Sorting in database

Guest lecture
CS564 - UW Madison

Thanh Do (Google Inc.)

About me

First boss at UW

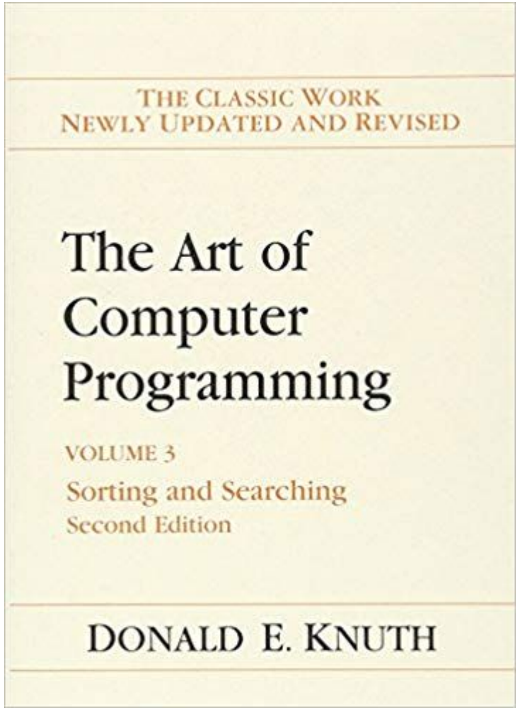
Second boss at
Microsoft GSL

Current boss at
Google

Why are we learning sorting in database?

Why are we learning sorting in database?

*"Computer manufacturers of the **1960's** estimated that **more than 25 percent** of the running time of their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for **more than half of the computing time**. From these statistics we may conclude that either*
1. There are many important applications of sorting, or
2. Many people sort when they shouldn't, or
3. Inefficient sorting algorithms have been in common use."



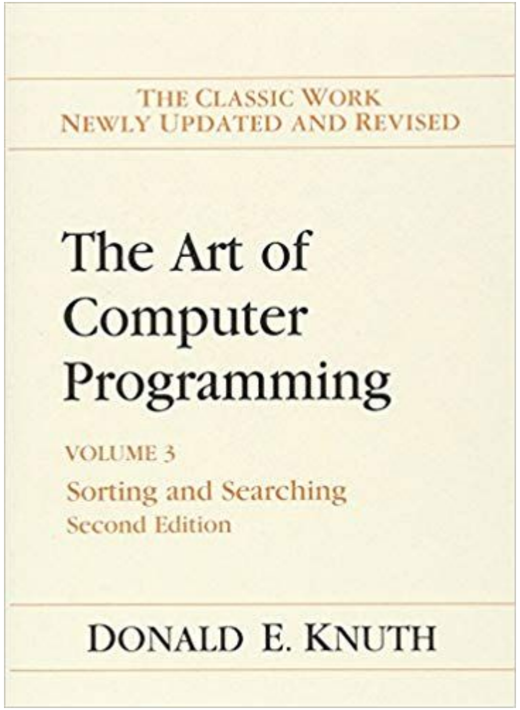
Why are we learning on sorting?

*"Computer manufacturers of the **1960's** estimated that **more than 25 percent** of the running time of their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for **more than half of the computing time**. From these statistics we may conclude that either*

1. There are many important applications of sorting, *or*

~~*2. Many people sort when they shouldn't, or*~~

3. Inefficient sorting algorithms have been in common use."



Agenda

- Use-cases of sorting in data processing
- In-memory sort - run generation
- External merge sort
- Parallel sort

The sorting problem: sort key is a single integer

Unsorted input	Sorted output
918	170
170	275
897	563
275	897
563	918

The sorting problem

Unsorted input

918, CA, 90245, Smith
170, CA, 90345, Jane
897, WI, 53713, Will
275, WI, 53705, Kate
563, CA, 90245, Andy
990, CA, 90001, Jane

Sorted output by name, zip code, phone number

563, CA, 90245, Andy
990, CA, 90001, Jane
170, CA, 90345, Jane
275, WI, 53705, Kate
918, CA, 90245, Smith
897, WI, 53713, Will

Sorted output by zip code, name, phone number

275, WI, 53705, Kate
897, WI, 53713, Will
990, CA, 90001, Jane
918, CA, 90245, Andy
918, CA, 90245, Smith
170, CA, 90345, Jane

Sort keys are composite, depending on what you want to slice

Use-cases of sorting

- **Index creation**
  More efficient to sort the input first, then perform bulk loading to create b-tree

- **Searching**
  If data is sorted, binary search is efficient
  In typical DBMS, tables are sorted by PK for fast look up

- **Database operations**
  “order by”, “distinct”, “group by”, top/limit, joins, set ops (the next two lectures)

Example

<u>Unsorted input</u>

918	CA	90245	Kate
170	CA	90345	Jane
897	WI	53713	Will
275	WI	53705	Kate
563	CA	90245	Andy
990	CA	90001	Jane

SELECT * FROM T ORDER BY name ASC;

563	CA	90245	Andy
990	CA	90001	Jane
170	CA	90345	Jane
275	WI	53705	Kate
918	CA	90245	Kate
897	WI	53713	Will

SELECT DISTINCT name FROM T;

Andy
Jane
Jane
Kate
Kate
Will

→

Andy
Jane
Kate
Will

Algorithm:

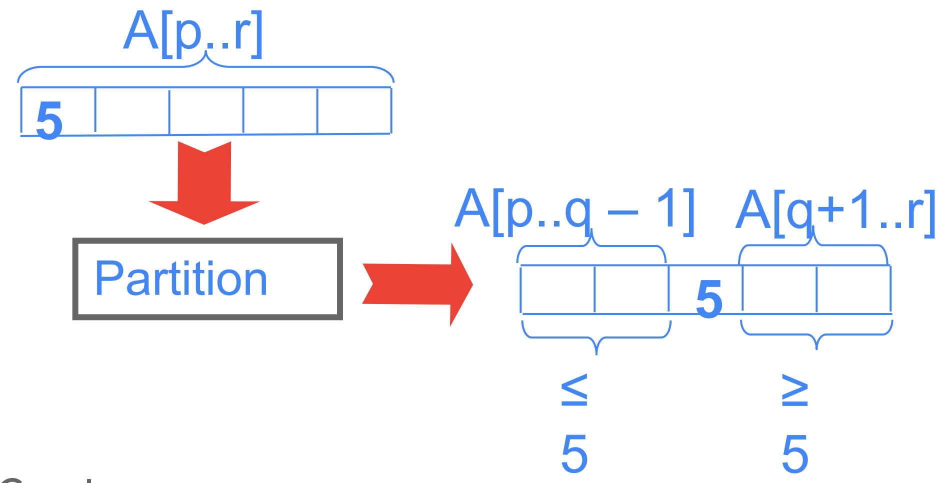
1. Sort input by name
2. For each row:
   Check if the next row has the same value, output if not

Sorting problem

- Given a set of N values, there can be N! permutations of these values.
- The sort output is *one* permutation among N! possibility.
- Each comparison essentially cuts the permutation space in half.
- Algorithms for in-memory sort
  - Quick sort
  - Priority queue
  - Tree of loser (see Donald Knuth, The Art of Computer programming, Volume 3)

QuickSort

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```



Example of partitioning

- choose pivot: **4** **3 6 9 2 4 3 1 2 1 8 9 3 5 6**

Example of partitioning

- choose pivot: <u>**4**</u> **3 6 9 2 4 3 1 2 1 8 9 3 5 6**
- search: <u>**4**</u> **3 6 9 2 4 3 1 2 1 8 9 3 5 6**

i **j**

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

  i j

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6

  i j

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6

Example of partitioning

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6

  i j
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6

  i j
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6

  j i (done)

Sort algorithms

Quick sort: the most commonly used (std::sort)

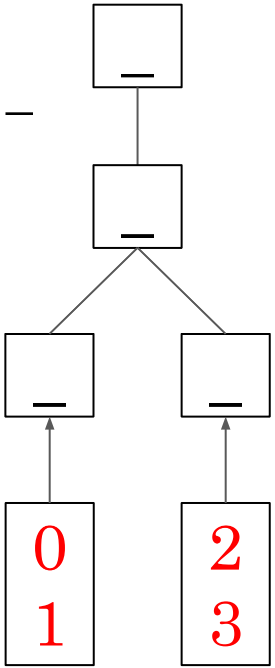
Sort with tree-of-loser priority queues
(by far the most efficient in my experience)

Notes: this is not std::priority_queue
typically used in heap-sort.

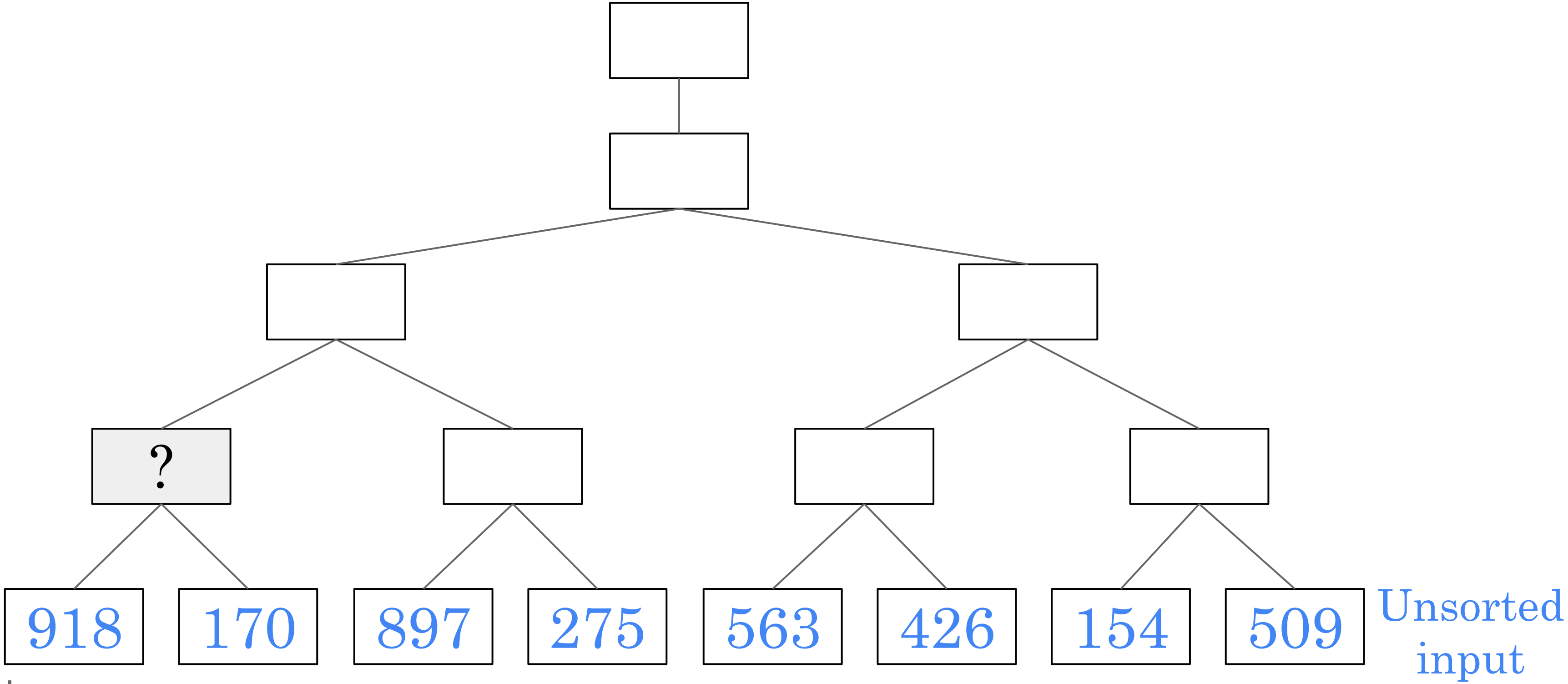
Only leaf-to-root passes –
no root-to-leaf passes

2 candidates per node
(except 1 in root)

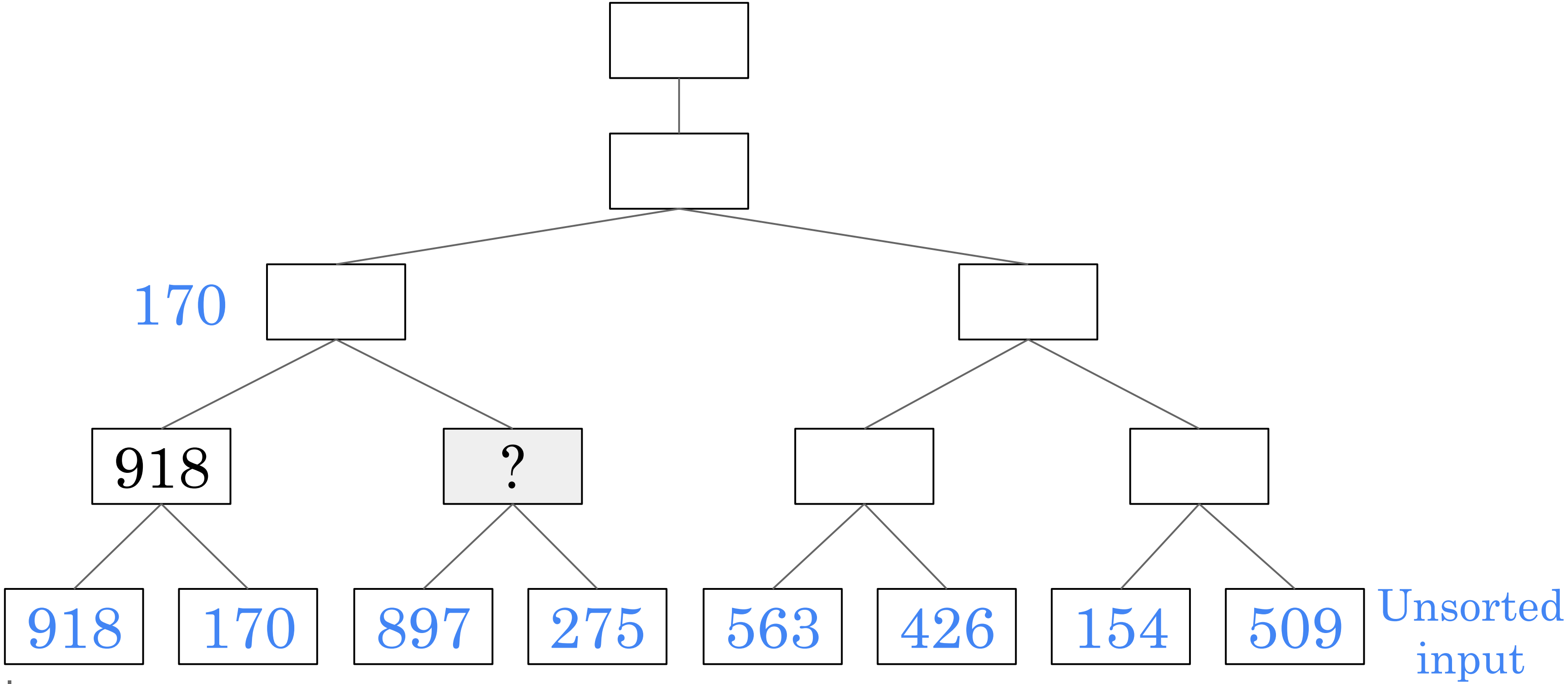
When competing:
winner moves up
loser stays



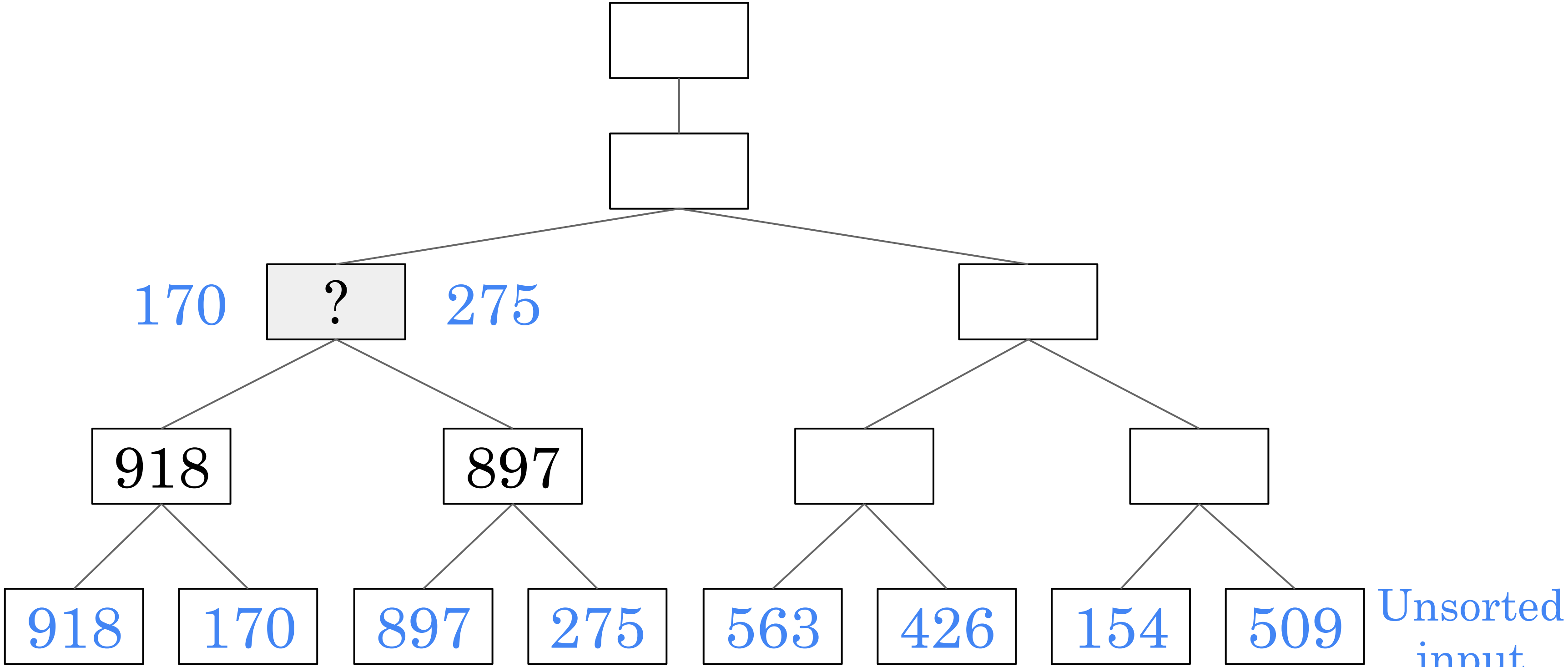
Sorting with tree-of-losers priority queue (Knuth's example)



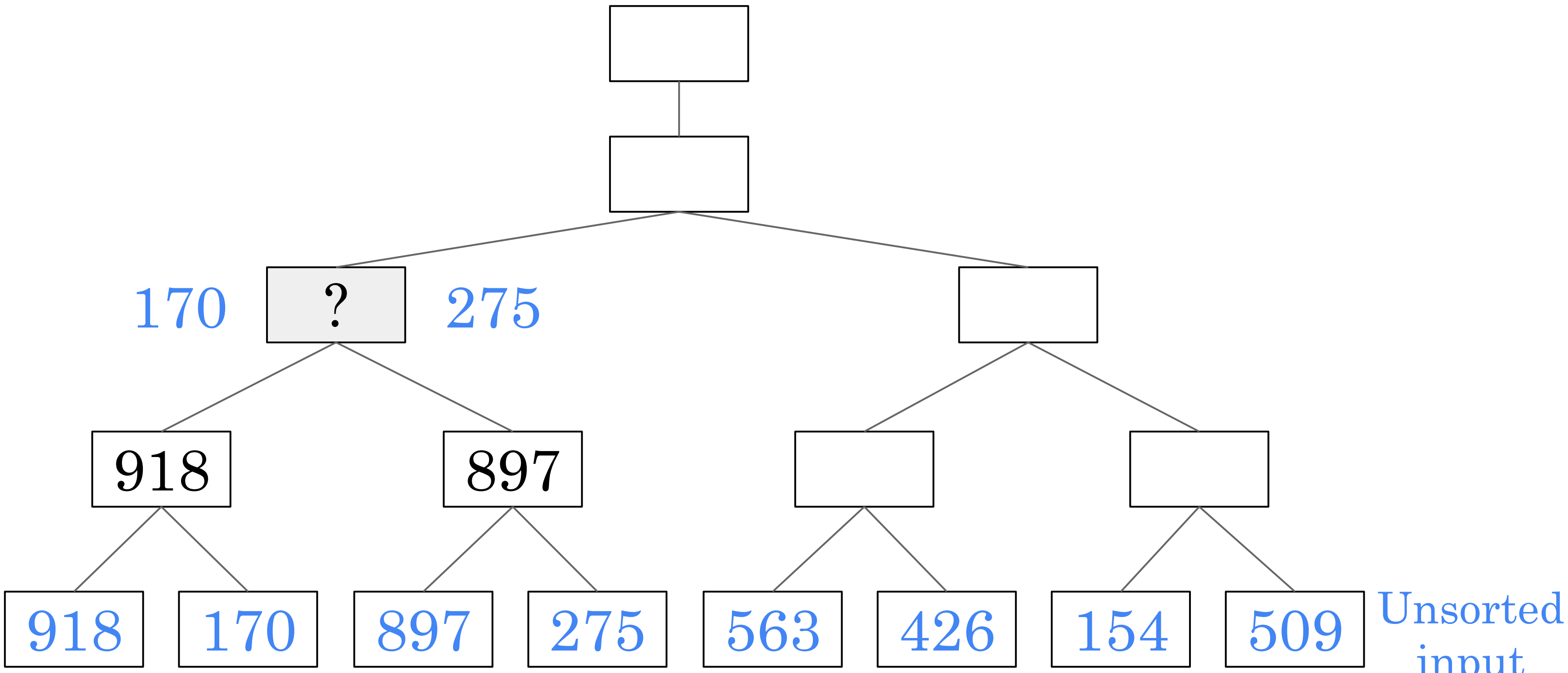
Sorting with tree-of-losers priority queue (Knuth's example)



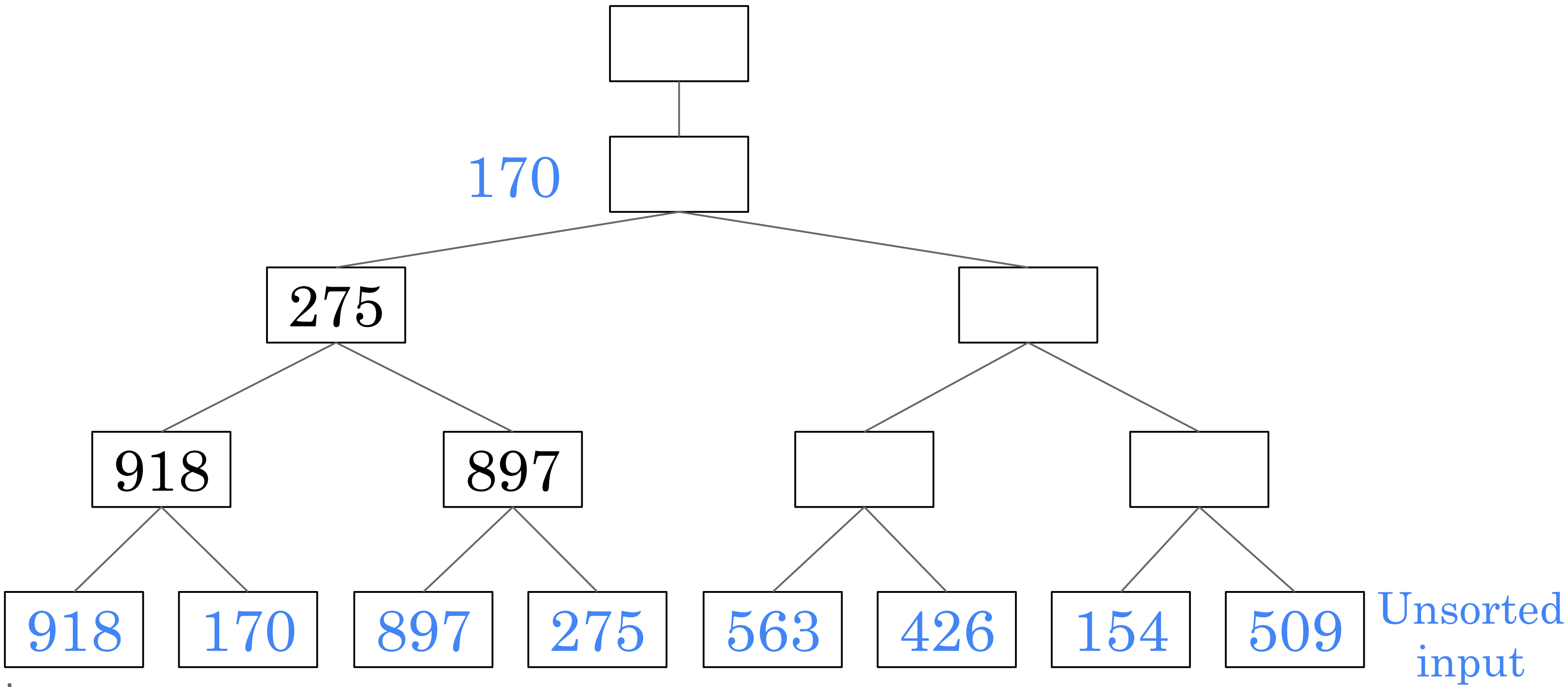
Sorting with tree-of-losers priority queue (Knuth's example)



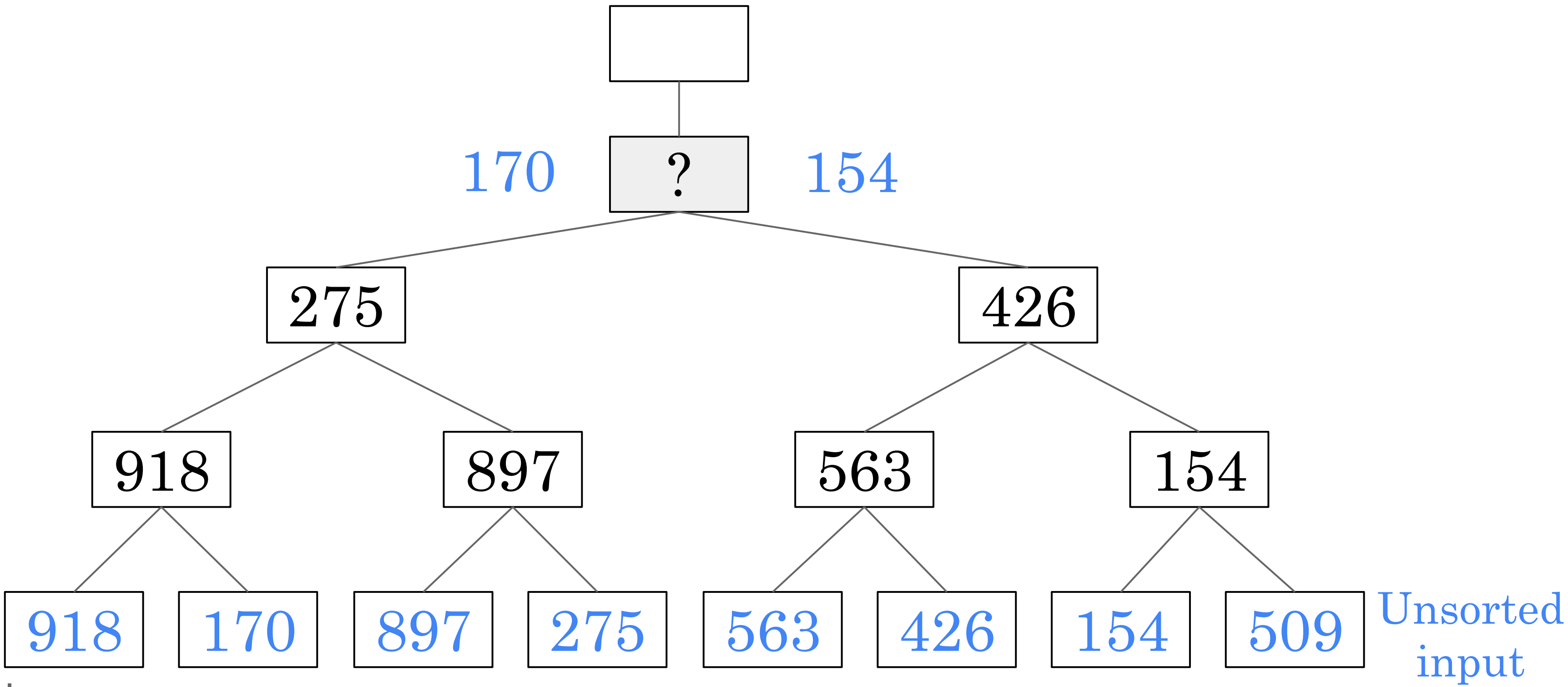
Sorting with tree-of-losers priority queue (Knuth's example)



Sorting with tree-of-losers priority queue (Knuth's example)

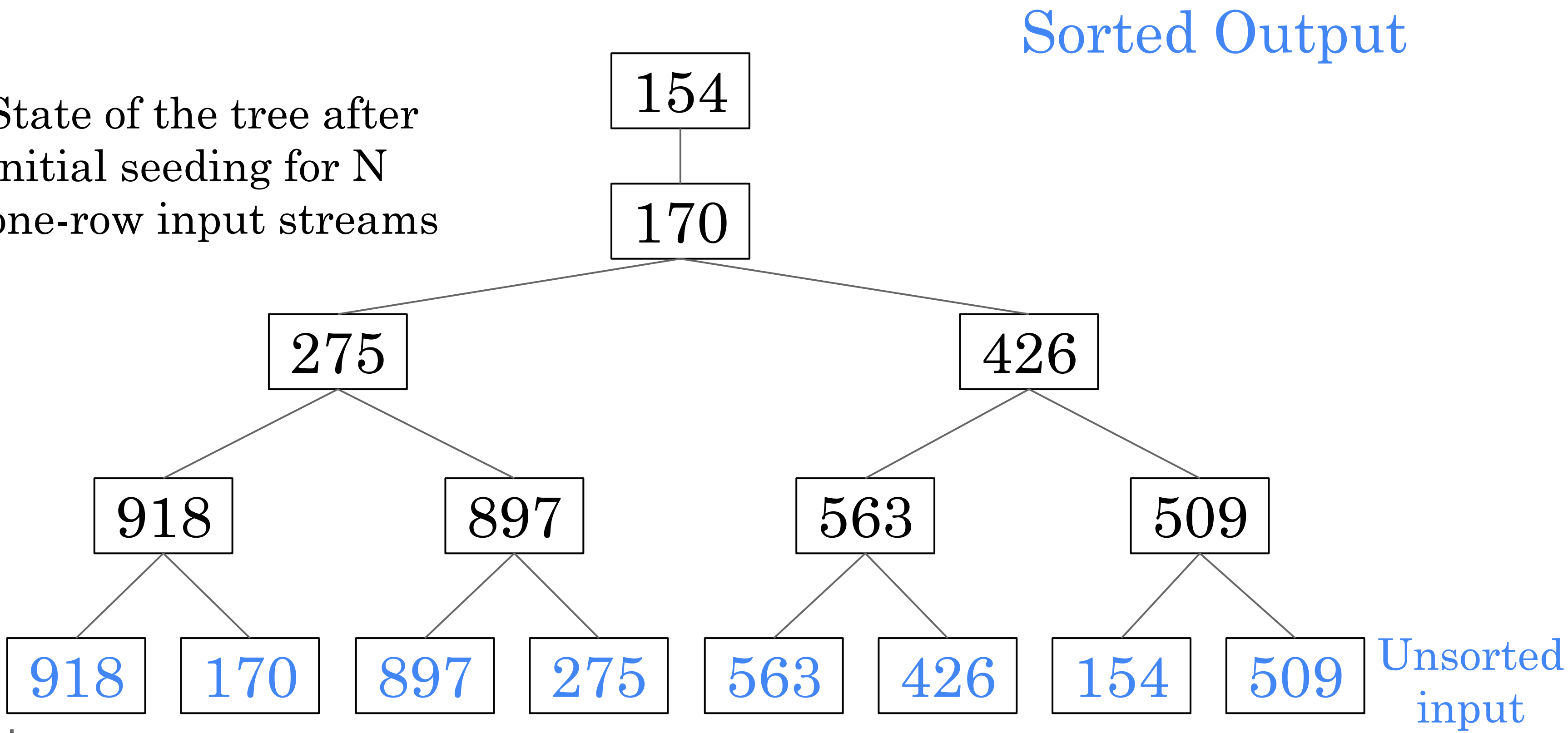


Sorting with tree-of-losers priority queue (Knuth's example)



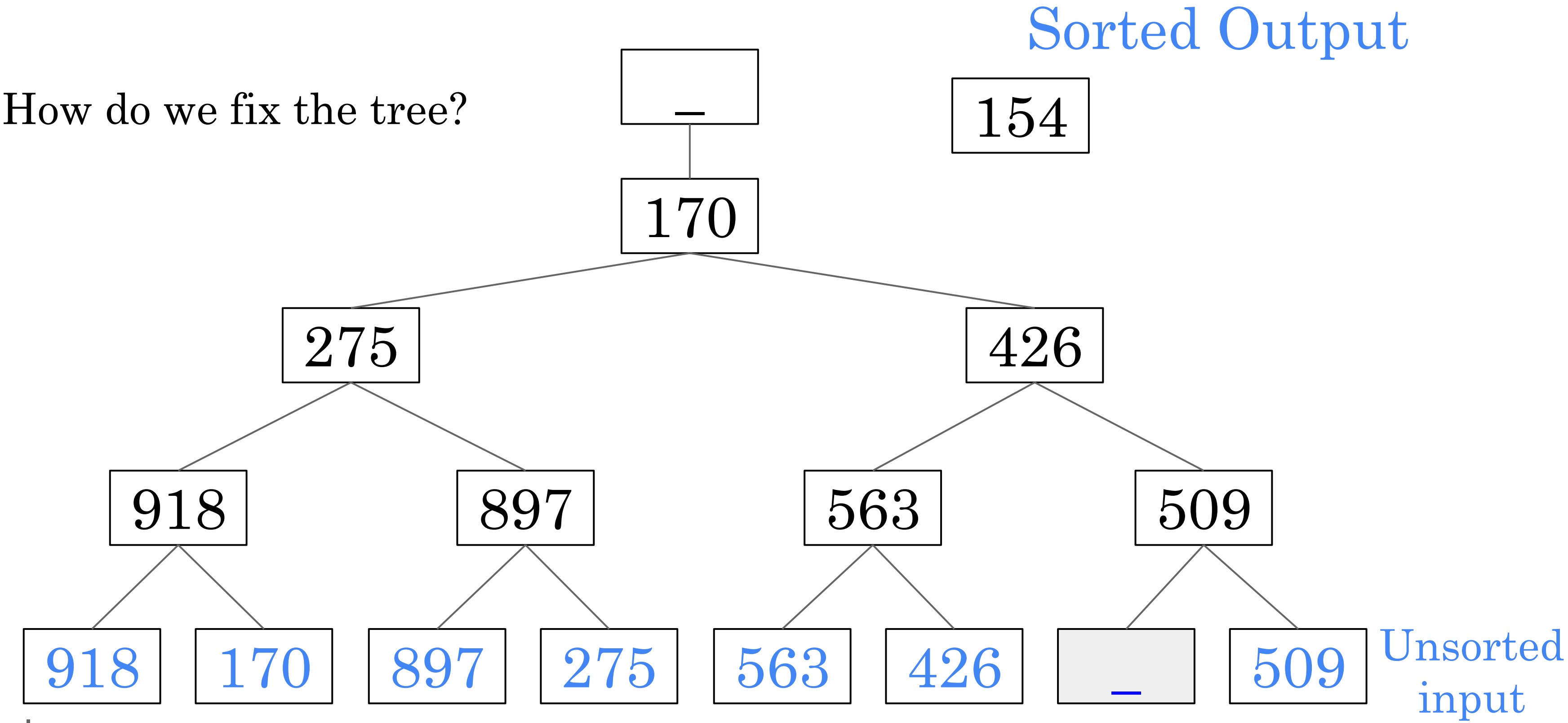
Sorting with tree-of-losers priority queue (Knuth's example)

State of the tree after initial seeding for N one-row input streams



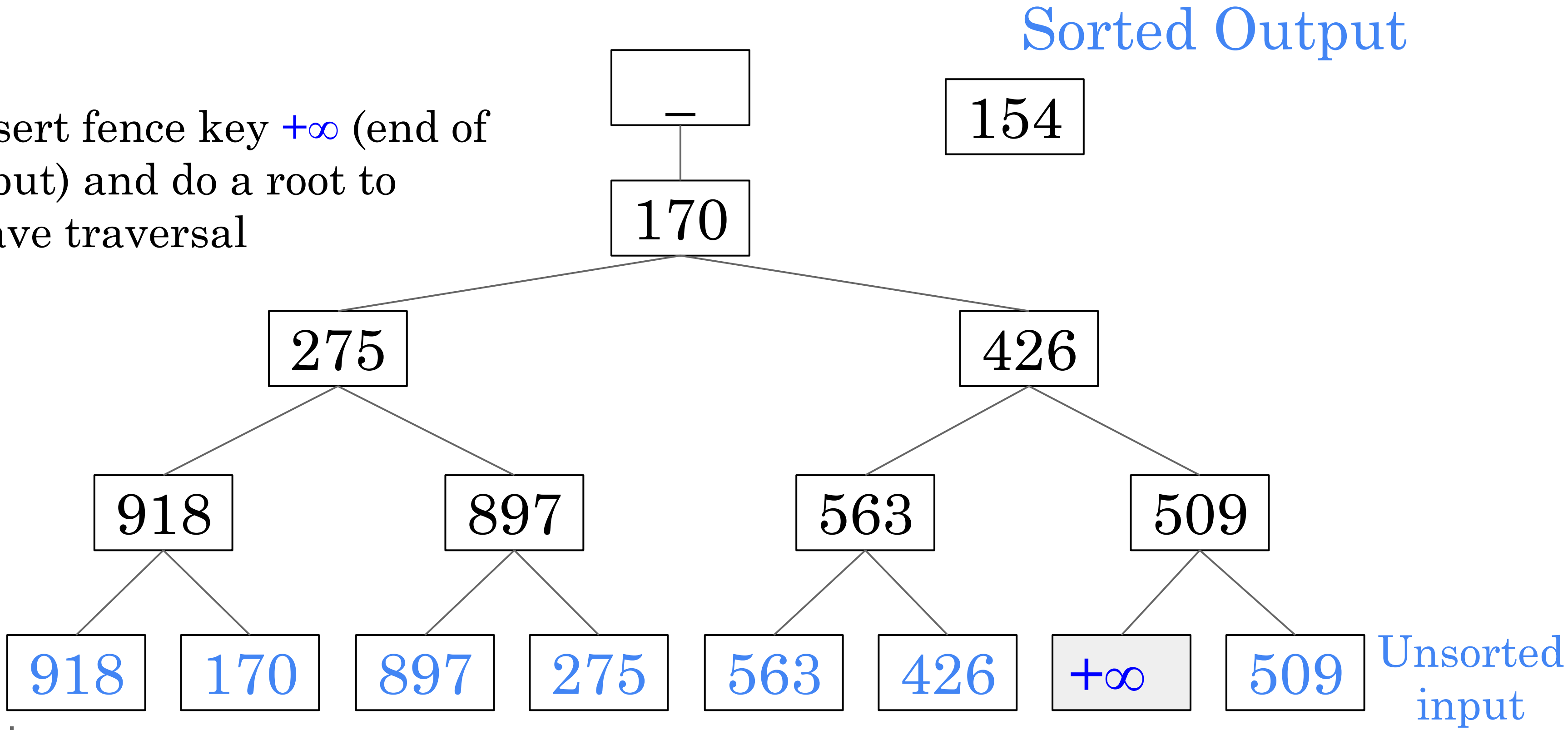
Sorting with tree-of-losers priority queue (Knuth's example)

How do we fix the tree?



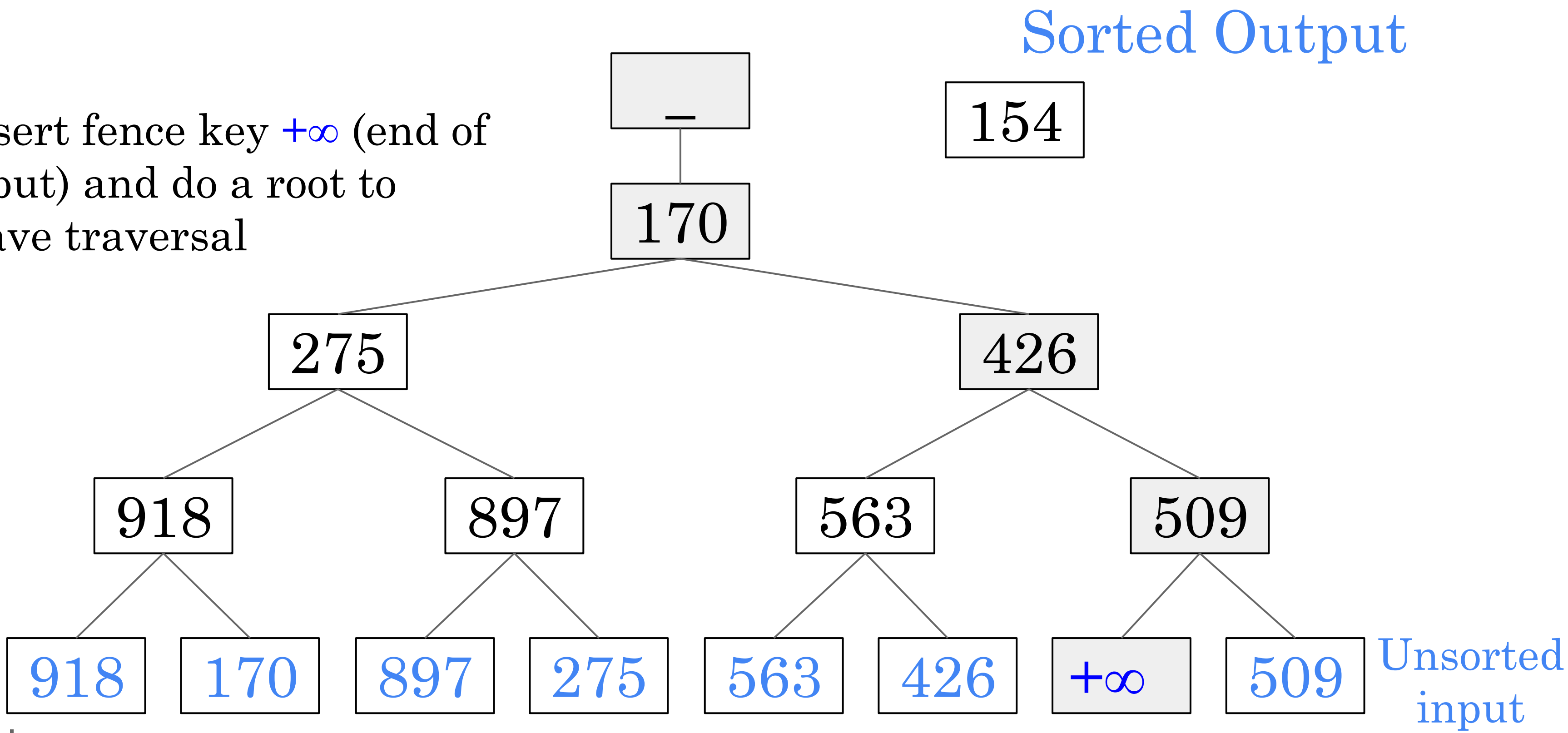
Sorting with tree-of-losers priority queue (Knuth's example)

Insert fence key +∞ (end of input) and do a root to leave traversal



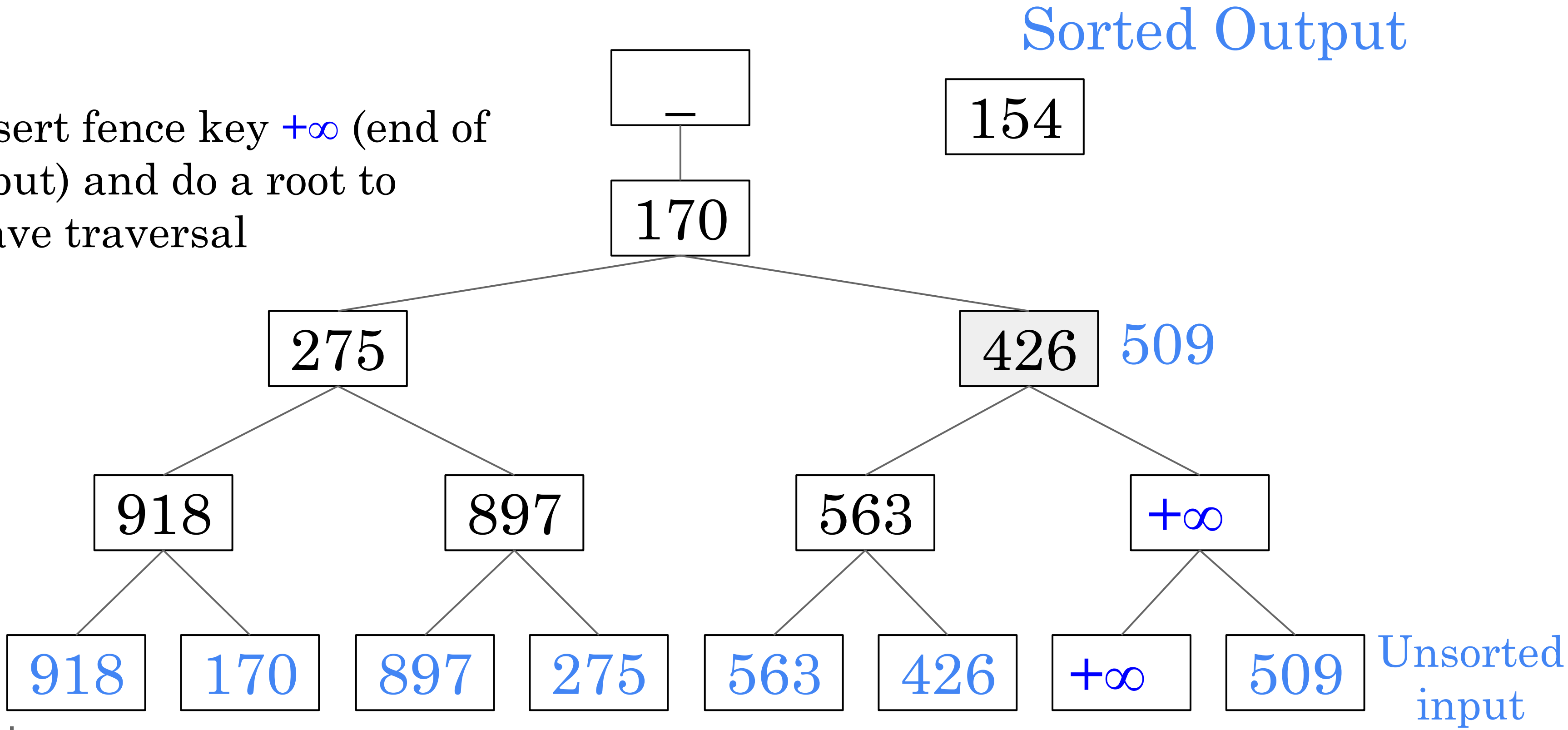
Sorting with tree-of-losers priority queue (Knuth's example)

Insert fence key +∞ (end of input) and do a root to leave traversal



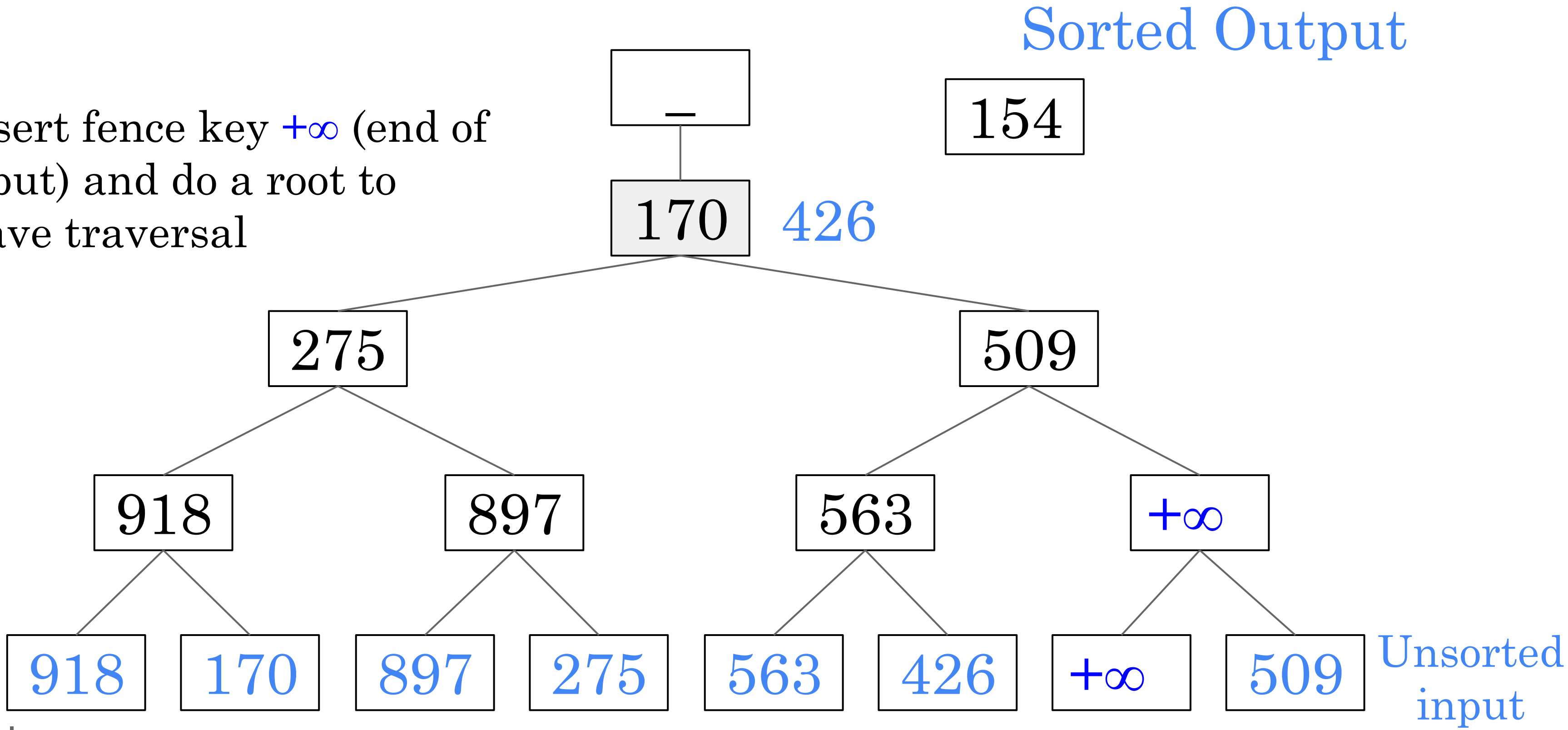
Sorting with tree-of-losers priority queue (Knuth's example)

Insert fence key $+\infty$ (end of input) and do a root to leave traversal



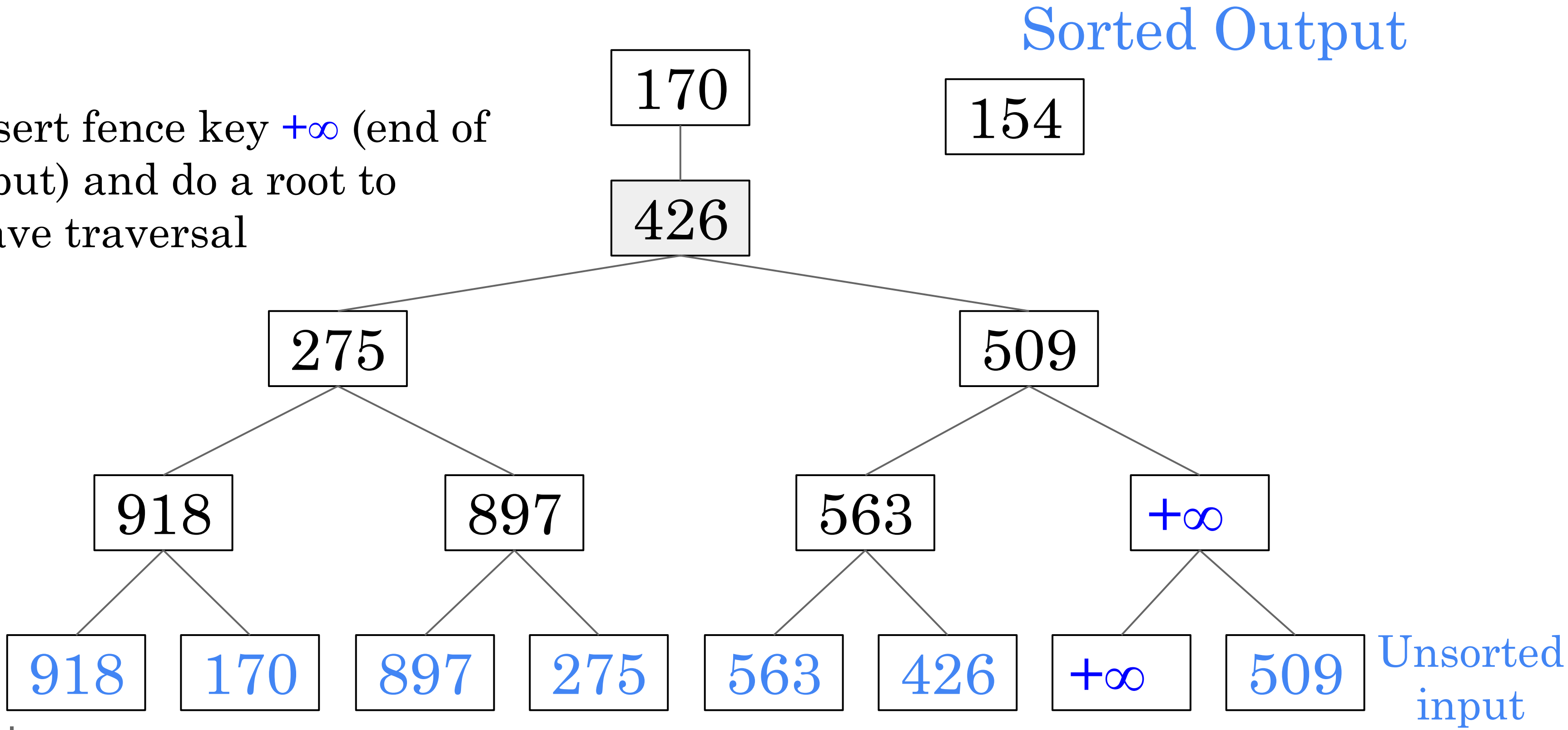
Sorting with tree-of-losers priority queue (Knuth's example)

Insert fence key +∞ (end of input) and do a root to leave traversal

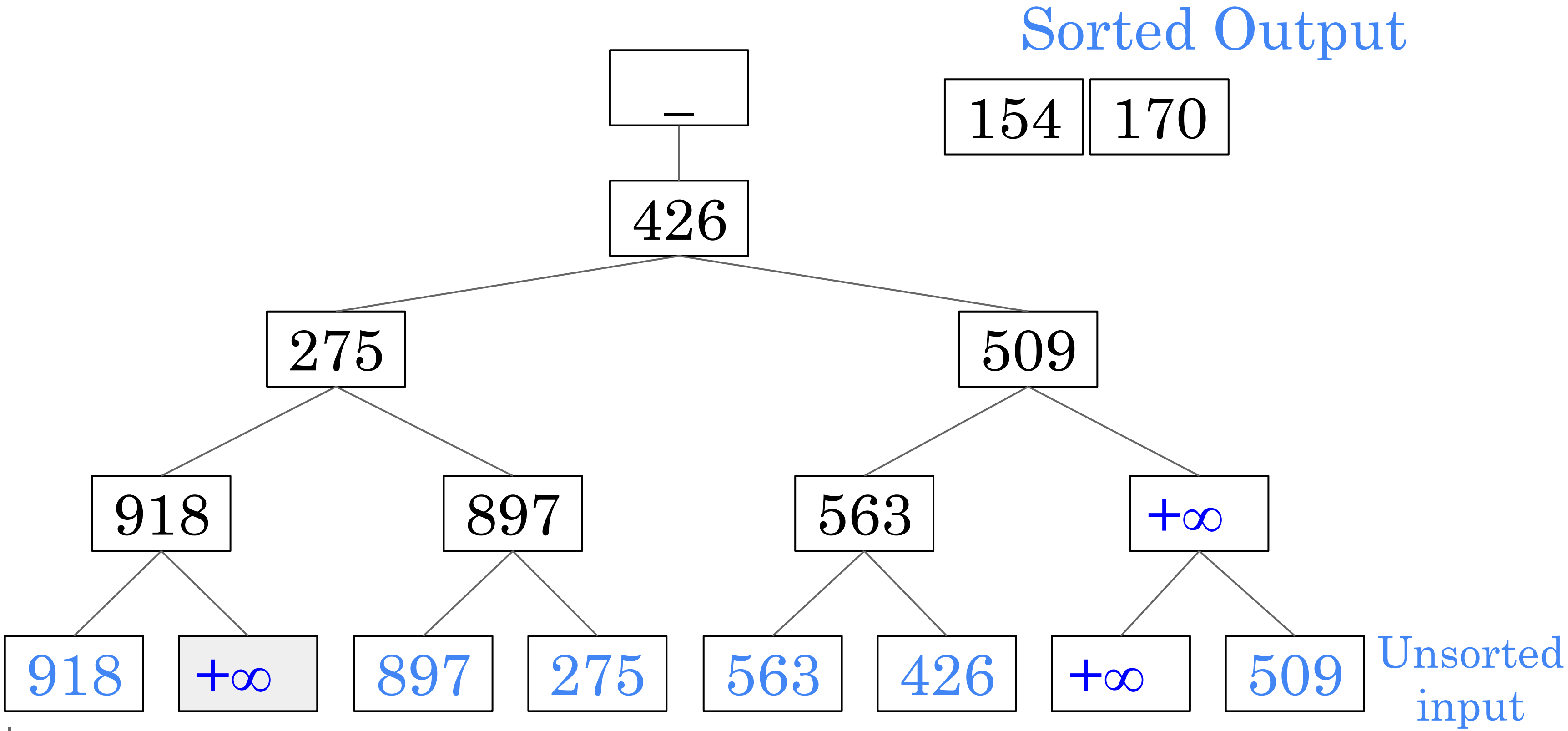


Sorting with tree-of-losers priority queue (Knuth's example)

Insert fence key $+\infty$ (end of input) and do a root to leave traversal



Sorting with tree-of-losers priority queue (Knuth's example)



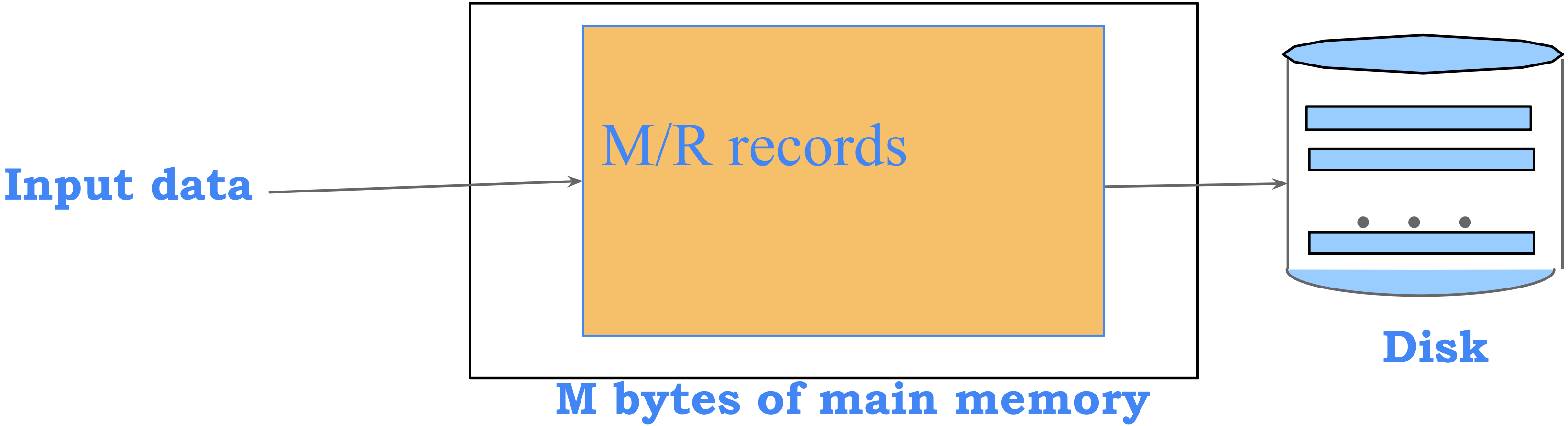
Run generation: comparison counts

Row count	QuickSort	Loser Tree	Lower bound	real/theory
1,000	11,696	8,722	8,525.8	1.023014
10,000	160,859	120,949	118,477.1	1.020864
100,000	2,020,269	1,542,713	1,516,964.0	1.016974
1,000,000	24,133,548	18,687,584	18,491,568.6	1.010600

Run Generation with tree-of-losers priority results into #data comparisons much closer to lower bound theory

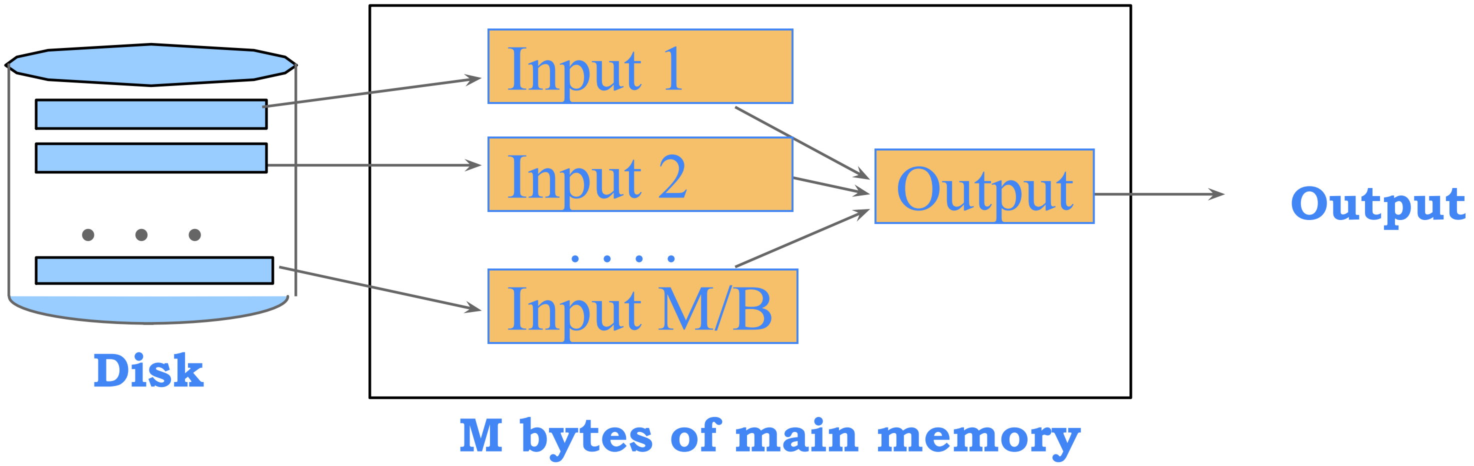
External Merge-Sort

- Phase one: read-sort-write cycle load M bytes in memory, sort, write to disk
    - Result: run size is as large as memory for quick sort (can be 2M for replacement selection)

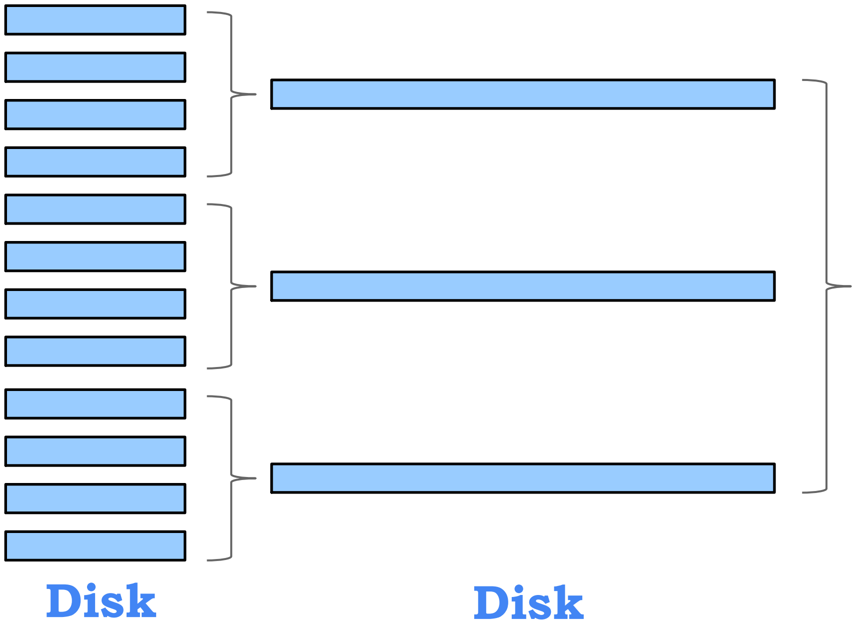


One-step merge
If everything can be merged in one pass

- Merge all the runs and returns the merged output.
- Only eligible when M is sufficient to hold all input buffers at once.

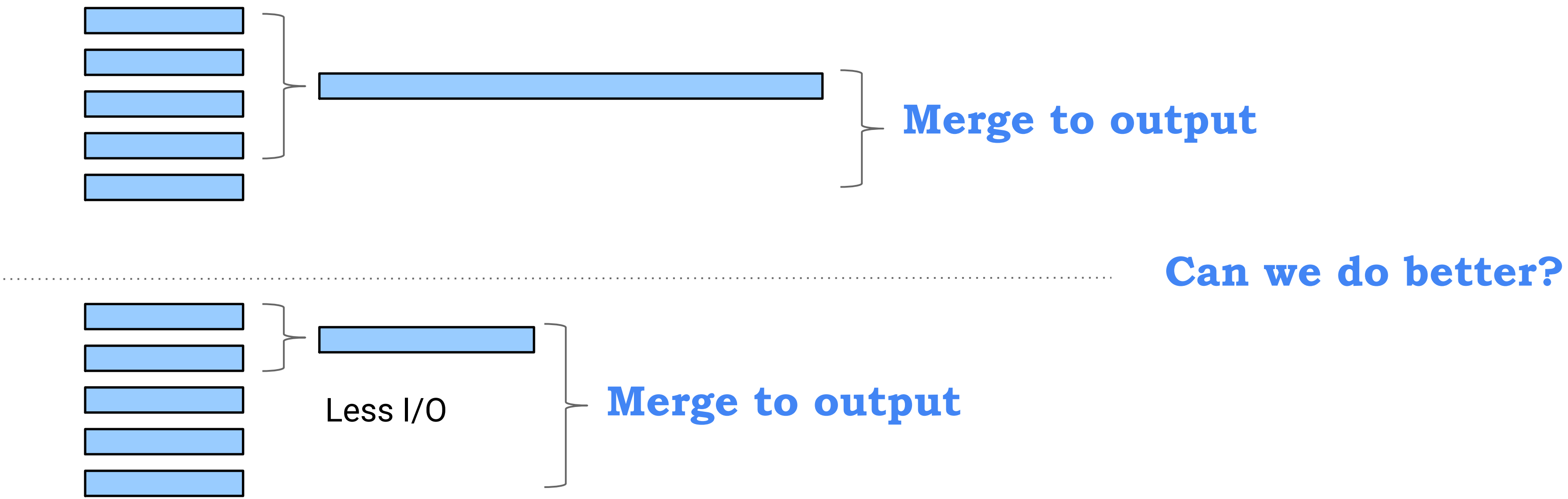


Multi-step merge: merge fan-in is 4

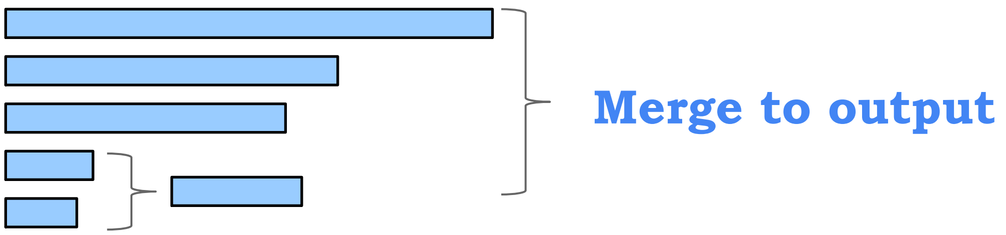


No need to write to disk, merge directly to output

Merge strategy, assuming merge fan-in is 4



Merge strategy, assuming merge fan-in is 4



Merge small runs first to minimize number of merge steps (and I/O)

Graceful degradation in external merge sort

Example:
Input size: 1,010 records
Memory size: 1,000 records
Q: How much many records to be written to disk for sorting?

Graceful degradation in external merge sort

Example:
Input size: 1,010 records
Memory size: 1,000 records
Q: How much many records to be written to disk for sorting?
A: Typical answer 1,010 (i.e, we spill the entire input)

Performance cliff problem:
Operation is fast when input fits in memory.
When it barely fits, the entire input is spilled, causing drastic change in performance

Graceful degradation in external merge sort

Example:
Input size: 1,010 records
Memory size: 1,000 records
Q: How much many records to be written to disk for sorting?
A: Typical answer 1,010 (i.e, we spill the entire input)

Performance cliff problem:
Operation is fast when input fits in memory.
When it barely fits, the entire input is spilled, causing drastic change in performance.

Solution: Spill as to disk as much as needed. Optimal strategy: spill only 10 records

Graceful degradation in external merge sort

Example:
Input size: 1,010 records
Memory size: 1,000 records
Q: How much many records we have to spill (write) to disk for sorting?
A: Typical answer 1,010 (i.e, we spill the entire cords)

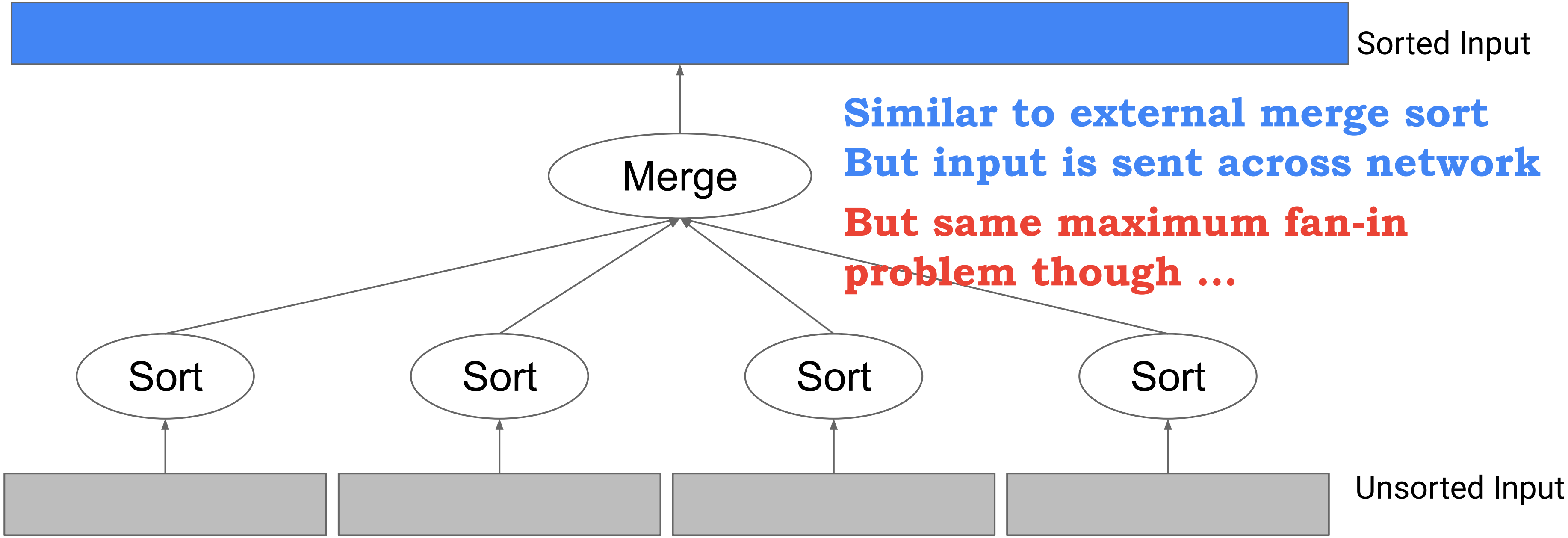
Performance cliff problem: operation is fast when input fits in memory. When it barely fits, the entire input is spilled.

Solution: Spill as to disk as much as needed.
Optimal spilling strategy: spill only 10 records

Distributed sort

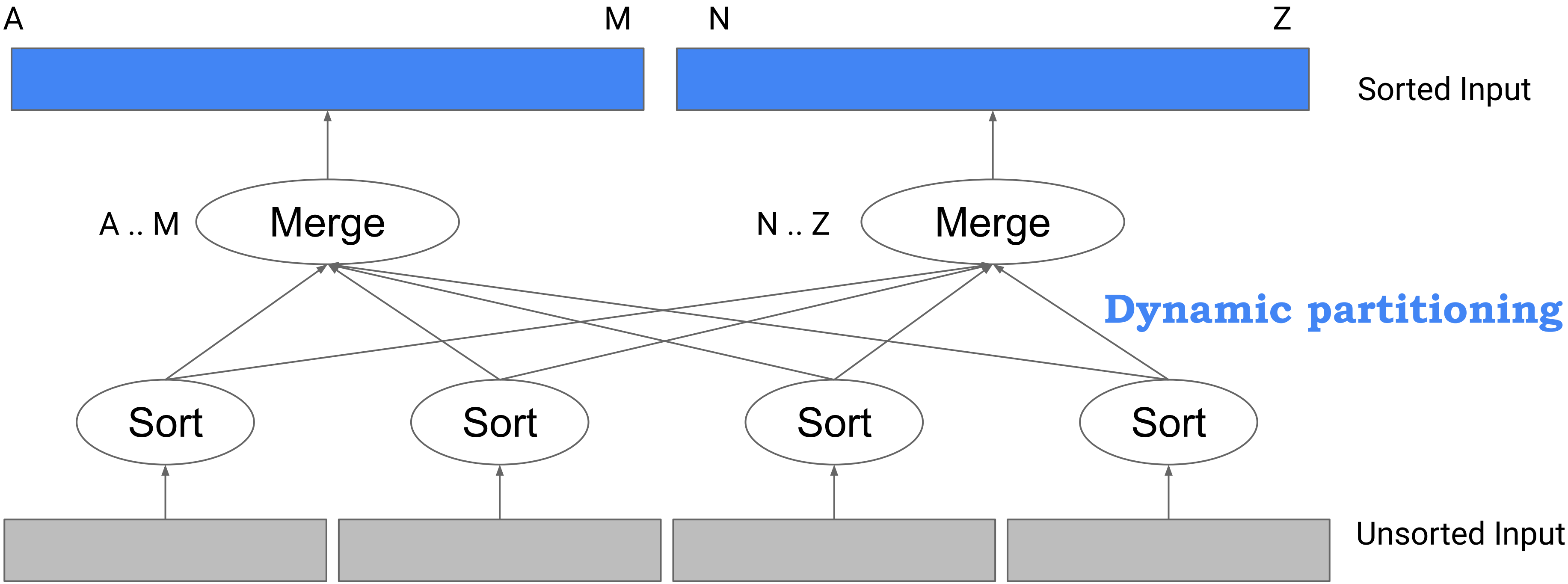
Problem: how to sort **a very large amount of data** that cannot fit in one machine?

Shuffle data during sort: many-to-one exchange

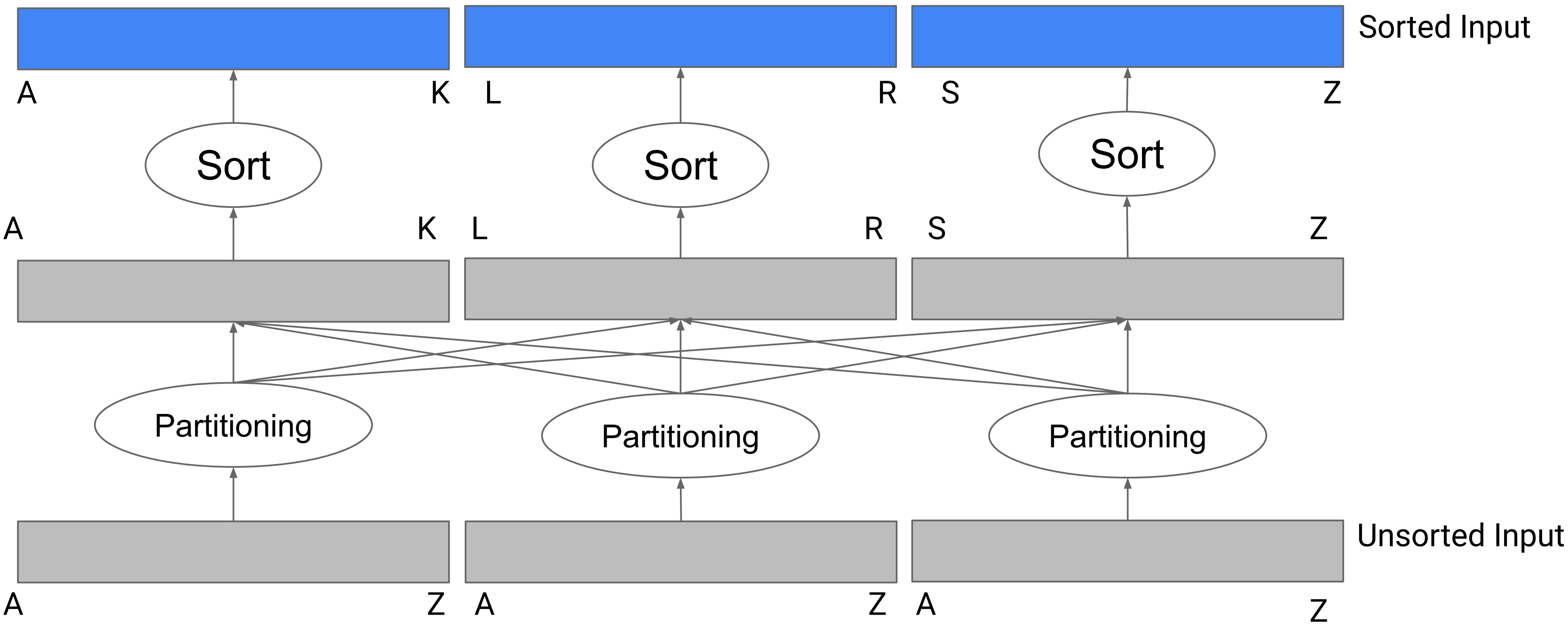


Similar to external merge sort
But input is sent across network
But same maximum fan-in problem though …

Shuffle data during sort: many-to-many exchange



Shuffle data before sort



Other topics

Double buffering in external merge sort (see the Cow book)

Normalized keys, offset-value code (see Goetz's computing survey paper on sorting)

Distributed sort in real world (MapReduce, Presto, Hadoop, …)

Q & A