Parallel Longest Increasing Subsequence and van Emde Boas Trees

Ziyang Men

UC Riverside

Joint work with Yan Gu, Zheqi Shen, Yihan Sun and Zijin Wan

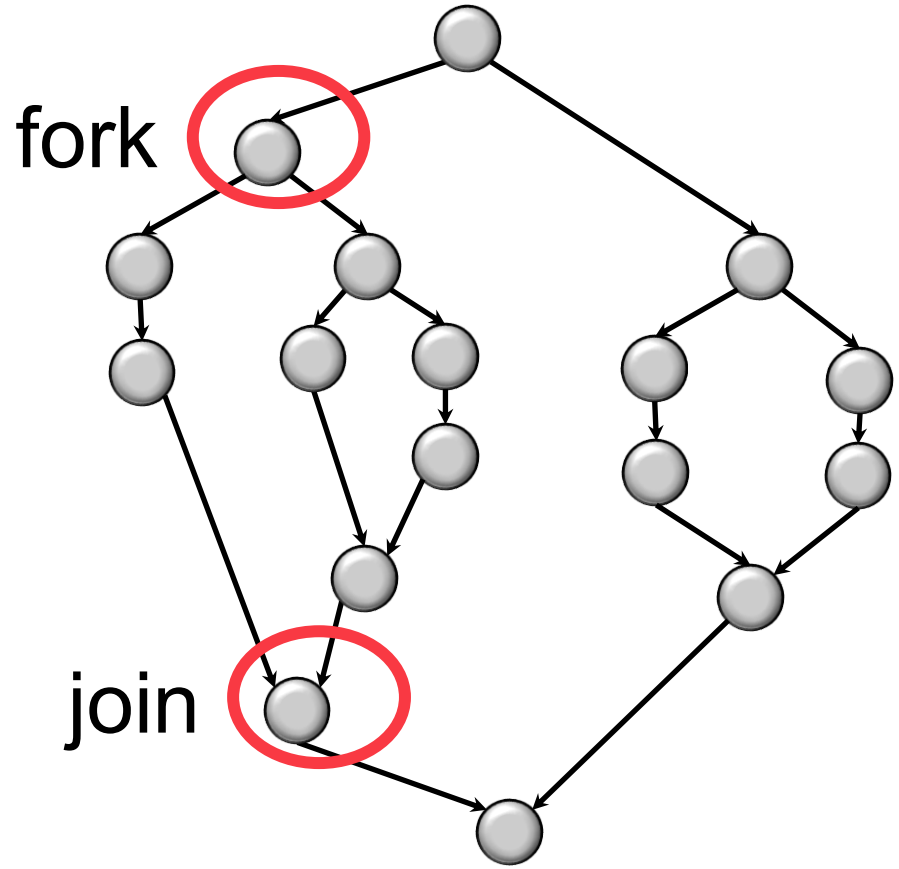
Models and Background

Shared-memory multi-core setting, using fork-join parallelism assuming binary-forking.

Work-span model

- Work: total number of operations (sequential time).
- Span/depth: longest dependence chain (parallel time).

Work-efficient: work asymptotically the same as the best sequential algorithm.



Longest Increasing Subsequence (LIS)

Given a sequence A, the LIS problem is to

- find the **longest subsequence** A^* of A,
- such that the elements in A^* are **strictly increasing**.

Denote the length for A as $\boldsymbol{n}$, and the LIS length as $\boldsymbol{k}$.
Call each element in A an **object**.

Unweighted LIS

Longest Length of LIS ending with the i-th object

$$dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + 1$$

Can be reported by a range query
(e.g., an augmented tree in $O(\log n)$ time)

$O(n \log n)$ sequential time complexity
(can also be $O(n \log k)$)

Weighted LIS

Maximum weight of IS ending with the i-th object

$$dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + \boldsymbol{w(i)}$$

Can be reported by a range query
(e.g., an augmented tree in $O(\log n)$ time)

$\boldsymbol{O(n \log n)}$ sequential time complexity

Existing Results

Not work-efficient and/or highly parallelized	Bound achieved
Galil et al.(Parallel Distrib 1994), Krusche, et al.(PPAM 2009), Nakashima et al. (ISPDC 2002), Semé, Thierry, et al. (ICCSA 2006), Thierry, et. al. (SPAA 2001)	$\Omega(n^{1.5})$ work
Alam and Rahman's algorithm (IPL 2013)	$\Theta(n)$ span

Nearly work-efficient ($\tilde{O}(n)$ work) with non-trivial parallelism	Bound achieved
Krusche and Tiskin's (SPAA 2010)	$O(n \log^2 n)$ work and $\tilde{O}(n^{2/3})$ span
Shen et al. (SPAA'22)	$O(n \log^3 n)$ work and $\tilde{O}(k)$ span
Cao et al. (SPAA'23):	$O(n \log^2 n)$ work and polylogarithmic span

Existing Results

Gap:

- No known work-efficient algorithm with non-trivial span.
  - i.e., $o(n)$ or $O(k)$ for LIS length k.
- No efficient implementations.

Solution:

- Sequential algorithm takes advantage of “range query”.
- The efficiency of the algorithm relies on efficient DATA STRUCTURES!

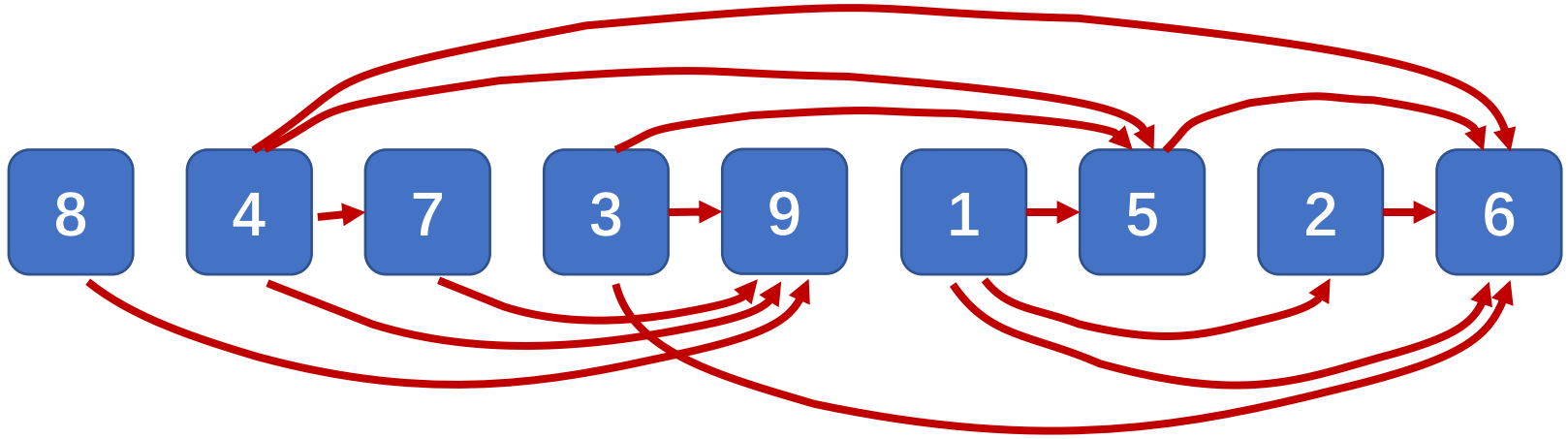
Our Work

LIS	Weighted LIS	
Theory/Practice	Theory	Practice
$O(n \log k)$ work	$O(n \log n \log \log n)$ work	$O(n \log^2 n)$ work
$\tilde{O}(k)$ span	$\tilde{O}(k)$ span	$\tilde{O}(k)$ span
Parallel Tournament tree	Parallel van Emde Boas Tree (vEB tree)	Parallel Range tree
Code Available[1]		Code Available[1]

[1]Code repository: `https://github.com/ucrparlay/Parallel-LIS`

Work-efficient Parallel LIS?

Dependences in LIS

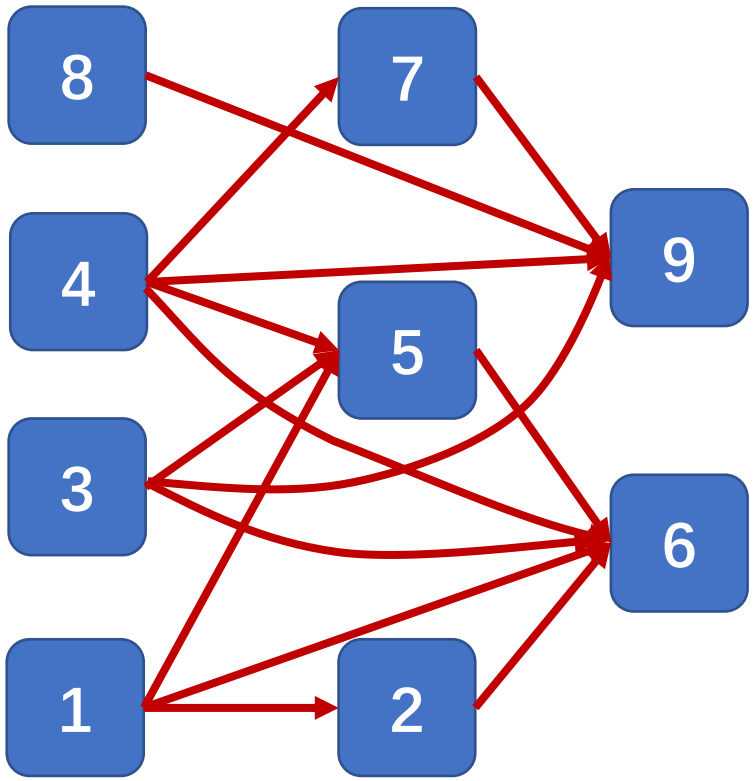


Longest Length of IS ending with the i-th object

$$dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + 1$$

- **Processing an object $A[i]$ depends only on all objects $A[j] < A[i]$ with $j < i$.**

Dependences in LIS



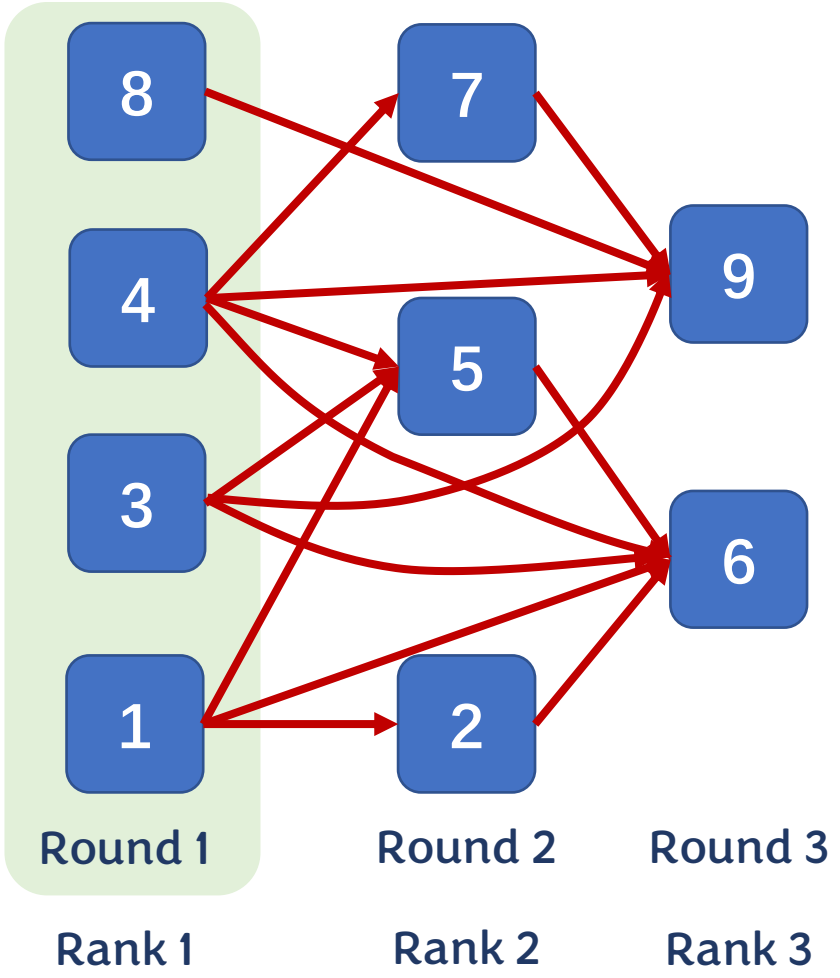
- Processing an object x depends **only** on all objects $y < x$ before it.

- The dependence can be shallower than $\Theta(n)$!
- The "Dependence Graph" (DG).

- The actual dependence chain length is $O(k)$ for LIS length k.
- For random sequences, the expected LIS length is $O(\sqrt{n})$.
- Good potential for parallelism!

Phase-Parallel Algorithm [Shen et al. SPAA 22]

- Define "rank" for each input object as the LIS length ending at this object.



Phase-Parallel Algorithm

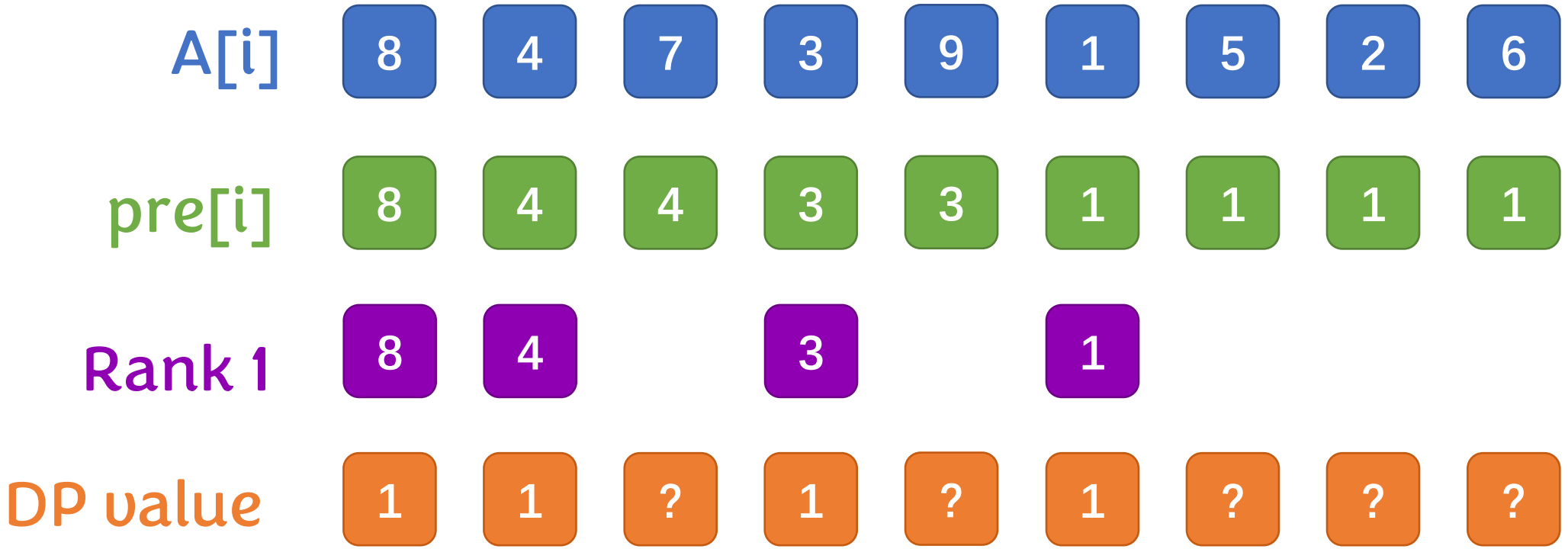
- In round r, we can process all rank r objects.
- Then in round $r+1$, all rank $(r+1)$ objects are ready!

How to efficiently identify all objects with rank r?

Key for work-efficiency!

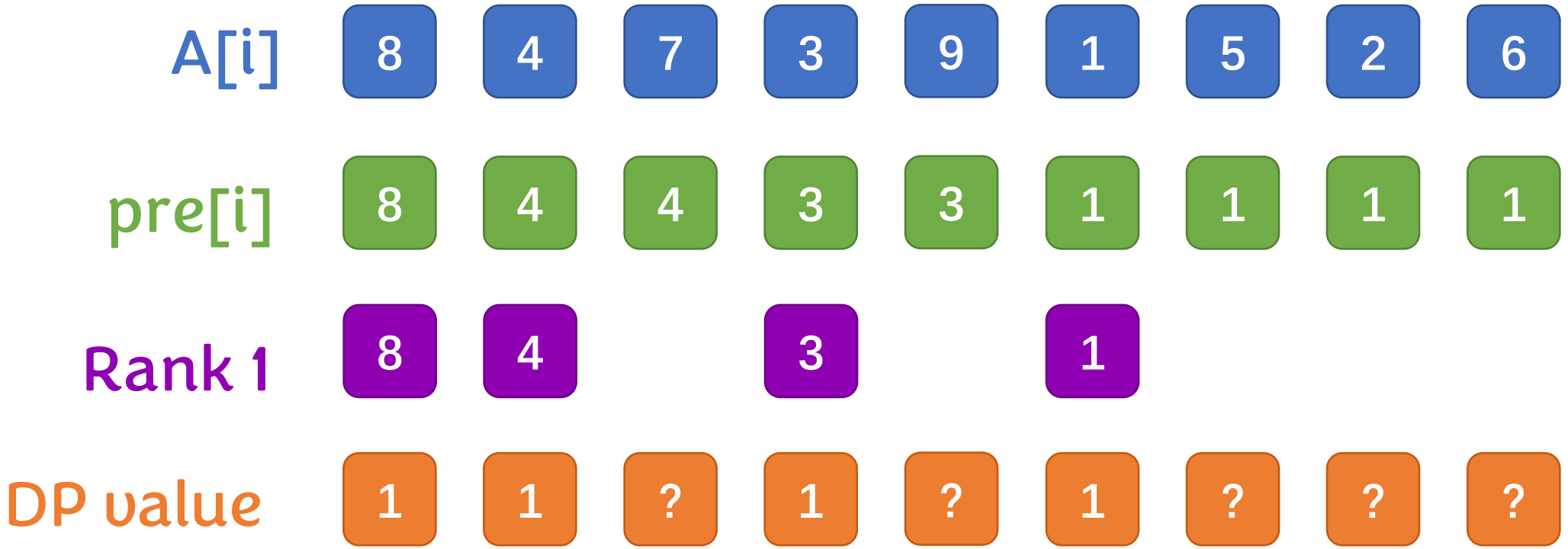
Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]` and `pre[i]`.
- Observation 1: an object `A[i]` has rank 1 iff `A[i]=pre[i]` - "prefix-min" objects.



Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]` and `pre[i]`.
- Observation 2: an object has rank r iff `A[i]=pre[i]` after removing all objects with rank $< r$.



Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]`and `pre[i]`.
- Observation 2: After removing all rank $< r$ objects, then an object has rank r if and only if `A[i]=pre[i]`.

Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]`and `pre[i]`.
- Observation 2: After removing all rank $< r$ objects, then an object has rank r if and only if `A[i]=pre[i]`.

Identify all objects with rank r

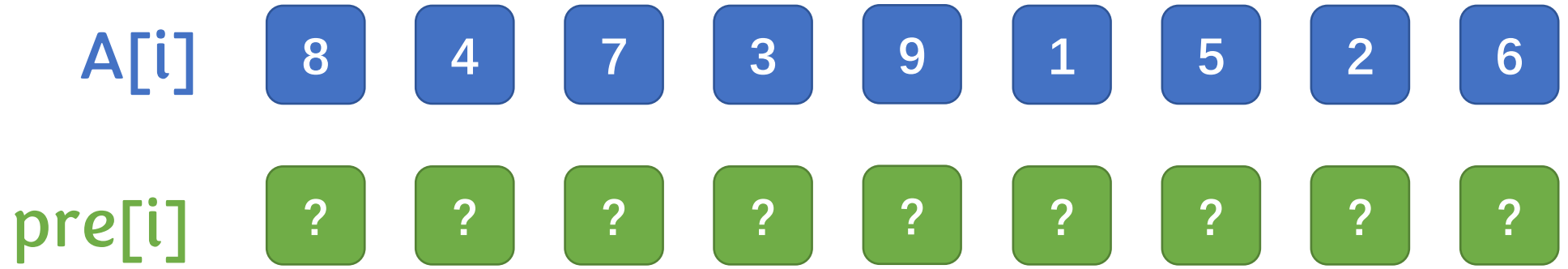
- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]`and `pre[i]`.
- Observation 2: After removing all rank $< r$ objects, then an object has rank r if and only if `A[i]=pre[i]`.

Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]`and `pre[i]`.
- Observation 2: After removing all rank $< r$ objects, then an object has rank r if and only if `A[i]=pre[i]`.

Identify all objects with rank r

- Given an index i, let `pre[i]` be the *prefix-min* of the input array up to the i-th object.
- The trick lies between `A[i]`and `pre[i]`.
- Observation 2: After removing all rank $< r$ objects, then an object has rank r if and only if `A[i]=pre[i]`.

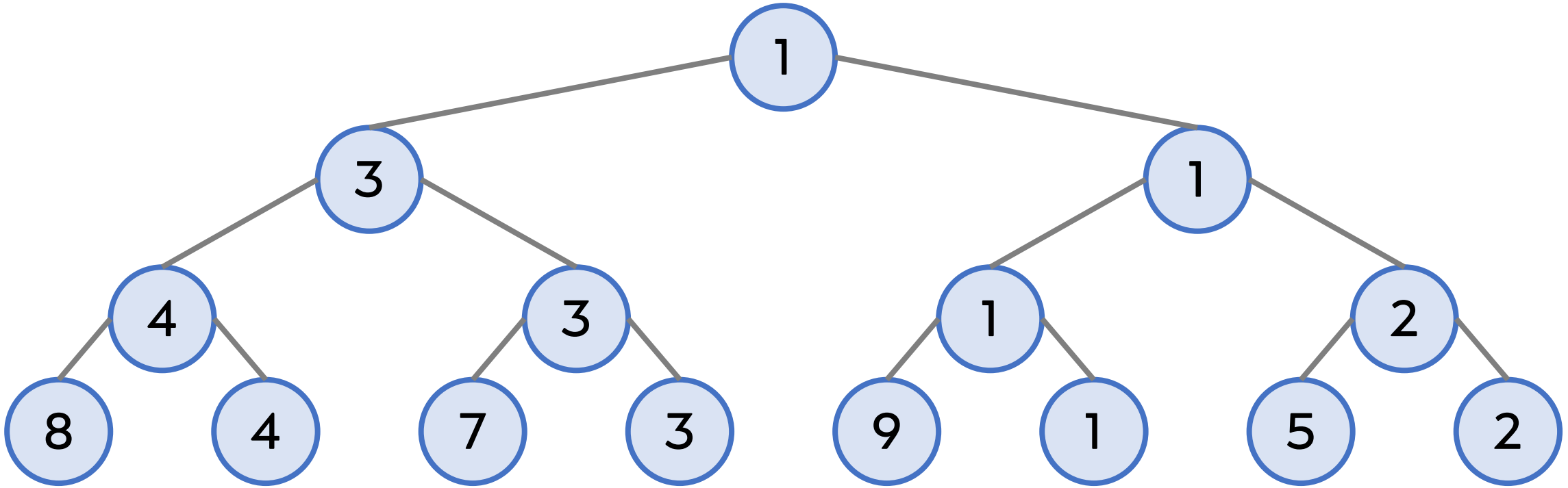


How to find all prefix-min objects?

Find all prefix-min objects

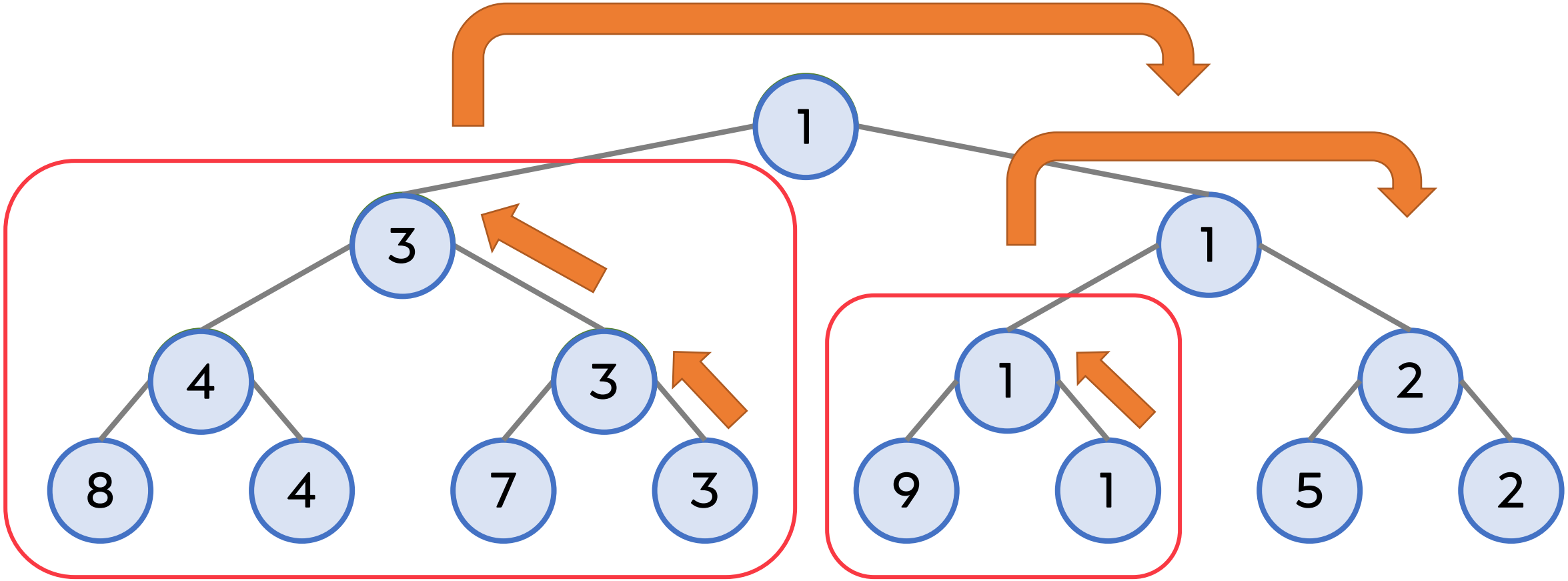
Tournament tree:

- **Leaves are input objects $A[i]$.**
- **Internal nodes are minimum of the two children.**
- **Support divide-and-conquer (parallelism) as a tree.**



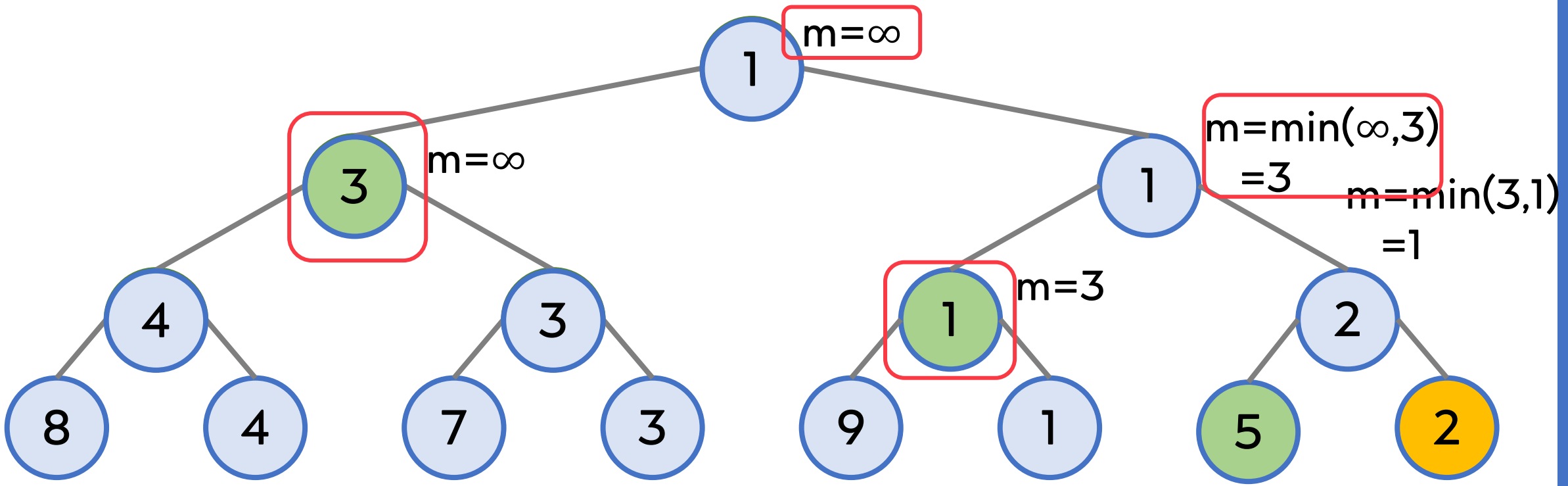
Find all prefix-min objects

- An object is a prefix-min object if its value equals to its prefix-min.



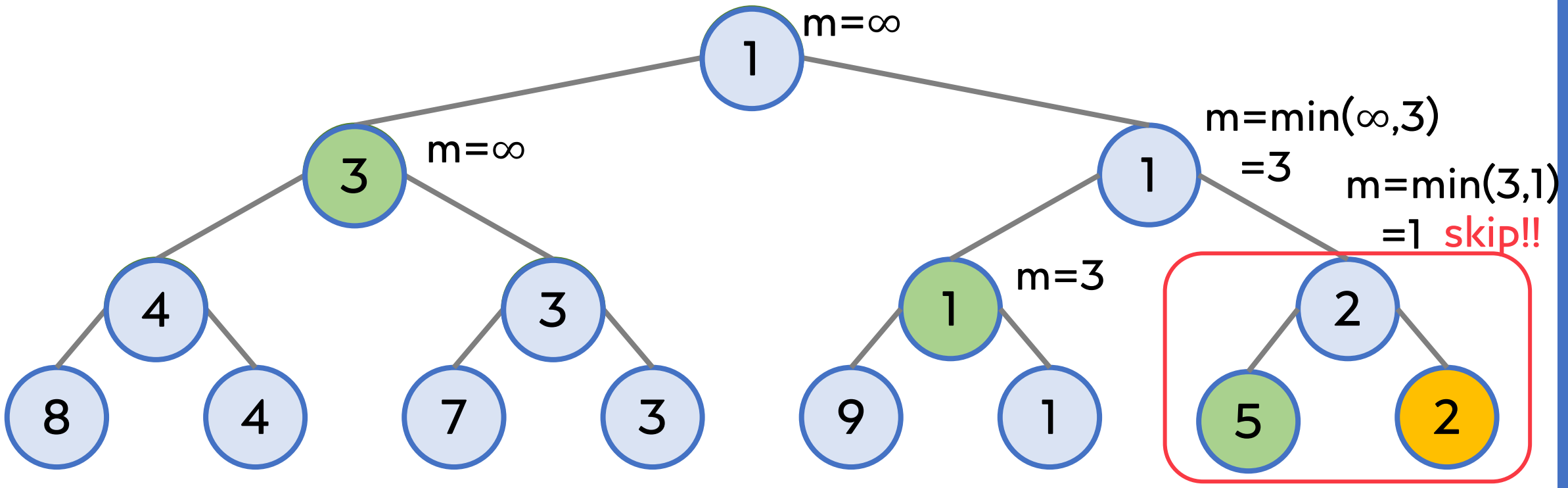
Find all prefix-min objects

- An object is a prefix-min object if its value equals to its prefix-min.
- Pass the *m-value* to each branch, which is the smallest object to the left of one subtree.
  - Left branch: the same as its parent,
  - Right branch: min(m of parent, min from left branch).



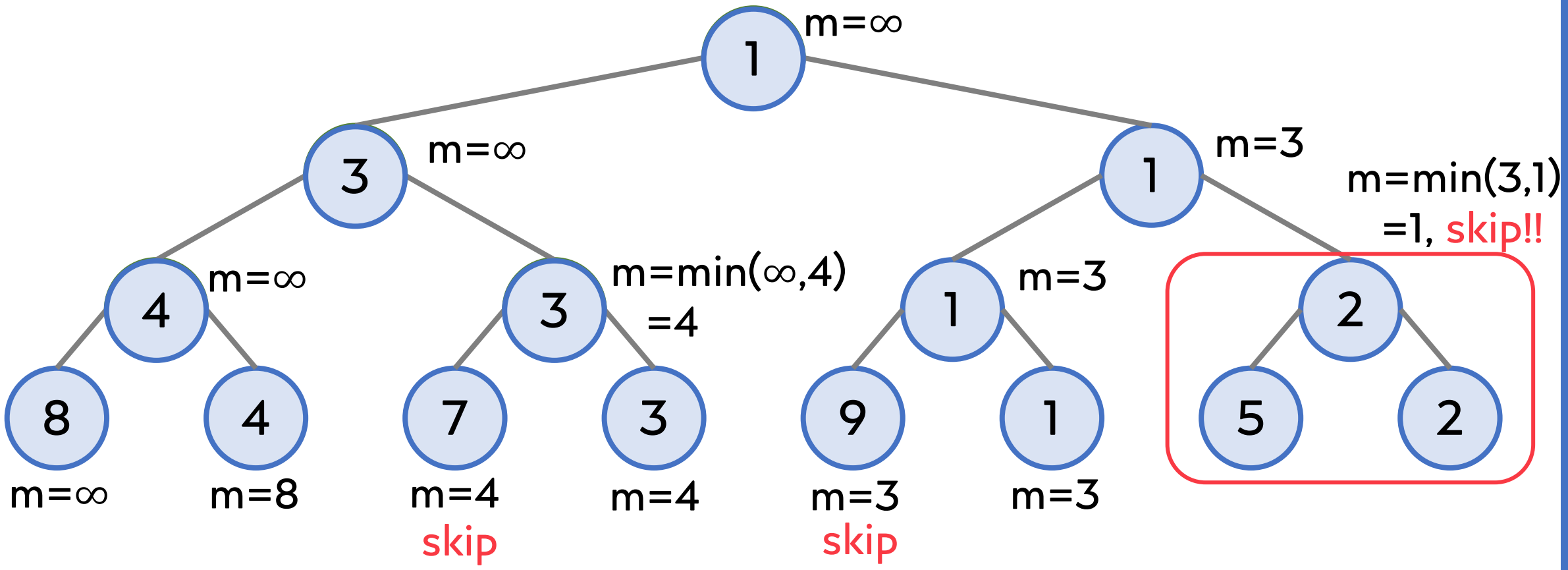
Find all prefix-min objects

- An object is a prefix-min object if its value equals to its prefix-min.
- Pass the *m-value* to each branch, which is the smallest object to the left of one subtree.
- m < the current node value, skip!



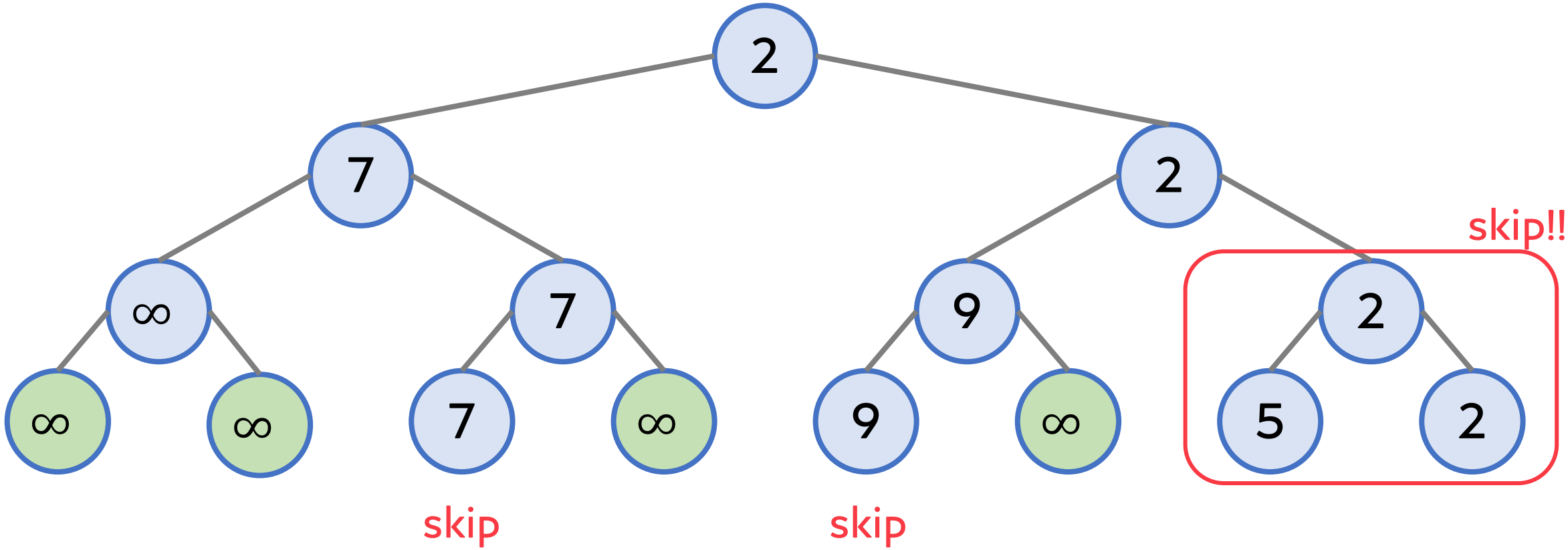
Find all prefix-min objects

- Traverse two branches in parallel
- The prefix-min objects are those not skipped in the leave.
- Remove all objects identified as “prefix-min” objects (mark to ∞).



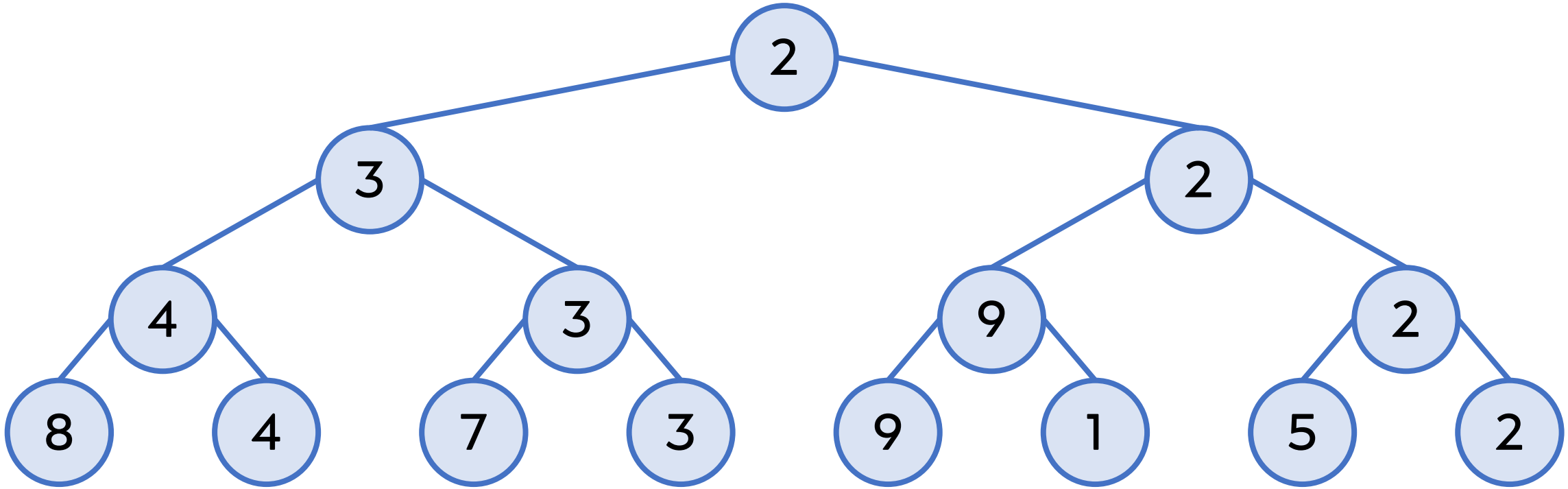
Find all prefix-min objects

- Traverse two branches in parallel.
- The prefix-min objects are those not skipped in the leave.
- Remove all objects identified as “prefix-min” objects (mark to ∞).
- Update the tree and continue.



Find all prefix-min objects

- No need to remove and re-compute all prefix-min values!
- $O\left(t \log \frac{n}{t}\right)$ for reporting t objects.

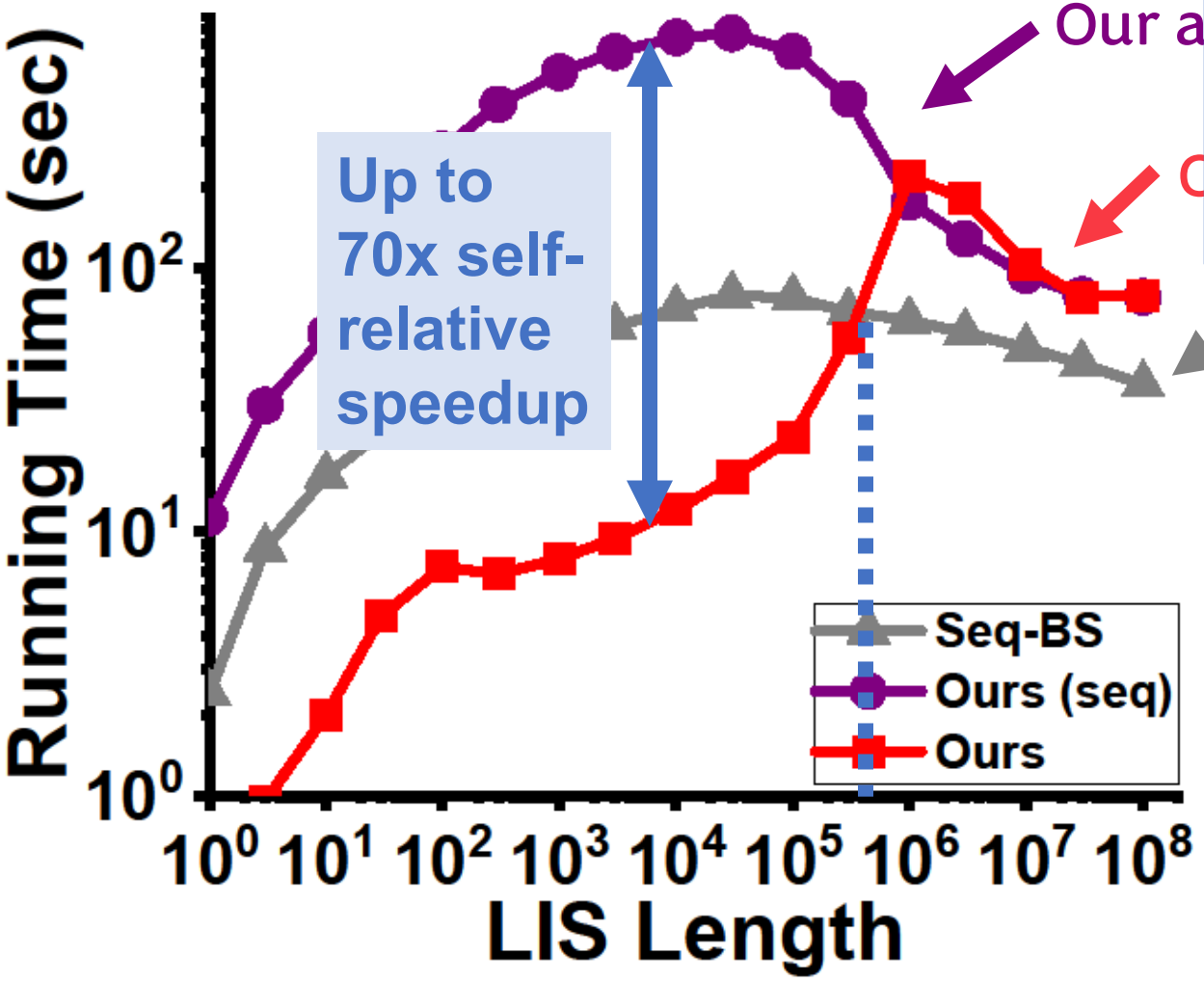


Phase-parallel LIS algorithm

- **Build an initial tournament tree T with all input objects $A[i]$**
- **$r = 0$**
- **While (objects remain) {**
  - **$r = r + 1$**
  - Find all prefix-min objects in T as set S
  - Let the dp values of all objects in S be r
  - Remove all objects in S from T
- **}**
- **Report r**

- $O(n \log k)$ work
- $O(k \log n)$ span

$n = 10^9$, varying k, LIS Running time. Lower is better.



Tested on a 96-core machine. Compared against highly-optimized sequential algorithm [Knuth73] with $O(n \log k)$ work.

Sequential $O(n \log k)$ algorithm

Our algorithm:

- Outperforms highly-optimized sequential algorithm for reasonably large value of k (< 1M) .
- High self-relative speedup – good scalability, promising on even larger data/more cores.

How about Weighted LIS?

Weighted LIS

8	4	7	3	9	1	5	2	6

Maximum weight of IS ending with the i-th object

$$dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + \boldsymbol{w(i)}$$

Can be reported by a range query
(e.g., an augmented tree in $O(\log n)$ time)

$O(n \log n)$ sequential time complexity

Parallel LIS

- $r = 1$
- Build an initial winning tree T with all input values $A[i]$
- While (objects remain) {
  - $r = r + 1$
  - Find all prefix-min objects in T as set S
  - Let the dp values of all objects in S be r
  - Remove all objects in S from T
- }
- Report r

$$\max_{j<i,\, A_j<A_i} dp[j] + w(i)$$

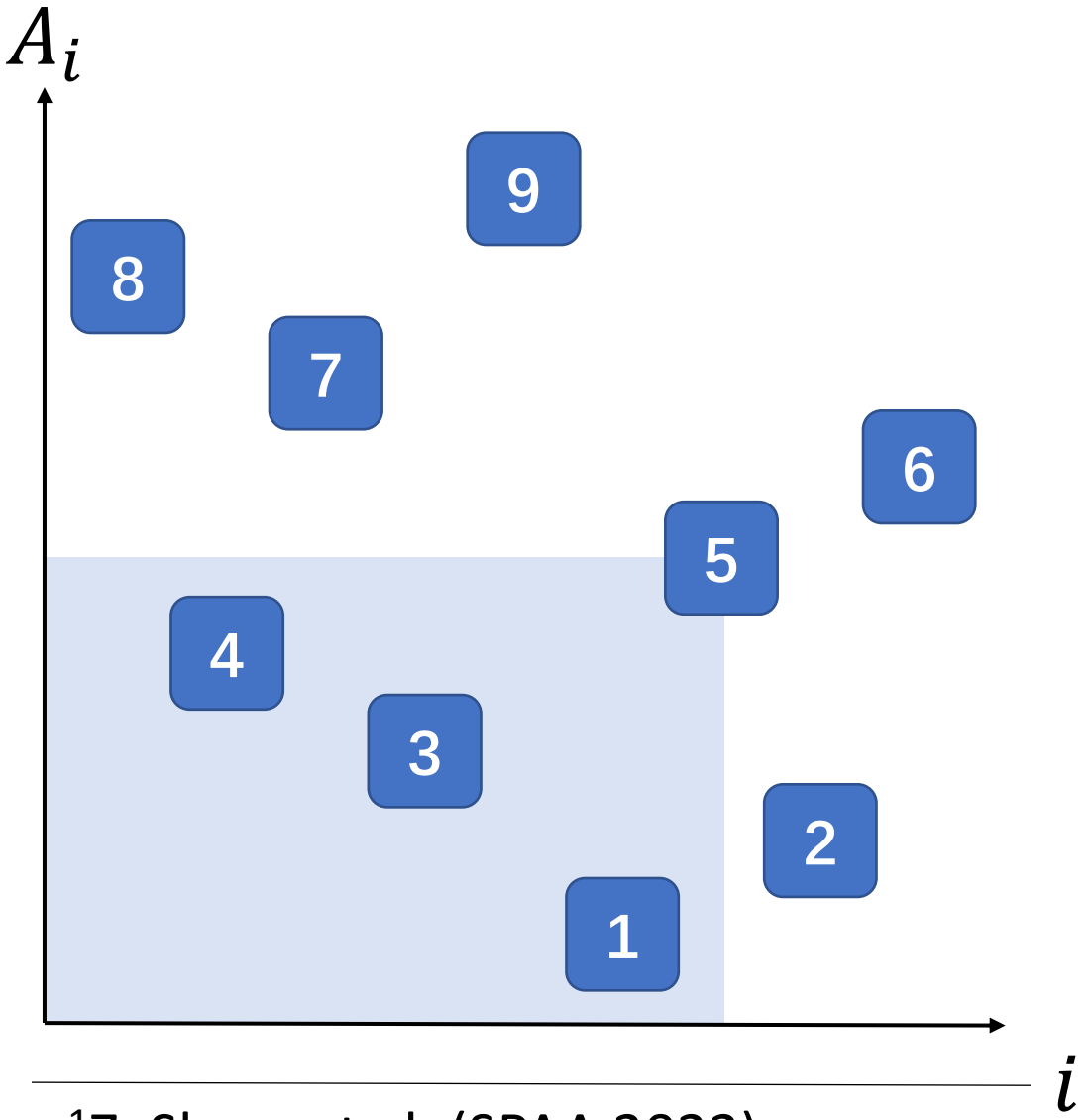
Not the case for weighted setting,
need additional technique to handle the weight!

Parallel Weighted LIS (framework)

- Run ***Parallel-LIS*** to compute the rank for all objects
- Sort and group objects by ranks
- for (rank r from 1 to k) {
  - for (every object with rank r){
    - $dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + w(i)$
  - }
- }

Find data structure supports 2D range-max query!

2D range-max query[1]



[1]Z. Shen, et al. (SPAA 2022)

$$dp[i] = \max_{j<i,\, A_j<A_i} dp[j] + \boldsymbol{w}[\boldsymbol{i}]$$

Find the max dp value in its lower-left area!

The range-max query can be performed by a 2D range tree in $O(\log^2 n)$.

[Y. Sun, et al. (PPoPP 2018)]

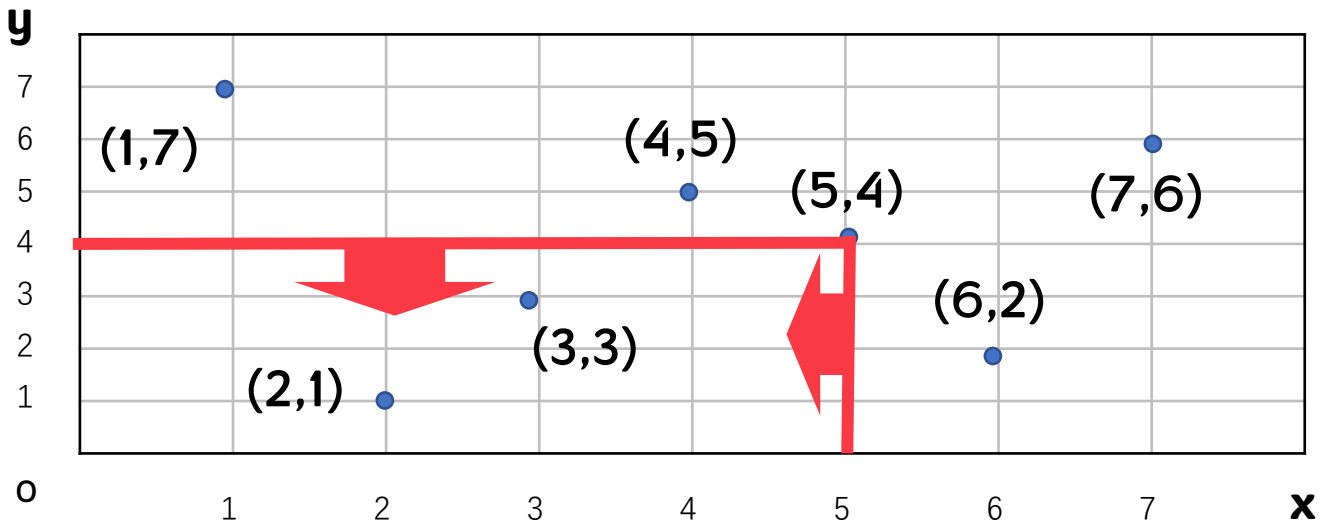
$\Rightarrow$ A parallel weighted LIS algorithm with $O(n \log^2 n)$ work and $\widetilde{O}(k)$ span.

Better work?

2D Range Tree

Outer tree:
order points by their y-coordinates.

Each node maintains an inner tree:
The same set of points in its subtree, but ordered by x-coordinates.

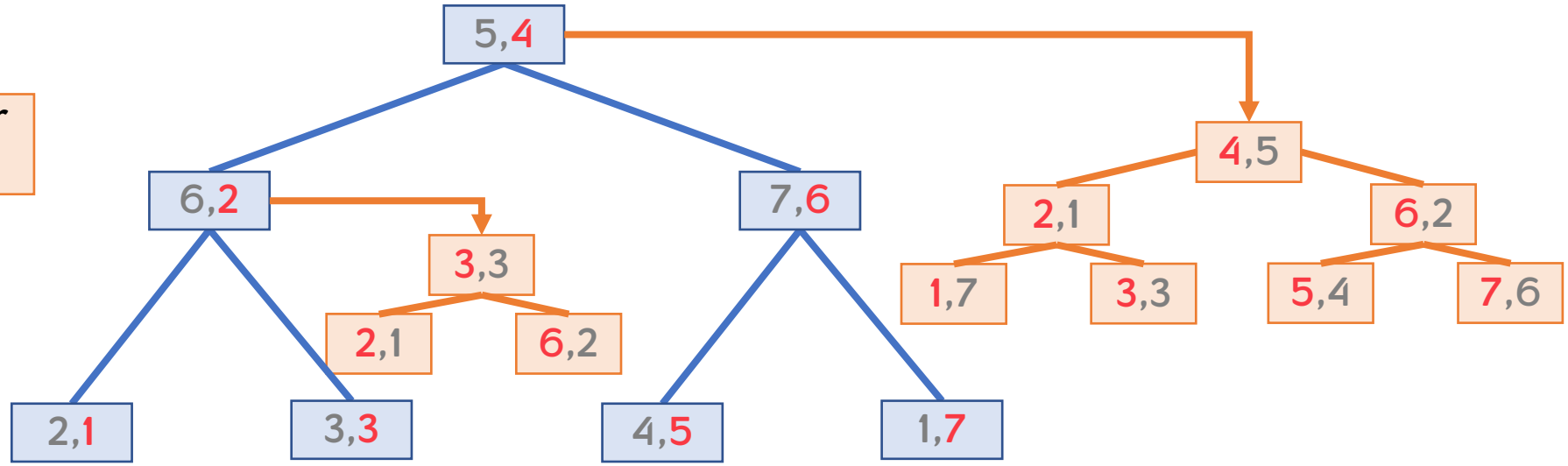


Outer tree Point (x,y)

Inner tree

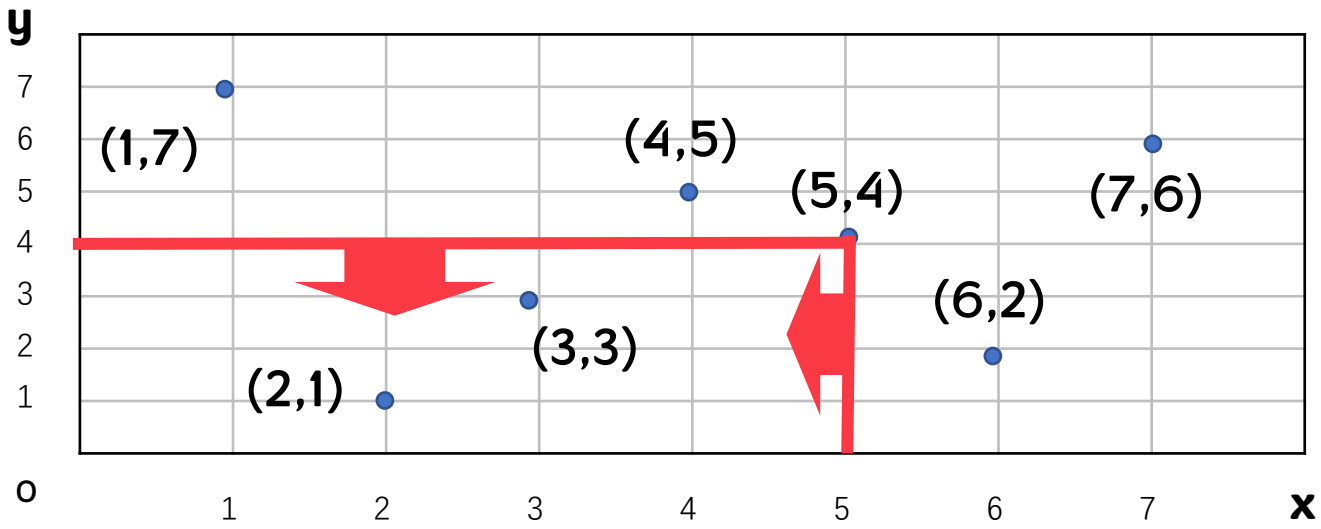
Inner tree point (x,y)

(associated with a dp value)



2D Range Tree

Outer tree:
ordered points by their y-coordinates.



Each node maintains an inner tree:
The same set of points in its subtree, but ordered by x-coordinates
using a van Emde Boas (vEB) tree!

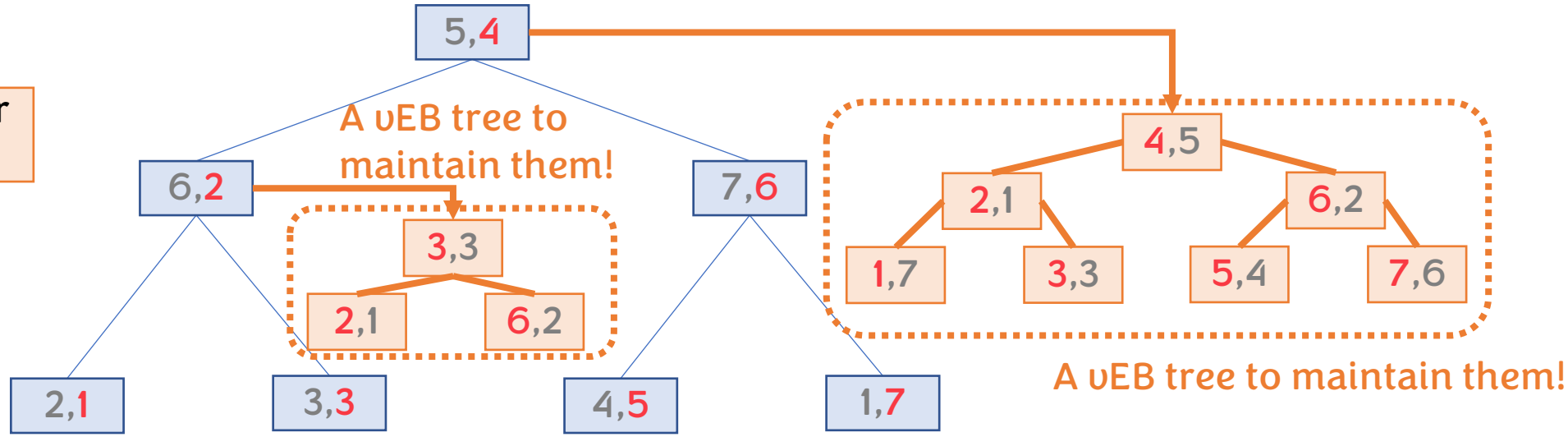
Range-vEB tree!

Outer tree Point (x,y)

Inner tree

Inner tree point (x,y)

(associated with a dp value)



Parallel Weighted LIS

- Run ***Parallel-LIS*** to compute the rank for all objects
- Sort and group objects by ranks
- **Initialize the Range-vEB tree $\mathcal{R}$**
- for (rank r from 1 to k) {
  - for (every object with rank r){
    - **Compute dp value using $\mathcal{R}$**
  - }
  - **Add updated dp values to $\mathcal{R}$**
- }

Parallel van Emde Boas Tree?

The van Emde Boas tree

A van Emde Boas (vEB) tree V is a search tree with integer keys in range $[0, U)$.

Advantage:

- single element operation in $O(\log \log U)$ work, including *insertion*, *deletion*, *predecessor*, *successor*, *min* and *max*.

Challenge:

- Most of range-related queries heavily rely on repeatedly calling *predecessor* or *successor.* Inherently sequential.
- Parallel updates are also sophisticated and no parallel design so far.

Our work: design work-efficient parallel interfaces for vEB tree: *Batch-Insertion*, *Batch-Deletion* and *Range Query*.

Parallel vEB tree

	Range Query	Batch-Insertion	Batch-Deletion
Interface	Report all keys within a range	Insert a sorted batch B	Delete a sorted batch B
Difficulty	Inherently sequential in vEB tree	Relative straightforward	After deletion of $V.\min/V.\max$, find and delete the substitution from subtree
Solution	• Divide-and-conquer • Amortization techniques	• Filter and Partition new high-bits/low-bits • Insert in parallel	• Survival Mapping • Delete the substitution in sequential
Complexity	Work-efficient and poly-logarithmic span		

Parallel Weighted LIS

- Run ***Parallel-LIS*** to compute the rank for all objects
- Sort and group objects by ranks
- **Initialize the Range-vEB tree $\mathcal{R}$**
- for (rank r from 1 to k) {
  - for (every object with rank r){
    - **Compute dp value using $\mathcal{R}$**
  - }
  - **Add updated dp values to $\mathcal{R}$**
- }

- $O(n \log n \operatorname{loglog} n)$ work
- $O(k \log^2 n)$ span

Summary

LIS	Weighted LIS	
Theory/Practice	Theory	Practice
$O(n \log k)$ work	$O(n \log n \log \log n)$ work	$O(n \log^2 n)$ work
$\tilde{O}(k)$ span	$\tilde{O}(k)$ span	$\tilde{O}(k)$ span
Parallel Tournament tree	**Parallel van Emde Boas Tree**	**Parallel Range tree**
Code Available	Work-efficient interfaces	Code Available

Full version paper: `https://arxiv.org/abs/2208.09809`
Code repository: `https://github.com/ucrparlay/Parallel-LIS`