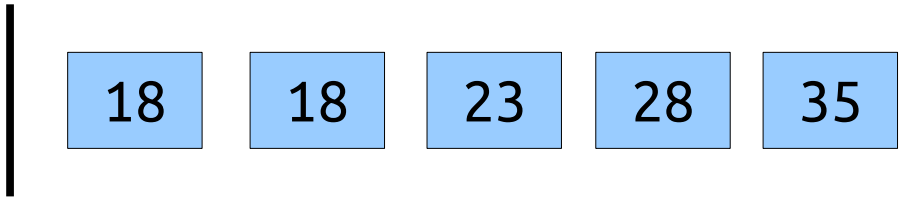
Treeless Tree Numbers

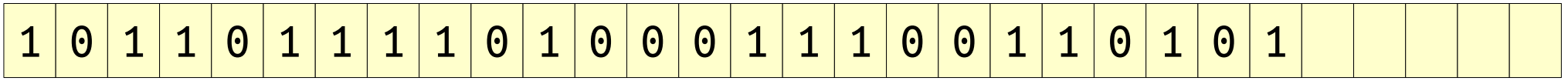






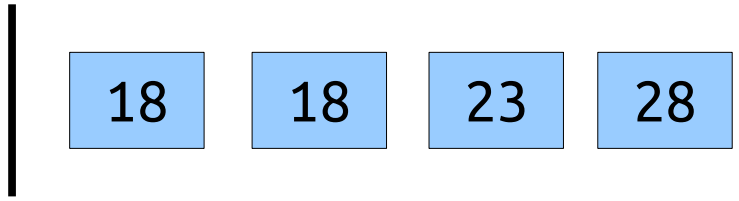
Treeless Tree Numbers





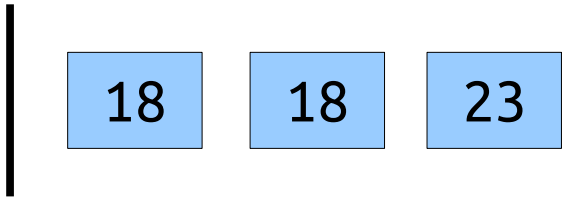
Treeless Tree Numbers

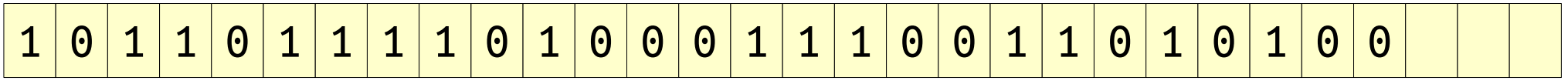
27	18	28	18	28	45	90	45	23	53	60	28	74	71	35



1	0	1	1	0	1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	1	0	1	0				

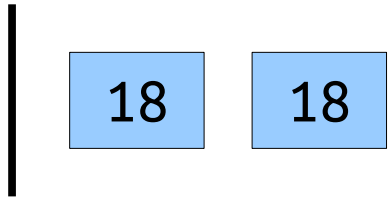
Treeless Tree Numbers





Treeless Tree Numbers

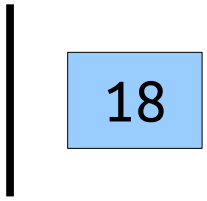
27	18	28	18	28	45	90	45	23	53	60	28	74	71	35



1	0	1	1	0	1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0		

Treeless Tree Numbers

27	18	28	18	28	45	90	45	23	53	60	28	74	71	35



1	0	1	1	0	1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	

Treeless Tree Numbers

27	18	28	18	28	45	90	45	23	53	60	28	74	71	35

1	0	1	1	0	1	1	1	1	0	1	0	0	0	1	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0

Treeless Tree Numbers

27	18	28	18	28	45	90	45	23	53	60	28	74	71	35

(770,238,112)

1 0 1 1 0 1 1 1 1 0 1 0 0 0 1 1 1 0 0 1 1 0 1 0 1 0 0 0 0 0

How can we tell when two blocks can share RMQ structures?

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When they have the same Cartesian tree number!

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When they have the same Cartesian tree number!
And we can check this in time O(b)!

How many block types are there, as a function of b?

How can we tell when two blocks can share RMQ structures?

When they have the same Cartesian tree number!
And we can check this in time O(b)!
And it's easier to store numbers than trees!

How many block types are there, as a function of b?

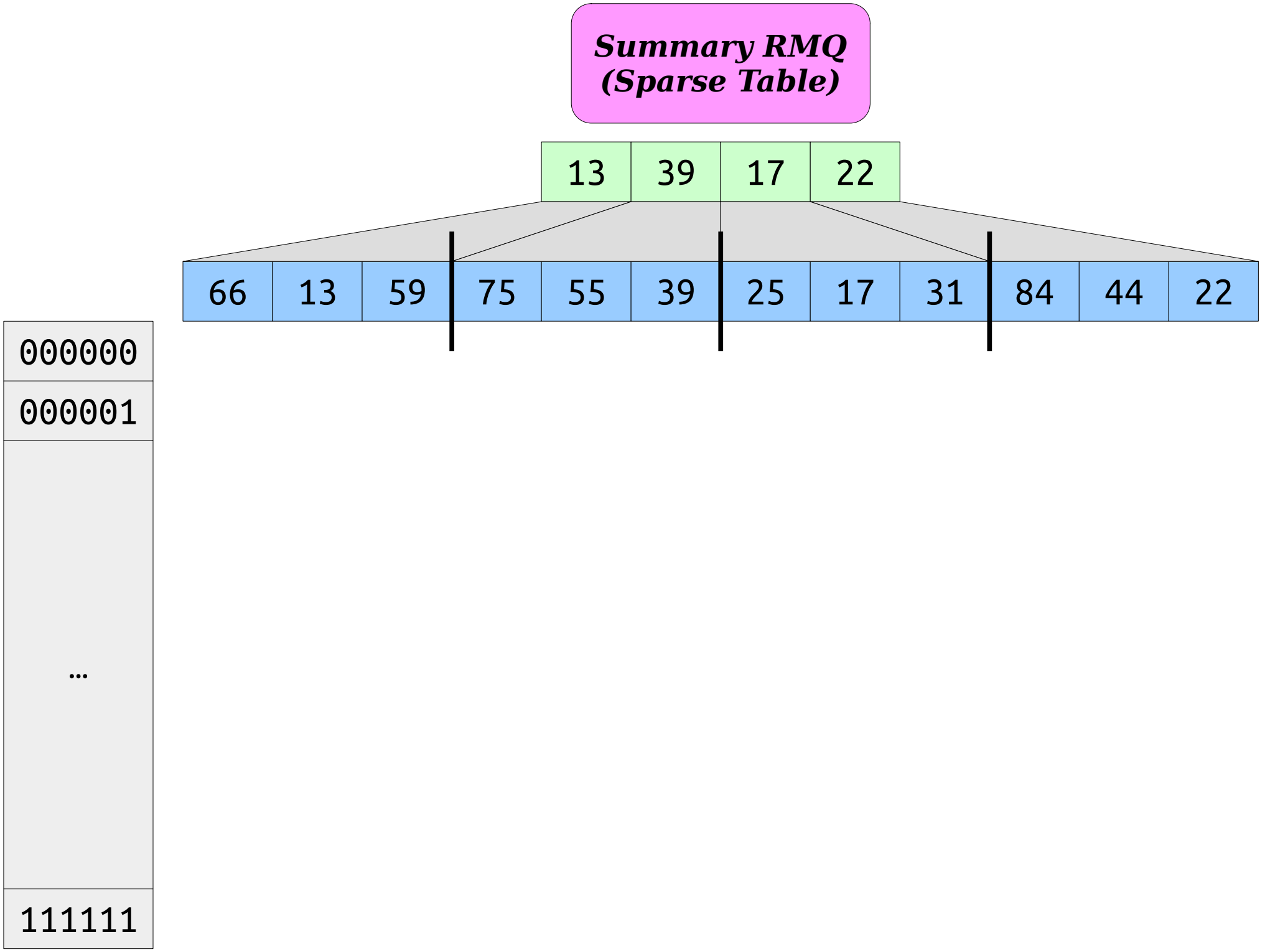
How can we tell when two blocks can share RMQ structures?

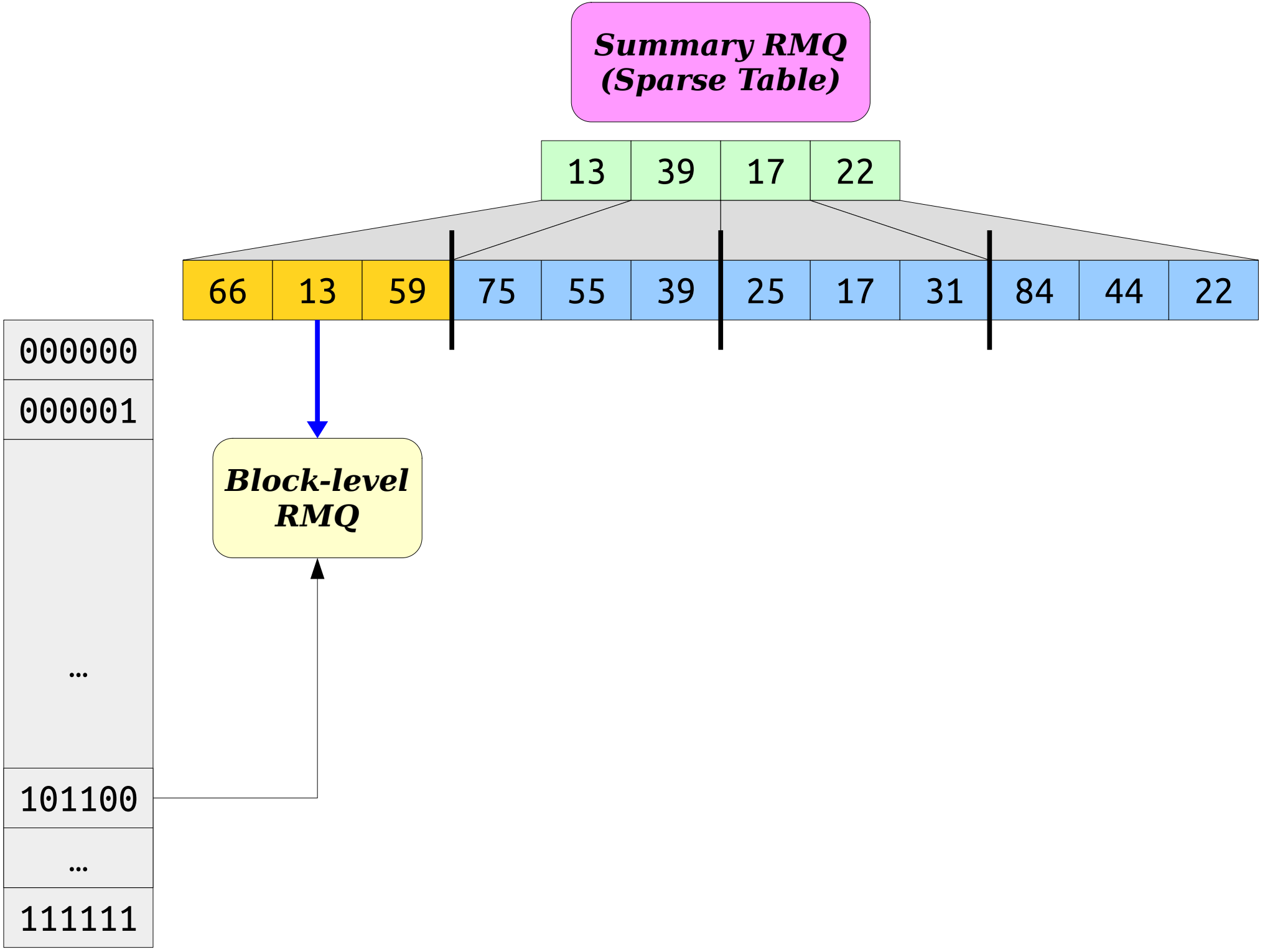
When they have the same Cartesian tree number!
And we can check this in time O(b)!
And it's easier to store numbers than trees!

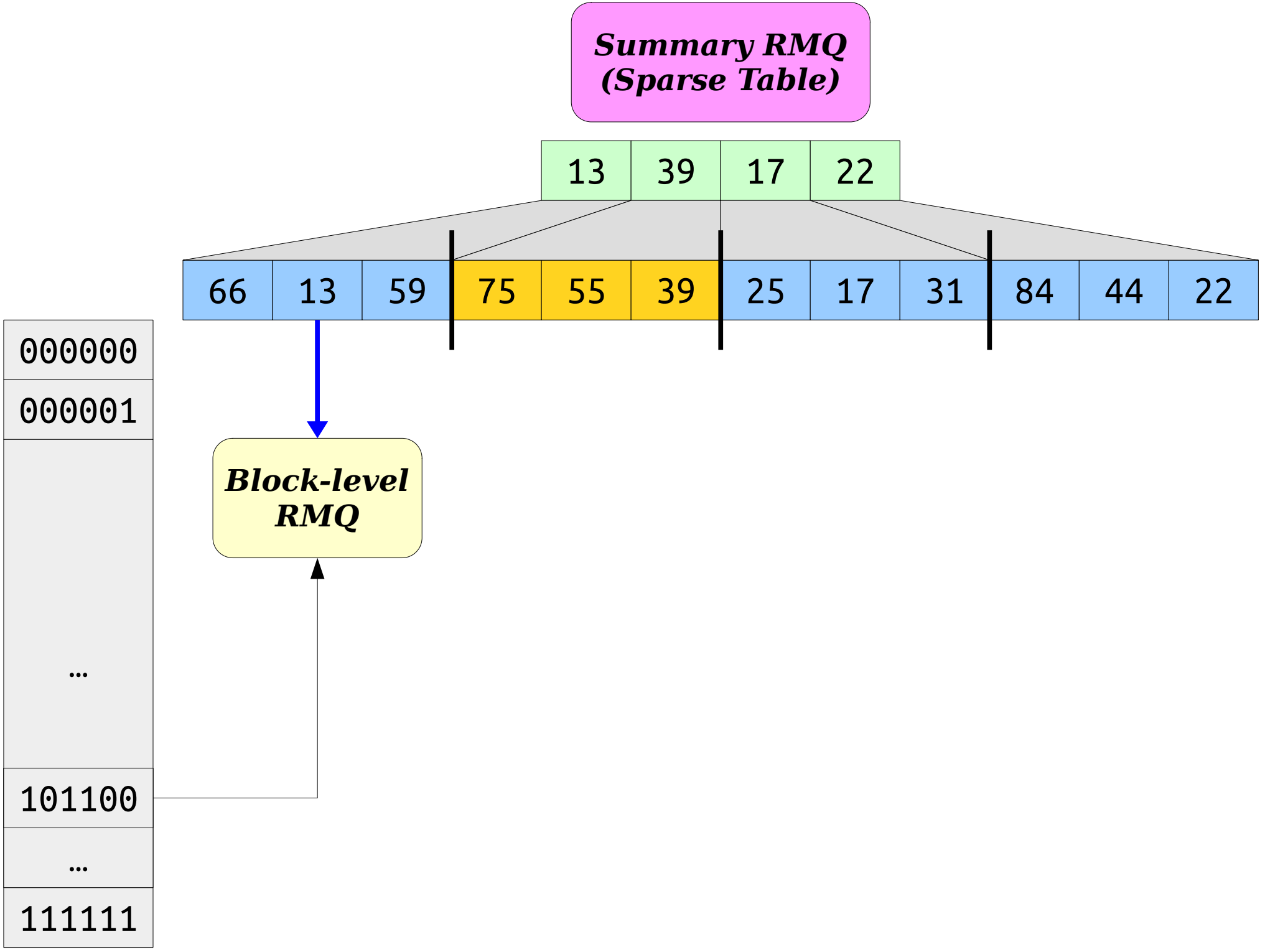
How many block types are there, as a function of b?

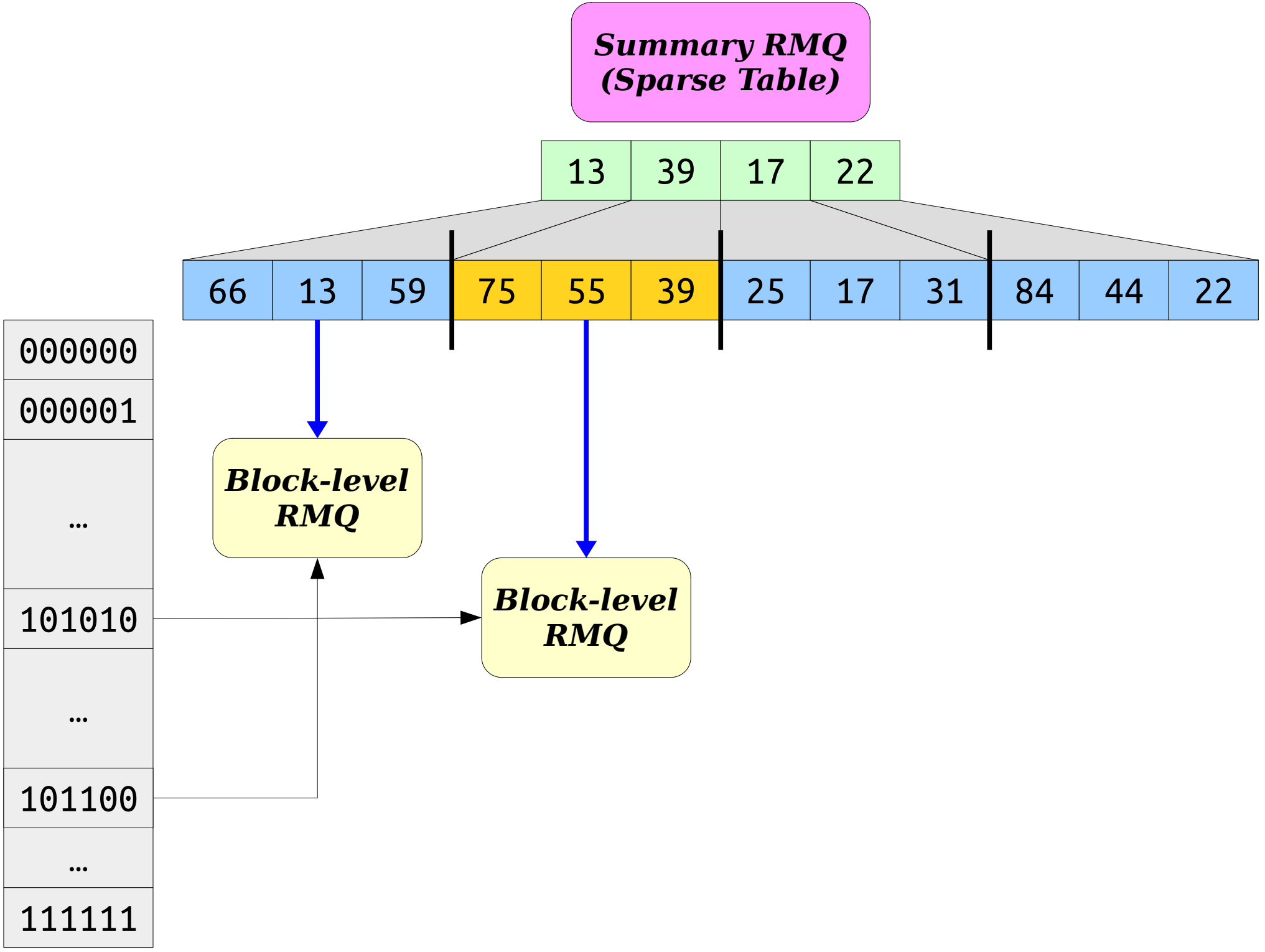
At most 4^b, because of the above algorithm!

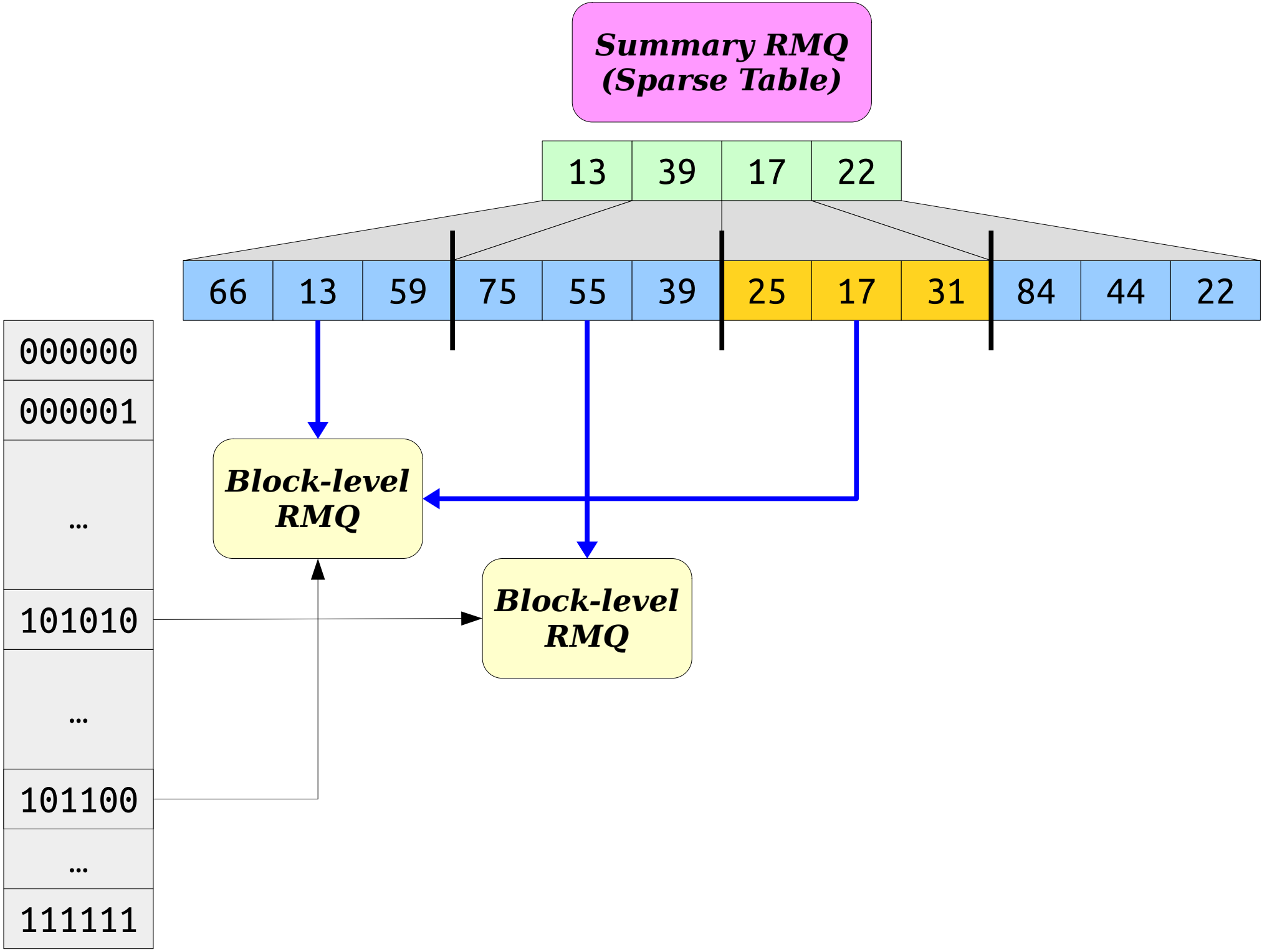
Putting it all Together

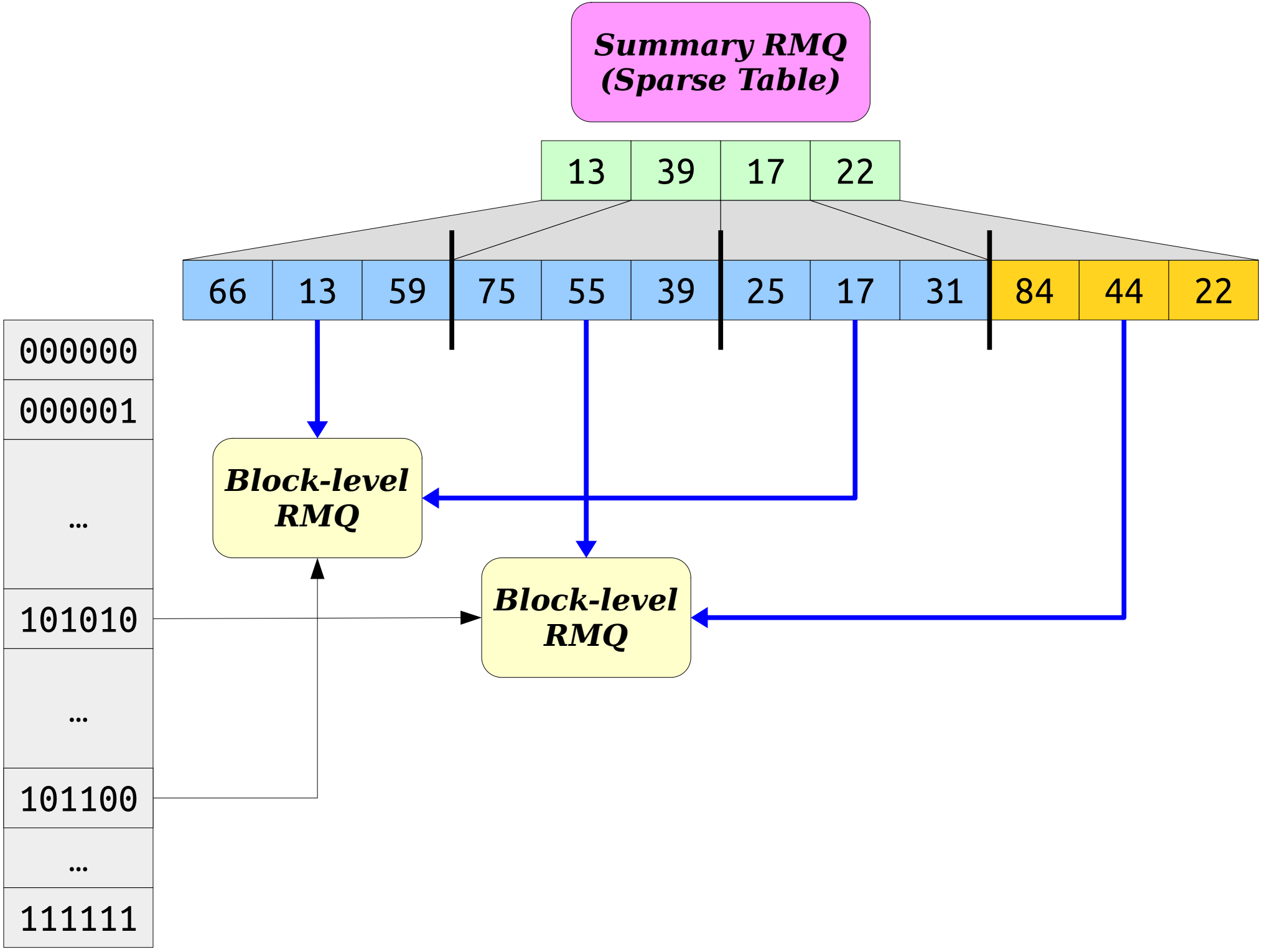

Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
111111

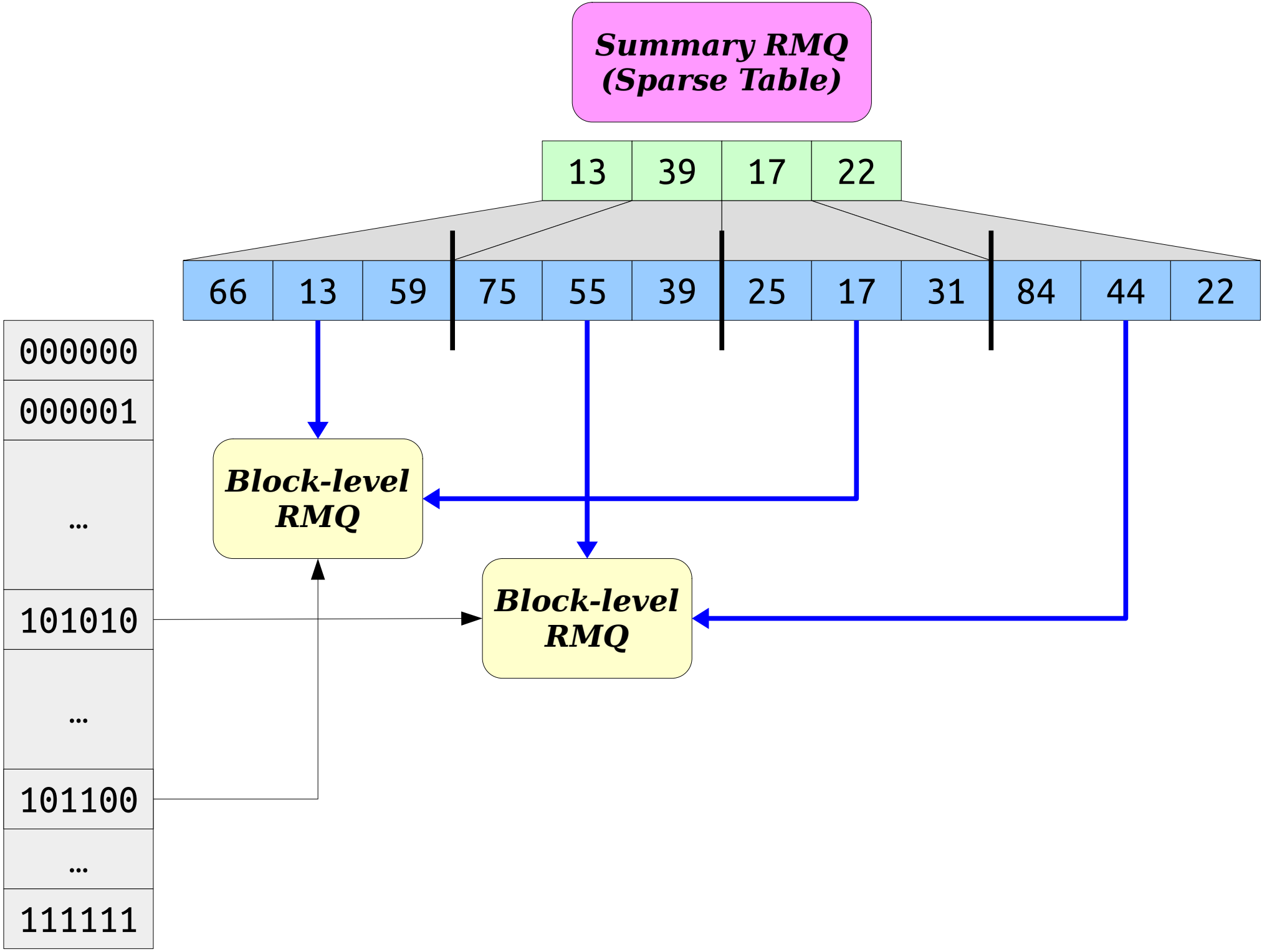

Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101100
...
111111
Block-level
RMQ

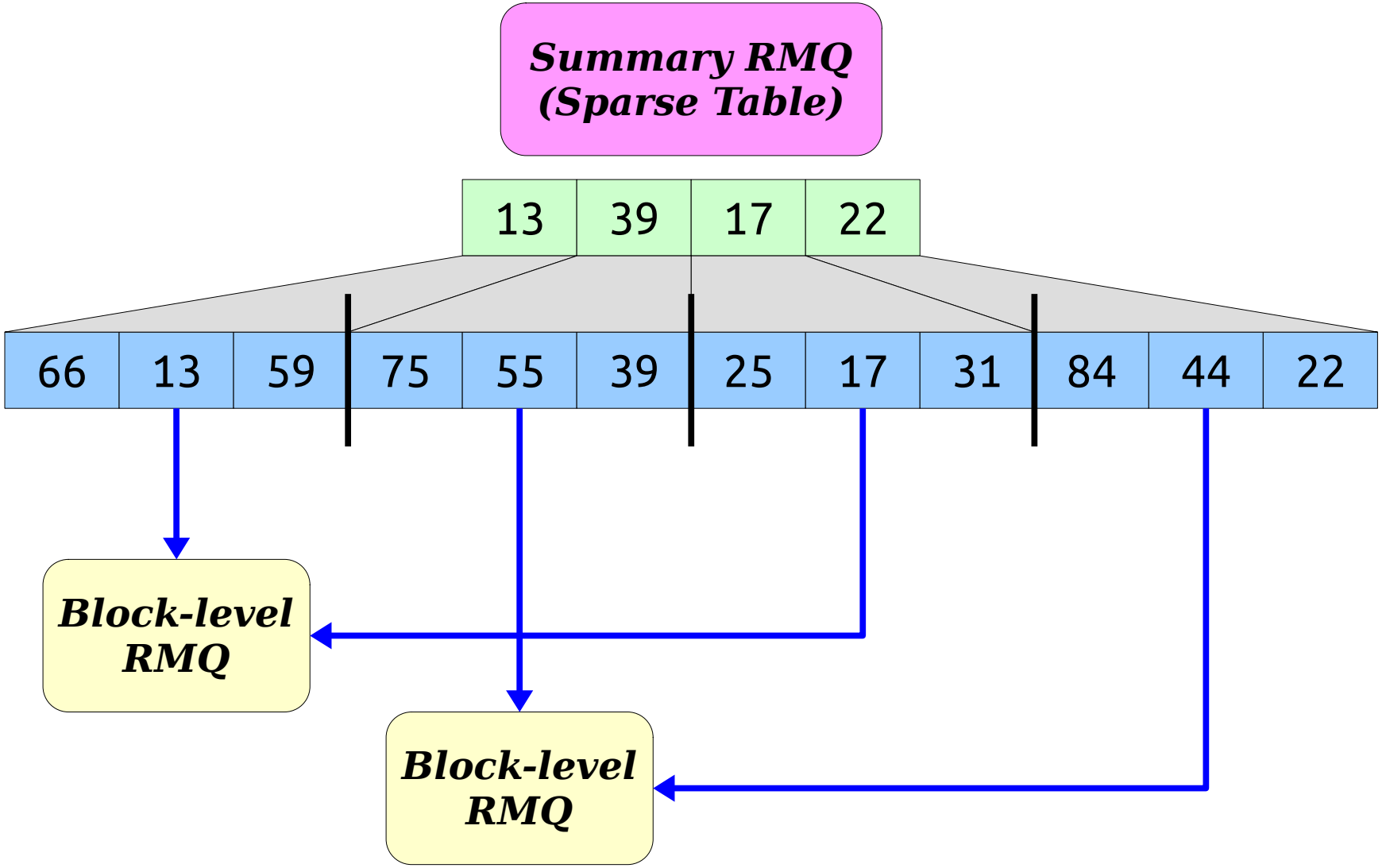

Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101100
...
111111
Block-level
RMQ


Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101010
...
101100
...
111111
Block-level
RMQ
Block-level
RMQ


Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101010
...
101100
...
111111
Block-level
RMQ
Block-level
RMQ


Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101010
...
101100
...
111111
Block-level
RMQ
Block-level
RMQ


Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
000000
000001
...
101010
...
101100
...
111111
Block-level
RMQ
Block-level
RMQ


Summary RMQ
(Sparse Table)
13
39
17
22
66
13
59
75
55
39
25
17
31
84
44
22
Block-level
RMQ
Block-level
RMQ

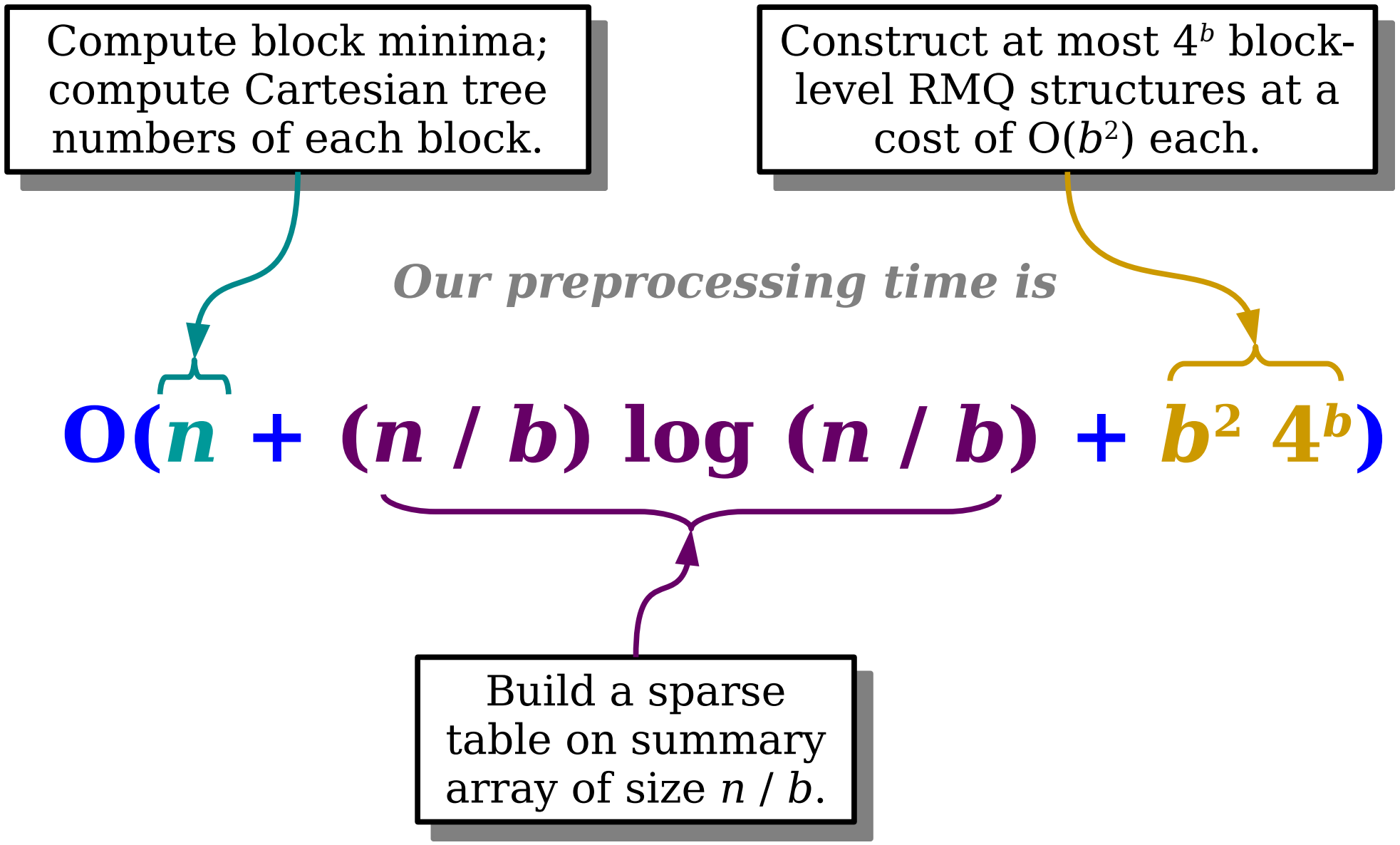
How Efficient is This?

We’re using the hybrid approach, and all the types we’re using have constant query times.

Query time: **O(1)**

Our preprocessing time is

$$O(n + (n / b) \log (n / b) + b^2 4^b)$$


Compute block minima; compute Cartesian tree numbers of each block.
Construct at most 4^b block-level RMQ structures at a cost of $O(b^2)$ each.
Our preprocessing time is
$O(n + (n / b) \log (n / b) + b^2 4^b)$
Build a sparse table on summary array of size n / b.

Our preprocessing time is

$$O(n + (n / b) \log (n / b) + b^2 4^b)$$

This term grows exponentially in n unless we pick $b = O(\log n)$.

Our preprocessing time is

$$O(n + (n / b) \log n + b^2 4^b)$$

This term will be superlinear unless we pick $b = \Omega(\log n)$.

Our preprocessing time is

$$O(n + (n / b) \log n + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + (n / b) \log n + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + (n / k \log_4 n) \log n + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + (n / k \log_4 n) \log n + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + (n / \log n) \log n + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + (n / \cancel{\log n}) \cancel{\log n} + b^2 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + \boldsymbol{n} + b^2\, 4^b)$$

Suppose we pick
$\boldsymbol{b = k \log_4 n}$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2\, 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2\, 4^b)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2\, 4^{k \log_4 n})$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2\, 4^{\log_4 n^k})$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2\, 4^{\log_4 n^k})$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + b^2 n^k)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + (k \log_4 n)^2 n^k)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

$$O(n + n + (k \log_4 n)^2 n^k)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

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

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

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

Suppose we pick
$b = k \log_4 n$
for some constant k.

Our preprocessing time is

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

Suppose we pick
$b = k \log_4 n$
for some constant k.

Now, set $k = ½$.

Our preprocessing time is

$$O(n + n + (\log n)^2 n^{1/2})$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Now, set $k = \frac{1}{2}$.

Our preprocessing time is

$$O(n + n + n)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Now, set $k = ½$.

Our preprocessing time is

$$\mathbf{O}(n)$$

Suppose we pick
$b = k \log_4 n$
for some constant k.

Now, set $k = \frac{1}{2}$.

The Fischer-Heun Structure

- This data structure is called the ***Fischer-Heun structure***. It uses a modified version of our hybrid RMQ framework:
  - Set $b = \frac{1}{2} \log_4 n = \frac{1}{4} \log_2 n$.
  - Split the input into blocks of size b. Compute an array of minimum values from each block.
  - Build a sparse table on that array of minima.
  - Build per-block RMQ structures for each block, using Cartesian tree numbers to avoid recomputing RMQ structures unnecessarily.
  - Make queries using the standard hybrid solution approach.
- This is an **⟨O(*n*), O(1)⟩** solution to RMQ!

The Method of Four Russians

- The technique employed here is an example of the ***Method of Four Russians*** or a ***Four Russians Speedup***.
  - Break the problem of size n into subproblems of size b, plus some top-level problem of size n / b.
    - This is called a ***macro/micro*** decomposition.
  - Solve all possible subproblems of size b.
    - Here, we only solved the subproblems that actually came up in the original array, but that's just an optimization.
  - Solve the overall problem by combining solutions to the micro and macro problems.
- Think of it as "***divide, precompute, and conquer***."
- Curious about the name? It comes from a paper by Арлазаров, Диниц, Кронрод, and Фараджев.

More to Explore

- ***Lowest Common Ancestors***
  - Given a tree, preprocess the tree so that queries of the form "what is the lowest common ancestor of these two nodes?" can be answered as quickly as possible. This reduces to RMQ, and is one of the main places it's used.
- ***Succinct RMQ***
  - Our ⟨O(n), O(1)⟩ solution to RMQ uses only O(n) *words* of memory. How few *bits* of memory are needed? Later work by Fischer and Heun (and others!) has reduced this to $2n + o(n)$ bits, using some very clever techniques.
- ***Durocher's RMQ Structure***
  - A professor teaching a data structures class found a way to solve RMQ in time ⟨O(n), O(1)⟩ using some of the techniques we've seen, but without needing the Four Russians speedup. The paper is very accessible and shows off some really clever techniques.

Why Study RMQ?

- I chose RMQ as our first problem for a few reasons:
  - ***See different approaches to the same problem***. Each approach we covered introduces some generalizable idea that we'll see later in the quarter.
  - ***Build data structures out of other data structures***. Many modern data structures use other data structures as building blocks, and it's very evident here.
  - ***See the Method of Four Russians***. This trick looks like magic the first few times you see it and shows up in lots of places.
  - ***Explore modern data structures***. This is relatively recent data structure (2005), and I wanted to show you that the field is still very active!
- So what's next?

Next Time

- ***Balanced Trees***
  - The perennial data structure workhorse.
- ***B-Trees***
  - A simple, flexible balanced tree.
- ***2-3-4 Trees***
  - Need to code up a balanced tree? Try this one.
- ***A Glimpse of Red/Black Trees***
  - Where did these things come from, anyway?